Technical Report TR06-025

Department of Computer Science
Univ. of North Carolina at Chapel Hill

SPQR: Formal Foundations and Practical Support for the Automated Detection of Design Patterns from Source Code

Jason McC. Smith

Department of Computer Science
University of North Carolina
Chapel Hill, NC 27599-3175

smithja@cs.unc.edu

Dec 12, 2005

SPQR: Formal Foundations and Practical Support for the Automated Detection of Design Patterns From Source Code

Jason McColm Smith

A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill
2005

Approved by:

P. David Stotts, Professor

Siddhartha Chatterjee, Reader

David Plaisted, Reader

Jan Prins, Reader

Albert H. Segars, Reader

ABSTRACT
JASON MCCOLM SMITH: SPQR: Formal Foundations and Practical Support for the Automated Detection of Design Patterns From Source Code.
(Under the direction of P. David Stotts.)

Maintenance costs currently comprise the majority of total costs in producing software. While object-oriented techniques and languages appear to have assisted in the production of new code, there is little evidence to support the theory that they have helped lower the high cost of maintenance. In this dissertation, I describe the current problem and provide a system ultimately aimed at reducing this cost. The System for Pattern Query and Recognition, or SPQR, consists of: the rho-calculus, a formal foundation for conceptual relationships in object-oriented systems; a suite of Elemental Design Patterns that capture the fundamentals of object-oriented programming and their expressions in the rho-calculus; an XML Schema, the Pattern/Object Markup Language, or POML, to provide a concrete method for expressing the formalisms in a practical manner; an example mapping from the C++ programming language to POML; an implementation which ties the above components together into a practical tool that detects instances of design patterns directly from source code using the Otter automated theorem prover. I will discuss each of the components of the system in turn, and relate them to previous research in the area, as well as provide a number of future research directions.

Using the results of SPQR, a system can be more easily documented and understood. The major contribution of SPQR is the flexible detection of design patterns using the formalisms of rho-calculus instead of static structural cues. Building on SPQR, I propose: a suite of metrics utilizing the Minimum Description Length principle that capture the salient conceptual features of source code as expressed in design patterns nomenclature as a method for measuring comprehensibility of code; an approach for mapping these metrics to cost-based metrics from current management principles. This combination should prove to be effective in facilitating communication between technical and managerial concerns in a manner that allows for the most efficient allocation of resources during maintenance of software systems.

ACKNOWLEDGMENTS

My deepest thanks to all those friends and family who have provided me with support, and patience...

...my advisor and friend, Dr. David Stotts. Here's to many more years of fruitful collaboration.

...my readers, editors, and all who have contributed to the discussions that proved to be so important in this endeavor.

...Dr. Marjorie Olmstead. Sometimes a simple compliment really does make all the difference in a life.

...Leandra Vicci. Your support was instrumental in my being here.

...but most of all to my wife, Leah, who has stood by me in each of the above ways, and so many more.

CONTENTS

LIST OF FIGURES **xv**

LIST OF TABLES **xix**

1 Introduction **1**

1.1 Introduction 1

1.2 Maintenance 1

1.3 Difficulties of Maintenance 2

1.3.1 Technical 2

1.3.2 Managerial 3

1.3.3 Psychological Limitations 4

1.4 Object-Oriented Programming 6

1.5 Patterns 9

1.5.1 Formalization of Patterns 11

1.5.2 Documentation Divide 13

1.6 Metrics 15

1.6.1 Syntactic metrics 15

1.7 Concepts Programming 16

1.7.1 From code to comprehension 17

1.7.2 Pattern Compilers 20

1.8 Conceptual metrics 20

1.8.1 Design metrics 20

1.8.2 Malignant pattern detection 21

1.8.3 Multi-scale analysis 22

1.8.4 Relationships 23

1.9 SPQR . . . 23
1.10 Advantages . . . 24
1.10.1 Technical . . . 24
1.10.2 Managerial . . . 24
1.10.3 Human limitations . . . 25
1.11 Unresolved issues . . . 25
1.11.1 No silver bullet . . . 25
1.11.2 False positives . . . 26
1.11.3 Maintenance . . . 26
1.12 Summary . . . 26

2 Related Work **28**
2.1 Decomposition of patterns . . . 28
2.1.1 Refactoring Approaches . . . 28
2.1.2 Fragments . . . 29
2.1.3 Minipatterns . . . 29
2.1.4 Structural Analyses . . . 29
2.1.5 Conceptual Relationships . . . 30
2.1.6 Other Work . . . 31
2.2 Analysis techniques . . . 32
2.2.1 Cohesion and coupling analysis . . . 32
2.3 Quantitative metrics . . . 33
2.4 Formalization of Patterns . . . 34
2.4.1 LePUS . . . 34
2.4.2 Other approaches . . . 36
2.5 Pattern Detection and Validation . . . 37
2.5.1 Pattern Enforcing Compiler . . . 37
2.5.2 FUJABA . . . 38
2.5.3 RML and CrocoPat . . . 40

3 Sigma Calculus **41**
3.1 Basics of ς-calculus . . . 42
3.1.1 Objects, method, fields and types . . . 43

3.1.2 Selection and update . . . 44
3.1.3 Reduction rules . . . 44
3.1.4 Classes and other constructs . . . 45
3.2 Inflexibility of ς-calculus . . . 45

4 Rho Calculus **47**
4.1 Notation Conventions . . . 48
4.2 Direct Definitions . . . 50
4.2.1 Method-body-based reliance operators: $<_\mu$, $<_\phi$ . . . 50
4.2.2 Update-target-based reliance operators: $<_\sigma$, $<_\kappa$ . . . 52
4.3 Inferred Definitions (Transitivity) . . . 55
4.3.1 Direct contextual inferences . . . 56
4.3.2 Chained inferences . . . 57
4.3.3 Inference examples . . . 59
4.4 Selection Partitioning . . . 61
4.5 Similarity Principle . . . 62
4.6 Consistency . . . 66
4.7 Objects vs. Types . . . 66
4.7.1 Left-Hand Side Type Replacement . . . 67
4.7.2 Right-Hand Side Type Replacement . . . 68
4.7.3 Ramifications . . . 68
4.8 Relation to other work . . . 70

5 Elemental Design Patterns **71**
5.1 Elemental Design Patterns . . . 71
5.2 Types . . . 73
5.2.1 Inheritance . . . 73
5.2.2 AbstractInterface . . . 73
5.3 Objects . . . 74
5.3.1 CreateObject . . . 74
5.3.2 Retrieve . . . 75
5.4 Methods and Objects - Leftdot similarity . . . 76
5.4.1 Conglomeration . . . 76

5.4.2 Recursion . . . 77
5.5 Methods and Objects - Leftdot dissimilarity . . . 77
5.5.1 Delegate . . . 77
5.5.2 Redirect . . . 78
5.6 Methods and Types . . . 79
5.6.1 DelegatedConglomeration . . . 79
5.6.2 RedirectedRecursion . . . 80
5.6.3 DelegateInFamily . . . 80
5.6.4 RedirectInFamily . . . 81
5.6.5 DelegateInLimitedFamily . . . 81
5.6.6 RedirectInLimitedFamily . . . 82
5.7 Methods and Supertypes . . . 82
5.7.1 ExtendMethod . . . 83
5.7.2 RevertMethod . . . 83
5.8 Examination of design patterns . . . 84
5.8.1 Method calls . . . 85
5.8.2 Relationships . . . 88
5.8.3 Method call EDPs . . . 91
5.8.4 Object and Type EDPs . . . 94
5.9 Isotopes . . . 94
5.10 Future Definitions . . . 98
5.10.1 Recurrence . . . 98
5.10.2 Iteration . . . 99
5.10.3 Collection . . . 99
5.10.4 Invariance . . . 99
5.11 Conclusion . . . 100

6 Pattern Composition **101**

6.1 FulfillMethod . . . 101
6.2 Objectifier . . . 102
6.3 Object Recursion . . . 102
6.4 Decorator . . . 103

6.5 Other Intermediates . . . 106
6.5.1 RetrieveNew . . . 107
6.5.2 RetrieveShared . . . 107
6.6 GOF Patterns . . . 107
6.6.1 Singleton . . . 108
6.7 Pattern Hierarchies . . . 108

7 Pattern/Object Markup Language 111

7.1 Basic ρ-calculus Support . . . 112
7.1.1 Objects, methods, fields, types . . . 112
7.1.2 Classes . . . 117
7.2 Patterns . . . 120
7.3 Additional ρ-calculus Concepts . . . 121
7.4 SPQR Support . . . 121
7.5 Unsupported Constructs . . . 123

8 Converting C++ to POML 124

8.1 Classes . . . 125
8.2 Templates . . . 126
8.3 Namespaces . . . 128
8.4 Methods . . . 130
8.4.1 Operators . . . 130
8.4.2 Overloading . . . 130
8.4.3 Parameters and Return Values . . . 131
8.5 Expressions . . . 131
8.6 Statements . . . 134
8.7 Linkages . . . 134
8.8 EDPs . . . 137
8.9 Unsupported constructs . . . 138

9 SPQR Implementation 140

9.1 Feature detection and description . . . 142
9.1.1 `gcc` . . . 142

9.1.2 `gcctree2poml` . . . 143
9.2 Inference . . . 145
9.2.1 Otter . . . 145
9.2.2 Otter inputs . . . 146
9.2.3 Mapping ρ-calculus to Otter . . . 149
9.2.4 Environment setup and invocation . . . 157
9.2.5 `poml2otter` . . . 158
9.2.6 Running Otter . . . 159
9.3 Query reporting . . . 159
9.3.1 `proof2poml` . . . 159
9.4 Support scripts . . . 160
9.5 Training . . . 163
9.6 Exploration vs. Validation . . . 164
9.7 Extension of SPQR . . . 165

10 SPQR Validation **167**

10.1 Testing . . . 167
10.1.1 Training - Gang of Four . . . 167
10.1.2 Isotopes - KillerWidget . . . 173
10.1.3 Validation - NotificationCenter . . . 179
10.1.4 Exploration - `gcc std` . . . 180
10.2 Performance . . . 182
10.2.1 `gcctree2poml` . . . 182
10.2.2 Otter . . . 186
10.2.3 Support Tools . . . 188
10.2.4 Performance bounding . . . 189
10.3 Summary . . . 194

11 Maintenance Metrics **195**

11.1 Software Project Risk Analysis . . . 195
11.2 Maintainability Metrics . . . 199
11.2.1 Basic metrics of pattern coverage . . . 201
11.2.2 Comprehensibility metrics . . . 205

11.2.3 Metrics for management . . . 208

12 Future Research **210**

12.1 EDPs as language design hints . . . 210

12.1.1 Delegation . . . 210

12.1.2 ExtendMethod . . . 211

12.1.3 Retrieve, RetrieveShared . . . 211

12.1.4 Collection . . . 211

12.1.5 AbstractInterface . . . 211

12.2 Educational uses of EDPs . . . 212

12.3 Malignant patterns . . . 212

12.3.1 'Bad smells' as refactoring opportunity discovery . . . 213

12.3.2 Semi-automated support for the refactoring process . . . 213

12.4 Aspect Oriented Programming . . . 214

12.5 Architecture 'Optimization' . . . 214

12.6 SPQR Refinements - Tools . . . 215

12.6.1 Level of Detail control . . . 215

12.6.2 Improved similarity extraction . . . 216

12.6.3 Alternate Automated Theorem Provers . . . 216

12.6.4 Alternate environments . . . 217

12.6.5 Alternate data sources . . . 217

12.6.6 Distributed analysis . . . 218

12.6.7 Expanding pattern libraries . . . 218

12.7 SPQR Refinements - Theory . . . 219

12.7.1 Temporal Logic . . . 219

12.7.2 Model Checkers . . . 219

12.7.3 ς-calculus interpretation . . . 220

12.8 Comprehension of code . . . 220

12.8.1 Experimental support . . . 221

12.8.2 Training and communication . . . 221

12.9 Analysis of procedural code . . . 221

12.10 Pattern Mining . . . 223

12.11 Genetics . . . 224

A Δ_ρ Fragment **225**

B EDP Catalog **228**

B.1 Object Elements . . . 230

B.1.1 CreateObject . . . 230

B.1.2 Retrieve . . . 235

B.2 Type Relations . . . 238

B.2.1 Inheritance . . . 238

B.2.2 AbstractInterface . . . 241

B.3 Method Invocation . . . 244

B.3.1 Delegate . . . 244

B.3.2 Redirect . . . 247

B.3.3 Conglomeration . . . 250

B.3.4 Recursion . . . 252

B.3.5 RevertMethod . . . 257

B.3.6 ExtendMethod . . . 261

B.3.7 DelegatedConglomeration . . . 264

B.3.8 RedirectedRecursion . . . 267

B.3.9 DelegateInFamily . . . 270

B.3.10 RedirectInFamily . . . 274

B.3.11 DelegateInLimitedFamily . . . 278

B.3.12 RedirectInLimitedFamily . . . 281

C Intermediate Patterns Compositions **284**

C.1 FulfillMethod . . . 285

C.2 RetrieveNew . . . 287

C.3 RetrieveShared . . . 290

C.4 Objectifier . . . 293

C.5 ObjectRecursion . . . 294

D GOF Patterns Compositions **295**

D.1 Creational Patterns . . . 296

D.1.1 Abstract Factory . . . 296
D.1.2 Builder . . . 297
D.1.3 Factory Method . . . 298
D.1.4 Prototype . . . 299
D.1.5 Singleton . . . 300
D.2 Structural Patterns . . . 301
D.2.1 Adapter . . . 301
D.2.2 Bridge . . . 303
D.2.3 Composite . . . 304
D.2.4 Decorator . . . 304
D.2.5 Facade . . . 305
D.2.6 Flyweight . . . 306
D.2.7 Proxy . . . 307
D.3 Behavioral Patterns . . . 308
D.3.1 Chain of Responsibility . . . 308
D.3.2 Command . . . 309
D.3.3 Interpreter . . . 309
D.3.4 Iterator . . . 310
D.3.5 Mediator . . . 311
D.3.6 Memento . . . 312
D.3.7 Observer . . . 313
D.3.8 State and Strategy . . . 314
D.3.9 Template Method . . . 316
D.3.10 Visitor . . . 316

E `gcc` Dump Tree Format **318**
E.1 The `gcc` dump tree format . . . 318
E.2 `gcc` created artifacts . . . 321

F POML Schema **329**

G Otter Inputs **337**
G.1 Headers . . . 337

G.1.1 Searching header . . . 337
G.1.2 Inference header . . . 337
G.1.3 Debugging header . . . 338
G.2 ς-calculus operators . . . 338
G.3 ς-calculus inferences . . . 339
G.4 ρ-calculus operators . . . 341
G.5 ρ-calculus inferences . . . 343
G.6 INH and SUB blocks . . . 352

H XSLTs **362**
H.1 POML2Otter.xsl . . . 362
H.2 POML2OtterFacts.xsl . . . 370
H.3 POML2OtterSearch.xsl . . . 371

Bibliography **373**

LIST OF FIGURES

1.1 The two halves of a constructive diagram, such as a pattern 10
1.2 Software as an analogue to pattern definition 11
1.3 Incomplete solution in software 14
1.4 Basic pattern structure 18
1.5 Path from code to comprehension 19
1.6 Completed path 22

2.1 Objectifier class structure 30
2.2 Object Recursion class structure 31

3.1 Singleton as expressed in ς-calculus 46

4.1 Pseudo-code for $<_\mu$ and $<_\phi$ example 51
4.2 Pseudo-code for object P 59
4.3 Polymorphic object 69

5.1 Example self-referential object and method call 76
5.2 Example generalized object and method call 77
5.3 Example code for convolving methods calls and typing relationships 79
5.4 DelegateInLimitedFamily in a class tree 82
5.5 Example super-referential object and method call 83
5.6 External call 88
5.7 Recursive call 88
5.8 Lexicographic similarity not indicative of intent 90
5.9 Initial List of method-call relationships from GoF patterns 92
5.10 RedirectInFamily class structure 95
5.11 RedirectInFamily Isotope 96

6.1 FulfillMethod intermediate pattern 102
6.2 Objectifier pattern with annotation 103
6.3 Object Recursion, annotated to show roles 104

6.4 Decorator class structure . . . 105
6.5 Decorator annotated to show EDP roles . . . 106
6.6 Singleton, annotated to show roles . . . 109
6.7 Hierarchy of patterns as defined in ρ-calculus . . . 110

7.1 Objects, methods and fields in POML . . . 112
7.2 Method call in POML . . . 113
7.3 Parameter declarations and result statements in POML . . . 114
7.4 Uses in POML . . . 115
7.5 Scoping in POML . . . 115
7.6 Update from a field access in POML . . . 116
7.7 Update from a method call in POML . . . 116
7.8 Emulating instance elements of a class in POML . . . 117
7.9 Using a class-object in POML . . . 117
7.10 Class inheritance in POML . . . 118
7.11 Implicitly inherited elements in POML . . . 119
7.12 Overridden methods in POML . . . 119
7.13 Static call to a superclass' method in POML . . . 119
7.14 Patterns in POML . . . 120
7.15 Field-based reliance operators in POML . . . 121
7.16 Example pattern definition snippet from **ObjectRecursion** . . . 122

8.1 Example C++ class . . . 126
8.2 C++ class converted to a class/object pair . . . 127
8.3 Example C++ nested namespaces and classes . . . 129
8.4 POML nested namespaces and classes . . . 129
8.5 Example C++ method details . . . 131
8.6 POML method details . . . 132
8.7 Example C++ nested expressions . . . 133
8.8 Equivalent code as expanded expressions . . . 133
8.9 Example external linkage declared C++ elements . . . 135
8.10 POML `__GLOBAL__` object corresponding to external linkage . . . 135
8.11 Example internal linkage declared C++ elements in SourceFile.cpp . . . 135

8.12 POML translation unit object corresponding to internal linkage . . . 136
8.13 Example **AbstractInterface** in source code . . . 137
8.14 POML emitted for **AbstractInterface** . . . 137

9.1 SPQR Overview . . . 141
9.2 SPQR `gcc` phase . . . 142
9.3 SPQR `gcctree2poml` phase . . . 143
9.4 Example OTTER input . . . 148
9.5 Example OTTER proof . . . 149
9.6 Example of mapping fundamental ς-calculus elements to OTTER . . . 151
9.7 Example of mapping ρ-calculus reliance operators to OTTER . . . 153
9.8 Representations in ρ-calculus and OTTER of a $<_{\mu}^{\circ \cdot +}$ derivation . . . 153
9.9 Multiple-form representation of Equation 4.6 . . . 154
9.10 Inheritance of implicitly defined method from superclass . . . 154
9.11 SPQR `poml2otter` phase . . . 158
9.12 SPQR `Otter` phase . . . 159
9.13 SPQR `proof2poml` phase . . . 160
9.14 Resulting default SPQR files . . . 162
9.15 SPQR training phase . . . 163

10.1 KillerWidget as UML . . . 174
10.2 C++ code for KillerWidget . . . 175
10.3 Portions of KillerWidget as OTTER input . . . 176
10.4 OTTER output for **Decorator** instance in KillerWidget . . . 177
10.5 Recovered **Decorator** in KillerWidget as POML . . . 177
10.6 Recovered **Decorator** pattern in KillerWidget as UML . . . 178
10.7 Explanation of symbols used in bounding formulae 10.17 - 10.25 . . . 193

11.1 Items related to comprehension . . . 197
11.2 Redistribution of items related to Comprehension . . . 198
11.3 Comparison of χ^2 for re-analyzed groups . . . 198
11.4 Reconfigured correlations . . . 200
11.5 Comparison of fit metrics for final models . . . 201

11.6 Recovered Gang of Four patterns in KillerWidget as UML 202
11.7 Pattern Involvement Metrics for Killer Widget 204

12.1 Example Pattern Dependency Graph 216
12.2 Two distinct methods 219
12.3 Equivalent non-temporal rules for both examples 220

B.1 Polymorphic approach 257
B.2 Subclassing approach 258

E.1 Example `gcc` dump tree format 319
E.2 Graph for example `gcc` dump tree 319
E.3 Example simple class 322
E.4 POML class element for example C++ code 323

LIST OF TABLES

5.1 Method calling styles in Gang of Four patterns 86
5.2 Reduced method calling styles in Gang of Four patterns 87
5.3 Final method calling styles in Gang of Four patterns 87

10.1 Creational Patterns from Gang of Four 169
10.2 Structural Patterns from Gang of Four 170
10.3 Behavioral Patterns from Gang of Four 171
10.4 All patterns found in NotificationCenter 181
10.5 All patterns found in C++ `std` namespace 183
10.6 File sizes 184
10.7 Performance data for major phases 184
10.8 Timing for phases of `gcctree2poml` 185
10.9 Timing information for OTTER inferences 187
10.10Comparison of normalized time estimates for search algorithms 188

Chapter 1

Introduction

1.1 Introduction

Maintenance costs in today's industry constitute the majority of expenditures for software development over the lifetime of a given product (Swanson, 1999). Estimates in various fields range from 60% of all business computing expenses due to COBOL alone (Banker et al., 1993), to a full 50-80% of all corporate information systems (IS) costs in the U.S. (Chan et al., 1996). Given the scope of the problem, it seems that reduction of maintenance costs will result in large overall savings to industry and consumers.

There is little previous research into reduction of maintenance costs through providing metrics to management and tools for software analysis. This introduction will present material from several aspects of software engineering research, economic and business sources, developer behavior studies, architecture, and current object-oriented programming practices, as supporting evidence for my thesis: that *it is possible to extract reliably high-level conceptual relationships known as design patterns directly from source code or other design artifacts, and do so in a practical, flexible, and automated manner.* I also further argue that the process of extraction, and its results, can provide two heretofore unattainable metrics: metrics which enable enhanced assistance for the maintenance task; metrics which provide relevant information to management for making economic decisions based on sound engineering principles.

1.2 Maintenance

I will use the standard definition of maintenance as per IEEE 729 for this document: "Modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment." (ANSI/IEEE, 1983; Schneidewind, 1987) Notice that this

definition is comprised of three distinct subdefinitions, elaborated upon by Swanson: one, work done to correct faults (again, as defined as by the IEEE), or bugs; two, adding minor features or performance to existing features; three, modifications made to reflect changing requirements or specifications. The cost of the latter two was estimated by Lientz and Swanson to be 75% of the total effort cost, in 1980 (Lientz and Swanson, 1980). The general assumption in recent literature is that the ratio is approximately the same (Chan et al., 1996; Swanson, 1999).

There seems to be a relative lack of technical and managerial attention focused on the maintenance phase of production when compared to initial development of a product; this is not to say that maintenance has not received any research or consideration. Some examples of current and past work include: Kemerer's work with maintenance oriented metrics (Kemerer, 1987), Banker *et al.*'s research on tying maintenance with complexity of software (Banker et al., 1993), Chidamber and Kemerer's now classic object-oriented design metrics (Chidamber and Kemerer, 1994), Hitz and Montazeri's measurement theory work (Hitz and Montazeri, 1996), Chan *et al.*'s production of an economic model for estimating the optimum replacement time for legacy software (Chan et al., 1996), Charette, Adams and White's analysis of managing risk during maintenance (Charette et al., 1997). In addition to these, Bieman, Kang and Ott's research has been instrumental in providing a link from the copious literature surrounding cohesion and coupling analysis to an object-oriented approach (Ott et al., 1995; Bieman and Kang, 1998; Bieman and Kang, 1995; Bieman and Ott, 1994; Kang and Bieman, 1996a; Kang and Bieman, 1996b).

All of these exemplary approaches have one or more drawbacks in the context of the maintenance cost problem as I define it below, and I will address each of them in turn in Section 1.6.

Only one thing is certain: "There is no commonly accepted measure of software maintainability." (Swanson, 1999) We only know that management of maintenance costs is a difficult problem.

1.3 Difficulties of Maintenance

There are three competing viewpoints on why maintenance is such a costly endeavor, falling along traditional lines in industry.

1.3.1 Technical

Because development and maintenance are, by their very natures, technical tasks, it is natural to ask if there is not something inherently wrong with the technology or its practitioners.

Management frequently blames developers as the cause of maintenance issues. Developers counter

that, if *allowed* to do their jobs correctly, quality would improve. Chester Barnard observed this tension as early as 1938 in other engineering disciplines: "Not to do something that is technologically 'necessary' because it conflicts with an organization code (as expressed for example in an economic interest) does great violence to the moral codes arising from technological fitness." (Barnard, 1960, p. 280) Disallowing work on economic grounds which developers determine 'necessary' on economic grounds will ultimately result in poor quality. Developers often have the training, desire, and focus to determine the technical needs of the work; they just may not have the tools to implement it.

A developer's most important tool is knowledge of programming languages. For the remainder of this dissertation I will concentrate on object-oriented languages. The abstractions inherent in object-oriented languages, and their relationship with comprehension, drove this decision. One of the most widely acknowledged problems with object-oriented designs is that of 'fragile code', where a seemingly innocuous change made to one module, which is presumably isolated and encapsulated, creates a domino effect of propagated changes. An obvious defense against this cascading effect is for developers to comprehend the code more thoroughly before making changes. An equally obvious counter-argument is that larger systems are more difficult to comprehend fully. I will address this in the next section.

To some extent, code fragility is dependent on the choice of implementation language and may be unavoidable. Some languages, such as Smalltalk, Eiffel, and Objective-C, have highly dynamic environments which are less susceptible to the issues associated with fragile code; others, namely C++ and Java, exhibit extreme fragility under certain conditions. Regardless of language choice, system design is the most important criterion for minimizing fragility (Charette et al., 1997). A system's architecture determines its flexibility for future modification. The larger the system, the larger the "difficulty in achieving conceptual integrity across different perspectives and different design requirements." (Goodhue et al., 1992)

To summarize, technical aspects lead to a need for tools for correctly implementing designs within constraints from management, increased comprehension of design, and a knowledgable system design.

1.3.2 Managerial

Management determines the resource allocations and thereby indirectly enforces design decisions. As a result, developers frequently blame management as the source of development issues. Management counters that their concern is the economic health of the group. Again, Barnard identified this timeless problem almost seventy years ago: "To do something that is technologically 'sound' but is economic heresy similarly destroys the general sense of economic appropriateness. It implies disregard of the

economy of the organization, and tends to its destruction." (Barnard, 1960, p. 280)

Management decisions are often based on deceptively simple numeric data, such as an expected completion date from group A, or a count of lines of code from group B, etc. These measures, or metrics, are often the foundation of economic decision making. Management understands this, yet fails to understand why engineering cannot provide simple numbers when programming is, ultimately, a numerical domain. Programming, however, is not simple; maintenance is even less so. Maintenance metrics are complex, and often poorly or inconsistently defined, as can be observed in the range of current metrics research (Banker et al., 1993; Basili et al., 1996; Chan et al., 1996; Chidamber and Kemerer, 1994; Etzkorn et al., 1999; Fenton, 1994; Kemerer, 1987; Gupta, 1997; Hitz and Montazeri, 1996; LaLonde and Pugh, 1994; Ott, 1992; Ott et al., 1995). Technical teams are, therefore, incapable of producing useful simple metrics which management requires.

One factor which deepens the rift between management and technical teams is that most metrics are produced for engineers, not managers with little technical knowledge. The context within which these metrics are valid is assumed to be a part of the recipient's worldview. Managers are generally not, nor should they be expected to be, technically knowledgable or competent to the levels of an engineer - their strengths and talents exist elsewhere. Expecting a manager to understand the various aspects and ramifications of using complex suites of metrics is unreasonable. Engineers should, however, be able to provide management with the necessary information, in the language they understand. Unfortunately, the tools and translations to fulfill this need do not currently exist in an easily-usable form. As systems grow, they become increasingly difficult to translate into simple quantities, and the issue quickly devolves into an endless cycle of inadequate understanding of the system.

1.3.3 Psychological Limitations

Perhaps the roots of maintenance cost problems transcend the debate between managerial and technical teams; perhaps, instead, they are based on general human traits. Both the technical and managerial discussions above resulted in aspects of the same key requirement: comprehension.

The works of von Mayrhauser and Vans, and Banker *et al.*, convincingly argue and show through rigorous experimentation and survey that comprehension is one of the most important facets of design and maintenance tasks (von Mayrhauser and Vans, 1996b; von Mayrhauser and Vans, 1995; von Mayrhauser and Vans, 1996a; Banker et al., 1993). Turley and Bieman show that one of the essential competencies for engineers rated 'exceptional' by management and peers is, "maintaining 'big picture' view," both in design and implementation (Turley and Bieman, 1995). Also, Wallace's 1999 doctoral work on software

project risk management provides the data which shows a strong correlation between comprehension issues and project success (Wallace, 1999), as I demonstrate in Chapter 11.

Comprehension of the external world is limited by the semantic tags attached to abstractions of concept and perception. Without these annotations, long-term retention is severely compromised. Unfortunately, short-term working memory is unable to capture effectively the structure, design, and nuances of large, or even medium size, systems. The average human short-term memory is only capable of holding seven, plus or minus two, items at any given time (Miller, 1957). For a human to retain larger-scoped or more numerous items, higher and higher levels of abstraction are required. The implication of this for software metrics design is that analysis of a larger system necessitates higher levels of abstraction, and that these more expansive abstractions must still, somehow, be reduced to meaningful numeric quantities.

Comprehension of large systems requires simultaneous use of multiple levels of abstraction (von Mayrhauser and Vans, 1996a), but does *not* require line-by-line understanding of the system *once abstractions have been identified* (von Mayrhauser and Vans, 1996a). von Mayrhauser and Vans define the terms 'clues' and 'beacons' as the various pieces of information the maintainer of a system uses during discovery and analysis of that system to create an abstract conceptual framework (von Mayrhauser and Vans, 1995). I will use these terms accordingly.

Brooks described the "one designer rule" (Brooks Jr., 1982), where a single architect is the repository for the entire design of a system. In light of the above requirements and limits of comprehension, there is a point at which further growth of a system necessitates higher levels of abstraction in order for that rule to hold. "Code size affects the level of abstraction on which maintenance engineers prefer to concentrate while building a mental representation of the program. [...] as code size increases subjects work less with low level program details than higher level functional descriptions of code." (von Mayrhauser and Vans, 1996a) Likewise, higher levels of abstraction allow systems to grow to sizes unattainable, for all practical purposes, with prior technologies. Each generation of language design has provided another level or style of abstraction, from pure binary to machine language, to structured programming, object-oriented programming, and now other recently-put-forth techniques such as aspect-oriented programming and design patterns. As might be expected, average code sizes have grown in step with this progression.

No single area outlined above can seriously be considered to address fully the maintenance cost problem, yet research in each area is highly compartmentalized. Refreshing exceptions to this are Kemerer and Chidamber's wide ranging works in metrics, management, and theory (Banker et al., 1993; Kemerer, 1987; Chidamber and Kemerer, 1994). A solution optimized for the creation of engineering design met-

rics is almost certain to be unsuitable for the creation of metrics quantifying the qualities desired by management (Fenton, 1994). A system which requires substantial retooling for use with another language, development platform, or business model will suffer from the same fragility of implementation that we seek to reduce in code. A good solution for the above problems would address each of the areas in some way, or allow for expansion into those areas through further research.

1.4 Object-Oriented Programming

Object-oriented programming is the latest successfully adopted attempt at managing conceptual complexity in software systems. Through its most common languages, notably C++, Java, Smalltalk, and Eiffel, object-oriented programming has spread the benefits of reuse, abstraction, and encapsulation, each which would seem to alleviate the probable technical causes for high maintenance costs outlined above. We find, however, that, "[...] in spite of the enormous growth of object-oriented technology, there seems little if any solid, repeatable evidence that it delivers any of its promised benefits." (Hatton, 1997) Object-oriented programming seems to have been widely adopted, but perhaps has not had as much impact as originally intended.

Object-oriented programming techniques extend the abstractions of encapsulation initiated with procedural programming, encapsulation of code, to the joining of data and code, hiding each from the user behind an interface. In theory, this increase in abstraction, reuse, and encapsulation has enabled increased comprehension of large systems (von Mayrhauser and Vans, 1995; von Mayrhauser and Vans, 1996a); whether or not it does so in practice is an issue of much debate.

One argument for the perceived increase in comprehension is that the project size has grown considerably in the past two decades. While some point to faster hardware as the enabling factor, decreased running times have no effect on the comprehension of large systems. Other arguments exist for other such advances.

Some point to improved software engineering processes. This is highly unlikely, for as DeMarco states in his analysis of the progress of process, "...I'm struck by two things: 1. our efforts to standardize and improve process have had positive effect, 2. but not by much." (DeMarco, 1996) He goes on to state that while the "net effectiveness was increased markedly, [...] little of it can be attributed to any kind of self-imposed standardization [of process]." (DeMarco, 1996)

The urban planner and architect Christopher Alexander remarked on this in his own field, citing the French philosopher and mathematician Poincaré: "Sociologists discuss sociological methods; physicists

discuss physics." Discussion of method and processes alone does not, and never will, bring the discussion to the subject under scrutiny. In Alexander's words: "No one will become a better designer... by following any method blindly." (Alexander, 1964)

Process management has a definite place in software production. It is not, and should not masquerade as, a substitute for software engineering and design.

If it is neither process nor hardware, then we must conclude it has something to do with the software technologies themselves. A survey of faults in code at NASA from 1976 to 1990 showed a drop from 8.8 faults/kLOC (thousands of lines of code), on average, to 6.0 faults/kLOC (Hatton, 1997). While is this only a decrease of 31%, the *high* number reported for the same period fell from 11.2 f/kLOC to 6.1 f/kLOC, a decrease of 45%. The low measure for the period changed only 18%. This indicates a normalizing of the number of faults injected into the system. While this may indicate the refining of a stable codebase,"between 20 and 50 percent of all enhancements or defect corrections made during maintenance introduce additional errors." (Charette et al., 1997) Design benefits are occurring, and we can only assume that they are coming about from increased experience of developers, and increasingly appropriate and powerful programming techniques.

Maintenance issues and techniques seem to be a moving target, because software systems continually grow in size, complexity, and cost, and object-oriented programming has changed little since its inception three decades ago.

Further metrics production

A basic precept of measurement theory is that a quality must be measurable before changing it in a reproducible, meaningful way (Fenton, 1994). Because many maintenance cost metrics were designed for use with programming languages implementing low levels of abstraction, they become less appropriate as systems grow. Recent work has produced object-level suites of metrics (Chidamber and Kemerer, 1994; Hitz and Montazeri, 1996; Chen and Lu, 1993), but none has much basis in object-oriented language theory. They must be adapted, instead, for use with each implementation language under scrutiny. I have yet to see proof that these metrics hold under such transformations. Without such proof, we run the risk of making these metrics non-meaningful according to measurement theory: "Formally, a statement involving measurement is *meaningful* if its truth or falsity remains unchanged under any admissible transformation of the measures involved." (Fenton, 1994) Furthermore, as the level of abstraction required to sustain manageable growth of systems increases, in order to remain relevant, metrics must be produced to accommodate the growth of systems, and the level of abstraction of the metrics themselves

must increase. Current metrics do not.

A few formalizations have been proposed for higher levels of abstraction (Eden, 2000; Etzkorn et al., 1999; Eden et al., 1999), but all fail to allow for the inclusion of the previous lower-level metrics within their analyses. Whether or not the transformations imposed by higher formalizations retain qualities which allow for code-level analysis is unclear.

Further levels of abstraction for comprehension

Increasing the level of useful abstraction available to the developer is worthwhile and should continue. The search for higher-level and useful abstraction to provide to the developer is worthwhile. This is a direct deduction from comprehension process research (von Mayrhauser and Vans, 1996a; von Mayrhauser and Vans, 1995), the historical progression of programming languages, and industry experience (Brooks Jr., 1982). Three areas of current research are of primary interest: packages, megaprogramming, and patterns.

The concept of a 'package' or 'framework' which captures multiple compilation units which provide a unified service is not a new one; recent research on producing language constructs, analysis tools, and metrics, however, is (Lakos, 1996). This is a variant of the encapsulation embodied in an object in object-oriented programming, though without the inheritance, polymorphism or other aspects of object-oriented programming which make it so powerful. The package concept in structured programming invigorated coupling and cohesion research, which has since found its way into the object-oriented world (Bieman and Ott, 1994; Lakhotia, 1993; Ott, 1992; Ott et al., 1995; Ott and Thuss, 1993; Patel et al., 1992).

Megaprogramming, "a technology for programming with large modules called *megamodules* which capture the functionality of services provided by large organizations like banks, airline reservation systems, and city transportation systems," (Weiderhold et al., 1992) is a more recent development. Fueled by a report by the Computer Science and Technology Board in 1990, megaprogramming seeks to encapsulate the specific problem domains of software systems, by providing "black box" systems which defy analysis from external sources. Instead, these systems rely on internal communities to maintain consistent interfaces to the outside world for extremely large and complex systems. While this seems to be a natural extension of the ideologies of object-oriented programming, it does not advance abstraction. The *size* of abstraction has increased, but the *level* of abstraction does not appear to have followed suit. Unfortunately, megaprogramming has recently disappeared from research literature, and it may or may not resurface.

One approach which offers increased abstraction is patterns. Patterns are a meta-language for programming which attempts to capture domain- and language-independent solutions for general classes of problems.

1.5 Patterns

One danger of increasing abstraction ad hoc is *reducing* the comprehension of a system by removing any association between the abstraction and actual semantics. A construct, or system, which ties the abstract and semantics together in a meaningful way is necessary.

Christopher Alexander posited in his 1964 work on architecture *Notes on the Synthesis of Form* (Alexander, 1964), that such a construct could be created, and termed it a *constructive diagram*, which later became known as a pattern. A pattern can be, depending upon whom you ask, "an abstract pattern of physical relationships which resolves a small system of interacting and conflicting forces, and is independent of all other forces, and of all other possible diagrams," (Alexander, 1964), "descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context," (Gamma et al., 1995), or as a consensus definition for this dissertation, *"a description of a general design problem that occurs repeatedly, and a general solution that is independent of the problem domain, the language in use, and any other constraints outside its intent."*

Alexander later expanded upon his patterns in two more groundbreaking books: *A Pattern Language* (Alexander et al., 1977) and *A Timeless Way of Building* (Alexander, 1979). I find it interesting that these two, which are much more specific to urban architecture and planning than his earliest book, are most frequently referred to in software patterns literature. During my research, I found exactly one reference to *Notes* (Coplien, 1996b). I will explore this oddity in section 1.5.1.

A pair of simple concepts exist at the core of Alexander's patterns: "the [pattern] is the bridge between requirements and form... every form can be described in two ways: from the point of view of what it is, and from the point of view of what it does. What it is is sometimes called the formal description. What it does... is sometimes called the functional description." (Alexander, 1964, p. 88-89) This is illustrated in Figure 1.1.

This dichotomy closely follows the requirements and code of software design, on a more general level. The requirements of the system are the forces which we must satisfy; they form the problem to be solved. The 'form', in Alexander's taxonomy, is the solution fitting those requirements, or the software product. The pattern is the general solution to the problem which we adapt for the specific problem domain we

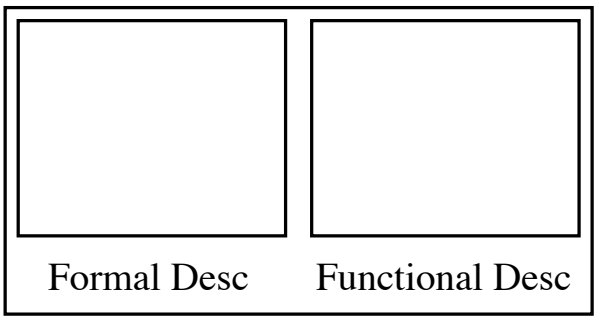


Figure 1.1: The two halves of a constructive diagram, such as a pattern

face.

We can further analyze this dual-nature in the context of the software system itself. If we view software as a complete solution, as in Alexander's definition, then it must have two components: formal and functional descriptions.

I assert that, in a software system, source code is the formal description, and documentation is the functional description. Source code is 'what it is', nothing more. Source code does, indeed, show you what it does, but *only if it is run through a translation system such as a compiler and run-time environment.* Even line-by-line human analysis requires the developer viewing the code to translate it into a meaningful runtime environment within his or her own comprehension framework. By itself, source code does not indicate what it does; it only provides a self-referential representation.

Similarily, if source code is the formal description, then documentation, in all its forms, such as manuals, comments, README files, etc., must be the functional description. Brooks observed, "As noted, formal definitions are precise... [what] they lack is comprehensibility. With English prose one can show structural principles, delineate structure in stages or levels, and give examples. One can readily mark exceptions and emphasize contrasts. Most important, one can explain *why.*" (Brooks Jr., 1982, p. 63)

Without a functional description, a formal description is inherently concrete and lacks the abstractions required for high-level comprehension. A slight blurring of this distinction occurs with names of variables, classes, and such, but this is hardly a sufficient functional description. The names given are generally mnemonics, at best, and do little to explain the larger design forces which determined the form of the solution. These names assist short-term working memory during localized modifications,

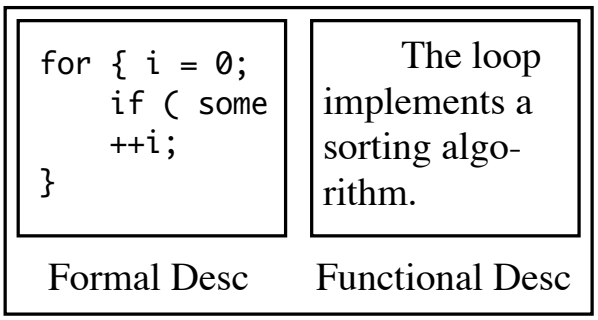


Figure 1.2: Software as an analogue to pattern definition

but are incomplete without a functional description. Note that the phrase 'functional description' is the same used by von Mayrhauser and Vans in their analysis of how engineers form abstract models (von Mayrhauser and Vans, 1996a). As shown in Figure 1.2, this description of software closely mirrors the definition of a constructive diagram, such as a pattern, as given by Alexander.

With the structure of a pattern established, and its relationship to software artifacts, I can address what patterns attempt to describe. Patterns are a description of a solution to a particular problem, along with all the forces and requirements leading up to the selection of a solution. In short, a pattern describes the relationships between all the positive and negative elements of a context, and the relationships between the finer-grained elements which comprise the proposed solution. These relationships become crucial as we further define and analyze them.

1.5.1 Formalization of Patterns

With the use of the term 'formal description' and my assertion that source code is a formal description for software products, it seems natural to assume that this level of formalism holds true for patterns as well, but that is not the case.

Patterns have been adopted in software engineering as a highly useful meta-language, one in which solutions can be expressed in an ambiguous way, and they follow the approach as later described by Alexander in *A Pattern Language* (Alexander et al., 1977) and *Timeless* (Alexander, 1979). These patterns can be adapted for use in almost any combination of programming language, programming environment, and problem domain. While this is useful for the functional description, it fails to support the formal description adequately. In *Notes*, Alexander described the two halves of a pattern using

two distinct types of language: very precise, specific quantitative descriptors (often graphical) for the formal description, and qualitative prose for the functional description. We see a similar distinction made in software engineering patterns: Unified Modeling Language (UML) (Rumbaugh et al., 2004) as a graphical notation for describing what the solution is, and natural language (generally English) for describing the contextual forces, implementation decisions, and so on.

Unfortunately, as James Coplien points out, there have been two rising camps of formalization of patterns: "The first school... attempts to automate pattern-based design... The second school of thought attempts to formalize patterns." (Coplien, 1996a) I would add to this a third, non-formal, camp: the third school of thought which attempts to eschew any formalizations of any sort, in an attempt to keep patterns 'pure'.

This third camp has adopted a Zen-like attitude towards patterns, asserting that patterns have a "quality without a name" which is undefinable, yet distinguishes a particular pattern from all others, or that which is not a pattern. There is no acknowledgement of the formal description side of a pattern, as Alexander originally defined it. It is my opinion that this attitude is antithetical to the very core of design patterns, as it violates the very purpose that Alexander originally put forth for patterns: lifting the unnameable and make it conscious. Asserting that there is an unnameable quality renders patterns as no better than the ambiguous and ad hoc principles of design that they are intended to supplant. This is, however, an unfortunate common belief perpetuated by some in the patterns community. In fact, quasi-patterns have now expanded to encompass many aspects of industry, development, and life, according to some practitioners. There are Process Patterns which describe processes, Managerial Patterns which lay down rules of thumb for management situations, etc. Most of these have no formal description at all, and fail to meet the requirements set down by Alexander. During the course of this paper, unless otherwise noted, I will use the term *patterns* to mean only Design Patterns: those patterns which provide a general solution for software design.

The second camp, pattern formalization, is growing, and comprised of much excellently done, if possibly mis-aimed, work (Eden, 2000; Eden, 1999; Eden et al., 1999; Allen and Garlan, 1997; Brown, 2000; Mikkonen, 1998). These researchers all have a common desire: to formalize the *entirety* of a pattern, both its formal and functional descriptions, as a single entity. These researchers make no distinction between the two halves, instead citing the ambiguous and fluid nature of the functional description as a barrier to automatic analysis.

Lastly, the first camp takes a similar tack to the second camp, but, instead of creating formal notations for patterns, it attempts to create tools for the automated application of patterns to specific problem

domains (Florijn et al., 1997; Bosch, 1998; Bosch, 1996). Again, this fails to account for the flexibility of the pattern which is inherent in the functional description.

The clash of ideology across these groups can possibly be traced back to Alexander himself. In *Notes*, Alexander put forth a highly formalized version of the pattern, complete with a fairly rigorous mathematical treatise on set theory of functionality, closely related in spirit to coupling and cohesion analysis in software engineering.

By the time of the writing of *Timeless* and *A Pattern Language*, Alexander had discarded most of the formalisms he created in *Notes*, stating instead "... once [*Notes*] was written, I discovered that it is quite unnecessary to use such a complicated and formal way of getting at the independent diagrams." (Alexander, 1964, Preface) Alexander is quite correct. His strict mathematical approach in *Notes* is *not* necessary for the *discovery* of patterns, but a formal description of some sort is necessary to an overall description *if at all practical*.

Alexander's discipline, urban architecture, does not lend itself well to formal descriptions. The forces that Alexander sought to capture include social, personal, biological and other such pressures which could not be quantified nicely. My belief is that this rationale is behind the move to more esoteric and abstract patterns in his later works.

Software, however, has a firm root in formal notations. The formal description of software most decidedly lends itself to a pattern's formal description using a formal notation. The entire pattern does not need to be formalized, nor would it be improved by doing so. The formal description, however, should be as formal as possible without losing the generality that makes patterns useful. Source code is, at its root, a mathematical symbolic language with well-formed reduction rules. We should strive to find an analogue for the formal side of patterns.

There must, however, be a limit to the formality of such an approach. A full, rigid formalization of objects, methods, and fields would only be another form of source code, invariant under some transformation from the actual implementation. This defeats the purpose of patterns. We must find another aspect of patterns to formalize.

1.5.2 Documentation Divide

Source code and documentation, the formal and the functional descriptions, comprise software solutions and might be assumed to provide a full solution. Unfortunately, that is not the general case. Source code has a specific purpose behind its creation: it must perform required tasks. Programmers often see documentation, a non-functional aid in comprehension, as incidental, cumbersome, and unnecessary.

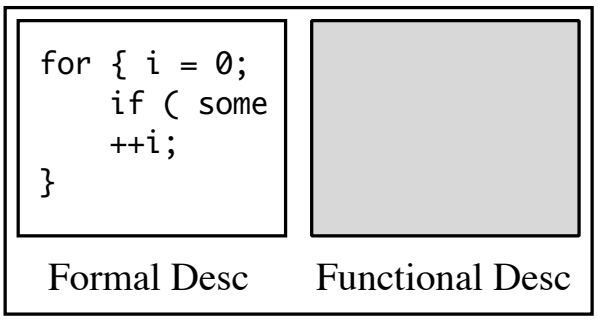


Figure 1.3: Incomplete solution in software

Code and documentation are rarely in sync, leaving the system as an incomplete solution.

There are several reasons why this occurs. Commonly, document rot sets in, where changes to the code are not reflected in the documentation or, worse yet, the comments within the source code. Complete documentation will not stay complete for long in most production environments.

A developer may also think that some aspect of the design or solution is obvious, trivial, or otherwise not worth mentioning. Unfortunately, that is rarely the case, even when the next programmer who must decode the source is the original author.

Lack of time and resources is another common reason for insufficient documentation. Frequently all available hands are required in order to ship a product, leaving no time or resources for updating the documentation.

While attempts have been made to help minimize this problem, such as literate programming (Knuth, 1984) and systems such as JavaDoc (Microsystems, 2005), the fundamental issue remains that documentation, even when tied to the specific point of interest, must be manually kept in sync with the algorithmic entities in the code.

von Mayrhauser and Vans state another view, indicating that keeping documentation current may not be enough: "current practice of documentation and coding does not encourage efficient understanding as it compartmentalizes knowledge by type of document and rarely provides the cross references that are needed to support programmers' cognitive needs." (von Mayrhauser and Vans, 1996a)

All of the above result in a lack of comprehension of the system, both by current developers and future maintainers. By abandoning the functional description through various means, they lose half the solution of the software, as shown in Figure 1.3, and lay the foundation for a maintenance nightmare.

1.6 Metrics

The addition of patterns to the current set of conceptual models and tools in software engineering may seem like overkill. After all, several techniques and notations for producing pseudo-formal documentation for code exist, and many are designed to be language-independent. Documentation, as a functional description, has similarly existed in one form or another. These tools, however, are not sufficient.

The error lies in not recognizing the prime contribution of patterns as a comprehensive tool: patterns are by their very nature comprised of the formal *and* functional descriptions. One without the other is not a pattern. A pattern encapsulates a concept, with the functional description performing the role of an interface, and the formal description acting as an implementation.

A pseudo-formal notation, such as the Unified Modeling Language (UML), or an abstracted pseudo-code interface description language (IDL), or even an abstract domain-specific specification language (DSSL), will provide only half of the pattern's information. This is evident by the natural language descriptive text which almost always accompanies notational documents such as these. If the formal half were enough, the natural language description would be superfluous.

It seems that we can replicate the functionality of patterns with existing tools, but there is one remaining aspect of patterns which makes them unique: community adoption of particular patterns as part of an accepted meta-language. Patterns are written up by their discoverers and submitted to the general patterns community for peer review. If a pattern is generally accepted as a) new, b) a previously unidentified solution/problem, and c) complete, it is brought into the canonical literature. In this way a standard taxonomy, a common and programming-language-independent vocabulary, of patterns is created, such that a pattern means the same abstraction to every developer familiar with it, and encapsulates a larger programming construct within a single term.

Patterns are one of the higher-level abstractions we previously identified as necessary for increased comprehension, *and* they come created with functional descriptions. For developers, the common lexicon of patterns allows intercommunication without resorting to formal descriptions, yet formal descriptions allow precise specification as to the implementation outline.

1.6.1 Syntactic metrics

Current metrics are primarily concerned with linguistic syntax counting, rather than with the relationships and interconnections which define patterns. While this is a useful and highly practical approach, it does not enable a level of abstraction which fosters broad comprehension.

In addition, those metrics which do push the boundaries of relational measurement (Eden, 2000; Eden et al., 1997; Etzkorn et al., 1999; Hitz and Montazeri, 1995) do so through additional syntactical measures. Furthermore, they fail to tie these with a meaningful measure of *semantic* difficulty of change. This change in the comprehension realm is most likely to be an indicator of cost of change to the system: "Factors that increase maintainer effort will increase project cost, since maintenance costs are most directly a function of professional labor component of maintenance projects." (Banker et al., 1993) Banker, *et al.*, go on to say that the necessary effort of the maintainer is determined highly by his or her "effort to comprehend the software." (Banker et al., 1993)

While the above systems are excellent at measuring various quantities, they all fail to measure *quality*. Quality is a highly subjective term. A singular system may be of high quality from one viewpoint, such as running time, and low quality from another, such as designing for maintenance. While measurement of specific quantities is intended to provide an indicator of the relative quality of a system, quality remains a highly intangible concept. This is one the few areas where the factions of measurement theory, metrics design, patterns, and formalization theories, agree (Fenton, 1994; Eden, 2000; Eden, 1997; Coplien, 1996b).

A meaningful measure of quality must be mapped to a quantifiable axis. Concepts of quality must be mapped to semantics, which, in turn, must be mapped to symbolic languages for quantification. These mappings have been poorly defined in current research, if defined at all.

Current research speculates that these syntactical quantification metrics are insufficient to produce the quantities necessary for efficient management. Management often requires a measure of costs and benefits, whether directly or indirectly derived. The production of software also eludes definition, except those portions selected as relevant to process analysis and management. Unfortunately, such analysis is often unrelated to the production of meaningful cost-based data for management.

1.7 Concepts Programming

Given the problems outlined above with current models and metrics, and given the existence of patterns and software as two systems following the same general definition of a complete solution, a path for research presents itself.

The maintenance problem is fundamentally a problem of comprehension. Comprehension is limited in no small part by a the amount of correct and useful functional descriptions accompanying source code. Comprehension of large systems relies on high levels of abstraction. Patterns are a common

language of high-level abstraction which are comprised of the same two components necessary for code comprehension, and tie the two together in an established and unbreakable way. A path from code to comprehension is necessary.

1.7.1 From code to comprehension

As discussed in the previous section, software systems are usually incomplete solutions. A formal description exists, necessarily, in the form of source code, but a functional description is lacking, and frequently missing altogether due to the reasons given above.

We can start to classify the parts of a software system solution by noting that every descriptive system is a symbolic one. Source code is a mathematical symbolic language, regardless of its implementation syntax. Every procedural language is, fundamentally, an implementation of lambda-calculus (λ-calculus) (Stansifer, 1995), and every object-oriented language based on procedural languages is reducible to λ-calculusas well. If it could not be, then an object-oriented language would have theoretical computing expressive power above and beyond that of procedural languages. This is not the case. The practical and conceptual expressiveness of object-oriented languages is, in many cases, higher than its procedural counterparts, but the computational power is unchanged (Abadi and Cardelli, 1996).

Documentation, however, is a *conceptual* symbolic language, generally in the form of natural language and/or diagrammatic notation. Conceptual languages are generally too ambiguous to be used for rigorous translation to a fully-reducible mathematical language or interpretation. Only by yoking the rigorous and ambiguous together do we get a meaningful whole for our purposes.

Documentation is, unfortunately, generally missing, incomplete, or, in some cases, *inarticulable* by the developer (Bosch, 1998). The latter frequently occurs when the developer has a strong intuition regarding a design which 'feels right', but is unable to state exactly why. Even more unfortunate is that fact that comprehension relies on this functional description, regardless of its form (von Mayrhauser and Vans, 1996a). Lacking a functional description which would aid design comprehension, and, lacking the comprehension which would enable a robust description, developers are locked into an unfulfilling cycle.

Patterns mirror the dual structure of complete software, comprising another complete solution. Patterns are language- and implementation-independent, and provide solutions to general classes of problems rather than concrete boilerplate code.

The formal description portion of patterns is generally achieved through UML, a pictoral symbolic language which represents the relationships between programmatic constructs in code; these relationships are the focus of pattern design. Natural language, as in software documentation, expresses the

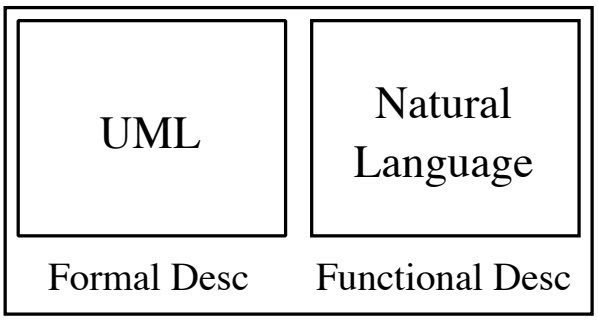


Figure 1.4: Basic pattern structure

functional description. Also, as in software, this is a conceptual symbolic language, and necessary for true comprehension of a pattern. The full solution is illustrated in Figure 1.4. Unlike software, however, patterns tie formal and functional descriptions together inextricably, and do so in a community-accepted language for their communication.

Given that patterns contain explanations of solutions to problems frequently found in code, matching existing code to the functional description of a pattern embodied in code would provide a powerful mechanism for discovery during maintenance by mapping elements of code to the *functional* descriptions of interest. Creating a path to match an existing or proposed implementation to functional pattern is not trivial, but possible. Assume a mathematical symbolic language to which both source code (mathematical symbolic) and UML (pictoral symbolic) can be mapped as in Figure 1.5. Also assume that, within this intermediate language, general patterns of constructs can be searched for efficiently. We can now seach for UML constructs in source code. By searching for the construct defined within a pattern, we can identify probable uses of that pattern. The functional description of the pattern then provides us with a powerful clue as to the developer's intent.

There are requirements for this intermediate language. It should be, at the very least, a pure mathematical language, with clean and simple reduction rules. The rules and notation of this language should properly capture not only the concrete items of programming languages, but also should be able to encode the relationships between such items that are expressed in UML.

This is not a formalization of patterns per se, as others have attempted. In no way do I claim that it is a formal language for the production of patterns. Instead, I have formalized those portions of a pattern amenable to formalization, and have allowed the functional descriptions to remain flexible and

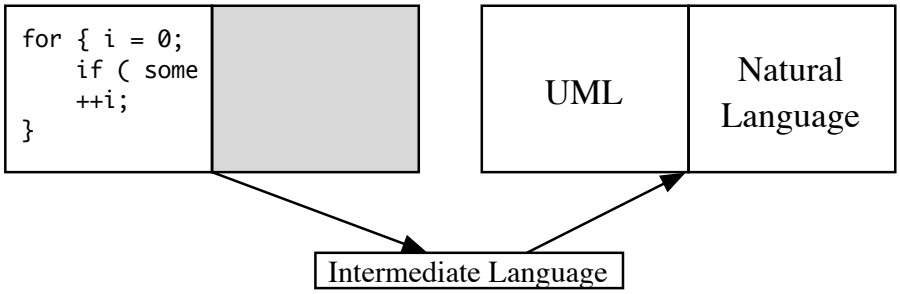


Figure 1.5: Path from code to comprehension

appropriately stylistic.

At first glance, this path might seem to be of limited usefulness because it ties source code to an ambiguous natural language description of a general solution. Patterns, however, are consensus solutions to *common* programming problems, and are, in many cases, the combined expertise of many developers. While this may seem to encourage a design-by-comittee attitude, and, therefore, a weakness of design, patterns have with only a few exceptions, held up remarkably well when subjected to a common interpretation. Those exceptions have generated discussion and debate and have given way, in turn, to more detailed and precise classes of patterns in a classic iterative manner. Patterns can be taken as canonical solutions to general problems of design, because of this history.

Consequently, if a pattern is found implemented in code, *chances are* that the same problem the pattern describes was being solved. The functional description component of the pattern provides a clue, or beacon, to the problem the developer intended to solve, and a set of likely forces which shaped the implementation design.

One ramification of the ability to find instances of patterns in code is that these beacons would still be visible in cases where the developer was not consciously using patterns, and arrived at the solution independently. Because patterns are derived from common solutions to common problems, this is a powerful tool, which allows for the deduction of comprehension clues and commonly accepted abstractions from code that was not designed with such explicitly in mind.

While this may seem highly speculative, it is not without precedent. Several approaches have been attempted that directly parallel this concept, such as Maintenance by Abstraction (Schneidewind, 1987), TMM (Arango, 1986), and composite module cohesion analysis (Patel et al., 1992). All have succumbed to a lack of common terminology as a basis for communication.

1.7.2 Pattern Compilers

Reversing the above path is a natural desire for engineers, creating code from patterns. Pattern language compilers came into existence soon after the introduction of patterns to software engineering.

The fundamental problem which pattern compilers face is that patterns, again, consist of two types of description. If the formal description, that which is translatable to source code, were complete, patterns would not be necessary. Instead, the designer must incorporate the pattern into the current problem domain in a meaningful way. Compilers cannot, except in the most trivial of cases, add to the implementation within a problem domain.

Given that limitation, however, some excellent compilers and related technologies have been produced. COGENT (Budinsky et al., 1996) is a scripting language which allows a developer to map patterns to an implementation language. These scripts, unfortunately, can become quite complex and may require retooling for new problem domains. Also, each implementation language requires a different set of scripts. van Winsen's OMT-Tool (van Winsen, 1996) is a re-engineering tool that supports some code generation from pattern specifications, but is more closely tied to a code implementation than to a general pattern compiler. Boschs' LayOM (Layered Object Model) (Bosch, 1996; Bosch, 1998) proposes extending language implementations to allow for direct inclusion of patterns semantics in programming. This would allow standard language compilers to handle pattern attributes directly, but would require redefinition of existing languages for widespread use. More recently, the Pattern Enforcing Compiler project, PEC (Lovatt et al., 2005), has emerged as a compiler that detects conformance of a class to a pattern. I will discuss PEC in more detail in the next chapter.

1.8 Conceptual metrics

With a demonstrable technique for assisting comprehension, it is time to revisit metrics. As stated in Section 1.3.3, any proposed method of reducing maintenance costs must address, in some way, each of the areas of identified problems of comprehension, communication, and collaboration.

1.8.1 Design metrics

Patterns express more than just code constructs; they express the *relationships* between constructs. This gives us another level of attributes to measure. While most current metrics in this area concentrate on more syntactical measures of design (Chidamber and Kemerer, 1994; Etzkorn et al., 1999), I expect a purely relational view significantly to benefit the production of more abstract metrics, such as 'mainte-

nance cost due to design issues', 'quality of design with respect to a certain design axis', and 'effort of reworking versus rewriting a code section'.

1.8.2 Malignant pattern detection

One such application is the search for patterns which are explicitly *poor* solutions to common problems. A controversial term, 'Anti-Patterns' (Brown et al., 1998), has arisen to describe solutions or designs which should be avoided. Given the explicitly ambiguous nature of the literature in this area, spanning the variants of patterns in Management, Behavior, etc, I will use the term *malignant patterns*, which I define as: *a design pattern which describes a solution to a general progamming problem, where the solution is known to cause multiple, or more severe, problems than the one it was designed to solve.* While this is close to the definition given for Anti-Patterns, I restrict it to the design of a software product, and do not presume to identify solutions that are globally 'sub-optimal'. Every design decision involves tradeoffs, and those tradeoffs cannot be known *a priori* for a particular domain or system.

Malignant patterns are widespread and well documented in one form or another. Every language community has its own, generally informal, taboo structures or solutions. The *Effective C++* series from Scott Meyers is an excellent example of this. While there are language-specific rules which result from the language definition and/or implementation, there are more universal rules and principles which are common among all object-oriented languages.

The pattern from the Anti-Patterns literature called The Blob, aka, The God Class, is an example of this type of malignant pattern. The Blob is described as a malignant pattern in which "procedural-style design leads to one object with numerous responsibilities and most other objects only holding data." (Brown et al., 1998) Unfortunately, The Blob is common, and arises from situations in which a developer has not planned proper encapsulations for what should be multiple classes. Frequently, this is the result of an attempt at refactoring a legacy procedural system into an object-oriented design, without fully understanding the differing methodologies. This is a classic case where cohesion and coupling analysis could provide a recommended solution, but the mere identification of such a problem requires the same extensive analysis as the production of the recommendation. Being able to encode the relationships between the methods, fields, and classes involved, instead of the underlying constructs themselves, speeds the process considerably.

Another example, not defined as a pattern, but involving a simplified cohesion and coupling analysis, is Lakos' identification of interlocked classes in C++ (Lakos, 1996). While language-specific in his definition and analysis, the same principles extend to object-oriented languages in general in one form

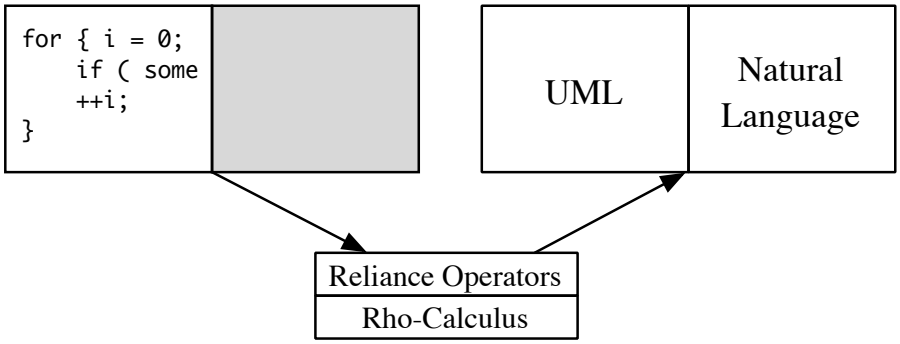


Figure 1.6: Completed path

or another. Lakos' solution of *layering* requires a substantial level of analysis similar to coupling and cohesion scrutiny, when attempting to reduce the problem.

1.8.3 Multi-scale analysis

The need for analysis of code at a fairly low level is a common theme in the above examples, as with many substantiative critical analysis techniques. At a minimum, this requires the ability to peer within method bodies. Creating an intermediate language which provides the expressive power of object-oriented languages as a whole enables the production of quantitative data on fine grained features.

The approach described in this chapter can work at many levels simultaneously, nicely mirroring the same observation by von Mayrhauser *et al.* on comprehension processes (von Mayrhauser and Vans, 1996a). A design can be submitted for analysis prior to implementation, and particular analyses can be performed. Alternately, the code for a previously implemented system can be submitted for comprehension assistance, or for identification of problematic constructs which will likely to lead to increased maintenance costs later.

We gain the ability to discard the vast amount of information we are *not* directly concerned with which source code carries around, now that we have distilled the essential qualities of relationships and reliances within the code with which patterns are concerned. The general executable dynamics, the techniques used on various hardware, the low-level details of implementation - these are all free for us to abandon as necessary, yet they are all available within a proper denotational semantic translation if we desire to work with them. This is a critical distinction between my technique and other attempts. I will discuss this with other related work in the next chapter. The completed path from code to comprehension is shown in Figure 1.6.

1.8.4 Relationships

Patterns are based on the interconnections and relationships between language constructs, not the constructs themselves. I have identified patterns as the leading candidate for automated production of beacons to aid comprehension. Extending the abilities of expressive formalisms to enable the inclusion of these relationships, is, therefore, necessary, before we can treat patterns and raw source code on an even basis.

James Coplien offers a more practical viewpont:

> Most architecture documents still capture just APIs and data structures, missing the important nuances of relationships. And most architecture documents still pretend that architecture can be divorced from downstream design and implementation concerns, which runs counter to predominate empirical practice, particularly in successful development organizations. And few architecture documents draw in issues of paradigm, process, expertise, organization, aesthetics, or even the obvious concern of long-term maintainability. (Coplien, 1996b)

Formalization researchers often attempt to divorce formalizations from implementation details, and have made limited progress towards useful metric production as a result. The formalizations I define in this dissertation merge the low-level quantification possibilities of multiple denotational calculi with higher level relational qualities. By doing so, I have produced a system which spans a vast range of potential usefulness, while retaining a simple unifying base.

Objects embody a set of relationships between data and associated methods, as a conceptual whole. This encapsulation of implementation and abstraction of concept has been a powerful tool for managing complexity and comprehension. The next step in this evolution is the encapsulation of relationships between objects co-existing in a system, and with their atomic elements. At some point, every such element, from data to object, will come in contact with some other element, and a relationship will be formed. These inter-object relationships are the basis of patterns, and as such are the focus of my research.

1.9 SPQR

With these pieces in place, a brief, but useful overview of my research is now possible. I have named the technique and toolkit the System for Pattern Query and Recognition, or SPQR. I will describe four main areas of research and implementation, and several more identified areas of potential future research.

SPQR consists of five major parts: a formal denotational semantics for describing relationships between elements of programming called the *ρ-calculus*; the *Elementary Design Patterns*, a catalog of

the basic concepts of programming organized in suites that reflect their formal natures; a common data format for representing the formal and design pattern elements of programming, the *Pattern/Object Markup Language*; a mapping of C++ to this common format; and a set of tools to operate on source code files and produce reports on the detected design pattern instances.

1.10 Advantages

I will now revisit the three areas of contribution to high maintenance costs which I specified in section 1.3, and show how the combination of a formal semantics based on relationships, and patterns can help.

1.10.1 Technical

A mapping from object-oriented languages and UML to ρ-calculus allows direct translation to a common language. Problematic interdependencies or relationships can be searched for within UML diagrams, within a design and before implementation. Alternately, existing code can be translated to the intermediate language and similar analyses performed. Most importantly, existing code and proposed design changes can be analyzed *together*, by incorporating both into the same system.

The ability to consider modifications in the context of the previous implementation is powerful. The bidirectional nature of the formalisms allows a system architect to make informed decisions and to observe the propagated effects of proposed changes. Also, improvements to the original code can be experimented with at a design level before incurring the high cost of implementation. This can help reduce the amount of fragile code, both before a system is implemented, and later, to reduce the amount already created. Most importantly, this is independent of the implementation language. By adding this layer of abstraction, language-dependent problems can be addressed independently of the larger design issues.

1.10.2 Managerial

As previously discussed, managers suffer from a lack of relevant metrics from their engineering staff. The most desirable metrics to management are those which require qualitative descriptions based on quantitative measurements. This is a parallel to the functional/formal nature of patterns, and I anticipate that the most useful metrics will take advantage of the patterns' duality.

Design-based metrics produced for management should adhere to, and incorporate, both computer science and management science literature (Benbasat and Weber, 1996). Matching these metrics with

a solid set of suggested practices for detected problems will provide a toolset that satisfies Benbasat and Zmud's requirements for relevance: a current (and future) problem, an interesting solution, and an applicable set of prescriptive guidelines (Benbasat and Zmud, 1999).

1.10.3 Human limitations

Enabling the discovery of existing patterns in code provides a mechanism through which a developer can obtain clues to the original design of a codebase; these clues are further expressed in the common language of patterns, and are easily documented.

Tying source code to the functional descriptions supplied by patterns allows the developer to identify high-level design issues more easily, comprehend them in an abstract fashion leading to better long-term retention, and spend precious resources concentrating on the problem domain at hand, rather than attempting to comprehend the larger design (von Mayrhauser and Vans, 1996a; von Mayrhauser and Vans, 1995; von Mayrhauser and Vans, 1996b).

1.11 Unresolved issues

It would seem from the above discussion that the combination of patterns and a relationship-enhanced calculus is a panacea to maintenance problems. Unfortunately, several issues remain untouched by this approach.

1.11.1 No silver bullet

Designing a system is not a straight-forward task. There are many axes of design to consider, such as runtime speed, memory use, feature set, flexibility of use, portability, and, of course, maintainability. The only axis I address here is that of maintainability. A combination of the other forces pushing a design in various directions is a constant at this time. The human architect makes the final decisions regarding where best to place the design within the design space.

My goal in this work is to provide a framework on which those metrics useful to the consideration of maintainability can be based, and a research platform from which investigation of those metrics can be implemented practically. This system has *not* been designed to create a single, all-encompassing metric of 'complexity'. As Fenton clearly shows, "the search for a general-purpose real-valued complexity measure is doomed to failure" (Fenton, 1994) by the very principles of scientific measurement theory. I do not claim that this system will enable the production of such a measure. Any metric must carry a context

and conceptual weight if it is to be useful in simplifying complex systems, and that context eliminates the possibility of a 'general-purpose' value of complexity.

1.11.2 False positives

In my argument for the use of pattern discovery as an aid to comprehension in Section 1.7.1, I mentioned that it may be possible for this technique to discover pattern use where the designer had no intention or knowledge of patterns. Similarly, spurious patterns may be identified through accidental artifacts of implementation. This could lead to improper clues being given to a developer using this approach. I will discuss my results concerning this problem, as well as potential avenues to reduce this. My assertion still stands, however, that comprehension is at this time a uniquely human endeavor, and that an engineer can only be aided by a system such as SPQR, not replaced. The formalisms defined in this document are a best attempt at minimizing conceptual false positives, but I am under no delusion that they will completely eliminate them. In my experience, confirmation of an abstraction is inevitably faster than detection of that abstraction in a large codebase. Therefore, I am less concerned with zero tolerance for false positives than I am with ensuring no false negatives.

1.11.3 Maintenance

Ultimately, we have to ask if the above limitations matter for the reduction of maintenance costs. A design space has many facets which cannot all be optimized simultaneously. A designer must use his or her own judgment in the final decisions regarding not only which aspects to refine, but also how much to pursue as valid, of the information given to them by the analysis on the comprehensibility and maintenance axes. In practice, this combination of shortcomings is not so burdensome as it may first appear. Weighing various axes of design *is* the designer's job, and lending relevance and credibility to various source of information is an ongoing part of that task.

1.12 Summary

In conclusion, the maintenance cost problem of software production is firmly rooted in a lack of system comprehension. Comprehension is hobbled by the lack of current and relevant functional descriptions to accompany the formal description of source code. When these functional descriptions do exist, they are frequently out of date with respect to the source code, or inappropriate in their level of abstraction. As

a system grows in size and complexity, it requires higher levels of abstraction for solid comprehension by the maintainer.

Patterns provide a level of abstraction above that of object-oriented programming, and can encapsulate entire design issues in a common language. Tying the functional descriptions of patterns to the formal descriptions of source code requires a translation mechanism for the formal descriptions involved, both UML and source code, to a common mechanism. This mechanism, based on existing and established denotational semantics, with extensions to support the relationships described by patterns, provides the missing link between source code and a general, common language for frequently used abstractions. Identification of these abstractions makes possible three things critical to the reduction of maintenance costs:

- Identification of pattern usage to facilitate comprehension of large systems at a high level of abstraction.
- Identification of malignant patterns in a design, prior to implementation, where these patterns are determined to be potential maintenance hazards.
- Identification of malignant patterns in an existing system, such that resources can be better allocated for maximum strategic benefit.

In the next chapter, I will introduce related work in the field of assisted software comprehension, with a primary focus on the detection of design patterns from source code and proposed designs. I will then discuss the formalisms of SPQR, providing a basic tutorial on ς-calculus, and extending that to introduce ρ-calculus and my notations. These will then be used to facilitate the discussion of the Elemental Design Patterns, the conceptual building blocks of object-oriented programming. These blocks will then be used to construct more abstract patterns from existing literature, using ρ-calculus as the conceptual glue. After these foundations of the theory of SPQR are established, I will introduce the Pattern/Object Markup Language, a common data format for the pieces of the SPQR toolset, and show how it is a concrete representation of the previously defined formalizations. I will then introduce SPQR's implementation and tools and present and discuss the results of validating SPQR on existing source code. To show how SPQR can be used to fulfill the original research proposal, a suite of metrics for patterns and comprehension will be defined. Finally, I will establish a number of future research directions.

Chapter 2

Related Work

I mentioned many technologies and research projects in the introductory chapter, but I would like to expand on some of them here. I will discuss existing and prior research related to pattern decomposition, pattern formalizations, other relationship analysis techniques, the state of conceptual metrics, and pattern detection.

2.1 Decomposition of patterns

The decomposition and analysis of patterns is an established idea, and the concept of creating a hierarchy of related patterns has been in the literature almost as long as patterns themselves (Coplien, 1998; Kristensen, 1994; Riehle, 1997; Zimmer, 1995). The few researchers who have attempted to provide a truly formal basis for patterns have most commonly done so from a desire to perform refactoring on existing code, while others have attempted the more pragmatic approach of identifying core components of existing patterns in use. Additionally, there is ongoing philosophical interest in the very nature of coding abstractions such as patterns, and their relationships.

2.1.1 Refactoring Approaches

Refactoring (Fowler, 1999) has been a frequent target of formalization techniques, with fairly good success to date (Demeyer et al., 2000; Moore, 1996; Opdyke and Johnson, 1993). Its primary motivation is facilitating the tool support for, and the validation of, the transformation of code from one form to another while preserving behavior. This is an important step in the maintenance and alteration of existing systems, and patterns are seen as the logical next abstraction upon which they should operate. To date, however, effective pattern use in automated refactoring assistance has been absent.

2.1.2 Fragments

Fragments, as developed by Florijm, Meijers, and van Winsen (Florijn et al., 1997), provide a practical implementation of pattern analysis and coding support in the Smalltalk language, and demonstrate the power of application of these concepts. Their *fragments* are abstractions of a design elements, such as classes, patterns, methods, or code, and contain roles, or *slots*, which are filled by other fragments. In this way they are bound to each other to produce an architecture in much the same way that objects, classes, and such are in a working system, but the single definition of a fragment allows them to work with all components of the system in a singular way. This approach, while successful in assisting an engineer working with a system, does have some limitations. Detection of existing patterns in the system was deemed unlikely, due to the fact that "many conceptual roles did not exist as distinct program elements, but were cluttered onto a few, more complex ones." This indicates that there may be a lower level of conceptual roles to address, below fragments.

2.1.3 Minipatterns

Ó Cinnéide's work in transformation and refactoring of patterns in code (Ó Cinnéide, 2001) is an example of the application of *minipatterns*, portions of patterns that are used to compose larger bodies. Ó Cinnéide treats the minipatterns as stepping stones along a refactoring path, allowing each to be a discrete unit that can be refactored under a *minitransformation*, in much the same way that Fowler's refactorings (Fowler, 1999) are used to incrementally transform code at and below the object level. These minipatterns are demonstrated to be highly useful for many applications, but cannot capture some of the more dynamic behavior of patterns, instead relying heavily on syntactic constructs for evidence of their existence.

2.1.4 Structural Analyses

An analysis of the 'Gang of Four' (GoF) patterns from the Design Pattens text (Gamma et al., 1995) reveals many shared structural and behavioral elements, such as the similarities between Composite and Visitor, for instance (Gamma et al., 1995). The relationships between patterns, such as inclusion or similarity, have been investigated by various practitioners, and a number of meaningful examples of underlying structures have been described (Bosch, 1998; Coplien, 1998; Riehle, 1997; Woolf, 1998a; Woolf, 1998b; Zimmer, 1995).

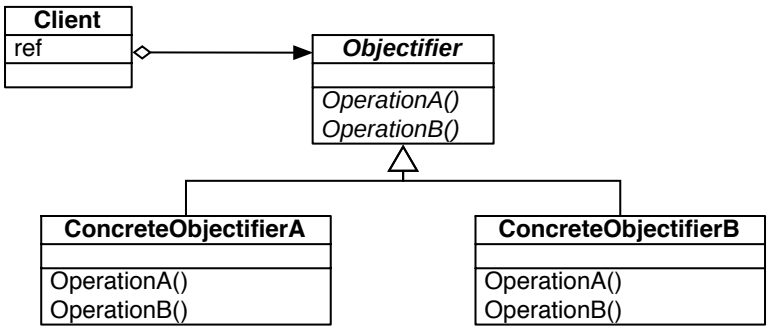


Figure 2.1: Objectifier class structure

Objectifier

The Objectifier pattern (Zimmer, 1995) is one such example of a core piece of structure and behavior that is shared between many more complex patterns. Its Intent section states:

> Objectify similar behavior in additional classes, so that clients can vary such behavior independently from other behavior, thus supporting variation-oriented design. Instances from those classes represent behavior or properties, but not concrete objects from the real world (similar to reification).

Zimmer uses the Objectifier as a 'basic pattern' in the construction of several other GoF patterns, such as Builder, Observer, Bridge, Strategy, State, Command and Iterator. It is a simple yet elegantly powerful structural concept that is used repeatedly in other patterns.

Object Recursion

Woolf takes this pattern one step further, adding a behavioral component, and naming it Object Recursion (Woolf, 1998b). The class diagram in Figure 2.2 is extremely similar to Objectify, with an important difference, namely the behavior in the leaf subclasses of *Handler*. Exclusive of this method behavior, however, it appears to be an application of Objectify in a more specific use. Note that Woolf compares Object Recursion to the relevant GoF patterns and deduces that: Iterator, Composite and Decorator can, in many instances, be seen as containing an instance of Object Recursion; Chain of Responsibility and Interpreter do contain Object Recursion as a primary component.

2.1.5 Conceptual Relationships

Taken together, the above instances of analyzed pattern findings seem to comprise two parts of a larger chain: Object Recursion contains an instance of Objectify, and both in turn are used by larger patterns.

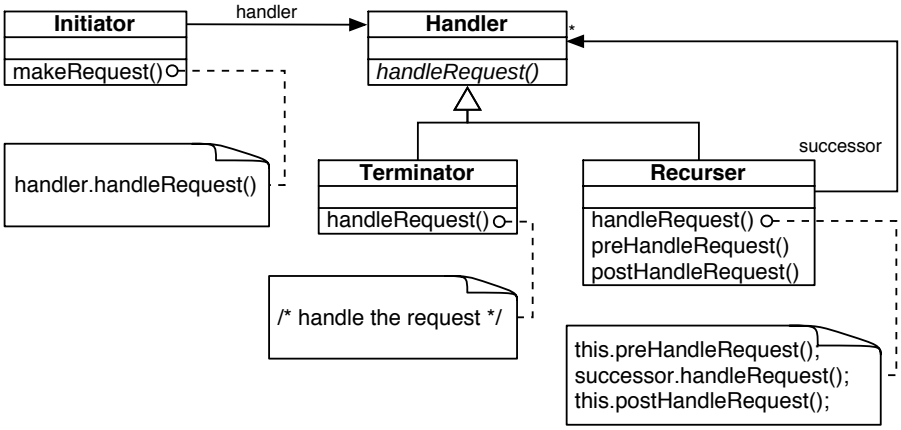


Figure 2.2: Object Recursion class structure

This indicates that there are meaningful relationships between patterns, yet past work has shown that there are more primary forces at work.

Idioms

Idioms are defined by Buschmann as "a low-level pattern specific to a programming language." Many such useful examples abound, for languages from C++ (Coplien, 1998) to SmallTalk (Beck, 1997), and provide a pragmatic set of solutions to common problems using best practices, the other common term for such constructs. Coplien in particular has been instrumental in investigating the relationships between idioms, constructing graphical hierarchies of use. While idioms are highly useful in practice, they fall short for generalized use and analysis of object-oriented systems in general, due to their language-specific nature. They do, however, exemplify the helpfulness of describing best practices in concrete terms, a technique continued in the design pattern philosophy.

2.1.6 Other Work

Elementary Patterns

These patterns attempt to provide a lower level of programming best practice, encoding constructs that will make sense to a student programmer, to facilitate education. This is a notable and worthy practice, and these patterns are indeed elementary, consisting of topics such as FORLOOP, LINKEDLIST and others that are of interest to the student (Bergin, 2003).

While these are a helpful area of documentation for the student, they offer little to no discussion of the architecture of *object-oriented* programming. Instead, they are solutions to problems that all procedural

programmers face. The naming is perhaps unfortunately close to my own Elemental Design Patterns, to be discussed in Chapter 5, but there is a distinct difference between 'elementary' and 'elemental'. Elementary patterns are simple patterns for the beginning student, illustrating various programming techniques that may or may not involve the elements of object-oriented programming. Elemental patterns are simple patterns that are binary relationships between object-oriented programming elements, and used as comprehensive building blocks for larger architectural patterns.

I do expect there to be an overlap between the two bodies of patterns, because they are not mutually exclusive, but the terminology should remain clear.

2.2 Analysis techniques

Research into increased comprehension of software is not a new area. Likewise, analysis of design or code in order to indicate relationships and interconnection is not unique. Here I present a short overview of such research, and what problems reliance operators solve that each does not.

2.2.1 Cohesion and coupling analysis

To provide a glimpse into the potential ties between reliance theory and the vast amount of literature in coupling and cohesion research, one has only to look at the definitions of the terms:

- Cohesion: "how tightly bound or related its internal elements are to one another." (Yourdon and Constantine, 1979)
- Coupling: "any evidence of a method of one object using methods or instance variable of another object..." (Chidamber and Kemerer, 1991)

Cohesion and coupling are two distinct aspects of the same general view of measuring the interconnections of object-oriented language elements: coupling is concerned with relationships *between* objects, while cohesion is concerned with relationships *within* objects.

ρ-calculus-level analysis provides information at both levels. ρ-calculus can provide a solid theoretical base for coupling and cohesion research in object-oriented systems, which until now has been absent. As an example, one only has to examine the use of the LCOM metric (Chidamber and Kemerer, 1994) with various languages. LCOM must be reformed depending on the particular language features (Hitz and Montazeri, 1996), indicating that there is a lower level of atomic restructuring upon which LCOM could be based.

The literature on coupling, and more-so cohesion, is vast, spanning from procedural languages up through object-oriented languages. The ties to legacy procedural codebases are strong, and this is a well-formed area of research. Being able to form stronger ties between patterns research and this closely linked older body of work would be highly useful. Currently, there is no unifying ideological or practical framework. I expect future close collaboration between this research and aspects of ρ-calculus, discussed in Chapter 4.

2.3 Quantitative metrics

Metrics quantifying attributes of code or designs are highly useful, if properly formed within the bounds of measurement theory. Unfortunately, most are not. Instead, many metrics are what I term 'common-sense' metrics which quantify rather obvious numerical values of code, such as number of lines of code (LOC), the depth of inheritance trees (DIT), number of children classes (subclasses) (NOC), and so on (Chidamber and Kemerer, 1994; Banker et al., 1993; Hitz and Montazeri, 1996). While these measurements can then be used to build more complex analyses, many times they are not, and are instead accepted as fundamental indicators of more complex attributes by less than well informed developers. A prime problem is that these metrics have little or no basis in formal language theory, but instead are derived from production analysis or anecdotal evidence. While these are valid sources for possible research directions, they do not necessarily lead to valid results: "one should avoid collecting data about meaningless aspects of the software document under investigation (just because they happen to be easily collectible)." (Hitz and Montazeri, 1996)

A consequence of this is that, by not basing a metric within a foundational theory, it runs the risk of becoming specific to a language. An example of this was given above with LCOM, one of the most commonly used metrics in practice. I believe that this greatly reduces the credibility of the metric as general to object-oriented languages. It may be that there is, as I propose in the reliance system, a mapping from the language features to the underlying metric computation, but to reform the very equation computing the metric, based on the language, seems to expose a lack of universality.

Producing metrics which relate to pattern-level features of relationships is difficult. With the exception of work by researchers such as Bosch (supporting Design Patterns directly in languages) (Bosch, 1998), Eden (LePUS) (Eden, 2000), and Mikkonen (formalization of patterns through an analysis of the temporal behaviors) (Mikkonen, 1998), there have been few attempts to create formalized notations for the relationships which patterns capture. Because of this, the vast majority of current common metrics

are concerned with, and are only capable of measuring, sub-pattern constructs.

2.4 Formalization of Patterns

While it may seem at first that related work in formalization of patterns is in direct contention with my own proposal, it turns out not to be the case. With the possible exception of Mikkonen's DisCo language, every formalization system I found attempted to formalize the *entirety* of patterns, both the formal and functional descriptions. I believe that this task is not only more difficult because it requires formalization of natural language descriptions, but also, unfortunately, because it is misguided. Research in this area is quite often of high quality, however, and an excellent piece of research in this area is Eden's LePUS.

2.4.1 LePUS

Amnon Eden's Language for Patterns Uniform Specification (LePUS) (Eden, 2000) is an attempt to formalize patterns by creating a hybrid 'compromise' language: "In LePUS we have made a compromise between the technically detailed level of the programming language and the "abstractness" of natural language." (Eden et al., 1999, p 6) This language consists of a graphical language to express relationships within and between patterns and a relational language which is a simplified Higher Order Monadic Logic. Ironically, in the attempt to unify the formalization process with respect to patterns, Eden produced yet again two halves, although they both satisfy only the definition of a formal description. Within the context of the specification of compromise, LePUS is excellent work. I believe, however, that the specification, as defined, is in error.

Let *Patterns* be the unifying language they were originally designed to be by Alexander in *Notes* (Alexander, 1964). The original definition, after all, made a distinction between formal and functional descriptions, and provided for formal and abstract languages for each. An attempt to merge the two into a single formalized language is no more suited to the overall problem, in my opinion, than using only an abstract natural language as the only syntax. There are elements of design which cannot easily be expressed in a highly abstract and ambiguous language, and there are elements which cannot easily be expressed in a highly rigourous and precise language. Forming a compromise between them limits the expressiveness to the intersection of those of the original languages. Using a unifying force such as patterns makes the expressiveness equal to the *union* of the two languages.

Given that difference in ideology, however, Eden produced a straight-forward language which is able

to express much of the relationship information within patterns. LePUS is based on fragments, as proposed by Florijn, *et al.* (Florijn et al., 1997), not the more basic calculi of language theory. By taking this approach, LePUS can discard much of the baggage of programming languages, as I propose is possible with reliance operators, but, unlike reliance operators, that information is lost for all time. No low level analysis can be undertaken within this framework. Again, this was not the focus of Eden's research.

Fragments, or "object oriented building block capturing elements of an object oriented design or program" (van Winsen, 1996) and their relationships allow the expression of patterns as a 'language unto themselves', which coincides with my own thoughts on the high level abstractions possible with patterns. While fragment theory does not disallow 'code fragments' consisting of execution behavior segments, Eden chose to remain above this level, instead concentrating on, and extending, the basic principles to the pattern level through class, object, and hierarchy levels of fragments. As discussed earlier, however, this rejection of the underlying data structures, particularly object fields, makes many metrics impossible to produce, as well as preventing much analysis with respect to cohesion or coupling.

This consequence is unacceptable within the context of maintenance cost reduction and the requirements outlined for a potential solution. It also unfortunately cripples LePUS with respect to certain aspects of programming. Nowhere in LePUS is provision made for the relationships between data members or fields, and other elements. I find it telling that, in the copious literature Eden and his colleagues have produced on LePUS analyzing most of the patterns contained within the *Design Patterns* text, the Singleton pattern is never mentioned, neither as an analysis case, nor a possible problem area for LePUS. This is one of the more simple patterns, so perhaps it was considered beneath the research level necessary for their work, but it is also one of the few which rely completely on access to data members as a primary element.

It may, in fact, be possible to handle data members artificially in LePUS through mocked up accessor functions. Abadi and Cardelli's treatment of ς-calculus does something similar in the earliest stages of development, but it still seems a glaring omission in the LePUS research. There is no provision for later inclusion of lower level elements, as there is in ς-calculus, with λ-calculus.

Fragment theory would seem to be a solid base for this type of research if properly included, but it appears to lack the necessary formalizations for the basic programmatic elements. While fragments are conceptually related in some ways to ς-calculus, they do not have the same ties to the more primitive λ-calculus.

Given the above limitations, it is odd that Eden did not pursue a unification of LePUS with lower

level abstractions such as λ- or ς-calculus. As it turns out, Eden did investigate the use of the second order calculus $F_{<:}$ as a base for his work, but rejected it due to it being the "wrong level of abstraction", "inexpressive due to being only operational semantics", and "hard to prove properties, reason, or implement." (Eden and Hirshfeld, 1999; Eden, 2001)

Each of these rational and valid arguments, however, has an equally rational and valid counterargument. Eden was indeed correct that $F_{<:}$ was the wrong level of abstraction from which to start. $F_{<:}$ is based on λ-calculus, and, therefore, does not have the inherent capability to directly express objects, classes, and so forth. This necessary capability is precisely why ς-calculus was created. ς-calculus alone still suffers from the inability to express beyond its own operational semantics, as does $F_{<:}$, but this is the very limitation reliance operators are designed to alleviate.

$F_{<:}$ is also very difficult to work proofs in directly, much like ς-calculus, which is why I used automated theorem proving techniques in SPQR. This facilitated my research, and allows for easy experimentation by others. When dealing directly with $F_{<:}$, it is also difficult to express higher level concepts. Here is where patterns' functional descriptions come in, as long as they are allowed to exist on their own merit. $F_{<:}$, as with ρ-calculus, is also difficult to reason in directly.

Tools providing a simpler view of the system to developers will be required, but the fact that UML can be used for a less rigorous formal description language removes much of the burden. In contrast, LePUS provides a clean and useful, but entirely nonstandard, graphical language. Finally, LePUS fails to offer a practical toolset for the practitioner, leaving it in the realm of excellent, but non-usable academic theory.

2.4.2 Other approaches

LePUS is not the only approach taken to formalize patterns in a meaningful way. SCRUPLE (Santaun and Prakash, 1994) is an early attempt at searching for patterns in source code. Its main flaw is that it requires a new pattern language representation for each implementation language under consideration. Changing one requires changing the other. Other approaches, some of which I have discussed before, include modifications to the implementation languages, such as LayOM (Bosch, 1996), expression of whole patterns as a subset of fragments (van Winsen, 1996), and enforcement of pattern *behavior* by such systems as CDL (Klarlund et al., 1996).

By this point my opinion regarding the formalization of patterns should be apparent. To summarize, there are two main areas of importance: language-independence, and future work.

Patterns are intended to be implementation-language-independent. It is notable, however, that

most of the research to date to formalize them has resulted in systems which must be fine-tuned for specific programming languages. A strict formalization based on the atomic elements of object-oriented programming is required to allow for true language-independence. This is important not only for broad use with existing languages, but also for extension to future languages as well. We do not know what the exact forms will be, but we can surmise that they will adhere to the same basic principles as today's object-oriented languages, and therefore be expressible within ς-calculus.

This formal approach also allows for future work to be incremental, building upon previous work, particularly if the formal nature is taken advantage of, and automated theorem proving techniques are applied. This will give rise to the possibility of a stable core calculus, which can be extended and *proven* robust and complete in the context of the new rules, which in turn can be easily expressed for the prover system. This also allows for increased flexibility in the interactive speculation of possible extensions.

2.5 Pattern Detection and Validation

Detecting and/or validating pattern instances in source code is the primary focus of my research, and there have been three recent advances in this area that I feel need to be discussed. This will provide an overview of the current state of the art, as well as outlining various problems that these projects do not address.

2.5.1 Pattern Enforcing Compiler

The Pattern Enforcing Compiler project, or PEC (Lovatt et al., 2005) is a compiler replacement that detects conformance of source code written in the Java language (Gosling et al., 2005) to particular design patterns. Source code is analyzed by PEC and cues within the code tell PEC which pattern a class is expected to conform to. If a class passes, the compiler produces the expected bytecode file. If a class fails, an error is reported. It appears that PEC does operate as expected, but it has several drawbacks as well as positive aspects.

First, PEC is Java only. This means that any insights learned from the production of PEC must be re-implemented and, more importantly, be re-conceptualized with other programming languages. Pattern definitions in PEC are written in Java, which is convenient because it means that someone wishing to extend PEC already knows the language. The downside is that this limits PEC further to being a Java only tool.

The PEC pattern definitions and execution mechanism are in may ways similar to unit testing,

allowing for dynamic analysis of the classes at compile time. While I think this is an excellent beginning, as it means that much more complex and free-form analysis can be performed, it has no formal basis. Every pattern definition relies on the behavior of the analysis code, for which there is no automated support to develop. Every definition must be written by hand, and also be manually debugged.

The dynamic analysis outlined above, at least in the current PEC incarnation, also seems to rely heavily on static name checking. For instance, in describing the PEC definition of the Static Factory pattern, the authors state: "...the supplied Static Factory pattern requires a class to have a method called `instanceXXX` or `xxxInstance` that returns an instance of the class or a class derived from the class (where `xxx` is a sequence of zero or more characters or numbers)." This requirement for methods to be named according to informal conventions is unacceptable in my opinion. It can work, in a limited fashion, in specific environments where such conventions are enforced by a central design body, such as with the Java APIs, but it fails to find situations that differ by even a slight amount. If the method in question was named `getInstanceOfClass`, for instance, the check would fail. Also, it fails to take into account cross-cultural naming conventions. A developer in Germany may prefer to use a method name in their own language, for instance, to help improve comprehension.

Definition of patterns in PEC are limited to those patterns that involve exactly one class. Each class can conform to one or more patterns, but patterns involving more than one class are not possible to describe using this methodology. Because of this, PEC currently contains seven pattern definitions, one from the Gang of Four text, Singleton, and six more simplistic patterns from other literature. This is a limited catalog, and omits many of the most valued and widely used patterns.

While PEC does refrain from injecting new keywords into the Java language, instead electing to reuse the `implements` keyword in a unique and clever way, it still requires developers to indicate manually to the compiler which patterns they are attempting to implement. There is no mechanism for pattern detection without explicit engineer tagging of the source code.

While I believe PEC is an interesting blend of static and dynamic analyses, as the authors state, in my opinion it fails to be flexible enough for practical use, is too limited by its Java-only nature, and cannot be extended easily to perform pattern detection without manual tagging.

2.5.2 FUJABA

FUJABA, which stands for From UML to Java And Back Again, is, as the name suggests, a UML and Java integration tool that has a strong interest in the Java community. It is an extensible tool, and one of the plug-in packages, FUJABA RE (Niere et al., 2002; Niere et al., 2003a; Niere et al., 2003b), has

much the same intent as SPQR: to find instances of patterns in source code. The approach taken is decidedly different, however. I will refer to the FUJABA RE package as simply FUJABA for brevity.

The FUJABA tool includes a GUI front-end that provides a diagrammatic approach to reverse engineering source code to UML, and generating code from UML. It is, like PEC, Java specific, although mention is made of the possibility of incorporating other parsers into the RE system. It is unclear to me how that will interact with the rest of the FUJABA toolset, given its focus on Java. Also, as with PEC, it takes a hybrid approach to pattern detection, using what they term a 'bottom-up/top-down' approach. Bottom-up because they start with the source code and identify possible pattern pieces, at which point they flip to top-down to attempt to verify the pieces before performing more code analysis. The core of FUJABA uses a graph-rewriting engine which operates on an abstract syntax graph, or ASG, that represents the source code. The bi-modal approach is required because of this, but in functionality it operates much like a tuned general inference engine, selecting clauses for consideration in order of relevance.

This is a highly efficient approach, but it comes with a lack of precision. As the authors state (emphasis mine): "instead of a large number of 100% precise descriptions of *each possible implementation variant for a certain pattern*, we use a small number of somewhat imprecise detection rules." (Niere et al., 2002) The FUJABA design requires that code features be found explicitly and directly. If the code varies from the detection rules too widely, even if it may conform conceptually to the pattern, it will fail to find it. This is a common problem in pattern detection tools, dealing with the large number of conceptually conformant but structurally altered variants of a pattern that may appear in an implementation. FUJABA has taken an interesting approach to solving this, by not requiring strict precision or accuracy.

Much of this is mitigated by a fuzzy logic inference system, that attaches weights to signify the confidence in each match. Unfortunately, the initial thresholds are manually set by the engineer in a training phase for a particular application or domain, and then further tuned during the search. This does allow the engineer to fine-tune the search system for specific needs, but I believe that this is unreasonable for large or diverse systems, particularly those that involve code from several sources where styles may differ. The authors mention this weakness briefly, stating that the tuning is "a tedious task requiring a lot of trial and error" (Niere et al., 2002). The end result is a tool that is fast in its internal mechanisms, and fairly precise, but requires significant intervention by a trained engineer to produce results with a high level of confidence.

While FUJABA is an excellent example of the application of imprecise rules in the face of a lack

of conceptual formalism, it is my opinion that it is not precise enough to be a useful general-purpose automated support tool. The system relies on an engineer to step in and take the balance of the conceptual load to bridge the gap between flexibility and automation. Future implementations may reduce this need through training of the system, but for now it necessitates much guiding by the user.

2.5.3 RML and CrocoPat

Unlike the above two approaches, the Relation Manipulation Language, RML (Beyer et al., 2005), is highly precise. It is designed to manipulate and make queries on relations within a graph. It is based on predicate calculus, and offers support for n-arity tuples of generalized relations. It also offers basic floating-point numeric calculation, and the fundamentals of imperative programming through variables, assignment, and control statements. Finally, it offers a transitive closure operator for simple set generation conforming to a given condition. This combination provides a strong support for the logical inferences required to find design patterns, as well as basic abilities to compute metrics, all within one language.

CrocoPat is an intepreter for RML (Beyer et al., 2005), and was designed for maximum efficiency. It uses a binary decision diagram internally for effective manipulation of the relation graph. The performance of CrocoPat on simple validations is quite good, but it is unclear how it scales to more complex searches on large systems.

RML is a highly expressive and solid formalism for mapping free-form relations between elements, and looks to be a highly efficient tool when paired with CrocoPat. While the paper referenced specifically states that detection of design pattern instances is possible, and provides a simple example using the Composite pattern from the Gang of Four text, it does not offer a methodology for doing so, nor does it offer any guidance on how to express more complicated patterns that may rely on behavioral aspects of the implementation. Indeed, it appears to only provide a very simple relational notation for analyzing object-oriented relations, comprised of `Inherits`, `Call`, and `Contain` relations, indicating subtyping inheritance, method and field definitions, respectively. While this is sufficient for some structural patterns, such as Composite, it is not clear that it will suffice for the more abstract patterns that are defined by their behavior, such as Chain of Responsibility or Strategy. It certainly has not been outlined in the publications on RML to date.

Chapter 3

Sigma Calculus

Software has historically been rooted firmly in formal notations. Formal descriptions of software most decidedly lend themselves to a pattern's formal description using a formal notation. The entire pattern does not need to be given a formal form, nor would it be improved by doing so. The formal descriptions, however, should be as formal as possible without losing the generality that makes patterns useful. Source code is, at its root, a mathematical symbolic language with well-formed reduction rules. We should strive to find an analogue for the formal side of patterns.

The question then arises as to how formal we can get with such an approach. A full, rigid formalization of static objects, methods, and fields would only be another form of source code, invariant under some transformation from the actual implementation. This defeats the purpose of patterns. We must find another aspect of patterns to encode as well, in order to preserve the flexibility of patterns.

An analysis of desired traits for an intermediate formalization language includes that it be mathematically sound, consist of simple reduction rules, have enough expressive power to directly encode object-oriented concepts, and have the ability to flexibly encode relationships between code constructs.

Given these constraints, there are few options. The most obvious possible solution is lambda calculus, written as λ-calculus, or one of its variants (Stansifer, 1995), but λ-calculus cannot directly encode object-oriented constructs. Various extensions which would enable λ-calculus to do so have been proposed, but they invariably produce a highly cumbersome and complex rule set in an attempt to bypass apparently fundamental problems with expressing typed objects with a typed functional calculus (Abadi and Cardelli, 1996). One final candidate, sigma calculus, written as ς-calculus, meets this requirement easily.

ς-calculus and its descendants are a fresh approach to creating a denotational semantics for object-oriented languages, and are "the first that does not require explicit reference to functions or procedures."

(Abadi and Cardelli, 1996) Defined and described in *A Theory of Objects* by Martín Abadi and Luca Cardelli, ς-calculus is an analogue to λ-calculus used in procedural theory. It concentrates on the aspects of object-oriented programming which are distinct from those of procedural programming, and makes no attempt to duplicate the efforts of the λ-calculus literature. Instead, it defines a notation providing a rigorous mathematical foundation for further object-oriented language theory. The prime elements are objects, types, methods and fields, the last two of which are treated as equivalent in ς-calculus, leading to some rather elegant solutions to some of the complex problems raised in formalizing highly dynamic languages. Classes are treated as a special case of objects, further simplifying the system.

ς-calculus is not an extension of λ-calculus. Attempts to produce such a hybrid have been made, but none has been particularly successful. A prime motivation for working to graft OO technologies onto λ-calculus is a desire to leverage off of the extremely large body of well done literature in that area. By starting anew, Abadi and Cardelli at first glance seem to have disposed of that body of work. On the contrary, they correctly recognize that the entirety of λ-calculus can be subsumed within the method calls of OOP. They even provide a mapping from λ-calculus to ς-calculus, resulting in "a simple and direct reduction semantics, instead of an indirect semantics involving both λ-abstraction and application." (Abadi and Cardelli, 1996, p. 66)

ς-calculus also scales well from pure theory research to practical applications of metric measurement not easily done otherwise, most likely with some simplifications to the calculus to more easily produce support tools. For instance, many metrics require reformation of their base assumptions when moved from one language to another (Basili et al., 1996; Chidamber and Kemerer, 1994). The highly theoretical nature of ς-calculus, and its ability to express current OO languages, means that a metric designed at this level of analysis will hold true for any language that can be mapped to ς-calculus.

3.1 Basics of ς-calculus

I will only describe the fundamentals of ς-calculus in this document, and it is suggested that the reader refer to (Abadi and Cardelli, 1996) for the formal details. In many ways, ς-calculus reads similarly to a pseudo-code notation, and should be at least passingly familiar to most practitioners of OO languages.

ς-calculus is broken up into twenty-four *fragments*, each a collection of equations concerning a particular concept of the calculus. In all, there are seventy-eight reduction rules in ς-calculus. The various formal languages described in (Abadi and Cardelli, 1996) are produced by mixing and matching these fragments to match certain desired language features. A very simple first-order calculus, such as O-1

(Abadi and Cardelli, 1996, pgs.153-165) will use a small number of these fragments as its foundation, while a more expressive and complex language such as O-3 (Abadi and Cardelli, 1996, pgs.305-324) will use many more.

3.1.1 Objects, method, fields and types

Everything in ς-calculus can be reduced to four elements of programming: objects, methods, fields, and types. There are no classes, no templates, no arrays or generalized containers, yet all of these, and much more, can be quickly constructed from the four basic elements.

Methods and fields are treated as equals in ς-calculus, and no attempt is made to distinguish between them: "a field can be viewed as a method that does not use its self parameter." (Abadi and Cardelli, 1996, pg. 52). Retrieving the value of a field is equivalent in this case to calling a method that returns a private data object. Considering the prevalence of precisely this idiom in object-oriented programming, this seems a reasonable approach. It does, however, bring up an interesting side-effect: if fields are directly assignable, then so are methods. While this may seem to be an oddity to the user of C++, Java, or other mainstream languages, other languages such as Self (Agesen et al., 1993) expressly allow it. ς-calculus therefore provides the most generalized and expressive approach, allowing for a broader range of languages that can be quickly converted to ς-calculus notation.

Method bodies are bound to instances via the sigma-binding, written as ς. This is the source of the name ς-calculus. $\varsigma(x_i : A)b_i\{x_i\}$ takes a parameter x_i, and binds it to a free variable in b_i. This parameter is the *self parameter* and binds the method body b_i to a particular instance object of type A.

Types are defined by listing explicitly the elements of the type, methods and fields, and then providing a type for each. Method types are the return value type, or the type of the reduction of the method to a single product.

Objects are similarly defined, by describing each element in turn, but giving a proper binding of Self to the methods and fields such that they are associated with a particular instantiation.

For example, a type A and an object a may be defined as in Equations 3.1 and 3.2. By convention in ς-calculus, l stands for any element (method or field) of the object or type in question, capitalized Roman letters stand for types, and b stands for element definitions, usually a method body. For instance, Equation 3.1 states that there are n elements of type A, l_1 through l_n. These elements have types B_1 through B_n, respectively. Equation 3.2 states that object a has n elements denoted as in A, and that each has a corresponding definition indicated by b_1 through b_n. We can say in such a case that a has type A, or $a : A$ in ς-calculus notation. In these equations then, l_i is of type B_i, and x_i is of type A.

$\triangleq$ in ς-calculus represents "equal by definition", distinguished from the $\equiv$ for "syntactically identical" (Abadi and Cardelli, 1996, pg. 59) and $=$ for "informally equal" (Abadi and Cardelli, 1996, pg.66).

$$A \equiv [l_i : {B_i}^{i \in 1..n}] \tag{3.1}$$

$$a \equiv [l_i = \varsigma(x_i : A) b_i\{x_i\}^{i \in 1..n}] \tag{3.2}$$

3.1.2 Selection and update

Selection is the operation that allows the naming of a subelement of an object or type. If object a has an element l_2, then $a.l_2$ names that element explicitly. This is usually read as "a dot l_2", and corresponds to the dot notation of many object-oriented languages, such as C++ and Java. It provides much the same function here. Selection of a method invokes that method, and selection of a field retrieves the object stored in that field. Invocation of a method returns an object (or none), and selection of a field always returns an object. In both cases the type of the selection operation is the type of the object returned.

Assignment of values to fields, or method bodies to methods, is performed with the update operator, $\Leftarrow$. As noted above, methods *can* be updated in ς-calculus, during which they are given a new method definition, or even converted to what would generally be considered a field.

3.1.3 Reduction rules

Reduction rules are the core of ς-calculus's fragments. They stipulate a set of criteria and premises, and an inference or conclusion. In these rules, E is understood to mean 'the environment', a shorthand for everything currently defined and under consideration. A 'judgment' is a singular criteria or conclusion, and is written $X \vdash Y$ for some X context, and some Y criteria. $E \vdash a : A$ therefore encodes "an environment E such that there is an object a of type A".

We can take these judgment definitions, Equations 3.1, 3.2 and the selection operator, and define a reduction rule for evaluation of a selection in a first-order calculus (Abadi and Cardelli, 1996, pg.90):

(Eval Select)

$$\frac{E \vdash A \equiv [l_i : {B_i}^{i \in 1..n}],\ E \vdash a \equiv [l_i = \varsigma(x_i : A) b_i\{x_i\}^{i \in 1..n}],\ E \vdash a : A \quad j \in 1..n}{E \vdash a.l_j \hookrightarrow b_j\{\!\{a\}\!\} : B_j}$$

Simply put, this states that given a type A, and an object a so defined as above, that the type of the evaluation of the selection of an element in a will reduce to that element's proper type B_j. All of ς-calculus is defined in such rules of reduction. If a reduction path exists between a premise p and a

conclusion c, it can be written $p \rightsquigarrow c$.

3.1.4 Classes and other constructs

As stated above, ς-calculus does not offer first-class support for language features other than the basic four of objects, methods, fields, and types. These can be emulated using only these for elements, however, as can be shown with the concept of classes.

ς-calculus defines a mapping of classes to objects as stated in A Theory of Objects [pg. 100]:

> IF $A \equiv [l_i : B_i^{i\in 1..n}]$ is an object type, then:
> $Class(A) \triangleq [new : A, l_i : A \rightarrow B_i^{i\in 1..n}]$
> is the type of classes generating obejcts of type A. These classes have the form:
> $[new = \varsigma(z : Class(A))[l_i = \varsigma(s : A)z.l_i(s)^{i\in 1..n}]$,
> $l_i = \lambda(s : A)b_i^{i\in 1..n}]$

This allows an object $c : Class(A)$ to be described as in the latter definition, whose method *new*, when invoked, returns an object of type A. In class-based languages, a class, A, is generally thought to be equivalent to the *type* A, and most often the terminology reflects this assumed equality. In ς-calculus, however, classes and types are distinct entities, requiring a conversion of classes to a pure object notation.

3.2 Inflexibility of ς-calculus

While ς-calculus is a rich and important work in formalization of object-oriented languages, it does not meet our needs for formalization of design patterns. As an example, I have translated the Singleton pattern, from Gamma, *et al.* (Gamma et al., 1995, pg 127-134), to ς-calculus. It has a combination of interesting static and dynamic behaviors, but is sufficiently simple to be presented here. Commonly, in design, it is desirable to have a class produce one and only one object of itself in a system. An example of this would be a memory manager class, which must have complete control of the memory space it is given command of. There is a need, however, for global access to that memory manager, so creating an instance and passing it around repeatedly would be problematic. This is a very simple design pattern which is almost completely syntactic in form, yet is highly convoluted in ς-calculus, as shown in Figure 3.1.

As can be seen, this is a highly complex description for such a simple pattern. Not only is it extremely unwieldly, but it also suffers from a complete rigidity of form, and does not offer any room for interpretation of the implementation description, or any necessary fungibility that may be required for

$$
\begin{aligned}
singleton \quad &\triangleq \quad \left[new = \varsigma(z)\left[l_i = \varsigma(s)z.l_i(s)^{i\in 1..n}\right],\right. \\
&\quad\; l_{1..n} = \varsigma(z)\lambda(s)b_i^{\,i\in 1..n}, \\
&\quad\; uniqueInstance = \varsigma(z)\lambda(s)nil, \\
&\quad\; getInstance = \varsigma(z)\lambda(s) \\
&\qquad\qquad s.uniqueInstance \rightarrow s.uniqueInstance; \\
&\qquad\qquad \left. s.uniqueInstance \Leftarrow s.new\ \right] \\
Singleton \quad &\triangleq \quad [\ uniqueInstance : Singleton, \\
&\quad\; getInstance : Singleton\ ] \\
Class(Singleton)_{Ins,Sub} \quad &\triangleq \quad \left[new : \left[l_i : B_i^{\,i\in 1..n}\right],\right. \\
&\quad\; uniqueInstance : Singleton \rightarrow Singleton, \\
&\quad\; \left. getInstance : Singleton \rightarrow Singleton\ \right] \\
class\left(Singleton, a_i^{\,i\in I}\right)_{Ins} \quad &\triangleq \quad [new = \varsigma(z : Class(Singleton)_{Ins,I}) \\
&\quad\; \left[l_i := \varsigma(s : Singleton)z.l_i(s)^{i\in I}\right], \\
&\quad\; uniqueInstance = \varsigma(z)\lambda(s)nil, \\
&\quad\; getInstance = \varsigma(z)\lambda(s) \\
&\qquad\qquad s.uniqueInstance \rightarrow s.uniqueInstance; \\
&\qquad\qquad s.uniqueInstance \Leftarrow s.new\ ]
\end{aligned}
$$

Figure 3.1: Singleton as expressed in ς-calculus

a specific application. This lack of adaptiveness means that there would be an explosion of definitions for just the Singleton pattern, each of which conformed to a single particular implementation. This breaks the distinction that patterns are implementation independent descriptions, as well as creating an excessively large library of possible pattern forms to search for in source code.

ς-calculus is not a particularly easy system or notation to learn. It is highly complex in abstractions, if not in implementation. The notation is cumbersome, and not intuitive without lengthy study. To make matters worse, the definitive text on the subject is inscrutable, ambiguous, and difficult to read. Consequently, there are few practitioners of the ς-calculus at this time.

In addition to the practical concerns, ς-calculus fails the fourth requirement I set forth for the intermediate language: relationship encoding. The ς-calculus suite does not directly support this requirement.

Chapter 4

Rho Calculus

While ς-calculus provides a solid basis for formalizing object-oriented concepts, it does not include any notion of conceptually tying two elements together other than object definition semantics. While method calls and field accesses are incorporated via the selection operator, it is difficult to ascertain the relationship between two entities without a full scan of the system at hand. These relationships of reliance are the core of the conceptually based analysis of SPQR, and I provide a formalism here for describing and manipulating them. This formalism is defined in the manner of ς-calculus's *fragments*, and is described as the *rho fragment*, or Δ_ρ, with rho for 'reliance'. A detailed treatise on Δ_ρ is the subject of this chapter, however the complete Δ_ρ definition with fully expanded definitions can be found in Appendix A for quick reference. ρ-calculus is defined as ς-calculus with the addition of Δ_ρ.

I term the relationship-defining formalizations *reliance operators*. The term 'relationship' is already heavily used in the literature in various ways, and 'dependency' has a strong heritage as well, but the core concept is similar: one element *relies on* another. Essentially, ρ-calculus can be seen as a single unifying formalism for the various method and data dependency approaches that have been produced. It does this by bridging the ς-calculus and λ-calculus worlds in a well formed but precisely restricted manner. ς-calculus concerns itself only with the formalisms of objects and their typing interactions. As soon as the boundary of a method body is reached, it defers to λ-calculus for further formalism. The definitions of ρ-calculus extend ς-calculus into the method bodies, but in only a few and discrete ways that do not break the fundamentals of either established formalism. This small number of rules, however, leads to a rich environment of information on which many conceptual inferences can be based.

As stated in Chapter 3, ς-calculus reduces the entirety of object-oriented programming to four elements: object, methods, fields, and types. If ρ-calculus is to successfully model reliances, it needs to provide conceptual reliances between these four elements.

Types that rely on one another for their definitions are already handled in the ς-calculus subtyping construct <:. The equation $A <: B$ can be considered to be read as 'the type A relies on the type B for its base definition'. Type-to-type relationships are, therefore, already a part of ς-calculus, leaving three elements to be handled: object, methods and fields.

Objects and fields can be reduced to one conceptual element. Any field is first and foremost an object. Its enclosing object is a scoping, but does not alter the semantics of the field object, nor its type. I will use the term 'field' from this point on to refer to objects in reliance relationships for simplification. Raw objects can be considered as fields of an enclosing artificial 'systemic' object conceptually, but in practice there are merely unscoped.

The remaining two elements, methods and fields, give rise to four combinations of reliance relationships: method/method, method/field, field/method, and field/field. These provide a conceptual coverage of the reliances in object-oriented programming, as they derive directly from the core elements of ς-calculus. If there are missing reliances, then either ς-calculus is missing elements in its definition, or Δ_ρ is conceptually incomplete. I assume the validity of ς-calculus as an axiom in this work, and will show in Section 4.7 that the remaining reliance permutations involving methods or fields and types are inconsistent with ς-calculus.

4.1 Notation Conventions

In the following discussion, I will use a few standard notations to clarify the content. $\mathcal{O}$ and $\mathcal{P}$ stand for generalized object abstractions in ς-calculus. m and n are placeholders for any method, while f and g mean any given field. s and t can stand for either a method or a field, ie any selected element. $\mathcal{O}.m$ is therefore to be read as "any method m of object $\mathcal{O}$," $\mathcal{P}.f$ is "any field of object $\mathcal{P}$" and $\mathcal{O}.s$ is "any selectable element of object $\mathcal{O}$." Abadi and Cardelli use l ubiquitously to indicate any selectable item, and rarely distinguish between methods and fields. This is appropriate for their foundational work, but a bit finer granularity is needed here. In general, $\mathcal{O}, m, f$ and s are used on the left-hand side of relationships, and $\mathcal{P}, n, g$, and t on the right-hand side. Additional objects, methods and fields will be indicated by use of one or more tick marks. $\mathcal{O}, \mathcal{O}', \mathcal{O}''$ are to be considered as three distinct and unique objects when seen in the same equation unless otherwise indicated. To conform with the formal notation in (Abadi and Cardelli, 1996), indices and integer ranges may contain an m or an n. It will be clear from context when this is the case, as opposed to representing a method.

For the purposes of clarity in notation, I define a few helper operators. $\mathbf{meth}(\mathcal{O})$ is the set of methods

defined in an object $\mathcal{O}$. Similarly, $\mathbf{field}(\mathcal{O})$ is the set of fields defined in object $\mathcal{O}$. $\mathbf{def}(\mathcal{O})$ is the union of these two sets. This becomes useful when required to assert that some selectable subelement s is defined in an object $\mathcal{O}$: $s \in \mathbf{def}(\mathcal{O})$. These are formally expressed in Equations 4.1 through 4.3.

$$\textbf{Methods Of} \qquad \frac{E \vdash \mathcal{O},\ \mathcal{O} \equiv [l_i \Leftarrow {b_i}^{i \in 1 \ldots n}, {l_j}^{j \in n+1 \ldots m}]}{\mathbf{meth}(\mathcal{O}) = \{{l_i}^{i \in 1 \ldots n}\}} \tag{4.1}$$

$$\textbf{Fields Of} \qquad \frac{E \vdash \mathcal{O},\ \mathcal{O} \equiv [l_i \Leftarrow {b_i}^{i \in 1 \ldots n}, {l_j}^{j \in n+1 \ldots m}]}{\mathbf{field}(\mathcal{O}) = \{{l_i}^{i \in n+1 \ldots m}\}} \tag{4.2}$$

$$\textbf{Definitions Of} \qquad \frac{E \vdash \mathcal{O},\ \mathcal{O} \equiv [{l_i}^{i \in 1 \ldots n}]}{\mathbf{def}(\mathcal{O}) = \{{l_i}^{i \in 1 \ldots\ n}\}} \tag{4.3}$$

Given the above definitions, the following discussions and formalizations will assume the following:

$$\begin{array}{lll} m \in \mathbf{meth}(\mathcal{O}) & f \in \mathbf{field}(\mathcal{O}) & s \in \mathbf{def}(\mathcal{O}) \\ n \in \mathbf{meth}(\mathcal{P}) & g \in \mathbf{field}(\mathcal{P}) & t \in \mathbf{def}(\mathcal{P}) \\ m' \in \mathbf{meth}(\mathcal{O}') & f' \in \mathbf{field}(\mathcal{O}') & s' \in \mathbf{def}(\mathcal{O}') \\ & \vdots & \end{array}$$

I follow ς-calculus's subtyping operator for the notation for Δ_ρ: Given that the colon ':' is used as a typing operator, and that $<:$ can be read as 'relies on the type of', I use the $<$ symbol to indicate reliance in general. Reliance forms are indicated by appending an appropriate mnemonic symbol to this base.

Additionally, I introduce the concept of *method-body context*. This becomes a key point of joining of ς-calculus and λ-calculus in the analyses that ρ-calculus was designed to facilitate. Informally, context is the method body within which a selection or update occurs. Formally, attempting to denote even something simple such as "there is an update of field f in method m of object $\mathcal{O}$" entails giving an extended definition of $\mathcal{O}$, and at least an illustrative partial definition of m to indicate the use of the update: $E \vdash \mathcal{O}, \mathcal{P},\ \mathcal{O} \equiv [l_i = {b_i}^{i \in 1..n},\ b_j = \{\ldots, \mathcal{O}.f \Leftarrow \mathcal{P}.s, \ldots\}]$. This quickly becomes tedious. Instead, the context indicator is used to illustrate that a relationship, update, or selection occurs within a specific method body, ie, within a specific context, as in Equation 4.4. $\mathcal{S}$ stands for any relationship or valid statement that can occur in the method body. The double bars, $\|\|$, are the delimiter pair, the contents of the scoping are contained between them, and the method within which $\mathcal{S}$ occurs is given as

a subscript to the delimiter pair.

$$\textbf{Context} \qquad \frac{E \vdash \mathcal{O},\ \mathcal{O} \equiv [l_i = {b_i}^{i \in 1..n},\ b_j = \{\ldots, \mathcal{S}, \ldots\}]}{\|\mathcal{S}\|_{\mathcal{O}.l_j}} \tag{4.4}$$

Finally, the symbols $\mathcal{X}$ and $\mathcal{Y}$ will represent any valid operand for the left-hand side and right-hand side of the reliance operators, respectively, as in $\mathcal{X} < \mathcal{Y}$. These may be selector statements or simple raw objects.

4.2 Direct Definitions

Each of the reliance operators can be defined in terms of pure ς-calculus constructs, or in terms of each other through relationships akin to transitivities. The former are those that are directly findable in source code, so I will begin with those. In the next section, I will discuss the various interrelationships that can arise through inference. In every case, the reliance operators are defined as judgments, not as equivalences. While the ς-calculus constructs are valid requirements for each conclusion, the conclusion reliance operator does not necessarily imply the presence of the constructs.

In the following discussions, I will group the reliance operators by their left-hand side elements. This allows discussion of the operators in terms of the ς-calculus constructs that lead to their production, which, in turn, provides valuable insights into how best to implement them in analysis tools. Grouping the operators by their right-hand side elements results in little added insight.

The groups are the method-body-based reliance operators, $<_\mu$and $<_\phi$, and the update-target-based reliance operators, $<_\sigma$and $<_\kappa$.

4.2.1 Method-body-based reliance operators: $<_\mu$, $<_\phi$

Method-body-based reliance operators are so named because they tie how a method body relies on either other methods, or fields, to complete its task. The left-hand side of the reliance operator is the method that forms the base of the reliance.

The right-hand side of the reliance operator is either a selected method or a selected field (including raw objects). A selected method on the right-hand side defines a $<_\mu$ (or *mu-form*) reliance operator, while a selected field on the right-hand side defines a $<_\phi$ (or *phi-form*) reliance operator. μ mnemonically stands for *method* while ϕ stands for *field*.

For example, consider the pseudo-code in Figure 4.1. There are two objects defined: `G` is a global

```
object G {}
object O {
        method m() {
                field retval;
                P.n(G);                 // (1)
                retval = P.n2();        // (2)
                G = retval;             // (3)
                return retval;          // (4)
        }
}
```

Figure 4.1: Pseudo-code for $<_\mu$ and $<_\phi$ example

object that is currently blank; `O` has a single method that utilizes a second object `P`, whose definition is not currently available. Lines 1-4 will illustrate several reliance styles.

Starting with line (1), `O.m` obviously relies on the call to `P.n`, as `P.n` presumably performs some amount of work that `O.m` wishes to have done. This would be written in ρ-calculus as $\mathcal{O}.m <_\mu \mathcal{P}.n$. `O.m` uses the object `G` as a parameter to be passed to `P.n`. This creates a $<_\phi$ reliance between them, $O.m <_\phi G$, as it is obvious that `O.m` expects that `G` will be required by `P.n` for some reason. It is useful to think of $<_\mu$ as 'calls' and $<_\phi$ as 'uses' in such direct cases: "method m of object O calls method n of object P" and "method m of object O uses object G". These are more commonly stated as "O dot m calls P dot n" and "O dot m uses G", mirroring not only common terminology in programming environments, but also the ρ-calculus notation. These two reliances can be formally defined as in Equations 4.5 and 4.6.

$$\textbf{Mu Call} \qquad \frac{E \vdash \mathcal{O}, \mathcal{P},\ \|\mathcal{P}.n()\|_{\mathcal{O}.m}}{E \vdash \mathcal{O}.m <_\mu \mathcal{P}.n} \tag{4.5}$$

$$\textbf{Phi Parameter} \qquad \frac{E \vdash \mathcal{O},\ \|\mathcal{P}.n(\mathcal{P}'.g')\|_{\mathcal{O}.m}}{E \vdash \mathcal{O}.m <_\phi \mathcal{P}'.g'} \tag{4.6}$$

The two update statements can be dissected into their component parts, and reliances formed between them and the enclosing method body. In line (2), `retval = P.n2()` we see that `retval`, which I have already established is required by `O.m`, in turn gets its value from `P.n2`. It is therefore reasonable to assert that `O.m` relies on `P.n2`. Since `P.n2` is a method, this is a mu reliance, and $O.m <_\mu P.n2$. Similarly, from update statement in line (3), `G = retval`, we can take the right-hand side of that update operator, `retval`, and make a phi reliance: $O.m <_\phi O.m.retval$. Notice the fully qualified name for

retval. Formalizing these results in Equations 4.7 and 4.8.

$$\textbf{Mu Update} \qquad \frac{E \vdash \mathcal{O}, \mathcal{O}', \mathcal{P},\ \|\mathcal{O}'.f \Leftarrow \mathcal{P}.n\|_{\mathcal{O}.m}}{E \vdash \mathcal{O}.m <_{\mu} \mathcal{P}.n} \tag{4.7}$$

$$\textbf{Phi Update1} \qquad \frac{E \vdash \mathcal{O}, \mathcal{X}, \mathcal{Y},\ \|\mathcal{X} \Leftarrow \mathcal{Y}\|_{\mathcal{O}.m}}{E \vdash \mathcal{O}.m <_{\phi} \mathcal{Y}} \tag{4.8}$$

The target of an update is also a source of reliance for the enclosing method body. In both update statements, we can create a phi reliance with the left-hand side. From this we can derive both $O.m <_{\phi} G$ and $O.m <_{\phi} O.m.retval$, and formalize the concept in Equation 4.9.

$$\textbf{Phi Update2} \qquad \frac{E \vdash \mathcal{O}, \mathcal{X}, \mathcal{Y},\ \|\mathcal{X} \Leftarrow \mathcal{Y}\|_{\mathcal{O}.m}}{E \vdash \mathcal{O}.m <_{\phi} \mathcal{X}} \tag{4.9}$$

Finally, in line (4) `O.m` relies on the local variable `retval` to complete its task, as the return value. This is written in ρ-calculus as $\mathcal{O}.m <_{\phi} \mathcal{O}.m.retval$. Equation 4.10 gives a formalization for this, and uses the weak reduction of ς-calculus (Abadi and Cardelli, 1996, pg. 64), to define the return value.

$$\textbf{Phi Return} \qquad \frac{E \vdash \mathcal{O},\ m \in \textbf{meth}(\mathcal{O}),\ \mathcal{O}.m \rightsquigarrow \mathcal{P}.f}{E \vdash \mathcal{O}.m <_{\phi} \mathcal{P}.f} \tag{4.10}$$

The above definitions of $<_{\phi}$ look overly complex, being based on the use of fields within the method. An alternate approach would be to define $<_{\phi}$ on the existence of the definition of a field, but this has the side-effect of potentially including unused fields in the reliance graphs. Section 4.3 will introduce the necessary formalisms to show how the proper reliances can be built up through inference such that only the fields relevant to the operation of the method are relied upon, and unused fields are ignored.

4.2.2 Update-target-based reliance operators: $<_{\sigma}$, $<_{\kappa}$

The other group of reliance operators are the update-target-based operators. These have direct definitions that use the update operator of ς-calculus, and the left-hand side of the update is the left-hand side of the appropriate reliance operator. Under the most basic assumptions of programming, these could also be called field-based operators, but any ς-calculus construct that is valid on the left-hand side of an update operator could theoretically be the basis for such a reliance. While methods are traditionally not allowed to be updated in most practical industry programming languages, ς-calculus specifically provides such a feature in the O-1, O-2, and O-3 languages (Abadi and Cardelli, 1996)[pgs. 153, 275,

307].

Consider that an object at any point in time has a state, the set of the various fields contained within that object. This state is mutable by changing either the entirety of the state at one time, by replacing the object with a new 'value', or by altering individual fields of the object on a per item basis. Either results in a modified state and a new 'value'. Whenever the state of an object changes, there is an opportunity to create a reliance to reflect how that state was altered, where, and by what.

The first option is the simplest to capture, it is simply an update with a field on the left-hand side. The vast majority of the time, the right-hand side of the update will be a method call, returning an object that is used as the new field state. This is the most primitive form of the $<_\sigma$, or *sigma-form*, reliance operator, sigma for *state change*. The sigma in this case may seem like an odd choice given that the basis of this theory is the sigma calculus, but Abadi and Cardelli chose the lesser used variant sigma form (ς) to forestall such problems. Referring back to Figure 4.1, the fact $O.m.retval <_\sigma P.n$ can be created from line 2, as per Equation 4.11. "Object retval, local to method m of object O, relies on method n of object P for its state."

A closely related reliance operator is the most basic form of the $<_\kappa$, or *kappa-form*, reliance, with kappa for *cohesion*. This creates a link between two fields using a field-to-field update, as in line 3 of Figure 4.1, where the example code would result in $G <_\kappa O.m.retval$. This is a less common case than the $<_\sigma$ form of update, but is shown in Equation 4.12.

Sigma Update

$$\frac{E \vdash \mathcal{O}, \mathcal{O}', \mathcal{P},\ \|\mathcal{O}'.f' \Leftarrow \mathcal{P}.n()\|_{\mathcal{O}.m}}{E \vdash \|\mathcal{O}'.f' <_\sigma \mathcal{P}.n\|_{\mathcal{O}.m}} \tag{4.11}$$

Kappa Update

$$\frac{E \vdash \mathcal{O}, \mathcal{O}', \mathcal{P},\ \|\mathcal{O}'.f' \Leftarrow \mathcal{P}.g\|_{\mathcal{O}.m}}{E \vdash \|\mathcal{O}'.f' <_\kappa \mathcal{P}.g\|_{\mathcal{O}.m}} \tag{4.12}$$

There are three final direct definitions of reliance operators to discuss that all involve objects crossing the boundary of a method call. In doing so, there are two mechanisms for objects to do so: as input parameters, or as return values. Both input parameters and return values can be viewed as call-by-value or call-by-name (Stansifer, 1995). For objects, call-by-name is a reference mapping using references to the same object both outside and inside the method, while call-by-value creates a copy of the object under consideration.

ς-calculus does not offer strong direct support for call-by-value (Abadi and Cardelli, 1996, pg 65), but it does offer several constructs that, when applied together, do offer an equivalent behavior. The

default for ς-calculus, both as input parameters and results, is call-by-name. A call-by-value can be emulated by enforcing a cloning (Abadi and Cardelli, 1996, pgs 36, 130) of the input parameter just prior to normal method invocation. This creates a new, fresh object that is equivalent to the original, and prevents any subsequent manipulations from affecting the original object. Likewise, a call-by-value result can be emulated by creating a clone just prior to returning from the method, and using that fresh object.

The standard substitution operation $b\{\!\{x \leftarrow c\}\!\}$ from ς-calculus indicates that there is a mapping from the parameter name c to the local name x within method body b, as per *A Theory of Objects* (Abadi and Cardelli, 1996, pg 58). I modify this to $b\{\!\{x \overset{v}{\leftarrow} c\}\!\}$ to indicate a call-by-value variant as defined in Equation 4.13. Even though the default behavior for ς-calculus is a call-by-name semantics, I will use the notation $b\{\!\{x \overset{n}{\leftarrow} c\}\!\}$ when call-by-name is specifically required, and leave the standard notation to stand for either of the call-by-name or call-by-value variants. A similar construct can be created to allow for call-by-value method results, as in Equation 4.14.

$$\frac{c' \Leftarrow clone(c),\ b\{\!\{x \leftarrow c'\}\!\}}{b\{\!\{x \overset{v}{\leftarrow} c\}\!\}} \qquad (4.13) \qquad \frac{\|...c' \Leftarrow clone(c)\|_b,\ b \rightsquigarrow c'}{b \overset{v}{\rightsquigarrow} c} \qquad (4.14)$$

Other calling styles, such as default input parameter values (Abadi and Cardelli, 1996, pg 67), and call-by-keyword (Abadi and Cardelli, 1996, pg 68) are also handled within the ς-calculus. This keyword-based parameter passing will become important when I introduce the method calling mechanism of POML in Section 7.1.1. As a brief note, however, a robust and flexible input parameter system can be achieved by assuming a call-by-keyword system that uses call-by-name. Orthogonally adding the cloning mechanism to support call-by-value provides a single approach that can map to a number of calling styles from assorted programming languages.

The Kappa Input Parameter rule, in Equation 4.15, is classified as update-target-related because the procedure of mapping to keywords creates a relationship between the external and internal variables that internally utilizes an update operator, as per Abadi and Cardelli (Abadi and Cardelli, 1996, pgs 58, 68). Additionally, the cloning mechanism for call-by-value semantics uses an update operator to assign the newly created object to a local name, as above. This rule simply states that internal variables rely on the parameter values being passed in via call-by-name semantics.

Mirroring this is the Kappa Return judgment in Equation 4.16, which creates a reliance between the

target of an update, and the return value from within the method used as the update source.

Kappa Input Parameter

$$\frac{E \vdash \mathcal{O}, \mathcal{P},\ \mathcal{O}.m\{\!\{f \overset{n}{\Leftarrow} \mathcal{P}.g\}\!\}}{E \vdash \|\mathcal{O}.m.f <_{\kappa} \mathcal{P}.g\|_{\mathcal{O}.m}} \tag{4.15}$$

Kappa Return Value

$$\frac{E \vdash \mathcal{O}, \mathcal{O}', \mathcal{P}, \mathcal{P}'\ \|\mathcal{O}'.f' \Leftarrow \mathcal{P}.n()\|_{\mathcal{O}.m},\ \mathcal{P}.n \rightsquigarrow \mathcal{P}'.g'}{E \vdash \|\mathcal{O}'.f' <_{\kappa} \mathcal{P}'.g'\|_{\mathcal{O}.m}} \tag{4.16}$$

The Sigma Call-by-Name derivation in equation 4.17 combines these concepts. It traces the path of a parameter sent in using call-by-name semantics, which ensures that any changes made within the method body to that parameter will be reflected in the external environment. It then requires a modification of that passed object via an update. In such a situation, the parameter object will have a $<_{\sigma}$ reliance on the invoked method.

Sigma Call-by-Name

$$\frac{E \vdash \mathcal{O}, \mathcal{O}', \mathcal{P},\ \|\mathcal{P}.n(\mathcal{O}'.f')\|_{\mathcal{O}.m},\ \mathcal{P}.n\{\!\{\mathcal{P}.n.g \overset{n}{\Leftarrow} \mathcal{O}'.f'\}\!\},\ \|\mathcal{P}.n.g \Leftarrow \mathcal{X}\|_{\mathcal{P}.n}}{E \vdash \|\mathcal{O}'.f' <_{\sigma} \mathcal{P}.n\|_{\mathcal{O}.m}} \tag{4.17}$$

When these four base reliance operators defined in Equations 4.5 through 4.17, are combined with the subtyping operator of ς-calculus, they create a complete coverage of reliances between the elements of object-oriented programming. There simply are not any other well-formed reliances that can be defined. I will revisit this in Section 4.7.

4.3 Inferred Definitions (Transitivity)

Discussing how relationships can relate to one another now becomes possible. If an element A of a system relies on another element B, and B in turn relies on C, then we can deduce that A relies on C, even if there is no direct link detectable in the original source code defining the two relationships. This is where the deductive power of ρ-calculus comes from, the ability to take disparate code bases and create new webs of reliances between them according to how the code is actually used. For example, a library designer may intend for a system to be used in a particular way, but often the users of said library will have other ideas. ρ-calculus allows the designer to find out how the library is being utilized in the field, using the code bases as direct evidence. This feedback can provide a much clearer picture for how the library should be designed to meet the users' needs.

I will continue to use the pseudo-code example from Figure 4.1, or rather, I will use the discovered reliances as the basis for the transitive examples in this section. Collecting all the reliances from Section 4.2, we have the list in Equations 4.18 through 4.23.

$$O.m <_\mu P.n \tag{4.18}$$
$$O.m <_\mu P.n2 \tag{4.19}$$
$$O.m <_\phi O.m.retval \tag{4.20}$$
$$O.m <_\phi G \tag{4.21}$$
$$\|O.m.retval <_\sigma P.n2\|_{O.m} \tag{4.22}$$
$$\|G <_\kappa O.m.retval\|_{O.m} \tag{4.23}$$

This is a good time to point out that the above list is quite short. When dealing with reliances, I am much less concerned with *where* relationships are formed than most approaches that work with dependencies. Such information can be deduced at a time when it is necessary, and not be a burden before then.

4.3.1 Direct contextual inferences

There are a few deductions that can be made from inspecting the above reliance operators. For instance, Equation 4.22 creates a sigma-form reliance between $O.m.retval$ and $P.n2$... but it does so within the context of $O.m$. It is logical to assume that $O.m$ therefore relies on $P.n2$ in some capacity. This creates a $<_\mu$ reliance between $O.m$ and $P.n2$. Similarly, a $<_\phi$ reliance can be inferred between $O.m$ and $O.m.retval$.

Mu Sigma
$$\frac{E \vdash \mathcal{O}, \mathcal{O}', \mathcal{P},\ \|\mathcal{O}'.f' <_\sigma \mathcal{P}.n\|_{\mathcal{O}.m}}{E \vdash \mathcal{O}.m <_\mu \mathcal{P}.n} \tag{4.24}$$

Phi Sigma
$$\frac{E \vdash \mathcal{O}, \mathcal{X}, \mathcal{Y},\ \|\mathcal{X} <_\sigma \mathcal{Y}\|_{\mathcal{O}.m}}{E \vdash \mathcal{O}.m <_\phi \mathcal{X}} \tag{4.25}$$

Likewise, if we inspect Equation 4.23, reliances can be made from the enclosing context to the operands of the kappa-form reliance: $O.m <_\phi G$, and $O.m <_\phi O.m.retval$.

Phi Kappa1
$$\frac{E \vdash \mathcal{O}, \mathcal{X}, \mathcal{Y},\ \|\mathcal{X} <_\kappa \mathcal{Y}\|_{\mathcal{O}.m}}{E \vdash \mathcal{O}.m <_\phi \mathcal{X}} \tag{4.26}$$

Phi Kappa2
$$\frac{E \vdash \mathcal{O}, \mathcal{X}, \mathcal{Y},\ \|\mathcal{X} <_\kappa \mathcal{Y}\|_{\mathcal{O}.m}}{E \vdash \mathcal{O}.m <_\phi \mathcal{Y}} \tag{4.27}$$

The four deduced reliances above all appear in the previous listing of reliances that were deduced

directly from the code artifacts. This would indicate that perhaps the direct definitions of Section 4.2 are enough. This approach quickly breaks down under situations that even just slightly more complex, and it becomes obvious that inferred reliances are required. I will revisit this at the end of this section.

4.3.2 Chained inferences

Given the four reliance operators, there are sixteen potential interactions between them. If two reliance operators, indicated by $<_1$ and $<_2$ exist such that $\mathcal{X} <_1 \mathcal{Y}$ and $\mathcal{X}' <_2 \mathcal{Y}'$ then it is possible that $\mathcal{X} < \mathcal{Y}'$, but not guaranteed. Only if the right-hand side of the first reliance operator and the left-hand side of the second reliance operator are the same sort of element, can there be a new reliance formed. For instance, if $\mathcal{O}.m <_\mu \mathcal{P}.n$ and $\mathcal{O}'.f' <_\sigma \mathcal{P}'.n'$, then no $\mathcal{P}.n$ (a method) or $\mathcal{O}'.f'$ (a field) would ever create a binding between the two reliances. Hence, a μ/σ transitivity is forbidden. Similarly, the μ/κ, ϕ/μ, ϕ/ϕ, σ/σ, σ/κ, κ/μ and κ/ϕ transitivities are nonsensical. This leaves eight combinations that are possible: μ/μ, μ/ϕ, ϕ/σ, ϕ/κ, σ/μ, σ/ϕ, κ/σ and κ/κ.

A simple example of the above informal transitivity is chaining method calls, the μ/μ interaction. If $\mathcal{O}.m$ calls $\mathcal{P}.n$ within its definition, and $\mathcal{P}.n$ in turn calls $\mathcal{P}'.n'$, then it is obvious that $\mathcal{O}.m$ relies on not only $\mathcal{P}.n$, but also $\mathcal{P}'.n'$ through a simple transitivity.

$$\textbf{Mu Transitivity} \qquad \frac{E \vdash \mathcal{O}, \mathcal{P}, \mathcal{P}',\ \mathcal{O}.m <_\mu \mathcal{P}.n,\ \mathcal{P}.n <_\mu \mathcal{P}'.n'}{E \vdash \mathcal{O}.m <_\mu \mathcal{P}'.n'} \tag{4.28}$$

The μ/ϕ chain results in a $<_\phi$ reliance being formed, and is fairly self-explanatory. If $O.m <_\mu P.n$, and $P.n <_\phi P'.g'$, then it stands to reason that $O.m$ will rely on $P'.g'$ through $P.n$.

$$\textbf{Phi Mu/Phi} \qquad \frac{E \vdash \mathcal{O}, \mathcal{X}, \mathcal{Y},\ \mathcal{O}.m <_\mu \mathcal{P}.n,\ \mathcal{P}.n <_\phi \mathcal{P}'.n'}{E \vdash \mathcal{O}.m <_\phi \mathcal{P}'.n'} \tag{4.29}$$

Chains that start with a $<_\phi$ reliance operator create a slightly more complex situation. A ϕ/σ chain results in the inference of *two* $<_\mu$reliance operators. Not only does the originating method for the $<_\phi$ reliance operator, $O.m$, rely on the terminal method of the $<_\sigma$, but it also relies on the *context* that the $<_\sigma$resides in. These two situations are shown in Equations 4.30 and 4.31.

$$\textbf{Mu Phi/Sigma1} \qquad \frac{E \vdash \mathcal{O}, \mathcal{O}', \mathcal{P}, \mathcal{P}',\ \mathcal{O}.m <_\phi \mathcal{P}.g,\ \|\mathcal{P}.g <_\sigma \mathcal{P}'.n'\|_{\mathcal{O}'.m'}}{E \vdash \mathcal{O}.m <_\mu \mathcal{P}'.n'} \tag{4.30}$$

$$\textbf{Mu Phi/Sigma2} \qquad \frac{E \vdash \mathcal{O}, \mathcal{O}', \mathcal{P}, \mathcal{P}',\ \mathcal{O}.m <_\phi \mathcal{P}.g,\ \|\mathcal{P}.g <_\sigma \mathcal{P}'.n'\|_{\mathcal{O}'.m'}}{E \vdash \mathcal{O}.m <_\mu \mathcal{O}'.m'} \tag{4.31}$$

Just as the ϕ/σ chain results in two $<_\mu$ inferences, the ϕ/κ chain gives rise to two inferences, one of $<_\mu$ and one of $<_\phi$. The originating method now relies on the context of the $<_\kappa$ reliance, and also relies on the secondary field, the right hand operand of the $<_\kappa$.

Mu Phi/Kappa

$$\frac{E \vdash \mathcal{O}, \mathcal{O}', \mathcal{P}, \mathcal{P}',\ \mathcal{O}.m <_\phi \mathcal{P}.g,\ \|\mathcal{P}.g <_\kappa \mathcal{P}'.g'\|_{\mathcal{O}'.m'}}{E \vdash \mathcal{O}.m <_\mu \mathcal{O}'.m'} \tag{4.32}$$

Phi Phi/Kappa

$$\frac{E \vdash \mathcal{O}, \mathcal{X}, \mathcal{Y},\ \mathcal{O}.m <_\phi \mathcal{P}.g,\ \|\mathcal{P}.g <_\kappa \mathcal{P}'.g'\|_{\mathcal{O}.m}}{E \vdash \mathcal{O}.m <_\phi \mathcal{P}'.g'} \tag{4.33}$$

Using $<_\sigma$ as a base, either the $<_\mu$ or $<_\phi$ reliance operators may be chained. The first results in another $<_\sigma$, while the latter results in a $<_\kappa$.

Sigma Sigma/Mu

$$\frac{E \vdash \mathcal{O}, \mathcal{O}', \mathcal{P},\ \|\mathcal{O}'.f' <_\sigma \mathcal{P}.n\|_{\mathcal{O}.m},\ \mathcal{P}.n <_\mu \mathcal{P}'.n'}{E \vdash \|\mathcal{O}'.f' <_\sigma \mathcal{P}'.n'\|_{\mathcal{O}.m}} \tag{4.34}$$

Kappa Sigma/Phi

$$\frac{E \vdash \mathcal{O}, \mathcal{P}, \mathcal{P}',\ \|\mathcal{O}'.f' <_\sigma \mathcal{P}.n\|_{\mathcal{O}.m},\ \mathcal{P}.n <_\phi \mathcal{P}'.g'}{E \vdash \|\mathcal{O}'.f' <_\kappa \mathcal{P}'.g'\|_{\mathcal{O}.m}} \tag{4.35}$$

The two κ based chains present their own set of quirks. These are the only two chains that involve the interaction of two method contexts. In such a case, which is to be used as the new context? There are three options: left-associative, right-associative, and tracking.

Left-associative simply assigns the left context to the resulting context: $\|\mathcal{X} < \mathcal{Y}\|_{\mathcal{A}},\ \|\mathcal{Y} < \mathcal{Z}\|_{\mathcal{B}} \rightarrow \|\mathcal{X} < \mathcal{Z}\|_{\mathcal{A}}$. This has the advantage of recording where the chain of inference *starts*. During maintenance and verification by an engineer, this may prove to be highly useful: a starting place for investigation is explicit.

Right-associative context reverses this, and uses the right context for the final context: $\|\mathcal{X} < \mathcal{Y}\|_{\mathcal{A}}$, $\|\mathcal{Y} < \mathcal{Z}\|_{\mathcal{B}} \rightarrow \|\mathcal{X} < \mathcal{Z}\|_{\mathcal{B}}$. It is my guess, in the absence of any data, that this will prove to be less useful for manual verification.

There is, however, a more substantial reason for not using right-associative contexts. It would be possible to institute contexts for the μ and ϕ form reliance operators as well. There is essentially an implicit context for each of these, equivalent to the left operand. If that is made explicit, then contexts from these could be chained as well. This provides a formal basis for traditional dataflow and cohesion/coupling analysis within ρ-calculus. Making this context explicit also illustrates a further justification for left-associative contexts. If Equation 4.32 is rewritten using an explicit context, as in Equation 4.36, then the conclusion judgment has the proper context from the left operand of the resulting

$<_\mu$ reliance operator.

$$\frac{E \vdash \mathcal{O}, \mathcal{P},\ \|\mathcal{O}.m <_\phi \mathcal{P}.g\|_{\mathcal{O}.m},\ \|\mathcal{P}.g <_\kappa \mathcal{P}'.g'\|_{\mathcal{O}'.m'}}{E \vdash \|\mathcal{O}.m <_\mu \mathcal{P}'.g'\|_{\mathcal{O}.m}} \tag{4.36}$$

This indicates that the right-associative context is inappropriate and can be eliminated from consideration.

The final option retains the full chain of inference points: $\|\mathcal{X} < \mathcal{Y}\|_{\mathcal{A}},\ \|\mathcal{Y} < \mathcal{Z}\|_{\mathcal{B}} \rightarrow \|\mathcal{X} < \mathcal{Z}\|_{\mathcal{A},\mathcal{B}}$ This is very appealing, in that it tracks the exact method bodies that contribute to the final reliance, making verification much easier. It also present interesting opportunities for support for refactoring. Implementation of ρ-calculus in SPQR is much more complex with such a scheme, however, and instead the more simple left-associative context chaining is used in the current implementation.

κ/σ chaining obviously results in a $<_\sigma$ reliance. If a field $\mathcal{O}'.f'$ relies on another field, and that second field relies on a method $\mathcal{P}'.n'$ for its state, then $\mathcal{O}'.f'$ may rely on $\mathcal{P}'.n'$ for its state as well. Finally, κ/κ is a straight transitivity, as the μ/μ transitivity that started this section.

Sigma Kappa/Sigma
$$\frac{E \vdash \mathcal{O}, \mathcal{O}', \mathcal{P},\ \|\mathcal{O}'.f' <_\kappa \mathcal{P}.g\|_{\mathcal{O}.m},\ \|\mathcal{P}.g <_\sigma \mathcal{P}'.n'\|_{\mathcal{P}''.n''}}{E \vdash \|\mathcal{O}'.f' <_\sigma \mathcal{P}'.n'\|_{\mathcal{O}.m}} \tag{4.37}$$

Kappa Transitivity
$$\frac{E \vdash \mathcal{O}, \mathcal{P}, \mathcal{P}',\ \|\mathcal{O}'.f' <_\kappa \mathcal{P}.g\|_{\mathcal{O}.m},\ \|\mathcal{P}.g <_\kappa \mathcal{P}'.g'\|_{\mathcal{P}''.n''}}{E \vdash \|\mathcal{O}'.f' <_\kappa \mathcal{P}'.g'\|_{\mathcal{O}.m}} \tag{4.38}$$

4.3.3 Inference examples

With these in place, it becomes possible to discuss how various elements of object-oriented programming relate in a natural and intuitive manner. For instance, if I introduce a definition for `P`, as in Figure 4.2, then the rules of Equations 4.39 through 4.44 are injected into the system.

```
object P {
        field instanceVal;
        method n() {
                Q.someMethod();
        }
        method n2() {
                R.aField = Q.anotherMethod(instanceVal);
        }
}
```

Figure 4.2: Pseudo-code for object P

$$P.n <_{\mu} Q.someMethod \quad (4.39)$$

$$P.n2 <_{\mu} Q.anotherMethod \quad (4.40)$$

$$P.n2 <_{\phi} P.instanceVal \quad (4.41)$$

$$P.n2 <_{\phi} R.aField \quad (4.42)$$

$$\|R.aField <_{\sigma} Q.anotherMethod\|_{P.n2} \quad (4.43)$$

$$\|R.aField <_{\kappa} P.instanceVal\|_{P.n2} \quad (4.44)$$

Some of the inferrable reliances can be found in Equations 4.45 through 4.52. Each equation is annotated with the equations that it is derived from and the corresponding rule. Many of these can be inferred through multiple steps as well. For instance, Equation 4.48 could also have been derived from Equations 4.49 and 4.44 through **Phi Phi/Kappa**.

$\|G <_{\sigma} P.n2\|_{O.m}$	4.22, 4.23	**Sig Sig/Kap**	(4.45)
$O.m <_{\mu} Q.someMethod$	4.18, 4.39	**Mu Trans**	(4.46)
$O.m <_{\mu} Q.anotherMethod$	4.19, 4.40	**Mu Trans**	(4.47)
$O.m <_{\phi} P.instanceVal$	4.19, 4.41	**Mu Mu/Phi**	(4.48)
$O.m <_{\phi} R.aField$	4.19, 4.42	**Mu Mu/Phi**	(4.49)
$\|G <_{\sigma} Q.anotherMethod\|_{O.m}$	4.45, 4.40	**Sig Sig/Mu**	(4.50)
$\|G <_{\sigma} P.instanceVal\|_{O.m}$	4.45, 4.41	**Kap Sig/Phi**	(4.51)
$\|G <_{\sigma} R.aField\|_{O.m}$	4.45, 4.42	**Kap Sig/Phi**	(4.52)

There is a weakness in the formal inferences of ρ-calculus as it currently stands. If lines (2) and (3) of Figure 4.1 were reversed, such that `G` was assigned the value of `retval` prior to `retval` being given a new value from the method call, one would still be able to derive Equation 4.45, even though `G` would no longer rely on `P.n2` in that invocation. This is due to ρ-calculus being completely ignorant of the execution order of statements in a method body. Equations 4.22 and 4.23 would still hold, and therefore 4.45 would be derivable. Temporal logic would be a welcome addition to ρ-calculus and would introduce the necessary formalisms to support more accurate forms of reliances. For the purposes of SPQR, however, where I am more interested in large scale pattern detection, this weakness creates a surprisingly small number of false positives. I will address this further as a direction for future research in Section 12.2.

Oddly, the usefulness of SPQR even in the absence of temporal formality indicates that it may be practical in situations where temporal execution is less rigid, such as parallel and concurrent computing (Jordan and Alaghband, 2002), or perhaps even ill-defined, such as quantum computing environments

(Feynman, 1999; Feynman, 1988; Steane, 1988). In such situations traditional analysis techniques are not likely to be valid, but ρ-calculus-based approaches may still prove viable.

4.4 Selection Partitioning

In any selection as defined in ς-calculus, one can look at the two halves of the selection operator, and consider them separately. Not only does selection pick an item out of an object for use, it also fulfills the basic requirements of a scoping operator in most programming languages. Selection in this light can be thought of as the :: operator in C++. Other languages such as Java have unified the scoping and selection operators as ς-calculus does.

When looked at in this manner, and considering ς-calculus's general ban on free or unbound methods, it should become obvious that every method and every field will be on the right side of a selector at all times. Any 'raw' object is therefore one that is not being considered as a field. In the following discussion, $o.m$ will represent the general selection of a method m from an object o. Similarly, $o.f$ will stand for the selection of a field f from an object o.

Selection follows the same common sense rules as scoping, including nesting. Just as a piece of source code may refer to `window.titlebar.title.setFont()` to select a deeply nested method, so too can ς-calculus perform the same nested selection: $window.titlebar.title.setFont$.

Only the rightmost item in such a nested selection chain is the current selection, the remainder provides a scoping for it. To make this distinction explicit, and allow for easier discussion of the two halves separately, I introduce the terms *leftdot* and *dotright*, written formally as ($\bullet$ and $\bullet$). These correspond respectively to the elements to the left of the most significant selection operator, and the final element in the chain. Equations 4.53 and 4.54 give a formal notation and definition for this. The names and symbols were chosen to reflect Dirac's *bra* ($<|$) and *ket* ($|>$) notation in quantum mechanics (French and Taylor, 1978). As in that discipline, this bisected operator indicates clearly that neither is part of a complete concept on its own. Each requires an instance of the other to form a whole.

$$\textbf{Leftdot} \qquad (\mathcal{O}.s\bullet \equiv \mathcal{O} \tag{4.53}$$

$$\textbf{Dotright} \qquad \bullet\mathcal{O}.s) \equiv s \tag{4.54}$$

For example, the application of the leftdot operator to `window.titlebar.title.setFont()` would

be `window.titlebar.title` and the result of dotright would be `setFont()`. Formally:

$$\begin{array}{rcl}(window.titlebar.title.setFont\bullet & = & window.titlebar.title \\ \bullet window.titlebar.title.setFont) & = & setFont\end{array}$$

Raw objects are a special case. They are without a selection operator, so there are three choices on how to proceed: disallow as an error, treat the raw object as a scoping, or treat the raw object as a selection target. The last is the course I choose, as it reflects how objects are used by programmers. If a global object is used as the parameter to a method, that object is being selected for use as the parameter. Assume that $\mathcal{O}$ represents a raw object without a scoping chain, and Equations 4.55 and 4.56 define this case. The leftdot of a raw object is a null element, an empty set. The dotright of a raw object is the object itself.

$$\textbf{Leftdot Raw} \qquad (\mathcal{O}\bullet \equiv \{\} \tag{4.55}$$

$$\textbf{Dotright Raw} \qquad \bullet\mathcal{O}) \equiv \mathcal{O} \tag{4.56}$$

We can now provide an identity relationship between leftdot, dotright, and selection that should make the choice of notation more intuitive, where $\mathcal{X}$ can be either a raw object $\mathcal{O}$, or a selected subelement $\mathcal{O}.s$:

$$\textbf{Selection Identity} \qquad (\mathcal{X} \bullet . \bullet \mathcal{X}) \equiv \mathcal{X} \tag{4.57}$$

A small tweak of the selection operator of ς-calculus makes the above well-formed for raw objects:

$$\textbf{Selection of Global} \qquad \{\}.\mathcal{O} \equiv \mathcal{O} \tag{4.58}$$

4.5 Similarity Principle

With a formal notation for selection parameters, precise discussion of one of the most important elements of ρ-calculus is possible: the *principle of similarity*. I define similarity to be 'of a common conceptual nature', a measure of how closely related in *intent* two elements of programming are. If two methods have the same intent of function, or two objects have the same intent of state encapsulation, it can be expected that they will perform similar tasks within a system. This gives strong cues as to the *relative* functionality of the two elements, providing a new aspect of the binary reliance operators, another axis

of relationship alongside reliance. In the type of relationship analysis performed by SPQR, it is more important to deduce the relationship between two elements in the design than it is to extract their precise functionality at runtime.

The question then becomes how to measure similarity of intent. I turn to Kent Beck for an initial answer (Beck, 1997), in his Intention Revealing Selector best practices pattern: (emphasis mine)

> You have two options in naming methods. The first is to name the method after how it accomplishes its task. ... *The most important argument against this style of naming is that it doesn't communicate well.* ... The second option is to name a method after what it is supposed to accomplish and leave "how" to the various method bodies. This is hard work, especially when you only have a single implementation. Your mind is filled with how you are about to accomplish the task, so it's natural that the name follow "how". The effort of moving the names of method from "how" to "what" is worth it, both long term and short term. *The resulting code will be easier to read and more flexible.*

Beck sums up this as "Name methods after what they accomplish." This is obviously and easily extensible to fields and objects as well, and can be more generalized as "Name elements of programming after what they accomplish or represent." Since communication of code intent and flexibility are two key components of comprehensibility and maintenance, this single principle serves as a basis for important conceptual analyses.

As a first attempt, then, comparing the names of elements and looking for commonalities would seem to be reasonable. Common naming would indicate common intent according to the above pattern. Examples of this can be seen in a number of practical applications. The JavaBeans architecture uses a 'set*Attribute*' idiom to indicate a settor method during runtime inspection (Monson-Haefel et al., 2004). Within Apple's Cocoa framework, naming conventions are ubiquitous and standardized - usually a method name can be guessed from the intent, without needing to look it up (Davidson and Inc., 2002).

The comparison logic then becomes an interesting exercise: what type of comparison is appropriate to conclude that two names are similar, and, by extension, that their corresponding elements have similar intended purpose? The most simple, and the approach initially used in SPQR, is a straight lexicographic equality check. While simple, this worked much better than I expected. This can be explained anecdotally by noticing that for any group of engineers, whether it is a group consisting of a single practitioner, or many, nomenclatures will tend to standardize fairly quickly. A communal naming scheme must arise for common conceptual discussion, to be able to communicate easily and precisely the very intents I am interested in, as well as to allow the code to be developed cleanly. While SPQR does not attempt to extract the exact intents, the namings can be leveraged to divine the relative intents much more easily. This social engineering observation is critical, and I will revisit this when describing

the current implementation of SPQR, and results of validation. For now, it is only necessary to have the concept of similarity at hand, and leave the implementation details for Chapter 9. I will discuss alternate similarity extraction techniques in Section 12.6.2.

The reverse is also useful, knowing when two elements are explicitly *dissimilar*. This is not the same as not knowing the similarity relationship of two elements, this is a specific negation of the similarity principle, when it is deduced that two elements simply do not have the same intent, to a high degree of confidence. The $\sim$ operator is used to indicate similarity between two elements, and $\nsim$ an explicit dissimilarity.

With this conceptual definition in place, I can now define the similarity imbued reliance operators. Because similarity turns out to be such an important analysis principle, the reliance operators are explicitly tagged with the similarity information. A superscript is added, with the following format: $^{\pm}_{\circ}.^{\pm}_{\circ}$, two *similarity indicators* separated by a selection operator. The $^{\pm}_{\circ}$ indicates that one of $+, -$ or $\circ$ can be used. Similarity is indicated by a $+$, dissimilarity by a $-$, and $\circ$ is a placeholder for an unknown relationship. The two similarity indicators are separated by a selection operator, mirroring the outermost selection operator on each side of the reliance operator. The left side of the similarity information indicates the similarity relationship between the leftdot of each side, and the right side of the tag indicates the similarity between the dotright of each side. This is formalized in Equations 4.59 through 4.62. $\mathcal{X}$ and $\mathcal{Y}$ stand for either a raw object, or a selector operator and parameters, as before. $<$ in this context is understood to mean any of the reliance operators.

Leftdot Similarity

$$\frac{\mathcal{X} < \mathcal{Y},\ (\mathcal{X}\bullet \sim (\mathcal{Y}\bullet}{\mathcal{X} <^{+.\circ} \mathcal{Y}} \tag{4.59}$$

Dotright Similarity

$$\frac{\mathcal{X} < \mathcal{Y},\ \bullet\mathcal{X}) \sim \bullet\mathcal{Y})}{\mathcal{X} <^{\circ.+} \mathcal{Y}} \tag{4.60}$$

Leftdot Dissimilarity

$$\frac{\mathcal{X} < \mathcal{Y},\ (\mathcal{X}\bullet \nsim (\mathcal{Y}\bullet}{\mathcal{X} <^{-.\circ} \mathcal{Y}} \tag{4.61}$$

Dotright Dissimilarity

$$\frac{\mathcal{X} < \mathcal{Y},\ \bullet\mathcal{X}) \nsim \bullet\mathcal{Y})}{\mathcal{X} <^{\circ.-} \mathcal{Y}} \tag{4.62}$$

Note that dotright similarity makes little sense for the reliance operators that have different types of selected elements on the two sides of the reliance operator: $<_\sigma$ and $<_\phi$. Since the left side of $<_\sigma$ has a dotright of a field, and the right side has a dotright of a method, attempting to extract similarity will be

useless at best, and downright erroneous at worst. (The mirror case holds for the $<_\phi$ reliance operator.) A method and a field have utterly different intents of functionality from their most basic definitions. A coincidental positive similarity would therefore be a false positive. Since they can be avoided at this early stage before propagating through the inferences, I do so.

The similarity information for these two reliance operators may therefore be written without the trailing similarity indicator and the separation operator. It does no harm to include these, as long as the trailing indicator is always a $\circ$, but it is redundant information.

One last operator has yet to be addressed, the subtyping operator of ς-calculus: $<:$. Consider what similarity means in this case. Leftdot similarity would indicate a similarity of scoping, perhaps that the two types are contained within the same package or namespace. Dotright similarity would indicate that the types are expected to perform the same type of role in a system. If the two dotright similar types are in different packages, then this provides a clue that the two types perform similar roles in the two domains. This sort of information can be as useful for types as it is for methods and fields. For example, most analysis of large systems attempts to make distinctions between nested classes, packages, modules, and so on, when they all fundamentally conform to the concept of scoping, just as with the rest of ρ-calculus. Unifying these into one mechanism allows analysis to shift transparently and effortlessly between levels of abstraction with regards to types.

For convenience of notation, and for completeness, I include the rules for reversing the above inferences. The Similarity Generalizations in Equations 4.63 and 4.64 state that any specialized similarity trait form of a reliance operator implies that the more general form is also valid. This is an obvious inference, but it is noted here with a warning that applies to all such generalization rules: some inference systems will throw away more specialized facts about a system if a more general rule can be inferred and kept instead. This point will be revisited in Chapter 9 when I discuss the current implementation of SPQR.

Leftdot Generalization

$$\frac{\mathcal{X} <^{\pm\,\pm}_{\circ.\circ} \mathcal{Y}}{\mathcal{X} <^{\circ.\pm}_{\circ} \mathcal{Y}} \tag{4.63}$$

Dotright Generalization

$$\frac{\mathcal{X} <^{\pm\,\pm}_{\circ.\circ} \mathcal{Y}}{\mathcal{X} <^{\pm}_{\circ.\circ} \mathcal{Y}} \tag{4.64}$$

4.6 Consistency

When supplementing a logical framework such as ς-calculus, there is always the risk of introducing new judgments that create inconsistencies within the core logic. A simple argument can be used to show that the four main reliance operators do not in any way introduce such inconsistencies within ς-calculus, nor can their productions.

None of the ρ-calculus judgments introduced in this chapter give rise to ς-calculus elements. The derivation is strictly and formally unidirectional. The similarity constructs likewise can not produce new ς-calculus elements. This prevents ρ-calculus from introducing inconsistencies in the underlying ς-calculus.

Additionally, the ρ-calculus creation and inference judgments produce positive clauses only. There is no case where two directly complementary clauses can be derived, indicating that the ρ-calculus rules are internally consistent. Complementary in this case is defined in the usual propositional logic sense, meaning direct negation of another clause with whom resolution will result in termination (Leitsch, 1997). The similarity principle judgments are reliant on the method of similarity determination, but given an exclusivity algorithm such that no two elements can be both similar and dissimilar, there is no inference chain that can result in a contradiction. The enumeration of the possible interactions between reliance operators is what gave rise to the judgments in Section 4.3. Because the only valid interactions have been handled in the inference equations in that section, all possible inconsistencies within ρ-calculus have been addressed.

The only reliance relationship that is inherent within ς-calculus is the inheritance or subtyping operator <:. When combined with subsumption, it does give rise to a peculiarity with the other reliance operators, as discussed and dealt with in the next section.

4.7 Objects vs. Types

It is natural to consider using types instead of raw objects in the reliance operators, but such a system must be approached carefully. There is an incompatibility between Δ_ρ and subsumption that prevents a purely type-based analysis. I will discuss the effects of allowing replacement of an object-based system with a type-based system in two parts - those rules that would allow replacement on the left-hand side of a reliance operator, and those that would allow replacement on the right-hand side. I will then discuss how SPQR avoids these issues.

4.7.1 Left-Hand Side Type Replacement

Let us add a rule that allows an object to be replaced by its type on the left-hand side of $<_\mu$:

$$\frac{\mathcal{O}.m <_\mu \mathcal{P}.n,\ \mathcal{O} : \tau}{\tau.m <_\mu \mathcal{P}.n}$$

On the face of it, this seems reasonable. Assuredly the definition of the object type τ includes the same invocation call to $\mathcal{P}.n$. If, however, we include type subsumption using the Val Subsumption rule of the $\Delta_{<:}$ fragment (Abadi and Cardelli, 1996, pg.93):

$$\frac{\tau.m <_\mu \mathcal{P}.n,\ \tau' <: \tau}{\tau'.m <_\mu \mathcal{P}.n}$$

we now have a problem, since $\tau'.m$ may in fact replace the method body defined in τ with one that does **not** include the call to $\mathcal{P}.n$. We have no way of asserting that the above is true in any first or second order language.

A similar problem appears with $<_\phi$:

$$\frac{\mathcal{O}.m <_\phi \mathcal{P},\ \mathcal{O} : \tau}{\tau.m <_\phi \mathcal{P}}$$

Here again this seems self-consistent, but we find that after adding type subsumption:

$$\frac{\tau.m <_\phi \mathcal{P},\ \tau' <: \tau}{\tau'.m <_\phi \mathcal{P}}$$

we arrive at the same problem. The method body of $\tau'.m$ may eliminate the reference to the object, if for instance, it was a local variable returned in the definition of Phi Reliance (Return).

The method-body-based reliance operators are therefore incompatible with subsumption as defined in ς-calculus's $\Delta_{<:}$ fragment. If we look at the update-based reliance operators, such as $<_\kappa$, we find no immediate problem:

$$\frac{\mathcal{O} <_\kappa \mathcal{P},\ \mathcal{O} : \tau}{\tau <_\kappa \mathcal{P}}$$

$$\frac{\tau <_\kappa \mathcal{P},\ \tau' <: \tau}{\tau' <_\kappa \mathcal{P}}$$

Since method/field extraction is prohibited in ς-calculus (Abadi and Cardelli, 1996)[pg. 14], we can be assured that any subtypes of τ will still include the reference we found in τ. A subtype cannot remove references defined in a supertype. It may hide them from external access, ignore them or otherwise make

them less than useful, but it cannot outright delete them, so our final inference holds for directly defined instances of $<_\sigma$ and $<_\kappa$.

There is, however a serious problem lurking. We do not know *how* the above instance of $<_\kappa$ was generated, and it may have been inferred from a chain of transitivities involving one or more $<_\mu$ or $<_\phi$ facts. This would invalidate the accuracy of the $<_\kappa$ under subsumption.

The above indicates that any type-based analysis of program structures that includes both ς-calculus and λ-calculus concerns, and that does not validate or invalidate object to type conversion based on the entire inference history of the independent elements, is inherently flawed.

4.7.2 Right-Hand Side Type Replacement

The above incompatibility of typing on the left side of reliance operators does not appear when we consider the right-hand side, but other oddities do appear that weaken the assertions of inference.

Consider a fact of the form $\mathcal{O}.\mu <_\mu \mathcal{O}'.\mu'$, and perform a right-hand side object-to-type replacement and then subsumption of types:

$$\frac{\mathcal{O}.m <_\mu \mathcal{P}.n,\ \mathcal{P} : \tau}{\mathcal{O}.m <_\mu \tau.n}$$

$$\frac{\mathcal{O}.m <_\mu \tau.n,\ \tau' <: \tau}{\mathcal{O}.m <_\mu \tau'.n}$$

What we find is that this is a *possible* reliance - it is an example of polymorphism, and it *may* be an inferable fact, but it also may never occur. It is interesting that we have come to a state that is not directly contradictable, yet not directly supportable. Right hand side type replacement is sound under subsumption but illuminates the depth of the intricacies with polymorphism.

4.7.3 Ramifications

A purely object-based system can be created in theory, as it would be essentially a ς-calculus interpreter. Attempting to perform exhaustive deterministic analysis on such a system using first-order logic would, however, be equivalent to solving the halting problem. A practical approach requires a different technique.

A purely type-based system is easily implemented, but it will fail to find many relationships in a dynamically typed language. The unique nature of ς-calculus will be lost in such an environment, as concepts such as Self (Abadi and Cardelli, 1996) become intractable.

This is precisely where ρ-calculus step in. By providing a slightly 'fuzzy' semantics, object-based

```
class Superclass;
class SubclassA : Superclass;
class SubclassB : Superclass;

int
main(int argc, char** argv) {
    Superclass* obj;
    if (argv[1] == "A") {
        obj = new SubclassA();
    } else {
        obj = new SubclassB();
    }
};
```

Figure 4.3: Polymorphic object

inferences can be generated wherever possible, but a mapping to a type-based system is then used for practical analysis using a first-order solver. Subsumption is ignored in the analysis altogether, as it constitutes a run-time behavior of dynamically typed languages. Without subsumption, type-base analysis becomes consistent under ρ-calculus.

Of course, this loses some run-time typing analyses which are possible under dynamically typed languages. Much of this is recaptured using a technique I call *superpositional polymorphism*. In an analogue to the superposition principle of quantum mechanics (French and Taylor, 1978), an object is said to be of many types simultaneously, based on creational statements in the code. Essentially, each object is treated as *all* possible types that it can be given in the system. For example, for the code in Figure 4.3, a deterministic analysis would state that the type of *obj* is unknowable. It may be $SubclassA$, or it may be $SubclassB$, but knowing which is impossible without the value of the parameter being passed in. Superpositional polymorphism, however, states that it is of the type set $\{Superclass, SubclassA, SubclassB\}$. It is then possible to analyze *obj* as if it were each of the possible types individually, but simultaneously. This differs from traditional polymorphism, at least as viewed by most programmers, where an object is one specific type of many, at any given point in time. Superpositional polymorphism states that the object is *all types at once*. To further push the analogy, the specific type of a polymorphic object cannot be known until run time, much as observation of a superpositional subatomic particle collapses the wave function into one specific state.

Because of this, a vast space of *potential* inferences are made. It is not guaranteed that each will be present in any particular run of a codebase, but each is a possibility. This non-determinism is what allows ρ-calculus to be used in a practical manner.

To recap, object-based analysis is the preferred approach, but is currently beyond the scope of current inference engines and solvers for practical use. Straight type-based analysis is unsound under subsumption, but can be useful for cases where subsumption can be ignored. Superpositional polymorphism provides a slightly less deterministic analysis than object-based analysis, but in a practical sense makes up for the lack of subsumption that type-based analysis requires.

4.8 Relation to other work

It is important to note that I have developed an extension to an existing formal notation for object theory, ς-calculus, which in turn is firmly linked to existing procedural theory, λ-calculus. In this manner ρ-calculus acquires the capabilities of a vast body of knowledge and analysis techniques. I am not setting patterns up as a language unto themselves, but rather am showing a definite chain of abstractions, from the lowest, most concrete levels of programming, to the highest concepts with which system designers work.

This approach provides more detail than the formal description provided by UML, for instance. ρ-calculus maps nicely to the concepts of IsA, HasA, HoldsA, UsesA, and so on that exist within UML, indicating that a simple mapping between the two should exist. Unlike UML, however, reliance operators encode entire paths of reliances in a concise notation. All the reliances and relationships in the UML graphing system are encoded within the element that is under scrutiny, reducing the need for extended, and generally recursive, analysis for each element when needed.

Common concepts such as IsA in UML are directly expressable in ς-calculus using constructs such as, for instance for IsA, the transitive subsumption operator $B <: A$, indicating a relationship between a superclass (A) and a subclass (B). Other relationships, however, such as HasA, HoldsA, and UsesA, have no simple analogues in the base ς-calculus.

The reader may recognize that the reliance operators are in many cases a reformation of existing data and method dependency research (Kennedy, 1981; Hecht, 1977; Bieman and Ott, 1994; Hitz and Montazeri, 1995) within the ς-calculus framework. I expect that with this basis, much more of such research can be formally expressed within ς-calculus, adding to a common knowledge pool that can be leveraged throughout the analysis techniques used in SPQR. I also expect, however, that temporal logic will be required, as outlined in Section 4.3.3, before this deep data-driven reliance research can begin in earnest.

Chapter 5

Elemental Design Patterns

In this chapter I will discuss the *Elemental Design Patterns*, or *EDPs*. The EDPs are the conceptual building blocks of object-oriented programming design, in the same way that the reliance operators of ρ-calculus are the formal building blocks of object-oriented programming. There is a strong correlation between the EDPs and the reliance operators, and ρ-calculus is used formally to define the EDPs. I will define the general class of EDPs, explore the relationship between EDPs and ρ-calculus, provide the basic EDP catalog and then illustrate how the EDPs are ubiquitous throughout the Gang of Four patterns. The complete EDP Catalog, written in the format of the Gang of Four patterns, can be found in Appendix B, and is an expanded version of the material which has appeared in publication previously (Smith, 2002; Smith and Stotts, 2002; Smith and Stotts, 2003). There the reader will find detailed and thorough discussions of each pattern. This chapter will only introduce them as basic concepts, and provide a formal mapping from ρ-calculus where appropriate.

5.1 Elemental Design Patterns

As discussed in Chapter 2, the decomposition of design patterns is not a particularly new idea. What is new in this work is the depth to which the decomposition is taken. Most of the literature in the design patterns field is concentrated on expanding the design patterns to ever higher levels of abstraction. While this has obvious worth, I believe that without a solid foundation for such work, it will only become increasingly difficult to use patterns in well-formed and meaningful ways. Tool support for engineers will simply not be able to keep up, and programmers will be left in the same morass, lacking formal support as they do today.

I defined the term Elemental Design Patterns for two reasons. First, these are the fundamental

building blocks of design and programming, just as the elements are the building blocks of chemistry. Second, these are elemental in the sense that they are very simple concepts that every programmer uses, usually instinctually. There is very little at this level that an experienced programmer will find new. Instinctual design is not conscious design, however, and it lacks the very traits that could make software design an actual engineering discipline: formal foundations, methods that are reproducible and teachable, and consistent. Self-conscious design is precisely the goal that Alexander set with his original work on design patterns in architecture, and we should not ignore the lowest levels when trying to build larger abstractions. These lower levels are the core of programming, and all design can be decomposed into them.

As shown in Chapter 4, the building blocks of object interaction in object-oriented languages can be defined as binary relationships: object use through update and selection, method calls, and typing relations are the *only* ways that objects, methods, fields, and types can interact. Any programming feature from a language expressible in ς-calculus will reduce to some combination of those interactions. It is, therefore, an obvious step to attempt to quantify what those interactions are, and how they are used to build the conceptually rich abstractions of software design. These interactions are the Elemental Design Patterns, and they quickly become capable of solving rather esoteric, yet familiar, design problems.

At first glance, these EDPs seem highly unlikely to be very useful, as they appear to be positively primitive. That very simplicity, when combined with the formal derivations of ρ-calculus, is what provides the opportunity for formalization of complex programming concepts. These are the core primitives that underlie the construction of patterns in general. Patterns are, to be precise, descriptions of relationships between objects, according to Alexander (Alexander, 1964), and method invocations and typing are the process through which objects interact. I believe that I have captured the elemental components of object-oriented languages, and the salient relationships used in the vast majority of software engineering. If patterns are the frameworks on which to create large understandable systems, these are the nuts and bolts that comprise the frameworks.

Each EDP is unique from the others, each satisfies a different set of constraints, a different set of forces, and solves a slightly different problem. Each provides a degree of semantic context and a bit of conceptual elegance, in addition to a purely syntactical construct. In this context these are still truly patterns, and provide us with an interesting opportunity: to begin to build patterns from first principles of programming, namely formalizable denotation.

EDPs, as well as all other patterns, are expressed in ρ-calculus as a tuple construct of the form

PatternName($participant1, participant2, ...$), with the ordering of the participants fixed for each definition. They are defined formally as reduction rules with required criteria and a conclusion, just as the primary reliance operators are, and form a convenient shorthand notation for highly abstract concepts. By judiciously mixing a combination of formalized EDP instances, reliance operators, and lower level constructs from ς-calculus and λ-calculus, a denotation environment is created that is extremely expressive and precise, yet flexible. By convention, when a pattern name is expressed as above, in bold text, it indicates a formal definition of the pattern in ρ-calculus, while standard text indicates discussion of the pattern as expressed in the literature.

5.2 Types

Types are an important component of ρ-calculus, but ρ-calculus is less concerned with particular type formalizations themselves than the relationships between types. ς-calculus handles the underlying type mechanics easily. Two relationships are considered primary, **Inheritance** and **AbstractInterface**.

5.2.1 Inheritance

The first and most obvious typing relationship is inheritance, a class-based version of subtyping. While subtyping is not precisely the same as inheritance, the latter terminology is most commonly used in practical engineering, and I adopt it here. This is precisely what it would appear to be, a simple reaffirmation of ς-calculus's $A <: B$. In practice, these are interchangeable principles, and I will frequently use the ς-calculus notation as shorthand.

$$\frac{E \vdash Subclass, Superclass,\ \ Subclass <: Superclass}{\textbf{Inheritance}(Superclass, Subclass)} \tag{5.1}$$

Not all languages support inheritance directly, it may be pointed out that, instead relying on dynamic subtyping analysis to determine appropriate typing relations, such as in Emerald (Jul et al., 1991), or cloning mechanisms in prototype based languages such as Cecil (Chambers, 1993) or NewtonScript (Apple, 1993).

5.2.2 AbstractInterface

Types create a mapping from a name to a set of definitions of elements of that type, methods and fields. There are cases where such definitions are incomplete, and rely on a later subtype to copmlete

the definition. In the case of methods, this is called a *virtual method*, an *abstract method*, or a *deferred definition*. While most binary interactions are defined ahead of time, this one is a bit different in that it states, at the point of method declaration, that a nebulous and unknown *someone* will fulfill the promise of a method. There is still a binary relationship between the declaring type and the 'someone'... the missing type will simply be filled in later. See the **FulfillMethod** pattern in Section 6.1 for the definition of that missing type.

$$\frac{E \vdash C,\ C : [l_i : B_i^{i \in 1..n}] \qquad C \equiv [l_i = b_i^{i \in 1..m-1, m+1..n}, l_m = []]}{\textbf{AbstractInterface}(C, l_m)} \tag{5.2}$$

5.3 Objects

Objects (and fields) are the second major piece of ς-calculus, and require a bit of runtime handling. Object creation is a runtime behavior, as is object retrieval. While certain classes of these cases can be detected in a generalized manner, these are EDPs that require a close understanding of the target language.

5.3.1 CreateObject

Once an object is defined and instantiated, the prior discussions of ς-calculus and ρ-calculus apply transparently, but the process by which objects are injected into the system has until now remained undefined. The reason for this is that it is extremely dependent on the semantics of the programming language being analyzed.

Some languages are class-based (Accredited Standards Committee X3 (Information Processing Systems) – Working Group 21, 1996; Gosling et al., 2005), some are object-oriented and use prototypes (Shalit, 1996), others use cloning (Agesen et al., 1993), and still other combine features of traits, cloning, and pseudo-classes (Madsen et al., 1993). All of these result in the creation of new objects, but the methods vary so widely that attempting to create a single ς-calculus-based rule in anything short of a higher-order language such as O-3 is certainly going to miss a large number of cases. This highlights a weakness of the ρ-calculus as implemented in this first iteration of SPQR, that of using only first-order semantics from ς-calculus. The reasons for this will become clear when I discuss the current implementation of SPQR in Chapter 9.

Because of this, detection of **CreateObject** is assumed to be done prior to analysis through inference

$$\frac{\begin{array}{rcl} A & \equiv & \mathbf{Object}(X)[l_i\nu_i : B_i^{i\in 1..n+m}] \\ \overline{A} : \mathbf{Class}(A) & \triangleq & \mathbf{subclass\ of}\ \overline{A'} : \mathbf{Class}(A') \\ & & \quad \mathbf{with}\ (\ \mathbf{self}\ : \lfloor A \rfloor) \\ & & \quad\quad l_i = b_i^{i\in n+1..n+m} \\ & & \quad \mathbf{override} \\ & & \quad\quad l_i = b_i^{i\in Ovr} \\ & & \quad \mathbf{end} \\ a & = & \mathbf{new}\overline{A} \end{array}}{\mathbf{CreateObject}(A, a)} \tag{5.3}$$

Where $A' = \mathbf{Object}(X)[l_i\nu_i : B_i^{i\in 1..n}]$ (and may be *Root*,) $\lfloor A \rfloor = A$ for O-1, $X <: A$ for O-2, and $X <\# A$ for O-3 compliant languages, respectively.

and ρ-calculus's transitivities. It is therefore appropriate to take instances of **CreateObject** as axiomatic in SPQR analysis. A full example of doing so for C++ can be found in Section 8.8.

It may seem odd to define an EDP that is not directly analyzable within SPQR from first principles, but this illustrates an important point: when analyzing highly complex and abstract concepts of programming, it is sometimes better to introduce another level of analysis to handle certain constructs. Appropriate tools at appropriate levels will produce appropriate analysis. The strength of SPQR to accommodate such varied other analyses into a common conceptual framework will also become evident during the discussion of its implementation.

It is, however, possible to define **CreateObject** within ρ-calculus using the elements of ς-calculus's O-1, O-2 and O-3 languages, as in Equation 5.3.

5.3.2 Retrieve

Retrieve is the primary mechanism by which objects dynamically become aware of one another during runtime. The most common form, and simplest, is when an object uses the return value of a method call. The value returned can either be a reference to an object, or an actual object. These correspond to the Equations 5.4 and 5.5. Other retrieval mechanisms may be created in the future with further thought regarding the $<_\sigma$ and $<_\kappa$ reliance operators.

$$\frac{\begin{array}{c} o.s \Leftarrow o'.s' \\ o'.s' \overset{v}{\rightsquigarrow} x \end{array}}{\mathbf{Retrieve}(o.s, o'.s', x)} \tag{5.4}$$

$$\frac{\begin{array}{c} o.s \Leftarrow o'.s' \\ o'.s' \overset{n}{\rightsquigarrow} x \end{array}}{\mathbf{Retrieve}(o.s, o'.s', x)} \tag{5.5}$$

5.4 Methods and Objects - Leftdot similarity

After types and objects, only methods remain to be discussed. The method-derived EDPs comprise the bulk of the EDP catalog, so I will begin with simple cases and show how adding information incrementally quickly creates extremely expressive conceptual elements of programming. In this section I will assume that some object *anObject* contains a method *callingMethod*, as before, and that *callingMethod* contains a method call to some method of *anObject*. Figure 5.1 illustrates this in simple pseudo-code.

```
object anObject:
        method callingMethod:
                anObject.x();
        method possibleTargetMethod:
                pass;
```

Figure 5.1: Example self-referential object and method call

From Section 4.2, this is defined as $anObject.callingMethod <_{\mu} anObject.x$. Furthermore, from Equation 4.59, it is an example of leftdot similarity: $anObject.callingMethod <_{\mu}^{+\cdot\circ} anObject.x$. In practical terms, this is an example of using the `self` or `this` construct of a programming language, and as such is easily detectable in source code analysis.

5.4.1 Conglomeration

Consider first the case where `x` is replaced by `possibleTargetMethod`, such that in addition to the leftdot similarity, we have a distinct dotright dissimilarity:

$$anObject.callingMethod <_{\mu}^{+\cdot-} anObject.possibleTargetMethod$$

Conceptually, we have a method of an object that is using helper methods within the same instance of the same object to perform a task. It is *conglomerating* behavior from a number of related methods to complete a procedure. It does not matter what that larger task is, and it does not matter what other methods are being called. The only criteria is that these are distinct methods within the same instance object. This defines the **Conglomeration** EDP, as in Equation 5.6.

$$\frac{\begin{array}{l} c : Conglomerator \\ c.operation <_{\mu}^{+\cdot-} c.operation2 \end{array}}{\textbf{Conglomeration}(c, operation, operation2)} \tag{5.6}$$

5.4.2 Recursion

It should be fairly obvious from the title of this EDP what converting the dotright dissimilarity to a similarity accomplishes. This is simple recursion of a method.

$$\frac{\begin{array}{l} Recursor : [l_i : {B_i}^{i \in 1 \ldots m}, operation : B_{m+1}], \\ r : Recursor, \\ r.operation <_{\mu}^{+\cdot +} r.operation \end{array}}{\textbf{Recursion}(r, operation)} \tag{5.7}$$

5.5 Methods and Objects - Leftdot dissimilarity

How do the two cases above differ if the target object is found to be dissimilar? This is the most common situation in most code, so it is important to have well defined.

```
object anObject:
        method callingMethod:
                anotherObject.x();
object anotherObject:
        method callingMethod:
                pass;
        method possibleTargetMethod:
                pass;
```

Figure 5.2: Example generalized object and method call

Inject another object into the system, as with Figure 5.2. Note that a `callingMethod` appears in both `anObject` as well as `anotherObject`. There is nothing being said about the relationship between the types of the two objects, however. They may be related in type, or their two types may have nothing explicitly defined to do with one another. At this stage the types are irrelevant, but such a situation will be discussed in Section 5.7. The initial reliance operator here is $anObject.callingMethod <_{\mu}^{-\cdot \circ} anotherObject.x$.

5.5.1 Delegate

Replacing `x` with `possibleTargetMethod`, and a dissimilarity is created on both leftdot and dotright sides of the selector operator. In doing so, not only is `anObject` disassociated from `anotherObject`, but after sufficient analysis no reasonable speculation can be made about the possible relationship between

`callingMethod` and `possibleTargetMethod`. This is the most general case for a method call, where one object's method is *delegating* a portion of its workload to another method in another object.

$$\frac{\begin{array}{l} Delegator : [target : Delegator, operation : B_i] \\ Delegator \equiv [operation \Leftarrow ..., target.operation2, ...] \\ Delegatee : [operation2 : B_i] \\ del : Delegator \\ del.operation <_{\mu}^{-\cdot -} target.operation2 \end{array}}{\textbf{Delegate}(del, target, operation, operation2)} \tag{5.8}$$

5.5.2 Redirect

If `x` is replaced by `callingMethod`, however, a semantic link can be made through the principle of similarity. As before, any appropriate criteria for establishing similarity is applicable, I am simply using a lexicographic tag for illustration.

$$\frac{\begin{array}{l} Redirector : [target : Redirectand, operation : B_i] \\ Redirector \equiv [operation \Leftarrow ..., target.operation, ...] \\ Redirectand : [operation : B_i] \\ red : Redirector \\ redirector.operation <_{\mu}^{-\cdot +} target.operation \end{array}}{\textbf{Redirect}(red, target, operation)} \tag{5.9}$$

The distinction here made between **Redirect** and **Delegate** is important. Redirection indicates that some portion of the current task has been redirected to another object to perform, while delegation indicates that some *subset* of the work has been shunted off to another object or method. The difference is one of encapsulation: **Redirect** uses the similarity principle between method invocations, while **Delegate** makes no such claim. We would expect to see **Redirect** within a 'for each' style call on a container, where the conceptual task asked of the container is then handed off directly to the individual components. **Delegate**, on the other hand, indicates that some smaller piece, not necessarily conceptually related to the larger task at hand, is being handed off, such as when a method calls helper methods to each perform some portion of the work, which the enclosing method then integrates. This is obviously close in spirit to **Conglomeration**, except now not even the same instance object is being utilized.

This break between **Redirect** and **Delegate** is admittedly at odds with the standard use of the term *delegation*, where any portion of the work is handed off, regardless of the conceptual relationship. The standard delegation, in general, determines that work is being handed *to someone else*. This disposes of any semantic information that may be gleaned from comparison of the methods involved. By establishing

a similarity between the calling and target methods, a tighter conceptual relationship can be inferred between the two objects and methods.

5.6 Methods and Types

Now that I have defined the four basic relationships between method calls as EDP concepts as well as ρ-calculus formalizations, I will add in other relationships involving relative typing: same, or similar, type, subtype, and sibling. Similar types are those that adhere to some measure of similarity as outlined in Section 4.5. A subtype relation is most commonly defined as with the ς-calculus subtyping operator, as with *Subtype* $<:$ *Supertype*. A sibling type is one that shares a common ancestor type with the reference type, but is not also in a similar or subtyping relation.

As with the above definitions, I will start with a snippet of example code, such as in Figure 5.3.

```
object anObject:
        method callingMethod:
                anotherObject.x();
        ...
object anotherObject:
        method callingMethod:
                ...;
        method targetMethod:
                ...;
```

Figure 5.3: Example code for convolving methods calls and typing relationships

This is a slightly more abstract version from Section 5.5, where I stated that no relationship between the types of `anObject` and `anotherObject` was being defined. I will now do so for each of the three cases listed above.

If `anObject` and `anotherObject` are two instances of the same type, such that *anObject* : *theType* and *anotherObject* : *theType*, more refined versions of the previous four method-call EDPs can be inferred. In fact, the four combine to create two new EDPs.

5.6.1 DelegatedConglomeration

Starting with the replacement of `x` with `targetMethod` as before, then the method call constituted a **DelegatedConglomeration**. It is a **Delegate** in the sense that it is "another object, and a dissimilar method", but it is a **Conglomeration** in that "a dissimilar method is being called on an object of the

same type". In **Conglomeration**, that similarity of type was established by object similarity, but in this case the information is provided through other means.

$$\frac{\begin{array}{l} del : DelConglomerator \\ conglomTarget : DelConglomerator \\ del.operation <_{\mu}^{-\cdot -} conglomTarget.operation2 \end{array}}{\textbf{DelegatedConglomeration}(del, conglomTarget, operation, operation2)} \tag{5.10}$$

5.6.2 RedirectedRecursion

Similarly, **Redirect** and **Recursion** combine to form a new EDP, **RedirectedRecursion** if `x` is replaced by `callingMethod`. It combines the dotright similarity of **Recursion** with the leftdot dissimilarity of **redirect**, and ties them conceptually by requiring the two distinct objects to have similarity of type, as defined in Equation 5.11. This turns out to be a core element in Bobby Wolff's Object Recursion pattern, as will be shown in Section 6.3.

$$\frac{\begin{array}{l} rec : Recursor \\ redirectTarget : Recursor \\ rec.operation <_{\mu}^{-\cdot +} redirectTarget.operation \end{array}}{\textbf{RedirectedRecursion}(rec, redirectTarget, operation)} \tag{5.11}$$

A more common typing relation is subtyping, or **Inheritance**. If such a relationship is given between the types of the two objects, such that $anObject : Subclass$ and $anotherObject : Superclass$ and $Subclass <: Superclass$, delegation and redirection can be restricted to a particular family of types.

5.6.3 DelegateInFamily

DelegateInFamily is created when the target object of a **Delegate** is of a type that is a supertype to the current object. This restricts the method call to a known group of related types by targeting the root of some tree of subtypes. Many generic **Delegate** instances will conform to this, but only the **DelegateInFamily** form can know for certain that it is accomplishing this, due to the originating object also being a part of the familial tree of types.

$$\frac{\begin{array}{l} Delegator <: FamilyHead, \\ d : Delegator, \\ fh : FamilyHead, \\ d.operation <_{\mu}^{-\cdot -} fh.operation2, \end{array}}{\textbf{DelegateInFamily}(d, fh, operation, operation2)} \tag{5.12}$$

5.6.4 RedirectInFamily

Like **DelegateInFamily**, **RedirectInFamily** restricts its method invocation to members of a particular family of types, of which itself is a member. This redirection form tends to be used a bit more frequently than its delegation cousin. For instance, in the **Composite** pattern (Gamma et al., 1995), each of the `ConcreteComposite` instances holds a reference to one or more objects whose type is only known locally as `Composite`, a superclass to `ConcreteComposite`. Polymorphism takes care of the rest. **RedirectInFamily** is therefore a key component of **Composite**.

$$\frac{\begin{array}{l} Redirecter <: FamilyHead, \\ r : Redirecter, \\ fh : FamilyHead, \\ r.operation <_{\mu}^{-\cdot +} fh.operation, \end{array}}{\textbf{RedirectInFamily}(r, fh, operation)} \tag{5.13}$$

5.6.5 DelegateInLimitedFamily

By extending the type relationship to include the concept of two types being *siblings*, I capture another layer of interesting behavior. These are very similar to the two above defined EDPs, but send the invocation message to another object with whom they share a common ancestor type. In some systems, where a root type is unavoidable, certain filtering would be necessary to ensure the relevance of these EDPs.

$$\frac{\begin{array}{l} Delegator <: FamilyHead, \\ Sibling <: FamilyHead, \\ Delegator \neq Sibling, \\ Delegator \not<: Sibling, \\ d : Delegator, \\ sib : Sibling, \\ d.operation <_{\mu}^{-\cdot -} sib.operation2, \end{array}}{\textbf{DelegateInLimitedFamily}(d, sib, operation, operation2)} \tag{5.14}$$

DelegateInLimitedFamily is very closely related to **DelegateInFamily**. The former restricts the target object of the method invocation to one whose type is related to the current object only as a sibling. As can be seen, a sibling is properly defined as one who shares a common ancestor type, but is not the same type, nor is the subling a strict supertype.

Note that the *FamilyHead* is *not* listed in the EDP instance description. This is because it simply does not matter in most cases, and it prevents the creation of large numbers of spurious EDP productions

in cases such as shows in Figure 5.4. If the $FamilyHead$ were explicitly listed, then an instance would be created for each of $Root$, $Sub1$ and $Sub2$, even though the EDP only truly occurs once.

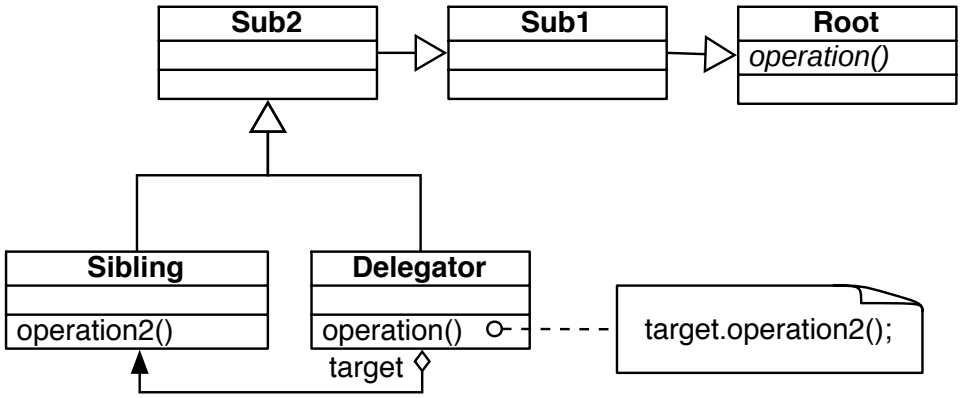


Figure 5.4: DelegateInLimitedFamily in a class tree

5.6.6 RedirectInLimitedFamily

RedirectInLimitedFamily performs the same limiting of scope of polymorphism as found in the EDP **DelegateInLImitedFamily**. As with the other **Delegate**/**Redirect** pairs of EDPs, this adds a method similarity to the mix.

$$\frac{\begin{array}{l} Redirecter <: FamilyHead, \\ Sibling <: FamilyHead, \\ Redirecter \neq Sibling, \\ Redirecter \not<: Sibling, \\ r : Redirecter, \\ sib : Sibling, \\ r.operation <_{\mu}^{-\cdot+} sib.operation, \end{array}}{\mathbf{RedirectInLimitedFamily}(r, sib, operation)} \tag{5.15}$$

5.7 Methods and Supertypes

In the last section an assertion was made that `anObject` and `anotherObject` were distinct instances, regardless of their type relationship. Investigating if they are in fact the same object, as I started with when discussing **Conglomeration** versus **Recursion**, brings the discussion full circle, and ties together all three axes of similarity neatly: method, object, and type.

If there is only object involved, and yet the target object is taken to be of a different type, new opportunities open up. The most common of these is the use of the *super* construct. As defined in ς-

calculus, *super* is a selector application that retrieves a superclass' implementation of the selected item. It is a precise way to bypass method overriding in subtyping, and most object-oriented programming languages support this directly. For example, C++ uses a statically typed `Superclass::method` syntax, while Smalltalk and Objective-C define it as `[super method]`, allowing for dynamic lookup of the proper type. In all cases, ς-calculus defines the selection of a superclass' method l from superclass c as $c\hat{}l(x)$.

Referring back to Figure 5.1, an analogue for using the `super` construct can be shown as in Figure 5.5.

```
object anObject:
        method callingMethod:
                super.x();
        method possibleTargetMethod:
                pass;
```

Figure 5.5: Example super-referential object and method call

5.7.1 ExtendMethod

A common use of directly accessing a superclass' implementation of a method is to add additional behavior to that method, to extend the method in some manner. It may be to fix a bug, change a default, or otherwise provide an alteration of the characteristics of the method, but in all cases it is an example of reusing existing code. When a method is extended in this way, a specific interaction of object similarity, type dissimilarity, type subtyping, and method similarity create the **ExtendMethod** EDP. This is a core piece of the **Decorator** pattern.

$$\frac{\begin{array}{l} operation \in \mathbf{meth}(OriginalBehavior), \\ ExtendedBehavior <: OriginalBehavior, \\ eb : ExtendedBehavior, \\ eb.operation <_{\mu}^{+.+} eb\hat{}operation \end{array}}{\mathbf{ExtendMethod}(OriginalBehavior, ExtendedBehavior, operation)} \tag{5.16}$$

5.7.2 RevertMethod

Not quite as common, but no less useful, is the ability to call a dissimilar method from the implementation of a `super` object or class. This *reverts* back to a prior implementation, bypassing the currently defined method in the instantiated type.

$$\frac{\begin{array}{l} OriginalBehavior : [l_i : {B_i}^{i \in 1 \ldots m}, operation : B_{m+1}, operation2 : B_{m+2}], \\ RevertedBehavior : [l_i : {B_i}^{i \in 1 \ldots m}, operation : B_{m+1}, operation2 : B_{m+2}], \\ RevertedBehavior <: OriginalBehavior, \\ rb : RevertedBehavior, \\ rb.operation <_{\mu}^{+,-} rb \wedge operation2 \end{array}}{\textbf{RevertMethod}(OriginalBehavior, RevertedBehavior, operation, operation2)} \quad (5.17)$$

This brings the discussion full circle, back to focussing on method calls to the same object, but in a new way. In the prior twelve EDP definitions based on method calls alone, I have demonstrated how a small number of well-formed and orthogonal principles of design can build into rather complex conceptual abstractions with little effort. Note that this has been a fleshing out of only the method-call based EDPs, those reliant on the $<_{\mu}$ reliance operator. The other three reliance operators were not treated in such a manner, but it is my belief that applying the same principles to them will produce equally robust and useful EDPs. In the next section I will discuss why the method-call EDPs were singled out for this treatment.

5.8 Examination of design patterns

A valid criticism of the EDP catalog is that it appears to be a formalization pushed onto the design pattern literature, breaking one of the tenets of design patterns: community consensus of existing solutions. I look forward to the inevitable discussion that the EDPs will create, and hope to have that community input in the following months and years. As a first approximation for validity, however, it is reasonable to inspect currently defined design patterns, and verify the existence of these solutions. It was this process of investigatory decomposition and cross-referencing that led to the creation of the EDP catalog in the first place, with ρ-calculus created later to provide a formal basis. I have reversed this ordering in this document to facilitate the growth of the concepts from extremely fine-grained to larger abstractions, and will continue to do so to build back to the more useful design patterns in the next chapter. It is necessary at this point, however, to visit the genesis of EDPs to provide validity to the catalog, and demonstrate that these are not formalizations to which patterns are expected to adhere, but rather that they are distilled from the existing literature. It is also only at this point that I have the proper language and notations to discuss the analysis effectively.

A natural place to start is the ubiquitous Gang of Four text (Gamma et al., 1995). Instead of a purely structural inspection, I chose to attempt to identify common concepts used in the patterns. A first cut of analysis resulted in eight identified probable core concepts:

AbstractInterface An extremely simple concept: a programmer wishes to enforce polymorphic behavior by requiring all subclasses to implement a method. Equivalent to Woolf's Abstract Class pattern (Woolf, 1998a), but on the method level. Used in most patterns in the GoF group, with the exception of Singleton, Facade, and Memento.

DelegatedImplementation Another ubiquitous solution, moving the implementation of a method to another object, possibly polymorphic. Used in most patterns, a method analog to the C++ *pimpl* idiom (Coplien, 1998).

ExtendMethod A subclass overrides the superclass' implementation of a method, but then explicitly calls the superclass' implementation internally. It extends, not replaces, the parent's behavior. Used in Decorator.

Retrieval Retrieves an expected particular type of object from a method call. Used in Singleton, Builder, Factory Method.

Iteration A runtime behavior indicating repeated stepping through a data structure. May or may not be possible to create an appropriate pattern-expressed description, but it would be highly useful in such patterns as Iterator and Composite.

Invariance Encapsulate the concept that parts of a hierarchy or behavior do **not** change. Used by Strategy and Template Method.

AggregateAlgorithm Demonstrate how to build a more complex algorithm out of parts that do change polymorphically. Used in Template Method.

CreateObject Encapsulates creation of an object, extremely similar to Ó Cinnéide's Encapsulate Construction minipattern (Ó Cinnéide, 2001). Used in most Creational Patterns.

Of these, AbstractInterface, DelegatedImplementation and Retrieval could be considered simplistic, while Iteration and Invariance are, on the face of things, extremely difficult.

5.8.1 Method calls

On inspection, five of these possible patterns are centered around some form of method invocation. This led me to investigate what the critical forms of method calling truly are, and whether they could provide insights towards producing a comprehensive collection of EDPs. I assumed, for the sake of this investigation, a dynamically bound language environment, and made no assumptions regarding features

Ownership	Obj Type	Method Type	Abstract	Used In
N/A	self	diff	Y	Template Method, Factory Method
N/A	super	diff		Adapter (class)
N/A	super	same		Decorator
held	parent	same	Y	Decorator
held	parent	same		Composite, Interpreter, Chain of Responsibility
ptr	sibling	same		Proxy
ptr/held	none	none	Y	Builder, Abstract Factory, Strategy, Visitor
held	none	none	Y	State
held	none	none		Bridge
ptr	none	none		Adapter (object), Observer, Command, Memento
N/A				Mediator, Flyweight

Table 5.1: Method calling styles in Gang of Four patterns

of implementation languages. Categorizing the various forms of method calls in the GoF patterns can be summarized as in Table 5.1, grouped according to four criteria:

Assume that an object a of type A has a method f that the program is currently executing. This method then internally calls another method, g, on some object, b, of type B. The columns represent, respectively, how a references b, the relationship between A and B, if any, the relationship between the types of f and g, whether or not g is an abstract method, and the patterns that this calling style is used in. Note that this is all typing information that is available at the time of method invocation, since we are only inspecting the types of the objects a and b and the methods f and g. Polymorphic behavior may or may not take part, but we are not attempting a runtime analysis. This is strictly an analysis based on the point of view of the calling code.

By eliminating the ownership attribute, the table vastly simplifies, as well as reducing the information to strictly type information. In a dynamic language, the concept of ownership begins to break down, reducing the question of access by pointer or access by reference to a matter of implementation semantics in many cases. By reducing that conceptual baggage in this particular case, such traits can be reintroduced later as needed. Similarly, other method invocation attributes could be assigned, but do not fit within the typing framework for classification. For instance, the concept of constructing an object at some point in the pattern is used in the Creational Patterns: Prototype, Singleton, Factory Method, Abstract Factory, and Builder, as well as others such as Iterator and Flyweight. This reflects the CreateObject component, but it can be placed aside for now to concentrate on the typing variations of method calls.

At this time, I can reorganize Table 5.1 slightly, removing the Mediator and Flyweight entry on the

Obj Type	Method Type	Abstract	Used In
self	diff	Y	Template Method, Factory Method
super	diff		Adapter (class)
super	same		Decorator
parent	same	Y	Decorator
parent	same		Composite, Interpreter, Chain of Responsibility
sibling	same		Proxy
none	none	Y	Builder, Abstract Factory, Strategy, Visitor, State
none	none		Adapter (object), Observer, Command, Memento, Bridge

Table 5.2: Reduced method calling styles in Gang of Four patterns

	Obj Type	Method Type	Used In
1	self	diff	Template Method, Factory Method
2	super	diff	Adapter (class)
3	super	same	Decorator
4	parent	same	Composite, Interpreter, Chain of Responsibility, Decorator
5	sibling	same	Proxy
6	none	none	Builder, Abstract Factory, Strategy, Visitor, State, Adapter (object), Observer, Command, Memento, Bridge

Table 5.3: Final method calling styles in Gang of Four patterns

last line, as no typing attributable method invocations occur within those patterns. The result, shown in Table 5.2, is a list of eight method calling styles. Note that four of these are simply variations on whether the called method is abstract or not. By identifying this as an instance of the AbstractInterface component from above, and considering this as an orthogonal issue, this list can be further simplified to a final collection of the six primary method invocation styles in the GoF text, shown in Table 5.3. I will demonstrate later how to reincorporate AbstractInterface to rebuild the calling styles used in the original patterns.

A glance at the first column reveals that it can be split into two larger groups, those which call a method on the same object instance ($a = b$) and those which call a method on another object ($a \neq b$).

The method calls involved in the GoF patterns now can be classified by three orthogonal properties:

- The relationship of the target object instance to the calling object instance.
- The relationship of the target object's type to the calling object's type
- The relationship between the method signatures of the caller and callee

5.8.2 Relationships

In retrospect, it is obvious that the above relationships involve an interaction between exactly *two* items. Programming languages focus on the description of entities (such as class, type, method, and field), while EDPs describe primary *relationships* between two entities. This is a core contribution of the EDP methodology, and what allows the formalization of it in the ρ-calculus. These are simple interactions between exactly two entities.

As such, much semantic information is derivable from this interaction, if we compare the *relative* known information of the entities under consideration. This frees the inspector from having to know any particulars of the actual types, methods or obejcts in question, vastly simplifying analysis both at the formal and practical levels. This leaves three cases of relative interaction in each of the method calls: object instances, object types, and method types (or signatures). This known *relative* information is where the concept of similarity arises.

Object similarity

Instances are simple to distinguish; they are either the same instance of an object or they are not. This simple difference can be exemplified by inspecting and comparing an external call and a recursive call.

```
Foo::methodA() {
        Foo f;
        f.methodA();
}
```

Figure 5.6: External call

```
Foo::methodA() {
        this->methodA();
}
```

Figure 5.7: Recursive call

The types of the two objects in each interaction are the same, the method name is the same. The only difference is that, in the second case the instance of the target object is the same as the calling object, commonly called 'self' or ''this'. While a simple distinction, this enables two highly different and powerful concepts, **Redirection** and **Recursion**.

Other advanced forms of object similarity can be envisioned, especially those for environments where object fragments can be composed at runtime into rich objects. Other constructs such as *Child* are another possibility here, and a call to *Same* maps to BETA's *inner*, for example. Within the Gang of Four patterns, however, a simple 'same or not' comparison suffices. For the purpose of this analysis, objects are either the same ($Self$) or not ($Other$).

Object type similarity

As with objects, analysis of types for similarity or relationships is relatively simple, at least as they are used in the Gang of Four patterns. I mentioned briefly in Section 4.5 that type similarity could be extended past the simplistic equivalence comparison, to include such relationships as 'contained within same package', or 'similarly named, non-colocated'. The Gang of Four patterns do not contain such details, however, and simple equivalence proves sufficient, when combined with any subclassing information contained within the system.

Equivalence, or similarity, of type is left up to the characteristics of the language and environment being analyzed. If two types are the same type within an environment, then they are equivalent: $A = B$. It is up to the environment to determine what 'the same type' is defined as. It may be dynamically determined, such that two types that are structurally identical but defined in very different subsystems are considered equivalent. It may be that scoping is taken into consideration, as in statically typed languages, such that formally equivalent types are considered distinct if they are defined in disparate packages. By allowing the environment to determine similarity of types, it becomes a powerful feature for code analysis by ρ-calculus. Most Gang of Four patterns seem to have an implicit understanding that typing is static, and therefore it is appropriate to use that determination here.

A definition for the *subtyping* relationship is similarly left to the environment, but here there is assistance from most object-oriented programming languages by way of the subclassing or subtyping construct. Commonly, this is **Inheritance**, and I will use the ς-calculus notation in the subsequent discussion: $A <: B$.

A third possibility arises, that of a *sibling* relationship. If A and B both derive from a common ancestor type C, but A and B have no such subtyping relationship, then they are sibling types: $A <: C,\ B <: C,\ A \not<: B,\ B \not<: A$. This is seen in the **Proxy** pattern.

An interesting but common situation arises when an equivalent object is combined with a subtyping relationship: the *super* construct. Ubiquitous in most object-oriented programming languages, it allows direct access to a superclass' definitions that have been perhaps overridden by a subclass, from within that subclass. Already the application of only object similarity and object type similarity has created powerful concepts and elements of software design.

Method similarity

As discussed in Section 4.5, one departure that this work takes from most design analysis is the concept of *similarity* of method signatures, and inferred semantic information from that property. When a method

body (*m1*) calls a second method (*m2*), we can deduce some basic assumptions regarding the relationship between the two methods from their signatures without understanding what those signatures stand for.

Take for example two methods named *calculateInterest*. They may be in different and, in the system at hand, unrelated types of objects, such as one in PersonalBankAccount and another in HomeMortgageLoan, but if they are intertwined in a body of code by means of a method call, we can state that we believe that they are related in their computational functionality. This is a consequence of proper application of Kent Beck's **IntentionRevealingSelector** pattern (Beck, 1997), which states "Name methods after what they accomplish." Most developers do this as a matter of course, to some extent, and we can use this to our advantage in determining the *relative* intent of methods.

```
PipeFitting::join( PipeFitting* otherPiece, Plumber* byThisGuy ) {
        if (!(byThisGuy->unionMember())) {
                PlumbersUnion::join(byThisGuy);
        }
        this->connectedTo.add(otherPiece);
}
```

Figure 5.8: Lexicographic similarity not indicative of intent

Of course, sometimes this may work against us. Consider two classes in a plumbing simulation, `PipeFitting`, and `PlumbersUnion`, where `PipeFitting::join` is defined as in Figure 5.8. Not a particularly well designed bit of code, but valid. Before the `PipeFitting` instance is joined, the current state of the calling `Plumber` is checked, and if they are not a member of the local union, they must join one before the job can be completed. In this case, the two instances of a method with the selector name of *join* have completely different intents. We can distinguish between them by inspecting their arguments, but this approach can break other cases where we do wish to retain an inferred conceptual relationship.

Despite some well-defined and, therefore, minimizable drawbacks, this similarity of method types, a relationship between two methods, is of prime importance in determining the intent of code. It frees us from having to understand the actual functionality of the methods, or perform any semantic analysis of the wording of the name, we merely have to check for a relative relationship between method names. Lexicographic equivalence is one such relationship, and checking this is a simple task. Methods therefore have either similarity, or not, much as objects.

Combining this with the relative typing of objects, we find that we can extract much about how two objects or methods relate to one another, and create a table of the various possible interactions.

5.8.3 Method call EDPs

For the purposes of this discussion, object similarity can be reduced to simply a dichotomy between $Self$ and $Other$. Object type similarity describes the relationship between A and B, if any, and method similarity compares the types (consisting of a function mapping type, F and G, where $F = X \rightarrow Y$ for a method taking an object of type X and returning an object of type Y) of f and g, simply as another dichotomy of equivalence. From these three axes, a rich conceptual framework can be created on which to hang the results of the Gang of Four pattern analysis above.

It is illustrative at this point to attempt creation of a comprehensive listing of the various permutations of these axes, and see where the identified invocation styles fall into place. For the possible relationships between A and B, I have started with the list items of 'Parent', where $A <: B$, 'Sibling' where $A <: C$ and $B <: C$ for some type C, and 'Unrelated' as a collective bin for all other type relations at this point. To these I add 'Same', or $A = B$, as an obvious simple type relation between the objects. $Child$ is possible here as an addition as well, although I do not do so at this time.

Initial list

On filling in the invocation styles from our final list from the GoF patterns, is is possible to map them to the six categories in Table 5.8.3. Each of these captures a concept as much as a syntax, as we originally intended. Each expresses a direct and explicit way to solve a common problem, providing a structural guide as well as a conceptual abstraction. In this way they fulfill the requirements of a pattern, as generally defined, and more importantly, given a broad enough context and minimalist constraints, fulfill Alexander's original definition as well as any decomposable pattern language can (Alexander, 1964). I treat these as meeting the definition of design patterns, and present them as such.

The nomenclature I have selected is a reflection of the intended uses of the various constructs, but requires some defining:

Conglomeration Aggregating behavior from methods of $Self$. Used to encapsulate complex behaviors into reusable portions within an object.

ExtendMethod A subclass wishes to extend the behavior of a superclass' method instead of strictly replacing it.

RevertMethod A subclass wants *not* to use its own version of a method for some reason, such as a namespace clash.

1. Self ($a = b$)
   (a) Self ($A = B$, or $a = this$)
      i. Same ($F = G$) .....
      ii. Different ($F \neq G$) ..... Conglomeration[1]
   (b) Super ($A <: B$, or $a = super$)
      i. Same ($F = G$) ..... ExtendMethod[3]
      ii. Different ($F \neq G$) ..... RevertMethod[2]
2. Other ($a \neq b$)
   (a) Unrelated
      i. Same ($F = G$) ..... Redirect[6]
      ii. Different ($F \neq G$) ..... Delegate[6]
   (b) Same ($A = B$)
      i. Same ($F = G$) .....
      ii. Different ($F \neq G$) .....
   (c) Parent ($A <: B$)
      i. Same ($F = G$) ..... RedirectInFamily[4]
      ii. Different ($F \neq G$) .....
   (d) Sibling ($A <: C, B <: C, A \not<: B, B \not<: A$)
      i. Same ($F = G$) ..... RedirectInLimitedFamily[5]
      ii. Different ($F \neq G$) .....

Figure 5.9: Initial List of method-call relationships from GoF patterns

Redirect A method wishes to redirect some portion of its functionality to an extremely similar method in another object. I chose the term 'redirect' due to the usual use of such a call, such as in the Adapter (object) pattern.

Delegate A method simply delegates part of its behavior to another method in another object.

RedirectInFamily Redirection to a similar method, but within one's own inheritance family, including the possibility of polymorphically messaging an object of one's own type.

RedirectInLimitedFamily A special case of the above, but limiting to a subset of the family tree, excluding possibly messaging an object of one's own type.

The full list

Attention should now be paid to those slots that have not been filled with items from the Gang of Four patterns. Again, I will present the listing, and briefly discuss each of the new items in turn.

1. Self ($a = b$)
   (a) Self ($a = this$)
      i. Same ($F = G$) ..... Recursion
      ii. Different ($F \neq G$) ..... Conglomeration
   (b) Super ($a = super$)
      i. Same ($F = G$) ..... ExtendMethod
      ii. Different ($F \neq G$) ..... RevertMethod
2. Other ($a \neq b$)
   (a) Unrelated
      i. Same ($F = G$) ..... Redirect
      ii. Different ($F \neq G$) ..... Delegate
   (b) Same ($A = B$)
      i. Same ($F = G$) ..... RedirectedRecursion
      ii. Different ($F \neq G$) ..... DelegatedConglomeration
   (c) Parent ($A <: B$)
      i. Same ($F = G$) ..... RedirectInFamily
      ii. Different ($F \neq G$) ..... DelegateInFamily
   (d) Sibling ($A <: C, B <: C, A \not<: B, B \not<: A$)
      i. Same ($F = G$) ..... RedirectInLimitedFamily
      ii. Different ($F \neq G$) ..... DelegateInLimitedFamily

Recursion The basic recursive method call behavior from introductory programming.

RedirectedRecursion A form of object level iteration.

DelegatedConglomeration Gathers behaviors from external instances of the current class.

DelegateInFamily Gathers related behaviors from the local class structure.

DelegateInLimitedFamily Limits the behaviors selected to a particular base definition.

In the introductory chapter to this document, I invoked Alexander's argument that conscious design was more efficient and led to better products than unconscious, reflexive design. Looking at the above list, I find it very intriguing that Recursion was not in the initial list of programming concepts extracted from the Gang of Four patterns. It is a programming approach used on a daily basis by engineers of all levels, and yet I can honestly say it never would have occurred to me to include it in a list of design principles, without performing the above analysis and looking for underlying commonalities of relationship, then attempting to produce a comprehensive coverage of the axes. The obvious and self-

evident elements of programming are precisely the level of detail that extensive analysis depends on, formalisms require to be effective, and yet they are the pieces most ignored in software analysis research.

5.8.4 Object and Type EDPs

At this point I have a fairly comprehensive array of method/object invocation relations, and will revisit the original list of concepts culled from the GoF patterns. Of the original eight, three are absorbed within the method invocations list: DelegatedImplementation, ExtendMethod, and AggregateAlgorithm. Of the remaining five, two are some of the more problematic EDPs to consider: Iteration, and Invariance. I will revisit these briefly in Section 5.10.

The remaining three concepts are found in the Object and Type EDPs. CreateObject, AbstractInterface and Retrieve deal with object creation, method implementation and object referencing, respectively. Detailed discussions of these core concepts are found earlier in this chapter, and in Appendix B.

5.9 Isotopes

Common wisdom holds that formalization of patterns in a mathematical notation will inevitably destroy the flexibility and elegance of patterns by reducing them to mere recipes and eliminating much of their usefulness. An interesting side effect of expressing the EDPs in ρ-calculus, however, is an *increased* flexibility in expression of code while conforming to the core *concept* of a pattern. This encapsulation of concepts is related to the encapsulation of implementation at the core of object-oriented programming.

Consider the class diagram for the structure of **RedirectInFamily**, in Figure 5.10. Taken literally, it specifies that a class wishes to invoke a similar method (where, again, similarity is evaluated based on the function types of the methods) to the one currently being executed, and it wishes to do so on an object of a its parent class' type. This sort of open-ended structural recursion is a part of many patterns.

If we take the Participants specification of **RedirectInFamily**, as described in the pattern in Appendix B, we find that:

- FamilyHead defines the interface, contains a method to be possibly overridden.
- Redirecter uses interface of FamilyHead through inheritance, redirects internal behavior back to an instance of FamilyHead to gain polymorphic behavior over an amorphous object structure.

Each of these requirements is expressible in ς-calculus, as in Equations 5.18 through 5.22. This is a concrete implementation of the **RedirectInFamily** structure, but one which fails to capture the

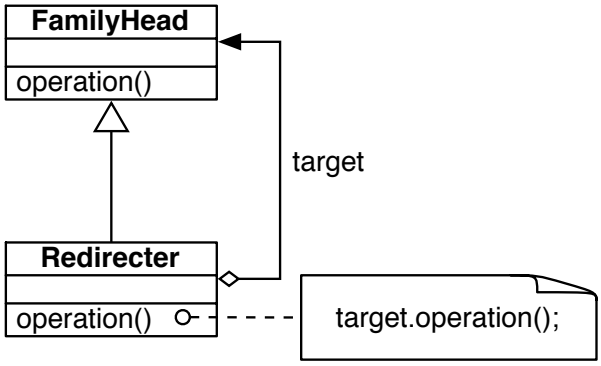


Figure 5.10: RedirectInFamily class structure

$$FamilyHead \equiv [operation : A] \tag{5.18}$$

$$Redirecter <: FamilyHead \tag{5.19}$$

$$Redirecter \equiv [fh : FamilyHead,\ operation : A = \varsigma(x_i)\{fh.operation\}] \tag{5.20}$$

$$r : Redirecter \tag{5.21}$$

$$r.fh : FamilyHead \tag{5.22}$$

reliance of *Redirecter.operation* on $FamilyHead.operation$'s behavior. So, I introduce the mu-form reliance operator:

$$r.operation <_{\mu}^{\circ,+} r.fh.operation \tag{5.23}$$

...and produce a necessary and sufficient set of clauses at this point to represent **RedirectInFamily**, as it was defined in Section 5.6.4:

$$\frac{\begin{array}{l} Redirecter <: FamilyHead, \\ r : Redirecter, \\ fh : FamilyHead, \\ r.operation <_{\mu}^{-,+} fh.operation, \end{array}}{\textbf{RedirectInFamily}(r, fh, operation)} \tag{5.24}$$

Consider now Figure 5.11, where I show what, at first glance, does not look much like the original specification. I have introduced a new class to the system, the static criteria that the subclass' method invoke the method of the superclass' instance is gone, and a new calling chain has been put in place. In fact, this construction looks quite similar to the transitional state while applying Martin Fowler's *Move Method* refactoring (Fowler, 1999).

I claim that this is precisely an example of an alternative form of **RedirectInFamily**, however, when viewed as a series of formal constructs, as follows, assuming the same class definitions given in

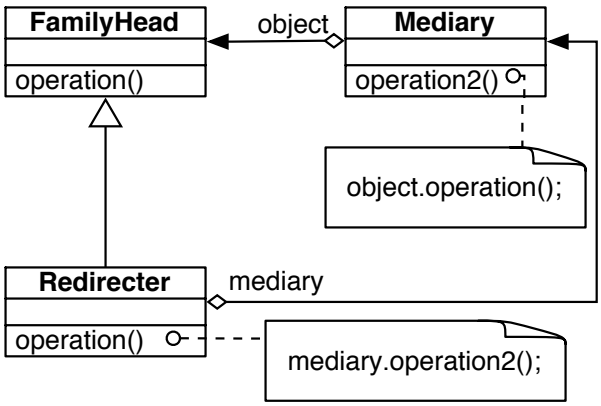


Figure 5.11: RedirectInFamily Isotope

Equations 5.18 and 5.20:

$$Redirection <: FamilyHead \tag{5.25}$$

$$r : Redirection \tag{5.26}$$

$$Mediary.object : FamilyHead \tag{5.27}$$

$$r.mediary : Mediary \tag{5.28}$$

$$r.operation <^{\circ\cdot -}_{\mu} r.mediary.operation2 \tag{5.29}$$

$$Mediary.operation2 <^{\circ\cdot -}_{\mu} Mediary.object.operation \tag{5.30}$$

If I start reducing this equation set, a new object-level clause can be derived from the typing for $r.mediary$ and Equation 5.30, resulting in Equation 5.31.

$$\frac{\begin{array}{c} r.mediary : Mediary, \\ Mediary.operation2 <^{\circ\cdot -}_{\mu} Mediary.object.operation \end{array}}{r.mediary.operation2 <^{\circ\cdot -}_{\mu} r.mediary.object.operation} \tag{5.31}$$

The transitive operations of Chapter 4 can then be performed on Equations 5.29 and 5.31:

$$\frac{\begin{array}{c} r.operation <^{\circ\cdot -}_{\mu} r.mediary.operation2, \\ r.mediary.operation2 <^{\circ\cdot -}_{\mu} r.mediary.object.operation \end{array}}{r.operation <_{\mu} r.mediary.object.operation} \tag{5.32}$$

This is almost the desired relationship, but indicates a more general **Delegate** operation instead of the **Redirect** being searched for. Fortunately, this is simple to fix.

The Dotright Similarity rule of Section 4.5, Equation 4.60, suggests the creation of the more specific

Redirect if the signatures of the operations match, from which it becomes evident that the needed form can indeed be derived:

$$\frac{r.operation <_{\mu} r.mediary.object.operation}{r.operation <_{\mu}^{\circ,+} r.mediary.object.operation} \tag{5.33}$$

Taking Equations 5.18, 5.20, 5.25, 5.26, 5.27, and 5.33, it is straightforward to satisfy the clause requirements set in the definition of **RedirectInFamily**, as per Equation 5.24. This alternate structure is an example of the **RedirectInFamily** pattern, without adhering to a strict class structure. I term this situation, where a variation on the original elemental design pattern class structure still conforms to the conceptual definition, as an *isotope*. The concepts of *object relationships and reliance* are the key.

The effect of this is that there is no explicit requirement that the relationships between *Redirecter* and *Mediary* or *Mediary* and *FamilyHead* be **Redirection** EDPs. In fact, they can be **Delegation** expressions, as I discuss above, with no change to the meaning of **RedirectInFamily**. Only the initial and terminal function signatures are important.

More generally, this is the key to the power of analysis with SPQR: combining the formal transitivities of ρ-calculus with the recognition that design patterns are encapsulation of *concepts*, and allowing the implementation of those concepts to vary within well-formed guidelines. This takes the same fundamentally important principles of encapsulation from object-oriented programming that have resulted in better designed systems, and hoists them up to a more abstract level, one of programming concepts. Without the formalizations of ρ-calculus, this approach would be extremely difficult to validate or provide guidance on. Without the conceptual encapsulations, design patterns become mere structural recipes and lose much of their expressiveness and importance. The two together provide the strong foundation of a reproducible, formalized science of design principles for software engineering.

It is worth noting that, while this may superficially seem to be equivalent to the common definition of *variant*, as defined by Buschmann (Buschmann et al., 1996), there is a key difference. Buschmann's variants are defined such that, "From a general perspective a pattern and its variants describe solutions to very similar problems." (Buschmann et al., 1996, p. 16) Isotopes solve the *same* problem, but in a slightly different manner, perhaps due to forces from an existing architecture. Variants solve *similar* problems, and in similar ways.

The distinction is that of the above-mentioned encapsulation. Isotopes may differ from the strict pattern structure in their implementation, but they provide fulfillment of the various roles required by the pattern, and the relationships between those roles are kept intact. From the view of an external calling body, the pattern is precisely the same no matter which isotope is used. In the above example, Figure 5.11

still shows *FamilyHead* and *Redirecter* fulfilling their assigned roles in the **RedirectInFamily** EDP. *Redirecter.operation* still performs the required conceptual task; how it implements it is immaterial, as per the principles of encapsulation.

Variants, on the other hand, do not have this restriction, and, indeed, of the 45 variants offered by Buschmann for his initial 16 design patterns, only two, Microkernel/Distributed Microkernel System (Buschmann et al., 1996, pg.188) and Whole-Part/Shared Parts (Buschmann et al., 1996, pg.233), can be said to properly meet the requirement of an isotope. The others all have fundamental changes that alter how they interact with the remainder of the system.

Variants are not interchangeable without retooling the surrounding code, but isotopes are. This is an essential requirement of isotopes, and precisely why the term was chosen. Isotopes, in the EDP realm, are analogous to isotopes in chemistry, where they are externally identical when interacting in chemical processes. Only an inspection of their nuclei provides a clue as to their differing nature.

This flexible internal representation allows the implementer a great degree of latitude in system design, while still conforming to the abstractions given by design patterns.

5.10 Future Definitions

As stated in Section 5.8, the two concepts of Iteration and Invariance as derived from the Gang of Four patterns were expected to be problematic, and, indeed, they have been. Both encapsulate concepts that are simultaneously highly abstract while intimately tied to the behavior of λ-calculus. Other possible EDPs exist also. The purpose of this section is merely to illustrate that they exist, but have neither been fully defined, nor validated. I include them here to entice the reader to explore a few possibilities that remain undiscovered.

5.10.1 Recurrence

At the end of the discussion concerning method-based EDPs, I mentioned that applying similar combinations of the orthogonal axes of similarity might lead to interesting new EDPs. The first of these can be found by expanding on the $<_\kappa$ reliance operator, such that full object similarity is established. This leads to a $<_\kappa^{+\cdot+}$ reliance operator, which may be best described as a new EDP corresponding to *recurrence* from more traditional analysis techniques (NIST, 1999).

5.10.2 Iteration

Iteration is an abstraction that has become so commonplace that it is now considered a first-class feature of many languages (van Rossum, 2003; Wall et al., 2000), and is ubiquitous in standard support libraries for many more (Accredited Standards Committee X3 (Information Processing Systems) – Working Group 21, 1996; Gosling et al., 2005). At the present time, Iterator has not been sufficiently analyzed with respect to ρ-calculus to obtain a clear direction for formalization in a meaningful manner. I am hopeful, however, that the $<_\sigma$ and $<_\kappa$ reliance operators will provide the proper analysis tools. For instance, iterators in most implementations contain a reference to the element in the container currently being inspected. The iterator is requested to move forward and back along its associated container. In this manner, it affects its own state. It is expected then to find that the iterator relies on itself in specific ways, through the reference. This could be detectable through the above described **Recurrence**.

5.10.3 Collection

An iterator is generally associated with a collection of elements to iterate over, and, therefore, a closely related concept that is likely to be co-defined with **Iterator** is that of a **Collection**. At this point it is not clear how this pattern should be formalized in ρ-calculus and the other EDPs.Because of the two concepts above that require detailed yet esoteric analysis of the inner runtime behavior of methods and data reliances, **Collection** is assumed at this time to be best detected by using semantic information already present in many languages. For instance, in C++ any use of `std::vector` would be tagged as fulfilling the main role of **Collection**. Likewise any use of `std::forward_iterator` would trigger the production of an **Iterator** design pattern instance. So much information can be found in standardized ways within languages and language libraries that it would be foolish not to leverage it wherever possible.

5.10.4 Invariance

Lastly, I revisit the final concept that was culled from the Gang of Four analysis: invariance. In first-order inference and analysis systems it is much easier to establish what is different, than to establish what must not vary. The current use of a first-order inference engine at the core of SPQR, as I will discuss in Section 9.2, limits this type of analysis at this time. I would expect to find that more traditional procedural analysis techniques such as dynamic invariant detection (Perkins and Ernst, 2004; Ernst et al., 2001) would find application here. After such analysis, the results could be added to an SPQR analysis as instance of the **Invariance** EDP.

5.11 Conclusion

The EDPs show how well-formed and methodical construction of core concepts of programming can be developed from a small number of relationship formalisms, as defined in ρ-calculus. These concepts are ubiquitous in any object-oriented system, by their very nature as elemental building blocks of design.

By expressing these concepts in a general form, instances of concepts found through a number of analysis approaches can be brought together to form the basis for a rich conceptually-based view of software systems. ρ-calculus provides one method for the most common analyses, but others may be introduced as applicable.

Ultimately, the expressive power of the EDPs is best demonstrated when they are brought together to compose larger, more abstract, concepts of programming, as I will show in the next chapter.

Chapter 6

Pattern Composition

With the foundations of ρ-calculus and Elemental Design Patterns in place, the same principles can be applied to compose much more complex design patterns. Just as the ρ-calculus elements were combined to form the EDPs, the EDPs can be combined, using ρ-calculus as the glue, to form the higher-level abstractions that engineers will be most interested in.

One such is the **Decorator** pattern (Gamma et al., 1995), and I will show how it can be expressed in ρ-calculus using intermediate design patterns to facilitate discussion and simplify the notation. These intermediate patterns are drawn from both my own research and the existing patterns literature. They are distinguished from the EDPs in that they rely on the EDPs, and generally involve more than the two programming elements comprising the binary relationships of the EDPs.

6.1 FulfillMethod

As mentioned in Section 5.2.2, and defined in Equation 5.2, **AbstractInterface** sets up a relationship between a supertype and an unnamed subtype. The **FullfillMethod** pattern names that subtype explicitly, providing the other conceptual half of this relationship. As with **AbstractInterface**, this pattern involves types exclusively. The addition of the new annotations to UML is designed to show which subpattern and role each element of the new structure satisfies. For instance, the *Abstractor* type in **FulfillMethod** appears in the *Abstractor* role of the **AbstractInterface** subpattern.

$$\frac{\begin{array}{l}\textbf{AbstractInterface}(\mathit{Abstractor}, \mathit{operation}_n),\\ \mathit{ConcreteClass} <: \mathit{Abstractor},\\ \mathit{ConcreteClass} \equiv [\mathit{operation}_i \Leftarrow b_i{}^{i \in 1..n-1, n+1..m}, \mathit{operation}_n \Leftarrow b_n]\end{array}}{\textbf{FulfillMethod}(\mathit{Abstractor}, \mathit{ConcreteClass}, \mathit{operation}_n)} \tag{6.1}$$

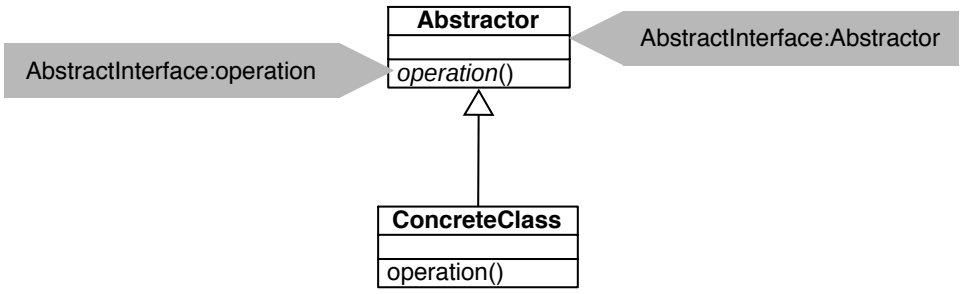


Figure 6.1: FulfillMethod intermediate pattern

6.2 Objectifier

It should be obvious by now that Objectifier is simply a class structure applying the **FulfillMethod** pattern to all methods in a class pair. This is equivalent to what Woolf calls an Abstract Class pattern. Referring back to Figure 2.1 from our earlier discussion in section 2.1.4, the core concept is to create a family of subclasses with a common abstract ancestor, as shown annotated in Figure 6.2. This is expressed in ρ-calculus for one such subclass as in Equation 6.2.

$$\frac{\begin{array}{l} ObjectifierBase : [l_i : {B_i}^{i \in 1 \ldots n}], \\ Client : [ref : Objectifier], \\ Client.someMethod <_\mu Client.ref.l_i, \\ \textbf{FulfillMethod}(ObjectifierBase, ConcreteObjectifier, l_i)^{i \in 1 \ldots n}, \end{array}}{\textbf{Objectifier}(ObjectifierBase, ConcreteObjectifier, Client)} \tag{6.2}$$

6.3 Object Recursion

I briefly described Object Recursion in section 2.1.4, and described its class structure in Figure 2.2. I now show that this is a melding of the Objectifier and RedirectInFamily patterns, as illustrated in Figure 6.3. The annotations indicate which roles of which patterns the various components of Object Recursion

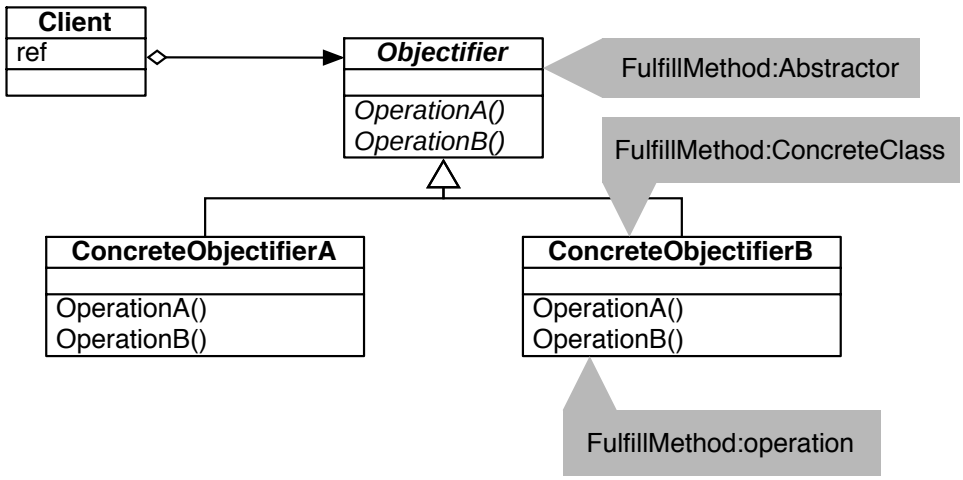


Figure 6.2: Objectifier pattern with annotation

play. A formal EDP representation is:

$$\frac{\begin{array}{c}\textbf{Objectifier}(Handler, {Recurser_i}^{i\in 1\ldots m}, Initiator),\\ \textbf{Objectifier}(Handler, {Terminator_j}^{j\in 1\ldots n}, Initiator),\\ init.someMethod <_\mu obj.handleRequest,\\ init : Initiator,\\ obj : Handler,\\ \textbf{RedirectInFamily}(Recurser, Handler, handleRequest),\\ \textbf{!RedirectInFamily}(Terminator, Handler, handleRequest)\end{array}}{\textbf{ObjectRecursion}(obj, {Recurser_i}^{i\in 1\ldots m}, {Terminator_j}^{j\in 1\ldots n}, init)} \tag{6.3}$$

6.4 Decorator

Now we can finally produce a pattern directly from the Gang of Four text, the **Decorator** pattern (Gamma et al., 1995, pg 175). It is simple enough to be defined from the ground up, illustrating my technique of using fully formal methods entrenched in ρ-calculus coupled with the elemental design patterns catalog to create rich and conceptually true formal descriptions of useful design patterns. **Decorator** is complex enough, however, to present a bit of a challenge, by adding a bit of behavioral elegance to a primarily structural pattern.

Figure 6.4 is the standard class diagram for **Decorator**. Figure 6.5 shows the same diagram, annotated to show how the **ExtendMethod** and **ObjectRecursion** patterns interact. Again, we provide a formal definition, where the keyword **any** indicates that any object of any class may take this role, as

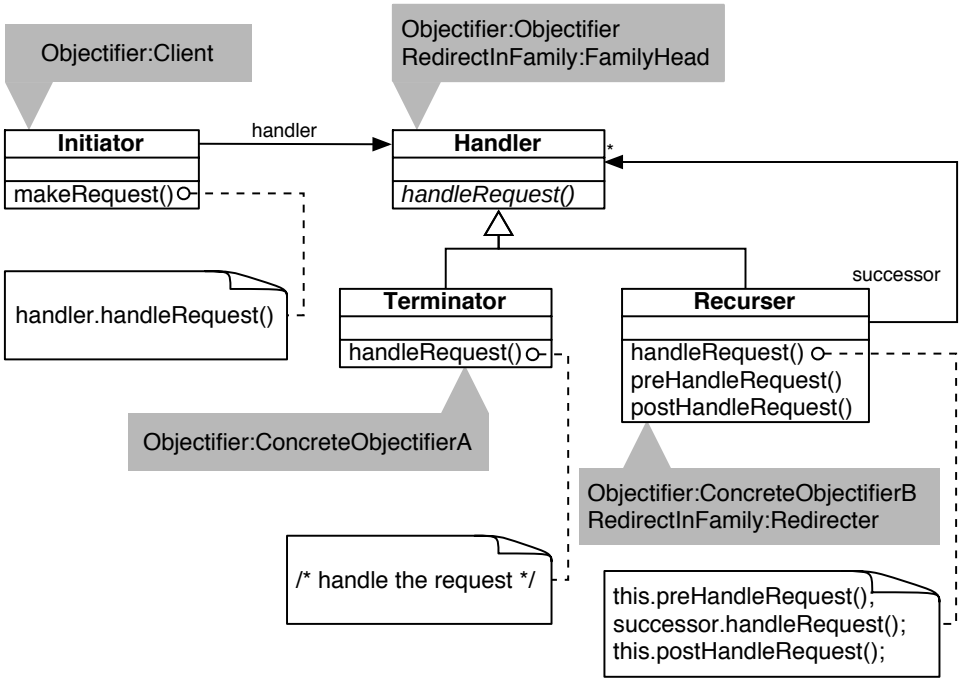


Figure 6.3: Object Recursion, annotated to show roles

long as it conforms to the definition of **ObjectRecursion**.:

$$\frac{\begin{array}{c}\textbf{ObjectRecursion}(Component, {Decorator_i}^{i\in 1\ldots m}, {ConcreteComponent_j}^{j\in 1\ldots n}, \textbf{any}),\\ \textbf{ExtendMethod}(Decorator, {ConcreteDecoratorB_k}^{k\in 1\ldots o}, {operation_k}^{k\in 1\ldots o}),\\ !\textbf{ExtendMethod}(Decorator, {ConcreteDecoratorA_l}^{l\in 1\ldots p}, {operation_l}^{l\in 1\ldots p})\end{array}}{\begin{array}{l}\textbf{Decorator}(Component, {Decorator_i}^{i\in 1\ldots m}, {ConcreteComponent_j}^{j\in 1\ldots n},\\ \quad {ConcreteDecoratorB_k}^{k\in 1\ldots o}, {ConcreteDecoratorA_l}^{l\in 1\ldots p},\\ \quad {operation_k}^{k\in 1\ldots o+p})\end{array}} \tag{6.4}$$

To illustrate the compactness of the EDP notation within ρ-calculus Equation 6.5 shows the same definition of **Decorator**, fully expanded to the raw ρ-calculus. It should be apparent that this larger collection of facts would be much more difficult to work with at the human level. The use of the EDPs and their compositions provide a much more comprehensible approach to understanding how patterns are defined conceptually, and provide an incremental path to detection and comprehension.

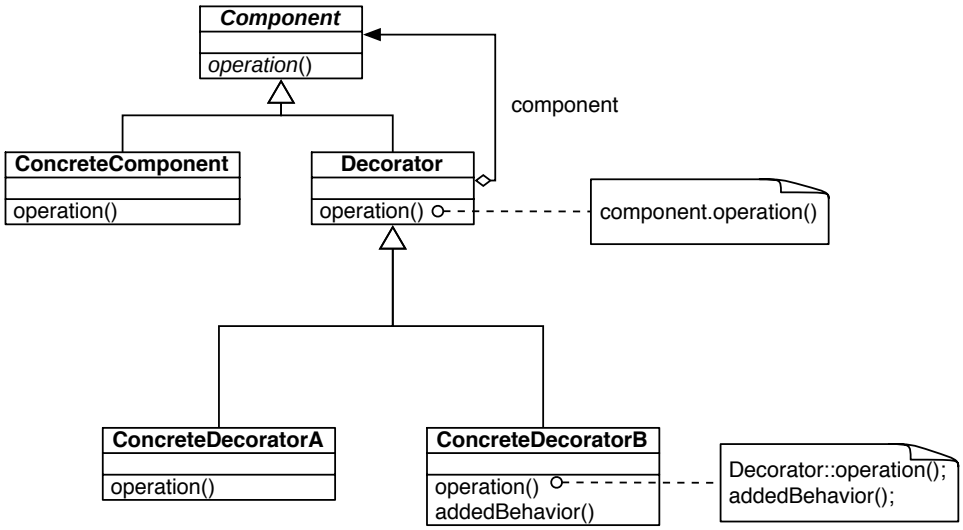


Figure 6.4: Decorator class structure

$$
\begin{array}{c}
Component : [l_i : {B_i}^{i\in 1\ldots n}], \\
Initiator : [component : Component], \\
Initiator.makeRequest <_\mu Initiator.component.operation, \\
E \vdash Component,\ Component : [l_i : B_i^{i\in 1..n}], \\
Component \equiv [l_i = b_i^{i\in 1..m-1,m+1..n}, l_m = []], \\
Decorator <: Component, \\
Decorator \equiv [operation_j \Leftarrow {b_j}^{j\in 1..n-1,n+1..m}, operation \Leftarrow b_i], \\
ConcreteComponent <: Component, \\
ConcreteComponent \equiv [operation_j \Leftarrow {b_j}^{j\in 1..n-1,n+1..m}, l_i \Leftarrow b_i], \\
Decorator.operation <_\mu^{-.+} Component.operation, \\
!ConcreteComponent.operation <_\mu^{-.+} Component.operation, \\
Decorator \equiv [operation_k \Leftarrow {b_k}^{k\in n'..n'+o}, {l'_k}^{k'\in 1..n'-1,n'+1..m'}], \\
{ConcreteDecorator_k}^{k\in 1\ldots o} <: Decorator, \\
{ConcreteDecorator_k.operation_k <_\mu^{+.+} Decorator\hat{}operation_k}^{k\in 1\ldots o}, \\
{ConcreteDecorator_l}^{l\in 1\ldots p} <: Decorator, \\
{!ConcreteDecorator_l.operation_l <_\mu^{+.+} Decorator\hat{}operation_l}^{l\in 1\ldots p} \\
\hline
\mathbf{Decorator}(Component, {Decorator_i}^{i\in 1\ldots m}, {ConcreteComponent_j}^{j\in 1\ldots n}, \\
{ConcreteDecoratorB_k}^{k\in 1\ldots o}, {ConcreteDecoratorA_l}^{l\in 1\ldots p}, \\
{operation_k}^{k\in 1\ldots o+p})
\end{array}
\tag{6.5}
$$

The above creates a formally sound definition of a description of how to solve a problem of software architecture design, but does so from first principles of the relationships between fundamental elements of programming. This definition is now subject to formal analysis, discovery, and metrics, and, following these examples of pattern composition, can be used as a building block for larger, even more intricate patterns that are *incrementally* comprehensible. At the same time, I believe this approach retains

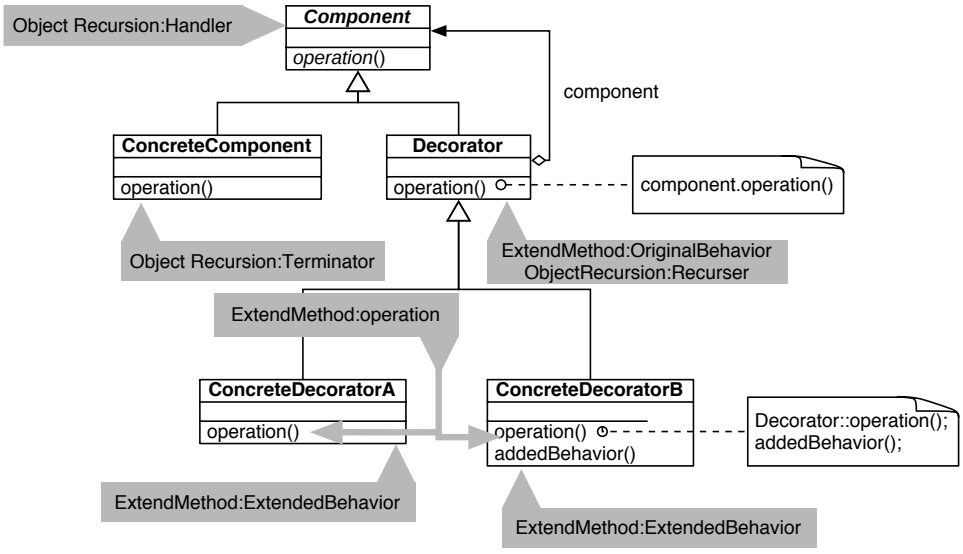


Figure 6.5: Decorator annotated to show EDP roles

the flexibility of implementation, as demonstrated with the **RedirectInFamily** isotope, that patterns demand. Also, I believe that the conceptual semantics of the pattern are intact, by making precise choices at each stage of the composition. Furthermore, by building this approach on an existing denotational semantics for object-oriented programming, ς-calculus, tools continue to be able to process the same system at an extremely low level. Cohesion and coupling analysis (Briand and Daly, 1997; Hitz and Montazeri, 1995; Kang and Bieman, 1996a; Kang and Bieman, 1996b; Samadzadeh and Khan, 1994), slice metrics production (Karstu, 2999; Ott, 1992; Ott and Thuss, 1989), and other traditional code analysis techniques (Chidamber and Kemerer, 1994; Demeyer et al., 2000; Ott and Thuss, 1993) are still completely possible within the combined calculi of $\lambda{+}\rho$-calculus. I have provided the link from patterns, as conceptual entity descriptions, to the formal semantics required and used by compilers and other traditional tools, without sacrificing the flexibility of implementation required by the patterns. I do not, however, see an explicit need always to resort to the full $\lambda{+}\rho$-calculus for all analysis. One of the key contributions of this system is that the practitioner can *choose* on which level to operate, and perform the analyses and tasks which are suitable without losing the possibility of integrating other layers of analysis at a later date.

6.5 Other Intermediates

Two other intermediate patterns refine how object retrieval can interact with object creation. This manner of collaboration provides a basic handle on the policies of object ownership management, such

as seen in the **Singleton** pattern (Gamma et al., 1995, pg 127).

6.5.1 RetrieveNew

When the requesting object requires exclusive access to a retrieved object, **RetrieveNew** should be used. It combines the creation of a new object with the return-by-value semantics to guarantee that the returned object has no other explicit references.

$$\frac{\begin{array}{c} o'.s' \overset{n}{\rightsquigarrow} x \\ \mathbf{CreateObject}(X, o'.s'.x) \\ \mathbf{Retrieve}(o.s, o'.s', x) \end{array}}{\mathbf{RetrieveNew}(Sink, object, target, Retrieved, Source, object', selected)} \tag{6.6}$$

6.5.2 RetrieveShared

Contrast **RetrieveNew** with **RetrieveShared**, where the object returned is done so via a reference. In such cases, exclusivity is *not* guaranteed, and, while proof of it being an explicitly shared resource is not easy to come by, it provides a conceptual basis for future work to provide refined definitions.

$$\frac{\begin{array}{c} o'.s' \overset{v}{\rightsquigarrow} x \\ \mathbf{CreateObject}(X, o'.s'.x) \\ \mathbf{Retrieve}(o.s, o'.s', x) \end{array}}{\mathbf{RetrieveShared}(Sink, object, target, Retrieved, Source, object', selected)} \tag{6.7}$$

6.6 GOF Patterns

Other Gang of Four patterns can be defined using the same techniques from the above composition of **Decorator**. Patterns that rely on the nascent **Iterator** and **Collection** patterns, as outlined in Section 5.10.2, are not easily definable using the current EDP catalog of Appendix B. The remainder of the Gang of Four patterns, defined in terms of the EDPs and ρ-calculus, can be found in Appendix D. Most of these are straightforward and self-explanatory, but a few require a note or two.

6.6.1 Singleton

Singleton was used as the example pattern definition in raw ς-calculus in Section 3.2. Here I give the ρ-calculus and EDP-based definition for contrast, in Figure 6.8. More straightforward and simpler to understand, **Singleton** also provides a great deal of flexibility in the use of the **RetrieveShared** pattern, which, in turn, is defined as reliance operators that provide the necessary transitivities.

The functional description of **Singleton** requires that there be one object of the type, and that there be one common interface for gaining that instance. In class based systems, this is most easily accomplished by using the class-object as the repository and interface for that type. In pure object systems, other approaches may be used, such as forbidding cloning of an existing object. Most implementations of **Singleton** concentrate on the access of the object. That is the conceptual model I follow here. In class-based languages, the use of the class-object allows the definition to have one unique object for access to the desired unique object. This apparently self-referential relationship may be a weakness in the current definition of **Singleton**. It may be more useful in the long run to concentrate on how the singleton object is created, and how its creation is forbidden. For instance, the privacy traits of various languages can be modeled within ς-calculus (Abadi and Cardelli, 1996, pgs102-103), providing a method by which to check for any possible external triggering of the type constructors. I have not implemented such privacy features in SPQR, however, so I will use the definition in Equation 6.8.

$$\frac{\begin{array}{l} client.method <_{\mu} singleton.gettor, \\ singletonClass = Class(Singleton), \\ singletonClass.instance : Singleton \\ \textbf{CreateObject}(singletonClass.gettor, Singleton, singletonClass.instance) \\ \textbf{RetrieveShared}(client.method, singletonClass.gettor, singletonClass.instance) \end{array}}{\textbf{Singleton}(Singleton, singletonClass.gettor, singletonClass.instance)} \tag{6.8}$$

6.7 Pattern Hierarchies

While other hierarchies of design patterns have been created (Gamma et al., 1995; Zimmer, 1995), none to date captures the low-level formalisms that ρ-calculus and the EDPs can produce. Figure 6.7 shows a partial hierarchy. The EDPs are grouped according to the sections in Chapter 5. It should now be obvious that the formalization and definition of the Gang of Four patterns is just a beginning. Using

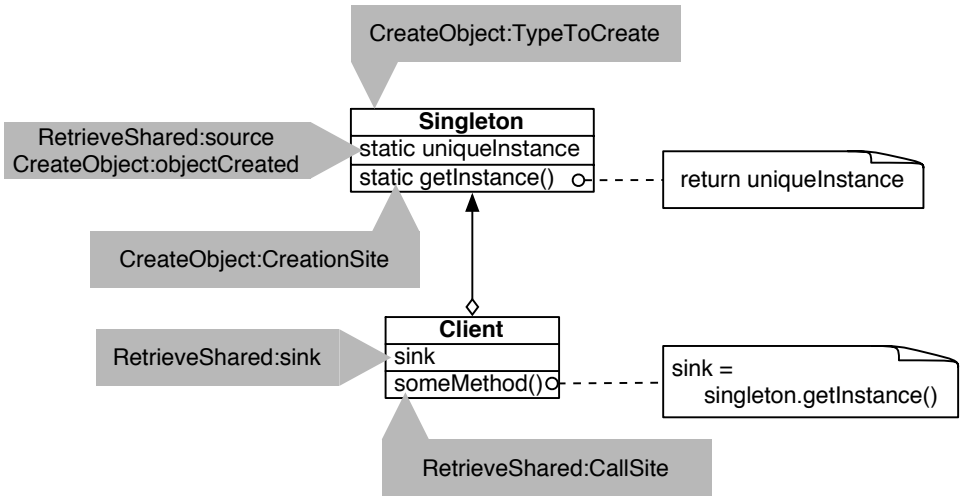


Figure 6.6: Singleton, annotated to show roles

precisely the same principles of composition used to produce the intermediate and Gang of Four patterns, much larger patterns can be built. This provides a continuous methodology from the finest details of the elements of programming, as exhibited in ς-calculus, to the very large scale architectural patterns that are now becoming of interest in systems research.

ρ-calculus and the EDPs provide a common framework and system within which a vast amount of research can be performed, depending on the level of interest of the researcher. I believe this creates an opportunity for a shift in programming methodologies towards what I term Intent-Oriented Programming, or IOP. IOP would, in principle, allow programmers and engineers to use the same conceptual basis and language to discuss the elements of software design from very basic implementation details to highly complex and abstract systems. This language is, of course, the language of design patterns. The fundamentals of this language are the EDP catalog, and ρ-calculus defines the permutation rules. By creating a common framework for both formal analysis and informal design discussion, and a series of mappings between the two, I assert that pervasive automated assistance of comprehension via intent extraction is not only feasible, but practical. SPQR is the first implementation of such assistance tools, and it is an example of how automated approaches can be used to shield the engineer from the intricacies and tedious nature of formal analysis. SPQR should be considered akin to a compiler, where highly formal methodologies are implemented in such a way that the average practitioner not trained in formal methods can use them successfully.

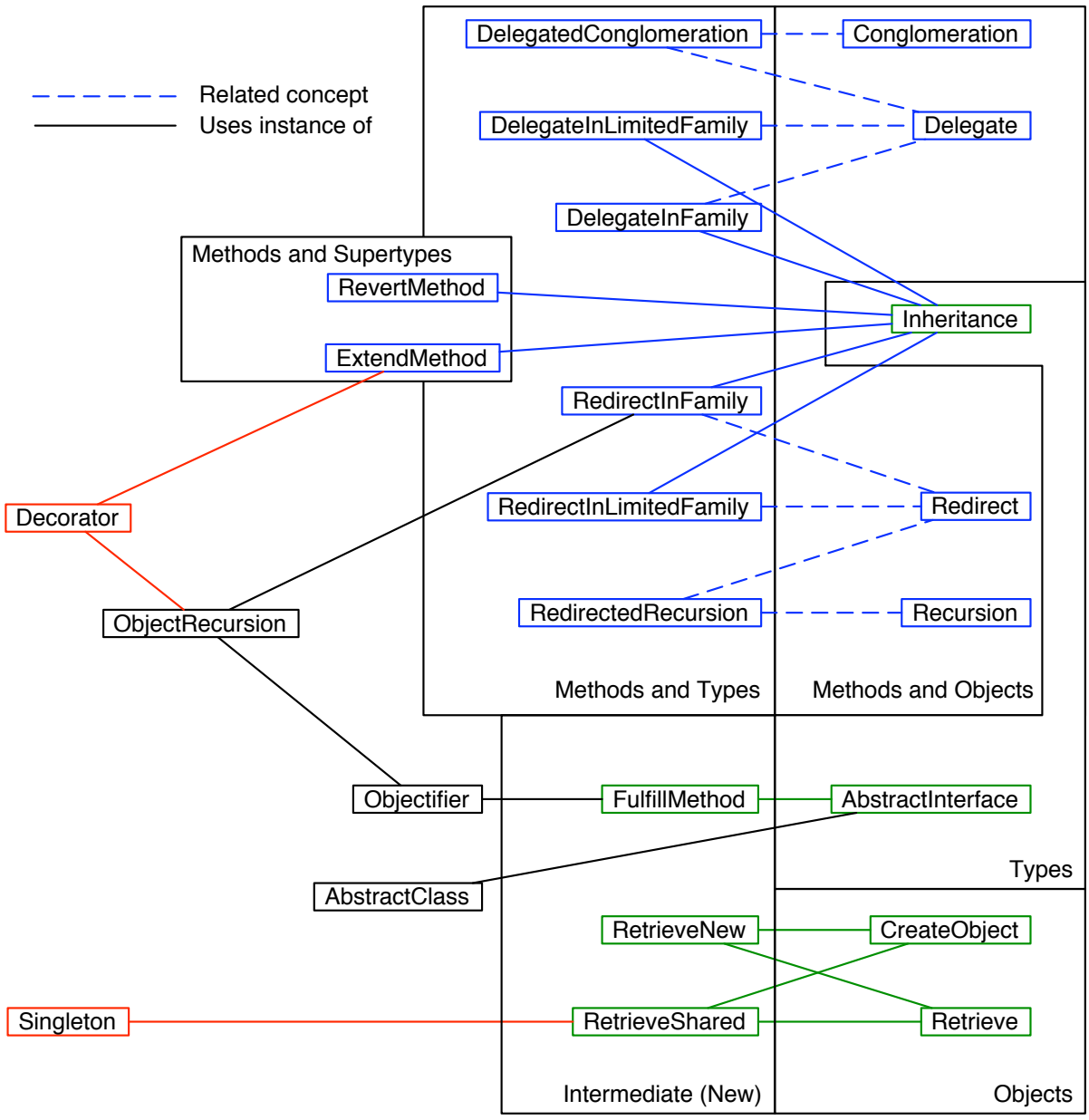


Figure 6.7: Hierarchy of patterns as defined in ρ-calculus

Chapter 7

Pattern/Object Markup Language

Having established the theory, implementation of a practical toolset can begin. Because the conceptual work to this point has focused on divining the proper representations of source code element relationships, it is only appropriate to continue with this problem in the implementation space.

The Pattern/Object Markup Language (POML) is the standard input and output data format for SPQR, an intermediate format for representing source code elements and their relationships. A unified format enables SPQR to operate on a number of source languages, after a transformation to POML. Likewise, POML, as an eXtensible Markup Language (XML) format, provides the opportunity to perform interesting transforms on the results of SPQR and the original code representations through the application of eXtensible Stylesheet Language Transforms (XSLT). Some examples are conversion to graphical UML diagrams, which include pattern information, traditional code metrics analysis, and new pattern metrics which I define in Chapter 11.

In this chapter, I will discuss POML v1.2 from the viewpoint of the practitioner wishing to use it as a code description system suitable for input to SPQR, and for novel ways to transform SPQR's results. I will also offer a concrete example of the theory behind ρ-calculus, to illustrate how ρ-calculus maps to object-oriented languages currently in regular use. Example fragments will illustrate the various pieces of POML. While each fragment is internally consistent, it will not be presented in the full context required to produce valid a POML file. The complete POML definition, as an XML Schema, can be found in Appendix F, and online at `http://www.cs.unc.edu/~smithja/spqr/poml.xsd`.

POML relies much more heavily on XML elements than attributes. This stylistic choice provides a useful separation between programming entity data and metadata, such as the source file and line number that generated the construct. The latter is useful during post-processing of SPQR results, but is in no way relevant to the analysis being performed, making it more appropriate as attribute data.

POML registers the XML namespace `poml`, and all elements are contained within that namespace. It uses the Dublin Core (`dc`) XML Schema found at `http://purl.org/dc/elements/` to enable internally documenting POML files. The outermost POML element is the `poml:system` element. All other POML descriptors are contained within this node. The elements that a `poml:system` can directly hold are the `poml:object`, `poml:class`, and `poml:pattern`.

7.1 Basic ρ-calculus Support

POML is a language-independent representation of the programmatic constructs of object-oriented programming. As such, it mirrors the needs and concepts of ρ-calculus. In many ways, POML can be thought of as a concrete format for ρ-calculus. This concrete format allows a number of tools to access the required information by utilizing well-known and tested XML parsing techniques. This section describes the various features of ρ-calculus as implemented in POML.

7.1.1 Objects, methods, fields, types

I start by again revisiting the basics of ρ-calculus as exhibited in ς-calculus: objects, methods, fields and types. Objects have a specific type, and contain methods and fields. POML directly reflects these relationships, and as clearly as possible by a `poml:object`:

```
<poml:object>
    <poml:type>SomeType</poml:type>
    <poml:method>
        <poml:name>m</poml:name>
    </poml:method>
    <poml:field>
        <poml:name>f</poml:name>
    </poml:field>
</poml:object>
```

Figure 7.1: Objects, methods and fields in POML

Methods and fields are formally considered a unified concept in ς-calculus for most object-oriented languages. As shown in Chapter 4, however, for the purposes of ρ-calculus, it becomes convenient to keep them separate, specifically to create the distinctions between the four reliance operators. Since POML is essentially a representation of ρ-calculus, it should to follow this convention. A side effect of this distinction is that it makes POML much more readily human-readable, as it more closely maps to a source code representation.

Methods have one unique feature that fields do not: the concept of the *abstract*, or *virtual*, method. These methods have no implementation definition, and, therefore, cannot be used on the left-hand side of either the $<_\mu$ or $<_\phi$ reliance operators. They may, however, be used on the right-hand side of either $<_\mu$ or $<_\sigma$. Since these names still need to be defined and listed in any representation of ρ-calculus, they can be tagged as non-implemented by using the `poml:abstract` empty element. If the `poml:abstract` tag appears in a `poml:method`, then no other element may appear in that method description, excepting `poml:name`.

A non-abstract method may have any or none of the three remaining ς-calculus elements that can be within ς-calculus method definitions: method selection, field selection, and update of a field. These are represented by the `poml:calls`, `poml:uses` and `poml:update` elements respectively, and are similarly associated with the $<_\mu$, $<_\phi$ and $\Leftarrow$ ρ-calculus notations.

POML elements only describe the right-hand sides of reliance operators, because the left-hand side of $<_\mu$ and $<_\phi$ are both known from the method being described. `poml:calls` contains all information needed to recreate a proper method call: object being used, method being selected, and a list of all parameters being passed to the method. Parameter passing is tagged as either using pass-by-value (copy) or pass-by-name (map) semantics. An example of this is shown in Figure 7.2, which illustrates a call to a method with a single parameter, perhaps `myWindow.moveTo(position);` in pseudo-code where `position` is a name alias, such as a pointer in C++.

```
<poml:method>
    <poml:name>exampleMethod</poml:name>
    <poml:calls>
        <poml:objectname>myWindow</poml:objectname>
        <poml:methodname>moveTo</poml:methodname>
        <poml:callingparameter>
            <poml:name>position</poml:name>
            <poml:type>Position</poml:type>
            <poml:keyword>pos</poml:keyword>
            <poml:callby>map</poml:callby>
        </poml:callingparameter>
    </poml:calls>
</poml:method>
```

Figure 7.2: Method call in POML

The `poml:keyword` element is a technique to ensure that the proper parameter at the call site is matched with the proper incoming parameter at the called method definition site. XML is not particularly suited to ordered data, and this situation requires absolute positional mapping. For languages

with parameter naming, this is an obvious fit. For those languages that do not have such a feature, something as simple as a sequence of names unique to a method will do. For instance, in the current C++ to POML implementation, keywords are generated as `kw` and are appended with ascending integers starting with zero: `kw0`, `kw1`, `kw2`, etc.

This mapping is completed by the definition of the incoming parameters for a method. The local name, parameter type, and keyword are provided, as with a calling parameter, as shown in Figure 7.3. A parameter declaration lacks the `poml:callby` element, however. The method should not care internally how the data is attached to the local parameter name.

Given that both the calling point of the method and the definition are at hand in this case, it would seem natural simply to use the name of the internal parameter of the called method instead of relying on the `poml:keyword` construct. It is not guaranteed, however, that the source code for the called method will be available for analysis. Consider a situation with two libraries, being maintained by two groups, or even two companies. Group A may not have the source code to Library B, but wishes to run a meaningful analysis. Group B can provide Group A with a POML representation of the code for analysis without having to hand over source code. The POML representation must therefore have some approach for mapping the parameters positionally, and `poml:keyword` is that solution.

As stated in Section 4.2.2, ς-calculus has established support for call-by-keyword. POML's method calling system is simply a practical implementation of this formal mechanism.

```
<poml:method>
    <poml:name>exampleMethod</poml:name>
    <poml:parameter>
        <poml:name>position</poml:name>
        <poml:type>Position</poml:type>
        <poml:keyword>pos</poml:keyword>
    </poml:parameter>
    <poml:result>
        <poml:name>errorFlags</poml:name>
        <poml:copyout/>
    </poml:result>
</poml:method>
```

Figure 7.3: Parameter declarations and result statements in POML

Figure 7.3 also introduces the `poml:result` element. It is not, as one might first guess, a return type declaration for the method, but the equivalent of a return *statement*. It appears once for each return statement in a method, and indicates the *value* that is being passed back, as well as the passing style.

Instances of $<_\phi$ are simpler, containing only a `poml:objectname` and `poml:fieldname` pair to indicate

the field being accessed, as shown in Figure 7.4. `poml:uses` instances can describe uses of objects and fields external to the method, or fields local to the method. The latter are indicated by a `poml:field` element that is exactly like the similar element in `poml:object`, as shown in Figure 7.1.

```
<poml:method>
    <poml:name>exampleMethod</poml:name>
    <poml:uses>
        <poml:objectname>myWindow</poml:objectname>
        <poml:fieldname>currentPosition</poml:fieldname>
    </poml:uses>
</poml:method>
```

Figure 7.4: Uses in POML

A brief aside about naming is appropriate here. The above split approach of object and method or field is strictly to facilitate certain analysis operations, and could easily be replaced by another approach involving scoping of names. Scoping is a standard way of partitioning namespaces and limiting potential clashes in languages. POML uses an analogous approach for most names. Scoping systems require a delimiter of some sort, a syntactic cue that a namespace partition is being entered. C++, for example, uses a combination of `::` and a period (Accredited Standards Committee X3 (Information Processing Systems) – Working Group 21, 1996), while Java (Gosling et al., 2005), Smalltalk (Goldberg and Robson, 1983) and other languages use a period for all scoping needs.. POML uses nested elements of the form `poml:scope`. Anywhere a name can appear in a POML file, that name can be optionally scoped. An unscoped name appears in the manner of the names in Figure 7.4. A scoped name will have a series of `poml:scope` elements listed in sequence from most embedded to most general. In the instance of the C++ scoped name `systemGUI.myWindow.currentPosition`, the POML representation would be as in Figure 7.5.

```
currentPosition
<poml:scope>myWindow
<poml:scope>systemGUI
</poml:scope></poml:scope>
```

Figure 7.5: Scoping in POML

Similarly, the `poml:methodname` and `poml:fieldname` in the `poml:calls` and `poml:uses` are strictly optional as well, given that they could be simply rolled into a scoped name, but they are recommended for general use.

The update operator completes POML's representation of the basic ς-calculus constructs. Assume that the source code has been converted to a conceptual ς-calculus form, as outlined in Section 3.1. There are, therefore, two forms of the update operator: those with a right-hand side that is a field access, and those with a right-hand side that is a method call.

The former is simple, as shown in Figure 7.6. As mentioned above, scoped names could be used in either the `poml:lhs` or `poml:rhs` positions.

```
<poml:update>
    <poml:lhs>currentPosition</poml:lhs>
    <poml:rhs>newGivenPosition</poml:rhs>
</poml:update>
```

Code: `currentPosition = newGivenPosition`

Figure 7.6: Update from a field access in POML

```
<poml:update>
    <poml:lhs>currentPosition</poml:lhs>
    <poml:fromcall/>
    <poml:rhs>
        <poml:objectname>myWindow</poml:objectname>
        <poml:methodname>moveTo</poml:methodname>
        <poml:callingparameter>
            <poml:name>position</poml:name>
            <poml:type>Position</poml:type>
            <poml:keyword>pos</poml:keyword>
            <poml:callby>map</poml:callby>
        </poml:callingparameter>
    </poml:rhs>
</poml:update>
```

Code: `currentPosition = myWindow.moveTo(position)`

Figure 7.7: Update from a method call in POML

The second case, where the return value of a method is used to set the target, has much in common with the `poml:calls` element. In fact, the structure of `poml:calls` is replicated in the `poml:rhs`, as in Figure 7.7. The `poml:fromcall` tag element distinguishes between this form and the former one, enabling parsing tools, such as XSLT transforms, to switch modes with ease. Alternatives would have been to rely on the format of `poml:rhs` to indicate which form of update was being used, or to provide two forms of `poml:update`, but neither was entirely satisfactory. This midpoint of explicit tagging, but reusing the `poml:update` format allows for XSLTs to treat all `poml:update` elements in the same fashion

for the `poml:lhs` element, and have only one decision point for determining the form of `poml:rhs`.

7.1.2 Classes

Now that I have established the basics of ρ-calculus in place, this section will show how POML can be extended to incorporate other language features. The most important addition, classes, is obvious. POML follows the ς-calculus mapping of classes to objects as stated in Section 3.1.4.

Following this model, POML includes the `poml:class` element. Much like a `poml:object` element, it includes methods, fields, and such. `poml:class`, however, only contains those methods and fields that are instance specific, that is those found in unique instantiated objects of a class type.

```
<poml:class>
    <poml:name>Position</poml:name>
    <poml:field>
        <poml:name>x</poml:name>
        <poml:type>Integer</poml:type>
    </poml:field>
    <poml:field>
        <poml:name>y</poml:name>
        <poml:type>Integer</poml:type>
    </poml:field>
</poml:class>
```

Figure 7.8: Emulating instance elements of a class in POML

Classes map to objects by the artificial creation of a class-object, an object that handles all object creation and destruction for that class. It also contains any methods or fields that are non-instance related, i.e., those declared static in most languages. Because of this dichotomy, and separation of methods and fields that are considered conceptually tied in a single class, POML needs a way to associate a class-object with the appropriate class description. This is achieved by adding the `poml:isclassobjfor` element to a `poml:object` description. The value of this element is the name of the class that the object is tied to. As shown in Figure 7.9, this aids a human reader, but is not necessary for the automated analyses of SPQR, when the name of the class object is a derivative of the original class name.

```
<poml:object>
    <poml:name>Position__Obj</poml:name>
    <poml:isclassobjfor>Position</poml:isclassobjfor>
</poml:object>
```

Figure 7.9: Using a class-object in POML

Technically, a `poml:class` describes a type to be instantiated, and the associated class-object is a factory for creating objects of that type. `poml:class` was initially named `poml:typedef` because this is the only explicit type definition location in POML, but `poml:class` is a more natural fit to the source languages represented by POML. Inheritance, the only purely type-oriented reliance operator in ρ-calculus, is handled here by inserting a `poml:parent` element into the `poml:class` element. This does not need to be done for the class-object, and, in fact, would be nonsensical.

```
<poml:class>
    <poml:name>FlippedPosition</poml:name>
    <poml:parent>Position</poml:parent>
</poml:class>
```

Figure 7.10: Class inheritance in POML

Including class inheritance necessitated certain decisions on the finer points of POML. Consider the situation where a subclass inherits a superclass' method definition, but does not override it. POML can handle this in two ways: explicitly, and implicitly. The explicit approach requires the complete method description to be copied to the subclass description. While this would be valid POML, this may result in a larger than desirable file that contains redundant information. To alleviate this, POML's implicit approach emulates what most source languages do at runtime with a lookup table, creating a list of inherited methods and fields, and the classes in which they are defined. This reduces the redundant information, while allowing each class to be completely self-describing. A simple coalescing of the method and field definitions by an analysis tool brings together all of the proper descriptions into a single unit.

The `poml:inheritedmethod` and `poml:inheritedfield` elements indicate the inherited pieces of the superclasses, as shown in Figure 7.11. The names are fully scoped, to clarify situations where multiple inheritance occurs.

Overridden methods are simply re-defined within the class, as in Figure 7.12. Because overriding methods hides the similarly named superclass method, and because subclasses frequently will seek to access those hidden methods, many languages offer directly calling those hidden methods through a construct such as `Superclass::method()` syntax in C++. This would cause problems in POML if modeled directly, however, because the method name is masked, and the above style of syntax would be attempting to select the method out of the `poml:class` element. As stated above, the `poml:class` element is a description of a type, and not an instantiated object, so the selection is poorly formed.

This problem is solved by the availability of a mapping element available at the `poml:class` level:

```
<poml:class>
    <poml:name>FlippedPosition</poml:name>
    <poml:parent>Position</poml:parent>
    <poml:inheritedmethod>getX
        <poml:scope>Position</poml:scope>
    </poml:inheritedmethod>
    <poml:inheritedmethod>getY
        <poml:scope>Position</poml:scope>
    </poml:inheritedmethod>
    <poml:inheritedfield>x
        <poml:scope>Position</poml:scope>
    </poml:inheritedfield>
    <poml:inheritedfield>y
        <poml:scope>Position</poml:scope>
    </poml:inheritedfield>
</poml:class>
```

Figure 7.11: Implicitly inherited elements in POML

```
<poml:class>
    <poml:name>FlippedPosition</poml:name>
    <poml:method>
        <poml:name>setX</poml:name>
        ....
    </poml:method>
    <poml:method>
        <poml:name>setY</poml:name>
        ....
    </poml:method>
</poml:class>
```

Figure 7.12: Overridden methods in POML

```
<poml:class>
    <poml:name>FlippedPosition</poml:name>
    <poml:directlycalls>setX<poml:scope>Position</poml:scope>
        <poml:as>Position_setX</poml:as>
    </poml:directlycalls>
</poml:class>
```

Figure 7.13: Static call to a superclass' method in POML

`poml:directlycalls`, shown in Figure 7.13. Calls of the method within the class will use the name given in the `poml:as` subnode. This mapping allows subclasses to statically call a superclass' definition of an overridden method. This is analogous to the $c\hat{}l$ class selection construct in ς-calculus, and specifically the O-1, O-2, and O-3 languages (Abadi and Cardelli, 1996)[pgs. 155, 275, 307], as previously noted in Section 5.7.

7.2 Patterns

The above discussion describes how POML models objects and classes, and I now introduce the final high level abstraction, `poml:pattern`. As discussed in Chapter 1, a pattern can be described conceptually as a series of *roles* and *fulfillers*. The roles come from the definition of a design pattern, as the necessary components required for a formal definition. The fulfillers are those source code constructs that fulfill those roles. Figure 7.14 demonstrates this element using the description of a Singleton pattern to obtain a global reference to a printer. See Section D.1.5 in Appendix D for the formalization that it derives from.

```
<poml:pattern>
    <poml:name>Singleton</poml:name>
    <poml:role>
        <poml:name>Singletor</poml:name>
        <poml:fulfilledBy>MyPrinterDescriptor</poml:fulfilledBy>
    </poml:role>
    <poml:role>
        <poml:name>gettor</poml:name>
        <poml:fulfilledBy>getPrinter
            <poml:scope>MyPrinterDescriptor_ClassObj</poml:scope>
        </poml:fulfilledBy>
    </poml:role></poml:pattern>
    <poml:role>
        <poml:name>instance</poml:name>
        <poml:fulfilledBy>commonPrinterSetting
            <poml:scope>MyPrinterDescriptor_ClassObj</poml:scope>
        </poml:fulfilledBy>
    </poml:role>
</poml:pattern>
```

Figure 7.14: Patterns in POML

7.3 Additional ρ-calculus Concepts

The `poml:calls` and `poml:uses` elements correspond to the $<_\mu$ and $<_\phi$ reliance operators of ρ-calculus, as noted before, leaving the $<_\sigma$ and $<_\kappa$ reliance operators to be supported. As noted in Section 4.3, the $<_\sigma$ and $<_\kappa$ operators can be derived from other ρ-calculus concepts, all of which have been defined in POML in the above sections. Direct support for these last two reliance operators can be had by using the `poml:sigma` and `poml:kappa` elements, however, as in Figure 7.15. These are child nodes of a `poml:method` element, corresponding to the correct context for the reliance operator. This unifies the POML representation of reliance operators as contained within their respective contexts. As defined in Section 4.3.2, $<_\mu$ and $<_\phi$ have an implicit context of the method, given by the left-hand side of the reliance operator.

```
<poml:method>
    <poml:name>exampleMethod</poml:name>
    <poml:sigma>
        <poml:lhs>position</poml:lhs>
        <poml:rhs>
            <poml:objectname>myWindow</poml:objectname>
            <poml:methodname>moveTo</poml:methodname>
        </poml:rhs>
    </poml:sigma>
    <poml:kappa>
        <poml:lhs>position</poml:lhs>
        <poml:rhs>
            <poml:objectname>myWindow</poml:objectname>
            <poml:fieldname>currentPosition</poml:fieldname>
        </poml:rhs>
    </poml:kappa>
</poml:method>
```

Figure 7.15: Field-based reliance operators in POML

7.4 SPQR Support

POML also has support for defining patterns and inference rules in SPQR. This makes POML suitable not only for representing input and results, but also as a defining language for the catalog of searchable patterns. The utility of this will be demonstrated more thoroughly in Section 9.5, but a short example will suffice here to finish the discussion of POML.

Defining patterns within POML is achieved by creating a file with the criteria elements, and then providing a `poml:resultpattern` element. I will use a portion of the **ObjectRecursion** from Equation

6.3 to illustrate this in Figure 7.16. This element defines a series of quantifiers for the inference as `poml:quantifier` elements, and is otherwise the same as a pattern instance, defining roles and which quantified variables fulfill them.

To support more generic operations that may be needed specific to particular automated theorem provers and inference systems, POML offers the `poml:atprule` element, which just encloses raw input for the prover indicated by the `atp` attribute value. This is also shown in Figure 7.16, where a rule is needed to ensure that *Recurser* is not the same element in the code as *Terminator*. Since these are quantified variables, it would be possible for the same code entity to fulfill both roles simultaneously. This criteria prevents that. POML does not attempt to ensure any validity of the contents of a `poml:atprule` element, it is passed through to the solver as appropriate.

```
<poml:atprule atp="otter">$NOT($ID(Recurser, Terminator))</poml:atprule>

<poml:resultpattern>
        <poml:name>ObjectRecursion</poml:name>
        <poml:quantifiers>
            <poml:quantifier>Initiator</poml:quantifier>
            <poml:quantifier>Handler</poml:quantifier>
            <poml:quantifier>Terminator</poml:quantifier>
            <poml:quantifier>Recurser</poml:quantifier>
            <poml:quantifier>handleRequest</poml:quantifier>
            <poml:quantifier>init</poml:quantifier>
            <poml:quantifier>obj</poml:quantifier>
            <poml:quantifier>m</poml:quantifier>
        </poml:quantifiers>
        <poml:role>
            <poml:name>Handler</poml:name>
            <poml:fulfilledBy>Handler</poml:fulfilledBy>
        </poml:role>
        <poml:role>
            <poml:name>Recurser</poml:name>
            <poml:fulfilledBy>Recurser</poml:fulfilledBy>
        </poml:role>
        <poml:role>
            <poml:name>Terminator</poml:name>
            <poml:fulfilledBy>Terminator</poml:fulfilledBy>
        </poml:role>
        <poml:role>
            <poml:name>Initiator</poml:name>
            <poml:fulfilledBy>Initiator</poml:fulfilledBy>
        </poml:role>
</poml:resultpattern>
```

Figure 7.16: Example pattern definition snippet from **ObjectRecursion**

7.5 Unsupported Constructs

The reader may notice that there are a few common programming language concepts that POML does not address, such as visibility and access policy, and abstract classes. This is deliberate, because POML is a representation of the conceptual model at the object level, as defined in ρ-calculus. POML is not a representation for checking the validity of code against a particular language implementation; that task is left to a compiler. The POML representation is assumed to be valid source code in *some* source code language. As translators for new source code languages are created, it is up to the implementor to ensure that both the original source code is consistent and valid for the language being used, and that the conceptual mappings are coherent. Many of the above unsupported concepts can be mapped to ς-calculus and, therefore, to POML, just as the classes were in Section 7.1.2. See (Abadi and Cardelli, 1996) for details on how these can best be accomplished. For example, on the subject of abstract classes or visibility issues, see (Abadi and Cardelli, 1996) Section 8.5.2, 'Variations on Class Types'.

Chapter 8

Converting C++ to POML

The previous chapter introduced POML as an abstraction suitable for representing object-oriented languages. This chapter will provide an example of mapping an object-oriented programming language, in this case C++, to POML. This is intended to be a guide for others who wish to implement a C++ front end to SPQR in another environment, and as an example conceptual mapping of many common language features, to facilitate the support of other languages for SPQR analysis. C++ was chosen for three reasons: static typing, ubiquitousness of code, and a mixed object style.

At the time that the language choice was made, I felt that the static typing would minimize the theoretical problems that could occur with mapping a higher-order language such as Smalltalk to a first-order solving system. Unfortunately, the unforeseen issues with syntax, parsing problems, and peculiarities of the tools used in an attempt to alleviate these problems utterly swamped the foreseen theory issues. In retrospect, a more purely object-oriented programming language such as Java or Smalltalk would have certainly provided a faster implementation, but this conversion process was educational, and I am now confident that more dynamic languages can be mapped quickly.

I also wanted a system that would be practical and useful to a large number of practitioners. At the time the language selection was made, C++ was still the language most used in industry. Again, Java has grown in popularity since then, and it is uncertain which is more prevalent in use today. Given the sheer amount of legacy code, however, I believe that C++ will be a highly useful implementation language.

Finally, languages such as Java and Smalltalk have a much more pure object-model, while C++ has a tremendous amount of design legacy from its roots in C. Because one of the assertions about SPQR was that it should provide a mixed-analysis environment, I felt that this would adequately test the mixing of λ-calculus, ς-calculus, and ρ-calculus concepts. It certainly proved to be a challenge.

The translation from C++ to POML is essentially one of mapping C++ to ρ-calculus. The majority of C++ maps cleanly to ς-calculus, particularly when expressed in a class-based form. Some major features require some explanation however. In this chapter I will illustrate how C++ is mapped to POML in the current implementation of SPQR, by way of the `gcc2poml` tool. The POML examples in this chapter will only be descriptive, and not guaranteed to be completely conforming. If, for example, a `poml:class` element appears with only a name inside the element delimiters, it will be assumed to representative of a fully defined and conforming POML element. Chapter 7 should be referred to when appropriate.

There are certain C++ elements and constructs which are not handled in this conversion. As stated in Chapter 7, POML does not handle visibility constructs, nor behavior-modifying qualifiers such as `const` or `volatile`. While POML could, it does not at this time, and I refer to Abadi and Cardelli when describing ς-calculus: "we aim to cover a wide, but not universal, range of object-oriented constructions." (Abadi and Cardelli, 1996, pg. 53) Other minor features of C++ are likewise currently ignored, and they will be mentioned when appropriate in the following discussion.

Wherever possible, I used the names that were used internally by `gcc`. I did *not* however use the fully mangled names. It is my assertion that the POML files and SPQR results should be, in all cases, as human readable as possible. Mangled names do not assist in that, in my opinion. `_ZN3Top7setDataEi` is less readable than `Top::setData`, and I have other approaches to handle overloading and such that mangling is designed to alleviate.

8.1 Classes

Because C++ is a class-based object-oriented language, the first item to address is classes. As shown in Section 3.1.4, ς-calculus does not directly support classes, but (Abadi and Cardelli, 1996) demonstrates using objects to emulate a class-based language. As described in Section 7.1.2, I follow their example, by creating a class-object for each class. All class-level elements of a class, such as constructors, destructors, and static methods and fields, will be moved to the class-object.

Figure 8.1 shows an example class with a single constructor, a single destructor, and a single static method. Two instance methods are also declared. As per POML and ς-calculus, the latter two methods go into a `poml:class` element, while the former go into a `poml:object` that is then linked to the `poml:class`.

The destructor was changed from `~Window` to `__dtor_Window` both to eliminate the tilde character,

and to mirror the constructors automatically and invisibly created by `gcc` for most every class. See Section E.2 for a discussion of artificial constructs created by `gcc`.

```
class Window {
public:
    // Window class that also retains links to each window instance generated
    // This lets the Window class keep track of how many Windows have been made
    Window();   // Default constructor
    ~Window();  // Destructor

    // Settor/gettor pair for the window position
    Window&     setPosition( Position pos );
    Position    getPosition();

    // Get the current number of instances
    static int  getNumberOfWindows();
private:
    Position    position;
    static int  numberOfWindows;
};
```

Figure 8.1: Example C++ class

Classes, structs, and unions are all handled as equivalents. A union differs from a struct only in memory management, a trait not used in SPQR analysis. A struct is simply a class with fully public members by default, but in which any access control can be imposed. Because access controls are also ignored, this reduces a struct and a class to the same conceptual object, and, by extension, a union as well. This effectively eliminates any type conversions that may occur when accessing the elements of a union through various means. I consider ignoring this aspect to be reasonable at this time, as it falls outside the behaviors of object-oriented programming, and is instead a peculiarity of the C++ runtime. Such behavior *could* be emulated by converting each union into a full class, with a number of settor and gettor methods, one matching pair for each element of the union, and one single data field. The settors and gettors would then be responsible for the type conversions. This is beyond the needs of SPQR at this time, but it is worth noting that it is entirely possible.

8.2 Templates

Templates are a very powerful feature of C++, yet they can be simplified greatly in the production of a POML file. When `gcc` compiles a translation unit, it automatically creates instantiations of the templates as needed *for that translation unit*. These are then treated as regular classes by `gcc`, and

```
<poml:object>
    <poml:name>Window_ClassObj</poml:name>
    <poml:isclassobjfor>Window</poml:isclassobjfor>
    <poml:type>Window_ClassObj_Type</poml:type>
    <poml:method>
        <poml:name>Window</poml:name>
    </poml:method>
    <poml:method>
        <poml:name>__dtor_Window</poml:name>
    </poml:method>
    <poml:method>
        <poml:name>getNumberOfWindows</poml:name>
    </poml:method>
    <poml:field>
        <poml:name>numberOfWindows</poml:name>
        <poml:type>int</poml:type>
    </poml:field>
</poml:object>

<poml:class>
    <poml:name>Window</poml:name>
    <poml:method>
        <poml:name>setPosition</poml:name>
    </poml:method>
    <poml:method>
        <poml:name>getPosition</poml:name>
    </poml:method>
    <poml:field>
        <poml:name>position</poml:name>
        <poml:type>Position</poml:type>
    </poml:field>
</poml:class>
```

Figure 8.2: C++ class converted to a class/object pair

I consider them to be just another type of classes with odd names. The compile-time behaviors of templates are already handled by the time `gcc` makes its internal representation available for analysis by SPQR, and lost. I believe this is acceptable for this implementation of SPQR, given that these behaviors are created and finished during compile time, not run time, and they are limited to simple static calculations and the like. If `gcc` provided information appropriate for this computation phase, modeling this behavior in SPQR would certainly be possible.

When converting a template name to something suitable for POML, the angle brackets, spaces, and commas must be stripped out. This can lead to some strange looking class names, but their syntax should make their template-based derivation obvious. For instance, `vector< int, float >` would be converted to `vector__int_float__` The double-underscores replace the brackets, and spaces and commas are each converted to a single underline. There is the possibility of name clashing using this simple algorithm, but, to date, it has been effective in my research. It is certainly an opportunity to increase the robustness of SPQR's tools.

I consider the above naming scheme to be no better, nor any worse, than the default `gcc` template naming system. It is not simple to read, but it is consistent with the needs of POML and SPQR, and provides enough clues to an engineer to allow for quick reverse engineering of the original template name and template parameters.

8.3 Namespaces

C++ namespaces can be considered as raw ς-calculus objects. Namespaces cannot be instantiated and they cannot be destroyed; they are containers for objects, methods and classes, which were shown in the previous section to be describable as objects. Because these are artificial constructs, we set their type to be an arbitrary one named after the object name. For instance, namespace `std` is mapped to an object named `std` with type set to `std_Type`. The object defines its own type by existing. Since this type information is never used in C++, I will never use it in this analysis, but it completes the conceptual mapping.

Namespaces and other elements are nested by using the scoping elements of POML. Figure 8.3 shows a simple nested element system with a class inside a namespace and a secondary nested namespace. This translates to Figure 8.4, which shows how each element is brought out to be a child element of the `poml:system` entity, and shows the scoping used to create the proper scoped name.

```
namespace CoLab {
    class Window {};
    namespace Facetop {};
};
```

Figure 8.3: Example C++ nested namespaces and classes

```
<poml:object>
    <poml:name>CoLab</poml:name>
    <poml:type>CoLab_Type</poml:type>
</poml:object>

<poml:object>
    <poml:name>Facetop
        <poml:scope>CoLab</poml:scope>
    </poml:name>
    <poml:type>Facetop_Type</poml:type>
</poml:object>

<poml:object>
    <poml:name>Window_ClassObj
        <poml:scope>CoLab</poml:scope>
    </poml:name>
    <poml:type>Window_ClassObj_Type</poml:type>
    <poml:isclassobjfor>Window</poml:isclassobjfor>
</poml:object>

<poml:class>
    <poml:name>Window
        <poml:scope>CoLab</poml:scope>
    </poml:name>
</poml:class>
```

Figure 8.4: POML nested namespaces and classes

8.4 Methods

In ρ-calculus and ς-calculus, methods must be tied within an object. I will show in Section 8.7 how global methods can be emulated in POML. For bound instance methods, there is a peculiarity of associating the instance with the method. Luckily, `gcc` provides an apparatus for this by injecting an artificial parameter as the first parameter for any instance method. This artificial parameter is aptly named `this`, matching the keyword of the same purpose. This allows accesses of `this` in the code to be treated as regular accesses of an object. Because C++ is a simple language in many respects, this is sufficient. Analysis of more dynamic languages corresponding to an $O-3$ level language with this technique would not be as effective.

8.4.1 Operators

In C++, operators are used to simplify syntax (Accredited Standards Committee X3 (Information Processing Systems) – Working Group 21, 1996, 13.5, over.oper). It is generally easier for a programmer to comprehend a numeric `posFinal = posInitial + offset + localOffset` than it is the more expansive form `posFinal.assign(posInitial.move(offset).move(localOffset))`. Instead of attempting to use the symbolic operators, `gcc` tags such methods with an `operator` attribute, and renames them to the proper operator symbol name. I combine these two pieces into a new name that is standardized across `gcc`. `operator ==` becomes `operator_equals`, `operator +` becomes `operator_plus`, and so on. In all other respects, operators are treated as regular methods.

8.4.2 Overloading

Overloading of methods is allowed in C++ based on the type of the parameters (Accredited Standards Committee X3 (Information Processing Systems) – Working Group 21, 1996, 13, over). This is handled internally in `gcc` by creating mangled names, cryptic strings that encode the various pieces of the overloading and naming into a unique name. Regularly difficult to read, I produce a modified mangling of overloaded method names in an effort to alleviate this. The first name in an overloading set is left intact. Starting with the second name identified to be an overloaded method, `_overload_N` is appended, where `N` is an integer starting with 1. The numbers are unique to the current scope, so three overloaded methods named `move` in the class `Window` will be named `move`, `move_overload_1` and `move_overload_2` in the resulting POML file. Two overloaded methods similarly named in another class `Player` will be named `move` and `move_overload_1` in the same file. The scoping of names takes care of the rest, as

always. Overloading is determined dynamically for methods that are moved into new POML objects, such as anonymous namespace or translation unit items.

8.4.3 Parameters and Return Values

Calling parameters and return values need to be tagged as either call-by-name of call-by-value to be POML compliant, and to support various data-oriented analyses of ρ-calculus and SPQR. These transport objects are most easily identified by inspecting their types. Those objects with a type of a pointer or a reference are tagged as call-by-name, while plain objects, those copied anew on crossing the method enclosure barrier, are tagged as call-by-value. This is a simplistic approach that addresses neither C++'s memory management issues, nor various new C++ features such as `auto_ptr`, but this logic can be easily remedied as necessary.

The above details are coalesced into the example in Figure 8.5. In the example POML in Figure 8.6 I ignore all elements of the POML not necessary to the current discussion.

```
class C {
public:
    B&  methodOne()  { return aBObjRef; };
    B   methodTwo()  { return aBObj; };
    B   methodTwo( B b ) { return b; };
    C&  operator - ( C& rhs );
private:
    B&  aBObjRef;
    B   aBObj;
};
```

Figure 8.5: Example C++ method details

8.5 Expressions

While the above satisfies the need for defining methods, some work was needed to handle method calls properly, particularly those embedded in expressions. Expressions come in many forms in C++, such as nested, chained, and conditional, and each must be normalized to a common form for the most straightforward analysis.

The form that most cleanly maps to ρ-calculus and POML is that of expanded expressions with explicit temporary variables. For instance, the code in Figures 8.7 and 8.8 is equivalent. The latter, however, is in a pure form. Each statement is either an update or a pure method call, and the normalized

```
<poml:class>
    <poml:name>C</poml:name>
    <poml:method>
        <poml:name>methodOne</poml:name>
        <poml:result>
            <poml:name>aBObjRef</poml:name>
            <poml:mapout/>
        </poml:result>
    </poml:method>
    <poml:method>
        <poml:name>methodTwo</poml:name>
        <poml:result>
            <poml:name>aBObj</poml:name>
            <poml:copyout/>
        </poml:result>
    </poml:method>
    <poml:method>
        <poml:name>methodTwo_overload_1</poml:name>
        <poml:parameter>
            <poml:name>b</poml:name>
            <poml:type>B</poml:type>
            <poml:keyword>kw0</poml:keyword>
        </poml:parameter>
        <poml:result>
            <poml:name>b</poml:name>
            <poml:copyout/>
        </poml:result>
    </poml:method>
    <poml:method>
        <poml:name>operator_minus</poml:name>
        <poml:parameter>
            <poml:name>c</poml:name>
            <poml:type>C</poml:type>
            <poml:keyword>kw0</poml:keyword>
        </poml:parameter>
        <poml:result>
            <poml:name>aBObj</poml:name>
            <poml:copyout/>
        </poml:result>
    </poml:method>
</poml:class>
```

Figure 8.6: POML method details

format provides for a much cleaner, if verbose, mapping to POML. The transitivities of ρ-calculus ensure that all relationships present in the original expression are reformable from the facts present in the expanded form.

```
myWindow.position = anotherWindow.getPosition() + windowingEnvironment.getOffset()
myWindow.setOffset(myWindow.position - anotherWindow.getPosition())
```

Figure 8.7: Example C++ nested expressions

```
temp1 = anotherWindow.getPosition()
temp2 = windowingEnvironment.getOffset()
temp3 = operator_plus(temp1, temp2)
myWindow.position = temp3
temp4 = myWindow.position
temp5 = anotherWindow.getPosition()
temp6 = operator_minus(temp4, temp5)
myWindow.setOffset(temp6)
```

Figure 8.8: Equivalent code as expanded expressions

For instance, from the code in Figure 8.7, one can define the ρ-calculus fact:

$$myWindow.position <_{\sigma} anotherWindow.getPosition$$

This same fact can be derived from the following facts found in Figure 8.8:

$$temp1 <_{\sigma} anotherWindow.getPosition$$

$$temp3 <_{\kappa} temp1$$

$$myWindow.position <_{\kappa} temp3$$

This technique of normalizing expressions is ubiquitous in compilers (Aho et al., 1986; Fischer and LeBlanc, 1991), and I refer the reader to sources on that topic for a more thorough discussion.

Chained expressions of the form `a = b, c();` are simply broken out into their constituent parts and handled as appropriate. Conditional expressions, such as `a = b ? c : d` are artificially broken out into a simple `if/then/else` statement. A nested conditional expression will be pulled out into the expanded form in the correct location by the method outlined above, and then it can be converted to a statement.

8.6 Statements

Some statements, likewise, need to be normalized into a simple, coherent form for the most effective production of POML. The most common is the selection statement (Accredited Standards Committee X3 (Information Processing Systems) – Working Group 21, 1996, 6.4, stmt.select), such as the `if` statement. Their conditional expression is analyzed for possible reliance operators, and their control blocks are treated as are any other statement.

Other scoping statements, such as blocks (Accredited Standards Committee X3 (Information Processing Systems) – Working Group 21, 1996, 6.3, stmt.block), are simply expanded in place and treated as normal statements. As stated in Section 4.3.3, ρ-calculus has no concept of statement ordering, so data analysis techniques such as conditional dependence are not currently supported. As such, *all* statements within a method body are treated equally, regardless of their nesting within blocks or subblocks. This is a known and acknowledged weakness in SPQR, and will require the addition of temporal logic analysis at a future date to handle appropriately.

Iteration statements such as `for`, `while` or `do` (Accredited Standards Committee X3 (Information Processing Systems) – Working Group 21, 1996, 6.6, stmt.iter) combine the features of the two above approaches, pulling out the conditional statement, then treating the body of the iteration loop as a block.

8.7 Linkages

C++ has three official linkages: external, internal, and none (Accredited Standards Committee X3 (Information Processing Systems) – Working Group 21, 1996, 3.5 basic.link). External linkage is defined for a name when 'the entity it denotes can be referred to by names from scopes of other translation units or from other scopes of the same translation unit.' There is one way that all scopes can access the same element in ς-calculus, and that is by creating a single object to represent this external, or global, linkage. Consequently, there is one object in any given SPQR run that is named `__GLOBAL__`. Any name not associated with a class or namespace, and determined to have external linkage, is moved to that object.

Similarly, internal linkage is defined for a name when 'the entity it denotes can be referred to by names from other scopes in the same translation unit.' Creating an object for the current translation unit provides a location for this linkage. Each translation unit will have its own object. For readability, these objects are named after the source file that, when compiled, created the logical translation unit.

```
int globalData;
void aGlobalMethod();
enum Direction{ North, South, East, West };
```

Figure 8.9: Example external linkage declared C++ elements

```
<poml:object>
    <poml:name>__GLOBAL__</poml:name>
    <poml:field>
        <poml:name>globalData</poml:name>
    </poml:field>
    <poml:method>
        <poml:name>aGlobalMethod</poml:name>
    </poml:method>
</poml:object>
```

Figure 8.10: POML `__GLOBAL__` object corresponding to external linkage

For instance, a file `testRun.cpp` would result in an object named `testRun_cpp_tu` being created to hold any items determined to have internal linkage. Examples of code elements with internal linkage are statically-declared methods and fields at the file level, anonymous namespaces and class-like entities, such as unions, structs, and such, and typedefs, shown in Figure 8.11. As these elements are detected, they are moved to the translation unit object as in Figure 8.12. Anonymous namespace entities are hoisted up to the translation unit object, while anonymous classes and unions are given artificial names unique to the translation unit object.

The default linkage, none, is handled by simply using the fully scoped names in POML. Visibility is handled by the scoping construct.

While `const` is not treated as a cv-qualifier for access control, it is taken as an initial value hint where

```
static int filespaceData;
static void filespaceMethod();
namespace {
    class ClassInAnonNamespace{};
    int anonData;
 }
union {
    int estimate;
    double precise;
};
typedef ReallyLongAnnoyingClassName BetterName;
```

Figure 8.11: Example internal linkage declared C++ elements in SourceFile.cpp

```
<poml:object>
    <poml:name>SourceFile_cpp_tu</poml:name>
    <poml:field>
        <poml:name>filespaceData</poml:name>
        <poml:type>int</poml:type>
    </poml:field>
    <poml:method>
        <poml:name>filespaceMethod</poml:name>
    </poml:method>
    <poml:field>
        <poml:name>anonData</poml:name>
    </poml:field>
</poml:object>

<poml:class>
    <poml:name>anonymous_union_1
        <poml:scope>SourceFile_cpp_tu</poml:scope>
    </poml:name>
</poml:class>

<poml:class>
    <poml:name>ClassInAnonNamespace
        <poml:scope>SourceFile_cpp_tu</poml:scope>
    </poml:name>
</poml:class>

<poml:class>
    <poml:name>BetterName
        <poml:scope>SourceFile_cpp_tu</poml:scope>
    </poml:name>
</poml:class>
```

Figure 8.12: POML translation unit object corresponding to internal linkage

```
class AbstractingClass {
public:
    virtual void absMethod() = 0;
};
```

Figure 8.13: Example **AbstractInterface** in source code

```
<poml:pattern>
    <poml:name>AbstractInterface</poml:name>
    <poml:role>
        <poml:name>Abstractor</poml:name>
        <poml:fulfilledBy>AbstractingClass</poml:fulfilledBy>
    </poml:role>
    <poml:role>
        <poml:name>operation</poml:name>
        <poml:fulfilledBy> absMethod </poml:fulfilledBy>
    </poml:role>
</poml:pattern>
```

Figure 8.14: POML emitted for **AbstractInterface**

appropriate. In cases such as `const int MaxWindows = 100;`, the variable definition is broken into two statements, a declaration, and an update with the initial value. The qualifier is otherwise completely ignored.

`enum` types are broken out into an equivalent set of fields that are given `const` values. For instance, `enum a {first, second, third};` produces three fields with the proper linkage, named `first`, `second`, and `third` all of which are type `a`.

8.8 EDPs

Due to the highly language-specific nature of the **AbstractInterface**, **CreateObject** and **Retrieve** EDPs, as described in Sections 5.2 and 5.3, detecting these directly from source code is best.

In C++, **AbstractInterface** is simple to detect, as it merely requires checking that the method body of a declared and defined method is set to `NULL`. Technically, it means looking for a construct of the form: `void virtual aMethod() = 0;`. When such an entity is detected, an **AbstractInterface** EDP can be emitted directly in the POML output, as shown in Figures 8.13 and 8.14.

CreateObject requires detection of both the `new` and implicit construction forms of object instantiation. The former is fairly simple to find, but the latter may be buried in the production of temporary variables in expression chains. It is up to the practitioner to determine if temporary variables are suitable

for analysis. I discuss this decision in more detail in Section 9.1.2.

The final EDP that requires detection directly from source code, **Retrieve**, is implemented by analysis during handling of update operators. If the right-hand side of the update is determined to be a method, a properly formed **Retrieve** EDP is emitted.

Two proposed features to the C++ language, `unique_ptr` and `shared_ptr`, may make the detection of **RetrieveShared** and **RetrieveNew** even more direct. See Section 12.1.3 for further explanation.

8.9 Unsupported constructs

Certain language features of C++ are unsupported, and these are essentially the same as described in Section 7.5 for POML. For the purposes of SPQR, all cv-qualifiers (Accredited Standards Committee X3 (Information Processing Systems) – Working Group 21, 1996, cvquals), `const`, `volatile` and so on, and access controls can be ignored - these are strictly to determine whether C++ code is conformant to the C++ standard, and are not considered in SPQR. Any code analyzed by SPQR is assumed to have been previously found to be correct and conformant code, and no attempt will be made to analyze it in that manner. SPQR is merely translating the code that exists, not determining whether that code should or should not be allowed.

Exceptions are an interesting case, as they step outside the normal runtime environment of C++, and can therefore be troublesome to describe effectively. While I believe I may have a workable model of exceptions in the context of ρ-calculus, it is not a feature of the language that I have implemented support for.

Additionally, the plain data types of C++ are generally considered to be black-box classes at the moment. This simplifies the creation of POML significantly, but could be rectified by the addition of artificial classes that conform to the semantics of the base types such as `int`, `float`, and so on. Because the behavior of these types is well formed in the C++ standard, treating them as artificial classes would be entirely possible. It is not certain, however, what that would gain someone wishing to understand the relationships between *classes* and *objects*. Base types as defined in C++ are not part of the object-oriented methodology, and there is some question as to the appropriateness of including them in SPQR analysis. I have at this time elected to include them in the POML files, but not to attempt to emulate their behavior.

Constants are similarly ignored in the current implementation of `gcc2poml`. They simply cannot take part in the relationships that SPQR is designed to analyze, so their existence in the POML file

would be a waste of space and parsing time. This leads to an extreme compacting of some expressions. For example, `a = 100;` in C++ will result in no POML emission - there are no objects or methods on the right-hand sideof the assignment to create a reliance with. On the other hand, `a = c.moveBy(b + 10)` will produce the expected $a <_{\mu} c.moveBy$, and $a <_{\phi} b$.

Arrays are possibly the most troublesome omission in the current implementation. They are intimately tied with the **Iteration** and **Collection** patterns I discussed in Section 5.10, but at a much lower level. Since there is not a good and practical way to determine how an array is being used in a particular context without significant analysis, I have elected to place arrays in the same category as other non-object elements of C++, and simply not support them at this time. I hope that further work on the data dependency reliances, including **Iteration** and **Collection**, will yield a workable model for arrays and pointers in such a context.

Another construct involving pointers that I do not handle at this time is the function pointer. This, at least, has a straightforward implementation path in the concept of a functor. A functor is an object that acts as a function. In C++, it has an implementation for the () operator, which allows the object to be treated syntacticly as a raw function or method. The type of the functor object can be mocked up as a variant of the method signature. Establishing an object-based equivalent for a function pointer looks to be a direct approach, although I have not implemented it.

This conceptual mapping of C++ is in some ways influenced by the internals of `gcc`, but there is no reason why these same abstractions cannot be used with other compiler systems. Additionally, the basic structure of the mapping should be of great utility to those wishing to map other object-oriented programming languages to POML. Every language so mapped will increase the usefulness of SPQR.

Chapter 9

SPQR Implementation

This chapter will describe the current SPQR implementation, which provides tools to support each of the abstractions defined in previous chapters. From the engineer's point of view, SPQR is a single fully automated tool that performs analysis from source code and produces a final report. A simple script provides the workflow, by chaining several modular component tools, centered around tasks of *source code feature detection*, *feature-rule description*, *rule inference*, and *query reporting*. These modules use freely available tools that implement open standards when possible, and tie them together with custom scripts and tools written in Python and XSLT. SPQR, therefore, is highly portable. The requirements for the current version of SPQR and their originating URLs are: the Python 2.3 programming language at http://www.python.org/, the xsltproc XSLT engine, part of the libxslt package at http://xmlsoft.org/XSLT/, and the Otter 3.3 automated theorem prover which can be found at http://www.mcs.anl.gov/AR/otter/. Once these are in place, the SPQR package can be obtained from http://www.cs.unc.edu/∼smithja/spqr/.

The SPQR toolchain consists of several components, shown in Figure 9.1. The top box is the language support 'front end' of SPQR, and must be replicated for each language to be analyzed. The output of this front end is in POML. The bottom box, the 'back end', is the unified toolset based on ρ-calculus abstractions that can operate on any POML file regardless of its source. This becomes of prime importance when discussing the opportunities for *training* SPQR later in this chapter.

Because the core of SPQR uses POML exclusively, any combination of source files can be analyzed simultaneously, from a number of languages. Assume that a project has the front end application in Java, but the back end in C++, and a suite of support scripts in Python. Analysis systems that concentrate on a single language cannot cross this language barrier, but SPQR can, without any modification to the core analysis engine or techniques. Each subsystem would be converted to POML by means of a

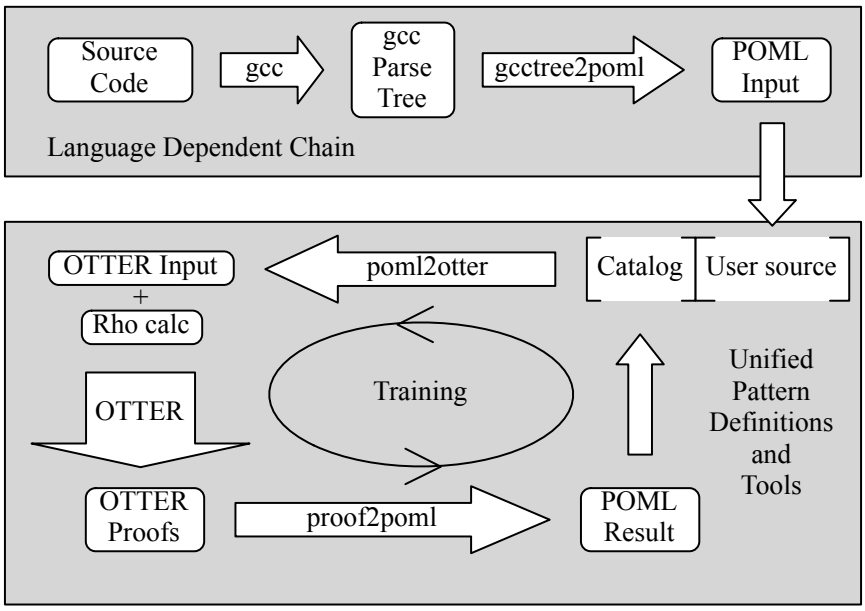


Figure 9.1: SPQR Overview

language-specific tool, as I have described here for C++, and then the POML files could be analyzed together as equals.

Each stage of SPQR is independent and was designed to allow other languages, compilers, workflows, inference engines, and report compilation systems to be added. Additionally, as new design patterns are described by the community, perhaps local to a specific institution or workgroup, they can be added to the catalog used for query. This holds for both benign and malignant design patterns. I will discuss in Section 9.5 how SPQR provides automated support for this training procedure, and how it can be used to bootstrap SPQR in complex environments.

I would like to note here that while SPQR is highly straightforward in implementation and use, it encapsulates a highly formalized semantics system that allows for the use of an automated theorem prover for rule inference. It is the formalization that, paradoxically, provides the flexibility of describing the complex programming abstractions of design patterns. The practitioner can avoid this level of detail, however, and SPQR can be adapted to work at multiple levels of formal analysis, depending on the particular need. SPQR should be considered not only as a practical tool for the engineer, but also as a general framework for research in design patterns detection.

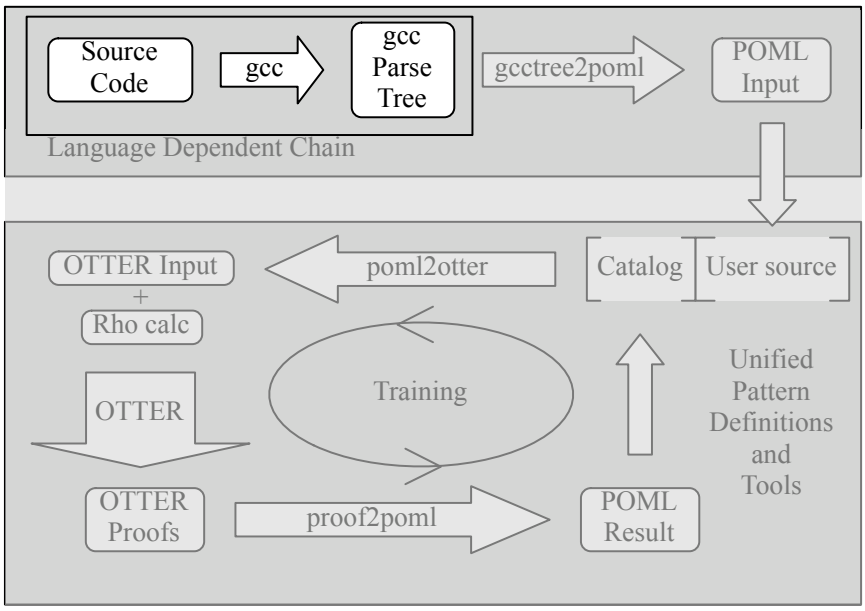


Figure 9.2: SPQR `gcc` phase

9.1 Feature detection and description

The first phase in SPQR analysis is to convert the source code to be analyzed into POML, corresponding to the top tool chain. The current implementation uses `gcc` 3.3 and a custom tool, `gcctree2poml`.

9.1.1 `gcc`

`gcc` is invoked as per normal for whatever system the programmer wishes to analyze, with the addition of two standard `gcc` 3.3 compile flags: `--dump-translation-unit` and `--dump-class-hierarchy`. The former creates files for each input source code file with a `.tu` suffix appended. The latter creates detailed information about the class hierarchies in a similar file with suffix `.class`. For example, if the source file `Window.cpp` is compiled with these flags, two new files `Window.cpp.tu` and `Window.cpp.class` will be generated. This is the only necessary addition to a user's existing tool chain to enable use of SPQR, shown in Figure 9.2. Unlike other systems that require user lock-in to use the tools in a round-trip fashion, SPQR currently only requires use of `gcc` 3.3. The remainder of the user's editing, build, and testing system is up to them. Future front ends will lift even this minor requirement and allow for a broader range of support.

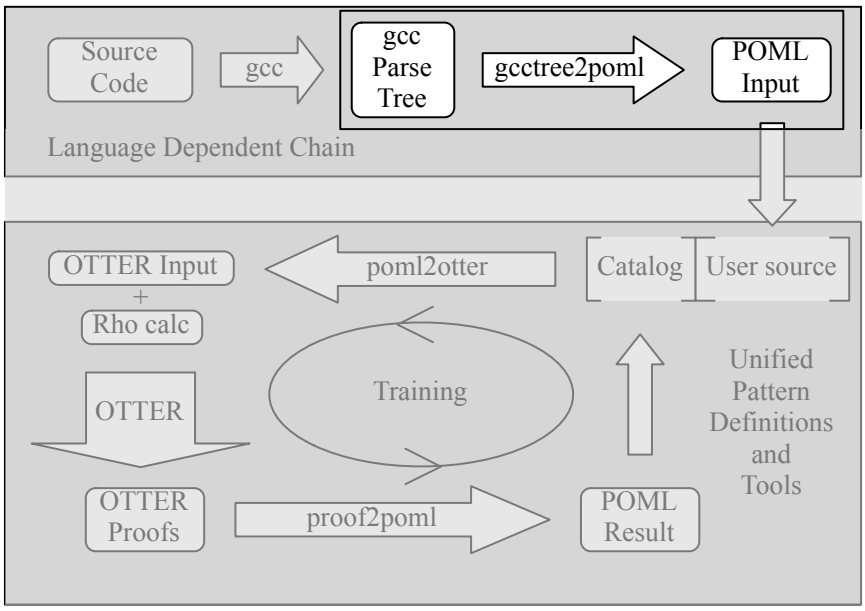


Figure 9.3: SPQR `gcctree2poml` phase

9.1.2 `gcctree2poml`

Once these dump trees have been emitted, they need to be converted to POML files for use in SPQR. `gcctree2poml` accomplishes this for C++ using the `gcc` 3.3 compiler suite and following the methodology described in Chapter 8. Appendix E provides an overview of the internal format of the `gcc` dump tree format, and some issues that arose when implementing `gcctree2poml`. `gcctree2poml` is currently only tested against version 3.3 due to the undocumented differences between versions of the compiler. Future alterations of the tool will include validating it against newer versions of `gcc`. Figure 9.3 illustrates the portion of the SPQR toolchain that `gcctree2poml` occupies.

`gcctree2poml` is a tool written in the Python language, using the Python 2.3 reference standard, and is therefore highly platform-independent. It is intended to be used within a shell environment, such as `tcsh` or `bash`. As such, it can be quickly integrated with a GUI front end if necessary. I will describe here the *use* of `gcctree2poml`. For implementation details beyond this discussion, I refer the reader to the SPQR website noted at the beginning of this chapter, where the source code is heavily documented. I encourage others to use the parsing capabilities of this tool to perform their own research on the `gcc` dump tree files.

`gcctree2poml` is run in its most basic form by giving it the name of the original source file from which the `gcc` dump files were generated. If, for instance, `Sourcefile.cpp` were compiled by `gcc`, and the files `Sourcefile.cpp.tu` and `Sourcefile.cpp.class` were produced, then `gcc2poml Sourcefile.cpp`

would convert these output files to a POML file named `Sourcefile.cpp.xml`. The `.tu` and `.class` files would be moved to a new directory, the workspace directory, that in the above case would be named `Sourcefile.cpp.g2p`. This allows `gcctree2poml` to keep an eye on whether a possibly computationally expensive run is required or not during batch processing, as well as providing a central location to place diagnostics files and other tool-oriented data. This directory may be expunged safely at any time without loss of critical data, since the `.tu` and `.class` files can always be recreated via `gcc`, and the POML file safely resides in the same directory as the original source file.

Several options are available for controlling the POML emission and can be produced at the command line by invoking `gcctree2poml` with the `-h/--help` option. `-o/--output` allows the user to indicate a file for POML output to be written to other than the default of input_file_name.xml. `-f/--filter` filters out the artificial code elements generated by `gcc`, as well as stripping out all declarations and definitions in the `std` namespace. This can result in extremely large reductions in the size of the resulting POML file, as well as cutting the runtime of this phase by 18% on average. `-e/--encode` writes the POML file with all code element names replaced by random, but consistent, strings. A dictionary file is emitted as input_file_name.decode, which contains the mappings. This allows a user to send a POML file containing a representation of proprietary source code to a public SPQR analysis server without fear of compromise of intellectual property. Only the holder of the appropriate `.decode` files will be able to map the resulting POML output back to the original code elements. Finally, `-v/--verbose` provides high level progress reports, as opposed to the silent default.

A number of options are also available for those wishing to understand or extend the internals of `gcctree2poml`. All file output from these options will be placed in the `gcctree2poml` workspace directory. `-d/--debug` emits basic `gcc` tree node information to the POML file, allowing a POML results file to be compared with the original `gcc` dump file more easily. `-p/--profile` produces a Python profiling output file for each of the five basic phases internal to `gcctree2poml`. This is of limited use except for attempting to optimize `gcctree2poml`. More interesting to the end user will be the `-t` option for enabling timing output to be written to the console. The raw timing information, appended to a running log file named `Sourcefile.cpp.time` in the `gcctree2poml` directory, is dumped via the `-T` option. This file is useful for automated analysis of runtimes. Extremely detailed debugging can be enabled by using a combination of `-l/--level` and the ten debugging passes outlined in the online help. This is highly verbose, and can increase easily the runtime by an order of magnitude or more.

During normal use, an engineer will simply invoke `gcctree2poml` with the name of the original source file, and add the filtering flag, or allow SPQR's support scripts to handle this.

9.2 Inference

The core of SPQR is the inference phase. This is where the judgments of ρ-calculus the conceptual building blocks of the EDPs, and the facts derived from the source code come together. The expressive power of POML becomes evident at this point, as it is used to describe both the EDPs and the fact base. This is also where language independence begins in earnest. No assumptions are made after this point about the source language or languages.

I will present here the choice of inference engine, OTTER, the conversion of the ρ-calculus judgments into OTTER's input format, and an XSLT based tool for converting the fact base and EDPs from POML to OTTER's input format.

9.2.1 Otter

There are a number of approaches I could have chosen for the inference phase of SPQR, including using Prolog (Deransart et al., 1996), using an inference engine such as OSHL (Plaisted and Zhu, 2000), using a deductive database (Elmasri and Navathe, 2000), or even writing a custom solver in a more traditional language. I chose to investigate automated theorem provers (Plaisted, 1999) as the general class of existing software packages for four reasons: I wanted an engine that was robust, established, powerful and well-documented. Robustness was important since it would be one less piece I would have to maintain or implement. An established engine with a proven track record would be convincing evidence of its validity. Deductive power would be critical, because, at the beginning of this project, I was unsure what inferences would be asked of it, and I wanted it to be capable of a variety of algorithmic approaches. Documentation would prove to be of utmost importance during the implementation process, particularly with the need for flexibility of approach.

Automated theorem provers, or ATPs, are a generalized class of inference engines that are designed to facilitate human-assisted production of algorithmic proofs (Plaisted, 1999). As such, they have a strong need for performance capable of near-interactive use, and must support a wide range of possible strategies for inference. As a class, they exhibit the algorithmic flexibility I was looking for. The OTTER ATP from Argonne National Laboratories (McCune, 1990) is a well-regarded and established system (Wos and Pieper, 2003) with extensive documentation (McCune, 1990; Kalman, 2001).

It has been asked why an ATP was considered crucial to SPQR, instead of a more common solution such as Prolog, or writing a simple graph walking system (Beyer et al., 2005). Simply answered, it is a matter of convenience and speed. SPQR is a research-oriented toolkit, and needs to support a variety of

styles of deductive reasoning to find the right balance of power and conciseness. A custom solver may be written in the future, but, even if such as solver were written in a deductive language such as Prolog, it would require substantial regression testing of the solver logic if a core rule of ρ-calculus or the solution criteria were altered. Using a proven, existing solver reduces the testing load to the concepts introduced in this research.

Using a graph-walking solver would necessitate a complete re-walking of the graph when new facts about a system were introduced. Each new fact would require a complete re-checking of the graph for new potential hits for the search criteria. Using an ATP, this is avoided, and the rules on which inferences are based become highly flexible. A solver performing walks on a graph to establish the presence of relationship groupings would require coding and testing of each algorithm. The flexibility of an ATP results in a system that can be quickly adapted to new techniques and approaches with confidence. As the concepts of SPQR evolved, having a flexible, proven, and powerful tool such as OTTER at hand was invaluable.

Relational databases are a flexible solution, but they simply cannot efficiently handle the inferences I require (Elmasri and Navathe, 2000). The current research in *deductive databases* (Ramamohanarao and Harland, 1994; Minker, 1988) could theoretically perform the same functionality as an automated theorem prover, and in many cases may provide a strong benefit in the form of persistent storage of factbases for large systems or reuse libraries. I expect that deductive databases will prove to be a benefit for SPQR in the future. For now though, OTTER is the best solution at hand.

9.2.2 Otter inputs

OTTER input is composed of a series of commands to set up the solving methods it should use, provide resource allocation limits, and list any number of rules and facts about a system. These are most easily fed to OTTER through the command line via pipes or file redirection, such as `cat inputfile.otter | otter` or `otter < inputfile.otter`. While OTTER can accept a few different styles of input, I chose to use a simple normative clausal form throughout SPQR, simplifying the production of `poml2otter`, as well as offering more straightforward diagnostics and debugging during SPQR's development. For use in SPQR, OTTER takes input that sets up the algorithmic environment, informs it about the semantics of ς-calculus, ρ-calculus, a descriptor of the pattern being searched for, and the facts derived from the source code. The OTTER manual is the reference document for the following discussion, and I refer the reader to it as necessary. OTTER is a highly complex and rich tool, and I currently use only a subset of its potential. Even so, it would be redundant to replicate a full treatise on even that subset here.

I will provide just enough information to make the ensuing discussion of conversion from ρ-calculus to OTTER meaningful, and refer the reader to the above references for a complete discussion of OTTER's capabilities.

The basics of an OTTER input file are the command, the list, the fact, and the formula. Each input element is on a separate line of the file and terminated with a period.

Commands are used to set up a particular environment for solving, and are also used to create new syntaxes for OTTER, to facilitate the entry of specialized input. I use this capability extensively in SPQR, and have created an OTTER syntax that reflects the origins in POML and ρ-calculus. Because of this file format, SPQR OTTER files are still quite human readable, containing a structure that mimics the original source code, and annotations that indicate the originating source file and line number.

Lists are used to partition the search space into established facts and ones still to be taken into consideration. OTTER maintains two basic types of lists that SPQR uses: the `usable` list and the `sos` list. The `usable` list contains clauses that can be used to create inferences. These are clauses that have been previously established and analyzed, and are deductive results. The `sos` list, on the other hand, is the 'set of support' list. It is essentially the list of clauses being held in wait for consideration. Clauses are selected from the `sos` list one by one, and moved to the `usable` list, where they are checked for use in new inferences against all other clauses in that list. Any new derivations are added to the `sos` list, and a new clause is then selected from there. Lists can be appended to at any time in the input. This becomes invaluable when bringing together disparate files into one search.

Facts are clauses that contain only literals, but no variables. The data extracted from the source code base via `gcctree2poml` will contain only facts, and be expressed as such. Parentheses, the AND operator (`&`), the OR operator (`|`), the conditional operator (`->`) and the negation operator (`-`) all act as expected.

Formulas provide the core of defining a deductive semantics for OTTER. Again, there are multiple types of formula lists, but SPQR uses only the `usable` form. At all times, every inference should be possible, and placing formulas in the `sos` list would only delay proper derivations. A formula looks much like a fact clause, but has a qualifier and variable list preceding it.

One peculiarity should be noted, however, as it affects the end-user to a certain degree. OTTER interprets *any* literal whose name starts with one of ['u'-'z'] to be a variable, not a literal, even when found in a fact clause. Because such identifiers are quite common in programming, the `gcctree2poml` tool searches for such identifiers and prepends an 'O_' to them as a guard for OTTER. Properly, this should be done in the `poml2otter` phase and stripped out during reporting of results, but for now

```
set(auto).
op(700, xfy, contains).

formula_list(usable).
all f g h ( f contains g & g contains h -> f contains h ).
end_of_list.

list(usable).
a contains b.
b contains c.
end_of_list.

list(usable).
-(a contains c).
end_of_list.
```

Figure 9.4: Example OTTER input

it exists in the `gcctree2poml` phase. Until SPQR can be enhanced to support this, the user should mentally ignore such prefixes when mapping results back to the original source code.

A sample of OTTER input can be found in Figure 9.4. The first command tells OTTER to use its best judgment in deciding what solving approach to use after scanning the input file. I found this to be an invaluable tool for investigating various methods when implementing SPQR. The second command defines a new operator, `contains`. The numeric operand is a precedence indicator, and the second argument indicates the associativity, in this case right-associative. Please see the OTTER documentation for details on the `op` command. The next formula sets up `contains` as transitive. Two facts are then provided, and finally a negated form clause is added as a search clause to test the transitivity.

Saving this file as `input.otter` and running this as `otter < input.otter` results in the proof listed in Figure 9.5 sent to the console. This is just a small portion of the full output, but it is the relevant information, the proof. Each clause is numbered, and the brackets on the left indicate how and from what the current clause was derived. An empty list states that it is a primary clause from the input. Clause 5 was derived using hyperresolution from clauses 4, 1 and 3. The refutation in clause 6 was derived from clauses 5 and 2. The sub-clause notation provides information on which literals were unified to form the current clause. More information on the proof format can be found in Chapter 16 of (McCune, 1990).

With these basics in place, I can discuss the intricacies of mapping the expressive denotational semantics of ρ-calculus to the first-order logic system of OTTER.

```
---------------- PROOF ----------------

1 [] -(x contains y)| -(y contains z)|x contains z.
2 [] -(a contains c).
3 [] a contains b.
4 [] b contains c.
5 [hyper,4,1,3] a contains c.
6 [binary,5.1,2.1] $F.

------------ end of proof -------------
```

Figure 9.5: Example OTTER proof

9.2.3 Mapping ρ-calculus to Otter

During the mapping process, it quickly became evident that the intricacies of ρ-calculus would be best expressed in OTTER as several well-defined pieces. There are five input files to OTTER covering the definitions of ς-calculus and ρ-calculus, the transitivities and inference rules for ς-calculus and ρ-calculus, and a file generated to handle deeply nested constructs involving inheritance and superclasses. For reference, these can be found in Appendix G.

This OTTER input form for ρ-calculus is not expected ever to be manually generated. It is used by the XSLTs to create automatically the proper files as needed. I include it here because it illustrates some of the decisions that must be made when implementing SPQR with a particular ATP.

The decision with the most ramifications was whether to use functional or infix expressions. I chose to mix these, using the most natural form when appropriate. For instance, the typing operator ':' from ς-calculus could have been defined in OTTER input as `isoftype(anObject, TheType)`, but, instead, I chose to define the colon as an operator so that it would more naturally read as `anObject:TheType` and closely correspond to the original mathematical notation.

On the other hand, ς-calculus's update operator, $\Leftarrow$, is expressed as a suite of functional forms due to the issues involved in assigning it an appropriate precedence as an operator. The basic form of an update is therefore `update(lhs, rhs)`. Functional form clauses do not need a definition in OTTER, their definitions, including -arity, are automatically deduced by OTTER during input.

ς-calculus Syntax

The `sigmaops.otter` file defines the operators that allow for simple production of ς-calculus, and by extension, ρ-calculus, within OTTER files. The type-of operator of ς-calculus, ':' , is used directly within OTTER, as in `anObject : TheObjectsType`. A synonym is proved with the `oftype` keyword.

Unfortunately, the selection operator of ς-calculus, the period, is used in OTTER as the end-of-input-clause delimiter, so I defined the operator `dot`. This leads to constructs such as *anObject.someMethod* being represented as `anObject dot someMethod`. A bit verbose, but still very readable.

Class descriptions are created by providing operators for inheritance, method declaration and definition, and tying a class-object to a class type. *Subclass* <: *Superclass* becomes `Subclass inh Superclass` by way of the `inh` operator. A method m such that $m \in \mathbf{meth}(AClassObj)$ is declared with `AClassObj declares m` and defined with `AClassObj defines m`. All instances of a definition will have a corresponding declaration, but the reverse does not hold. This dichotomy is to allow for methods that are abstract, as well as to provide a practical way for OTTER to distinguish between methods and fields when necessary. If *AClassObj* and *AClassType* are semantically tied, such as with the POML `poml:isclassobjfor` element, then `AClassObj isclassobjfor AClassType` performs the same task in OTTER.

In an effort to keep OTTER files concise, as with POML in Section 7.1.2, subclasses can either explicitly or implicitly define inherited, but not overridden, methods and fields from superclasses. I chose to use the implicit form, and, instead, to provide inference rules that create the necessary relationships during execution of OTTER. The `inherits` operator performs the same function in OTTER as in POML, associating a subclass with a superclass' implementation. See the discussion on ς-calculus inferences below for the derivation rules.

As stated above, other basics of ς-calculus are not represented in operator form in OTTER, and therefore do not need to be defined in the input, but I will describe them here. First, the parameter passing styles for method calls, and then perhaps the most important functional form, the update operator, $\Leftarrow$. There are two update forms, depending on the nature of the right-hand side of the operator.

In POML, parameters and return values are tagged as being passed by copying, also known as pass-by-value, or by mapping, i.e. pass-by-name. The same need exists in OTTER, but is done via a functional form expression that can be dropped in where needed. `{copy,map}in(method, keyword, parameter)` accomplishes this for input parameters. One of `copy` or `map` is used. Similarly, for return values `{copy,map}in(method, parameter)` defines the proper relationship.

If a target is being updated by a simple object or field, then `updatebyfield(lhs, rhs)` corresponds to POML's `poml:update` element with no internal `poml:fromcall` tag. If that tag is present, however, then the right-hand side is a method invocation, and the `updatebycall(lhs, rhs)` form is used. Additionally, if there are parameters to that call, a semantic tie may be required between the update and

ς-calculus		OTTER
$anObject : AType$		`anObject : AType`
	or	`anObject oftype AType`
$Subclass <: Superclass$		`Subclass inh Superclass`
$anObject.someMethod$		`anObject dot someMethod`
$lhs \Leftarrow rhs()$		`updatebycall(lhs, rhs)`
$lhs \Leftarrow rhs(p)$		`updatebycallparam(lhs, callstyle(rhs, p))`
$lhs \Leftarrow rhs$		`updatebyfield(lhs, rhs)`

Figure 9.6: Example of mapping fundamental ς-calculus elements to OTTER

the parameters for proper inference of the $<_\kappa$ reliance in certain situations. These are generated via `updatebycallparam(lhs, callstyle(rhs, kw, param))`. The callstyle is as above, one of `copyin` or `mapin`. I do *not* use equality in OTTER as a signifier of an update. SPQR is concerned with only relationships, not runtime semantics. Update is considered a relationship in ρ-calculus, and not a logical equivalency.

In the above cases, all but {`copy,map`}`in` could be defined as operator forms. To be truthful, I find `method returnsbycopy retval` easier to read than `copyout(method, retval)`, but keeping it in a functional form mirrors the necessary form for input parameters. These format decisions may be revisited at a later date. By having the OTTER input files be automatically generated from POML, only these hand-defined input files and the `POML2Otter.xsl` XSLT need to be altered to adapt to a new format.

The basics of ς-calculus as represented in OTTER are shown in Figure 9.6. The more convoluted elements such as `isclassobjfor` that do not have simple ς-calculus representations should be understandable from example files. The equivalent POML form for each construct can be found in Chapter 7.

ς-calculus Judgments

Because SPQR and ρ-calculus deal primarily with relationships, the only judgments from the core ς-calculus that I add to OTTER are the type-related transitivities and inferences, in the `sigmacalc.otter` file. Instances of `declares` and `defines` need to be propagated from superclasses to subclasses, as each subclass will automatically inherit such relationships or provide their own overriding versions. If there is no overriding of the method, then these inferences will ensure proper relationships begin inferred.

Inheritance of type relationships need to be made explicitly transitive, in that if $A <: B$ and $B <: C$

then $A <: C$ through subsumption. This is explicitly added to OTTER.

There is one artificial construct present in the OTTER input files that needs to be addressed - the `this` keyword. It is used as a demarcation between the enclosing scope of the method and the internal method pieces. This will be used in the below discussion of ρ-calculus judgments, but the added complexity is worth it for a particular type of required inference.

The `this` construct requires that, for every defined method in a class, an artificial `this` field of the same type as the class is created. In addition, all instance fields are propagated to appear to be field of the `this` construct.

Also in `sigmacalc.otter` are a series of demodulators (McCune, 1990) that create an easy, if not fast, way for OTTER to determine similarity of two constructs at various levels of detail. The most commonly used is `dciseq`, for 'dot chain is equal', which determines if two given chains of the `dot` operator are equal under the lexicographic equality approach that I have previously discussed. This extends the OTTER evaluable symbol `$OCCURS` (McCune, 1990, pg 35-36) to be aware of the length of the dot chains. This length determination is done via the `dclen` demodulator.

Additionally, the `leftdot` and `dotright` demodulators allow the extraction of the leftdot and the dotright of any given term in an SPQR run. This is used in the above length determination.

Because `dot` is left-associative, only `leftdot` is of any real use at the moment, since for any given construct `a dot b`, `b` is directly extractable, while `a` may be a chain of indeterminate length.

These demodulators are extremely useful, but adding demodulators to an OTTER run causes a tremendous slowdown. It is primarily for this reason that the inference rules are kept separate from the operator definitions in the header files, allowing some flexibility in how OTTER is invoked.

ρ-calculus Syntax

The most important definitions in ρ-calculus are the reliance operators. Each of the four base reliance operators are defined, as well as all of their permutations under similarity. The four bases are the Greek letter mnemonics used for each: `mu`, `phi`, `sigma`, `kappa`. Combinations of `ld`, `ls`, `rd` and `rs` are appended to indicate leftdot or dotright occurrences of similarity or dissimilarity. Figure 9.7 shows some examples of how this maps.

In addition to this, ρ-calculus defines the context relationship from Section 4.1. This is defined in OTTER as the `iscontextfor` operator. $\|stmt\|_{method}$ then becomes `method iscontextfor stmt`. This order inversion is an unfortunate legacy issue which should be rectified at a later date.

ρ-calculus	OTTER
$object.m <_{\mu} object2.n$	`object dot m mu object2 dot n`
$object.m <_{\mu}^{+\cdot -} object.someMethod$	`object dot m mulsrd object dot someMethod`
$object.f <_{\kappa}^{\circ\cdot +} anotherObject.f$	`object dot f kappars anotherObject dot f`

Figure 9.7: Example of mapping ρ-calculus reliance operators to OTTER

ρ-calculus Judgments

The file `rhocalc.otter` includes the transitivities introduced in Section 4.3. Most of these are self-explanatory, such as the two halves of Figure 9.8. The former is the ρ-calculus representation of an application of the dotright similarity of Equation 4.60 to a mu-form reliance operator, and the latter is the OTTER representation. This illustrates the use of the OTTER evaluable `$ID` to implement a lexicographic equality algorithm for similarity determination. The `$ID` evaluable works well for dotright constructs, but fails for leftdot chains of the `dot` operator of indeterminate length. It is for this reason that the `dciseq` demodulator was created. If the `dot` operator were right-associative, then `$ID` would be sufficient for leftdot similarity, and `dciseq` would be used for dotright.

$$\frac{ob1.mu1 <_{\mu} ob2.mu2,\ mu1 \sim mu2}{ob1.mu1 <_{\mu}^{\circ\cdot +} ob2.mu2}$$

```
all ob1 ob2 mu1 mu2 (
        ((ob1 dot mu1) mu (ob2 dot mu2)) &
        ($ID(mu1, mu2)) ->
        ((ob1 dot mu1) murs (ob2 dot mu2))
).
```

Figure 9.8: Representations in ρ-calculus and OTTER of a $<_{\mu}^{\circ\cdot +}$ derivation

Other translations of ρ-calculus judgments have multiple forms that need to be addressed. For instance, Equation 4.6 relies on the return value from the called method. According to the subsequent discussion in Section 4.2.2, however, it is apparent that two forms of this equation are needed, one for calling by call-by-value, and one for call-by-name. While these can be unified in the formal notation, both POML and the OTTER input format require them to be expressed in both forms. Some rules are best expressed using a logical 'or' to connect the subclauses, such as in Figure 9.9, which shows the two clauses that correspond to Equation 4.6. Others are more readable if separated into two clauses. Several rules need one of these conversions, and these are annotated in the file.

```
all context calledmethod keyword input (
    context iscontextfor mapin(calledmethod, keyword, input) |
    context iscontextfor copyin(calledmethod, keyword, input) ->
    context phi input
).
```

Figure 9.9: Multiple-form representation of Equation 4.6

```
all class subclass method (
    class defines method &
    subclass inherits class dot method ->
    subclass defines method
).
```

Figure 9.10: Inheritance of implicitly defined method from superclass

Expansion of implicit inheritance and subclass items

In Section 7.1.2, I mentioned how implicitly inherited methods and fields could be left implicit, allowing for both derivation of an explicit form at a later date and reducing POML and, therefore, OTTER file size. This derivation occurs during the inference generation phase in OTTER. There are two cases where explicit forms need to be derived: inheritance of implicit elements and direct access of the elements of a superclass.

In the case of implicit inheritance, the `poml:inheritedmethod` and `poml:inheritedfield` elements are converted to complete definitions of the original implementations. This is accomplished by the rules in the `INH` block. For instance, the most basic conclusion to be drawn is that if the superclass defines a method, then the subclass has that method defined as well. This is defined in the `INH.def` rule, shown in Figure 9.10.

At first glance, this may appear to be a slide backwards to a record-based definition of object type inheritance, but this is done on the fly using inference. If the superclass is altered through dynamic inheritance, then the implementation body will be changed as well. By allowing this dynamic typing, the deductive analysis does not restrict the object model to a static inheritance model.

A similar approach is taken with the direct invocation of superclass method implementations. The `poml:directlycalls` element is translated as a necessary input clause to judgments in the `SUP` block of rules. The basic approach is that the reliances in the body of the superclass method m are translated to use a new method name, nm, that does not appear in the set of defined methods of the subclass. Anywhere the superclass name appears, it is replaced by the subclass name, and anywhere the superclass

method name appears, it is replaced by the new method name. This term rewriting is a common approach in automated theorem provers. Unfortunately, in OTTER, this is highly time-intensive, to the point of being impractical on input files of the size that SPQR uses. Also, I was unable to determine how to force OTTER to do a rewriting on an entire clause as a whole, instead of partial rewritings, which led to false inferences.

Assume for example that a rewrite is defined for $A \rightarrow B$, and all instances of A will be replaced by B. Now, given the clause $A <_\mu A$, and a demodulator to perform the rewriting, there will be two possible outcomes: $A <_\mu B$, or $B <_\mu A$. Each of these will then lead to the final, desired inference of $B <_\mu B$. Due to the way OTTER handles the results from demodulators, however, the intermediate clauses will be placed in the `sos` list, and left to produce new inferences. A regular judgment rule will handle this so that the final form is derived directly. The resulting rule will be strongly restricted on what constructs it can handle. The clause $A <_\mu C$ would not result in $B <_\mu C$, for instance, given the above global replacement rule, nor would $C <_\mu A$ results in the desired $C <_\mu B$.

Because of this, three rules are required to properly handle this case: one for each of the possibilities on how A can appear on the left-hand side and/or right-hand side of the $<_\mu$ reliance operator. Unfortunately, even this approach quickly becomes more cumbersome. Not only do the method definitions need to be inherited, but so does every possible relationship within the context of the inherited method. This means that for every rule from the ς-calculus and ρ-calculus input files, there must be corresponding `INH` and `SUP` rules to bring it forward to the subclass in the appropriate manner.

Even this is not enough however, due to the limitations of the `dot` operator in OTTER. `dot` is left-associative, so given a pattern matching requirement such as $\forall x\{x.b\}$, both $a.b$ and $a.c.d.e.f.b$ will match, with a or $a.c.d.e.f$ matching to x. This does not hold true for dotright of the input however. $\forall x\{a.x\}$ will be matched by $a.b$, with b fulfilling x, but $a.b.c.$ will *not* match the pattern rule.

This presents a problem, because the inferences generated by OTTER will expand to the right of scoping chains, and not the left. This means that for each of the many rules already created for the `INH` and `SUP` blocks, there must be a rule for each cardinality of the scoping chain to the right of the construct we want to rewrite. Since this is, in theory, an infinite number of rules, I made a design decision to limit the length of the chain to a pre-set length. This has the potential of eliminating valid inferences, but it is the only practical approach at this time.

In addition, to facilitate detection of the constructs that need to be rewritten, a mildly artificial, yet theoretically sound approach was taken. Because the constructs in question are of the form *class.method* in all cases, the rules look for the keyword `this` as a distinguishing marker from which to derive the

method and *class*, as in a clause of the form *class.method*.`this`.*something*. This results in *method* being exactly one literal, and not a left-associative chain of the `dot` operator, while allowing *class* to be precisely that. This looks like an arbitrary choice, yet, on inspection of the abstract tree generated by `gcc`, it is evident that this is precisely the conceptual construction they use as well in instance methods. Also, going back to ς-calculus, this corresponds to the `self` construct they identify and define. Initially, I was suspect of the `this` construct in the `gcc` dump tree, and attempted to eliminate it, but this need resurrected its appearance in the POML output. Another keyword such as `self` would work just as well, but using the particular keyword of the language under analysis assures that no naming clash can occur.

Since the rules that need to be written for the `INH` and `SUP` blocks are simple but repetitive, I chose to write a tool to generate them as needed. The `genblocks.py` Python script takes a single argument to indicate to what length the rules should be generated. The code for this script can be found in Appendix G.6, and can be inspected for details on the rules and their format. The resulting file is saved as `otherblocks.otter`, and is included only in OTTER runs designed for generating new ρ-calculus inferences. I have found through informal trial and error that, because the number of rules is exponential on the chain length, a chain length of 3 provides good performance, with no detected missed valid inferences in the codebases I have tested against.

An alternative is worth mentioning at this point: make the `dot` operator right-associative. This would result in flipping the above requirements for the rules, such that the *class* chain would have to be explicitly handled. This has one advantage, however, in that the length of such scoping chains is known at source code analysis time. This would allow a tool such as `genblocks.py` to generate *precisely* the rules needed, and let the constructs to the right of the instance determinant keyword grow to their theoretical limit. Of course, that presents its own set of problems concerning performance. See Section 9.2.4 for a suggestion on how to fix such issues using the `$IGNORE` feature of OTTER.

Defining and searching for patterns

Patterns are expressed in OTTER precisely as they are in ς-calculus, taking advantage of the formula form. For example, an instance of the **AbstractInterface** EDP will be represented in ς-calculus as $AbstractInterface(AbstractingObject, abstractMethod)$ and will appear in OTTER as a clause of the form `AbstractInterface(AbstractingObject, abstractMethod)`.

Searching for a pattern, on the other hand, requires converting this clausal form to a negated formula: `all Abstractor operation (-(AbstractInterface( Abstractor, operation ) ) ).`

9.2.4 Environment setup and invocation

Preparing OTTER for inference and searching is achieved in SPQR through including one of two main header files, `Header.otter` and `Inferences.otter`. The former is used by default for searching for a pattern in a fact base, while the latter is used to *generate* the complete set of derivable facts of a system from the original source code to be re-used for each search run. This is accomplished by OTTER being run once with `Inferences.otter`, which includes all five of the above input files, including the computationally expensive demodulators. This triggers a comprehensive production of all ρ-calculus inferences that can be made within the system. The output of this OTTER run is then converted to OTTER *input*, and is fed to each search run in turn.

The search runs are given `Header.otter` as their setup, which only includes the syntax definition files described above. The inferred relationships from the prior run are then input, eliminating the need for continual reproduction of the same inferences. This caching of inferred facts results in a significant increase in performance, and was the primary reason for splitting the previously described input files into operator definitions and inference rules.

In both types of searches, the algorithmic approach is set to simple hyperresolution (Leitsch, 1997). It appears that this is sufficient for the types of inferences being performed at this time. Considerable work went into forming the rules such that forward and backwards subsumption would be minimized. Initially, the similarity inferences in `sigmacalc.otter` was causing serious problems, but the production of the dot construct equality demodulators, along with their inclusion only in inference runs, alleviated this. As a result, a simple and fast approach has turned out to be sufficient and practical.

By default, OTTER searches for only the first proof in a system. Because SPQR wants to find *all* proofs in a system until the search space is exhausted, this is set to the maximum. The obvious danger is that endless chains of inferences might be produced in some pathological cases. Circular chains are not an issue, since, at the first closing of the cycle, the generated clause will be found to be already present and then discarded. This potential problem can be solved by locking OTTER down to chains of no longer than a specific length of inference. The current rules allow the reliance operators to keep spooling out however possible, but they can be limited. OTTER provides a mechanism, `$IGNORE`, for adding information to clauses, both primary and inferred, that does not participate in the inferences. An *inference length* of zero can be added to every primary rule, and this counter can be incremented with each reliance operator inference. Once a certain length is reached, no further inferences would be possible. Not only does this potential change circumvent possible runaway inferences, but it also offers

a potential method for eliminating spurious detected patterns. See Section 12.6.1 for more on this as a possible future enhancement of SPQR. Also see Section 12.5 for how this could be leveraged to provide some intriguing metrics.

9.2.5 `poml2otter`

The practical matter of converting POML files to an input suitable for OTTER is handled by the `poml2otter` tool. This is a simple shell script that uses the `xsltproc` tool described in the introduction to this chapter, and feeds it the POML file to be converted, and the `POML2OtterFacts.xsl` XSLT, which uses a second XSLT, `POML2Otter.xsl`, as its core engine logic. This core file is rather complex, and used by a number of other wrapper XSLTs. I will discuss neither XSLT as a technology, nor how to write a transform file properly, but, instead, refer the reader to other authoritative sources (Kay, 2004), and to inspect the XSLT files, which can be found in Appendix H.

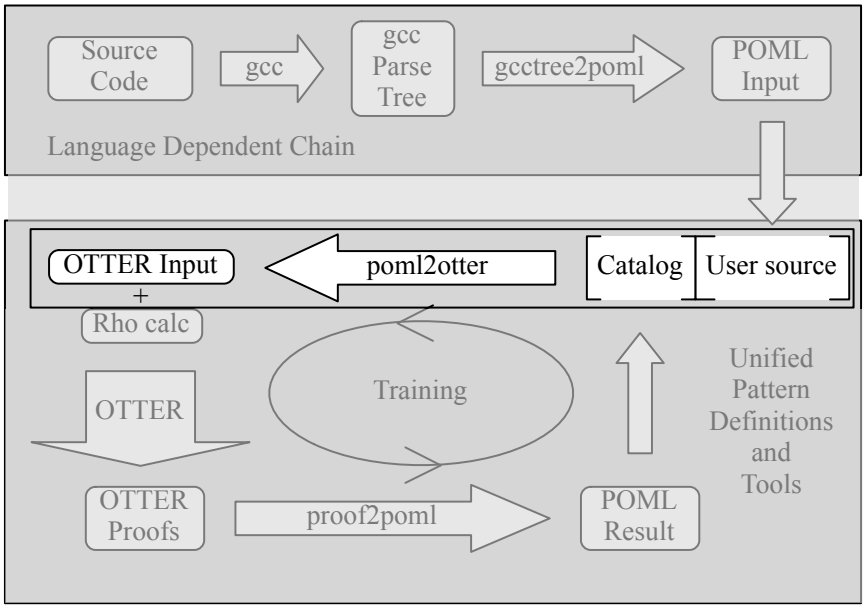


Figure 9.11: SPQR `poml2otter` phase

This tool converts not only the facts extracted from the source code in the earlier tool phases, but also converts the POML representations of the design patterns, both EDP and more complex, as necessary for searching. Under normal conditions, these files are generated at installation time of SPQR, but new pattern POML files can be added to the catalog repository and converted as needed.

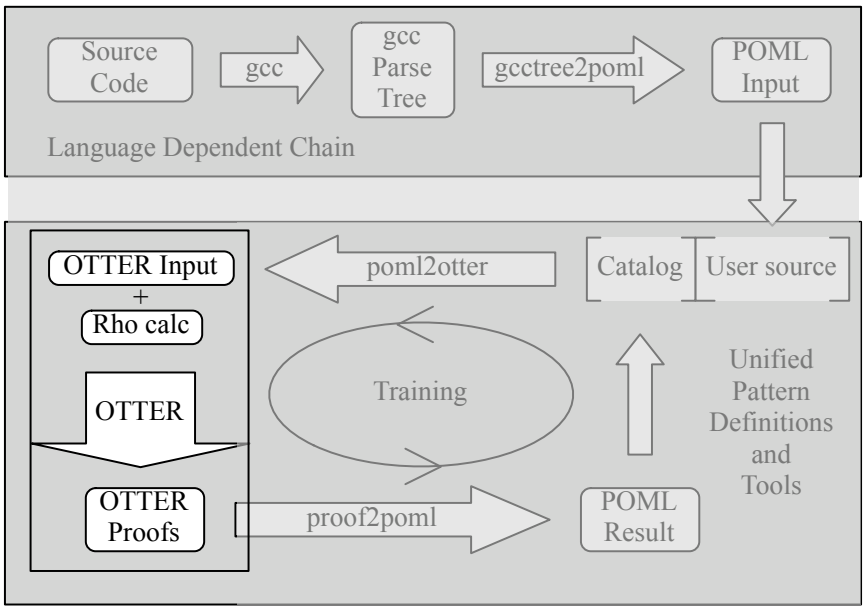


Figure 9.12: SPQR `Otter` phase

9.2.6 Running Otter

As stated above, Otter expects its input from standard console. The input should be: the appropriate header file as described above, with the optional inference results; the output of `poml2otter`'s run on the source code base; the pattern definitions needed; a pattern search definition as a negated formula.

An example run would be: `cat Header.otter SourceCode.cpp.otter RedirectInFamily.otter RedirectInFamilySearch.otter | otter > results.proof`. The file results.proof will hold the raw proof from Otter, which can then be analyzed for the desired results.

9.3 Query reporting

Finally, SPQR needs to report the found patterns back to the engineer. This is done in the current implementation by analyzing the output data from Otter and converting any results back to POML for either conversion into a reporting format, or for inclusion in the patterns search catalog.

9.3.1 `proof2poml`

The `proof2poml` tool is another Python script that reads the output from an Otter run and extracts the proofs, then works backwards through the proof to find the proper pattern name, and role fulfillers. There is not much of note in this tool, it was a very straightforward piece to implement, as the Otter

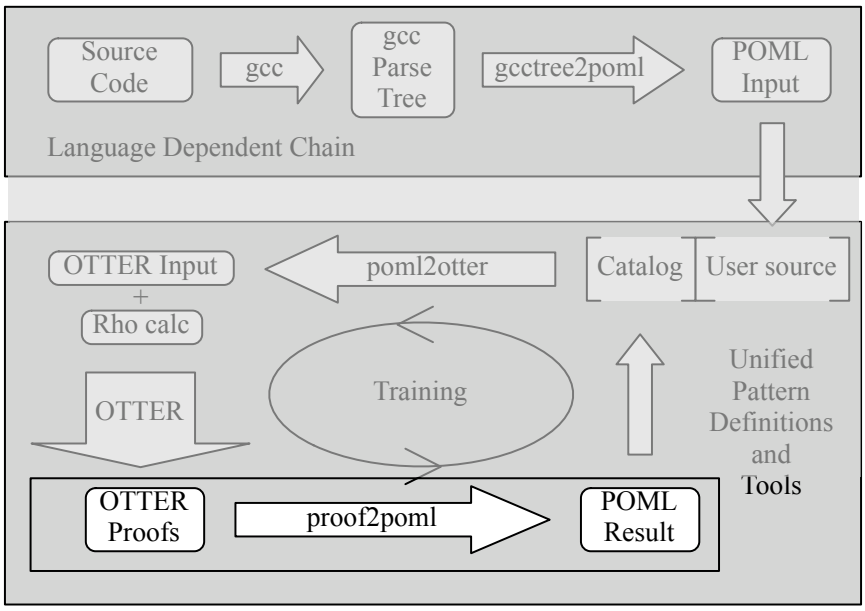


Figure 9.13: SPQR `proof2poml` phase

output format, as shown in Figure 9.5, is very simple. The parsing of the OTTER output is likewise simplistic, relying on Python's strong string and stream parsing tools. The tool is used by invoking it on the command line, and providing an OTTER output file to parse, or a `-` as the filename to indicate standard console input. Output is written to console.

Several interesting additions could be made to an ATP output parsing tool, however, including the ability to trace back through proofs and construct detailed program traces for further analysis. Right now the only output of SPQR is POML files, and it is assumed that the user will have a further tool for producing a final report. One enhancement expected to be useful is to allow the engineer to have control over the level of detail in the report. EDPs will be ubiquitous throughout a system of even moderate size, and while these form the important building blocks, they are not, in general, the patterns that will be of the most interest. A reporting tool that culls low level patterns up to a specified depth in the pattern hierarchy tree will be highly welcome.

9.4 Support scripts

The individual tools that comprise SPQR can be run independently, but it is expected that they will most often be used as a continuous suite. To this end I have produced a wrapper tool in Python, `spqr.py`, that uses the pattern dependency hierarchy to create an efficient and reusable cache of SPQR results in POML format.

The hierarchy of pattern dependence is derived automatically by SPQR by the use of an XSLT that is run on every pattern POML definition in turn when SPQR is installed, or when new patterns are added to the catalog. This transform, `POML2PattDep.xsl`, outputs a list of direct dependencies as a Python list. These code fragments are written together into a file, `PattDeps.xml` that is then read in by `spqr.py` and turned into an internal graph which is then walked in order from most primitive to most complex, on demand.

This hierarchy allows SPQR to do an efficient job of searching over large fact spaces. Take, for example, a system with a half a million lines of code. It would be folly to search for **Decorator** and **Composite** without taking advantage of the fact that they share a mutual dependency on the subpattern **RedirectInLimitedFamily**. Unless the source code changes, the search results from one run of **RedirectInLimitedFamily** will be the same as the next, so these results can be cached and added to both higher-level searches.

A search run is initiated with `spqr.py` by invoking the tool with the name of the original source code file under consideration. As with the previous tools, this allows the underlying implementation to vary in the future as needed without breaking support scripts. A second argument is the name of the pattern that is to be searched for. Four special names are recognized by `spqr.py` in addition to the elements of the patterns catalog: EDP, Intermediate, GOF, and All. These trigger, respectively, comprehensive searches for every pattern defined in the EDP subdirectory, the Intermediate subdirectory, the GOF subdirectory, and finally, a run of all patterns defined in the catalog. For example a search for all EDPs within the POML file produced from `VideoConf.cpp` would be: `spqr.py VideoConf.cpp EDP`.

Assume that a file `~/src/SourceCode.cpp` is compiled by `gcc`, and the appropriate dump files have been written into the same directory. Executing `spqr.py SourceCode.cpp <somepattern>` will create a directory named `~/src/SourceCode.cpp.spqr` if it does not already exist. This directory will contain the intermediate files for analyzing `SourceCode.cpp`. This conveniently allows for post-analysis cleanup when desired. By default, these files include the original OTTER representation of the POML input, the OTTER derived inferences described in Section 9.2.4, and the POML result files for each pattern search. The POML file for the original code is *not* included in this directory, as I see it as a first-class artifact of the code. Also present in the main directory will be the final result file for the latest SPQR search run. Assuming that `spqr.py` has been run on `SourceCode.cpp`, the resultant files and directories will be as listed in Figure 9.14.

The output of `proof2poml` is placed in the `SourceCode.cpp_SPQR.xml` results file. It will only contain the results of the most recent run as it is overwritten with each new search request. The cached results

```
% ls
SourceCode.cpp               SourceCode.cpp.spqr/    SourceCode.cpp_SPQR.xml
SourceCode.cpp.g2p/          SourceCode.cpp.xml

% ls SourceCode.cpp.spqr
SourceCode.cpp.infer.otter              SourceCode.cpp_Redirect.xml
SourceCode.cpp.otter                    SourceCode.cpp_RedirectInFamily.xml
SourceCode.cpp_AbstractInterface.xml    SourceCode.cpp_Retrieve.xml
SourceCode.cpp_Conglomeration.xml       SourceCode.cpp_RevertMethod.xml
...
```

Figure 9.14: Resulting default SPQR files

will always be present as needed.

This hierarchical and caching approach has one distinct advantage: it is extremely well suited for distributed computing. As independent runs of patterns are generated, they can be quickly sent across a server cluster to waiting compute nodes, and the independent results cached into the waiting cache directory when complete. A central disk space would alleviate even the need for manual transport of the initial POML or Otter input. I have not implemented a distributed version of SPQR, but see Section 12.6.6 for a discussion of one possible approach for future research. Also see Section 10.2.2 on how the validation of SPQR against real code may have uncovered a weakness in this approach, and why a conglomerated monolithic search methodology may prove to be more efficient in some cases. In either case, the core foundations of SPQR remain unaltered, and this only supports in my mind the flexibility in the toolkit.

Several options exist to alter the behavior of `spqr.py` as needed. As usual, `-h/--help` will produce a simplified version of this information when passed to `spqr.py` on the command line.

Changing the output file for the final results of `spqr.py` from `SourceCode.cpp_SPQR.xml` to another file can be accomplished with the `-o/--output` option. All output files can be repressed for testing purposes with the `--noresults` option.

Simple reporting can be sent to the command line by way of the `-r/--report` option, which will simply print the kind of patterns found during that search, and how many, on the console. This is accomplished by running an extremely simple XSLT, `POML2PatternCounts.xsl`, on the results file. Timing information is available with the `-t/--time` option.

More thorough debugging and diagnostic information is available with the following options. The `-e/--emitcmd` flag will print to console the command being used to invoke the ATP. `-f/--forcerefresh` causes all subpattern searches to be performed by removing their results files before the search begins. A

complete run refresh, including the expensive inference phase, can be triggered with `-F/--forceinput`. `-d/--debug` emits internal diagnostics for the `spqr.py` tool, while the option `-D/--debugatp` sends an appropriate debugging option to the inference engine. In the case of OTTER, this is accomplished by including the file `Debug.otter` in its input.

During the search runs, the output of OTTER is redirected to a temporary file within the SPQR working directory, as `SourcCode.cpp.proof.otter`. These are deleted as a matter of course, but can be retained with the `-k/--keep` option. This is useful for inspecting the internals of OTTER's proofs.

9.5 Training

SPQR now comes full circle, and the output of `proof2poml` can be sent back through as a set of new facts about the system, to move up the pattern dependency hierarchy efficiently, as explained in the previous section. More interestingly, however, is the opportunity for *training* SPQR to handle new code constructs and cases that may be peculiar to a given application domain or organization.

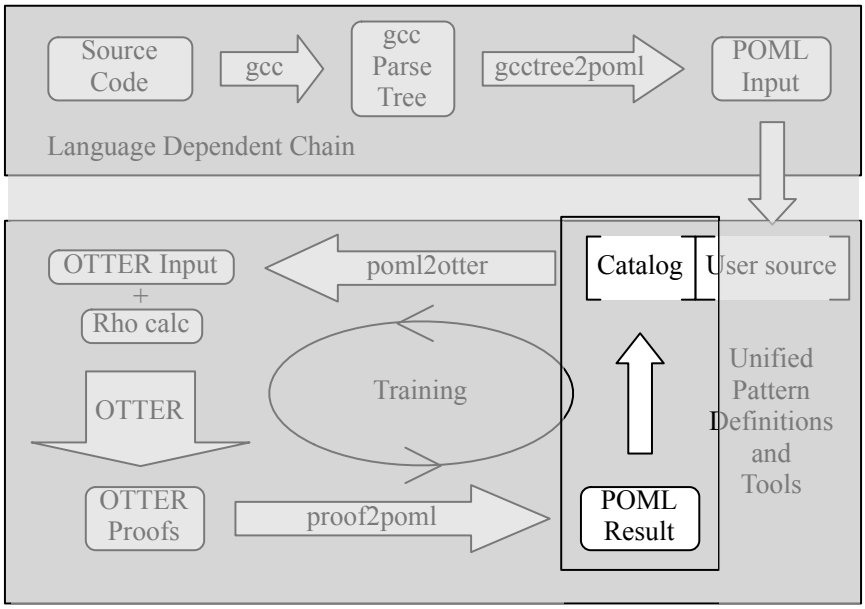


Figure 9.15: SPQR training phase

Source code that is considered exemplary of a candidate pattern can be produced to be analyzed by SPQR. From this, a POML representation can be made, and an exploratory search of known patterns can be made within this candidate code. The results of this can be manually inspected via a chosen reporting format, and elements considered extraneous to the concept the engineers are attempting to capture can be omitted from the POML file as necessary. The result is then a definition of a *new* pattern

description in POML, and only requires the addition of a `poml:resultpattern` element to define it.

By this method, end-users of SPQR can train it for their own needs without having to master the intricacies of ρ-calculus or POML, or produce the definitions from scratch. Only a familiarity with the EDPs and higher patterns is required. Assistance tools can handle the interaction with the engineers, and show them precisely how known patterns are interacting within their new example. New pattern definitions are added to the POML catalog, and all necessary support files are auto-generated by SPQR as needed.

SPQR's ability to train itself, essentially automatically, is precisely how the current catalogs of intermediate and Gang of Four patterns were produced. Once the EDPs were in a format deemed sufficient, example code for the various patterns were created, and SPQR was used to explore their composition. The results were inspected manually and validated against the descriptions of the patterns found in the literature. An iterative process was required, letting SPQR's results guide the production of appropriately minimalist code examples, but even the initial analysis results were instrumental in finding the salient relationships in the desired pattern.

SPQR performs the formal analysis required, and frees the programmer to use only informal approaches to achieve a formal definition.

An important feature of this approach is that, as new low-level patterns are identified and defined, the canonical examples can be re-scanned for that new pattern, perhaps helping to strengthen and refine the higher-level patterns by describing them in more precise concepts. As with the rest of SPQR, this process is fully automatable, allowing the engineer to concentrate on the concepts, and not the formalisms.

9.6 Exploration vs. Validation

SPQR's training mode is important to both the researcher and the practitioner. SPQR can be used in two main modes, each of which is suited to a particular domain. It is expected that SPQR will normally be used in the Validation mode, where a specific pattern is requested to be searched for within the input source code. Any subpatterns will be searched for in turn as needed. An expanded version of this, however, can be triggered by asking the `spqr.py` tool to look for the pattern `All`. This extends the search criteria to include every pattern in the catalog, starting with the EDPs and working up to include all pattern definitions. Obviously, this is time-intensive, but it can be done as a baseline search once during the discovery phase of a system, and incrementally validated as needed.

The result of such a search will be every instance of every pattern known to SPQR. It is useful for

generating large scale documentation of an unknown or poorly understood system. Similarly, it can be used to explore possibilities for refactoring a known system, looking for those opportunities that have grown into place during development, but were obscured.

9.7 Extension of SPQR

As stated in the introduction to this chapter, SPQR was designed to be a flexible toolset. One goal of this chapter is to aid practitioners and researchers looking to move SPQR into new situations, and there are two areas where that is most likely to occur: practical language and IDE support, or theoretical deduction research.

Extending SPQR to handle new languages would require the replacement of `gcctree2poml`, but nothing else.

Exploring alternate inference engines would require the production of new XSLTs to convert POML files to the new input form, mapping ρ-calculus and ς-calculus to that input format as per Section 9.2.3, and a replacement for `proof2poml`, to convert the output to POML.

All pattern definitions would remain untouched, with the exception of ATP-specific additions such as through `poml:atprule`, but even then the new clauses would be adding to, not replacing, the current ATP rules, due to the `atp` attribute of the `poml:atprule`element. The theoretical core of SPQR, the reporting mechanisms, and the definitions of the patterns, from EDPs on up, can be carried through to a number of experimental and practical environments.

While I will leave the formal validation of SPQR for the next chapter, it is important to point out that SPQR contains an extensive regression testing suite for each of the above phases. `gcctree2poml` has over fifty individual examples that illustrate the `gcc` dump tree format and subsequent POML file for discrete language features. These can be invoked sequentially by the `runtests` script in the `gcc2poml/tests` directory. In addition to using XSLT to create searchable formulae for OTTER input, test examples are also generated from the POML definitions for patterns for a quick check of the definition and search formula. This is not intended to test the patterns themselves, but merely to ensure that the OTTER inputs and invocation mechanisms are internally consistent. The OTTER inputs of ρ-calculus and ς-calculus have their own test suite, which invokes OTTER with specific test cases for each rule in the input, and reports success of failure. It is highly recommended that practitioners wishing to extend SPQR follow this model to ensure both internal and cross-tool correctness.

SPQR currently stands unique in the field as a practical and flexible platform for language, systems,

and inference engine research. It was also designed from the beginning to be a useful tool for engineers and non-theoreticians, and in the next chapter I will demonstrate how it can be useful and practical in that arena as well.

Chapter 10

SPQR Validation

SPQR having been implemented, testing of the theory and tool can be performed. Of primary concern is the validation of the concepts and the basic approach from a theoretical point of view. The design requirement for practicality as an assistive toolset requires a critical look at the performance of the tools, however, and a discussion on how they may be improved.

10.1 Testing

I performed four primary validation tests which I will describe in this section. The first utilizes examples of the Gang of Four patterns to train the system as per Section 9.5. Secondly, I will describe the Killer-Widget example that has been used throughout the publications up to this point and which illustrates an isotopic example pulled from industry code. The discussion includes a detailed walk-through of the process from source code to final results. The third test uses a library that has been used in two research projects within the CoLab at UNC and was designed using patterns. The final test is an exploratory search within the entire C++ `std` namespace.

All tests were performed on an Apple PowerBook with a 1.25GHz PowerPC G4 CPU, 512MB of RAM, and on MacOS X 10.3.9. The support infrastructure for the tools was as described in the introduction to Chapter 9.

10.1.1 Training - Gang of Four

Validation of simple concepts should require only simple test cases, and I chose to use the addition of the higher-level patterns as a testing system for the EDPs. Taking advantage of SPQR's training abilities made this a doubly productive test. Not only did it provide validation of the underlying concepts, but it

extended the patterns catalog to contain the definitions of most interest. The resulting definitions can be found in Appendices C and D. Here I will discuss the process and the general lessons learned.

In general, I followed the training approach described in Section 9.5. Using the example structure from each pattern definition, I produced a skeletal source code representation for that pattern. Invoking SPQR in Training mode led to an exhaustive search of all known patterns within the code. The results of each were then manually inspected, and the appropriate identified concepts were retained to create a definition for the pattern being explored.

Adapter has a third form, **Adapter − SuperClass** that is not found explicitly in the Gang of Four text. This was created due to my misreading of the canonical pattern during the initial discovery phase from Section 5.8.3. It was my understanding that an explicit call to the superclass' implementation was required, as with **RevertMehod**, and this is incorrect. Ironically, it was the distinction between **RevertMehod** and **ExtendMethod** that generated the initial discussion on the distinguishing factors between the EDPs, and led to the current axes of determination.

The results of comparing the three forms of **Adapter**, however, are illuminating, and the process can be used to generally refine pattern definitions that may have two or more variants. It is my speculation that the class adapter and superclass adapter forms should be unifiable. The object adapter feels a bit too much like a variant at this stage, and not an isotope. While the syntactic differences between the class and superclass adapters are significant, the general concept is that a class is using a superclass' method implementation, either explicitly, as in superclass adapter, or implicitly, as in class adapter. A specialization of **Conglomeration** that utilizes the fact that a method implementation was implicitly inherited from a superclass may be appropriate here.

This variety of *en masse* training also allows for interesting comparisons between patterns. Consider the **Composite** and **Proxy** patterns from Table 10.2. The intents of the patterns are quite different: **Composite** "lets clients treat individual objects and compositions of objects uniformly" (Gamma et al., 1995, pg.163), while **Proxy** provides "a surrogate or placeholder for another object to control access to it." (Gamma et al., 1995, pg.207)

The results from SPQR, however, show a strong correlation between **Composite** and **Proxy**. In fact, for the canonical example structures as given in Gang of Four, the most obvious difference at the EDP level of analysis is that **Composite** contains an instance of **RedirectInFamily** while **Proxy** contains **RedirectInLimitedFamily**. Interestingly, this slight difference in structure makes a significant difference in capabilities and in intent. Because of this shift, **Proxy**'s RealSubject class stands in as both Composite and Leaf from **Composite**, and the **ObjectRecursion** pattern is lost in **Proxy**. Neither

	Abstract Factory	Builder	Factory Method	Prototype	Singleton
AbstractInterface	2	1	1	1	-
Conglomeration	-	2	2	6	-
CreateObject	4	1	1	4	1
Delegate	4	8	4	63	6
DelegatedConglomeration	-	2	2	6	-
DelegateInFamily	-	-	-	-	-
DelegateInLimitedFamily	-	-	-	-	-
ExtendMethod	-	-	-	-	-
Inheritance	6	1	2	2	-
Recursion	-	-	-	-	1
Redirect	-	-	-	4	1
RedirectedRecursion	-	-	-	-	-
RedirectInFamily	-	-	-	-	-
RedirectInLimitedFamily	-	-	-	-	-
Retrieve	-	-	-	-	1
RevertMethod	-	-	-	-	-
FulfillMethod	4	1	1	2	-
Objectifier	-	1	-	-	-
ObjectRecursion	-	-	-	-	-
RetrieveShared	-	-	-	-	1

Table 10.1: Creational Patterns from Gang of Four

pattern is mentioned in the Related Patterns section of the other in the Gang of Four text, yet there is an interesting parallel that can be shown by SPQR analysis. In essence, both patterns can be thought of controlling access to an object or collection of objects. In the case of **Composite**, the *collection* nature is explicitly what is hidden, while in **Proxy**, the primary focus is on hiding the existence of the other object completely.

Training illustrates another useful feature of SPQR - automated regeneration as new theory is implemented. During the training process, I discovered a minor error in the definition of **Redirect**. If a manual approach had been used to create the definitions of the Gang of Four catalog, this may or may not have required a reanalysis and redefinition of each. Using the automated training approach, however, I was able to determine that the error was not one that altered the definitions as extracted from the code examples, as the results both prior to, and after, the definition alteration, were the same. This principle can be extended, such that if any new low-level definitions are added to SPQR, retraining on an existing canonical form code base can result in a fast reconceptualization of larger definitions, possibly adding previously untenable patterns to the catalog as a result.

In addition, experimental low-level pattern definitions can be tested against a suite of canonical code sources. In this manner, the training can be bidirectional, using known code examples to validate new definitions for concepts under consideration for the base catalog.

	Adapter (Class)	Adapter (Object)	Adapter (SuperClass)	Bridge	Composite
AbstractInterface	1	1	1	1	1
Conglomeration	1	1	1	-	6
CreateObject	-	-	-	-	4
Delegate	1	1	1	3	69
DelegatedConglomeration	-	-	-	-	6
DelegateInFamily	-	-	-	-	-
DelegateInLimitedFamily	-	-	-	-	-
ExtendMethod	-	-	-	-	-
Inheritance	2	1	2	3	2
Recursion	-	-	-	-	-
Redirect	-	-	-	-	8
RedirectedRecursion	-	-	-	-	-
RedirectInFamily	-	-	-	-	1
RedirectInLimitedFamily	-	-	-	-	-
Retrieve	-	-	-	-	-
RevertMethod	1	-	1	-	-
FulfillMethod	1	1	1	2	2
Objectifier	-	-	-	2	2
ObjectRecursion	-	-	-	-	1
RetrieveShared	-	-	-	-	-

	Decorator	Facade	Flyweight	Proxy
AbstractInterface	1	-	1	1
Conglomeration	1	-	4	6
CreateObject	-	-	3	4
Delegate	1	-	35	71
DelegatedConglomeration	-	-	4	6
DelegateInFamily	-	-	-	-
DelegateInLimitedFamily	-	-	-	-
ExtendMethod	1	-	-	-
Inheritance	4	-	1	2
Recursion	-	-	-	-
Redirect	2	-	2	8
RedirectedRecursion	-	-	-	-
RedirectInFamily	1	-	-	-
RedirectInLimitedFamily	-	-	-	1
Retrieve	-	-	2	-
RevertMethod	-	-	-	-
FulfillMethod	3	-	1	2
Objectifier	3	-	-	4
ObjectRecursion	2	-	-	-
RetrieveShared	-	-	-	-

Table 10.2: Structural Patterns from Gang of Four

	Chain Of Responsibility	Command	Interpreter	Iterator	Mediator
AbstractInterface	-	1	1	2	4
Conglomeration	2	-	-	4	-
CreateObject	-	-	-	2	-
Delegate	2	4	2	33	-
DelegatedConglomeration	-	-	-	4	-
DelegateInFamily	-	-	-	-	-
DelegateInLimitedFamily	-	-	-	-	-
ExtendMethod	2	-	-	-	-
Inheritance	2	1	2	2	3
Recursion	1	1	-	1	-
Redirect	1	1	2	3	-
RedirectedRecursion	1	-	-	-	-
RedirectInFamily	-	-	1	-	-
RedirectInLimitedFamily	-	-	-	-	-
Retrieve	-	-	-	-	-
RevertMethod	-	-	-	-	-
FulfillMethod	-	1	2	2	-
Objectifier	-	1	4	-	-
ObjectRecursion	-	-	2	-	-
RetrieveShared	-	-	-	-	-

	Memento	Observer	State	Strategy	Template	Visitor
AbstractInterface	-	3	1	1	1	3
Conglomeration	10	-	-	-	1	-
CreateObject	4	-	-	-	-	-
Delegate	37	5	2	2	3	2
DelegatedConglomeration	10	-	-	-	-	-
DelegateInFamily	-	-	-	-	-	-
DelegateInLimitedFamily	-	-	-	-	-	-
ExtendMethod	-	-	-	-	-	-
Inheritance	-	2	2	3	1	4
Recursion	4	-	-	-	-	-
Redirect	4	-	-	-	-	-
RedirectedRecursion	-	-	-	-	-	-
RedirectInFamily	-	-	-	-	-	-
RedirectInLimitedFamily	-	-	-	-	-	-
Retrieve	1	1	-	-	-	-
RevertMethod	-	-	-	-	-	-
FulfillMethod	-	1	2	3	1	6
Objectifier	-	1	2	3	1	4
ObjectRecursion	-	-	-	-	-	-
RetrieveShared	-	-	-	-	-	-

Table 10.3: Behavioral Patterns from Gang of Four

This experiment convinced me that, while SPQR is currently highly useful, the inclusion of even rudimentary object-dependency reliances in the EDP catalog will be highly effective for this type of analysis. While method-dependency reliances were accurately and quickly identified and reported, the intricacies of data reliance remained elusive in some cases.

For example, the **Composite** pattern is, essentially, an instance of **ObjectRecursion** with runtime behavior to enable the dynamic building or tearing down of the composite structure, as well as the iteration over the structure for distributing the method calls. This behavior is difficult to determine statically, but is an example of **Collection**, either by the use of an object conforming to a **Collection**, or by the Composite class *being* a **Collection**. There is an argument to be made that the Participants section of **Composite** (Gamma et al., 1995, pg.165) requires the Component class, and, therefore, the Composite class, to be a **Collection** instance. In either case, the iteration over the composite members in the appropriate method would appear as an instance of using an **Iterator** over the appropriate objects with which an **ObjectRecursion** relationship exists. Since **Collection** and **Iterator** are as yet undefined, it is difficult at this time to distinguish this pattern from **ObjectRecursion** reliably.

Furthermore, my speculation regarding the necessity of moving between levels of abstraction for effective training was supported by this test. For example, the **Facade** pattern will require the inclusion of a metrics-based analysis to properly detect an instance with any reliability. The simplicity of the pattern at the structural *and* behavioral levels means that there is little relationship information to use in a quantitative manner. What *can* be determined is, for a given candidate class or object, the set of objects and methods external to itself that call into it, and the set of methods that it calls *to*. This sort of information can easily be extracted from the POML represenations. For a **Facade**, the two sets will be nearly, or exactly, distinct. This type of lower-level coupling analysis can be combined with the conceptual abstractions of ρ-calculus to, perhaps, generate a new EDP. One possible name for such a concept would be a **Gateway**. If an object or class is determined to be a **Gateway**, regardless of how that was determined, or what tools were used, that fact can be added to the facts for a system, and SPQR can then use that information for deeper analysis. Such a pattern may be useful in refining the definition of **Bridge** as well as **Facade**.

The POML representation is what enables a simple XSLT-based tool to perform such an analysis. I foresee SPQR evolving into a suite of independent tools that use POML as a common format. In this way, each tool can provide the analysis it does best, and all other tools can benefit.

Two EDPs appear in none of the Gang of Four patterns used in training: **DelegateInFamily**, and **DelegateInLimitedFamily**. This is not that surprising, as these are EDPs created to ful-

fill the orthogonal axes of discovery outlined in Section 5.8.3. What is more satisfying is that the three remaining EDPs created during that same process, **Recursion**, **RedirectedRecursion**, and **DelegatedConglomeration** *do* appear in the Gang of Four patterns.

This training process produced the catalog of definitions for the Gang of Four patterns, expressed in ρ-calculus, that can be found in Appendix D. I have provided notes regarding the sufficiency of the current definitions, as well as what concepts would be required to complete certain definitions.

10.1.2 Isotopes - KillerWidget

In many ways, KillerWidget has been the driving example for SPQR from the beginning. Based on a real-world example that I encountered in industry in 1996, it inspired the research that led to SPQR.

Three libraries were being developed and maintained by three separate developer groups within a corporation. Each was well-documented, and the behavior was well understood by each group, but an unexpected behavior was being reported by a fourth group that used all three libraries. Though a welcome and useful functionality, it was not reported in any of the documentation for the three libraries.

I was in one of the library development groups. Members of the other two groups and I were intrigued by this behavior that seemed to have developed on its own, quite accidentally, and we decided to tease out the situation. We expected to spend a few hours on it, since we were quite familiar with the system in general, and our libraries in particular.

In the end, we spent approximately 120 man-hours on the problem over the course of a few weeks. Eventually, we were able to deduce that the unique and unexpected behavior arose from an instance of a Decorator design pattern that only formed when all three libraries were brought together. Pieces of the pattern were scattered among the libraries, and, to make things more complex, the pattern did not exactly conform to the structure given in the defining text. It was indeed a Decorator, but one that had a method calling chain through only vaguely related objects, in other words, an isotope.

I will use KillerWidget as an example of the production of an SPQR run, from beginning to end. It is large enough to be interesting, but small enough to be verifiable manually.

The code I use here is not the original source, as I no longer have access to it, but it is derived from my original notes on the project. I have distilled it down to the basics required to illustrate the isotopic principle. A UML diagram for the code is given in Figure 10.1, and the code is shown in Figure 10.2. This is converted to POML by `gcctree2poml`, which I omit as simply too large for inclusion here, and it can be generated by the reader if desired.

Running `spqr.py` on the example converts the POML to Otter input, portions of which I provide

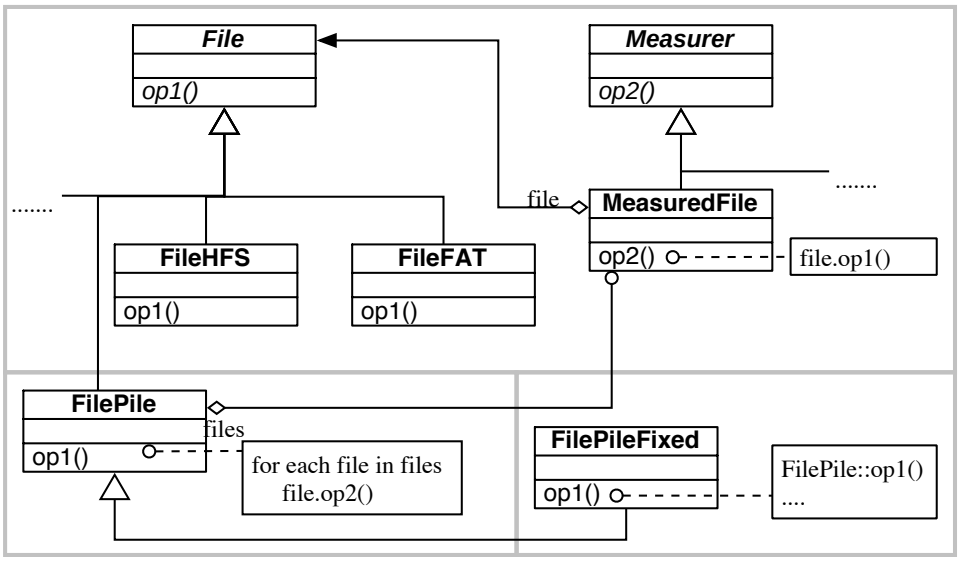


Figure 10.1: KillerWidget as UML

in Figure 10.3. Shown here are only 35 of 239 lines of the OTTER file, but these are just the portions needed for the production of the proof. The comments in the figure are auto-generated by `gcctree2poml` and `poml2otter`. This aids in debugging and in providing examples such as this. The indenting of lines at the end of the example are manually added for formatting reasons here, and are not present in the actual input file.

OTTER produces the proof in Figure 10.4 as part of its output. Here it can be seen that this instance was created from the instances of **ExtendMethod** and **ObjectRecursion** shown as clauses #41 and #47, respectively. Similar proofs exist for the production of those clauses, although I do not include them here. By parsing these proofs, a hierarchy of pattern instances can be built up to facilitate reporting of results to an engineer. At this time, SPQR does not perform this massaging of output, but the opportunity exists.

The **Decorator** pattern was searched for explicitly by SPQR, and the instance was successfully found. The POML output for the pattern is shown in Figure 10.5, and the UML diagram has been annotated in Figure 10.6, using *pattern:role* notation (Vlissides, 1998), to show the mapping of the **Decorator** instance with the original design. Other instances have been omitted for clarity. Note that the intermediate class has been tagged as **Decorator** : **support**. This will become useful when discussing pattern coverage metrics in Section 11.2. The search was initiated and completed, from source code to extracted pattern, in slightly more than 21 seconds, a substantial improvement over 120 man-hours. Note also that this search required no expert intervention or guidance.

```
// Component
class File {
public:
  virtual void op1()=0;
  virtual void op2()=0;
};

// ConcreteComponent
class FileFAT : public File {
public:
  void op1() {};
  void op2() {};
};

// Helper (Sits between Decorator and Component)
class MeasuredFile {//: public File {
public:
  File* file;
  MeasuredFile() { file = new FileFAT(); };
//  void op1() { file->op1(); };
  void op2() { file->op1(); };
};

// Decorator
class FilePile : public File {
public:
  MeasuredFile* mfile;
  FilePile() { mfile = new MeasuredFile(); };
  void op1() { mfile->op2(); };
  void op2() {};
};


// ConcreteDecorator
class FilePileFixed : public FilePile {
public:
  void op1() { FilePile::op1(); fixTheProblem(); };
  void op2() {};
  void fixTheProblem() {};
};


// Initiator
int
main(int, char**) {
  File* fpf = new FilePileFixed();
  fpf->op1();
};
```

Figure 10.2: C++ code for KillerWidget

```
%-----  Class File file=KillerWidget.cpp line=2
         %-----   Method File dot op1 file=KillerWidget.cpp line=4
AbstractInterface( File, op1 ).
         %-----   Method File dot op2 file=KillerWidget.cpp line=5
AbstractInterface( File, op2 ).
%-----  Class FileFAT file=KillerWidget.cpp line=9
FileFAT inh File.
%-----  Class MeasuredFile file=KillerWidget.cpp line=16
         %-----   Method MeasuredFile dot op2 file=KillerWidget.cpp line=21
MeasuredFile defines op2.
( MeasuredFile dot op2 ) mu ( MeasuredFile dot op2 dot this dot file dot op1 ).
         %-----   Field MeasuredFile dot file file=KillerWidget.cpp line=18
MeasuredFile dot file : File.
%-----  Class FilePile file=KillerWidget.cpp line=25
FilePile inh File.
         %-----   Method FilePile dot op1 file=KillerWidget.cpp line=29
FilePile defines op1.
( FilePile dot op1 ) mu ( FilePile dot op1 dot this dot mfile dot op2 ).
         %-----   Field FilePile dot mfile file=KillerWidget.cpp line=27
FilePile dot mfile : MeasuredFile.
%-----  Class FilePileFixed file=KillerWidget.cpp line=35
FilePileFixed inh FilePile.
directcallfrom_to_as( FilePileFixed, FilePile dot op1, FilePile_op1 ).
         %-----   Method FilePileFixed dot op1 file=KillerWidget.cpp line=37
FilePileFixed defines op1.
( FilePileFixed dot op1 ) mu ( FilePileFixed dot op1 dot this dot FilePile_op1 ).
( FilePileFixed dot op1 ) mu ( FilePileFixed dot op1 dot this dot fixTheProblem ).
%-----  Object __GLOBAL__ file=__internal__ line=0
         %-----   Method __GLOBAL__ dot main file=KillerWidget.cpp line=45
__GLOBAL__ defines main.
         %-----   Field __GLOBAL__ dot main dot fpf file=KillerWidget.cpp line=46
__GLOBAL__ dot main dot fpf : File.
__GLOBAL__ dot main iscontextfor updatebycall( __GLOBAL__ dot main dot fpf,
                              FilePileFixed__ClassObj dot __comp_ctor_overload_3 ).
( __GLOBAL__ dot main ) mu ( __GLOBAL__ dot main dot fpf dot op1 ).
CreateObject( __GLOBAL__ dot main, FilePileFixed, __GLOBAL__ dot main dot main_anonvar_0 ).
```

Figure 10.3: Portions of KillerWidget as OTTER input

```
---------------- PROOF ----------------
41 []
ExtendMethod(FilePile,FilePileFixed,op1).
47 []
ObjectRecursion(File,FilePile,FilePileFixed,__GLOBAL__).
57 []
-ObjectRecursion(x30,x31,x32,x33) |
-ExtendMethod(x31,x34,x35) |
Decorator(x30,x31,x32,x34,x35).
58 []
-Decorator(
    xComponent,
    xDecoratorBase,
    xConcreteComponent,
    xConcreteDecorator,
    xoperation
).
1434 [hyper,57,47,41,eval]
Decorator(File,FilePile,FilePileFixed,FilePileFixed,op1).
1435 [binary,1434.1,58.1]
.
------------ end of proof -------------
```

Figure 10.4: OTTER output for **Decorator** instance in KillerWidget

```
<poml:pattern>
        <poml:name>Decorator</poml:name>
        <poml:role>
                <poml:name>Component</poml:name>
                <poml:fulfilledBy>File</poml:fulfilledBy>
        </poml:role>
        <poml:role>
                <poml:name>Decorator</poml:name>
                <poml:fulfilledBy>FilePile</poml:fulfilledBy>
        </poml:role>
        <poml:role>
                <poml:name>ConcreteComponent</poml:name>
                <poml:fulfilledBy>FileFAT</poml:fulfilledBy>
        </poml:role>
        <poml:role>
                <poml:name>ConcreteDecorator</poml:name>
                <poml:fulfilledBy>FilePileFixed</poml:fulfilledBy>
        </poml:role>
        <poml:role>
                <poml:name>operation</poml:name>
                <poml:fulfilledBy>op1</poml:fulfilledBy>
        </poml:role>
</poml:pattern>
```

Figure 10.5: Recovered **Decorator** in KillerWidget as POML

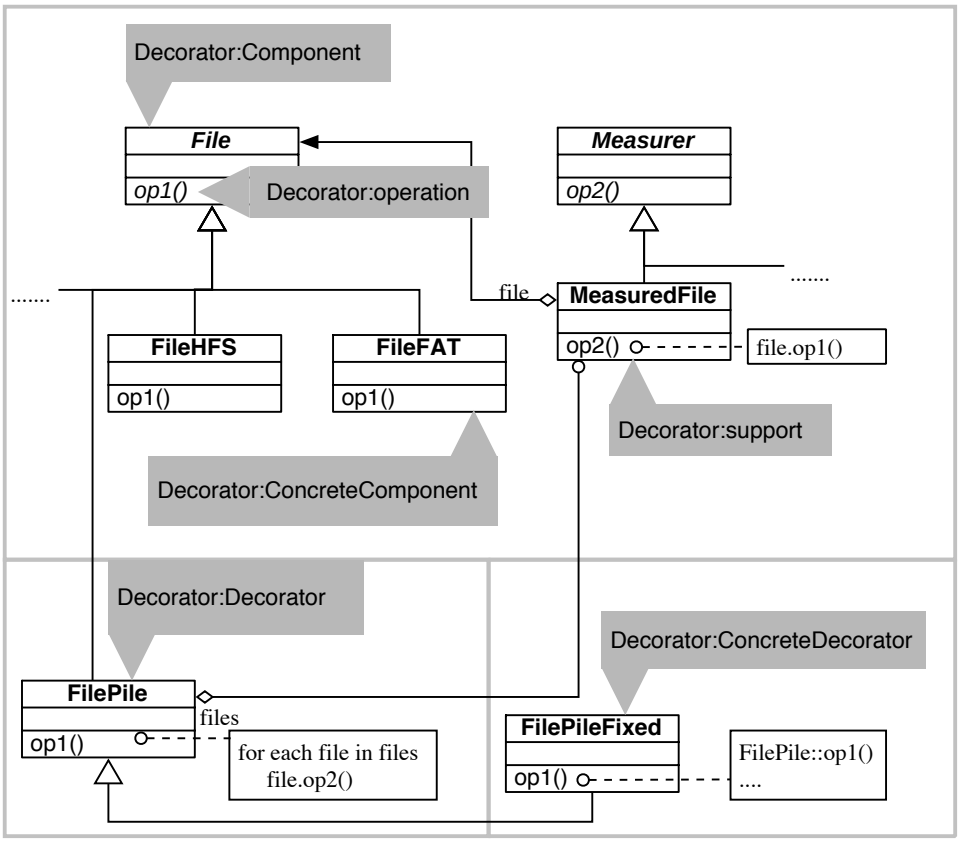


Figure 10.6: Recovered **Decorator** pattern in KillerWidget as UML

I performed further testing to establish the behavior of SPQR and validate the pattern definitions. As a result, in addition to the expected **Decorator** pattern, SPQR found instances of **Bridge** and **Composite**. The relevant clauses are shown in Equations 10.2 through 10.5. The instances of **Bridge** are not surprising, since, as I state in the discussion in Section D.2.2, the definition is so broad as to likely be triggered by a number of instances where the functionality is used unconsciously by developers, but without the explicit intent of the pattern. Likewise, as I discuss in Section D.2.3, without the proper semantics of iteration over a collection, **Composite** is going to be difficult to distinguish from a **Decorator**. Neither of these results is unexpected, and detection of the expected incongruities is

actually rather reassuring.

$$\textbf{Bridge}(MeasuredFile, op2, file, File, FileFAT, op1) \tag{10.1}$$

$$\textbf{Bridge}(MeasuredFile, op2, file, File, FilePile, op1) \tag{10.2}$$

$$\textbf{Bridge}(MeasuredFile, op2, file, File, FilePileFixed, op1) \tag{10.3}$$

$$\textbf{Composite}(File, FilePile, FilePileFixed, op1) \tag{10.4}$$

$$\textbf{Composite}(File, FilePile, FileFAT, op1) \tag{10.5}$$

An interesting incident occurred when initially performing this test. After both a substantial upgrade to SPQR and a simultaneous rewriting of KillerWidget to C++ from the hand-rolled OTTER input used in previous publications, the example stopped passing the test. In my haste, I assumed that SPQR was to blame, and invested a great deal of time verifying the tools and definitions of the EDPs, the mapping of ρ-calculus to OTTER, and every aspect of the tool chain. It turned out that there was a simple typographical error in the C++ code for KillerWidget. SPQR was correct, the expected patterns were *not* in the code, despite appearing to exist on quick review. Once I thought to check the output from OTTER more closely, it became obvious where the bug was. After it was fixed, the test ran successfully. This experience gives me confidence that SPQR will perform as intended as a validation tool for existing code.

10.1.3 Validation - NotificationCenter

TrackerLib is a library for tracking objects in a live video stream. It was developed to support the OvalTine (Stotts and Smith, 2002) and Facetop (Stotts et al., 2003; Stotts et al., 2004) projects, of which I was also a part. Because I designed TrackerLib to use patterns from the beginning, it gives a good example of real-world code, and offers a good validation test for SPQR, as I know the intent of the code, and can manually verify the results.

One subsystem of TrackerLib is the NotificationCenter, a message passing system that is implemented as a **Singleton** pattern. I targeted this as the first validation test from existing production code due to its small size, non-trivial behavior outside the expected pattern, and regular use of standard libraries. The code was left intact, and the `--dump-translation-unit` and `--dump-class-hierarchy` flags were added to the existing `Makefile` used in the normal project production. The library was then made using the present build system to provide the files necessary for SPQR. As intended, the impact on the

production environment was minimal.

NotificationCenter consists of six source code files, each of which was compiled as its own translation unit. This resulted in six sets of `gcc` dump files for `gcctree2poml`, and six POML files. This was also an excellent example of SPQR's ability to combine various POML representations into one analysis factbase. Because the translation units had been successfully compiled and linked during the build phase, it was guaranteed that they were appropriately named and scoped internally as to not cause naming conflicts. Any names that were consistent across translation units, such as `__GLOBAL__`, would also be ensure to point to the same conceptual entity in the final system. These two facts allow SPQR to safely combine POML files from different translation units, libraries, and subsystems, into one large representation for analysis.

To be sure, there is a certain amount of redundancy among such files, and the current unification tool does not attempt to reduce this. It simply concatenates the POML elements within the `poml:system` document element into one new element list. While a reduction in redundancy would be effective in minimizing the conversion time to the OTTER format, it proves to be impractical and unnecessary. First, since translation units may differ in how much information they carry about a certain programming element, a deep search would have to be invoked to check for equality of generalized subtrees. This is not a trivial task. Secondly, as the clauses are read into OTTER, redundant ones are discarded on input, reducing the inference space immediately. A much simpler unification algorithm can, therefore, be implemented.

The resulting unified POML file was then searched for the **Singleton** pattern, and it was successfully found. Providing rather firmer validation of the definitions at the Gang of Four level, an exploratory search was initiated, and, as anticipated, it was the *only* Gang of Four pattern found. There were no false positives of the other patterns, as shown in Figure 10.4. Despite the large number of some EDPs that were found, which is to be expected, the number of higher level patterns quickly winnowed down. I expect that this pattern will be replicated in further testing.

10.1.4 Exploration - `gcc std`

The final test was the biggest stress test performed to date on SPQR, and demonstrates the robustness of the approach. I created a simple C++ file that included every standard header from the C++ standard, essentially pulling in the entire `std` namespace as implemented in `gcc`. This created a rich environment in which to attempt an exploratory search of patterns.

The sheer amount of code brought in and subsequently generated from templates by `gcc` was a true

	NCTest_SPQR
AbstractInterface	-
Conglomeration	96
CreateObject	48
Delegate	1334
DelegatedConglomeration	48
DelegateInFamily	-
DelegateInLimitedFamily	-
ExtendMethod	-
Inheritance	3
Recursion	5
Redirect	75
RedirectedRecursion	-
RedirectInFamily	-
RedirectInLimitedFamily	-
Retrieve	18
RevertMethod	-
FulfillMethod	-
Objectifier	-
ObjectRecursion	-
RetrieveShared	12
AdapterClass	-
AdapterObject	-
AdapterSuperclass	-
Bridge	-
ChainOfResponsibility	-
Command	-
Composite	-
Decorator	-
FactoryMethod	-
Interpreter	-
Mediator	-
Proxy	-
Singleton	1
Strategy	-
TemplateMethod	-

Table 10.4: All patterns found in NotificationCenter

test of SPQR's efficiency. There is no easy way to estimate the equivalent lines of code, but the POML file for `std` was nearly 7MB. Compare this with the combined POML representation for NotificationCenter which was 267kB in size, even with substantial redundancy between individual translation units. Especially interesting is that POML was designed to contain relationship data, and, as such, it is almost certainly a better indicator of 'large' systems than a simplistic metric such as lines of code. The user code for this test was only thirty-one lines, yet it produced a POML file over 1600% larger than did the 1083 lines of NotificationCenter. It indicates that much more complexity is being imported. To be fair, the `std` namespace was culled from the POML representation of NotificationCenter, but this is an excellent example of ubiquitous reuse of assumed code and the intricacies that are encapsulated for the user.

10.2 Performance

One recurring concern with SPQR is that of scalability. While it has proven to be a robust and efficient system at small code sizes, larger code bases may uncover weaknesses in the approach, and become intractable in a practical sense. I will discuss in this section the results found so far, and various techniques for limiting this scalability risk. Table 10.7 shows the performance metrics for each test. I will extract subsections of this data to drive the ensuing discussion.

10.2.1 `gcctree2poml`

The `gcctree2poml` tool is admittedly a weak link in SPQR from a performance standpoint. Because it relies on `gcc`'s dump tree format, and because it requires a re-parsing and conceptual massaging of the data, it is much slower than it could be if it were tied directly into the internals of `gcc`. One goal of the SPQR implementation was that it would require no custom development tools for the end user, and integrate with the system already in use. Because of this desire, the current `gcc` and `gcctree2poml` pairing is reasonable, but, in retrospect, a significant reduction in the time for production of the POML representation could be achieved by direct access of the internal data structures of `gcc`. Looking at the relative execution times for `gcc` vs. `gcctree2poml` on the same source shows that the POML production takes approximately thirty times the time for the binary production by `gcc`. Add in that `gcc` is being slowed down by the emission of the dump trees in these tests, and it becomes obvious that this is not a tool for interactive use.

How much of that process can be stripped out by integrating the POML production process with the

	std_SPQR
AbstractInterface	38
Conglomeration	254
CreateObject	96
Delegate	6624
DelegatedConglomeration	244
DelegateInFamily	110
DelegateInLimitedFamily	1080
ExtendMethod	-
Inheritance	214
Recursion	241
Redirect	7042
RedirectedRecursion	-
RedirectInFamily	176
RedirectInLimitedFamily	1568
Retrieve	106
RevertMethod	10
FulfillMethod	56
Objectifier	-
ObjectRecursion	-
RetrieveShared	-
AdapterClass	-
AdapterObject	-
AdapterSuperclass	-
Bridge	-
ChainOfResponsibility	-
Command	-
Composite	-
Decorator	-
FactoryMethod	-
Interpreter	-
Mediator	-
Proxy	-
Singleton	-
Strategy	-
TemplateMethod	-

Table 10.5: All patterns found in C++ `std` namespace

	Gang of Four Training	Killer-Widget	Notification-Center	C++ `std` namespace
File Sizes	(avgs)		(combined)	
C++ (user code)				
kB	0.4	0.8	28.1	0.5
LOC	23.6	48	1083	31
gcc .tu				
kB	142.2	202.5	15462.2	14484
# of Nodes	1248	1768	141792	137637
POML				
kB	22.9	47	267	6816.4
classes	4.3	5	28	1542
objects	6.2	7	36	512
methods	34.1	46	307	8404
fields	3.3	4	56	2015
OTTER				
clauses	108	171	2227	30322

Table 10.6: File sizes

	Gang of Four Training	Killer-Widget	Notification-Center	C++ `std` namespace
Timings (wall sec) (avg of five runs)	Exploratory	Validation (Decorator)	Validation (Singleton)	Exploratory
gcc	0.19	0.23	6.90	7.91
gcctree2poml	4.99	6.89	233.46	257.44
spqr - setup				
poml2otter	0.14	0.52	0.46	34.38
otter inference	2.08	3.63	10.50	4009.20
extract inference	0.07	0.34	0.36	11.86
total	7.47	11.61	251.68	4055.44
spqr - search	0.35†	9.5	18.704	1929.9†

† Average over all patterns in exploratory search

Table 10.7: Performance data for major phases

	Gang of Four Training	Killer-Widget	Notification-Center	C++ `std` namespace
Timings (CPU sec) (avg of five runs)				
total	4.99	6.89	233.46	257.44
per knode	4.00	4.07	1.646	1.87
phases in sec				
parse	0.90	1.28	51.39	54.69
flatten	1.43	1.87	74.02	66.59
resolve	0.82	1.10	29.64	34.32
gather	1.43	2.00	73.77	64.64
emit	0.41	0.64	4.64	37.20
as % of total				
parse	18.11	18.59	22.01	21.25
flatten	28.59	27.20	31.71	25.87
resolve	16.36	15.93	12.70	13.33
gather	28.69	29.05	31.60	25.11
emit	8.25	9.23	1.99	14.45
per knode				
parse	0.72	0.76	0.36	0.39
flatten	1.14	1.11	0.52	0.48
resolve	0.65	0.65	0.21	0.25
gather	1.15	1.12	0.52	0.47
emit	0.33	0.34	0.03	0.27

Table 10.8: Timing for phases of `gcctree2poml`

parsing systems and internal representation can be estimated by looking at the various phases internal to `gcctree2poml`. I designed the tool to use the Visitor pattern extensively, and there are five main phases: parsing, flattening, gathering, resolving, and emission. Parsing reads in the `gcc` dump tree, and creates an internal object graph that is an exact mirror of the file structure. Flattening removes much of the graph structure by collapsing the various tree descriptors into lists, a data structure that Python is highly optimized for using. This effectively turns the graph into a representation closer to the original code, undoing much of the tree production in `gcc`. It also deletes a large amount of most graphs, as the support nodes are removed, saving memory and processing time for later phases. Resolving ensures that all cycles are appropriately traversed, and all the code is brought into conformance with the conceptual representation of ρ-calculus. Gathering then traverses this graph and extracts the appropriate ρ-calculus relationships, as well as detecting instances of **Retrieve**, **CreateObject**, and **AbstractInterface** and making these patterns explicit in the graph as new nodes. Finally, emission of the POML occurs.

Table 10.8 shows the timings for the various phases within `gcctree2poml`, reported by using the `-t` option. Assuming that the parsing, flattening and resolving phases are essentially just overhead for this

particular tool, it becomes obvious that integration with a parser or IDE could result in an approximate 65% reduction in time for this tool. This assumes that the remainder of the tool is left in the Python scripting language. Incorporating the algorithms into the compiler suite's native language should show an even larger increase in speed.

As stated before, however, replacement of `gcctree2poml` does not require the alteration or replacement of any other piece of SPQR. Further thought must be done on the tradeoffs of producing a dedicated tool against integration with existing workflows.

10.2.2 Otter

Otter's performance has been pleasantly surprising. It has handled tens of thousands of input clauses with ease, even when these resulted in an order of magnitude growth in the generated inferences. In addition to this robustness, Otter has demonstrated a speed sufficient for practical use. There may be solvers more suited to SPQR's needs in future research, but this was one area where the SPQR design has exceeded expectations.

I distinguish the *inference* phase of the toolset from the *search* phase by the intent. Inference is when the inter-reliance-operator judgments are produced, to cover the relational space as completely as possible. Once these relationships are in place, the search phase is when specific instances of design patterns are found in the relational space.

Inference

The core of the SPQR analysis is done during the inference phase of the Otter runs, as indicated in Section 9.2.4. As the ρ-calculus details become more refined, it is possible that this phase may become even more computationally intensive. The current implementation, however, offers adequate performance during this phase. The cumulative performance data is shown in Table 10.9.

One weakness in the current system is that if any code in a translation unit is altered, the entire translation unit must be reanalyzed from the beginning. It would be much more efficient if there were a way to track the inferences made by Otter, so that only the clauses affected by the code change could be regenerated. This may be possible by enhancing the tools that convert proofs to POML, and that extract the inferred clauses for search runs. Each would have to determine from the Otter output what clauses went into the production of a given inference. This information could be used to establish a dependency tree from the original clauses. The original clauses can be mapped back to the originating source code by the `source` and `file` attributes of the POML file. If the code changes, then the tree can

	Gang of Four Training	Killer-Widget	Notification-Center	C++ `std` namespace
OTTER Input Clauses	108	171	2227	30322
Timings (CPU sec) (avg of five runs)				
total	1.18	2.88	9.92	3617.19
per knode	0.93	1.63	0.07	26.28
per kOIC	3.36	16.84	4.45	119.29
phases in sec				
input	0.90	1.18	2.04	180.79
inference	0.28	1.70	7.88	3436.40
as % of total				
input	76.9	41.0	20.6	4.99
inference	23.1	59.0	79.4	95.01
per knode				
input	0.73	0.67	0.014	1.313
inference	0.23	0.96	0.056	26.49
per kOIC				
input	8.75	6.90	0.92	5.96
inference	2.5	9.94	3.54	113.33

Table 10.9: Timing information for OTTER inferences

be consulted to determine which clauses to strip from the inference clause set, and only the new clauses would be added to the `sos` list. This minimizes the regeneration.

In addition, the dependency tree can be consulted for identifying hot spots within the code. Clauses that have a large number of dependent clauses may be locations in the code that are central nodes of reliance. I expect that these would be of interest to refactoring analysis.

Search

For the three smaller tests from Section 10.1, the search times on a per-pattern basis were sufficiently small that they were manageable in the aggregate. The exploratory search on `std`, however, indicates that, for larger systems, an alternative algorithm may be more effective. Each pattern search is currently given a fresh run of OTTER, with all necessary subpattern results included. For a particular pattern with a deep conceptual hierarchy, this is fairly effective. The driving constraint behind this design decision was enabling SPQR to be used in distributed computing. I initially thought that being able to spawn off parallel searches on a grid cluster would provide a better performance than serial searches on one machine. In contrast with the trends in Table 10.9, the loading time of large files into OTTER is quite substantial in comparison to the actual computation time when performing pure searches on the result of an inferential pass. Given the range of test data size, the ratio for input time vs. search time has held

n	$O(n(C_1 + C_2))$	$C_1 + O(n(C_2))$ 10:1	$C_1 + O(n(C_2))$ 15:1
1	1	1	1
2	2	1.09	1.06
3	3	1.18	1.13
4	4	1.27	1.19
5	5	1.36	1.25
10	10	1.91	1.56
15	15	2.27	1.88
20	20	2.73	2.19
25	25	3.18	2.50
50	50	5.45	3.06

Table 10.10: Comparison of normalized time estimates for search algorithms

remarkably consistent across all tests run to date with a range of 9.7-14.2:1. This can be modeled as an algorithmic time for each search of $O(n(C_1 + C_2))$ where $C_1 : C_2$ is the ratio just given.

Because of this, I recommend that in future SPQR implementations using OTTER, two approaches be provided: the current distributable parallel search algorithm, and a monolithic search algorithm. The latter can be directed within OTTER by using the `weight_list` construct, to indicate to OTTER which search formula to select at each stage, and ensure proper walking of the pattern dependency tree. I expect that the performance savings for large input files will be significant in single compute node environments. This effectively pulls C_1 in the above time estimate out of the sum, and produces a run time estimate of $C_1 + O(n(C_2))$. With $C_1 : C_2$ of even just 10:1, this results in tremendous savings of run time as n gets larger, while not adversely affecting low n runs, as shown in Table 10.10.

10.2.3 Support Tools

The smaller tools that tie the former two main phases together are a small part of the run-time consideration of SPQR, but they are the critical juncture points for future research possibilities. In light of the above discussions, I briefly mention here how these tools may be modified to support SPQR enhancements, and their potential performance impacts.

`poml2otter`

As an XSLT, this tool is highly dependent on the XSLT engine chosen. For instance, the `4xslt` package, available from `http://www.4suite.org`, was originally used in SPQR, and while adequate, replacing it with the `xsltproc` engine from the Gnome project, found at `http://www.gnome.org`, resulted in an

order of magnitude speedup on average.

Results for the `xsltproc` engine are shown for the four tests, and can be compared to the POML file size to estimate the execution time for such transformations. I suspect a dedicated POML tool based on a stream-based XML parser would offer even better performance.

`extractinferences`

In Section 9.2.4, I mentioned that the output of OTTER during the inference phase was converted to OTTER input for the search runs. A small support script, `extractinterfaces`, is embedded in SPQR for this purpose. It is currently quite simple, and is $O(n)$ to the number of inferred rules, but, if the inference-dependency-tracking during the inference phase is implemented, this phase would need to parse not only the final inferences, but also their deduction paths as well.

`proof2poml`

Extracting the proofs from the OTTER output, identifying pattern instances, and producing the appropriate POML file is currently only a tiny fraction of the analysis time. As the search strategy is refined, however, this phase may prove to be more time intensive than it is now. Particularly, if the single-pass search system is implemented, then this phase would be responsible for partitioning out the proofs into the appropriate POML files. While I do not expect that this will result in a large increase in the analysis time relative to the rest of the toolkit, it will certainly be a point at which poor algorithmic decisions could cause problems. Given the linear nature of the search order in the proposed single-pass method, and given that OTTER emits its output file during processing, it can be shown that the ordering of pattern proofs within the output file will be sorted by search order. I, therefore, suggest a simple scanning of the file, following the current tool, but changing the output stream on detection of a new pattern name. This will keep the execution time linear on the size of the output file.

10.2.4 Performance bounding

For any given ATP or inference engine, its performance will be determined by the size of the input. The particular performance characteristics of the ATP are unique to the engine, but it would be useful to be able to make some estimates on the size of the input from the original source code. From that, the time for a search run may be properly estimated for the performance characteristics of the engine being used.

This becomes essentially a problem of bounding the number of ATP input rules generated from the code. In the case of OTTER the input follows the POML model very closely, and a 1:1 ratio can be

asserted. It is expected that, given that POML is a concrete representation of ρ-calculus, and that ρ-calculus is expected to be a simple mapping for most inference engines and logical systems, this ratio of POML elements to inference clauses will hold relatively steady. The mapping from the source language to POML will provide the information needed to determine the specific bounding formula for a given language. For C++, the derivation of this formula is straightforward. I will work from the inside out to produce the bounding formula, building up the summations along the way.

A method call is the first primitive, and has two main parts that must be addressed - the method invocation, and the parameter list. The latter is the easiest to deal with, as each parameter leads to the emission of a `poml:parameter` element, which, in turn, produces one OTTER input line. As a result, the cost for a parameter in a method call is fixed at 1, as in Equation 10.6.

$$P_i = 1 \tag{10.6}$$

The method call produces a $<_\mu$ relationship and therefore a `poml:calls` element in POML, for another singular OTTER clause. This is added to the summation of the cost for each parameter, as in Equation 10.7, assuming n parameters in the call.

$$C_i = \sum_{i=1\ldots n} P_i + 1 \tag{10.7}$$

Updates have two possible forms: update with a return value from a call, and update with a field. The former incurs the cost of the method call, while both convert to a single POML element and subsequent OTTER clause.

$$U_c = C_i + 1 \tag{10.8}$$

$$U_f = 1 \tag{10.9}$$

Each field declaration produces just one POML element, which leads to one OTTER input clause, to describe the type relationship. For instance, `PrinterQueue pq;` will produce a single OTTER rule of the form $pq : PrinterQueue$. This simple relationship can be expressed as in Equation 10.10, with U_i as above for any initialization that is present, such as in `PrinterQueue pq = getTheQueue();`. If no such default value is given in the declaration, then U_i for that statement is simply zero.

$$F_i = U_i + 1 \tag{10.10}$$

A result statement reduces to a single POML element, and a single OTTER clause.

$$R_i = 1 \tag{10.11}$$

A method will have a cost equal to the sum of its constituent statements: the total of the local field declarations, with optional initializers, the total update statements, and the total of the method calls. Added to this are POML elements for the parameters within the method declaration, and the method declaration and definition within POML. The parameters are defined as above for method calls, so the same cost can be reused here. The declaration and definition are each one POML element and resulting clause in OTTER.

$$M_i = \sum_{i=1..p} P_i + \sum_{i=1..f} F_i + \sum_{i=1..u} U_i + \sum_{i=1..c} C_i + \sum_{i=1..r} R_i + 2 \tag{10.12}$$

If, however, the method is abstract, the method-body statements above reduce to zero, and the total cost is the number of parameters in the method declaration, the declaration itself, and the indicator for the method being abstract. In POML this is the `poml:abstract` null element, while in OTTER it is an instance of **AbstractInterface**. In both cases, the cost is 1.

$$AM_i = \sum_{i=1..p} P_i + 2 \tag{10.13}$$

Namespaces will generate rules for each of their methods and fields. Since a namespace in C++ is an untyped entity, it can neither have abstract methods, nor can it be involved in inheritance relationships. The cost of a namespace is therefore just the sum of its parts, plus 1 for the definition of the namespace itself, and 1 for the type declaration of the namespace object.

$$N_i = \sum_{i=1..m} M_i + \sum_{i=1..f} F_i + 1 \tag{10.14}$$

As with namespaces, classes will pass forward their method and field costs. In addition, classes may contain abstract methods, inherited methods, and superclass methods which are accessed directly. Inherited methods, IM_i, are those that are brought forward from superclasses intact, and neither replicated in place in the subclass, nor overridden. Each of these is a simple POML statement, and generates only a single clause in the initial OTTER file. Superclass methods are those that are called directly from a superclass, bypassing the superclass' own implementation. Each of these, given as SM_i, likewise creates

just one POML element and one OTTER rule. The number of superclasses for the class is given as S_i, and each inheritance primitive is a single rule. The class declaration adds a single POML element, and finally, there will be three clauses generated for the associated class-object above and beyond the normal namespace cost, given as KO_i.

$$K_i = \sum_{i=1..m} M_i + \sum_{i=1..am} AM_i + \sum_{i_1..im} IM_i + \sum_{i_1..sm} SM_i + \sum_{i=1..f} F_i + KO_i + S_i + 2 \tag{10.15}$$

The total cost of the system is therefore simply the summation of the namespaces and classes.

$$T = \sum_{i=1..n} N_i + \sum_{i=1..k} K_i \tag{10.16}$$

With Equation 10.16, the exact number of clauses generated by a source code base can be computed, assuming a perfect mapping. The previous computations can be performed and compared against the actual POML files generated from the source code to find the overhead created by the production environment. What is more useful is to use this computation as an estimator based on average counts of methods per class, parameters per method signature, average method calls per method, and so on.

The estimator forms of the above cost computation formulae can be found in Equations 10.17 through 10.25. The barred form of each variable indicates the average number of that element in the current scope. These averages will hold across the code base, however, so an average number of parameters per method, $\overline{P}$, is also the average number of parameters in method calls. The variables with a hat are those that are fractions of a whole, such as the fraction of field declarations with initializers leading to updates is indicated by $\widehat{U}_I$. These equations can be used to estimate the ATP input size for C++ code bases.

Symbol	Average num of
$\overline{C}$	calls per method
$\overline{P}$	parameters per method
$\overline{U}_M$	update statements per method
$\overline{R}$	return statements per method
$\overline{F}_M$	fields per method
$\overline{F}_K$	fields per class
$\overline{F}_N$	fields per namespace
$\overline{S}$	superclasses per class
$\overline{N}$	namespaces in system
$\overline{P}$	classes in system
$\overline{AM}$	methods per class that are abstract
$\overline{IM}$	methods per class that are implicitly inherited
$\overline{SM}$	methods per class that are direct calls to a superclass

Symbol	Fraction of
$\widehat{U}_C$	updates with call sources
$\widehat{U}_F$	updates with field sources
$\widehat{U}_I$	field declarations with initializer

$$\widehat{U}_C + \widehat{U}_F = 1$$

Figure 10.7: Explanation of symbols used in bounding formulae 10.17 - 10.25

$$C = \overline{P}P + 1 \tag{10.17}$$

$$UC = C \tag{10.18}$$

$$U = \widehat{U}_C UC + \widehat{U}_F UF \tag{10.19}$$

$$F = \widehat{U}_I U + 1 \tag{10.20}$$

$$M = \overline{P}P + \overline{F}_M F + \overline{U}_M U + \overline{C}C + \overline{R}R + 2 \tag{10.21}$$

$$AM = \overline{P}P + 2 \tag{10.22}$$

$$N = \overline{M}_N M + \overline{F}_N F + 1 \tag{10.23}$$

$$K = \overline{M}_K M + \overline{AM}AM + \overline{IM}IM + \overline{SM}SM + \overline{F}_K F + KO + \overline{S}S + 2 \tag{10.24}$$

$$T = \overline{N}N + \overline{K}K \tag{10.25}$$

Equations 10.17 through 10.20, with the knowledge that $\widehat{U}_C + \widehat{U}_F = 1$, can be reduced to Equation 10.26.

$$F = \widehat{U}_I(\widehat{U}_C \overline{P}P + 1) + 1 \tag{10.26}$$

Substituting in the known constants for the C++ to POML mapping, a final estimation equation set can be defined as in Equations 10.27 through 10.32.

$$F = \widehat{U}_I(\widehat{U}_C\overline{P} + 1) + 1 \tag{10.27}$$

$$M = \overline{P} + \overline{F}_M F + \overline{U}_M(\widehat{U}_C\overline{P} + 1) + \overline{C}(\overline{P} + 1) + \overline{R} + 2 \tag{10.28}$$

$$AM = \overline{P} + 2 \tag{10.29}$$

$$N = \overline{M}_N M + \overline{F}_N F + 1 \tag{10.30}$$

$$K = \overline{M}_K M + \overline{AM}AM + \overline{IM} + \overline{SM} + \overline{F}_K F + \overline{S} + 5 \tag{10.31}$$

$$T = \overline{N}N + \overline{K}K \tag{10.32}$$

If a project, group, or corporation has available metrics data to use as input for the estimation, it can help guide their use of SPQR, and pinpoint which areas they may wish to target first. If such metrics do not exist, then POML provides a common format for quickly establishing such metrics. Once a system is described in POML, XSLTs can be used to extract most metrics available in current software engineering literature. The `POML2SimpleMetrics.xsl` file, included in the SPQR installation, is a small example of what can be done in a short amount of time. With that transform, any code that eventually is expressible in POML can be quantified for those metrics in a matter of seconds. As a group uses SPQR to analyze systems for patterns, they can be gathering useful metrics to be used in establishing baselines for analysis estimates.

10.3 Summary

In this chapter I outlined the four tests suites performed to validate SPQR, and the performance data derived from them. The tests demonstrated the utility of SPQR in training, validation of code, proved the isotopic approach, and showed an exploratory search on a larger codebase. From the resulting performance data, I was able to point out several opportunities for future tool enhancement, and provide a calculation for estimating the time for an analysis based on the structure of a system.

Chapter 11

Maintenance Metrics

I started this document with a discussion of maintenance, and this research with wanting to provide meaningful data to management that could guide economic decisions governing technical projects. With the technical theory and implementation for analysis in hand, it is, therefore, appropriate to resume the subject of management and maintenance.

It is even appropriate now to ask whether it is *possible* to alleviate the maintenance problem, and the lack of communication between management and engineers. I first turn to the technical management literature for clues as to how the business world views technical matters as they relate to project cost and, ultimately, success.

11.1 Software Project Risk Analysis

I researched the issue with Dr. Albert Segars of the Kenan-Flagler Business School at the University of North Carolina, by whom I was directed to a 1999 Georgia State University dissertation by Linda Wallace, now of Virginia Polytechnic Institute and State University. Wallace's doctoral dissertation study (Wallace, 1999) is a recent and comprehensive view of software project elements that predict project success or failure. This risk analysis is particularly interesting, since it does not assume ahead of time what the underlying predictors will be, but, instead takes a wide ranging approach that intends to discover the appropriate elements. In doing so, it follows the lead of other established work in the field (McComb and Smith, 1991).

Wallace determined six axes of risk analysis in her work, each consisting of six to eleven items selected by a statistically significant pool of software project practitioners and managers. These six are: User (having to do with the end users), Development Team (related to the development team

itself), Requirements (level of interaction between the development team and users and/or management), Organizational Environment (overall systemic corporate culture and management of the corporation or developing body as a whole), Project Management (management of this specific project), and Project Complexity (technical complexity of the project). Note that of these, only Complexity is directly tied to the technical aspects of software production. While reading this work, I fell into the same trap that many others in management and even engineering do, equating Complexity with the difficulty of maintenance. It was, therefore, distressing to see that Complexity had a low correlation with software project success, .2992 (Wallace, 1999, Figure 5.15), especially in light of the large amount of technical literature to the contrary.

I decided to review Wallace's data with criteria determined by software comprehension research, not management, and the results were decidedly different.

Item selection

The items selected for reevaluation are shown in Figure 11.1, and each is defensible from an engineering comprehension standpoint. The values shown are the correlations of the data elements with the final process outcome (Wallace, 1999, Table 5.22).

All four Team items relate to educating team members on the workings of the project. This is intimately and obviously tied to ease of comprehension of the system. The easier it is to comprehend, the easier it is to relate the appropriate information to the necessary developers. As presented in Chapter 1, training becomes a major component of the cost of maintenance.

Complex2, Complex10, and Complex11 again require training of developers on new technology. As before, if this technology is easily comprehended, the difficulty of this task is reduced.

Complex4 is an obvious choice given the necessity for higher level abstractions in improving comprehensibility.

All four Requirement related items were selected because they are indicative of a system in flux. The more requirements change, are incomplete, lead the developers to wrong conclusions, or are just plain wrong, the more critical it is that the software design be flexible and forgiving of such changes. This is a key item in the IEEE729 definition of maintenance (ANSI/IEEE, 1983), the reasons for which have been amply shown previously (von Mayrhauser and Vans, 1996b; von Mayrhauser and Vans, 1995; von Mayrhauser and Vans, 1996a; Banker et al., 1993).

Item	Description	Correlation
Team1	Inadequately trained development team members	.76
Team3	Inexperienced team members	.74
Team5	Frequent turnover within the team	.52
Team10	Team members lack specialized skills required by the project	.68
Complex2	Project involved the use of new technology	.77
Complex4	High level of technical complexity	.66
Complex10	Immature Technology	.66
Complex11	Project involves use of technology not used in prior projects	.77
Require2	Conflicting system requirements	.63
Require3	Continually changing system requirements	.73
Require7	Unclear system requirements	.88
Require8	Incorrect system requirements	.80

Figure 11.1: Items related to comprehension

Results of regrouping

With this new grouping in place, the methodology of Wallace was recreated in the Mplus statistics package (Muthen and Muthen, 2001), to provide a better comparison. Wallace's original analysis was performed in the LISREL (Hayduk, 1987) statistics package, and I wanted a reduction of variables in the comparison. An initial grouping of all comprehension related items as a single collection resulted in no convergence of the data, and no fit was found. Taking each of the above sections of items, however, and regrouping the original pools into two subgroups, resulted in a better fit than Wallace's approach.

For example, Team was split into Team and Training, Complexity was split into Technical Complexity and Organizational Complexity, and Requirements was split into Requirements and Flexibility. The new groups were created from the above items, as in Table 11.2.

If one compares the analysis of Complexity as a supergroup of Organizational and Technical Complexity groups to the original Complexity grouping, it is apparent that there is good justification for the new improved subgrouping based on the χ^2 values as reported by Mplus. This mirrors Wallace's approach, using this metric as one measure of 'goodness' of fit. Similar results are found by inspecting the original Team group against the new Team combined with Training, and also by comparing the original Requirements against the new Requirements and Flexibility as a paired grouping. Table 11.3 shows the χ^2 values reported by Wallace for the original groupings as computed by LISREL, my recreation of the same groupings in Mplus, and the new groupings described above. The new groupings show a better fit to the data than do Wallace's, when using the χ^2 metric.

Now that Training, Technical Complexity and Flexibility have been teased out of the original groups, a new second order grouping, Comprehension, can be created that encompasses these three subgroups. Again, following Wallace's methodology, Figure 11.4 shows the final correlations between the groups and

Wallace	Smith	Items
Team	Team	Team2
		Team4
		Team7
	Training	Team1
		Team3
		Team5
		Team10
Complexity	Organizational Complexity	Complex3
		Complex6
		Complex8
		Complex9
	Technical Complexity	Complex2
		Complex4
		Complex10
		Complex11
Requirements	Requirements	Req2
		Req3
		Req7
		Req8
	Flexibility	Req1
		Req6
		Req10
		Req11

Figure 11.2: Redistribution of items related to Comprehension

Group	Wallace (LISREL)	Wallace (Mplus)	Smith (Mplus)
Team	100.18	100.61	68.97
Complexity	124.76	124.91	76.29
Requirements	79.53	79.90	76.36

Figure 11.3: Comparison of χ^2 for re-analyzed groups

project risk and success. Comprehension now shows a 0.86 correlation, which is much more in line with the expected outcome from an engineering point of view. Technical Complexity is still quite low, but slightly higher than Wallace's original analysis. The important lesson is that *Comprehension* is highly critical when taken as a whole. It is of particular note that the other axes of factors did *not* appreciably drop in correlation with project risk. The χ^2 fit for Wallace's model in Mplus was 3332.07, a significant increase over the 2745.76 reported from her initial analysis using LISREL. This is an expected change between these two packages, however, according to Sharon Christ, Research Statistician at the University of North Carolina Odum Institute. My model had a χ^2 metric of 3050.24 in Mplus, indicating a slightly better fit than Wallace's model when analyzed by that package.

The above analysis prompted me to contact Dr. Wallace, and we have been discussing the results. She graciously performed an analysis of my re-interpretation using the AMOS statistics package. This yielded fit metrics that were extremely close to those reported in her original dissertation, as shown in Figure 11.5, further establishing this as a valid re-analysis as compared to her original results. She was kind enough to share, however, that subsequent re-analysis on her own has produced a new model that provides a much better fit to the data than either her earlier work or my re-analysis (Wallace et al., 2004). As with her original model, the technical aspects of comprehension are not explicit, and I look forward to applying the insights gained from this exercise to her new model, to see if the strong correlation between comprehensibility factors and project success can again be shown.

Given the variability in the statistics tools used, I do not believe it is appropriate to discuss which may be the *better* model, but the close correlation between the various metrics makes me confident in stating that the Comprehension-oriented model is a *valid* alternative.

Comprehensibility of a system is the core technical trait necessary for maintainability of that system, and should be considered a primary axis of measurement for addressing project risk, as Wallace's data indicates. Unfortunately, at this point in time little experience exists to measure comprehensibility from a direct analysis of the system. Instead, case studies and surveys of the development team are used to obtain indirect data. I believe that the techniques demonstrated in SPQR can produce the necessary data directly from the artifacts of engineering.

11.2 Maintainability Metrics

Given that the above analysis supports that providing comprehension cues to engineers is a critical component of reducing the most costly aspect of software engineering, the next problem becomes how

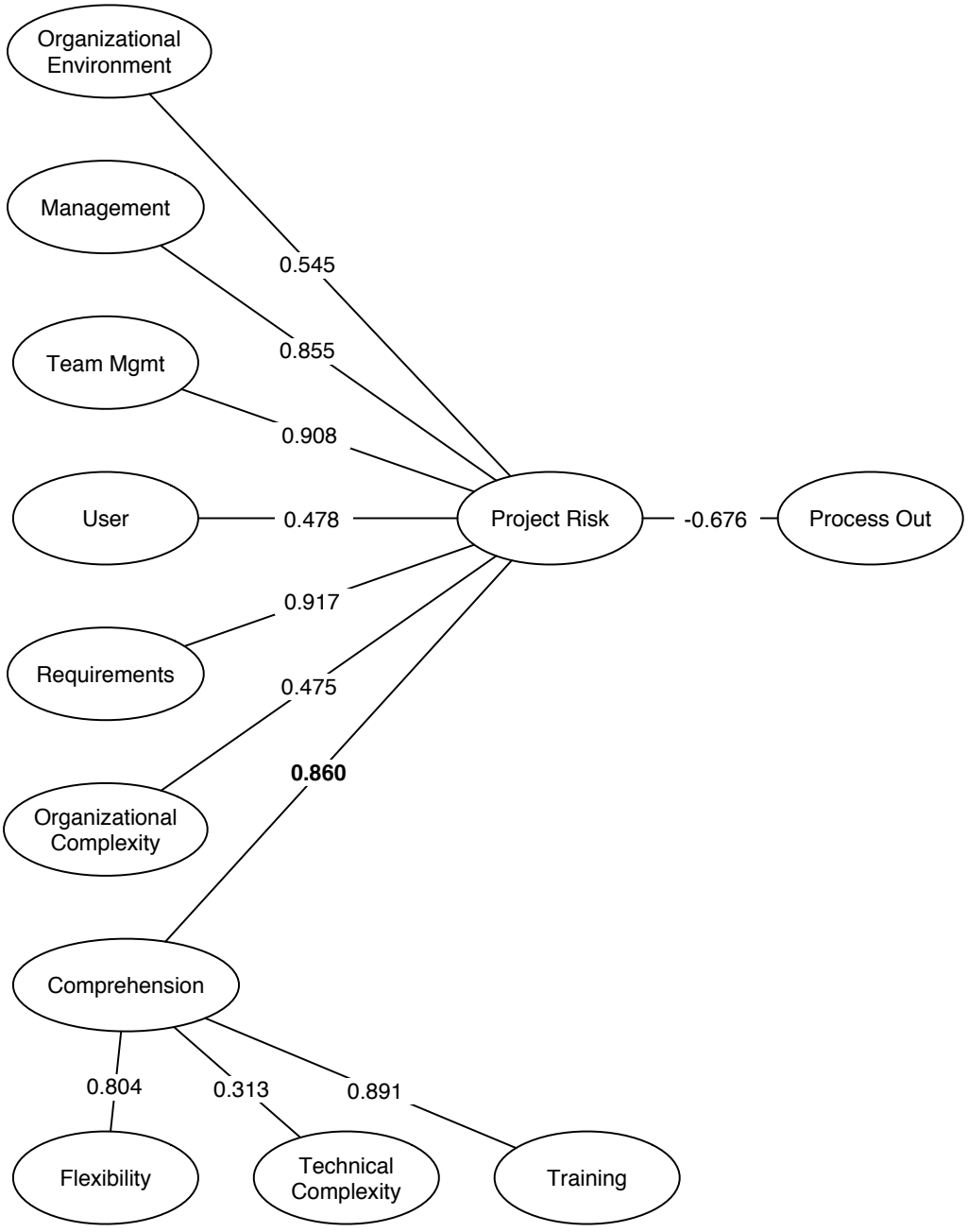


Figure 11.4: Reconfigured correlations

	CFI	GFI	AGFI	RMSEA	RMSR
Wallace (LISREL)	0.85	0.80	0.77	0.06	0.07
Smith (AMOS)	0.843	0.795	0.774	0.06	0.075

Figure 11.5: Comparison of fit metrics for final models

best to convey the information to management. As I stated in the introductory chapter, I believe that only numerical data will be appropriate, and that the fundamentals of metrics must be adhered to. I will present here proposed metrics for comprehensibility factors, and then illustrate how those can be used by management to estimate not only costs of maintenance, but also to estimate costs of maintenance savings after an architectural refactoring.

11.2.1 Basic metrics of pattern coverage

As established in Chapter 1, and further supported in this chapter, comprehension is the basis for maintainability of a system. Revisiting the four basic element of object-oriented programming, objects, methods, fields, and types, metrics can be defined for how each is involved in pattern use in the system.

Involvement

The simplest and most obvious metric is to state whether a fundamental element is used directly in a pattern instance. This can be determined by inspecting the pattern instances reported by SPQR. For instance, in the Killer Widget example from Section 10.1.2, a more complete UML diagram for the detected patterns can be generated as in Figure 11.6. I show only the Gang of Four level patterns, eliminating the lower patterns from concern at this point.

By inspection, the File class occurs directly in the Component role of the **Decorator** instance found. Similar observations can be made regarding FileFat, FilePile, and FilePileFixed. Furthermore, the op1 method of the same class also occurs directly in the **Decorator** instance, and ties these classes together.

If a class such as File has a primary role in a pattern, a point value can be given to it for each such fulfillment it is involved in. The sum of all points leads to the Direct Involvement measure, given in Equation 11.1. DI for the File class is therefore 2 in Figure 11.6. If the lower level patterns, such as the EDPs, were to be included in this analysis, the sum would of course be much higher. Due to this, the context of the analysis must be provided. A convenient shorthand is to annotate the summation with the level of analysis being performed, as in DI_{GOF}, to indicate that only Gang of Four and above patterns are being considered. If the analysis were to extend down to the EDP level, then DI_{EDP} would

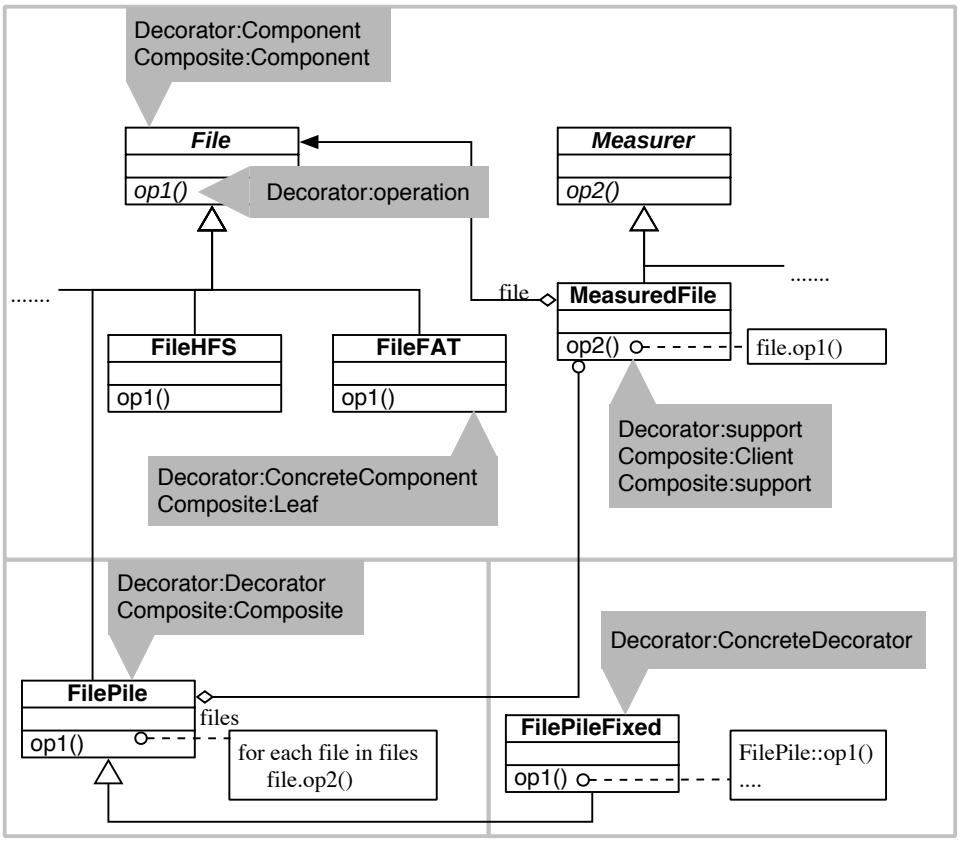


Figure 11.6: Recovered Gang of Four patterns in KillerWidget as UML

be appropriate. For the purposes of this discussion, DI will be assumed to be DI_{GOF}.

$$\mathrm{DI} = \sum \text{roles filled in all patterns in context} \tag{11.1}$$

Just as a class can be directly involved with a pattern role, so can methods and fields. They also receive a DI measure, computed in the same manner for classes. op1 in the above diagram would have a DI of 1.

DI gives a measure of how many patterns a programmatic element is involved in. A high DI can mean that a particular element is a conceptual hot spot, a singular point where two or more patterns intertwine in specific ways. These hot spots are generally fragile, as they can be critical architectural points that many behaviors rely on, and can be seen as prime candidates for refactoring.

Because the relative importance of an element within a pattern can differ, the Weighted Direct Involvement, shown in Equation 11.2, allows each pattern to provide a cue about the more important conceptual pieces of its definition. For instance, the **operation** role is central to **Decorator**. It is the

binding agent that brings the other elements together. Because of this, it may be given a higher weight than the other elements. Assume that **Decorator** provides a mapping function such that ConcreteComponent has a weight 0.8, operation has a weight of 1.3, and the remainder of the roles are normalized at 1.0. The WDI of File::op1 would then be 1.3, as would the WDI of FilePile::op1, FilePileFixed::op1, and FileFAT::op1.

$$\text{WDI} = \sum \text{roles filled in all patterns in context} \times \text{weight factor for pattern role} \tag{11.2}$$

The WDI allows patterns to adjust the relative conceptual importance of the roles needed to fulfill their purpose, providing important clues to the underlying interactions without making them explicit. As with DI, a context must be provided for WDI, and the same notation can be utilized.

While the above metrics can be used to measure explicit occurrences of programmatic elements within patterns, a more interesting metric is to measure the *implicit* occurrences. The MeasuredFile class, for instance, has a DI of 1, but it appears as a support role in two other patterns. A support role is one in which the object, method, field or type is involved in the transitivity of a reliance operator. To revisit the isotopic example from Section 5.9, MeasuredFile is required to fulfill the **RedirectInFamily** requirement of the **Decorator** pattern. MeasuredFile does not appear in either the left-hand side or right-hand side of the final reliance operator, but it was needed to derive the relationship. This is the definition of a support role: used in the inference of a requirement of the pattern, but not directly appearing in the final pattern instance. A Support Involvement metric can be measured by adding one point for each occurrence of the programmatic element in an inference leading to a pattern instance, as in Equation 11.3. In the above example, MeasuredFile has an SI of 2.

$$\text{SI} = \sum \text{instances in inferences leading to pattern requirements} \tag{11.3}$$

As might be expected, a Weighted Support Involvement can also be created, allowing patterns to establish the relative criticality of their requirements, and any inferences that may be involved. The weighting factors can be provided just as with the WDI metric.

$$\text{WSI} = \sum \text{instances in inferences leading to pattern requirements} \times \text{weight factor for pattern role} \tag{11.4}$$

Both the SI and WSI metrics require information that is currently not captured by SPQR. As I stated in Section 4.3.2, establishing a tracking associativity for the reliance operator transitivities would

	DI	WDI	SI	WSI
File	2	2	0	0
op1	1	1.3	0	0
FileFAT	2	1.8	0	0
op1	1	0.8	0	0
FilePile	2	2.4	0	0
op1	1	1	0	0
files	0	0	1	1
FilePileFixed	1	1	0	0
op1	1	1.3	0	0
MeasuredFile	1	0.8	2	2.7
op2	1	0.8	2	2.7
file	1	0.8	2	2.7

Figure 11.7: Pattern Involvement Metrics for Killer Widget

be useful, and this is where it would provide the biggest benefit. The tools surrounding Otter can be adjusted to provide this information as needed. The implicit support involvement metrics can provide cues to where hidden critical paths of comprehension reside.

An example set of pattern involvement metrics for the Killer Widget example, at the Gang of Four level, are show in Figure 11.7. Assume that the DI weighting matrix for **Decorator** is as above, and the SI weighting matrices are such that MeasuredFile::op2 is considered highly important to **Decorator** at 1.5, but not quite so much to **Composite**, at 1.2.

Coverage

Classes and objects can also be measured by how much of their contents are covered by pattern instances. In the above example, with no extraneous code, this will be a perfect fit of patterns to source. In most code, however, there will be code that is not involved in patterns in any way. As with the above metrics, context is important at all stages of measurement.

The DI and WDI metrics above measure the direct involvement of methods and fields of a class or object in pattern instances as role fulfillers. Such entities can be tagged as Direct. Similarly, the SI and WSI metrics involve entities that can be labeled Support. All other elements of a class or object will be tagged as Unrelated to pattern instances. Coverage metrics for methods, classes and fields can then be provided as follows.

Pattern Coverage Count is a raw count of the number of entities involved in pattern instances, weighted as shown in Equation 11.5. The elements that provide support should be downgraded slightly, in my opinion, as they contribute to the overall coverage, but do not have as explicit a role to play. For

that reason, a weighting factor < 1.0 is assumed. For clarity, I define it here as 0.7.

$$\text{PCC} = \sum \text{Direct elements} + \text{Support elements} \times 0.7 \tag{11.5}$$

The Relative Pattern Coverage (Methods) and Relative Pattern Coverage (Fields) are percentages of the number of methods and fields involved in the pattern coverages. Defined in Equations 11.6 and 11.7, they indicate how involved an entity is in the defined patterns. Classes and methods with low RPC values are expected to require more documentation and explanation than those with high values, as the very presence of patterns facilitates the documentation and comprehension process.

$$\text{RPCM} = \frac{\sum \text{Direct methods} + \text{Support methods} \times 0.7}{\#\text{ all methods}} \tag{11.6}$$

$$\text{RPCF} = \frac{\sum \text{Direct fields} + \text{Support fields} \times 0.7}{\#\text{ all fields}} \tag{11.7}$$

Because the Killer Widget example has no extraneous code, the pattern coverage metrics are of little utility, and I will omit the production of the metrics for it. Assume, however, a class A that has ten methods, three of which are found directly in pattern roles, two of which are used in support modes, and one which is used both directly and indirectly. In addition, A has five fields, three of which are in indirect support roles, and one which is explicit in a pattern role. The PCC for A would then be $5 + (6 \times 0.7) = 9.2$. The RPCM would be .61, and the RPCF would be .62. With both RPC* metrics above .5, this class shows a good relative coverage compared to its PCC. A class with poor coverage but a high PCC would indicate a class that would be a prime candidate for refactoring into smaller and more discrete classes. On the other hand, a class with a low PCC but high RPC* metrics would suggest a well-formed and task-specific class. The WDIs for the prominent factors leading to the RPC* values would then be inspected to determine if the elements involved were direct or support. Support elements have a higher probability of being extracted into a side-class during refactoring efforts.

11.2.2 Comprehensibility metrics

The above metrics are useful for determining pattern hot spots and the relative importance of classes and objects within a system with respect to patterns, but they are poorly designed for giving a sense of how well-designed a system is with respect to comprehensibility.

For that, it is useful to turn to an offbranch of Kolmogorov Complexity theory known as Minimum Description Length theory, or MDL (Grünwald, 2005). MDL formalizes the notion that if two

representations of a system are functionally equivalent, and one is 'smaller' than the other by some well-formed metric related to information density, then the smaller one is 'better', as it has less non-essential information and a smaller number of pieces of information to transmit and understand.

The usual application of MDL is to selection of a statistical model as related to the length of code required to express a particular set of data (Grünwald, 2005; Hansen and Yu, 2001). Lutz has outlined an approach to high-level design recovery using MDL, however, that attempts to capture the salient design characteristics of a system in a general sense (Lutz, 2002). Following a suggestion by Dr. Stotts, I have considered the application of MDL in a similar sense to the detection of design patterns, and more specifically, the detection of EDPs and the underlying concepts of ρ-calculus. It is my strong belief that the same principles can be used to create measures of comprehensibility of design of software.

ρ-calculus can be trivially extended to include the length of inference chains that led to a particular reliance. This *inference length* (IL) is a measure of how many components of the system took part in producing that final reliance. The larger the IL for a particular reliance, the more pieces of the system are required to be understood by the engineer to truly comprehend the reliance in question. The links in the IL chain are those elements that are tagged as Support for the purpose of pattern involvement and coverage metrics.

For instance, returning once again to the isotopic **RedirectInFamily** example in Section 5.9, the inference required to complete the EDP provides that clause with an IL of 1, as one indirect step is required to bridge the gap. The instance of **RedirectInFamily** would then, assuming all other necessary reliances were direct, have an IL of 1 as well. In general, the IL of a clause is the sum of the ILs of the clauses that led to its inference.

An interesting observation can be made at this point: design patterns describe, canonically, systems with an IL of 0. As the standard derived from community consensus, and with a goal of transmission of the minimum amount of information needed to distill the concepts at hand, this is not a surprising result. What is surprising, however, is that it indicates that SPQR analysis may be useful in helping to reduce pattern discussions to a form that can be considered canonical. A valid criticism is that canonical design patterns have an IL of 0 only when expressed in ρ-calculus, and I cannot deny it. I would point out, however, that it was established in Chapter 4 that ρ-calculus is based directly on the fundamental properties of object-oriented programming as expressed in ς-calculus. I find it difficult to conceive of a more primitive layer from which to analyze design patterns and the relationships of concepts they embody.

Even more interesting is the ability now to measure the comprehensibility of a system relative to the

canonical forms of the patterns. Not just the pattern coverage, but the relative *quality* of the pattern coverage can be computed, where quality is strictly a measure of distance from the normative form described in the patterns literature.

The principle is that pattern instances that are closer to the accepted form are most directly expressing their concepts in the code, and the underlying behaviors are made explicit and clear. With an eye towards comprehension, and, therefore, maintenance, this is the critical factor of measure. Any code that is involved in and necessary to the pattern, but which adds to the IL can be considered obfuscatory. It may be that this code is essential for other reasons, such as performance, but *from a design point of view aimed at improving comprehension*, it is sub-optimal.

From the above, a Comprehensibility Difference can be created, measuring the overall fit of the architecture to what could be considered an 'optimal' form if every instance of patterns, and therefore abstractions, adhered to the simplest form possible. This distance from normative form can be used to as a measure of architectural quality *with respect to comprehensibility*. I cannot stress this point enough, that this is not a measure of overall quality of design any more than is the number of seconds a system takes to execute. It is *one* measure for architects and engineers to consider when designing systems. When it comes to maintainability, however, I believe it will prove to be crucial.

$$\text{CD} = \sum \text{IL of constituent clauses} \tag{11.8}$$

The CD can be weighted as to the relative importance of the constituent clauses, just as the coverage and involvement clauses can. Each pattern, class, or other construct can define a mapping for the relative criticality of the clauses, and provide further clues as to the distance from optimal for a conceptual instance, computed as the Weighted Comprehensibility Difference in Equation 11.9.

$$\text{WCD} = \sum \text{IL of constituent clauses} \times \text{weight factor for clause} \tag{11.9}$$

From this, a Comprehensibility Quotient can be derived, where the CD is divided by the number of pattern instances in the context under consideration. This gives a relative measure of the average distance for each pattern, and helps normalize the metric for large systems. Equation 11.10 defines the CQ, where S is any package, library, or system being analyzed.

$$\text{CQ} = \frac{WCD(S)}{\#\text{ of patterns in } S} \tag{11.10}$$

For KillerWidget, the overall system design has a CD_{GOF} of 1, from the isotopic **Decorator** pattern. This single link can be easily found by decomposing the patterns list to find the core isotopic **RedirectInFamily**. Assuming an equal weighting, the WCD_{GOF} will also be 1. The CQ_{GOF} is then 0.5, a value I would consider reasonable for any real system, as it indicates that half of the patterns involved in the system have, on average, one indirect reliance.

Currently, engineers are left without answers when management asks the pertinent questions regarding any large systemic change: "How much will it cost, and how much will we save later?" Given this lack of pertinent data from the engineering staff, it is no wonder that many technically critical tasks for the long-term survivability of large projects are never approved. Armed with the above metrics, however, an engineering staff can provide relative measures of comprehensibility of design for a system for which a refactoring has been proposed. "Better" is no longer just an instinctual feeling to the engineer, but a measurable quality of the code.

11.2.3 Metrics for management

The above set of metrics allows engineers to begin to relay significantly complex systems analysis information to non-technical management by means of simple numbers related to training, comprehensibility, and expected maintainability. Management can then correlate these numbers with historical data on maintenance efforts, known costs for past projects, and other data that should be on hand for any institution that conforms to a process standard such as the Capability Maturity Model Integration for Software (CMMI-SW) Level 3 or above (Team, 2002). In this way, management can do the cost analysis that they are trained in, and be less concerned with the technical details. This separation of concerns between technical and economic analysis is important, and the comprehensibility metrics are the key commonality.

In particular, illuminating results should be found by correlating with training costs. For instance, if 10% of a system's code contributes to known and documented patterns, the other 90% will be necessarily manually documented by the engineering teams, and that knowledge will need to be taught to new engineers as time goes on. If, however, 50% of the system contributes to known patterns, then there is presumably a much smaller training task to be performed, resulting in expected increased efficiency when hiring new talent or communicating engineering tasks between teams.

While the production of comprehensibility metrics is important moving forward, the ability to analyze past project states from a source revision control system should provide the correlation between system design and project costs that will be needed for accurate forecasting of future expenditures. This means

that these metrics will not only be useful in the future, but can also be useful after automated runs on past code.

There is, of course, the danger that these metrics will be seen to be absolute. They are not, and should not be taken as rigid rules. A system may very well be better served be eschewing known design patterns, particularly as the system requirements become more esoteric and unique. In addition, restricting engineering professionals to an established set of known solutions precludes innovation into unknown solutions.

With these metrics in mind, an engineer can communicate to management a measure of relative maintainability, and pinpoint the portions of the code that are critical to such a task. Now, instead of being without an answer, an engineer can give precisely the answer needed to gain authorization for project tasks that previously had only a technical, and therefore opaque to management, justification.

Chapter 12

Future Research

This work offers many future research possibilities to the software engineering community. They range across the full spectrum from formal analysis through human comprehension assistance, much as the ρ-calculus and Elemental Design Patterns do.

12.1 EDPs as language design hints

It is expected that some will see the EDPs as truly primitive, but I would point out that the development of programming languages has been a reflection of directly supporting features, concepts, and idioms that practitioners of the previous generation languages found to be useful. Cohesion and coupling analysis of procedural systems gave rise to many object-oriented concepts, and each common object-oriented language today has features that make concrete one or more EDPs. EDPs can therefore be seen as a path for incremental additions to future languages, providing a clue to which features programmers will find useful based precisely on what concepts they currently use, but must construct from simpler forms.

12.1.1 Delegation

A recent and highly touted example of such a language construct is the *delegate* feature found in C# (Microsoft Corporation, 2002). This is an explicit support for delegating calls directly as a language feature. It is in many ways equivalent to the decades old Smalltalk and Objective-C's selectors, but has a more definite syntax which restricts its functionality, but enhances ease of use. It is, as one would expect, an example of the **Delegation** EDP realized as a specific language construct, and demonstrates how the EDPs may help guide future language designers. Patterns are explicitly those solutions that have been found to be useful, common, and necessary in many cases, and are therefore a natural set of

behaviors and structures for which languages to provide support.

12.1.2 ExtendMethod

Most languages have some support for this EDP, through the use of either static dispatch, as in C++, or an explicit keyword, such as Java and Smalltalk's **super**. Others, such as BETA (Madsen et al., 1993), offer an alternative approach, deferring portions of their implementation to their children through the *inner* construct. Explicitly stating 'extension' as a characteristic of a method, as with Java's concept of *extends* for inheritance, however, seems to be absent. This could prove to be useful to the implementers of a future generation of code analysis tools and compilers.

12.1.3 Retrieve, RetrieveShared

The C++ Standards Working Group is currently considering the inclusion of the `shared_ptr` and `weak_ptr` templates from the Boost libraries (Dimov et al., 2003). These classes offer semantic tags with support for sharing an object among several pointers. When `shared_ptr` is in one of its canonical intended uses, to accept a newly created object, it mirrors the **RetrieveShared** EDP closely. `weak_ptr` performs the same purpose, but without enforced reference counting and resource destruction of `shared_ptr`. The analogue for singly ownership pointers to objects, the proposed `unique_ptr` (Abrahams et al., 2005), similarly compares to the **RetrieveNew** concept.

12.1.4 Collection

Many languages now have direct support for collections such as lists, sets, and dictionaries. Some like Python have support via built-in operators, while others such as C++ offer libraries that are easily detectable through code analysis. In such cases, a language to POML tool can take advantage of these language or library specific features to add additional information in the form of pattern instances in the fact pool related to the **Collection** pattern.

12.1.5 AbstractInterface

The **AbstractInterface** EDP is, admittedly, one of the simplest in the collection. Every object-oriented language supports this in some form, whether it is an explicit programmer created construct, such as C++'s pure virtual methods, or an implicit dynamic behavior such as Smalltalk's exception throwing for an unimplemented method. It should be noted though that the above are either composite constructs,

such as *virtual foo() = 0;* in C++, or a non-construct runtime behavior, as in Smalltalk, and as such are learned through interaction with the relationships between language features. In each of the cases, the functionality is not directly obvious in the language description, nor is it necessarily obvious to the student learning object-oriented programming, and more importantly, object-oriented design. Future languages may benefit from a more explicit construct.

12.2 Educational uses of EDPs

I believe the EDPs can provide a path for educators to guide students to learning OO design from first principles, demonstrating best practices for even the smallest of problems. Note that the core EDPs require only the concepts of classes, objects, methods (and method invocation), and data fields. Everything else is built off of these most basic OO constructs, which map directly to the core of UML class diagrams. The new student needs only to understand these extremely basic ideas to begin using the EDPs as a well formed approach to learning the larger and more complex design patterns. As an added benefit, the student will be exposed to concepts that may not be directly obvious in the language in which they are currently working. These concepts are language independent, however, and should be transportable throughout the nascent engineer's career.

This transmission of best practices is one of the core motivations behind design patterns, but even the simplest of the usual canon requires some non-trivial amount of design knowledge to be truly useful to the implementer. By reducing the scope of the design pattern being studied, one can reduce the background necessary by the reader, and therefore make the reduced pattern more accessible to a wider audience, increasing the distribution of the information. This parallels the suggestions put forth by Goldberg (Goldberg, 1995).

12.3 Malignant patterns

One of the exciting aspects of this research is that it provides the foundation for finding not just established design patterns, but also any combination of relationships between language entities and programming concepts one might choose to define. One such application is looking for common *poor* solutions to common problems, where one can identify common mistakes made by programmers when attempting to follow design pattern guidelines. Also, one can describe situations where common solutions are found to exist, but known to be poor fits for many applications. These are both similar to what is

often termed 'anti-patterns', but that phrasing has become so diluted by misuse as to have lost much of its original precision. Instead, I call these 'malignant patterns'. They do indeed solve the problem at hand, but are likely to cause more problems in the long run, particularly during maintenance, than they solved in the first place. SPQR is as capable of finding these as it is the more common design patterns, since they are still only code constructs and relationships.

12.3.1 'Bad smells' as refactoring opportunity discovery

Fowler (Fowler, 1999) uses the term 'bad smells' to describe many such situations... the code looks okay from many perspectives, but an experienced practitioner feels there is just something not quite right about it. These are assumed to be cues for opportunities for refactoring, which is usually the case. Usually these 'bad smells' are quantifiable as relationships between concepts - precisely the type of constructs that SPQR is designed to extract. It is entirely possible to provide an engineering team with information not only on the conformance of a system with the design as expressed as design patterns, but also to provide them with indicators that there may be refactoring opportunities that they had not considered.

12.3.2 Semi-automated support for the refactoring process

Refactoring is not likely to ever be, in my opinion, a fully automatable process. At some point, the human engineer must make decisions about the architecture in question and guide the transformation of code from one design to another. Several key steps, however, may benefit from SPQR. The isotope example in Section 5.9 indicates that it may be possible to support verification of Fowler's refactoring transforms through use of the ρ-calculus, as well as various other approaches currently in use (Fowler, 1999; Opdyke and Johnson, 1993; Moore, 1996). Ó Cinnéide's minitransformations likewise could be formally verified and applied to not only existing patterns, but perhaps to code that is not yet considered pattern ready, as key relationships are deduced from a formal analysis (Ó Cinnéide, 2001; Ó Cinnéide and Nixon, 1998). Furthermore, I believe the fragments-based systems such as LePuS can now be integrated back into the larger domain of denotational semantics. It is my belief that the current research in refactoring, minipatterns, and automated support for both will provide the necessary formalisms if they are recast in the language of ρ-calculus.

12.4 Aspect Oriented Programming

Related to fragments in many ways, Aspect Oriented Programming (AOP) provides a language support framework for working with concepts that may have low locality (Elrad et al., 2001; Pace and Campo, 2001). SPQR can be used in this context as well, due to the flexibility of the reliance operators. As aspects are added or deleted from the codepath, SPQR can quickly and easily take into account new reliances while ensuring conformance with the original pattern intents. AOP is, at its core, an establishment of an **Observer** design pattern injected into a language's runtime with high-level support within the language itself. Since the **Observer** design pattern is an established design pattern in the SPQR lexicon, this is directly supportable by SPQR. As AOP matures, the combination of SPQR and dedicated AOP environments will prove to be very powerful.

12.5 Architecture 'Optimization'

SPQR is designed specifically to find instances of patterns that are not immediately recognizable as adhering to the usual structure, yet which fulfill the same conceptual needs. There is a comprehension distance between these two endpoints, which lead to the questions: Why are these more difficult to find, and why are design patterns listed in a minimalist form?

As discussed in Section 11.2.2, I believe that MDL theory holds a clue to how to describe architectures as relative to their conceptual optimal format as defined by the design patterns literature. Furthermore, I believe that it may lead to approaches to provide assistance to optimize the architecture in well-formed ways. Given that:

- SPQR can find, and was indeed designed specifically for the task of finding, pattern instances in systems no matter the IL of the components;
- a lower IL will be 'better', from the above argument;
- each component adding to the IL is known;
- and that we have, in the design pattern definition, a perfect reference IL of 0;

a methodology becomes conceivable that starts with a known conceptual system, the source code, and creates a new set of chains that preserve the relationships, a *new architecture*, that globally minimizes the IL into an MDL for the entire system.

Now that a beginning system is represented, and an optimal representation is in hand, the question becomes one of transformation. As I state above, I believe that automated assistance for refactoring from a conceptual perspective, as well as a behavioral basis, is now possible. I believe the relevant information can be found in refactoring planning research, specifically in Fowler and Cinnide's approaches on refactoring transformations, among others (Fowler, 1999; Moore, 1996; Ó Cinnéide, 2001).

It is not unreasonable to foresee a tool that can optimize an existing architecture with respect to maintenance, and do so automatically, providing a refactoring plan for consideration. I make no assertions or claims about the perfection of such an approach, since it would undoubtedly impact other concerns and the axis of measurement for any system, but it is exciting to have such a tantalizing possibility within reach.

12.6 SPQR Refinements - Tools

SPQR is, of course, a work in progress. It was designed to be as flexible as possible to take advantage of new programming environments, target languages, and solver engines, but there are specific improvements that I have already identified as important refinements.

12.6.1 Level of Detail control

SPQR needs to provide the user with finer control over the level of detail of patterns that are searched for and expressed. A large system will be comprised of an immense number of EDPs, for example, yet the practitioner is not likely to be interested in them as a primary comprehension cue. By default, reporting of EDPs should be suppressed, but available as an option for research purposes and more precise interaction with the system.

One approach for fulfilling this need is to allow the user to select which sections of the pattern dependency graph they wish to have reported. The pattern dependency graph is a directed acyclic graph that indicates which design patterns are composited from others. The EDPs form the base of the graph, the lowest layer, while patterns such as Objectifier lie at the next level up the hierarchy, and so on. The example patterns from Chapter 6 are shown in Figure 12.1 as an example graph. An engineer may select a depth level slice ("patterns no lower than X, no higher than Y"), in addition to the current selection of particular patterns ("only patterns in the Decorator subgraph"), or a combination ("only patterns in the Decorator subgraph, lower than Decorator, but higher than the EDPs").

Alternately, the inferences found by SPQR may create technically true, but unwanted, positives. One

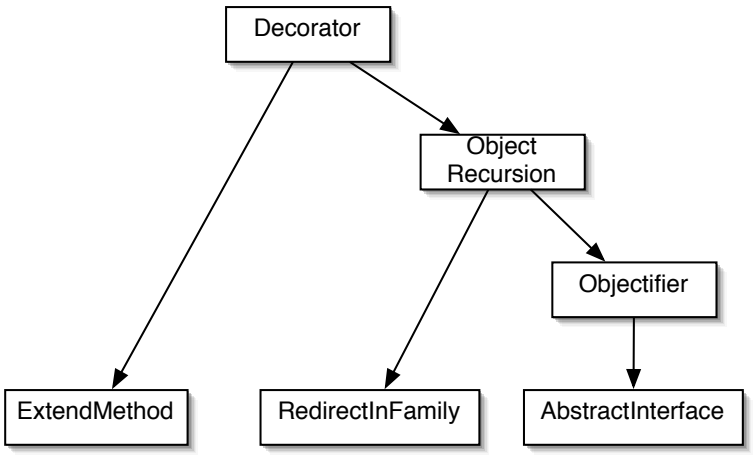


Figure 12.1: Example Pattern Dependency Graph

such situation where this can occur is when elements from extremely far-flung aspects of the system are tied together by extremely long reliance operator chains. Such relationships are likely to be tenuous at best, and not representative of what may be happening in the system. To prevent such hits on valid but artificial pattern instances, the reliance chain limiting technique of Section 9.2.4 can be utilized to put a hard limit on the depth to which OTTER will create relationships.

12.6.2 Improved similarity extraction

While the lexicographic equality approach currently used in SPQR has proven to be quite robust, it could be improved by incorporating semantic analysis techniques (Maletic and Valluri, 1999; Maletic and Marcus, 2000). While similarity is not interested in the precise intent of a single element, it may be that relatively simple semantic analysis can provide a more robust relative intent extraction. For example, `getInstance` and `getTheInstance` would be recognized by a human engineer as semantically equivalent, but a strict lexicographic comparison would fail to find this. Incorporating even fairly simple rules could increase the robustness of similarity detection.

12.6.3 Alternate Automated Theorem Provers

OTTER is a highly generalized theorem prover, and while more than adequate for SPQR, may not be optimal. As ATPs and my familiarity with the field improve, other engines may provide a better fit with the needs of SPQR. One move that can be done immediately is to use the TPTP intermediate language (Sutcliffe and Suttner, 1998) for expressing the ATP form of the fact base. It would mean yet another step in the analysis process, but would open up a wide array of ATPs to investigate as SPQR options.

The fundamental fact is that SPQR allows, today, for any inference engine to be incorporated as part of the toolchain.

12.6.4 Alternate environments

Currently gcc 3.x is the target environment, but several others are obvious choices as well, including .NET, Xcode, Eclipse, and more direct language sources such as the Squeak Smalltalk runtime, and the Java Virtual Machine.

IDE Integration

The former would be treated much like gcc, but with the possibility of providing the results of SPQR unified within the environment offering some intriguing possibilities. For instance, Eclipse's extensible UI gives one the opportunity to integrate SPQR directly into the IDE, with a direct menu item or button click triggering SPQR, feeding it a simplified (and pre-selected) fact base on the source code under its control, and displaying the results in a textual or graphical format as another View on the system. Xcode and .NET have similar capabilities for plugin tools, and SPQR is well suited for this task.

Runtime environment snapshots

Interacting directly with the language runtimes leads to a wider array of languages that SPQR can handle. While SPQR is a static analysis engine and would seem to be unusable for analysis of dynamically bound and/or loaded languages, it should be noted that at any given moment in time the typing environment of a dynamic runtime environment is a static fact base. That set of facts about the system may change over time, but, if we select the point at which we choose to initiate the analysis, such as post-loading of all expected classes, we can analyze a snapshot of the environment just as if it were described in source code. In fact, SPQR could take a series of snapshots over time and view the changes that a design architecture makes during its lifetime, a fascinating and tantalizing research possibility. Sample-based profiling of a system's performance is now an established technique, I suggest the same for a system's architectural design.

12.6.5 Alternate data sources

SPQR was designed to be language, framework, platform, and OS independent, and it meets this requirement. It is unimportant what language, library, framework, or system is to be analyzed, as long as it can be expressed as an abstract description of the relationships between code entities. ρ-calculus, as

the formal basis for these relationships, ensures that all object-oriented languages that are expressible in ς-calculus can be so expressed. POML, as part of SPQR, allows for a unified approach to describing source code relationships, independent of the source language, the operating environment, or the application domain. This is a critical piece to SPQR, by providing a single definition for object-oriented design elements, and through which SPQR gains an unprecedented level of flexibility and applicability. I currently have a *gcc*-based translator for producing POML from C++, but as shown in the previous section, any environment is possible, and here I claim that any ς-calculus conformant language is possible.

Further, design documents such as UML specifications can be analyzed by SPQR, as long as the model design can be converted to POML. The XML Metadata Interchange language (XMI) (Group, 2005) is a common interchange format for UML tools, such as Rational ROSE (Quatrani, 2002; Kruchten, 2003), and offers a good target for conversion to POML through an XSLT.

12.6.6 Distributed analysis

SPQR is inherently distributable in the manner in which it analyzes a system. Assume that there is a set of patterns, P, that we wish to search for in a codebase, C. We transform C into a set of ρ-calculus facts, as per normal operation of SPQR, let's call it $\overline{C}$. Break P into n subsets, $P_1 ... P_n$. Send a search subset, P_i, along with $\overline{C}$, to another thread, process or machine capable of running SPQR and complete the process there. All n sets of search results are then compiled into a final report.

12.6.7 Expanding pattern libraries

Of course, SPQR will continue to incorporate new design patterns as they are defined in the literature, and new combinatorials may come to light as research continues. I expect that the libraries will also start to include site, application, and domain-specific patterns as SPQR is used in real world practice and useful constructs are identified through its use. As described in Section 9.5 and demonstrated in Section 10.1.1, SPQR facilitates its own training by performing the exact task for which it was designed - finding patterns.

This is an important advancement of the state of the art in code analysis - while the Gang of Four design patterns reference was the initial inspiration, I have produced a tool and methodology that go far beyond that. Instead, I can, with the same system, incorporate any architecturally well formed design pattern currently known or yet to be discovered.

In addition, while SPQR was designed to discover instances of design patterns in source code, it has, in fact, solved a much more fundamental problem, finding generalized relationships within source code. As such, I foresee SPQR working on a much broader base of possible constructs to find within source code than just design patterns.

12.7 SPQR Refinements - Theory

The SPQR toolset will continue to evolve as outlined above, but the more theoretical aspects offer some research opportunities as well. SPQR is currently a static analysis engine only - a highly flexible analysis engine, but purely static. SPQR will grow to incorporate more dynamic analysis features, preferably without requiring actual run-time analysis. One design goal for SPQR is to have it require only the source code for full analysis, and it would be desirable to maintain that goal for reasons of practicality as a toolset.

12.7.1 Temporal Logic

Temporal logic would allow SPQR to distinguish between the case where rule 1 occurs before rule 2 as distinct from when rule 2 occurs before rule 1, as in the two method calls shown in Figure 12.2, both of which currently derive to the equivalent non-temporal ruleset in Figure 12.3. Currently the two situations are seen as equivalent, and many dynamic run-time situations are beyond SPQR's analysis. Temporal logics would provide for refinement of the search space, confirming or excluding candidate patterns from the results. This would of course require a new theorem prover, and it is now that the inclusion of TPTP begins to bear fruit, as it would allow for simple switching of ATPs as necessary.

```
method( obj )                          method( obj )
{                                      {
        obj.method1();                         obj.method2();
        obj.method2();                         obj.method1();
}                                      }
```

Figure 12.2: Two distinct methods

12.7.2 Model Checkers

Related to temporal logic is the opportunity to incorporate model checking techniques (Clarke et al., 2000; Clarke et al., 1986) into SPQR. It is possible that the levels of abstraction produced by SPQR can help tame the state explosion problem that model checkers generally have to deal with (McMillan,

$$method <_{\mu} method1$$
$$method <_{\mu} method2$$

Figure 12.3: Equivalent non-temporal rules for both examples

1992). Simultaneously, it is expected that the model checking research can offer proven methods for working with temporal logic relationships effectively. A proven software model checker toolkit such as HOL (Agerholm, 1991) could be used to supplement the work of an ATP such as OTTER.

12.7.3 ς-calculus interpretation

Finally, the last piece of dynamicism, true run-time analysis, is in fact theoretically possible within SPQR without needing to run the binary of the target system. *Sigma* is a ς-calculus interpreter created by Santiago M. Pericas-Geertsen (Pericas-Geertsen, 2001), and is an open-source research project at Boston University. I suspect strongly that the SPQR analysis produced to date, combined with the temporal logic extensions, would reduce the amount of code that would actually need to be run for final verification of a pattern candidate to a very small set. Since ρ-calculus is a strict superset of ς-calculus, and since SPQR can encode the source code as ρ-calculus, small sections of the original code can be targeted for translation and interpretation. *Sigma* shows that this is not only possible, but practical. The results of this interpretation can be directly analyzed within SPQR for the run-time behaviors we wish to confirm. In this manner SPQR gains much greater analysis capabilities. Prior to SPQR, however, *Sigma*'s practicality has been limited to examples of theory, as it is only able to provide reasonable performance on extremely small sets of code. By using SPQR to definitively pare down the code base to small sample selections of source code, SPQR brings added value to *Sigma*, raising it from a purely theoretical tool to one of practical worth. This is simply an example implementation, however, and I am by no means tied to using *Sigma* as the interpreter engine.

12.8 Comprehension of code

I revisit the original motivation for this research, reduction of the time and effort required for an engineer to comprehend a system's architecture well enough to guide the maintenance and modification thereof. I believe that the approach outlined in the document, along with the full catalog of EDPs and ρ-calculus, creates a formal basis for some very powerful source code analysis tools such as *Choices* (Sefika et al.,

1996), or KT (Brown, 2000), that operate on a higher level of abstraction than just "class, object and method interactions" (Sefika et al., 1996). Discovery of patterns in an architecture should be become much more possible than it is today, and I expect that the discovery of *unintended* pattern uses should prove enlightening to engineers. In addition, the flexibility inherent in the ρ-calculus will provide some interesting possibilities for the identification of new variations of existing patterns.

12.8.1 Experimental support

A potential test of SPQR in the assistance of comprehension would be to take a fairly large OO system that is designed with patterns in mind, and run it through SPQR both as-is, and also after being run through a code obfuscator, thereby demonstrating the identical results. The experiment would then involve measuring engineer comprehension across four groups. Compare the work required to understand four variants of the same code: original code, original code + SPQR output, obfuscated code, and obfuscated + SPQR output.

12.8.2 Training and communication

SPQR will become a vital part of training new engineers on existing codebases, and EDPs are poised to become the lingua franca for describing design patterns and other architectural constructs. Using the two in tandem provides a powerful tool for corporate and academic education. The EDPs are too trivial in detail to be practical on their own without automated support, and SPQR is not effective without the common semantics of the EDPs. The combination of SPQR and the EDPs creates a unique training tool and communication platform.

12.9 Analysis of procedural code

Given that SPQR can provide a path from OO source code to the discovery, documentation, and comprehension of pattern use within the code, then it is theoretically possible to produce the same end result from procedural code using slice and interface analysis.

Slice analysis (Ott and Thuss, 1989; Weiser, 1984) is one method to help in refactoring procedural code to OO code by finding common data to function bindings, and suggesting which groupings would be natural candidates for class descriptions. These pseudo-, or degenerate, objects are not true objects in the practical sense, but have many of the same features especially when viewed from a formal ς-calculus

point of view. Multi-agent calculus (Grossman et al., 2000) is a way of considering them more closely as object and class analogues.

So, it comes down to this: can these degenerate classes and objects be properly represented in ς-calculus such that SPQR can be applied? I believe so. Will it be easy? Definitely not. However, others are working on refining this segment of the analysis and their work is highly promising (Ott, 1992; Ott and Thuss, 1993). This provides an interesting tie-in with the previously described opportunities in refactoring.

The reason that I believe SPQR can succeed in providing a working conceptual model of the code, is that SPQR does not, in its current incarnation, care what the actual tags used to describe programming elements are, only that the similarity principle applies correctly. Whether a method is named `processWithFourierTransform` or `f78t2kdgs8`, SPQR will give the same results. The former is obviously more useful to the human attempting to learn the system, but in either case, SPQR will provide the same conceptual cues. Whether the system architecture is created meticulously by hand, or by machine with random tags, SPQR will still give the same extracted patterns and an initial architecture description.

Now, given the above path from procedural to patterns, I believe it can be taken several more steps, drilling down all the way to a binary representation. Consider that moving from a binary to assembly code is a more or less solved problem, a purely mechanical transformation (Partsch and Steinbrueggen, 1983; Charlton et al., 1988). Further consider that conversion of assembler to procedural code is likewise a *fairly* solved problem (Cifuentes and Gough, 1995; Emmerik and Waddington, 2004). Aliasing of data is still an issue, but the core problems are well understood. The problem with such reverse engineered code is that it is, in most cases, completely incomprehensible to human engineers; the main semantic tags used to comprehend the system, the names of variables and functions, are missing. This limits the usefulness of this technique to small pieces of code that can be understood as a whole under manual analysis.

Using the above approach to transform procedural code into pseudo-object-oriented code, it then prepares it for analysis by SPQR. This is where SQPR's reliance on *relative* relationships between programming elements becomes a strong benefit. The analysis to identify pattern use is *completely independent* of the nomenclature used in the source code. Therefore, discovery of pattern use is just as capable on highly obfuscated code as it is on well named code.

Now, assuming the above binary to patterns path remains viable when researched, several *extremely* interesting possibilities come to light:

- Reverse engineeering of legacy binaries

  All too often source code to existing libraries or applications is lost. The above analysis would allow some degree of comprehension of the larger architecture of the system. Combine this nascent architecture with the engineers' knowledge of the domain, and the question of what a particular pattern construct was used *for* should be answerable. Again, this is not a silver bullet, but an aid to comprehension.

- Comparative design

  Library designers designing for reuse of their library make assumptions about how the library will be used. By extracting the architecture of a fully compiled system that is a client of the library, the designer can easily see how the library was *actually used*. If the use conflicts with the intended use, or the assumptions of the library designer, the pattern of use can give the library designer clues as to how to better design the library to fit the needs of the client.

- Comprehensible genetic programming

  Genetic algorithms are extremely powerful ways of optimally solving particular problems, but the resultant solutions suffer from a couple of serious flaws. The source code that is created is generally incomprehensible to humans, largely due to little or no naming mnemonics to assist, which means the solutions are rarely if ever reusable. Perhaps more important, any possible *insights* as to the more general lessons learned are lost. The full binary through patterns analysis path could help alleviate this. An interesting side-project presents itself here as well - genetic manipulation of patterns themselves, using elemental patterns as the 'DNA', allowing discovery of interesting variations.

12.10 Pattern Mining

The fact that SPQR is fully automated allows it to be used for brute-force problems that are currently beyond practical consideration. Once such application is mining for previously unidentified patterns amongst systems. Unidentified in this sense means previously undescribed in the patterns literature, not just undiscovered within a specific system.

Begin with the set of EDPs, E, and the ρ-calculus rules for combining them. Mechanically generate all possible combinations, call this set $E' = E \times \rho$. Use this as the input search catalog for SPQR, and search on a codebase C. The subset $\widehat{E'}$ of E' that was found in C is retained. Generate a new

search catalog $E'' = \widehat{E'} \times \rho$, and repeat the process on C. Continue generating such search sets and retain the found elements of those sets. When the found set has been reduced to one member, the union of all found sets ($\widehat{E'} \cup \widehat{E''} \cup \widehat{E'''} \cup ...$) comprises every combinatorial possibility of programmatic elements within the system. This will be a massive set. Remove those elements which are known, described patterns, call this set P, again by using SPQR to search the set. The remaining elements are all constructs that are previously unknown to or undescribed in the patterns community. Call this results set $R = (\widehat{E'} \cup \widehat{E''} \cup \widehat{E'''} \cup ...) - P$.

If this process is performed on a large number of systems, then the result sets from each can be cross-referenced. The intersection of these sets, $(R_1 \cap R_2 \cap R_3 \cap ...)$ will be a set of constructs who appear in several systems, yet are undescribed, since the known patterns were eliminated from consideration. This is now a candidate set for manual inspection by an engineer, as a place to start identifying new architectural patterns.

The intermediate sets generated by this approach will be immense... but since it is fully automatable, and since SPQR is inherently distributable (section 12.6.6), a researcher can merely initiate and walk away until results are compiled. What size $\cap R_i$ might be is completely unknown.

12.11 Genetics

More than one researcher or engineer has commented that the EDPs are 'akin to the DNA of programming'. Informal talks with members of the genetics research community have led to the belief that this may not be that far off, and that there is a unique opportunity for cross-collaboration. While a gross simplification, there is some credence to viewing the elements of object-oriented programming as the four DNA building blocks, the EDPs as the basic codons, ρ-calculus as defining the semantics of interaction much like RNA does, and so on. One researcher in genetics stated that SPQR seems to implement concepts of isotopic transitivities that are similar to current areas of genetic research in isomorphic proteins. I think it would be an interesting area of research to see what possibilities exist to exchange ideas with the genetics research community. They have more experience with extremely large data sets and heuristic approaches, while SPQR may prove to be an exceptional proving ground for the basic concepts.

Appendix A

Δ_ρ Fragment

This is the full definition of the rho fragment (Δ_ρ), including expanded notation where necessary to reduce the assumptions made in each definition. The sub-fragments are concerned with proper definition of selectables within an object ($\Delta_{\rho\in}$), selector separation semantics ($\Delta_{\rho(\bullet)}$), similarity ($\Delta_{\rho\sim}$), and finally the four reliance operator forms and their direct definitions and transitivities ($\Delta_{\rho<_\mu}, \Delta_{\rho<_\phi}, \Delta_{\rho<_\sigma}, \Delta_{\rho<_\kappa}$).

$\Delta_{\rho\in}$

(Methods Of)
$$\frac{E \vdash \mathcal{O},\ \mathcal{O} \equiv [l_i \Leftarrow b_i{}^{i\in 1\ldots n}, l_j{}^{j\in n+1\ldots m}]}{\mathbf{meth}(\mathcal{O}) = \{l_i{}^{i\in 1\ldots n}\}}$$

(Fields Of)
$$\frac{E \vdash \mathcal{O},\ \mathcal{O} \equiv [l_i \Leftarrow b_i{}^{i\in 1\ldots n}, l_j{}^{j\in n+1\ldots m}]}{\mathbf{field}(\mathcal{O}) = \{l_i{}^{i\in n+1\ldots m}\}}$$

(Definitions Of)
$$\frac{E \vdash \mathcal{O},\ \mathcal{O} \equiv [l_i{}^{i\in 1\ldots n}]}{\mathbf{def}(\mathcal{O}) = \{l_i{}^{i\in 1\ldots\ n}\}}$$

(Context)
$$\frac{E \vdash \mathcal{O},\ \mathcal{O} \equiv [l_i = b_i{}^{i\in 1..n},\ b_j = \{\ldots, \mathcal{S}, \ldots\}]}{\|\mathcal{S}\|_{\mathcal{O}.l_j}}$$

$\Delta_{\rho(\bullet)}$ Where $s \in \mathbf{def}(\mathcal{O})$

(Leftdot) $(\mathcal{O}.s\bullet \equiv \mathcal{O}$

(Dotright) $\bullet\mathcal{O}.s) \equiv s$

(Leftdot Raw) $(\mathcal{O}\bullet \equiv \{\}$

(Dotright Raw) $\bullet\mathcal{O}) \equiv \mathcal{O}$

(Sel Identity) $(\mathcal{X} \bullet . \bullet \mathcal{X}) \equiv \mathcal{X}$

(Sel Global) $\{\}.\mathcal{O} \equiv \mathcal{O}$

$\Delta_{\rho\sim}$

(Leftdot Sim)
$$\frac{\mathcal{X} < \mathcal{Y},\ (\mathcal{X}\bullet \sim (\mathcal{Y}\bullet}{\mathcal{X} <^{+.\circ} \mathcal{Y}}$$

(Dotright Sim)
$$\frac{\mathcal{X} < \mathcal{Y},\ \bullet\mathcal{X}) \sim \bullet\mathcal{Y})}{\mathcal{X} <^{\circ.+} \mathcal{Y}}$$

(Leftdot Dissim)
$$\frac{\mathcal{X} < \mathcal{Y},\ (\mathcal{X}\bullet \nsim (\mathcal{Y}\bullet}{\mathcal{X} <^{-.\circ} \mathcal{Y}}$$

(Dotright Dissim)
$$\frac{\mathcal{X} < \mathcal{Y},\ \bullet\mathcal{X}) \nsim \bullet\mathcal{Y})}{\mathcal{X} <^{\circ.-} \mathcal{Y}}$$

(Leftdot Gen)
$$\frac{\mathcal{X} <^{\pm\ \pm}_{\circ.\circ} \mathcal{Y}}{\mathcal{X} <^{\circ.\pm}_{\circ} \mathcal{Y}}$$

(Dotright Gen)
$$\frac{\mathcal{X} <^{\pm\ \pm}_{\circ.\circ} \mathcal{Y}}{\mathcal{X} <^{\pm}_{\circ.\circ} \mathcal{Y}}$$

$\Delta_{\rho<_\mu}$ Where $f \in \mathbf{field}(\mathcal{O}),\ m \in \mathbf{meth}(\mathcal{O}),\ g \in \mathbf{field}(\mathcal{P}),\ n \in \mathbf{meth}(\mathcal{P})$

(Mu Calls)
$$\frac{E \vdash \mathcal{O}, \mathcal{P},\ \|\mathcal{P}.n()\|_{\mathcal{O}.m}}{E \vdash \mathcal{O}.m <_\mu \mathcal{P}.n}$$

(Mu Update)
$$\frac{E \vdash \mathcal{O}, \mathcal{O}', \mathcal{P},\ \|\mathcal{O}'.f \Leftarrow \mathcal{P}.n\|_{\mathcal{O}.m}}{E \vdash \mathcal{O}.m <_\mu \mathcal{P}.n}$$

(Mu Phi/Kappa)
$$\frac{E \vdash \mathcal{O}, \mathcal{O}', \mathcal{P}, \mathcal{P}',\ \mathcal{O}.m <_\phi \mathcal{P}.g,\ \|\mathcal{P}.g <_\kappa \mathcal{P}'.g'\|_{\mathcal{O}'.m'}}{E \vdash \mathcal{O}.m <_\mu \mathcal{O}'.m'}$$

(Mu Phi/Sigma1)
$$\frac{E \vdash \mathcal{O}, \mathcal{O}', \mathcal{P}, \mathcal{P}',\ \mathcal{O}.m <_\phi \mathcal{P}.g,\ \|\mathcal{P}.g <_\sigma \mathcal{P}'.n'\|_{\mathcal{O}'.m'}}{E \vdash \mathcal{O}.m <_\mu \mathcal{P}'.n'}$$

(Mu Phi/Sigma2)
$$\frac{E \vdash \mathcal{O}, \mathcal{O}', \mathcal{P}, \mathcal{P}',\ \mathcal{O}.m <_\phi \mathcal{P}.g,\ \|\mathcal{P}.g <_\sigma \mathcal{P}'.n'\|_{\mathcal{O}'.m'}}{E \vdash \mathcal{O}.m <_\mu \mathcal{O}'.m'}$$

$\Delta_{\rho<_\phi}$ Where $f \in \mathbf{field}(\mathcal{O}),\ m \in \mathbf{meth}(\mathcal{O}),\ g \in \mathbf{field}(\mathcal{P}),\ n \in \mathbf{meth}(\mathcal{P})$

(Phi Return)
$$\frac{E \vdash \mathcal{O},\ m \in \mathbf{meth}(\mathcal{O}),\ \mathcal{O}.m \rightsquigarrow \mathcal{P}.f}{E \vdash \mathcal{O}.m <_\phi \mathcal{P}.f}$$

(Phi Param)
$$\frac{E \vdash \mathcal{O},\ \|\mathcal{P}.n(\mathcal{P}'.g')\|_{\mathcal{O}.m}}{E \vdash \mathcal{O}.m <_\phi \mathcal{P}'.g'}$$

(Phi Update1)
$$\frac{E \vdash \mathcal{O}, \mathcal{X}, \mathcal{Y},\ \|\mathcal{X} \Leftarrow \mathcal{Y}\|_{\mathcal{O}.m}}{E \vdash \mathcal{O}.m <_\phi \mathcal{Y}}$$

(Phi Update2)
$$\frac{E \vdash \mathcal{O}, \mathcal{X}, \mathcal{Y},\ \|\mathcal{X} \Leftarrow \mathcal{Y}\|_{\mathcal{O}.m}}{E \vdash \mathcal{O}.m <_\phi \mathcal{X}}$$

(Phi Kappa1)
$$\frac{E \vdash \mathcal{O}, \mathcal{X}, \mathcal{Y},\ \|\mathcal{X} <_\kappa \mathcal{Y}\|_{\mathcal{O}.m}}{E \vdash \mathcal{O}.m <_\phi \mathcal{X}}$$

(Phi Kappa2)
$$\frac{E \vdash \mathcal{O}, \mathcal{X}, \mathcal{Y},\ \|\mathcal{X} <_\kappa \mathcal{Y}\|_{\mathcal{O}.m}}{E \vdash \mathcal{O}.m <_\phi \mathcal{Y}}$$

(Phi Sigma)
$$\frac{E \vdash \mathcal{O}, \mathcal{X}, \mathcal{Y},\ \|\mathcal{X} <_\sigma \mathcal{Y}\|_{\mathcal{O}.m}}{E \vdash \mathcal{O}.m <_\phi \mathcal{X}}$$

(Phi Mu/Phi)
$$\frac{E \vdash \mathcal{O}, \mathcal{X}, \mathcal{Y},\ \mathcal{O}.m <_\mu \mathcal{P}.n,\ \mathcal{P}.n <_\phi \mathcal{P}'.n'}{E \vdash \mathcal{O}.m <_\phi \mathcal{P}'.n'}$$

(Phi Phi/Kappa)
$$\frac{E \vdash \mathcal{O}, \mathcal{X}, \mathcal{Y},\ \mathcal{O}.m <_\phi \mathcal{P}.g,\ \|\mathcal{P}.g <_\kappa \mathcal{P}'.g'\|_{\mathcal{O}.m}}{E \vdash \mathcal{O}.m <_\phi \mathcal{P}'.g'}$$

$\Delta_{\rho<_\sigma}$ Where $f \in \mathbf{field}(\mathcal{O}),\ m \in \mathbf{meth}(\mathcal{O}),\ g \in \mathbf{field}(\mathcal{P}),\ n \in \mathbf{meth}(\mathcal{P})$

(Sigma Update)
$$\frac{E \vdash \mathcal{O}, \mathcal{O}', \mathcal{P},\ \|\mathcal{O}'.f' \Leftarrow \mathcal{P}.n()\|_{\mathcal{O}.m}}{E \vdash \|\mathcal{O}'.f' <_\sigma \mathcal{P}.n\|_{\mathcal{O}.m}}$$

(Sigma Sigma/Mu)
$$\frac{E \vdash \mathcal{O}, \mathcal{O}', \mathcal{P},\ \|\mathcal{O}'.f' <_\sigma \mathcal{P}.n\|_{\mathcal{O}.m},\ \mathcal{P}.n <_\mu \mathcal{P}'.n'}{E \vdash \|\mathcal{O}'.f' <_\sigma \mathcal{P}'.n'\|_{\mathcal{O}.m}}$$

(Sigma Call-by-Name)
$$\frac{E \vdash \mathcal{O}, \mathcal{O}', \mathcal{P},\ \|\mathcal{P}.n(\mathcal{O}'.f')\|_{\mathcal{O}.m},\ \mathcal{P}.n\{\!\{\mathcal{P}.n.g \overset{n}{\leftarrow} \mathcal{O}'.f'\}\!\},\ \|\mathcal{P}.n.g \Leftarrow \mathcal{X}\|_{\mathcal{P}.n}}{E \vdash \|\mathcal{O}'.f' <_\sigma \mathcal{P}.n\|_{\mathcal{O}.m}}$$

(Sigma Kappa/Sigma)
$$\frac{E \vdash \mathcal{O}, \mathcal{O}', \mathcal{P},\ \|\mathcal{O}'.f' <_\kappa \mathcal{P}.g\|_{\mathcal{O}.m},\ \|\mathcal{P}.g <_\sigma \mathcal{P}'.n'\|_{\mathcal{P}''.n''}}{E \vdash \|\mathcal{O}'.f' <_\sigma \mathcal{P}'.n'\|_{\mathcal{O}.m}}$$

$\Delta_{\rho<_\kappa}$ Where $f \in \mathbf{field}(\mathcal{O}),\ m \in \mathbf{meth}(\mathcal{O}),\ g \in \mathbf{field}(\mathcal{P}),\ n \in \mathbf{meth}(\mathcal{P})$

(Kappa Update)
$$\frac{E \vdash \mathcal{O}, \mathcal{O}', \mathcal{P},\ \|\mathcal{O}'.f' \Leftarrow \mathcal{P}.g\|_{\mathcal{O}.m}}{E \vdash \|\mathcal{O}'.f' <_\kappa \mathcal{P}.g\|_{\mathcal{O}.m}}$$

(Kappa Return)
$$\frac{E \vdash \mathcal{O}, \mathcal{O}', \mathcal{P}, \mathcal{P}'\ \|\mathcal{O}'.f' \Leftarrow \mathcal{P}.n()\|_{\mathcal{O}.m},\ \mathcal{P}.n \rightsquigarrow \mathcal{P}'.g'}{E \vdash \|\mathcal{O}'.f' <_\kappa \mathcal{P}'.g'\|_{\mathcal{O}.m}}$$

(Kappa Input Parameter1)
$$\frac{E \vdash \mathcal{O}, \mathcal{P},\ \mathcal{O}.m\{\!\{f \overset{n}{\leftarrow} \mathcal{P}.g\}\!\}}{E \vdash \|\mathcal{O}.m.f <_\kappa \mathcal{P}.g\|_{\mathcal{O}.m}}$$

(Kappa Input Parameter2)
$$\frac{E \vdash \mathcal{O}, \mathcal{O}', \mathcal{P},\ \mathcal{O}.m\{\!\{f \leftarrow \mathcal{P}.g\}\!\}}{E \vdash \|\mathcal{O}.m.f <_\kappa \mathcal{P}.g\|_{\mathcal{O}.m}}$$

(Kappa Trans)
$$\frac{E \vdash \mathcal{O}, \mathcal{P}, \mathcal{P}',\ \|\mathcal{O}'.f' <_\kappa \mathcal{P}.g\|_{\mathcal{O}.m},\ \|\mathcal{P}.g <_\kappa \mathcal{P}'.g'\|_{\mathcal{P}''.n''}}{E \vdash \|\mathcal{O}'.f' <_\kappa \mathcal{P}'.g'\|_{\mathcal{O}.m}}$$

(Kappa Sigma/Phi)
$$\frac{E \vdash \mathcal{O}, \mathcal{P}, \mathcal{P}',\ \|\mathcal{O}'.f' <_\sigma \mathcal{P}.n\|_{\mathcal{O}.m},\ \mathcal{P}.n <_\phi \mathcal{P}'.g'}{E \vdash \|\mathcal{O}'.f' <_\kappa \mathcal{P}'.g'\|_{\mathcal{O}.m}}$$

Appendix B

EDP Catalog

Presented here are the full writeups for the Elemental Design Patterns as defined in Chapter 5. These follow the form defined by Gamma et al (Gamma et al., 1995), with the addition of the formal ρ-calculus definitions for each.

There are sixteen Elemental Design Patterns in this initial catalog. They are organized under three main categories: Object Elements, Type Relation, and Method Invocation. I expect additional EDPs to be added to this collection and the body of work as whole to be refined over time, as with any design pattern catalog.

Object Elements are those elemental patterns that deal with the creation and definition of objects: CreateObject and Retrieve. CreateObject describes when, how, and why we instantiate objects, what makes them special over procedural systems, and why they are not merely syntactic sugar. Retrieve outlines how and why one uses objects as data fields inside an enclosing object.

Type Relation has two simple patterns. Inheritance, providing a discussion of the primary method by which typing information (*and* method body definitions!) are reused effectively in object-oriented systems. AbstractInterface offers a solution for needing to defer implementations to type definitions that may be created at a later date.

The last grouping, Method Invocation, contains the final twelve patterns in this document. I created these after an inspection of the most common method-call patterns in the Gang of Four text, which led to an expanded view of orthogonal invocation issues. Two axes of similarity and one of various typing relationships led to the twelve patterns described here. A more complete treatment of this discovery process and their derivation can be found in Section 5.8.

These patterns, while small and precise, are important exactly because of their ubiquity in object-oriented programming, and their possible formalization. Every programmer uses these patterns on a daily basis, usually without conscious effort. Because the very purpose of patterns is to bring into conscious awareness that which is subconsciously and reflexively seen as useful, I believe this research is the obvious next step in patterns research. Other patterns have been produced that provide insights

into programming concepts in general, but these are unique to object-oriented systems, and, I believe, are a necessary, if not sufficient, set of design patterns for object-oriented programming from which all other design patterns can be built and composed. I follow the basic format used in the Gang of Four text in order to create a common framework which fosters discussion of these issues and how they relate to the larger design patterns in use today.

CreateObject — Object Creation

Also Known As

Instantiation

Intent

To ensure that newly-allocated data structures conform to a set of assertions and pre-conditions before they are operated on by the rest of the system, and that they can only be operated on in pre-defined ways.

Motivation

An object is a single indivisible unit comprised of data and applicable methods that are conceptually related. We wish to ensure that this unit is in a particular coherent and well-defined state before we attempt to operate on the object, and we wish to ensure that only well-defined operations can be performed on the object. In procedural languages, we can emulate an object rather well using records (C structs) and groupings of functions (libraries, perhaps with namespace encapsulation). A practitioner cannot, however, ensure either of the conditions of initial coherence or restricted operations.

For example, they cannot guarantee that, at the time of allocation of the record, that the record's contents conform to *any* specific assertion they may choose to make. They can make certain that particular classes of static assertions will hold "All records will have their third entry be set to '5'"),

```
typedef struct {
        int a;
        float b;
        short c = 5;
} myStruct;
```

but others are beyond their grasp, such as "Every third record created on a Tuesday will have the first entry set to '1', otherwise '0'.".

We can get around this problem in imperative languages by creating an initializer function that is to be called on all new allocated data records before use.

```
void initializeMyStruct( myStruct* ms ) {
        static int modulo3 = 0;
        if (modulo3 == 2) {
                modulo3 = 0;
                if (checkDay('tue')) {
                        ms->a = 1;
                }
        } else {
                modulo3++;
                ms->a = 0;
        }
}
```

This will be an effective solution, but not an enforceable one. Enforcement relies on policy, documentation, and engineer discipline, none of which have proven to be ultimately accurate or reliable. An engineer is, therefore, back to the original problem of not being able to guarantee that any given assertion holds true on the newly allocated data. A malicious, careless, or lazy programmer could allocate the structure, and then fail to call the proper initialization procedure, leading to potentially catastrophic consequences.

Object-based and class-based systems provide an alternative. When an object is allocated by a runtime, it is initialized in a well-formed way that is dependent on the language and environment. All object-oriented environments provide some analogous mechanism as a fundamental part of their implementation. This mechanism is the hook at which the implementor can create a function (usually called the initializer or constructor) that performs the appropriate setup on the object. In this way *any* specific assertion can be imposed on the data before it is available for use by the rest of the system.

More formally, we can say that by using the **CreateObject** pattern, for some assertion A, and some object $\mathcal{O}$, $A(\mathcal{O})$ is always true immediately following the creation of $\mathcal{O}$. This holds for any criteria checking function A. Without the **CreateObject** pattern, we can only state that either $A(\mathcal{O})$, or $D(\mathcal{O})$, where D is the default state of allocated storage in our system. Obviously the former is preferred for reducing errors.

There is no way for a user of the object to bypass this mechanism; it is enforced by the language and runtime environment. The hypothetical malicious, careless, or lazy programmer is thwarted, and a possible error is avoided. Since this type of error is generally extremely difficult to track down and identify, avoidance is preferred.

Similarly, we could emulate an object by the record and library approach of an imperative language and document that only certain operations from a pre-defined library may be used on

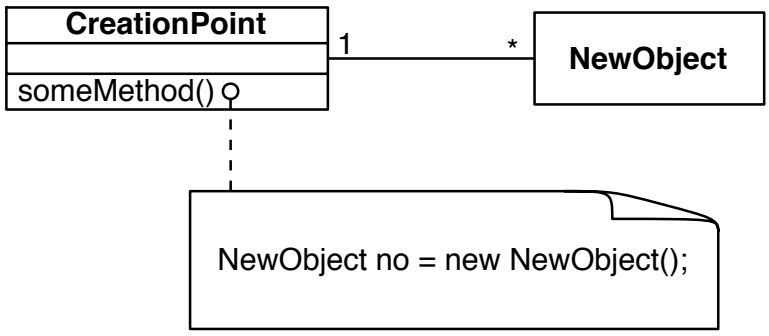
that data structure, but there is no way to enforce such a policy. An object has a pre-defined set of methods that correspond to the functionality of that object, and only they may interact directly with the embedded data structure. In this way objects are more than just mere syntactic sugar or convenience, and provide a strong policy-enforcement mechanism at the language level.

Applicability

Use **CreateObject** when:

- You wish to provide a data representation and enforce that only certain operations can be performed on any particular instance.
- You wish to provide allocated instances of a data representation and ensure that some set of pre-conditions are met before use.

Structure



Participants

CreationPoint

The object that requests the creation of a new object of type NewObject

NewObject

The object type to be created

Collaborations

An instance of CreationPoint requests that a new instance of object type NewObject be created. The exact mechanism for this will vary from language to language, but it generally consists of making the request 'of' the object type itself. Naturally, the request is actually given to the runtime environment, with the object type as an argument, but the syntax of most languages creates the appearance that the request is made directly to the object type. This is, as an astute

reader will notice, a self-referential definition - you must have an object to be the CreationPoint before making a new object. The initial object that kicks off this whole chain comes into play in the run-time mechanism of the language being used, either implicitly (such as in dynamic languages such as SmallTalk) or explicitly (such as for hybrid class-based languages such as C++ that requires a top-level procedural `main`, which can be considered a member of a global object.)

Consequences

Most object-oriented languages do not allow for the creation of data structures using any other method, but do not require the definition of a developer-supplied initialization routine. A default initialization routine is generally supplied that performs a minimum of setup.

Some languages, while object-oriented, allow the creation of non-object data structures. C++ and Objective-C are two examples, both derived from the imperative C language. Python, Perl 6 and other languages have similar historical reasons for allowing such behavior.

Once an object has been created, only the set of methods that were supplied by the developer of the original class are valid operations on that object. See the **Inheritance** pattern for an example of how to alter this state of affairs.

Objects have a time at the end of their existence when they are disposed of. This deallocation has an analogous function, the deallocator, or destructor, that is called before the storage space of the data is finally released. This allows any post-conditions to be imposed on the data and resources of the object. While also a best practice to have a well-formed and definite deconstruction sequence for objects, it turns out that from a computational standpoint this is a matter of convenience only. If we had infinite resources at our disposal, objects could continue to exist, unused, for an indefinite amount of time. It is only because we have finite resources that destruction of objects is determined essential in most systems. Construction of objects, however, injects them into the working environment so they can be used in computation, and is therefore a requirement, not a convenience. Conceptually, some object types may rely on the destruction of objects to enforce certain abstract notions (fixed elements of a set, etc), but this is a matter for the class designer.

Implementation

In C++:

```
class Bunny {
```

```
public:
        Bunny( int earLength );
};

int
main(int argc, char** argv) {

        Bunny b = new Bunny(1);

}
```

Rho Calculus Definition

$$
\begin{array}{rcl}
A & \equiv & \textbf{Object}(X)[l_i\nu_i : B_i^{i\in 1..n+m}] \\
\overline{A} : \textbf{Class}(A) & \triangleq & \textbf{subclass of } \overline{A'} : \textbf{Class}(A') \\
 & & \quad \textbf{with } (\ \textbf{self } : \lfloor A \rfloor) \\
 & & \quad\quad l_i = b_i^{i\in n+1..n+m} \\
 & & \quad \textbf{override} \\
 & & \quad\quad l_i = b_i^{i\in Ovr} \\
 & & \quad \textbf{end} \\
a & = & \textbf{new}\overline{A} \\
\hline
\multicolumn{3}{c}{\textbf{CreateObject}(A, a)}
\end{array}
$$

Where $A' = \textbf{Object}(X)[l_i\nu_i : B_i^{i\in 1..n}]$ (and may be *Root*,) $\lfloor A \rfloor = A$ for O-1, $X <: A$ for O-2, and $X <\# A$ for O-3 compliant languages, respectively.

Note that it may very well be easier to implement this as an axiomatic output directly from a source code analysis tool, rather than trying to derive it from ς-calculus constructs.

Retrieve — Object Structural

Intent

To use an object from another non-local source in the local scope, thereby creating a relationship and tie between the local object and the remote one.

Motivation

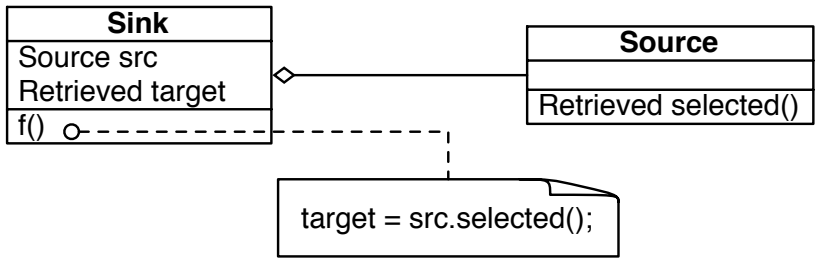
Objects are a solid mechanism for encapsulating common data and methods and enforcing policy, as shown in **CreateObject**. Singular objects, however, are of extremely limited power. In fact, if there were only one object in a system, and nothing external to it, it could be considered a procedural program - all data and methods are local and fully exposed to one another. Non-object data types can be faked in any OO system that supports the use of function objects and method-less classes. It is, therefore, critical that a well formed methodology be put into place for transporting objects across object boundaries. There are two situations where this is applicable and they differ only slightly. The simplest case is where an external object has an exposed field that is being accessed. The more complex is where an external object has a method which is called, and the return value of that method is being used in the internal scope.

Applicability

Use Retrieve when:

- A remote object provides a value object that is required for local computation and is either:
- provided by a method call's return value or,
- provided by an exposed field object.

Structure



Participants

Source

The object (or class) type that contains *selected*.

Sink

The object (or class) type that includes the item, *target*, to be given a new value.

Retrieved

The type of the value to be updated, and the value that is returned.

target

The field that is given a new value.

selected

The method or field that produces the new value.

Collaborations

This is a very simple relationship, consisting of two objects and two methods. The distinguishing factors are the transferral of a return value into the local object space, and an update to a local field using that retrieved object.

Consequences

Most object-oriented languages do not allow the updating of methods with new values (method bodies). Instead, only data fields can be updated in this manner. From a theoretical point of view, there is no appreciable difference between these two scenarios. We allow for both, relying on the language semantics to govern which cases are valid and which are not. By deferring our definition until we have a set of well formed ρ-calculus facts, we avoid much of the complexity of handling the myriad of language quirks.

Tying two objects and/or types like this is an everyday occurrence, but it is one that should not be done without thought.

Implementation

In C++:

```
class Source {
public:
        Retrieved giveMeAValue();
};

class Sink {
        Retrieved target;
        Source srcobj;
public:
        void operation() { target = srcobj.giveMeAValue(); }
};
```

Rho Calculus Definition

There are three basic forms for the **Retrieve** pattern:

$$\frac{o.s \Leftarrow o'.s' \quad o'.s' \overset{v}{\rightsquigarrow} x}{\textbf{Retrieve}(o.s, o'.s', x)}$$

$$\frac{o.s \Leftarrow o'.s' \quad o'.s' \overset{n}{\rightsquigarrow} x}{\textbf{Retrieve}(o.s, o'.s', x)}$$

Inheritance — Type Relation

Intent

To use all of another class' interface, and all or some of its implementation.

Also Known As

IsA

Motivation

Often, an existing class will provide an excellent start for producing a new class type. The interface may be almost exactly what you are looking for, the existing methods may provide *almost* what you need for your new class, or, at the very least, the existing class may be conceptually close to what you wish to accomplish.

In such cases, it would be highly useful and efficient to reuse the existing class instead of rewriting everything from scratch. One way of doing so is by the use of **Inheritance**. Every object-oriented language supports this approach of code reuse, and it is usually a core primitive of that language.

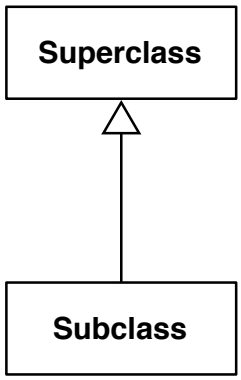
At its most basic level, this pattern offers a relationship between a *superclass* or *base class*, and a *subclass* or *derived class*. The superclass (let us call it Superclass) is an existing class in the system, one that provides at a minimum an interface of concepts for methods, and/or data structures. A second class can be defined as being *derived* from Superclass, let us call it Subclass. The Subclass class *inherits* the interface and implementations of all methods and fields of Superclass, and this provides a starting point for a programmer to begin work on Subclass.

Applicability

Use Inheritance when:

- Two classes are related conceptually, providing the same services.
- One of those class' implementation or interface is a superset of the other...
- *or* one class is mostly the same as the other.

Structure



Participants

Superclass

An existing class in a system that is used as a basis for producing a further class

Subclass

The secondary class that relies on the first for its basic interface and implementation

Collaborations

The Superclass class creates a basic set of method interfaces and possibly accompanying method implementations. Subclass inherits all the interface elements of Superclass, and, by default, all of the implementations as well, which it may choose to override with new implementations.

Consequences

Inheritance is a powerful mechanism, but has some interesting limitations and consequences. For one thing, a subclass may not *remove* a method or data field in most languages. There a few exceptions, but these are rare, and beyond the scope of this pattern, as this gets to the heart of a subtlety of object-oriented theory. A subclass is limited to overriding a method with a new implementation, or adding new methods or data fields.

It may seem that overriding is a waste of good code in the base class, and in many cases this is true. Look to the **ExtendMethod** pattern for a solution to this problem.

In some cases, one may not wish to inherit an entire existing class, but instead just small pieces of functionality may be desired. This may be due to a lack of confidence in the actual implementations of the method bodies, often when the original source code is unavailable, a reluctance to absorb the cost of integrating a large class into the current system when only a

small segment of that class is needed, or other scenarios. In such situations, consider instead the **Delegate** pattern.

Implementation

The mechanism for creating an inheritance relationship will vary from language to language, but is almost always readily apparent. Assume that we are modeling a rabbit farm, and wish to keep track of the health of the rabbits on hand. We make a class Rabbit that contains the basic info for all rabbits, of any type. In addition, though, for lop-eared rabbits, we want to keep track of the ear length for breeding purposes. We can subclass a new class LopEaredRabbit from Rabbit, gaining all the necessary data and methods to model a rabbit, and we can then add additional information as we see fit:

In C++:

```
class Rabbit {
public:
        Rabbit( );
};

class LopEaredRabbit : public Rabbit {
public:
        LopEaredRabbit( );
        float   getEarLength();
        void    setEarLength(float newEarLength);
private:
        float earLength;
};
```

Rho Calculus Definition

$$\frac{E \vdash C,\ C : [l_i : B_i^{i \in 1..n}] \qquad C \equiv [l_i = b_i^{i \in 1..m-1, m+1..n}, l_m = []]}{\textbf{AbstractInterface}(C, l_m)}$$

AbstractInterface — Type Relation

Intent

To provide a common interface for operating on an object type family, but delay definition of the actual operations to a later time.

Also Known As

Virtual Method, Polymorphism, Defer Implementation

Motivation

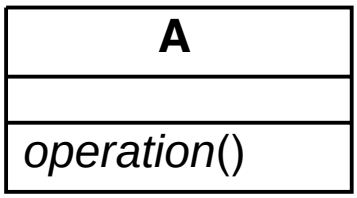
Often, while we have a hierarchy of classes using the **Inheritance** pattern that conforms well to our conceptual design, we run into a situation where we simply cannot provide a meaningful method implementation. We know what we want to do *conceptually*, we are just not entirely sure exactly how to go about doing it.

Applicability

Use Abstract Interface when:

- Implementation of a method can not be known until a later date.
- The interface for that method *can* be determined.
- You expect subclasses to handle the functionality of the method.

Structure



Participants

A

The class type that declares an interface for *operation*

Collaborations

A defines an interface for a method that some unknown later subclass will implement.

Consequences

The **CreateObject** pattern lets us instantiate objects, and the **Retrieve** pattern shows how to fill in the fields of that newly created object. **AbstractInterface** is unusual in that it indicates the *absence* of a method implementation; instead of showing us how to fill in the method, it shows us how to defer the method definition until a later date. Note that this pattern does not actually provide the full picture, including the later definition of the method. See the related **FulfillMethod** pattern for the rest of the solution.

In this case, the method is declared in order to define the proper interface for our conceptual needs, yet the method body is left undefined. This does *not* mean that we simply define an empty method, one that does nothing; the method has *no definition at all*. This is a critical point, and one that is often missed by new object-oriented programmers. How this is done will vary from language to language. For instance, in C++ the method is set 'equal to zero' as shown in the example code.

Implementation

In C++:

```
class AbstractOperations {
public:
     virtual void operation() = 0;
};

class DefinedOperations : public AbstractOperations {
public:
     void operation();
};

void
DefinedOperations::operation() {
        // Perform the appropriate work
}
```

Rho Calculus Definition

$$\frac{E \vdash C,\ C : [l_i : B_i^{i \in 1..n}] \qquad C \equiv [l_i = b_i^{i \in 1..m-1, m+1..n}, l_m = []]}{\textbf{AbstractInterface}(C, l_m)}$$

Delegate

Object Behavioral

Intent

To parcel out, or delegate, a portion of the current work to another method in another object.

Also Known As

Messaging, Method Invocation, Calls, The Executive

Motivation

In the course of working with objects, the situation often arises in which 'some other object' can provide a piece of functionality we want to have. **Delegate** embodies the most general form of a method call from one object to another, allowing one object to send a message to another, to perform some bit of work. The receiving object may or may not send back data as a result.

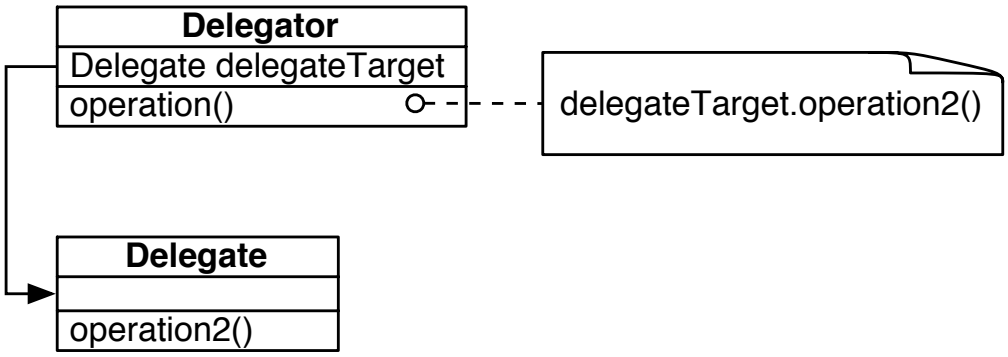
A real world example of this is a corporate executive. When they have a large task to be done, they ask subordinates to do pieces of it, with any two subordinates rarely in charge of the same portion of the implementation. The subordinates' tasks are different from each other, and very different from the executive's. See **Redirect** for another anthropomorphized example.

Applicability

Use Delegate when:

- Another object can perform some work that your current method body wishes to have done.

Structure



Participants

Delegator

The object type sending the message to the Delegate

operation

The method within the Delegator that is currently being executed when the message is sent - the point of invocation of the *operation2* method call

Delegate

The object type receiving the message, with an appropriate method to be invoked

operation2

The method being invoked from the call site

Collaborations

This is a simple binary relationship, one method calls another, just as in procedural systems, and with the same sorts of caveats and requirements. Since we are working in an object-oriented realm, however, we have a couple of additional needs. First, we require that the object being called upon to help with the current task must be visible at the point of invocation. Second, we require that the method being invoked must be visible external to the enclosing object.

Consequences

All operations between any two objects can be described as an instance of the **Delegate** pattern, but it is much more useful to be able to describe further attributes of the relationship. See the further Method Invocation EDPs for refinements of **Delegate** that will be more useful.

Implementation

The most generalized and basic style of method invocation in object-oriented programming, **Delegate** describes how two objects communicate with each other, as the sender and receiver of messages, performing work and returning values.

In C++:

```
class Delegator {
public:
        Delegatee       target;
        void operation() {target.operation2();};
}

class Delegatee {
public:
        void operation2();
}
```

Rho Calculus Definition

This is the first of our reliance operator-defined EDPs, and we introduce the transitive reliance operator $<_\mu$, for a method invocation.

$$\frac{\begin{array}{l} Delegator : [target : Delegator, operation : B_i] \\ Delegator \equiv [operation \Leftarrow ..., target.operation2, ...] \\ Delegatee : [operation2 : B_i] \\ del : Delegator \\ del.operation <_\mu^{-\cdot -} target.operation2 \end{array}}{\textbf{Delegate}(del, target, operation, operation2)}$$

Redirect

Object Behavioral

Intent

To request that another object perform a tightly-related subtask to the task at hand, perhaps performing the basic work.

Also Known As

Tom Sawyer

Motivation

A small refinement to **Delegate**, **Redirect** takes into consideration that methods performing similar tasks are often named similarly. We can take advantage of this clarify the intent of this pattern over the more general form.

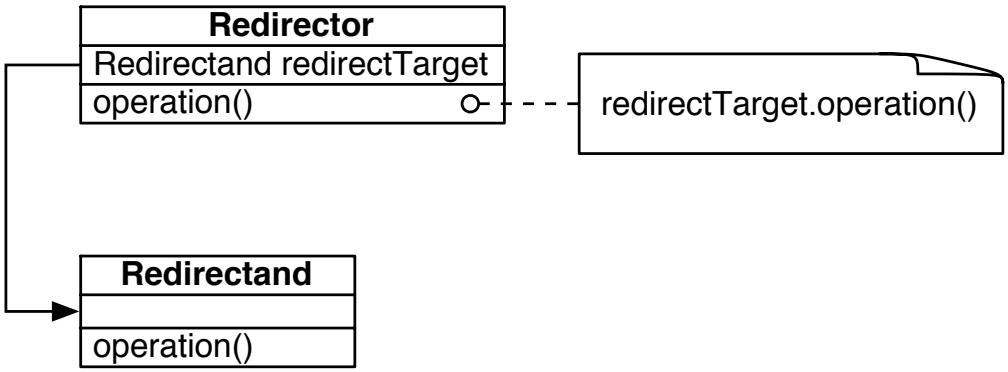
Another name for this pattern is **TomSawyer**, a literary figure who was famous for convincing other people to do his tasks for him. Unlike the **Executive** alias in **Delegate**, Tom generally sought to have others do exactly the task he was asked to do.

Applicability

Use Redirect when:

- A task to be performed by a method can be broken down into subtasks.
- One of those subtasks can be achieved by using another object.
- That target object has a method that has a similar intent, expressed through its signature name after Kent Beck's **Intention Revealing Selector** pattern.
- There is no distinct type relationship between the two objects.

Structure



Participants

Redirector

The originating site of the method call contains a method named *operation* which has a subtask to be parceled out to another object. This object, redirectTarget, is a field element of type Redirectand. Redirector's *operation* calls redirectTarget.*operation* to perform a portion of its work.

Redirectand

The receiver of the message, which performs the subtask asked of it.

Collaborations

As with **Delegate** (and indeed all the Method Invocation EDPs), **Retrieve** describes a binary relationship between two objects and their enclosed methods.

Consequences

Almost identical to the **Delegate** pattern, the seemingly small addition of similarity of methods has some far-reaching effects. We will see in later patterns that when this pattern is combined with other EDPs and typing information, complex interactions can quickly be formed and described simply.

By leveraging the fact that both methods have the same name, and that this is a common way of declaring the intent of a method, we can deduce that the two methods have a common intent of functionality. Furthermore, it becomes obvious that this is an appropriate way of indicating that our originating call site method is requesting the invoked method to do some portion of work that is tightly related to the core functionality of the original method.

Implementation

In C++:

```
class Foo {
public:
        operation();
};

class Bar {
        Foo f;
public:
        operation() { f.operation(); };
};
```

Rho Calculus Definition

The '+' annotation to the μ subscript in the reliance operator indicates that the method names are similar, under the principle of similarity. This distinguishes the **Redirect** group of patterns from the **Delegate** group. We will see this distinction appear again and again, with some interesting results.

$$\frac{\begin{array}{l} Redirector : [target : Redirectand, operation : B_i] \\ Redirector \equiv [operation \Leftarrow ..., target.operation, ...] \\ Redirectand : [operation : B_i] \\ red : Redirector \\ redirector.operation <_{\mu}^{-.+} target.operation \end{array}}{\textbf{Redirect}(red, target, operation)}$$

Conglomeration — Object Behavioral

Intent

To bring together, or conglomerate, diverse operations and behaviors in order to complete a more complex task within a single object.

Also Known As

Decomposing Message

Motivation

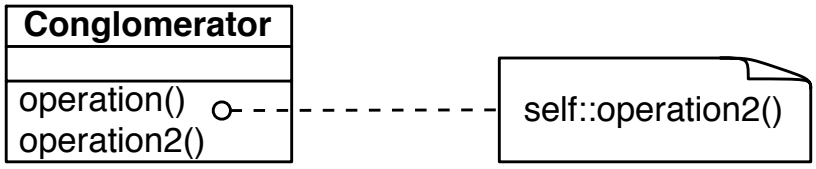
An object is often asked to perform a task which is too large or unwieldily to be performed within a single method. Breaking the task into smaller parts to be handled individually as discrete methods, and then building them back into a whole result by the method responsible for the larger task usually makes conceptual sense. Kent Beck refers to this as the **Decomposing Message** pattern (Beck, 1997). It may also happen that related subtasks can be unified into single methods, resulting in reuse of code inside a single object.

Applicability

Use Conglomeration when:

- A large task can be broken into smaller subtasks.
- The subtasks must be performed on a single object instance.
- Several subtasks may be unified into a single method body.

Structure



Participants

Conglomerator

Enclosing object type

operation

Master controlling method that is parcelling out subtasks

operation2

Subservient method performing a particular subtask

Collaborations

A specialization of **Delegate**, the object is calling a method of itself. The calling site is *operation*, and the method being called is *operation2*.

Consequences

As with **Delegate**, this pattern ties two methods into a reliance relationship, with *operation* relying on the behavior and implementation of *operation2*. In this case, unlike **Delegate**, there may be immediate side effects to shared data within the confines of the object that they share.

Implementation

In C++:

```
class Conglomerate {
public:
void operation() { operation2(); };
void operation2();
}
```

Rho Calculus Definition

$$\frac{\begin{array}{l} c : Conglomerator \\ c.operation <_{\mu}^{+\cdot -} c.operation2 \end{array}}{\textbf{Conglomeration}(c, operation, operation2)}$$

Recursion — Object Behavioral

Intent

To accomplish a larger task by performing many smaller and similar tasks, using the same object state.

Motivation

Sometimes we find that, after analysis, a problem can be broken down into smaller subtasks that are identical to the original task, except on a smaller scale. Sorting an array of items using the Merge Sort algorithm is one such example. Merge Sort takes an array and divides it into two halves, sorting each individually, then merges the two sorted arrays into a unified whole. The two half-problems are also sorted using Merge Sort, so they are subject to the same halving of the problem, and so on. Eventually, arrays of a single item are reached, at which point the merging begins.

The process by which a method calls itself is known as *recursion*, and it is ubiquitous in general programming. In object-oriented programming, the same principle applies, except that we have the added requirement that the object must be calling on itself, through an implicit or explicit use of *self*. See **Redirect** for an example of calling another object, and **RedirectedRecursion** for calling another object of the same type. **Recursion** is a specialization of **Redirect** using the same object instance.

Recursion is, generally speaking, a way of folding a large amount of computation into a small conceptual space. Assume we want to sort an array, and we decide that a simple way to sort a large array is to split it into two roughly equal sized arrays, and then merge the two subarrays after they are individually sorted. The merging will be easy - if the head of array A is less than the head of array B, then the head of A is copied to the new, larger array, otherwise, copy the head of B. Remove the copied item from the appropriate array, and repeat until both arrays are merged.

Because the Merge Sort scheme relies on the proper sorting of the subarrays, we correctly surmise that we can perform the sort algorithm on the two subarrays, by splitting, sorting, then

merging each in turn. We are again faced with the same sorting problem, so we continue in the same manner until we reach the smallest indivisible array, namely a single item. At that point, merging of sorted subarrays at each step can begin.

Assume we have an Array class that has the usual methods of add and delete. If the length of the beginning array was 4 items, then we could hardcode the entire sorting in pseudo-code:

```
Array
sort_merge(Array a) {
        a11 = a[0];
        a12 = a[1];
        a21 = a[2];
        a22 = a[3];
        // Sort first half
        if (a11 < a12) {
                a1.add(a11);
                a1.add(a12);
        } else {
                a1.add(a12);
                a1.add(a11);
        }
        // Sort second half
        if (a21 < a22) {
                a2.add(a21);
                a2.add(a22);
        } else {
                a2.add(a22);
                a2.add(a21);
        }
        // Merge
        if (a1[0] < a2[0]) {
                res.add(a1[0]);
                a1.delete(0);
        } else {
                res.add(a2[0]);
                a2.delete(0);
        }
        if (a1[0] < a2[0]) {
                res.add(a1[0]);
                a1.delete(0);
        } else {
                res.add(a2[0]).
                a2.delete(0);
        }
        if (a1.length == 0) {
                res.add(a2[0]);
                res.add(a2[1]);
        }
        if (a2.length == 0) {
                res.add(a1[0]);
                res.add(a1[1]);
        }
```

```
        if (a1.length == 1 && a2.length == 1) {
                res.add(a1[0]);
                res.add(a2[0]);
        }
}
```

While highly efficient, this has an obvious drawback: it is limited to only working for arrays of length 4. A much more general version of this is one using looping. Here we illustrate a for loop implementation, assuming for simplicity that the length of the array is a power of 2:

```
sort_array(Array a) {
        // Slice a into subarrays
        subarray[0][0] = a
        for (i = 1 to log_2(a.length())) {
                for (j = 0 to 2^i) {
                        prevarray = subarray[i-1][floor(j/2)]
                        subarray[i][j] = prevarray.slice(
                                (j mod 2) * (prevarray.length() / 2),
                                ((j mod 2) + 1) * (prevarray.length() / 2))
                }
        }
        // Sort subarrays and merge
        for (i = log_2(a.length()) to 0) {
                for (j = 1 to 0) {
                        subarray[i][j]
        }
        return subarray[0][0]
}
```

This succeeds in making our algorithm much more flexible, but at a cost of making it almost unreadable due to the overhead mechanisms. Note that we are looping inward to the base case (length of array is a single unit), storing state at every step, then looping outward using the state previously stored. This process of enter, store, use store, and unroll is precisely what a function call accomplishes. We can use this fact to vastly simplify our implementation using Recursion:

```
Array
sort_array(Array a, int beg, int end) {
        if (a.length() > 1) {
                return merge(sort_array(a, beg, end-beg/2),
                                sort_array(a, (end-beg/2) + 1, end));
        } else {
                return a
        }
}

Array
merge(Array a, Array b) {
        Array res;
        while (b.length() > 1 || a.length() > 1) {
```

```
                // If a or b is empty, then a[0] or b[0] returns a min value
                if (a[0] < b[0]) {
                        res.add(a[0]);
                        a.delete(0);
                } else {
                        res.add(b[0]);
                        b.delete(0);
                }
        }
        return res;
}
```

This is a much cleaner conceptual version that performs the same task as our looping variation. We allow the runtime of the language to handle the overhead for us.

There are times, such as in a high-performance embedded system, when that manner of overhead handling may be determined to be excessive for our needs. In such cases, optimizing away the recursion into loops, or further into linear code, is a possibility. These instances are extreme, however, and tend to be highly specialized. In most cases the benefits of Recursion, conceptual cleanliness, simple code, greatly outweigh the small loss of speed.

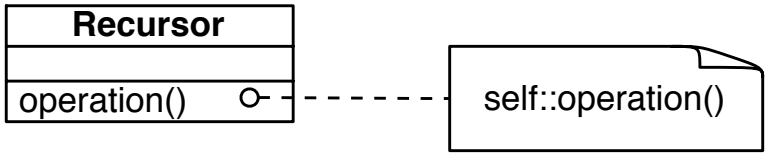
The example above is based on the recursion of a **method**, but that is strictly for simplification of the problem. In an object-oriented system, there are no free methods; any method that is truly recursive is, therefore, bound to a particular object, not just a specific type.

Applicability

Use Recursion when:

- A task can be divided into highly similar subtasks.
- The subtasks must be, or are preferred to be, performed by the same object, usually due to a necessity of access to common stored state.
- A small loss of efficiency is overshadowed by the resulting gain in simplicity.

Structure



Participants

Recursor

The only participant, Recursor has a method that calls back on itself, within the same instantiation.

Collaborations

Recursor's method *operation* collaborates with itself, requesting that smaller and smaller tasks be performed at each step, until a base case is reached, at which point results are gathered into a final result.

Consequences

The reliance of Recursion on a properly-formed base case for termination of the recursion stack is the weakest point of this pattern. It is nearly impossible in most modern systems to consume wantonly all available resources, but recursion can do it easily by having a malformed base case that is never satisfied.

Implementation

In C++:

```
class SortedArray {
sort_merge() {

}
```

Rho Calculus Definition

$$\frac{\begin{array}{l} Recursor : [l_i : B_i{}^{i \in 1 \ldots m}, operation : B_{m+1}], \\ r : Recursor, \\ r.operation <_{\mu}^{+.+} r.operation \end{array}}{\textbf{Recursion}(r, operation)}$$

RevertMethod — Object Behavioral

Intent

Bypass the current class' implementation of a method, and instead use the superclass' implementation, reverting to an 'earlier' method body.

Motivation

Polymorphism sometimes works against us. One such instance is providing multiple versions of classes for simultaneous use within a system. Imagine a library of classes for an internet data transfer protocol. A base library is shipped as 1.0. With the 1.1 library, changes to the underlying protocol are made, and an application using the 1.1 protocol must be able to fall back to the 1.0 protocol when it detects that the application at the other end of the connection is only 1.0 enabled.

One approach would be to use polymorphism directly, and have a base class that abstractly provides the protocol's methods, as in Figure B.1, and create the proper class item on protocol detection. Unfortunately, this does not offer a lot of dynamic flexibility, and would not allow a protocol to adapt on the fly, so that the two ends can handle graceful degradation of the connection. You could instantiate objects of each protocol type, and swap back and forth as needed, but there may be protocol state issues that would be troublesome.

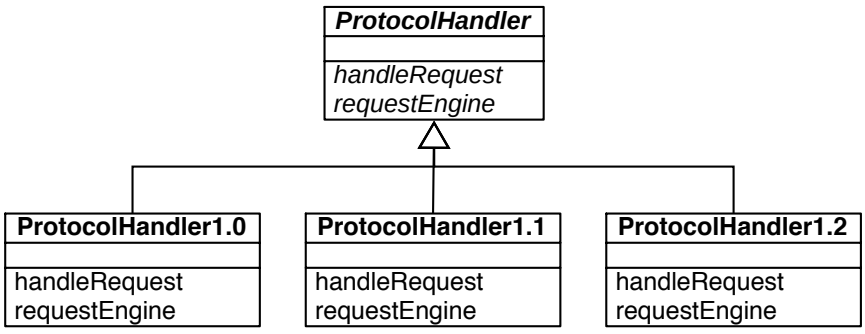


Figure B.1: Polymorphic approach

We may instead elect to instantiate an object of a base protocol class subclassed by a 1.1

version class. We now have only a single class to deal with, but we still need to be able to revert to the previous version. In Figure B.2 we show an extended variant of this approach, including several versions.

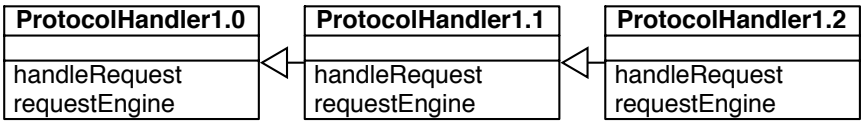


Figure B.2: Subclassing approach

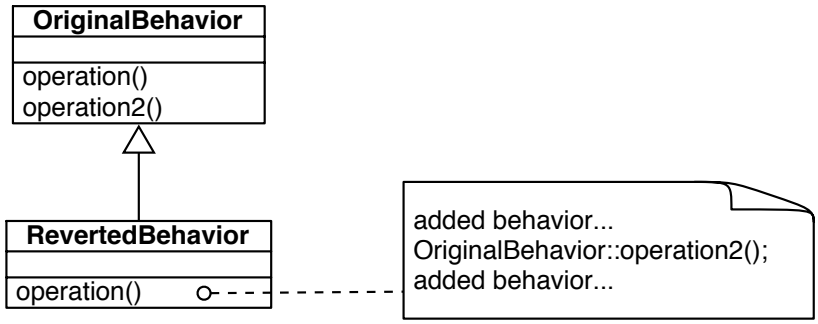
In this case we can instantiate an object of just the last class in the chain, ProtocolHandler1.2, and conditional statements in the code will pass the protocol handling back up the chain to the appropriate version if needed. We can also make a simple test for whether or not the current object's version is appropriate for the protocol, and, if not, pass it simply to *super*, and let the test be re-enacted there, and so on. This vastly simplifies further maintenance of code that supports this protocol, since only the instance(s) of the ProtocolHandler need to be changed to the latest version anytime a new update to the library comes out, and all the graceful degradation is handled automatically. A specific version could also be hardcoded for various reasons; it is up to the application developer.

Applicability

Use RevertMethod when:

- A class wishes to *re-enable* a prior implementation of a method that it has overridden.

Structure



Participants

OriginalBehavior

A base class, defining two methods, *operation* and *operation2*.

RevertedBehavior

A subclass of OriginalBehavior, with *operation* and *operation2* overridden. *operation* calls the OriginalBehavior implementation of *operation2* when one would normally expect it to call its own implementation of that method.

Collaborations

In most cases, when a subclass overrides a parent class' method, it is to replace the functionality, but the two method definitions can work together to allow an extension of the behavior. RevertedBehavior relies on OriginalBehavior for a core implementation.

Consequences

There is an odd conceptual disconnect between the overriding of a base class' method, and the utilization of that same method that can be confusing to some students. Overriding a method does not erase the old method; it merely hides it from public view for objects of the subclass. The object still has knowledge of its parent's methods, and can invoke them internally without exposing this to the external world.

Implementation

In C++:

```
class OriginalBehavior {
public:
        void operation() { operation2(); };
protected:
        void operation2();
};

class RevertedBehavior: public OriginalBehavior {
public:
        void operation() {
                if (oldBehaviorNeeded) {
                        OriginalBehavior::operation2();
                } else {
                        operation2();
                }
                                };
private:
        void operation2();
};
```

Rho Calculus Definition

$$\frac{\begin{array}{l} OriginalBehavior : [l_i : {B_i}^{i \in 1 \ldots m}, operation : B_{m+1}, operation2 : B_{m+2}], \\ RevertedBehavior : [l_i : {B_i}^{i \in 1 \ldots m}, operation : B_{m+1}, operation2 : B_{m+2}], \\ RevertedBehavior <: OriginalBehavior, \\ rb : RevertedBehavior, \\ rb.operation <_{\mu}^{+\cdot -} rb \wedge operation2 \end{array}}{\textbf{RevertMethod}(OriginalBehavior, RevertedBehavior, operation, operation2)}$$

Extend Method

Object Behavioral

Intent

Supplement, not replace, behavior in a method of a superclass while reusing existing code.

Also Known As

Extending Super

Motivation

The behavior of a method often needs to be altered or extended, such as when fixing a bug in the original method when the source code is unavailable, or adding new functionality without changing the original method.

One of the most common ways of doing this is, of course, to cut and paste the old code into the new method, but this presents a host of problems, including consistency of methods, and results in a potential maintenance morass. Adhering to the Single Point Principle and, instead, reusing the existing code, tweaking the results as needed is a much less error-prone approach.

It is possible, of course, to create a reference to a delegate object with the original behavior, and call into it when needed. This is the approach taken in **Redirect**.

In other cases, it is necessary or desired to subclass directly off of the original class. In such cases, we have two options for extending the original method. One is to cut and paste the old code into the new subclass' method, but this is not only undesirable from a maintenance standpoint, but it also may not be possible, as in cases where the original source code is unavailable.

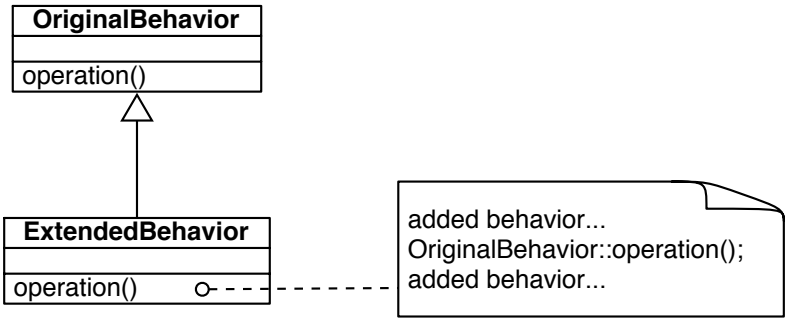
The second is **ExtendMethod**. In this case, we use **Inheritance** to provide the mechanism for reuse. We then override the original method, but make a call back to the superclass' implementation of the method. This provides us with simple maintenance, reuse, and encapsulation of the altered behavior.

Applicability

Use Extend Method when:

- Existing behavior of a method needs to be extended, but not replaced.
- Reuse of code is preferred or necessitated by lack of source code.
- Polymorphic behavior is required.

Structure



Participants

OriginalBehavior

Defines interface, contains a method with desired core functionality.

ExtendedBehavior

Uses interface of OriginalBehavior, re-implements method as call to base class code with added code and/or behavior.

Collaborations

In most cases when a subclass overrides a parent class' method, the purpose is to replace the functionality of that original method. The two method definitions, however, can work together to allow an extension of the behavior of the parent class instead of a replacement. ExtendedBehavior relies on OriginalBehavior for both interface and core implementation.

Consequences

As with **RevertMethod**, the concept of calling an overridden version of a method can be confusing to some students. In **ExtendMethod** it can be even more so, because the calling method seems to be invoking a ghost method. Code reuse is optimized using this pattern, but the method Operation in OriginalBehavior becomes somewhat fragile - its behavior is now

relied upon by ExtendedBehavior::Operation to be invariant over time. Behavior is extended polymorphically and transparently to clients of OriginalBehavior.

Implementation

In C++:

```
class OriginalBehavior {
public:
     virtual void operation();
};

class ExtendedBehavior : public OriginalBehavior {
public:
     void operation();
};

void
OriginalBehavior::operation() {
     // do core behavior
}

void
ExtendedBehavior::operation() {
     this->OriginalBehavior::operation();
     // do extended behavior
}
```

This pattern should translate very easily to most any object-oriented language that supports inheritance and invocation of a superclass' version of a method.

Rho Calculus Definition

$$\frac{\begin{array}{l} operation \in \mathbf{meth}(OriginalBehavior), \\ ExtendedBehavior <: OriginalBehavior, \\ eb : ExtendedBehavior, \\ eb.operation <_{\mu}^{+\cdot +} eb\hat{}operation \end{array}}{\mathbf{ExtendMethod}(OriginalBehavior, ExtendedBehavior, operation)}$$

DelegatedConglomeration — Object Behavioral

Intent

A **Conglomeration** pattern is appropriate, but we need to work with a distinct instance of our object type, resulting in a need for the **Delegate** pattern to be used.

Motivation

We often have objects of the same type working in concert to perform tasks. Homogeneous data environments with heterogeneous behavior are frequently coded as a collection of like objects collaborating to produce a larger functionality. Take, for instance, an atomic fission simulation. The atoms are all of the same type, and therefore can be of the same class.

```
class Atom {
    Atom*   target1;
    Atom*   target2;

    void split() {
        target1->hit();
        target2->hit();
    }

    void hit() {
        if (this->isUnstable()) { this->split(); };
    };
};
```

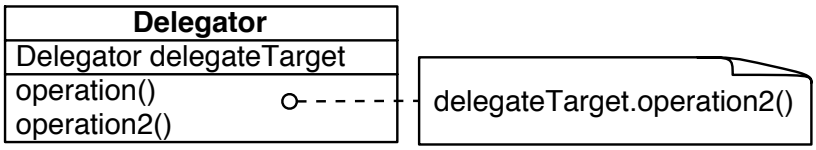
Each atom, on fissioning, sends out two neutrons which, it is assumed, will strike other neighboring atoms. The atoms then are responsible for determining if they are stable enough to absorb the neutron, or if they are going to continue the reaction.

Applicability

Use DelegatedConglomeration when:

- A task can be broken into subtasks that are properly handled by the same object type.
- Many objects of the same type work in concert to complete a task.
- A single object is unable to complete the task alone.

Structure



Participants

Delegator

The object type that contains references to other instances of its own type

delegateTarget

The enclosed instance that is called upon to perform a task

operation

The calling point within the first object

operation2

The subtask to be completed

Collaborations

Only one type is involved, but two instances. One object relies on the other to perform part of the task, as with **Delegation**, and that sub-task is not directly associated with the current request, as with **Conglomeration**.

Consequences

As with **RedirectedRecursion**, this pattern offers the ability to distribute tasks among a number of objects. Unlike its cousin, however, **DelegatedConglomeration** offers a clear decision making point, separating the behavior-determining logic from the behavior implementation. This is most useful in situations where the objects are the same, but their behavior may be determined by local effects.

Implementation

In C++:

```
class Delegator {
        Delegator       delegateTarget;
public:
        void operation() { delegateTarget.operation2(); };
        void operation2();
}
```

Rho Calculus Definition

$$\frac{\begin{array}{l} del : DelConglomerator \\ conglomTarget : DelConglomerator \\ del.operation <_{\mu}^{-\cdot -} conglomTarget.operation2 \end{array}}{\textbf{DelegatedConglomeration}(del, conglomTarget, operation, operation2)}$$

RedirectedRecursion — Object Behavioral

Intent

To perform a recursive method, in particular, one that requires interacting with multiple objects of the same type.

Motivation

Frequently, we will wish to perform an action that is recursive in nature, but it requires multiple objects working in concert to complete the task. Imagine a line of paratroopers getting ready for a jump. Space it tight, so instead of the commander indicating to each trooper to jump at the door, he stands at the back of the line, and when time has come, taps the last trooper on the shoulder. He knows to tap the shoulder of the trooper in front of him, and when that soldier has jumped, jump himself. This can continue on down a line of arbitrary length, from 2 to 200 troopers. The paratroopers' only tasks are to, when they feel a tap on their shoulder, tap the next person in line, wait, shuffle forward as space is available, and when they see the soldier in front of them go, jump next. The commander issues one order, instead of one to each soldier. A sample coding of this might look like:

```
Paratrooper::jump() {
        nextTrooper->jump();
        while (! nextTrooper->hasJumped() ) {
                shuffleForward();
        }
        leap();
}
```

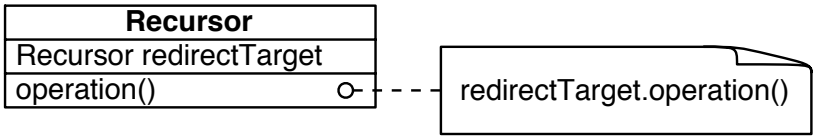
The current paratrooper cannot jump until the trooper in front has completed the task, and so on, and so on.

Applicability

Use RedirectedRecursion when:

- Recursion is a clean way of breaking up the task into subparts.
- Multiple objects of the same type must interact to complete the task.

Structure



Participants

Recursor

An object type that holds a reference to another instance of its own type

redirectTarget

The enclosed instance

operation

A method within Recursor that is recursive on itself, but through redirectTarget

Collaborations

Multiple instances of the same object type interact to complete a task. Each instance knows how to message the next object for its turn.

Consequences

This is a powerful method for recursive behavior when using a number of objects, and it is found in many systems. It allows functionality to be split among disparate data sets, but constrains the functionality to a particular behavior. It is analogous to SIMD computing in hardware, except that in this case the data is responsible for passing along the instruction. This can be a highly flexible approach to solving problems that require a divide and conquer algorithm, as changing one method's implementation propagates across all objects equally.

Implementation

In C++:

```
class Recursor {
        Recursor        redirectTarget;
public:
        void operation() { redirectTarget.operation(); };
}
```

Rho Calculus Definition

$$\frac{\begin{array}{l} rec : Recursor \\ redirectTarget : Recursor \\ rec.operation <_{\mu}^{-\cdot+} redirectTarget.operation \end{array}}{\textbf{RedirectedRecursion}(rec, redirectTarget, operation)}$$

DelegateInFamily — Object Behavioral

Intent

Related classes are often defined as such to perform tasks collectively. In these cases, multiple objects of related types can interact in generalized ways to delegate tasks to one another.

Motivation

User interfaces are a familiar type of system in which to find **DelegateInFamily** and related patterns (**DelegateInFamily**, **RedirectInFamily**, **RedirectInLimitedFamily**). This pattern allows one to parcel out tasks within a family of classes, often called a class cluster, when the interface, and method name, are known, but the precise object type, and therefore method body, may not be. It is a form of polymorphic delegation, where the calling object is one of the polymorphic types.

Consider a windowing system that includes slider bars and rotary dials as input controls, and text fields and bar graphs as display widgets. An input control is tied to a particular display widget and sends it updates of values when the control is adjusted by the user. The input controls don't need to know precisely what kind of display widget is at the other end, they just need to know that they must call the *updateValue* method, with the appropriate value as a parameter. Since input controls also display a value implicitly, it is possible to programmatically change their adjustment accordingly, so they too need an *updateValue* method. By our **Inheritance** pattern, it seems the input controls and display widgets are of the same family and, in fact, we want to make sure that they can all interact, so we create a class hierarchy accordingly:

```
class UIWidget {
    void updateValue( int newValue );
};

class InputControl : UIWidget{
    UIWidget*   target;

    void userHasSetNewValue()  {...  target->updateValue(myNewValue); ...}
};

class SliderBar : InputControl {
```

```
    void updateValue( int newValue );   // Moves the slider bar accordingly
    void acceptUserClick() {... userHasSetNewValue(); ...};
};

class RotaryKnob : InputControl {
    void updateValue( int newValue );   // Rotates the knob image accordingly
    void acceptUserClick() {... userHasSetNewValue(); ...};
};

class DisplayWidget : UIWidget {
    GraphicsContext gc;
};

class TextField : DisplayWidget {
    void updateValue( int newValue ) { gc.renderAsText( newValue ); };
};

class BarGraph : DisplayWidget {
    void updateValue ( int newValue ) { gc.drawBarLengthOf( newValue ); };
};
```

In the above example, the InputControl and UIWidget classes fulfill the necessary roles in the **DelegateInFamily** pattern.

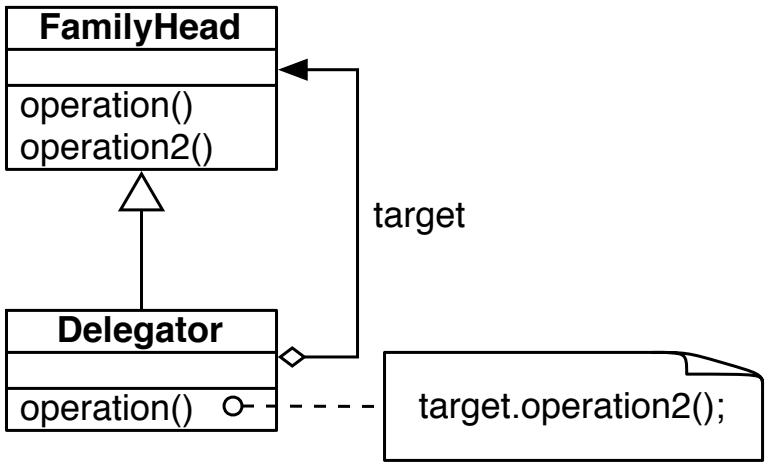
The SliderBar and RotaryKnob objects do not have to know anything about where their value is going and, in fact, they could be tied to each other, with each adjusting the other in sync. You could even have two objects of the same InputControl subclass tied together, such as two SliderBar instances. We have separated the concerns of who is sending what data and who is receiving it. Our only concerns are that the data is being sent to a properly-receiving client polymorphically, and that the current calling object is *of that polymorphic family*. This allows for a single, unified interface for many classes that can work in tandem to perform many tasks.

Applicability

Use DelegateInFamily when:

- Delegation is appropriate, with related and/or unrelated subtasks to be performed.
- Polymorphism is required to properly handle the message request.
- The calling object is of a type in the polymorphic hierarchy.

Structure



Participants

FamilyHead

The base class for a polymorphic class cluster

Delegator

A subclass of FamilyHead

target

A polymorphic instance of FamilyHead that is contained by Delegator

operation

The calling site

operation2

The called subtask

Collaborations

FamilyHead provides not only a base interface for Delegator, but an instance of FamilyHead resides within the Delegator to handle requests. This instance is understood to be polymorphic, and may handle a request in a number of different ways. **DelegateInFamily** differs from **RedirectInFamily** in that it is a more generalized form, and is parcelling out subtasks that are related to, not refinements of, the initiating method.

Consequences

As with any subtyping relationship, Delegator is tied to the interface of FamilyHead. The implementation of the target method is subject to the particular subclass that it ends up being

contained in, via polymorphism. Because of this, this pattern may end up with unintended consequences if another class within the class hierarchy implements its methods in an unexpected way. On the other hand, this is a powerful mechanism for extending functionality by adding additional classes to the class family. If the possible extensions need to be limited in some way, consider using the **DelegateInLimitedFamily** pattern instead.

Implementation

The *target* can be defined in either the base class or the subclass; it just needs to be accessible from with the subclass.

In C++:

```
class FamilyHead {
        void operation();
        void operation2();
};

class Delegator : public FamilyHead {
        FamilyHead      target;
        void operation() { target->operation2(); };
};
```

Rho Calculus Definition

$$\frac{\begin{array}{l} Delegator <: FamilyHead, \\ d : Delegator, \\ fh : FamilyHead, \\ d.operation <_{\mu}^{-\cdot -} fh.operation2, \end{array}}{\textbf{DelegateInFamily}(d, fh, operation, operation2)}$$

RedirectInFamily — Object Behavioral

Intent

Redirect some portion of a method's implementation to a possible cluster of classes, of which the current class is a member.

Motivation

Frequently, a hierarchical object structure of related objects will be built at runtime, and behavior needs to be distributed among levels. This is an effective way to move responsibility for the handling of a request through a number of objects, and is a core piece of the **ChainofResponsibility** and **Composite** patterns (Gamma et al., 1995). **RedirectInFamily** can be considered a single link of such chains.

The Redirection implies that the intent behind the primary method and the target method are similar, and that there is a strong correlation between the two methods. By ensuring that the type of the recipient object is a supertype of the current object, a much stronger correlation can be established.

For example, in event handling systems, it is common to have a hierarchy of event-aware elements. Data views and controls are contained within panes, which are composed into windows, which sit in in a global UI environment, perhaps with other elements. Any of these elements may be asked to handle a user-generated event, but it may be a context-driven handling. A window that is in the background may react to mouse events differently than one in the foreground, for example. All the elements are expected be *able* to handle an event appropriately, but they may not all be asked to handle them the same way *at any specific point in time*. This temporal dynamicism is a common, but not necessary, constraint leading to the use of **RedirectInFamily**.

Example code for an implementation of an event handler could be defined as:

```
class EventHandler {
    EventHandler*   nextHandler;

    void handle(Event* e) { ...
        if ( !(this->isActive()) ) {nextHandler->handle(e);}
         ...};
};
```

```
class Event {};

class MouseEvent : Event {
    bit_array   modifierKeys;
};

class Button : EventHandler {};

class TabPane : EventHandler {};
```

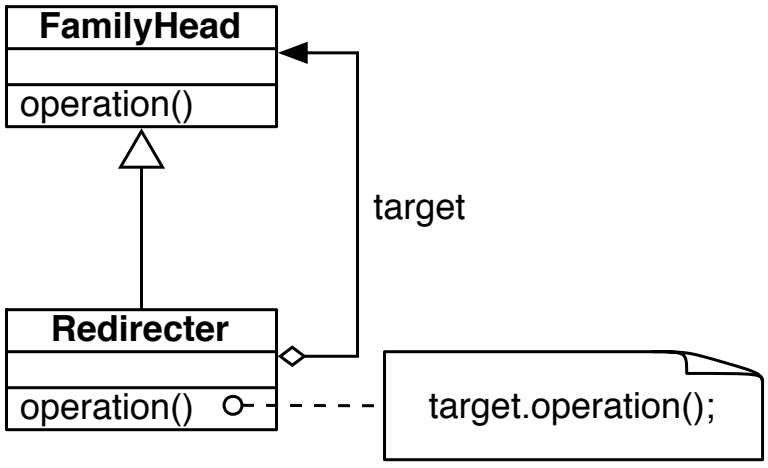
Each of the subclasses inherits the default behavior from EventHandler, although it can be overridden at any time. This behavior has each subclass using an instance of EventHandler as the next item to be asked to handle an event, in the case that the current object is not suitable. Common alterations to this behavior include checking to see what type of event has been passed in before making a decision regarding whether to handle it.

Applicability

Use RedirectInFamily when:

- An aggregate structure of related objects is expected to be composed at compile or runtime.
- Behavior should be decomposed to the various member objects.
- The structure of the aggregate objects is not known ahead of time.
- Polymorphic behavior is expected, but not enforced.

Structure



Participants

FamilyHead

Defines interface, contains a method to be possibly overridden

Redirecter

Uses interface of FamilyHead, redirects internal behavior back to an instance of FamilyHead to gain polymorphic behavior over an amorphous object structure

Collaborations

Redirecter relies on the class FamilyHead for an interface, and an instance of same for an object recursive implementation. The **Redirect** relationship is a critical part of this pattern, as it drives the concept of 'do the same as I was asked to do'. If a more general form is required, **DelegateInFamily** is appropriate.

Consequences

Redirecter is reliant on FamilyHead for its interface, but it falls to the entirety of the class hierarchy to provide the various implementations. Because of this, other classes in the hierarchy may exhibit unexpected behavior. If this is occurring, instead consider using the **RedirectInLimitedFamily** pattern, to restrict the possibilities to a manageable set.

Implementation

In C++:

```
class FamilyHead {
public:
        virtual void operation();
};

class Redirecter : public FamilyHead {
public:
        void operation();
        FamilyHead*     target;
};

void
Redirecter::operation() {
        // preconditional behavior
        target->operation();
        // postconditional behavior
}
```

Rho Calculus Definition

$$\frac{\begin{array}{l} Redirecter <: FamilyHead, \\ r : Redirecter, \\ fh : FamilyHead, \\ r.operation <_{\mu}^{-\cdot +} fh.operation, \end{array}}{\textbf{RedirectInFamily}(r, fh, operation)}$$

DelegateInLimitedFamily — Object Behavioral

Intent

When **DelegateInFamily** is too generalized, and it is necessary to pre-select a sub-tree of the class hierarchy for polymorphism.

Motivation

Static typing is a way of pre-selecting types from a well-defined pool, and forming more concrete notions of an object's type at runtime. Polymorphism is a technique for abstracting out typing information until runtime. Sometimes we need a balance of the two. This pattern and the related **RedirectInLimitedFamily** both weigh these opposing forces but for slightly different outcomes.

This pattern is concerned with *Delegation*, the more generalized form of computational sub-tasking. It is a specialization of **DelegateInFamily**.

Consider the driving example from **DelegateInFamily**. It really does not make a lot of sense to have a SliderBar UI control altering the value of a TextField. This design can be fine-tuned by adding the following classes and changes:

```
class TextInputControl : InputControl {
    TextWidget* textTarget;

    void userHasSetNewValue() {... textTarget->updateValue(myNewValue); ...}
};

class TextWidget : DisplayWidget {
    TextData    td;
};

class TextField : TextWidget {
    // as before
};
```

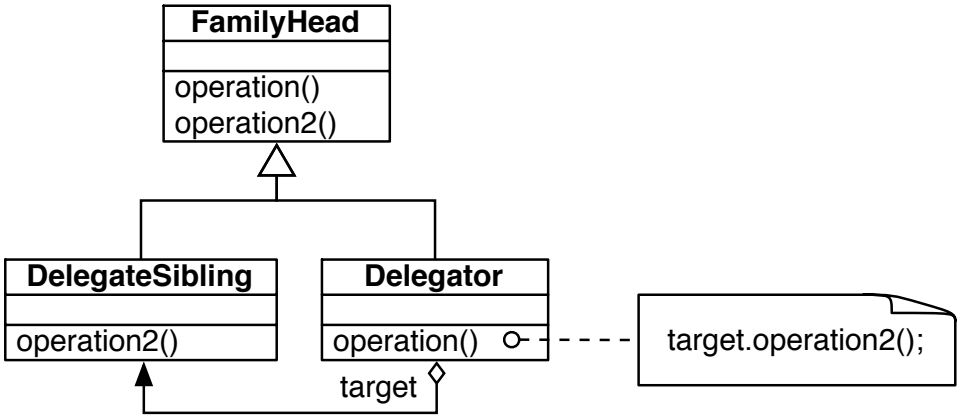
Now the TextInputControl is available for altering TextFields, but it will not be able to alter instances of BarGraph or other non-text display items. SliderBar and RotartKnob could similarly be limited to a class hierarchy based on a NumericWidget, for example.

Applicability

Use DelegateInLimitedFamily when:

- **DelegateInFamily** is the appropriate general pattern.
- Greater control over the possible types of objects is required.

Structure



Participants

FamilyHead

The base class for a polymorphic class cluster

Delegator

A subclass of FamilyHead

DelegateSibling

Another subclass of FamilyHead

target

A polymorphic instance of DelegateSibling that is contained by Delegator

operation

The calling site

operation2

The called subtask

Collaborations

In addition to the collaborations involved in **DirectInFamily**, this pattern adds the Delegate-Sibling class as the target for the operation implementation. By doing so, it limits the possible behavior to a subclass of the common ancestor class.

Consequences

This pattern differs from its cousin **DirectInFamily** in that a decision is made during design to limit the polymorphism statically to a sub-tree of the original class family. While this can show large benefits in containing complexity, it can also lead to issues later, if, for example, it is determined that the sibling class chosen is *too* limiting. While the syntactic change to the code is minimal, the addition of a broader set of possible classes and behaviors can have far reaching effects that need to be carefully considered.

Implementation

In C++:

```
class FamilyHead {
        void operation();
        void operation2();
};

class DelegateSibling : public FamilyHead {
        void operation2();
};

class Delegator : public FamilyHead {
        DelegateSibling*        target;
        void operation() { target->operation2(); };
};
```

Rho Calculus Definition

$$\frac{\begin{array}{l} Delegator <: FamilyHead, \\ Sibling <: FamilyHead, \\ Delegator \neq Sibling, \\ Delegator \not<: Sibling, \\ d : Delegator, \\ sib : Sibling, \\ d.operation <\!\!\sim_{\mu}^{-\cdot -} sib.operation2, \end{array}}{\textbf{DelegateInLimitedFamily}(d, sib, operation, operation2)}$$

RedirectInLimitedFamily — Object Behavioral

Intent

When **RedirectInFamily** is too generalized and it is necessary to pre-select a sub-tree of the class hierarchy for polymorphism.

Motivation

Static typing is a way of pre-selecting types from a well defined pool and forming more concrete notions of an object's type at runtime. Polymorphism is a technique for abstracting out typing information until runtime. Sometimes we need a balance of the two. This pattern and the related **DelegateInLimitedFamily** both weigh these opposing forces but for slightly different outcomes.

This pattern is, obviously, concerned with *redirection*, having some advance knowledge of the intimacies of the task at hand to be able to determine that similarly-named methods will be calling each other in a chain of subtasking.

In **RedirectInFamily**, the code provided a very general event handling system. It may be, however, that this open of a system is not appropriate for all cases. For instance, a slider bar is only going to respond to mouse events, and not, except in special circumstances, to keystrokes. Entire classes of events can therefore be eliminated at one time. Given the code in **RedirectInFamily**, make the following changes:

```
class MouseEventHandler : EventHandler {
    MouseEventHandler*  nextMEHandler;

    void handle(MouseEvent* me) { ...
        nextMEHandler->handle(me);
    ...};
};

class SliderBar : MouseEventHandler {
};
```

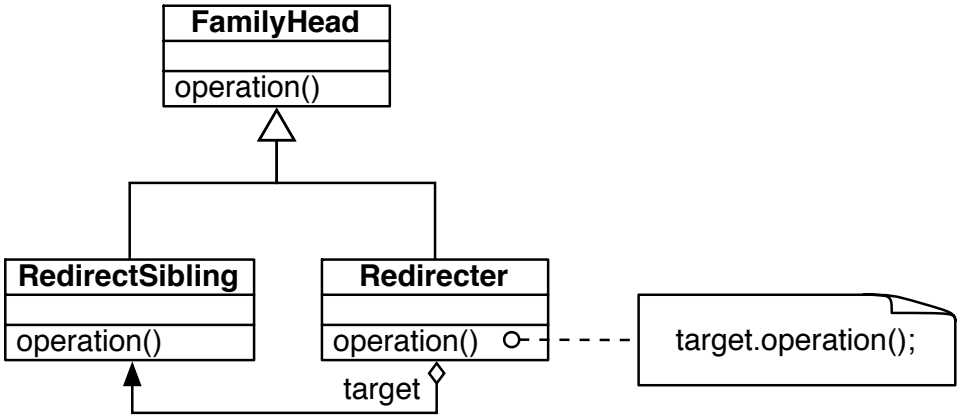
Now, SliderBars will only respond to MouseEvents, and they are guaranteed to only hand off mouse events to another possible handler of such events.

Applicability

Use RedirectInLimitedFamily when:

- **RedirectInFamily** is the appropriate general pattern.
- Greater control over the possible types of objects is required.

Structure



Participants

FamilyHead

Defines interface, contains a method possibly to be overridden, and is the base class for both Redirecter and RedirectSibling

Redirecter

Uses interface of FamilyHead; redirects internal behavior back to an instance of RedirectSibling to gain polymorphic behavior over an amorphous *but limited in scope* object structure

RedirectSibling

The head of a new class tree for polymorphic behavior

Collaborations

As with **RedirectInFamily**, Redirecter and FamilyHead are tied by interface and intent. The RedirectSibling provides a conceptual starting point for a particular type of implementations.

Consequences

This pattern differs from its cousin **RedirectInFamily** in that a decision is made during design to limit the polymorphism statically to a sub-tree of the original class family. While this can show large benefits in containing complexity, it can also lead to issues later, if, for example, it is determined that the sibling class chosen is *too* limiting. While the syntactic change to the code is minimal, the addition of a broader set of possible classes and behaviors can have far reaching effects that need to be carefully considered.

Implementation

In C++:

```
class FamilyHead {
public:
        virtual void operation();
};

class RedirecterSibling : public FamilyHead {
        void operation();
}

class Redirecter : public FamilyHead {
public:
        void operation();
        RedirecterSibling*      target;
};

void
Redirecter::operation() {
        // preconditional behavior
        target->operation();
        // postconditional behavior
}
```

Rho Calculus Definition

$$\frac{\begin{array}{l} Redirecter <: FamilyHead, \\ Sibling <: FamilyHead, \\ Redirecter \neq Sibling, \\ Redirecter \not<: Sibling, \\ r : Redirecter, \\ sib : Sibling, \\ r.operation <_{\mu}^{-,+} sib.operation, \end{array}}{\textbf{RedirectInLimitedFamily}(r, sib, operation)}$$

Appendix C

Intermediate Patterns Compositions

This Appendix contains design pattern definitions that are drawn from existing literature other than the Gang of Four text, and several nascent design patterns that may or may not be more appropriate in the EDP Catalog of Appendix B. As they are more well-defined, they will be placed appropriately. The well-defined patterns in this section are those that build on the EDPs and are composed of more than one EDP. These are then in turn the building blocks for establishing proper Gang of Four definitions.

FulfillMethod — Object Structural

Intent

To provide an implementation for a previously abstracted method, thereby fulfilling the contract of deferment.

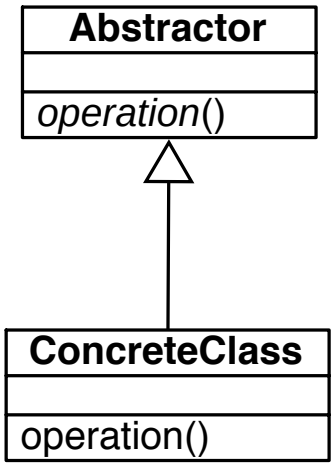
Motivation

Applicability

Use FulfillMethod when:

- Implementation of a method has been deferred using **AbstractInterface**.
- A subclass is ready to handle the functionality of the method as promised.

Structure



Participants

Abstractor

The class type that declares an interface for *operation*

ConcreteClass

The class type that defines a method body for *operation*, inherits from Abstractor

Collaborations

Abstractor defines an interface for a method, and ConcreteClass provides the implementation. ConcreteClass subclasses from Abstractor to gain the appropriate typing and interface.

Consequences

ConcreteClass relies on Abstractor for the interface to the implemented method. These two must be kept in sync, and there are times when a shuffling interface at the top of a class hierarchy can have far reaching effects to the subclasses. In cases where this is expected, a solution based on **Delegate** may be preferred. This allows the external interface to be independent of the implementation object. Alternately, if only one class/object is mandated, **Conglomeration** can be used to hand off the task to private methods internal to the type, thereby preserving the interface/implementation encapsulation.

Implementation

In C++:

```
class AbstractOperations {
public:
     virtual void operation() = 0;
};

class DefinedOperations : public AbstractOperations {
public:
     void operation();
};

void
DefinedOperations::operation() {
        // Perform the appropriate work
}
```

Rho Calculus Definition

$$\frac{\begin{array}{l}\textbf{AbstractInterface}(Abstractor, operation_n),\\ ConcreteClass <: Abstractor,\\ ConcreteClass \equiv [operation_i \Leftarrow {b_i}^{i \in 1..n-1, n+1..m}, operation_n \Leftarrow b_n]\end{array}}{\textbf{FulfillMethod}(Abstractor, ConcreteClass, operation_n)}$$

RetrieveNew — Object Management

Intent

Used when a new instance is desired, but the creation of the instance is left to another object. The returned object is guaranteed not to have any other references to it.

Motivation

Often, a newly created object needs to be a pristine object, with well-defined ownership. If the object can be referenced by a number of sources, then any one of them can request the destruction of that object, leaving the remainder of the references pointing to an invalid object. Alternatively, none of the referring objects may request its destruction, leading to a memory leak as the object becomes unreachable, but still resident. Garbage collection can help with this issue, but well-defined memory management starts with small decisions.

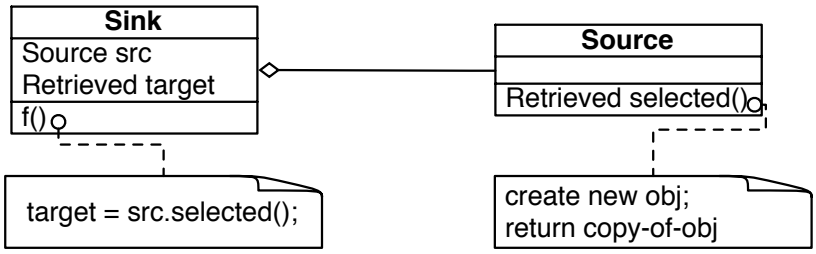
One such decision is to make sure that only the recipient of a **Retrieve** has access to the object being retrieved. This can be done most transparently by returning a copy of an object, since the copy will create a fresh instance. Other schemes involving passing back a method-local reference that is destroyed on method exit can be used, but they are more prone to errors during later maintenance, without a strong sense of the ownership issues.

Applicability

Use Retrieve when:

- A remote object provides a value object that is required for local computation and is provided by an exposed field object.
- The object returned needs to be a fresh copy without other references to it.

Structure



Participants

Source

The object (or class) type that contains *selected*.

Sink

The object (or class) type that includes the item, *target*, to be given a new value.

Retrieved

The type of the value to be updated, and the value that is returned.

target

The field that is given a new value.

selected

The method or field that produces the new value.

Collaborations

There are only three objects in this collaboration, and they merely play the parts of the request originator, the request fulfiller, and the passed object.

Consequences

The separation between object ownership and object creation has some advantages. For example, the type of the created object can be determined in a flexible manner, possibly using polymorphism, such as in an **AbstractFactory**. It means, however, that the recipient object is solely responsible for the lifetime of the created object. If references to the created object are handed out, a good reference management policy must be in place, unless garbage collection is allowed.

Implementation

In C++:

```
class Source {
public:
        Retrieved giveMeAValue() { Retrieved ret;  return ret; };
};

class Sink {
        Retrieved target;
        Source srcobj;
public:
        void operation() { target = srcobj.giveMeAValue(); }
};
```

Rho Calculus Definition

$$\frac{\begin{array}{c} o'.s' \overset{n}{\rightsquigarrow} x \\ \textbf{CreateObject}(X, o'.s'.x) \\ \textbf{Retrieve}(o.s, o'.s', x) \end{array}}{\textbf{RetrieveNew}(Sink, object, target, Retrieved, Source, object', selected)}$$

RetrieveShared — Object Management

Intent

To obtain a reference to a shared object, without holding ownership of that object. Ubiquitous in object-oriented programming.

Motivation

Objects are conceptual entities, and any particular instance may hold state that many be of interest to many other objects. Consider a printer queue, for example. Many applications are going to want to have access to the queue, but there is no reason for each application to have its own queue. In fact, this would be certain to lead to eventual resource collisions as differing queues competed. Instead, a queue can be shared among many applications simultaneously. Requests from disparate objects can be handled in the order best suited by the queue. The queue is a shared resource.

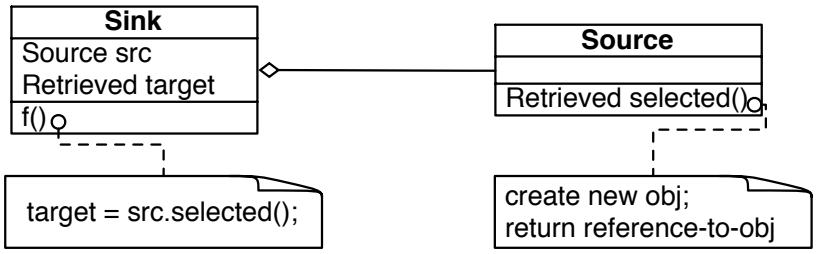
This pattern is so common that many languages have some level of support for it directly. C++, for example, has the proposed `shared_ptr` construct which is now being considered for adoption for the C++x0 standard (Dimov et al., 2003).

Applicability

Use RetrieveShared when:

- A remote object provides a value object that is required for local computation and is either:
  - provided by a method call's return value or,
  - provided by an exposed field object.
- The object should be shared with other objects, and not considered a private resource.

Structure



Participants

Source

The object tasked with handing off the retrieved object. It serves as the initial source, creating the object.

Sink

The object which requests the retrieved object.

target

The final repository for the retrieved object.

retrieved

The retrieved object - in this scenario, it is a reference to an object, and the reference is not guaranteed to be associated with ownership in any way.

Collaborations

There are only three objects in this collaboration, and they merely play the parts of the request originator, the request fulfiller, and the passed object.

Consequences

The retrieved object gains a reference count, in garbage collection parlance, by using this pattern, but does not limit the generation of new references. This can lead to memory leaks fairly readily, so be certain that there is a well-defined ownership policy elsewhere. If an object needs to be retrieved with assured uniqueness, or wishes to be the primary owner of a newly allocated shared resource, consider **RetrieveNew** instead.

Implementation

In C++:

```
class Source {
public:
```

```
        Retrieved* giveMeAValue() { return new Retrieved();} ;
};

class Sink {
        Retrieved* target;
        Source srcobj;
public:
        void operation() { target = srcobj.giveMeAValue(); }
};
```

Rho Calculus Definition

$$\frac{o'.s' \overset{v}{\rightsquigarrow} x \quad \textbf{CreateObject}(X, o'.s'.x) \quad \textbf{Retrieve}(o.s, o'.s', x)}{\textbf{RetrieveShared}(Sink, object, target, Retrieved, Source, object', selected)}$$

Objectifier

The full definition and discussion of this pattern can be found in (Zimmer, 1995). Please refer to that text for the base document.

Rho Calculus Definition

$$\frac{\begin{array}{l} ObjectifierBase : [l_i : {B_i}^{i \in 1 \ldots n}], \\ Client : [ref : Objectifier], \\ Client.someMethod <_{\mu} Client.ref.l_i, \\ \textbf{FulfillMethod}(ObjectifierBase, ConcreteObjectifier, l_i)^{i \in 1 \ldots n}, \end{array}}{\textbf{Objectifier}(ObjectifierBase, ConcreteObjectifier, Client)}$$

Object Recursion

The full definition and discussion of this pattern can be found in (Woolf, 1998b). Please refer to that text for the base document.

Rho Calculus Definition

$$\frac{\begin{array}{c}\textbf{Objectifier}(Handler, {Recurser_i}^{i\in 1\ldots m}, Initiator),\\ \textbf{Objectifier}(Handler, {Terminator_j}^{j\in 1\ldots n}, Initiator),\\ init.someMethod <_\mu obj.handleRequest,\\ init : Initiator,\\ obj : Handler,\\ \textbf{RedirectInFamily}(Recurser, Handler, handleRequest),\\ !\textbf{RedirectInFamily}(Terminator, Handler, handleRequest)\end{array}}{\textbf{ObjectRecursion}(obj, {Recurser_i}^{i\in 1\ldots m}, {Terminator_j}^{j\in 1\ldots n}, init)}$$

Appendix D

GOF Patterns Compositions

The Gang of Four patterns are defined in this Appendix, following the principles described in Chapter 6. These are preliminary definitions, and it is expected that they will be revised according to community consensus as the discussion of the EDPs and their established definitions progresses.

The annotated UML diagrams show the relationships between the conceptually important sub-patterns found during the training described in Section 10.1.1. Concepts considered important, even critical, but that are currently not defined in SPQR for searching are parenthesized. **Collection** and **Iteration** are two such examples. I present the concepts here as works in progress to facilitate discussion and to illustrate the density of concepts embedded and intertwined in the Gang of Four catalog. Those patterns that rely fundamentally on such undefined patterns include: **Prototype**, **Composite**, **Facade**, **Flyweight**, **Iterator**, **Memento**, and **Observer**. I have every confidence that the required concepts are definable within ρ-calculus, but they fall outside the method-based EDPs on which I concentrated my research to date.

Several other patterns have definitions provided here that I believe to be necessary, but not sufficient, for reliable detection by SPQR. These include **AdapterObject**, **Bridge**, **Mediator**, **State**, and **Strategy**. Each has a set of constraints that are not clearly mapped to the current EDP catalog. I find it not at all a coincidence that the last three of these are listed in the Gang of Four as behavioral patterns. The first two are structural patterns, but the characteristics are so general, that I expect they will prove to be so ubiquitous within most code, as to prove the current definitions in need of refinement. I suspect that the further research in the data-related elements of ρ-calculus will produce the necessary pieces, but at the moment, these are presented to facilitate discussion.

The remainder of the Gang of Four patterns are, in my opinion, well formed and defined using the EDP catalog and ρ-calculus. Time and further testing with SPQR will prove the truth of that.

The final section of the Gang of Four text (Gamma et al., 1995, p. 358) presents a quote from Christopher Alexander that they use to provide insight to what is "good design" and I can think of no more fitting description of their own patterns when viewed as dovetailed and intertwined examples of the EDPs:

> It is possible to make buildings by stringing together patterns, in a rather loose way. A building made like this, is an assembly of patterns. Is it not dense. It is not profound. But it is also possible to put patterns together in such a way that many patterns overlap in the same physical space: the building is very dense; it has many meanings captured in a small space; and through this density, it becomes profound. (Alexander et al., 1977, p. *xli*)

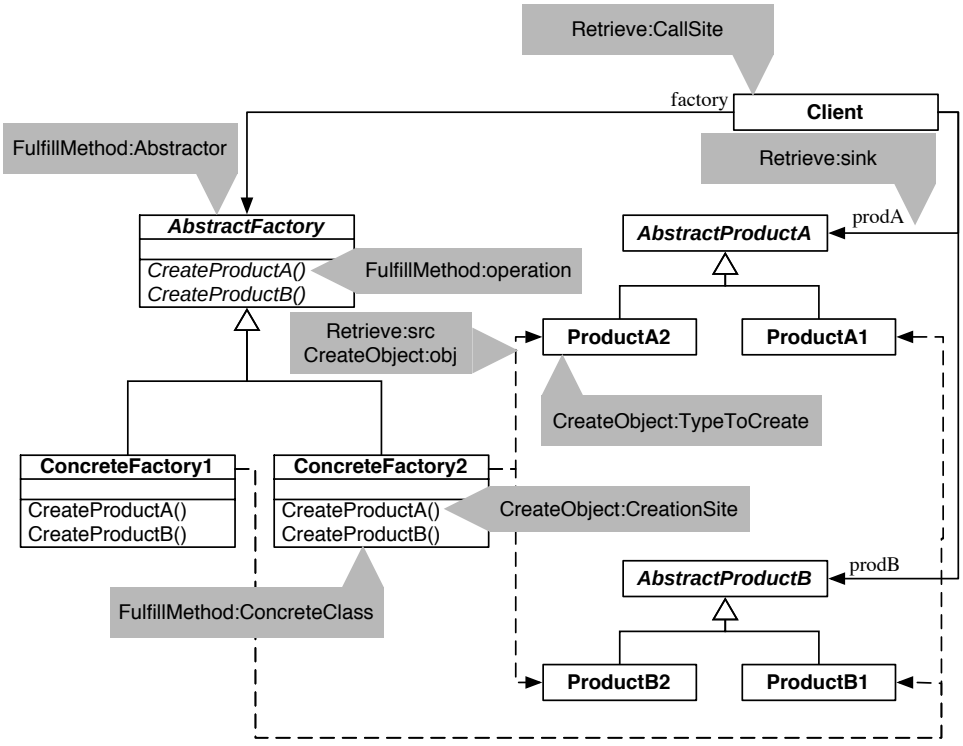
D.1 Creational Patterns

The creational patterns are all concerned with object creation and management. As would be expected, **CreateObject** features heavily in their definitions, as do the **Retrieve** related patterns. How these relate to each other and to the other EDPs provide the necessary distinctions and create a wide array of differing behaviors.

D.1.1 Abstract Factory

The first creational pattern, **AbstractFactory**, is a perfect example of how simple concepts can intertwine in complex ways to produce an interesting concept. Only three simple patterns are needed to capture most of the semantics of this pattern: **FulfillMethod**, **Retrieve**, and **CreateObject**. It is the manner in which they connect that provides the rich behavior.

The diagram shows just one interconnection, between AbstractProductA and ConcreteFactory2 through the CreateProductA method and the ProductA2 class. Similar connections exist for the other three Product classes.

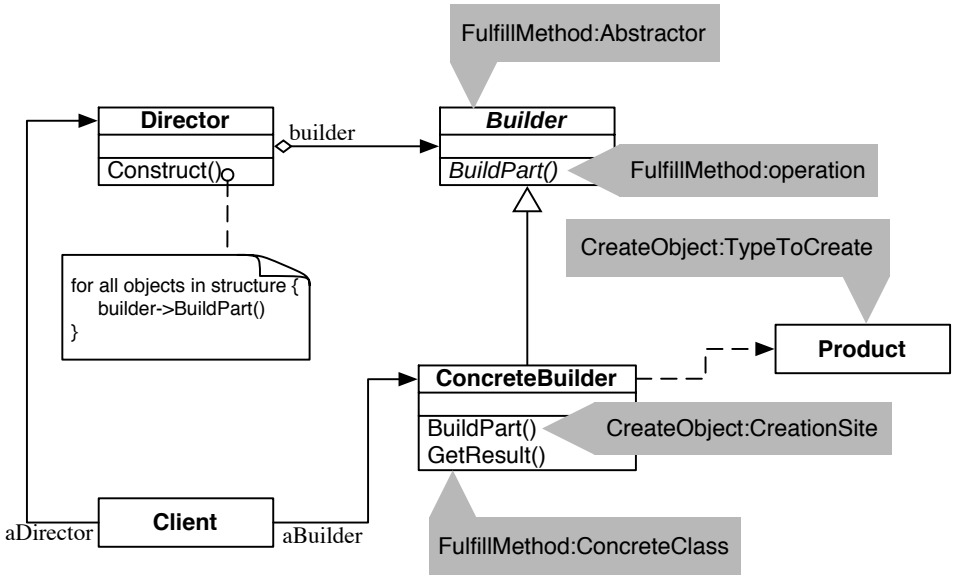


$Client.prodA : AbstractProductA$
$ProductA2 <: AbstractProductA$
Retrieve$(Client.someMethod, Client.prodA,$
$ConcreteFactory2.CreateProductA.rtnVal)$
CreateObject$(ConcreteFactory2.CreateProductA, ProductA2,$
$ConcreteFactory2.CreateProductA.rtnVal)$
FulfillMethod$(AbstractFactory, ConcreteFactory2, CreateProductA)$

AbstractFactory$(AbstractFactory, ConcreteFactory2, CreateProductA,$
$AbstractProductA, ProductA2)$

D.1.2 Builder

Builder contains quite a bit of behavior that is not easily captured by the current EDPs. For example, the requirement that the Client construct a Director instance, and give the Director an instance of a ConcreteBuilder to work with. While it is possible to formalize this requirement, as shown in the accompanying equational definition, it is certainly not easy to work with on a conceptual level.



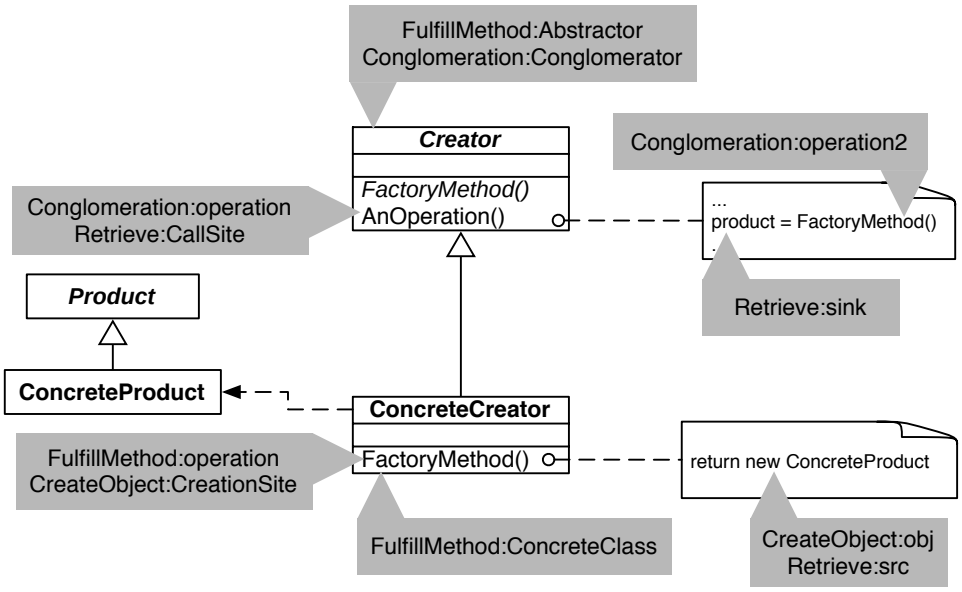
$Client.aDirector : ConcreteBuilder$
$\|Director.builder <_\kappa Client.aDirector\|_{Director.someMethod}$
$Director.Construct <_\mu Director.builder.BuildPart$
FulfillMethod$(Builder, ConcreteBuilder, BuildPart)$
CreateObject$(ConcreteBuilder.BuildPart, Product, ConcreteBuilder.somePart)$
$\|ConcreteBuilder.finalProduct <_\kappa ConcreteBuilder.somePart\|_{ConcreteBuidler.BuildPart}$
$ConcreteBuilder.GetResult \rightsquigarrow ConcreteBuilder.finalProduct$

Builder$(Director, Construct, Builder, ConcreteBuilder, BuildPart)$

D.1.3 Factory Method

This pattern defers creation of an actual object to a subclass, but provides a firm interface and base method for creating the object and using it internally. It weaves the EDPs **CreateObject**, **Retrieve**, **FulfillMethod** and **Conglomeration** all within one type hierarchy.

FactoryMethod is also the first of the Gang of Four to use another Gang of Four pattern in its definition. The first two clauses of the definition below could be replaced with an equivalent **TemplateMethod**. In the Gang of Four discussion of **FactoryMethod**, this is mentioned as the norm (Gamma et al., 1995, p. 116).

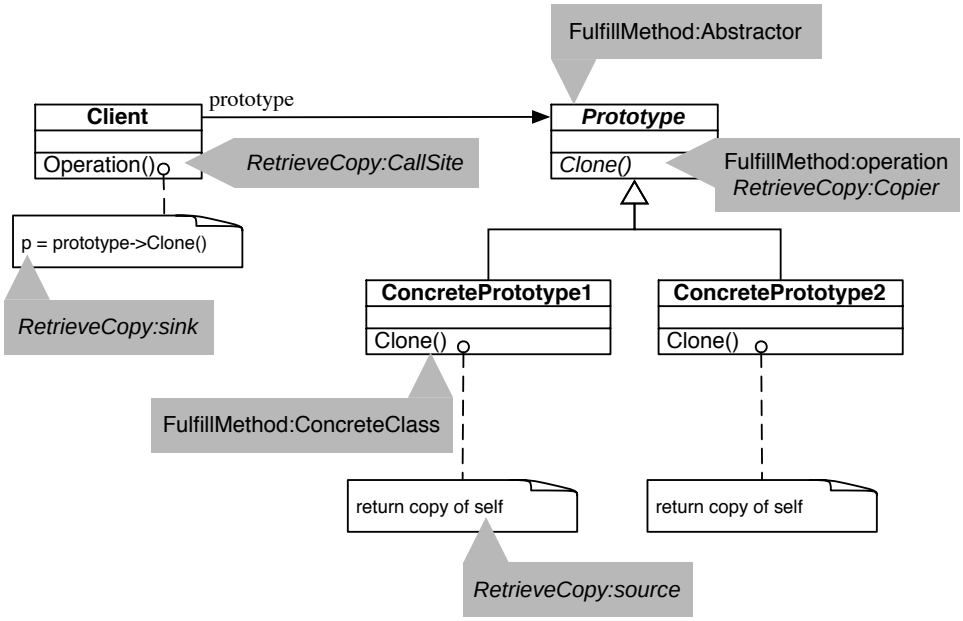


Conglomeration$(Creator, AnOperation, FactoryMethod)$
FulfillMethod$(Creator, ConcreteCreator, FactoryMethod)$
$Creator.product : Product$
$ConcreteCreator.FactoryMethod \rightsquigarrow retVal$
Retrieve$(Creator.AnOperation, Creator.product, retVal)$
$ConcreteProduct <: Product$
CreateObject$(ConcreteCreator.FactoryMethod, ConcreteProduct, retVal)$

FactoryMethod$(Creator, ConcreteCreator, AnOperation, FactoryMethod,$
$Product, ConcreteProduct)$

D.1.4 Prototype

To date, while the **Retrieve** related methods capture object management semantics when it comes to initial creation, they do not define retrieval of a *copy* of an object. I do not see this as being difficult to detect or to define, as the use of an assignment operator, or other equivalent language construct, can be used to flag this situation. Other than that, **Prototype** is a straightforward combination of two patterns.

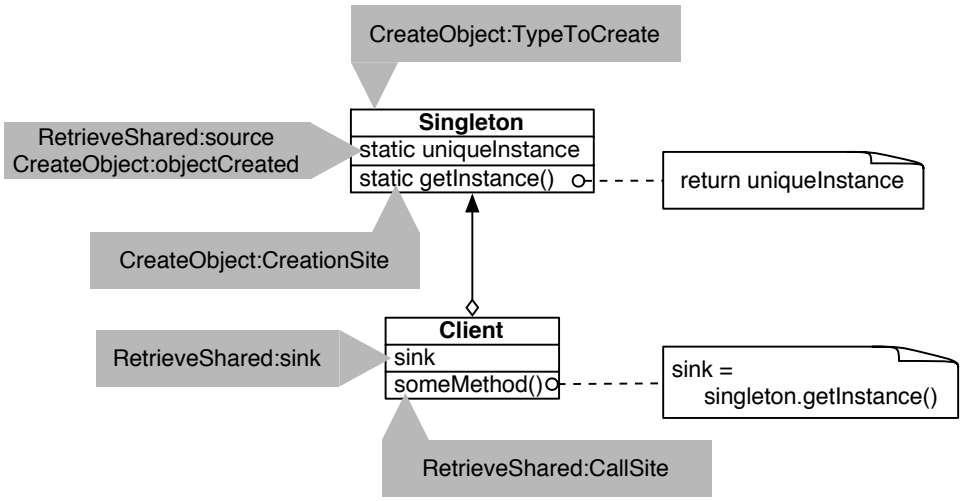


$$\frac{\textbf{FulfillMethod}(Prototype, ConcretePrototype1, Clone)\ (\textbf{RetrieveCopy}(Client.operation, Prototype.Clone, p, prototype.self))}{\textbf{Prototype}(Client, Operation, p, Prototype, ConcretePrototype1, Clone)}$$

D.1.5 Singleton

Singleton has a **CreateObject** that is creating an object of the type holding the new instance, and a mechanism for sharing that common resource with others through **RetrieveShared**.

Note that this definition makes no mention of whether other mechanisms for creating new instances, such as public constructors, are available. While such a requirement could be created, such public or private indicators are not currently used in SPQR.



$client.method <_\mu singleton.gettor$,
$singletonClass = Class(Singleton)$,
$singletonClass.instance : Singleton$
CreateObject($singletonClass.gettor, Singleton, singletonClass.instance$)
RetrieveShared($client.method, singletonClass.gettor, singletonClass.instance$)

Singleton($Singleton, singletonClass.gettor, singletonClass.instance$)

D.2 Structural Patterns

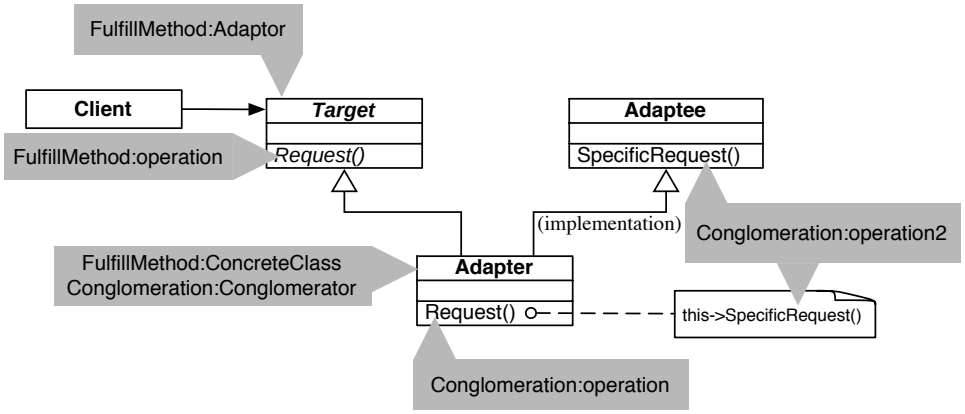
It was my assumption at the beginning of this research that the structural patterns would be the simplest to analyze and produce necessary and sufficient definitions for. Surprisingly, they have been neither more simple nor problematic than the other major divisions within the Gang of Four patterns catalog.

D.2.1 Adapter

The **Adapter** cluster is actually three versions of the same basic principle, and it may be possible to unify two of them. The class adapter and superclass adapter are two alternate forms of the same concept: to use a superclass' implementation of a method as the target for adaption. The difference lies in the calling semantics. The class adapter relies on the natural *Self* construct of the language, while the superclass adapter performs a direct call to the method body via a *super* mechanism. The latter is more fragile with respect to code hierarchies, but direct, and disallows the overriding of the method that is possible with the class adapter. In such a situation, there is no adherence to the idea of an **Adapter**, but instead just a use of **Conglomeration**.

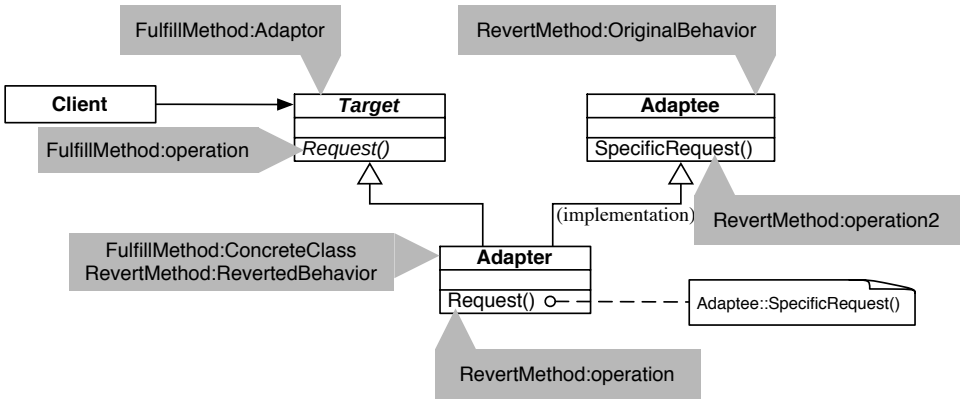
This difference in calling features and policy enforcement may be a sign that **Conglomeration**, where the target method is inherited from a superclass, may be unifiable with **RevertMethod** in some sense. Indeed, the underlying implementation of `gcc` gives weight to this argument, as it creates precisely the same structure for accessing the SpecificRequest method in both code cases. This can be observed by the results of the training procedure in Figure 10.2, where **RevertMethod** is found in both forms of the pattern. While `gcc` is doing so for reasons of efficiency in method lookup, it supports the concept that a call to a non-overridden method and a call to that method directly to bypass an overridden method may have more in common in intent than at first glance.

Adapter:Class



$$\frac{\begin{array}{l} Adapter <: Adaptee \\ SpecificRequest \in \mathbf{meth}(Adaptee) \\ \mathbf{Conglomeration}(Adapter, Request, SpecificRequest) \\ \mathbf{FulfillMethod}(Target, Adapter, Request) \end{array}}{\mathbf{AdapterClass}(Adapter, Target, Request, Adaptee, SpecificRequest)}$$

Adapter:Superclass

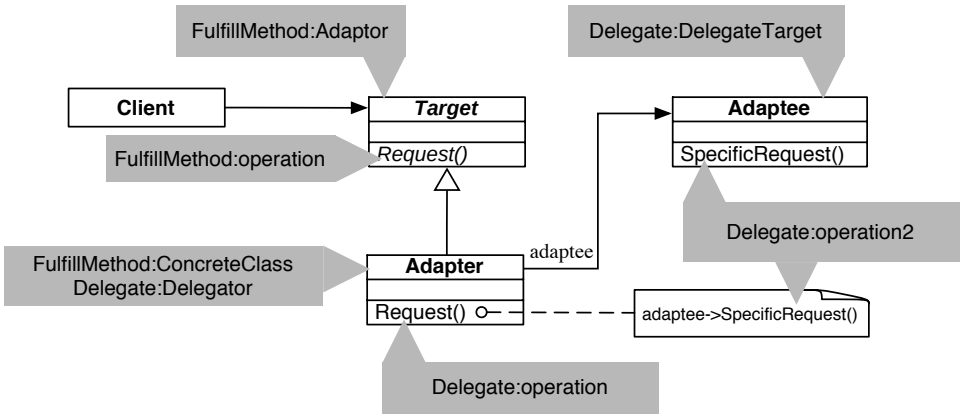


$$\frac{\begin{array}{l} \mathbf{RevertMethod}(Adapter, Adaptee, Request, SpecificRequest) \\ \mathbf{FulfillMethod}(Target, Adapter, Request) \end{array}}{\mathbf{AdapterSuperclass}(Adapter, Target, Request, Adaptee, SpecificRequest)}$$

The object adapter, however, is a problematic pattern. It is even more generalized than the class adapter, in that it loses any type relationship between the Adapter and the Adaptee. For this reason I suspect that this is a pattern that will require metrics to adequately refine the definition to eliminate false positives. One such metric would be to establish for a class that appears to be an Adapter, how many methods conform to the pattern. If there is a high percentage of non-constructor/destructor methods that do, then the probability of the class or object conforming to this pattern increases, as it reinforces that the class is designed for the specific task of adaption. If the methods that fit this characteristic

in turn all trigger methods from one object, then that is further evidence of the intent of the class. I present a definition that I feel is too broad, but is a starting point for discussion.

Adapter:Object

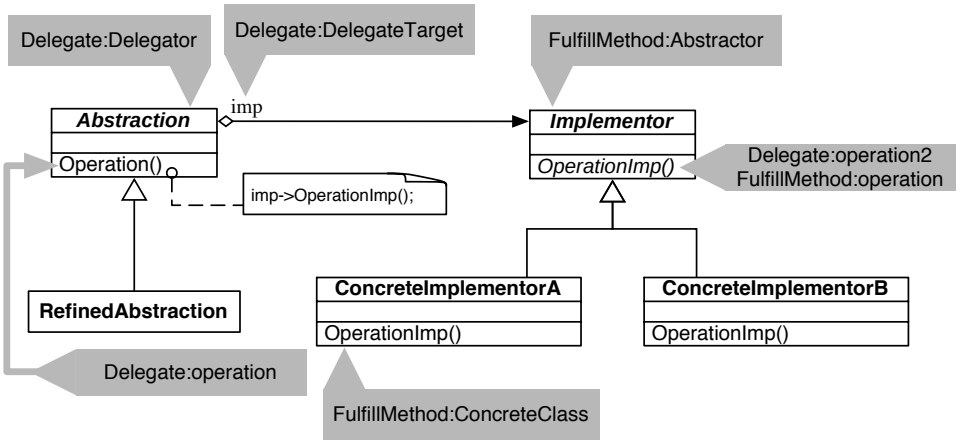


Delegate$(Adapter, Adaptee, Request, SpecificRequest)$
FulfillMethod$(Target, Adapter, Request)$

AdapterObject$(Adapter, Target, Request, Adaptee, SpecificRequest)$

D.2.2 Bridge

Unfortunately, **Bridge** offers little in the way of explicit structural or behavioral details to distinguish it from other polymorphic structures. It is my speculation that this pattern will prove to be ubiquitous in systems, to the point of being nearly without utility for comprehension of large-scale systems. There may be metric-based techniques, however, to help refine this definition.



$Abstraction.imp : Implementor$
Delegate$(Abstraction, Abstraction.imp, Operation, OperationImp)$
FulfillMethod$(Implementor, ConcreteImplementationA, OperationImp)$

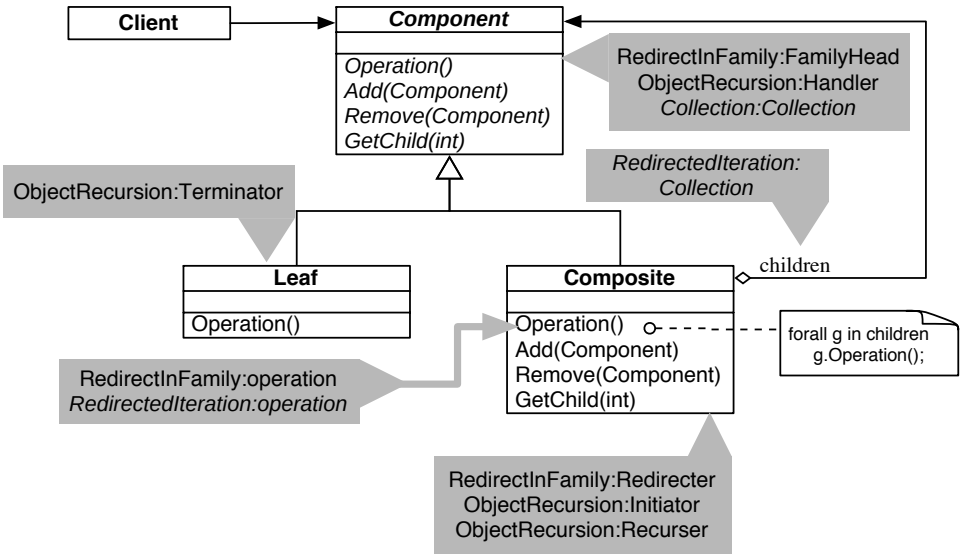
Bridge$(Abstraction, Operation, imp, Implementor, ConcreteImplementorA, OperationImp)$

D.2.3 Composite

Composite shares much in common with patterns such as **Decorator**, **Proxy**, and **Interpreter**. The details are the discerning factors. For example, **Composite** rather inherently contains the concept of a dynamic collection of objects, as defined in a **Collection**, and **Iteration** over that collection.

I introduce here the undefined **RedirectedIteration** pattern, which refines the **Iteration** concept to include a **Redirect** relationship via dotright similarity between the enclosing method and the target method for the collection members. **ForAll** or **ForAllRedirect** may be a better candidate name for this pattern.

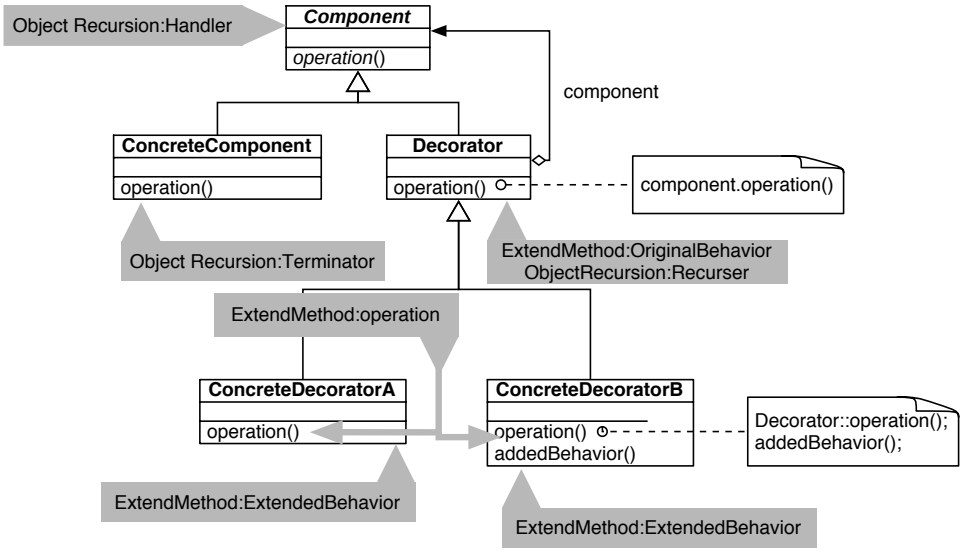
The intertwining of **RedirectedIteration** and **RedirectInFamily** may prove to be the distinguishing relationship of this pattern.



$$\frac{\begin{array}{l}\textbf{FulfillsMethod}(Component, Leaf, Operation)\\ (\textbf{Collection}(Composite, children, Component))\\ (\textbf{RedirectedIteration}(Composite, children, Operation))\\ \textbf{RedirectInFamily}(Composite, Component, Operation)\\ \textbf{ObjectRecursion}(Component, Composite, Leaf, Composite)\end{array}}{\textbf{Composite}(Component, Composite, Leaf, Operation)}$$

D.2.4 Decorator

Decorator is similar in form to **Composite**, except that it lacks the **Collection** semantics, and therefore the **RedirectedIteration**, and instead substitutes **ExtendMethod** as the primary conceptual basis.



$$\frac{\begin{array}{c}\mathbf{ObjectRecursion}(Component, {Decorator_i}^{i\in 1\ldots m}, {ConcreteComponent_j}^{j\in 1\ldots n}, \mathbf{any}),\\ \mathbf{ExtendMethod}(Decorator, {ConcreteDecoratorB_k}^{k\in 1\ldots o}, {operation_k}^{k\in 1\ldots o}),\\ !\mathbf{ExtendMethod}(Decorator, {ConcreteDecoratorA_l}^{l\in 1\ldots p}, {operation_l}^{l\in 1\ldots p})\end{array}}{\begin{array}{l}\mathbf{Decorator}(Component, {Decorator_i}^{i\in 1\ldots m}, {ConcreteComponent_j}^{j\in 1\ldots n},\\ \quad {ConcreteDecoratorB_k}^{k\in 1\ldots o}, {ConcreteDecoratorA_l}^{l\in 1\ldots p},\\ \quad {operation_k}^{k\in 1\ldots o+p})\end{array}}$$

D.2.5 Facade

Facade is an interesting pattern to include in the Structural Patterns block, since it has so very little structure to work with from a formal basis. While the underlying principle is structural, there are no relationships between types or type families to use as definition launching points, and indeed there is not even a good working definition of a relationship between the methods defined in the Facade class and the methods it utilizes. The closest that can be produced is that, if one considers the case where a Facade class hides exactly one other object or class, it is equivalent to an object adapter. One possible definition may be that a **Facade** is when one class acts as a plurality of **AdapterObject** instances.

There is, however, a metrics-based approach that may provide the information needed for SPQR. POML contains the information needed to produce sets of caller/callee relationships from the $<_\mu$ reliance operators. Define a set of callers S for a class C such that:

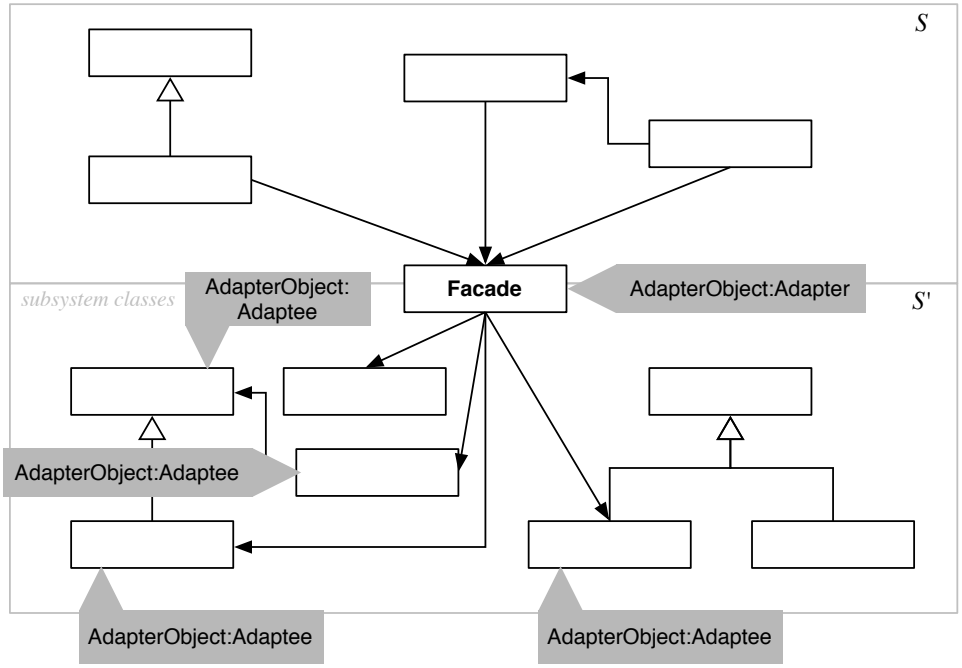
$$S = \{lhs\} : \forall m \in \mathbf{meth}(C)\{lhs <_\mu C.m\}$$

Define another set of callees S' is defined for C as:

$$S' = \{rhs\} : \forall m \in \mathbf{meth}(C)\{C.m <_{\mu} rhs\}$$

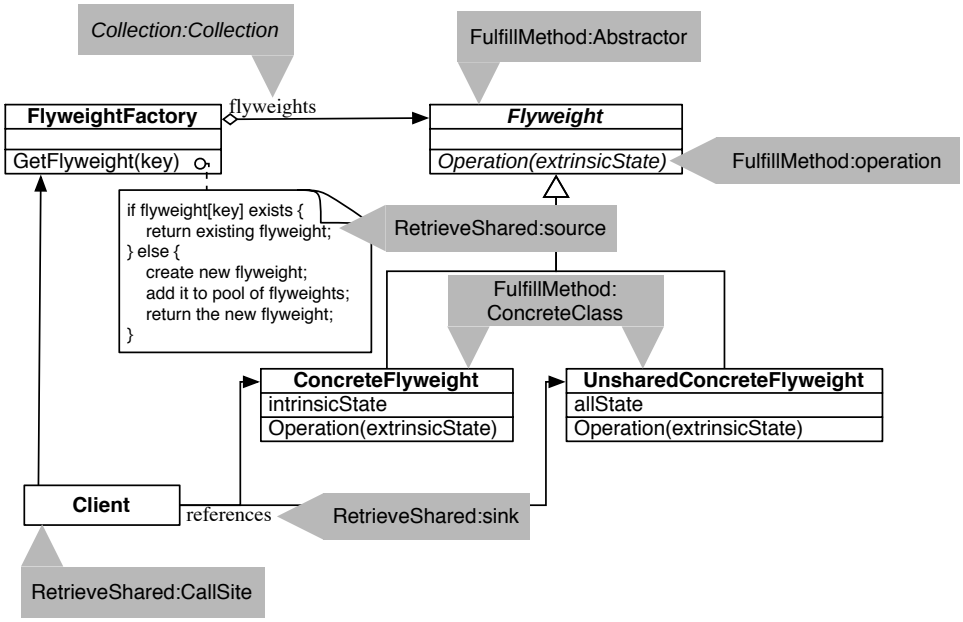
The set $S \cap S'$ will be empty, or nearly so, for a class acting as a **Facade**. This may be a simple way to capture the nature of a 'gateway' class and have it be extensible to other patterns such as the **Layers** architectural pattern, which essentially stacks a number of **Facade** instances into one concept (Buschmann et al., 1996).

The Gang of Four text supports this approach, as it states in the Related Patterns discussion for **Mediator**: "[Facade's] protocol is unidirectional; that is, Facade objects make requests of the subsystem classes but not vice versa." (Gamma et al., 1995, p. 282)



D.2.6 Flyweight

Flyweight is a considerably complex pattern, weaving aspects of object creation, state modification, and resource management into one concept. It is another pattern where **Collection** becomes important, and further illustrates the need for a solid definition of such semantics.

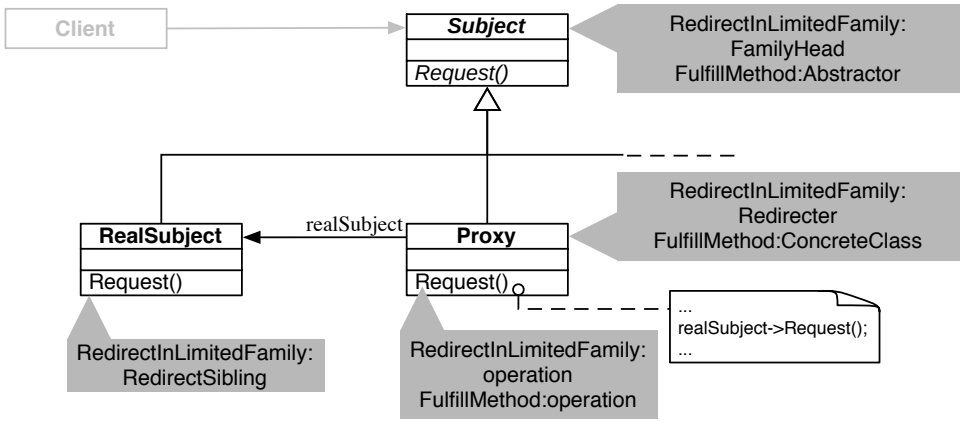


FulfillMethod$(Flyweight, ConcreteFlyweight, Operation)$
RetrieveShared$(Client.someMethod, Client.references, FlyweightFactory.fw)$
(**CollectionElement**$(FlyweightFactory.fw, FlyweightFactory.flyweights)$)
(**Collection**$(FlyweightFactory.flyweights)$)

Flyweight$(FlyweightFactory, Flyweight, ConcreteFlyweight, Client)$

D.2.7 Proxy

As with **Composite**, **Proxy** limits access to a an object or object structure, hiding details from the clients. The use of **RedirectInLimitedFamily** here is the important characteristic of this pattern.



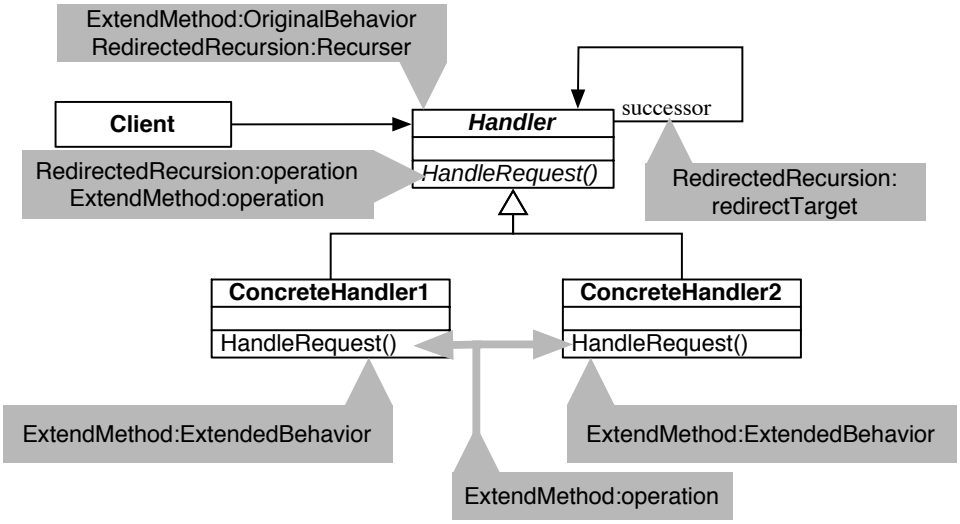
FulfillMethod$(Subject, Proxy, Request)$
FulfillMethod$(Subject, RealSubject, Request)$
RedirectInLimitedFamily$(Proxy, RealSubject, Subject, Request)$

Proxy$(Subject, Proxy, RealSubject, Request)$

D.3 Behavioral Patterns

In general, the Behavioral patterns should be the most difficult to detect, given the current state of the EDP catalog being primarily based on method to method reliances in the form of $<_{\mu}$. It turns out, however, that the expanded axes of discriminators introduced in Chapter 5, such as similarity between types, objects, and methods, produces a set of EDPs that can capture an interesting enough set of behaviors between methods that some rather unique constructs of multiple EDPs can give at least primary definitions for a number of the patterns in this block.

D.3.1 Chain of Responsibility

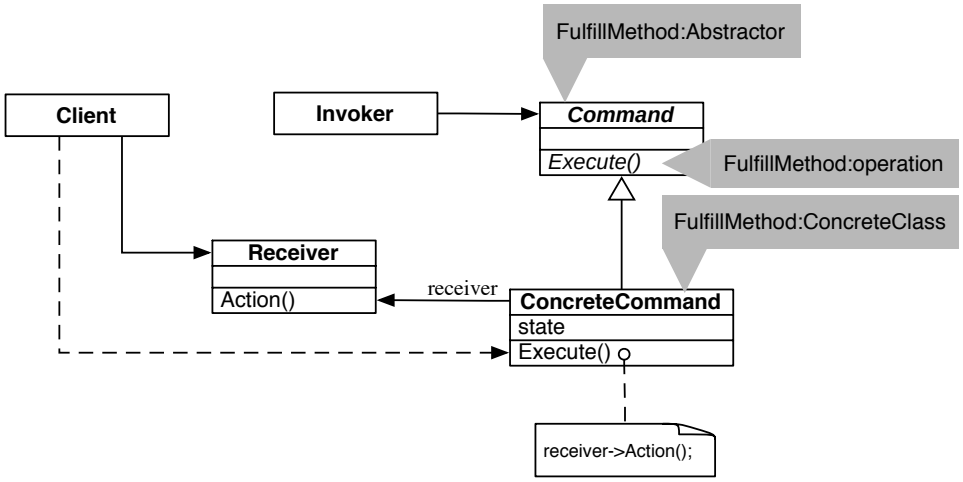


ChainofResponsibility is another pattern that has much in common with **Composite**, **Proxy** and **Decorator**. In fact, if you inspect the EDP derived definition of **ChainofResponsibility** against **Decorator**, it becomes apparent that the salient difference is that the **RedirectedRecursion** of **ChainofResponsibility** replaces the **ObjectRecursion** of **Decorator**. If one of the variants on **ChainofResponsibility** is used such that the ConcreteHandlers define the successor explicitly (Gamma et al., 1995, p. 225), then **ObjectRecursion** will be found here as well, and the only difference then becomes one of where in the pattern that **ObjectRecursion** occurs. It is an interesting exercise in minimalist conceptual changes resulting in greatly different intents.

$$\frac{\textbf{RedirectedRecursion}(Handler, successor, handle)\quad \textbf{ExtendMethod}(ConcreteHandler, Handler, handle)}{\textbf{ChainOfResponsibility}(Handler, ConcreteHandler, HandleRequest)}$$

D.3.2 Command

With **Command**, we run into the first example of why the non-$<_\mu$forms of the reliance operators should prove to be a rich research area for defining new EDPs and adding to the conceptual catalog. The interactions between these objects are almost strictly done via data reliances.



$ConcreteCommand.receiver <_\kappa Client$
FulfillMethod$(Command, ConcreteCommand, Execute)$
$ConcreteCommand.Execute <_\mu^{-\circ} ConcreteCommand.receiver.Action$
$Invoker.someMethod <_\mu Invoker.cmd.Execute$

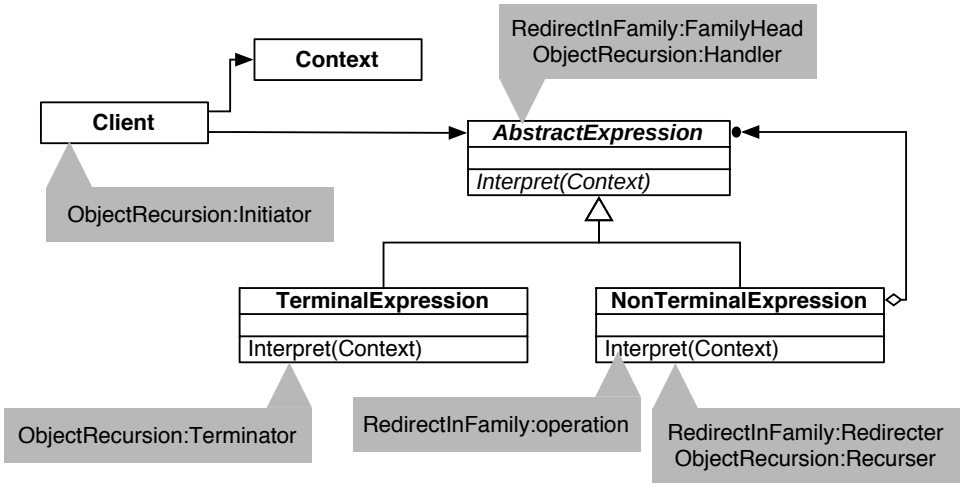
Command$(Command, ConcreteCommand, Execute, Receiver, Action, Client, Invoker)$

D.3.3 Interpreter

Interpreter has much in common with **Composite**. In fact, they are in many ways indistinguishable. Even the Gang of Four text recognizes this: "Considered in its most general form (i.e., an operation distributed over a class hierarchy based on the Composite pattern), nearly every use of the Composite pattern will also contain the Interpreter pattern." (Gamma et al., 1995, p. 255)

The text continues: "But the Interpreter pattern should be reserved for those cases in which you want to think of the class hierarchy as defining a language." There is no structural or behavioral difference between the two patterns, only one of vague intent. Perhaps the best approach is to look at the Context parameter to the Interpret method. The same Context is required from top to bottom while traversing the abstract syntax tree. This has **Interpreter** using an instance of **Composite**, with added requirements regarding the parameter passing. While it is entirely possible that other instances of **Composite** may also fulfill this requirement, it becomes a matter of verification on the part of the engineer to determine

which pattern most closely fits the found instance.



Composite$(AbstractExpression, NonTerminalExpression, TerminalExpression, Interpret)$
$Client.context : Context$
$NonTerminalExpression.Interpret\{\!|context \overset{n}{\leftarrow} Client.context|\!\}$
$NonTerminalExpression.Interpret.nextExpr.Interpret(context)$

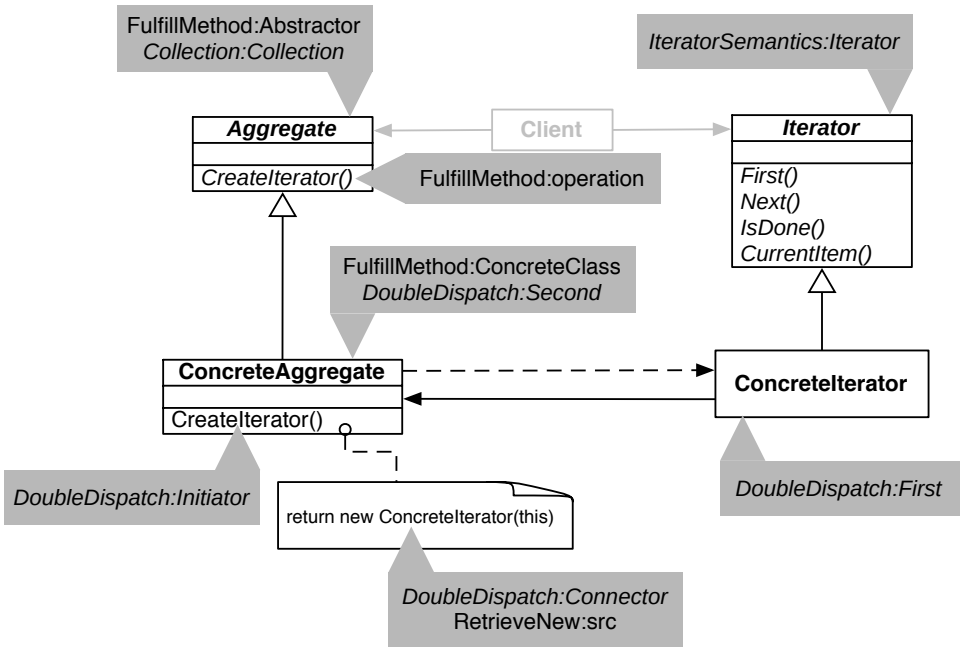
Interpreter$(AbstractExpression, NonTerminalExpression, TerminalExpression, Interpret, Context)$

D.3.4 Iterator

This definition nearly requires the core definition in the first place to adequately be detected. As with **Collection**, an iterator is a gathering of methods with particular behaviors with relation to a collection. It may be simplest in many languages to rely on the semantics of the language constructs that adhere to the concept. An iterator, in the general sense, has the proper semantics to provide access into a collection and traverse that collection. This is difficult at this time to detect statically, but, as with **Collection**, many languages have first class support for iterators, or provide them in their standard library. In these cases, since it is expected that developers will utilize the tools at hand, and not recreate them from scratch, it is reasonable to perform detection of the iterator itself during language conversion to POML. This pattern definition can then be used to connect the proper iterators with the proper collections, while allowing each hierarchy to be extended within developer code.

The connection mechanism is one of double dispatch, tying two objects and types together into a cross-referencing decision point. While some languages directly support this feature, others do not. It is therefore vital to the definition of **Iterator** that this concept be extracted out into a common format.

This leads me to believe that a new EDP, **DoubleDispatch**, is a necessary addition to the catalog.

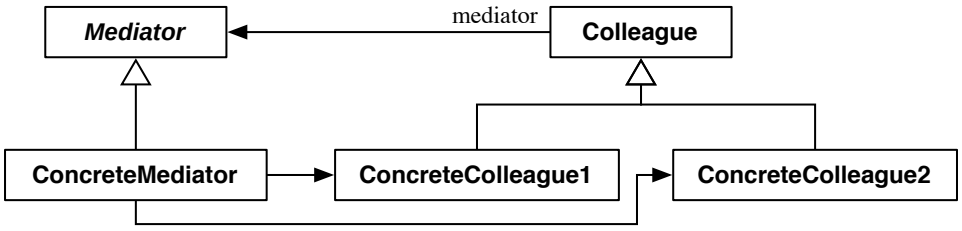


$$\frac{\begin{array}{l}(\textbf{Collection}(Aggregate, someType))\\ \textbf{FulfillsMethod}(Aggregate, ConcreteAggregate, CreateIterator)\\ (\textbf{DoubleDispatch}(ConcreteAggregate.CreateIterator, ConcreteIterator.constructor,\\ \qquad ConcreteIterator, ConcreteAggregate))\\ (\textbf{IteratorSemantics}(Iterator))\\ ConcreteAggregate.CreateIterator \overset{v}{\rightsquigarrow} rtnVal\\ \textbf{RetrieveNew}(Client.someMethod, ConcreteAggregate.CreateIterator.rtnVal, someSink)\end{array}}{\textbf{Iterator}(Iterator, ConcreteIterator, Aggregate, ConcreteAggregate)}$$

D.3.5 Mediator

Mediator presents an interesting problem, as there are *no* EDPs from the current catalog that appear in the general form presented in Gang of Four. Not even a determination of **Delegate** vs. **Conglomeration** can be made, since the calling semantics are left completely arbitrary. Because of this, the definition here is likely to be triggered by a plethora of instances in systems that do fit, but do not have mediation between objects as their sole, or even primary, function.

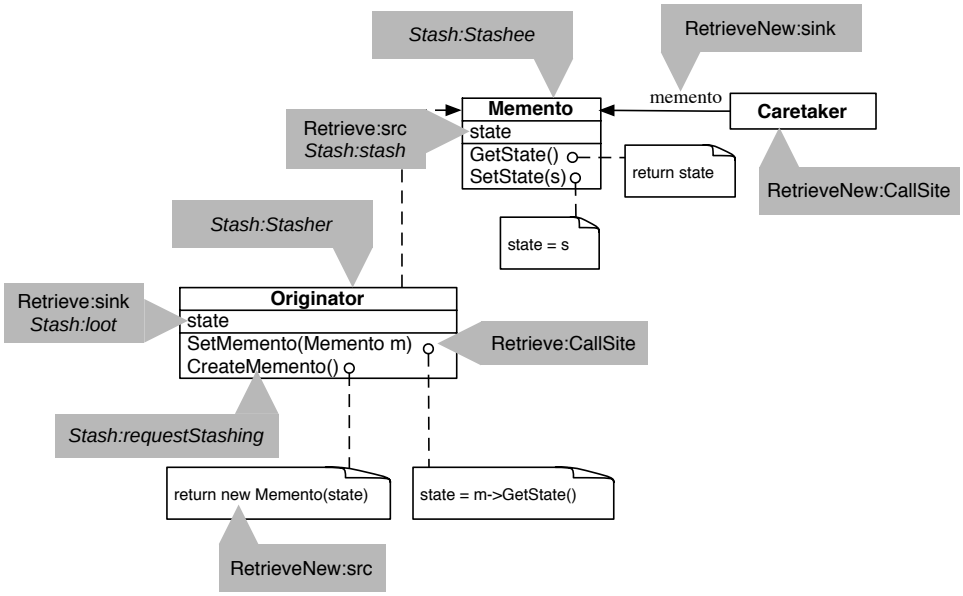
I expect that, like **Facade**, this is a pattern that will be best determined by metrics that establish a 'hot spot' of object interaction, where one object sits as the center of a rich calling graph. It is possible that one or more instances of **Collection** may be detectable within Mediator or ConcreteMediator as a common implementation feature for mediation between dynamic groups of objects.



$$\frac{\begin{array}{l} Colleague.mediator : Mediator \\ ConcreteMediator <: Mediator \\ ConcreteMediator.anObj : ConcreteColleague \\ ConcreteColleague <: Colleague \\ ConcreteMediator.someMethod <_{\mu} ConcreteColleague.aMethod \end{array}}{\textbf{Mediator}(Mediator, ConcreteMediator, Colleague, ConcreteColleague)}$$

D.3.6 Memento

Another pattern with complex runtime dynamics, **Memento** introduces another low-level concept that may make a useful addition to the EDP catalog: **Stash**. **Store** would be another possible name, but given its existing history as a general synonym for assignment, I would prefer to have a more specific term. This is created when a parameter passed to an instance method is assigned, without modification, to an instance field. While this behavior can be captured directly by a ρ-calculus judgment, it is useful to be able to discuss composition in higher-level terms. Stashing is the distinguishing characteristic of **Memento**. I present here an initial definition for **Stash**. Note that the calling parameter *loot* is copied in by value, not mapped in by name. This ensures that the *Stashee* has a unique copy for later retrieval. If a map or reference were passed in, the stashed object could not be considered safely stashed, as other objects may have a reference to it, and could alter it behind the scenes. A **Stash** differs from the above **DoubleDispatch** because the stashed object has no knowledge of the enclosing Memento object.



$$\frac{\begin{array}{l} Stasher.requestStashing <_{\mu} Stashee.acceptLoot \\ Stashee.acceptLoot\{\!\{s \overset{v}{\leftarrow} loot\}\!\} \\ \|Stashee.stash \Leftarrow Stashee.acceptLoot.s\|_{Stashee.acceptLoot} \end{array}}{\mathbf{Stash}(Stasher, requestStashing, loot, Stashee, acceptLoot, stash)}$$

$$\frac{\begin{array}{l} \mathbf{RetrieveNew}(Caretaker.someMethod, Originator.CreateMemento.rtnVal, \\ \qquad\qquad Caretaker.memento) \\ (\mathbf{Stash}(Originator, CreateMemento, Originator.state, Memento, constructor, state)) \\ Originator.m : Memento \\ \mathbf{Retrieve}(Originator.SetMemento, Originator.m, Originator.state) \end{array}}{\mathbf{Memento}(Originator, CreateMemento, SetMemento, Memento)}$$

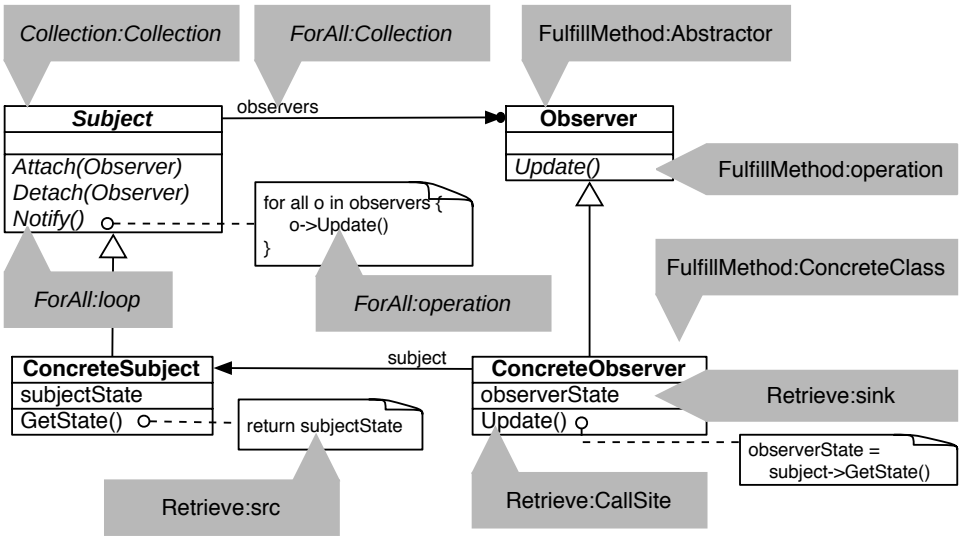
D.3.7 Observer

Observer has much in common with **Composite**, in that much behavior is left implicit in the Gang of Four discussion. For instance, given the requirements for a Subject to provide an interface for adding or removing instances of Observer, Subject starts to look much like an instance of a **Collection**. The notification mechanism then becomes iteration over the internal collection, another instance of **ForAll** mentioned in the **Composite** discussion here.

The ChangeManager example (Gamma et al., 1995, p. 300) indicates that a **Mediator** is a frequent interloper in the middle of an **Observer** instance. The movement of the **ForAll** semantics from the Subject to the ChangeManager indicates that transitivity may be needed for that concept.

The state synchronization aspect of **Observer** as described in Gang of Four, is problematic, as it is

optional.



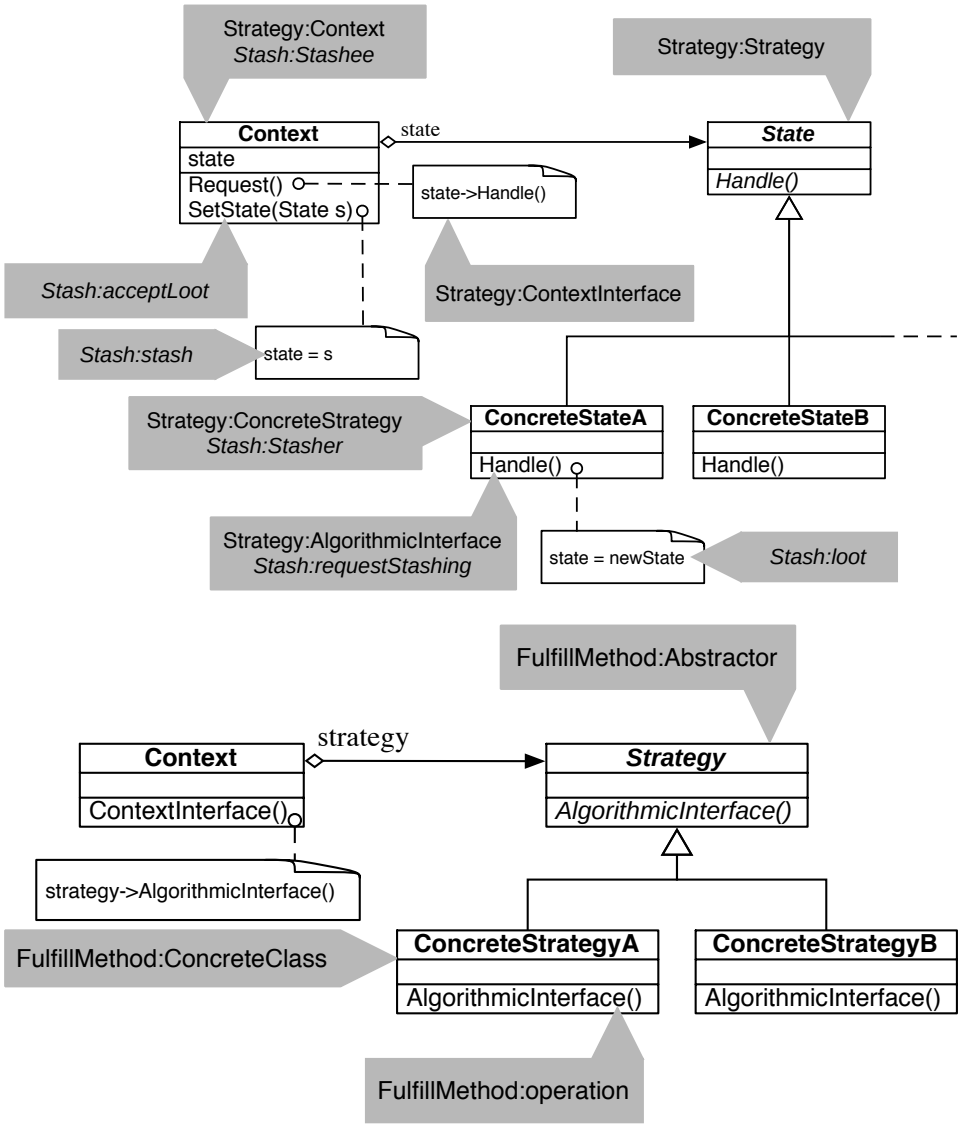
$$\frac{\begin{array}{l}(\textbf{Collection}(Subject.observers, Observer))\\ (\textbf{ForAll}(Subject.Notify, observers, Update))\\ ConcreteSubject <: Subject\\ \textbf{FulfillsMethod}(Observer, ConcreteObserver, Update)\\ ConcreteObserver.subject : ConcreteSubject\\ \textbf{Retrieve}(ConcreteObserver.Update, ConcreteSubject.subjectState,\\ \qquad\qquad ConcreteObserver.observerState)\end{array}}{\textbf{Observer}(Observer, ConcreteObserver, Subject, ConcreteSubject)}$$

D.3.8 State and Strategy

I am choosing to discuss these two patterns simultaneously because they are, at many levels, the same pattern. For instance, if one looks at the original structural diagrams from Gang of Four, they are nearly indistinguishable. **State** can be considered a special case of **Strategy**, one in which the object representing the strategy might be replaced *by* the instance it points to - or it may not. These two patterns are, in my mind, variations of the same theme, and cannot be easily teased apart as separate patterns.

One possible solution is to use **Stash** from **Memento** to determine cases in which one subclass of State is setting the instance of State in Context to an instance of another subclass of State. This will work for situations where the States are responsible for altering the state of the Context, but will likely, again, be indistinguishable from **Strategy**, as the Context there is almost certainly going to be responsible for the algorithm chosen. In situations where algorithms are self-modifying, then the selection of the state of the algorithm can be considered an implementation of **State**.

In any case, the weak definition of **Strategy** here affects both pattern definitions. I hope that further testing of real world examples will provide further insights into the necessary refinements.

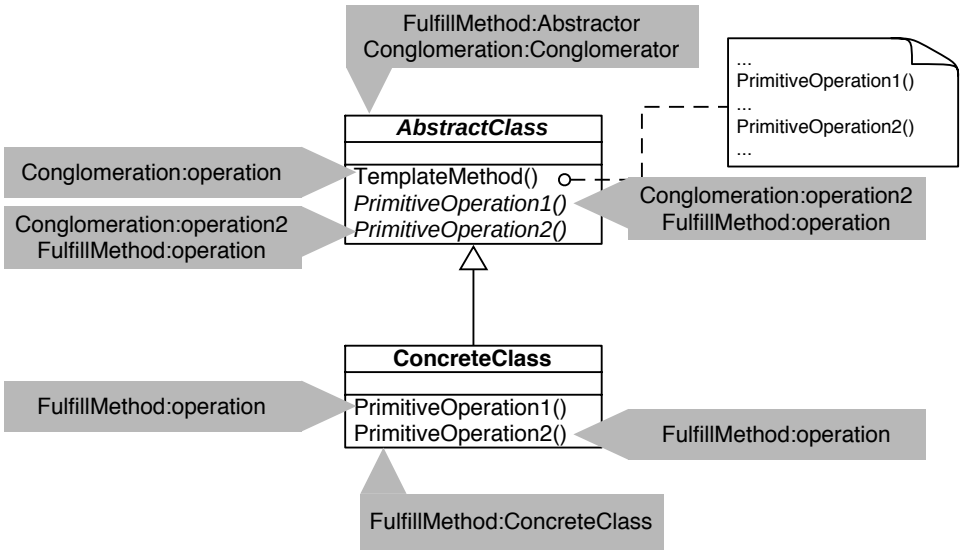


$$\frac{\begin{array}{l}\mathbf{FulfillMethod}(State, ConcreteStrategy, AlgorithmicInterface)\\ Context.strategy : Strategy\\ Context.Request <_{\mu} Context.strategy.AlgorithmicInterface\end{array}}{\mathbf{Strategy}(Strategy, Context, ContextInterface, ConcreteStrategy, AlgorithmicInterface)}$$

$$\frac{\begin{array}{l}\mathbf{Strategy}(State, Context, Request, ConcreteStateA, Handle)\\ state : ConcreteStateB\\ ConcreteStateB <: State\\ (\mathbf{Stash}(ConcreteStateA, Handle, newState, Context, SetState, state))\end{array}}{\mathbf{State}(State, Context, Request, ConcreteStateA, Handle)}$$

D.3.9 Template Method

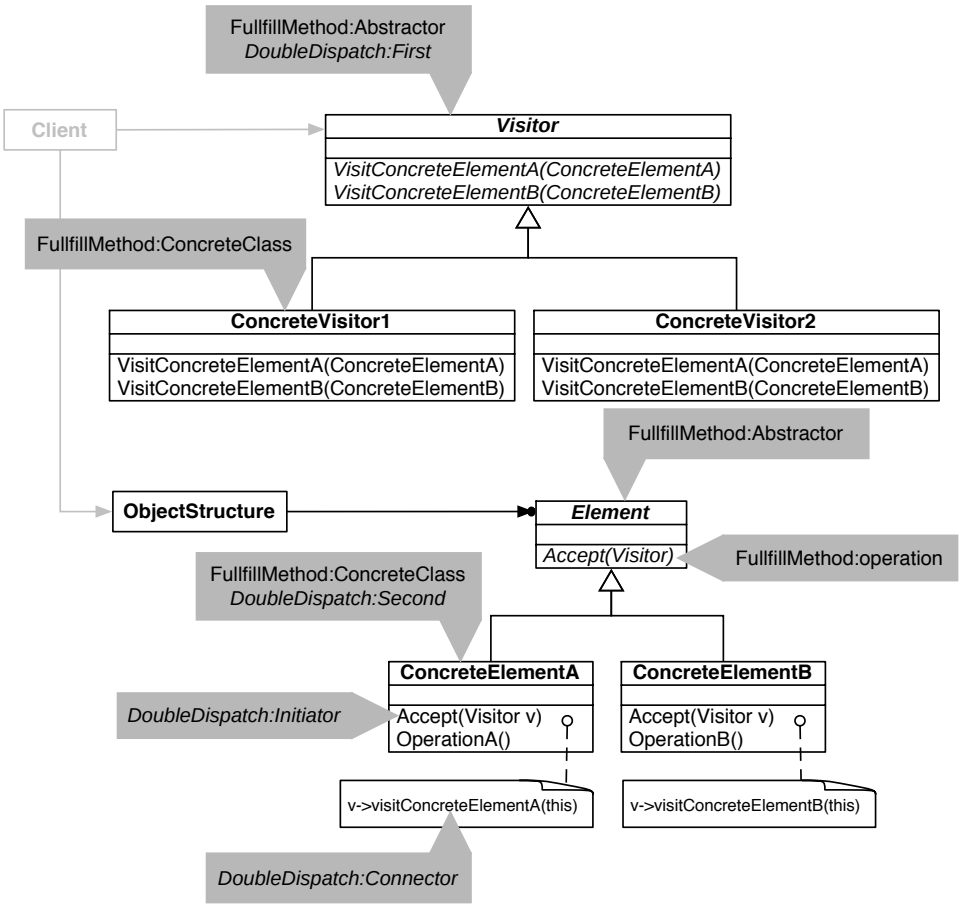
TemplateMethod is another excellent example of taking two very simple concepts and combining them in a very specific way to produce a much more powerful construct. This patterns relies simply on two lower patterns, each of which is rather primitive. Requiring the target method of a **Conglomeration** to be the abstract method in an instance of **FulfillMethod**, however, forces a subclass to handle the details in a way such that the implementation of portions of the algorithm can be adjusted dynamically.



$$\frac{\textbf{FulfillMethod}(AbstractClass, ConcreteClass, PrimitiveOperation1) \quad \textbf{Conglomeration}(AbstractClass, TemplateMethod, PrimitiveOperation1)}{\textbf{TemplateMethod}(AbstractClass, ConcreteClass, TemplateMethod, PrimitiveOperation1)}$$

D.3.10 Visitor

The nascent pattern **DoubleDispatch** from **Iterator** becomes a primary component of **Visitor**. By hoisting this concept out from the class inheritance example structure of the Gang of Four discussion, the definition allows languages that directly support such a feature, such as CLOS (Steele, 1984), to provide simpler implementations that still conform to the core concepts and intents of **Visitor**.



FulfillMethod$(Visitor, ConcreteVisitor, VisitConcreteElement)$
FulfillMethod$(Element, ConcreteElement, Accept)$
(**DoubleDispatch**$(ConcreteElement.Accept, Visitor.VisitConcreteElement, Visitor,$
$ConcreteElement)$)

Visitor$(Visitor, ConcreteVisitor, VisitConcreteElement, Element,$
$ConcreteElement, Accept)$

Appendix E

gcc Dump Tree Format

E.1 The gcc dump tree format

Unfortunately, the `gcc` dump tree format is essentially undocumented, and most of the details required extensive testing and reverse engineering to comprehend. I will provide here only a basic rundown of the format. The source code for `gcctree2poml` and an upcoming document detailing its production contain a few essential lessons learned from painful experience. It is my hope that this can provide feedback to the `gcc` team to produce a dump tree format in future versions that is more logically consistent.

The basic format for `gcc` dump trees is seen in Figure E.1, showing a piece from an example dump file. I use the term 'tree' to conform with `gcc`'s use of the term, but the data properly creates a graph with cycles. The dump file describes a single graph as a series of numbered nodes. Each node is described by one or more lines, and starts with a `@` in column 1 of the text file. This is followed by the numerical identifier of the node in column 2, and then by the type of the node in column 9. After this comes the data specific to the node type in column 26. If it does not fit in one line, it is offset in the following line to be aligned with the data in the originating line. The data is composed of four-letter tags and an appended colon, a space, and then data of varying lengths. If the data is a number prepended by a `@`, it is a pointer to the corresponding node in the file. The graph corresponding to the nodes in Figure E.1 can be seen in Figure E.2.

In Figure E.1, there are thirteen nodes. The first one describes a namespace declaration, with name stored in node `@2`, the declarations starting in node `@3`, and the source given as file `<internal>` and line 0. `<internal>` indicates that this data was generated automatically be `gcc` and does not appear in the source code being compiled.

The second node is simply an identifying string. The string data is "::", of length 2. The length becomes important when parsing strings with spaces, colons, or `@` symbols. Now we know that the name of the namespace in node `@1` is `::`.

Node `@3` declares a function. It has a name, a type, it comes from line 8 of source file `test.cpp`, it is a C function, declared `extern` and has both an argument list and defined method body. It also indicates through the `chan` key that it is part of a list, and the next link in the chain can be found in node `@6`.

```
@1      namespace_decl   name: @2       srcp: <internal>:0
                         dcls: @3
@2      identifier_node  strg: ::       lngt: 2
@3      function_decl    name: @4       type: @5       srcp: test.cpp:8
                         chan: @6       C              args: @7
                         extern         body: @8
@4      identifier_node  strg: main     lngt: 4
@5      function_type    size: @9       algn: 64       retn: @10
                         prms: @11
@6      type_decl        name: @12      type: @13      srcp: test.cpp:1
                         artificial     chan: @14
@7      parm_decl        type: @10      scpe: @3       srcp: test.cpp:8
                         chan: @15      argt: @10      size: @16
                         algn: 32       used: 0
@8      compound_stmt    line: 9        body: @17      next: @18
@9      integer_cst      type: @19      low : 64
@10     integer_type     name: @20      size: @16      algn: 32
                         prec: 32       min : @21      max : @22
@11     tree_list        valu: @10      chan: @23
@12     identifier_node  strg: A        lngt: 1
@13     record_type      name: @6       size: @24      algn: 8
                         struct         flds: @25      fncs: @26
                         binf: @27
```

Figure E.1: Example gcc dump tree format

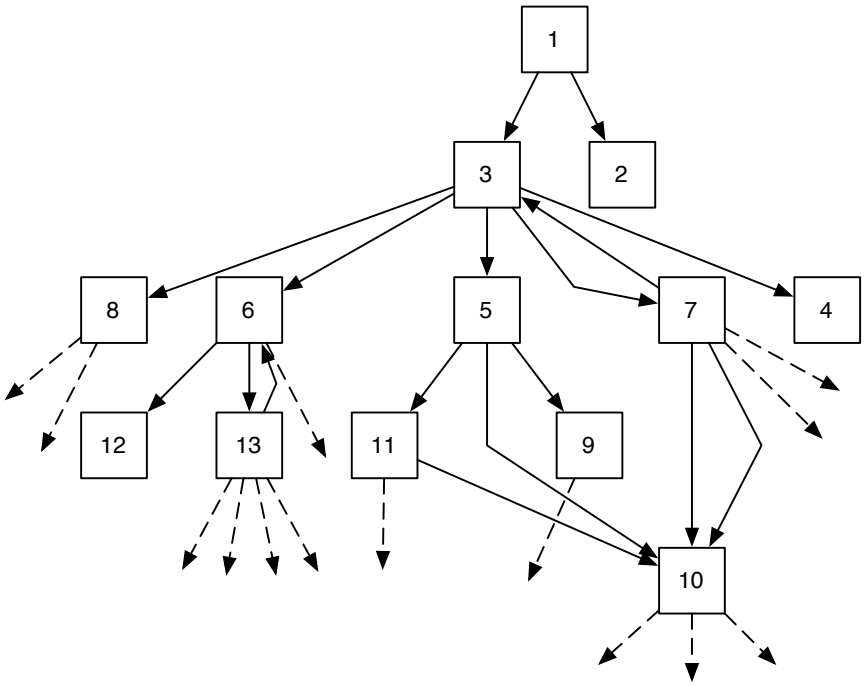


Figure E.2: Graph for example gcc dump tree

The name of the function in node `@3` is defined in node `@4` as "main".

Node `@5` provides the function type of node `@3`, and includes a type size and byte alignment for the compiler's use, and pointers to the return type and parameter types list. Note that this is distinct from and redundant in light of the arguments list in node `@3`.

Long before we have a full definition for node `@3`, node `@6` gives us another declaration to work with, this time for a type. Skipping ahead a bit, the name is given in node `@12` as 'A', and the type of this declared type is in node `@13`, a `record_type` node. This indicates a class being declared. Note that although this class declaration does appear in the source file as indicated, it is illogically tagged as `artificial`, while the definition in node `@13` is not.

Node `@7` pulls us back into the function declared in node `@3` by starting the argument list. It has a type, a scope that points back to the function, a second argument type (which is the same as the original type), a size, an alignment, and a flag as to whether or not the parameter is used in the method body. It also has the requisite source data that all declarations have.

The method body for function `main` starts in node `@8`. It is a compound statement, describing the way that statements are chained together. Note that this is a distinct mechanism from declaration chaining; it points to two statements which are not shown in our dump file sample. Almost certainly they are each further compound statements. A line number appears, but no file information is provided. Again, this is distinct from the information provided for declarations. Also, there is no pointer back to the previous statement or originating scope, meaning that, if a tool is inside a method body and wishes to know where that method body came from, it has to start walking the graph from the beginning again, or keep state tracking the sourcefile while creating and walking a graph from this file.

Constants are handled as in node `@9`, which describes the integer constant '64'. The type must be looked up, however, in order to find the precise type information, even though it is implicit in the constant string.

An example of an integer type is in node `@10`, which points to a name and provides numeric information such as precision, minimum and maximum values. It also gives the size and byte alignment for the compiler.

In addition to the two chaining mechanisms already described, node `@11` introduces a *third* chaining mechanism, the `tree_list`. The current link's value is pointed to by the `valu` key, and the next link is given by `chan`. It should be noted that three chaining mechanisms exist. I have found nothing of note distinguishing them other than their context. There are also subtleties and special cases that crop up in strange circumstances.

The gcc dump files are, as seen above, highly verbose, redundant in some areas, and woefully lacking in content in others. Reverse-engineering this format constituted the bulk of the implementation time of SPQR, and, in retrospect, was a poor choice for this project despite the advantages that C++ gave as a language choice. I hope that others can benefit from the gcctree2poml tool and especially its parsing code.

E.2 gcc created artifacts

C++ automatically creates a number of methods with default implementations for each class defined by the user, unless the user explicitly provides versions of them. Among these is the default constructor, the assignment operator, and a default destructor. Furthermore, gcc creates many internal constructs to be used at runtime to support basic features such as virtual methods, multiple inheritance, and so on. Generally these are invisible to the engineer, but they need to be considered when converting C++ to POML, since their behavior is assumed by the semantics of the language. An example of a POML conversion created by gcctree2poml from the code in Figure E.3 can be seen in at the end of this section. This is an exceptionally verbose representation of such a simple class. All gcc-compiled classes are similar.

Of course, this class gets split into two pieces, the poml:class and the poml:object class-object. The poml:class is the shorter of the two by far, but it is not without its peculiarities. Note in Figure E.4 the existence of the _vptr_$Top field in the class descriptor. This is the virtual method lookup table for each instance of the type Top. It is used internally by gcc and can be generally ignored for SPQR analysis. A default assignment operator method was obviously created as well. gcc does not expose the internals of these artificial methods in most cases, so their usefulness in SPQR analysis is extremely limited.

The class-object is where the majority of the gcc artifacts appear. There are six constructors and four destructors, despite there only being one of each defined in the original source code. The constructors are as follows. Top, __base_ctor, and __comp_ctor, which take a second parameter that indicates what memory allocation style to be using. Top_overload_1 corresponds to the method defined in the code. __base_ctor_overload_2, and __comp_ctor_overload_3 in turn call Top_overload_1 after performing basic setup.

Likewise, of the four destructors, only __dtor_Top is found in the original code. Even there, the form in gcc's internal representation differs from the code, ensuring clean up of the virtual table, and calling the global operator_delete to free the memory. The remaining destructors, __base_dtor, __comp_dtor,

```
//      Top test class
//      Jason McC. Smith Jul 28, 2004

class Top {
public:
        Top() { d _data = 100; };
        virtual ~Top() {};
        void setData( int newData ) { d_data = newData; };

private:
        int d_data;
};
```

Figure E.3: Example simple class

and __deleting_dtor, use the basic method as well, but are used for cases of inheritance and virtual methods or array deletion.

In all cases, once a basic, normalized form is achieved in the conversion tool, the artificial methods can be treated as any other.

POML class-object for class Top

```
<poml:object sourcefile="Top.hpp" line="4">
    <poml:name>Top__ClassObj</poml:name>
    <poml:type>Top__ClassObjType</poml:type>
    <poml:isclassobjfor>Top</poml:isclassobjfor>
    <poml:method sourcefile="Top.hpp" line="4">
        <poml:name>Top</poml:name>
        <poml:parameter>
            <poml:name>this</poml:name>
            <poml:type>Top</poml:type>
            <poml:keyword>kw0</poml:keyword>
        </poml:parameter>
        <poml:parameter>
            <poml:name>_ctor_arg</poml:name>
            <poml:type>Top</poml:type>
            <poml:keyword>kw1</poml:keyword>
        </poml:parameter>
    </poml:method>
    <poml:method sourcefile="Top.hpp" line="4">
        <poml:name>__base_ctor</poml:name>
        <poml:parameter>
            <poml:name>this
                <poml:scope>__base_ctor
                <poml:scope>Top__ClassObj</poml:scope></poml:scope>
            </poml:name>
            <poml:type>Top</poml:type>
            <poml:keyword>kw0</poml:keyword>
        </poml:parameter>
```

```
<poml:class sourcefile="Top.hpp" line="4">
    <poml:name>Top</poml:name>
    <poml:method sourcefile="Top.hpp" line="4">
        <poml:name>operator_assign</poml:name>
        <poml:parameter>
            <poml:name>_ctor_arg</poml:name>
            <poml:type>Top</poml:type>
            <poml:keyword>kw0</poml:keyword>
        </poml:parameter>
    </poml:method>
    <poml:method sourcefile="Top.hpp" line="8">
        <poml:name>setData</poml:name>
        <poml:parameter>
            <poml:name>newData
                <poml:scope>setData
                <poml:scope>Top</poml:scope></poml:scope>
            </poml:name>
            <poml:type>int</poml:type>
            <poml:keyword>kw0</poml:keyword>
        </poml:parameter>
        <poml:update sourcefile="Top.hpp" line="8">
            <poml:lhs>d_data
                <poml:scope>this
                <poml:scope>setData
                <poml:scope>Top</poml:scope></poml:scope></poml:scope>
            </poml:lhs>
            <poml:rhs>newData
                <poml:scope>setData<poml:scope>Top</poml:scope></poml:scope>
            </poml:rhs>
        </poml:update>
    </poml:method>
    <poml:field sourcefile="Top.hpp" line="4">
        <poml:name>_vptr_$Top</poml:name>
        <poml:type>__vtbl_ptr_type__ptr</poml:type>
    </poml:field>
    <poml:field sourcefile="Top.hpp" line="11">
        <poml:name>d_data</poml:name>
        <poml:type>int</poml:type>
    </poml:field>
</poml:class>
```

Figure E.4: POML class element for example C++ code

```
    <poml:parameter>
        <poml:name>_ctor_arg
            <poml:scope>__base_ctor
            <poml:scope>Top__ClassObj</poml:scope></poml:scope>
        </poml:name>
        <poml:type>Top</poml:type>
        <poml:keyword>kw1</poml:keyword>
    </poml:parameter>
</poml:method>
<poml:method sourcefile="Top.hpp" line="4">
    <poml:name>__comp_ctor</poml:name>
    <poml:parameter>
        <poml:name>this
            <poml:scope>__comp_ctor
            <poml:scope>Top__ClassObj</poml:scope></poml:scope>
        </poml:name>
        <poml:type>Top</poml:type>
        <poml:keyword>kw0</poml:keyword>
    </poml:parameter>
    <poml:parameter>
        <poml:name>_ctor_arg
            <poml:scope>__comp_ctor
            <poml:scope>Top__ClassObj</poml:scope></poml:scope>
        </poml:name>
        <poml:type>Top</poml:type>
        <poml:keyword>kw1</poml:keyword>
    </poml:parameter>
</poml:method>
<poml:method sourcefile="Top.hpp" line="6">
    <poml:name>Top_overload_1</poml:name>
    <poml:parameter>
        <poml:name>this
            <poml:scope>Top_overload_1
            <poml:scope>Top__ClassObj</poml:scope></poml:scope>
        </poml:name>
        <poml:type>Top</poml:type>
        <poml:keyword>kw0</poml:keyword>
    </poml:parameter>
    <poml:uses sourcefile="Top.hpp" line="6">
        <poml:objectname>Top</poml:objectname>
        <poml:fieldname>_ZTV3Top</poml:fieldname>
    </poml:uses>
    <poml:update sourcefile="Top.hpp" line="6">
        <poml:lhs>_vptr_$Top
            <poml:scope>this
            <poml:scope>Top_overload_1
            <poml:scope>Top__ClassObj</poml:scope></poml:scope></poml:scope>
        </poml:lhs>
        <poml:fromcall/>
        <poml:rhs sourcefile="Top.hpp" line="6">
            <poml:objectname>_ZTV3Top
                <poml:scope>Top</poml:scope>
            </poml:objectname>
```

```
            <poml:methodname>plus</poml:methodname>
        </poml:rhs>
    </poml:update>
</poml:method>
<poml:method sourcefile="Top.hpp" line="6">
    <poml:name>__base_ctor_overload_2</poml:name>
    <poml:parameter>
        <poml:name>this
            <poml:scope>__base_ctor_overload_2
            <poml:scope>Top__ClassObj</poml:scope></poml:scope>
        </poml:name>
        <poml:type>Top</poml:type>
        <poml:keyword>kw0</poml:keyword>
    </poml:parameter>
    <poml:calls sourcefile="Top.hpp" line="6">
        <poml:objectname>Top__ClassObj</poml:objectname>
        <poml:methodname>Top_overload_1</poml:methodname>
        <poml:callingparameter>
            <poml:name>this
                <poml:scope>__base_ctor_overload_2
                <poml:scope>Top__ClassObj</poml:scope></poml:scope>
            </poml:name>
            <poml:type>Top</poml:type>
            <poml:keyword>kw_0</poml:keyword>
            <poml:callby>map</poml:callby>
        </poml:callingparameter>
    </poml:calls>
</poml:method>
<poml:method sourcefile="Top.hpp" line="6">
    <poml:name>__comp_ctor_overload_3</poml:name>
    <poml:parameter>
        <poml:name>this
            <poml:scope>__comp_ctor_overload_3
            <poml:scope>Top__ClassObj</poml:scope></poml:scope>
        </poml:name>
        <poml:type>Top</poml:type>
        <poml:keyword>kw0</poml:keyword>
    </poml:parameter>
    <poml:calls sourcefile="Top.hpp" line="6">
        <poml:objectname>Top__ClassObj</poml:objectname>
        <poml:methodname>Top_overload_1</poml:methodname>
        <poml:callingparameter>
            <poml:name>this
                <poml:scope>__comp_ctor_overload_3
                <poml:scope>Top__ClassObj</poml:scope></poml:scope>
            </poml:name>
            <poml:type>Top</poml:type>
            <poml:keyword>kw_0</poml:keyword>
            <poml:callby>map</poml:callby>
        </poml:callingparameter>
    </poml:calls>
</poml:method>
<poml:method sourcefile="Top.hpp" line="7">
```

```
    <poml:name>__dtor_Top</poml:name>
    <poml:parameter>
        <poml:name>this
            <poml:scope>__dtor_Top
            <poml:scope>Top__ClassObj</poml:scope></poml:scope>
        </poml:name>
        <poml:type>Top</poml:type>
        <poml:keyword>kw0</poml:keyword>
    </poml:parameter>
    <poml:parameter>
        <poml:name>__in_chrg
            <poml:scope>__dtor_Top
            <poml:scope>Top__ClassObj</poml:scope></poml:scope>
        </poml:name>
        <poml:type>int</poml:type>
        <poml:keyword>kw1</poml:keyword>
    </poml:parameter>
    <poml:uses sourcefile="Top.hpp" line="7">
        <poml:objectname>Top</poml:objectname>
        <poml:fieldname>_ZTV3Top</poml:fieldname>
    </poml:uses>
    <poml:update sourcefile="Top.hpp" line="7">
        <poml:lhs>_vptr_$Top
            <poml:scope>this
            <poml:scope>__dtor_Top
            <poml:scope>Top__ClassObj</poml:scope></poml:scope></poml:scope>
        </poml:lhs>
        <poml:fromcall/>
        <poml:rhs sourcefile="Top.hpp" line="7">
            <poml:objectname>_ZTV3Top
                <poml:scope>Top</poml:scope>
            </poml:objectname>
            <poml:methodname>plus</poml:methodname>
        </poml:rhs>
    </poml:update>
    <poml:calls sourcefile="Top.hpp" line="7">
        <poml:objectname>__GLOBAL__</poml:objectname>
        <poml:methodname>operator_delete</poml:methodname>
        <poml:callingparameter>
            <poml:name>this
                <poml:scope>__dtor_Top
                <poml:scope>Top__ClassObj</poml:scope></poml:scope>
            </poml:name>
            <poml:type>Top</poml:type>
            <poml:keyword>kw_0</poml:keyword>
            <poml:callby>map</poml:callby>
        </poml:callingparameter>
    </poml:calls>
</poml:method>
<poml:method sourcefile="Top.hpp" line="7">
    <poml:name>__base_dtor</poml:name>
    <poml:parameter>
        <poml:name>this
```

```
            <poml:scope>__base_dtor
            <poml:scope>Top__ClassObj</poml:scope></poml:scope>
        </poml:name>
        <poml:type>Top</poml:type>
        <poml:keyword>kw0</poml:keyword>
    </poml:parameter>
    <poml:calls sourcefile="Top.hpp" line="7">
        <poml:objectname>Top__ClassObj</poml:objectname>
        <poml:methodname>__dtor_Top</poml:methodname>
        <poml:callingparameter>
            <poml:name>this
                <poml:scope>__base_dtor
                <poml:scope>Top__ClassObj</poml:scope></poml:scope>
            </poml:name>
            <poml:type>Top</poml:type>
            <poml:keyword>kw_0</poml:keyword>
            <poml:callby>map</poml:callby>
        </poml:callingparameter>
    </poml:calls>
</poml:method>
<poml:method sourcefile="Top.hpp" line="7">
    <poml:name>__comp_dtor</poml:name>
    <poml:parameter>
        <poml:name>this
            <poml:scope>__comp_dtor
            <poml:scope>Top__ClassObj</poml:scope></poml:scope>
        </poml:name>
        <poml:type>Top</poml:type>
        <poml:keyword>kw0</poml:keyword>
    </poml:parameter>
    <poml:calls sourcefile="Top.hpp" line="7">
        <poml:objectname>Top__ClassObj</poml:objectname>
        <poml:methodname>__dtor_Top</poml:methodname>
        <poml:callingparameter>
            <poml:name>this
                <poml:scope>__comp_dtor
                <poml:scope>Top__ClassObj</poml:scope></poml:scope>
            </poml:name>
            <poml:type>Top</poml:type>
            <poml:keyword>kw_0</poml:keyword>
            <poml:callby>map</poml:callby>
        </poml:callingparameter>
    </poml:calls>
</poml:method>
<poml:method sourcefile="Top.hpp" line="7">
    <poml:name>__deleting_dtor</poml:name>
    <poml:parameter>
        <poml:name>this
            <poml:scope>__deleting_dtor
            <poml:scope>Top__ClassObj</poml:scope></poml:scope>
        </poml:name>
        <poml:type>Top</poml:type>
        <poml:keyword>kw0</poml:keyword>
```

```
        </poml:parameter>
        <poml:calls sourcefile="Top.hpp" line="7">
            <poml:objectname>Top__ClassObj</poml:objectname>
            <poml:methodname>__dtor_Top</poml:methodname>
            <poml:callingparameter>
                <poml:name>this
                    <poml:scope>__deleting_dtor
                    <poml:scope>Top__ClassObj</poml:scope></poml:scope>
                </poml:name>
                <poml:type>Top</poml:type>
                <poml:keyword>kw_0</poml:keyword>
                <poml:callby>map</poml:callby>
            </poml:callingparameter>
        </poml:calls>
    </poml:method>
</poml:object>
```

Appendix F

POML Schema

Reference for the POML.xsd file included with SPQR. This is the XML Schema that defines the Pattern/Object Markup Language. Documentation is included within the file, as well as more fully described in Chapter 7.

```
<?xml version="1.0"?>
<xs:schema version="1.2" attributeFormDefault="unqualified" elementFormDefault="qualified"
  targetNamespace="http://www.cs.unc.edu/~smithja/spqr"
  xmlns="http://www.cs.unc.edu/~smithja/spqr"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:cl="http://www.cs.unc.edu/~smithja/docs">

  <xs:import namespace="http://www.cs.unc.edu/~smithja/docs"
    schemaLocation="http://www.cs.unc.edu/~smithja/docs/changelog.xsd"></xs:import>

  <xs:annotation>
    <xs:documentation xml:lang="en">
    Pattern/Object Markup Language (or POML) is a simple XML Schema for describing
    object-oriented code in a unified manner with support for Design Patterns as well.
    (It can be used for pure procedural code also.)  It supports class-based and object-
    based systems (and the two can be, and often are, mixed and matched.)  Any OO (or
    procedural) language should be definable in POML, given enough thought.  (This does
    not mean that every aspect of any language can be mapped to POML, only that the
    aspects that POML addresses should be representable in most any language.)

    For instance, 'a = b + c' is best thought of as its compiler-representation
    equivalent: update(a, operator+(b, c)), and so on.  C++, for example, is best
    translated to POML from a basic AST with class annotation.  POML follows the basic
    principles of the sigma calculus, as described by Martin Abadi and Luca Cardelli in
    _A Theory of Objects_, Springer-Verlag, 1996, and the rho calculus as defined by the
    author in UNC-CS reports TR03-07, TR03-33 and other publications.

    Copyright 2004-2005, Jason McC. Smith, all rights reserved.
    </xs:documentation>
  </xs:annotation>

  <xs:element name="system">
    <xs:complexType>
      <xs:annotation>
        <xs:documentation xml:lang="en">
        Wrapper for ObjectML description.  Top level element in POML file. Contains
        instances of changelog entries, classes, objects, patterns, and at most one
        resultpattern.  All elements are optional.
        </xs:documentation>
      </xs:annotation>
      <xs:sequence>
        <xs:element name="changelog" type="cl:changelog" maxOccurs="unbounded"
```

```
          minOccurs="0"/>
        <xs:element name="class" type="class" maxOccurs="unbounded" minOccurs="0"/>
        <xs:element name="object" type="object" maxOccurs="unbounded" minOccurs="0"/>
        <xs:element name="pattern" type="pattern" maxOccurs="unbounded" minOccurs="0"/>
        <xs:element name="resultpattern" type="resultpattern" maxOccurs="1" minOccurs="0"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:complexType name="nameditem">
    <xs:annotation>
      <xs:documentation xml:lang="en">
      Base type for 'named elements'.  These include objects, classes, methods, fields,
      parameters, method results, and update arguments.  The 'name' element is a
      scopeablename.  Optional elements include 'source' and 'line' for adding information
      about source code files that may have been used to generate the POML files.
      Certain types of nameditems, such as method names, can be explicitly scoped, or can
      be left without a scope element, resulting in an implicitly scoped name, in which
      case the proper scope may be deducible by inspecting the name element of the
      grandparent node.  (See POML2Otter.xsl for an example of this.)
      </xs:documentation>
    </xs:annotation>
    <xs:sequence>
      <xs:element name="name" type="scopeablename"/>
      <!-- Optional source/line information to perform reverse location searches -->
      <xs:element name="source" type="xs:string" minOccurs="0" maxOccurs="1"/>
      <xs:element name="line" type="xs:string" minOccurs="0" maxOccurs="1"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="scopeablename" mixed="true">
    <xs:annotation>
      <xs:documentation xml:lang="en">
      A name that has an optional 'scope' element.  Since 'scope' is also of this type,
      nested scopes are supported transparently.
      </xs:documentation>
    </xs:annotation>
<!--    <xs:complexContent>
      <xs:extension base="xs:string">
        <xs:sequence>
          <xs:element name="scope" type="scopeablename" minOccurs="0" maxOccurs="1"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent> -->
    <xs:sequence>
      <xs:element name="scope" type="scopeablename" minOccurs="0" maxOccurs="1"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="object">
    <xs:annotation>
      <xs:documentation xml:lang="en">
      A raw (instantiated) object.  Useful for emulating class-based systems in a pure
      object notation.  One can follow the example of Abadi and Cardelli in _A Theory of
      Objects_, and create an explicit object that contains the constructors, destructors,
      and static fields and methods for a class.  Such objects need to have the
      poml:isclassobjfor element set.
      </xs:documentation>
```

```
  </xs:annotation>
  <xs:complexContent>
    <xs:extension base="nameditem">
      <xs:sequence>
        <xs:element name="type" type="scopeablename"/>
        <xs:element name="isclassobjfor" type="scopeablename" minOccurs="0"
          maxOccurs="1"/>
        <xs:element name="method" type="method" minOccurs="0"/>
        <xs:element name="field" type="field" minOccurs="0"/>
        <xs:element name="update" type="update" minOccurs="0"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="class">
  <xs:annotation>
    <xs:documentation xml:lang="en">
    Descriptor for all instance-specific items in a class type definition.  Contains
    inheritance information, methods, fields.
    </xs:documentation>
  </xs:annotation>
  <xs:complexContent>
    <xs:extension base="nameditem">
      <xs:sequence maxOccurs="unbounded" minOccurs="0">
        <xs:element name="parent" type="type_reliance" minOccurs="0"/>
        <xs:element name="inheritedmethod" type="scopeablename" minOccurs="0"/>
        <xs:element name="inheritedfield" type="scopeablename" minOccurs="0"/>
        <xs:element name="directlycalls" type="dircall" minOccurs="0"/>
        <xs:element name="method" type="method" minOccurs="0"/>
        <xs:element name="field" type="field" minOccurs="0"/>
        <xs:element name="update" type="update" minOccurs="0"/>
        <!-- Implicit phi_reliance in field? -->
        <xs:element name="uses" type="phi_reliance" minOccurs="0"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="dircall">
  <xs:annotation>
    <xs:documentation xml:lang="en">
    Mapping from a superclass' method to a local name that prevents overriding (masking)
    of the original method name, but allows local access.
    </xs:documentation>
  </xs:annotation>
  <xs:complexContent>
    <xs:extension base="scopeablename">
      <xs:sequence>
        <xs:element name="as" type="scopeablename"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="method">
  <xs:annotation>
    <xs:documentation xml:lang="en">
```

```
    Methods have a name (nameditem), an optional list of parameters, an optional result,
    and possibly a 'static' tag indicating a class-level (as opposed to instance-level)
    ownership.  A method can be tagged as 'abstract' (no definition), *or* it can include
    'calls', 'uses' and 'update' relationships.
    </xs:documentation>
  </xs:annotation>
  <xs:complexContent>
    <xs:extension base="nameditem">
      <xs:sequence>
        <xs:element name="parameter" type="parameter" minOccurs="0"
          maxOccurs="unbounded"/>
        <xs:element name="result" type="resulttype"/>
        <xs:element name="static" minOccurs="0" maxOccurs="1">
          <xs:complexType>
          </xs:complexType>
        </xs:element>
        <xs:choice>
          <xs:element minOccurs="0" maxOccurs="1" name="abstract">
            <xs:complexType>
            </xs:complexType>
          </xs:element>
          <xs:sequence minOccurs="0">
            <xs:element name="calls" type="mu_reliance" minOccurs="0"
              maxOccurs="unbounded"/>
            <xs:element name="uses" type="phi_reliance" minOccurs="0"
              maxOccurs="unbounded"/>
            <xs:element name="update" type="update" minOccurs="0"
              maxOccurs="unbounded"/>
          </xs:sequence>
        </xs:choice>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="resulttype">
  <xs:annotation>
    <xs:documentation xml:lang="en">
    A result is another named type that indicates whether it is passed back by
    reference (map) or by value (copy).
    </xs:documentation>
  </xs:annotation>
  <xs:complexContent>
    <xs:extension base="nameditem">
      <xs:sequence>
        <xs:element name="passby" type="callbytype"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="update">
  <xs:annotation>
    <xs:documentation xml:lang="en">
    Update is the assignment operator.  It has a left side (target), and a right
    side (source).  The left side is a named item, such as a field.  The right side
    can be another field, or, most often, a call to a method with parameters.
    If the rhs is a call, then the tag element "fromcall" will be included, and the rhs
```

```
    will take the form of a mu-reliance, with objectname, methodname, and parameters.
    </xs:documentation>
  </xs:annotation>
  <xs:sequence>
    <xs:element name="lhs" type="nameditem"/>
    <xs:choice>
      <xs:sequence>
        <xs:element name="fromcall" minOccurs="0" maxOccurs="1">
          <xs:complexType></xs:complexType>
        </xs:element>
          <xs:element name="rhs" type="mu-reliance"/>
      </xs:sequence>
      <xs:sequence>
          <xs:element name="rhs" type="nameditem"/>
      </xs:sequence>
    </xs:choice>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="field">
  <xs:annotation>
    <xs:documentation xml:lang="en">
    Fields are also named items, with an optional 'static' tag, and a required type,
    which is a scoped name.
    </xs:documentation>
  </xs:annotation>
  <xs:complexContent>
    <xs:extension base="nameditem">
      <xs:sequence>
        <xs:element name="static" minOccurs="0" maxOccurs="1">
          <xs:complexType>
          </xs:complexType>
        </xs:element>
        <xs:element name="type" type="scopeablename"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="parameter">
  <xs:annotation>
    <xs:documentation xml:lang="en">
    Parameters have a name, a type, and a 'keyword'.  Some languages (such as
    Objective-C) have required keyword support for arguments.  In some (Python) it is
    optional, and in others (C++), unknown.  When translating an optional or no keyword
    language, simply make up a string unique to the parameter within the method.
    'kw1', 'kw2', 'kw3' and so on.   This allows for mapping external names to internal
    ones in a methodical manner.
    </xs:documentation>
  </xs:annotation>
  <xs:complexContent>
    <xs:extension base="nameditem">
      <xs:sequence>
        <xs:element name="type" type="scopeablename"/>
        <xs:element name="keyword" type="xs:string"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
```

```
</xs:complexType>

<xs:complexType name="callingparameter">
  <xs:annotation>
    <xs:documentation xml:lang="en">
    A calling parameter is a parameter with one additional element, 'callby',
    mirroring the 'result' element.  It indicates whether a parameter is passed in
    via reference (map) or value (copy).
    </xs:documentation>
  </xs:annotation>
  <xs:complexContent>
    <xs:extension base="parameter">
      <xs:sequence>
        <xs:element name="callby" type="callbytype"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:simpleType name="callbytype">
  <xs:restriction base="xs:string">
    <xs:enumeration value="copy"/>
    <xs:enumeration value="map"/>
  </xs:restriction>
</xs:simpleType>

<!-- From here down this is rho-calculus and pattern related material.  We have the three
types of reliance, patterns made of roles, and resultpatterns used in POML documents
intended as input for theorem provers.-->
<xs:complexType name="mu_reliance">
  <xs:annotation>
    <xs:documentation xml:lang="en">
    Corresponds to the 'calls' element of methods, and also the mu form reliance
    operator of SPQR.  Indicates a method calling another method, with a list of calling
    parameters.
    </xs:documentation>
  </xs:annotation>
  <xs:sequence>
    <xs:element name="objectname" type="scopeablename"/>
    <xs:element name="methodname" type="scopeablename"/>
    <xs:element name="parameter" type="callingparameter" minOccurs="0"
      maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="phi_reliance">
  <xs:annotation>
    <xs:documentation xml:lang="en">
    The 'uses' element of methods, equivalent to the phi form reliance operator of SPQR.
    Indicates that the holding method uses a field in some manner.
    </xs:documentation>
  </xs:annotation>
  <xs:sequence>
    <xs:element name="objectname" type="scopeablename"/>
    <xs:element name="fieldname" type="scopeablename"/>
  </xs:sequence>
</xs:complexType>
```

```
<xs:complexType name="mu_reliance">
  <xs:annotation>
    <xs:documentation xml:lang="en">
    Corresponds to the 'calls' element of methods, and also the mu form reliance
    operator of SPQR.  Indicates a method calling another method, with a list of calling
    parameters.
    </xs:documentation>
  </xs:annotation>
  <xs:sequence>
    <xs:element name="objectname" type="scopeablename"/>
    <xs:element name="methodname" type="scopeablename"/>
    <xs:element name="parameter" type="callingparameter" minOccurs="0"
      maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="kappa_reliance">
  <xs:annotation>
    <xs:documentation xml:lang="en">
    The 'uses' element of methods, equivalent to the phi form reliance operator of SPQR.
    Indicates that the holding method uses a field in some manner.
    </xs:documentation>
  </xs:annotation>
  <xs:sequence>
    <xs:element name="objectname" type="scopeablename"/>
    <xs:element name="fieldname" type="scopeablename"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="type_reliance">
  <xs:annotation>
    <xs:documentation xml:lang="en">
    The inheritance relationship between classes.
    </xs:documentation>
  </xs:annotation>
  <xs:sequence>
    <xs:element name="classname" type="scopeablename"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="role">
  <xs:annotation>
    <xs:documentation xml:lang="en">
    Pattern roles have a name, and indicate which object/method/field in the system is
    playing that part for this particular instance of a pattern.
    </xs:documentation>
  </xs:annotation>
  <xs:sequence>
    <xs:element name="name" type="xs:string"/>
    <xs:element name="fulfilledBy" type="scopeablename"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="pattern">
  <xs:annotation>
    <xs:documentation xml:lang="en">
    A design pattern has specific roles that must be fulfilled for the pattern to exist
    in a system.   This type provides the name of the pattern, and a list of roles.
```

```
      </xs:documentation>
    </xs:annotation>
    <xs:sequence>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="role" type="role" maxOccurs="unbounded"/>
      <xs:element name="reliesOn" type="xs:string" maxOccurs="unbounded" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="quantifierslist">
    <xs:annotation>
      <xs:documentation xml:lang="en">
      Somewhat of a hack for the production of logic formulas representing resultpatterns,
      this is a list of the universal quantifiers that need to be described in an example
      codebase for a pattern to be defined.  See the existing design pattern definition
      files for examples.
      </xs:documentation>
    </xs:annotation>
    <xs:sequence>
      <xs:element name="quantifier" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="resultpattern">
    <xs:annotation>
      <xs:documentation xml:lang="en">
      A pattern that is being described and defined in the current POML file.  Only one
      resultpattern can exist in a system at a time, in which case the system is assumed
      to define that pattern.  Since this is a logical construct, we have a list of
      quantifiers to maintain.
      </xs:documentation>
    </xs:annotation>
    <xs:complexContent>
      <xs:extension base="pattern">
        <xs:sequence>
          <xs:element name="quantifiers" type="quantifierslist"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <xs:simpleType name="atprule">
    <xs:annotation>
      <xs:documentation xml:lang="en">
      There are time when, sadly, POML is simply not enough to render logic constructs in
      automated theorem provers.  To accomodate these instances, the atprule is an
      xs:string that is specific to an ATP (as indicated by the 'atp' attribute, much like
      the 'xml:lang' attribute of xs:documentation elements) and is passed unchanged to an
      input file for that type of ATP.  It allows small tweaks to be added to a formula for
      an ATP, such as ensuring that two quantifiers in Otter are not the same, via the
      $NOT($ID(quant1, quant2)) construct.
      </xs:documentation>
    </xs:annotation>
    <xs:extension base="xs:string"/>
    <xs:attribute name="atp" type="xs:string"/>
  </xs:simpleType>

</xs:schema>
```

Appendix G

Otter Inputs

G.1 Headers

Used as header files for OTTER runs. These set up the general OTTER environment, including the algorithmic approach, and any memory or resource limits.

G.1.1 Searching header

Used for search runs, not inference production runs.

```
%%%%%%%%%%%%%%%
%
% Header for automatically generated Otter run
% Used for pattern *searches* - the full calculus isn't needed for these
%
% Jason McC. Smith
% v2.0 2004/04/03
%
%%%%%%%%%%%%%%%

set(hyper_res).

assign(stats_level, 2).
assign(max_proofs, -1).
assign(max_mem, -1).
set(pretty_print).


include('sigmaops.otter').
include('rhoops.otter').
```

G.1.2 Inference header

Used for full inference production runs, creates a massive table of facts about the system based on the transitivities of ρ-calculus.

```
%%%%%%%%%%%%%%%
%
% Header for inference file generating Otter run
% Includes the full rho-calculus to generate every inference possible
%   as an up-front cost for searching.
```

```
% Prints lists at end so the inferred rules can be extracted.
%
% Jason McC. Smith
% v1.0 2005/07/18
%
%%%%%%%%%%%%%%%

set(hyper_res).
assign(stats_level, 2).
assign(max_proofs, -1).
assign(max_mem, -1).
%set(pretty_print).

set(print_lists_at_end).

include('rhocalc.otter').
```

G.1.3 Debugging header

Not used in normal SPQR use, it provides valuable but performance expensive insights into how OTTER may or may not be producing inferences. Simply added to the run along with a proper setup header, such as the two previously defined.

```
assign(stats_level, 1).
set(very_verbose).
set(print_lists_at_end).
```

G.2 ς-calculus operators

The basic definitions for ς-calculus as expressed in OTTER.

```
%%%%%
%%
%%  sigmaops.otter
%%  Jason McC. Smith
%%  v1.1, August 30, 2005
%%
%%  Operator definitions for the basic sigma-calculus.
%%

%%%%%
%%
%%  Basics
%
% Typing: anObject : SomeType
op( 690, xfx, : ).
% Synonym for ':'
op( 690, xfx, oftype ).
% Selection operator.  Maps to '.' in sigma-calculus.
op( 630, yfx, dot).
```

```
%%%%%
%%
%%  Classes
% Inheritance via type:  Subclass inh Superclass
op( 690, xfy, inh ).
% Explicit inheritance of a superclass' method or field.
op( 650, xfx, inherits ).
% Method declarations/definitions within a class
op( 650, xfx, declares ).
op( 650, xfx, defines ).
% Ties a classobj to the class type
op( 690, xfx, isclassobjfor ).


%%%%%
%%
%%  Constructs used in Otter input, as functional forms
%%
%%  Update
% By call - used when rhs is a method invocation
%   updatebycall(lhs, rhs)
% By call, parameter - to hook a parameter to an update by call
% - see below for 'callstyle'
%   updatebycallparam(lhs, callstyle(called, kw, param))
% By field - used when rhs is an object selection
%   updatebyfield(lhs, rhs)

%%  Calling style
% Call-by-name, or call-by-value?
% Copy (value)
%   copyin(methodcalled, keyword, input)
% Map (name)
%   mapin(methodcalled, keyword, input)

%%  Reception of parameters
% Maps keyword in calledmethod to localval
%   receive(calledmethod, keyword, localval)

%%  Return values
% As above, either return by name or value
% Copy (value)
%   copyout(method, retval)
% Map (name)
%   mapout(method, retval)
```

G.3 ς-calculus inferences

Sets up basic inferences of type, and defines a suite of demodulators for extracting leftdot/dotright and calculating the length of scoping chains. This is useful for determining equivalence for similarity detection.

```
%%%%%
%%
%%  sigmacalc.otter
%%  Jason McC. Smith
%%  April 14, 2005
%%
%%  Definitions for the Otter system for creating inferences in the sigma-calculus.
%%

% Get the basics first
include('sigmaops.otter').

% dot clause list utilities.
list(demodulators).
  % dclen: returns the length of a dot clause chain.
  % dclen(a) = 1, dclen(a dot b) = 2,
  % dclen(class dot method dot obj dot obj2 dot m) = 5, etc
  dclen(x) = $IF(   $ATOMIC(x),
             1,
             $SUM(dclen(leftdot(x)), 1)
         ).

  % leftdot/dotright: return the proper side of the dot clause
  % Since dot is currently left-associative, only leftdot has any utility
  leftdot(xleft dot xright) = xleft.
  $ATOMIC(x) -> leftdot(x) = x.
  dotright(xleft dot xright) = xright.

  % dciseq: Determines if two clauses are equivalent.  $OCCURS would also return $T
  % for: $OCCURS(a dot b, a dot b dot c) which is useful, but not *equivalence*.
  dciseq(x, y) = $AND($OCCURS(x, y), $EQ(dclen(x), dclen(y))).

  % dcissub: x is a proper left subclause of y
  dcissub(x, y) = $AND($OCCURS(x, y), $NE(dclen(x), dclen(y))).
end_of_list.

formula_list(usable).
  % These next two should really be part of the INH block of rules, conceptually.
  % declares propagation
  all SubType SuperType op1 (
    (SubType inh SuperType) &
    (SuperType declares op1) ->

    (SubType declares op1)
  ).

  % defines propagation
  all SubType SuperType op1 (
    (SubType inh SuperType) &
    (SuperType defines op1) ->

    (SubType defines op1)
  ).
```

```
  % These are here because gcc produces a this element pervasively.  If that
  % were not the case, then the following two rules wouldn't be necessary.
  % Revisit fixing this in the gcc2poml tool.

  % this definition
  all Class method (
    Class defines method ->
    Class dot method dot this : Class
  ).

  % Propagation of instance fields to local method scope
  all Class field FieldClass method (
    Class dot field : FieldClass &
    Class defines method &
    $NOT($OCCURS(this, Class)) ->
    Class dot method dot this dot field : FieldClass
  ).

  % Superceded by the SUP block of rules.
% % super generation
% all SubType SuperType obj (
%   SubType inh SuperType &
%   obj : SubType ->
%   SubType declares super
% ).

% all SubType SuperType obj (
%   SubType inh SuperType &
%   obj : SubType ->
%   (obj dot super) : SuperType
% ).

  % Inheritance transitivity
  all SuperClass SubClass Grandparent (
    SubClass inh SuperClass &
    SuperClass inh Grandparent ->
    SubClass inh Grandparent).

end_of_list.
```

G.4 ρ-calculus operators

Defines the reliance operators of Section 4.2 in Otter format.

```
%%%%%
%%
%%  rhoops.otter
%%  Jason McC. Smith
%%  v1.1, August 30, 2005
%%
```

```
%%  Operator definitions for rho-calculus and the reliance operators.
%%

%%%%%
%%
%%  Context operator
% || stmt ||_{method}  => method iscontextfor stmt
op( 670, xfy, iscontextfor ).
%op( 670, xfy, protoiscontextfor ).


%%%%%
%%
%%  Reliance operators
%%  Four basic forms
%%
op( 660, xfy, mu ).   % method -> method
op( 660, xfy, phi ).  % method -> field
op( 660, xfy, sigma ).  % field -> method
op( 660, xfy, kappa ).  % field -> field

%% proto forms for S block (sigma binding)
%op( 660, xfy, protomu ).   % method -> method
%op( 660, xfy, protophi ).  % method -> field
%op( 660, xfy, protosigma ).  % field -> method
%op( 660, xfy, protokappa ).  % field -> field

%%%%%
%%
%%  Specialization operators
%%
%% Crude ascii of relop notation:
%%
%%    x.y x = leftdot (dis)similarity, y = dotright (dis)similarity
%%   <
%%    op  op = one of mu, phi, sigma, kappa
%%
%% Transforms into one of following relops by:  oplxry
%% If there is no l or r entry, then it is indeterminate (ie, o instead of + or -
%% in the notation)

%% mu and kappa both have the same type of terminal selectors on lhs and rhs, so
%% they can have both leftdot and dotright (dis)similarities.  phi and sigma forms,
%% however, have differing terminal selectors on the dotright, (field/method or
%% method/field), so having a dotright signifier doesn't make a lot of sense.

op( 660, xfy, mulsrs ).
op( 660, xfy, mulsrd ).
op( 660, xfy, muldrs ).
op( 660, xfy, muldrd ).
op( 660, xfy, muls ).
op( 660, xfy, muld ).
op( 660, xfy, murs ).
```

```
op( 660, xfy, murd ).

op( 660, xfy, kappalsrs ).
op( 660, xfy, kappalsrd ).
op( 660, xfy, kappaldrs ).
op( 660, xfy, kappaldrd ).
op( 660, xfy, kappals ).
op( 660, xfy, kappald ).
op( 660, xfy, kappars ).
op( 660, xfy, kappard ).

op( 660, xfy, phils ).
op( 660, xfy, phild ).

op( 660, xfy, sigmals ).
op( 660, xfy, sigmald ).
```

G.5 ρ-calculus inferences

The transitivities as outlined in Section 4.3.

```
%%%%%
%%
%%  rhocalc.otter
%%  Jason McC. Smith
%%  April 14, 2005
%%
%%  Definitions for the Otter system for creating inferences in the rho-calculus.
%%

% Get the basics first
include('sigmacalc.otter').
include('rhoops.otter').

% Get the INH and SUP rule blocks that were created
include('otherblocks.otter').

%%% Relops rules
formula_list(usable).

%%%%%
%%
%%  Similarity Specialization Derivations
%%
%%  mu and kappa both have 8 possible, 2 for leftdot (dis)sim, 2 for dotright
%%  (dis)sim, and then 4 combinatorials of those.
%%  phi and sigma just have 2 for leftdot (dis)similarity
%%%%
% mu
all ob1 ob2 mu1 mu2 (
  ((ob1 dot mu1) mu (ob2 dot mu2)) &
  ($ID(mu1, mu2)) ->
```

```
  ((ob1 dot mu1) murs (ob2 dot mu2))
).

all ob1 ob2 mu1 mu2 (
  ((ob1 dot mu1) mu (ob2 dot mu2)) &
  ($NOT($ID(mu1, mu2))) ->
  ((ob1 dot mu1) murd (ob2 dot mu2))
).

all ob1 ob2 mu1 mu2 (
  ((ob1 dot mu1) mu (ob2 dot mu2)) &
  ($TRUE(dciseq(ob1, ob2))) ->
  ((ob1 dot mu1) muls (ob2 dot mu2))
).

all ob1 ob2 mu1 mu2 (
  ((ob1 dot mu1) mu (ob2 dot mu2)) &
  ($NOT(dciseq(ob1, ob2))) ->
  ((ob1 dot mu1) muld (ob2 dot mu2))
).

all ob1 ob2 mu1 mu2 (
  ((ob1 dot mu1) muls (ob2 dot mu2)) &
  ($ID(mu1, mu2)) ->
  ((ob1 dot mu1) mulsrs (ob2 dot mu2))
).

all ob1 ob2 mu1 mu2 (
  ((ob1 dot mu1) muls (ob2 dot mu2)) &
  ($NOT($ID(mu1, mu2))) ->
  ((ob1 dot mu1) mulsrd (ob2 dot mu2))
).

all ob1 ob2 mu1 mu2 (
  ((ob1 dot mu1) muld (ob2 dot mu2)) &
  ($ID(mu1, mu2)) ->
  ((ob1 dot mu1) muldrs (ob2 dot mu2))
).

all ob1 ob2 mu1 mu2 (
  ((ob1 dot mu1) muld (ob2 dot mu2)) &
  ($NOT($ID(mu1, mu2))) ->
  ((ob1 dot mu1) muldrd (ob2 dot mu2))
).

all ob1 ob2 mu1 mu2 (
  ((ob1 dot mu1) murs (ob2 dot mu2)) &
  ($TRUE(dciseq(ob1, ob2))) ->
  ((ob1 dot mu1) mulsrs (ob2 dot mu2))
).

all ob1 ob2 mu1 mu2 (
  ((ob1 dot mu1) murs (ob2 dot mu2)) &
```

```
  ($NOT(dciseq(ob1, ob2))) ->
  ((ob1 dot mu1) muldrs (ob2 dot mu2))
).

all ob1 ob2 mu1 mu2 (
  ((ob1 dot mu1) murd (ob2 dot mu2)) &
  ($TRUE(dciseq(ob1, ob2))) ->
  ((ob1 dot mu1) mulsrd (ob2 dot mu2))
).

all ob1 ob2 mu1 mu2 (
  ((ob1 dot mu1) murd (ob2 dot mu2)) &
  ($NOT(dciseq(ob1, ob2))) ->
  ((ob1 dot mu1) muldrd (ob2 dot mu2))
).

%%%%
% phi
all ob1 ob2 mu1 mu2 (
  ((ob1 dot mu1) phi (ob2 dot mu2)) &
  ($TRUE(dciseq(ob1, ob2))) ->
  ((ob1 dot mu1) phils (ob2 dot mu2))
).

all ob1 ob2 mu1 mu2 (
  ((ob1 dot mu1) phi (ob2 dot mu2)) &
  ($NOT(dciseq(ob1, ob2))) ->
  ((ob1 dot mu1) phild (ob2 dot mu2))
).

%%%%
% kappa
all context ob1 ob2 kappa1 kappa2 (
  (context iscontextfor ((ob1 dot kappa1) kappa (ob2 dot kappa2))) &
  ($ID(kappa1, kappa2)) ->
  (context iscontextfor ((ob1 dot kappa1) kappars (ob2 dot kappa2)))
).

all context ob1 ob2 kappa1 kappa2 (
  (context iscontextfor ((ob1 dot kappa1) kappa (ob2 dot kappa2))) &
  ($NOT($ID(kappa1, kappa2))) ->
  (context iscontextfor ((ob1 dot kappa1) kappard (ob2 dot kappa2)))
).

all context ob1 ob2 kappa1 kappa2 (
  (context iscontextfor ((ob1 dot kappa1) kappa (ob2 dot kappa2))) &
  ($TRUE(dciseq(ob1, ob2))) ->
  (context iscontextfor ((ob1 dot kappa1) kappals (ob2 dot kappa2)))
).

all context ob1 ob2 kappa1 kappa2 (
  (context iscontextfor ((ob1 dot kappa1) kappa (ob2 dot kappa2))) &
  ($NOT(dciseq(ob1, ob2))) ->
```

```
  (context iscontextfor ((ob1 dot kappa1) kappald (ob2 dot kappa2)))
).

all context ob1 ob2 kappa1 kappa2 (
  (context iscontextfor ((ob1 dot kappa1) kappals (ob2 dot kappa2))) &
  ($ID(kappa1, kappa2)) ->
  (context iscontextfor ((ob1 dot kappa1) kappalsrs (ob2 dot kappa2)))
).

all context ob1 ob2 kappa1 kappa2 (
  (context iscontextfor ((ob1 dot kappa1) kappals (ob2 dot kappa2))) &
  ($NOT($ID(kappa1, kappa2))) ->
  (context iscontextfor ((ob1 dot kappa1) kappalsrd (ob2 dot kappa2)))
).

all context ob1 ob2 kappa1 kappa2 (
  (context iscontextfor ((ob1 dot kappa1) kappald (ob2 dot kappa2))) &
  ($ID(kappa1, kappa2)) ->
  (context iscontextfor ((ob1 dot kappa1) kappaldrs (ob2 dot kappa2)))
).

all context ob1 ob2 kappa1 kappa2 (
  (context iscontextfor ((ob1 dot kappa1) kappald (ob2 dot kappa2))) &
  ($NOT($ID(kappa1, kappa2))) ->
  (context iscontextfor ((ob1 dot kappa1) kappaldrd (ob2 dot kappa2)))
).

all context ob1 ob2 kappa1 kappa2 (
  (context iscontextfor ((ob1 dot kappa1) kappars (ob2 dot kappa2))) &
  ($TRUE(dciseq(ob1, ob2))) ->
  (context iscontextfor ((ob1 dot kappa1) kappalsrs (ob2 dot kappa2)))
).

all context ob1 ob2 kappa1 kappa2 (
  (context iscontextfor ((ob1 dot kappa1) kappars (ob2 dot kappa2))) &
  ($NOT(dciseq(ob1, ob2))) ->
  (context iscontextfor ((ob1 dot kappa1) kappaldrs (ob2 dot kappa2)))
).

all context ob1 ob2 kappa1 kappa2 (
  (context iscontextfor ((ob1 dot kappa1) kappard (ob2 dot kappa2))) &
  ($TRUE(dciseq(ob1, ob2))) ->
  (context iscontextfor ((ob1 dot kappa1) kappalsrd (ob2 dot kappa2)))
).

all context ob1 ob2 kappa1 kappa2 (
  (context iscontextfor ((ob1 dot kappa1) kappard (ob2 dot kappa2))) &
  ($NOT(dciseq(ob1, ob2))) ->
  (context iscontextfor ((ob1 dot kappa1) kappaldrd (ob2 dot kappa2)))
).

%%%%
% sigma
```

```
all context ob1 ob2 mu1 mu2 (
  (context iscontextfor ((ob1 dot mu1) sigma (ob2 dot mu2))) &
  ($TRUE(dciseq(ob1, ob2))) ->
  (context iscontextfor ((ob1 dot mu1) sigmals (ob2 dot mu2)))
).

all context ob1 ob2 mu1 mu2 (
  (context iscontextfor ((ob1 dot mu1) sigma (ob2 dot mu2))) &
  ($NOT(dciseq(ob1, ob2))) ->
  (context iscontextfor ((ob1 dot mu1) sigmald (ob2 dot mu2)))
).


% Generalizations
% Undo all the hard work above, and make sure we can break down a
% chain of specialization as well as build it up.

%% mu
all lhs rhs (lhs mulsrs rhs -> lhs muls rhs).
all lhs rhs (lhs mulsrs rhs -> lhs murs rhs).
all lhs rhs (lhs mulsrd rhs -> lhs muls rhs).
all lhs rhs (lhs mulsrd rhs -> lhs murd rhs).
all lhs rhs (lhs muldrs rhs -> lhs muld rhs).
all lhs rhs (lhs muldrs rhs -> lhs murs rhs).
all lhs rhs (lhs muldrd rhs -> lhs muld rhs).
all lhs rhs (lhs muldrd rhs -> lhs murd rhs).
all lhs rhs (lhs muls rhs -> lhs mu rhs).
all lhs rhs (lhs muld rhs -> lhs mu rhs).
all lhs rhs (lhs murs rhs -> lhs mu rhs).
all lhs rhs (lhs murd rhs -> lhs mu rhs).

%% kappa
all lhs rhs (lhs kappalsrs rhs -> lhs kappals rhs).
all lhs rhs (lhs kappalsrs rhs -> lhs kappars rhs).
all lhs rhs (lhs kappalsrd rhs -> lhs kappals rhs).
all lhs rhs (lhs kappalsrd rhs -> lhs kappard rhs).
all lhs rhs (lhs kappaldrs rhs -> lhs kappald rhs).
all lhs rhs (lhs kappaldrs rhs -> lhs kappars rhs).
all lhs rhs (lhs kappaldrd rhs -> lhs kappald rhs).
all lhs rhs (lhs kappaldrd rhs -> lhs kappard rhs).
all lhs rhs (lhs kappals rhs -> lhs kappa rhs).
all lhs rhs (lhs kappald rhs -> lhs kappa rhs).
all lhs rhs (lhs kappars rhs -> lhs kappa rhs).
all lhs rhs (lhs kappard rhs -> lhs kappa rhs).

%% phi
all lhs rhs (lhs phils rhs -> lhs phi rhs).
all lhs rhs (lhs phild rhs -> lhs phi rhs).

%% sigma
all lhs rhs (lhs sigmals rhs -> lhs sigma rhs).
all lhs rhs (lhs sigmald rhs -> lhs sigma rhs).
```

```
%%%%%
%%
%%  Conversion and transitivity rules.
%%
%%  Numbering scheme comes from notebook II, pg 126  Will likely need to
%%  modify to match whatever numbering scheme ends up in dissertation
%%
%%%%%
% mu
% Mu Call
% mu.A: calls -> mu
all obj obj2 method method2 (
  obj dot method iscontextfor calls(obj2 dot method2) ->
  obj dot method mu obj2 dot method2
).

% Mu Transitivity
% mu.B: transitivities
all lhs rhs bridge (
  lhs mu bridge &
  bridge mu rhs ->
  lhs mu rhs
).

% Mu Sigma
% mu.C: sigma -> mu
all context lhs rhs (
  context iscontextfor (lhs sigma rhs) ->
  context mu rhs
).

% Mu Update
% mu.D: update -> mu of rhs
all context lhs rhs (
  context iscontextfor updatebycall(lhs, rhs) ->
  context mu rhs
).

%%%%%%
% phi
% Phi Parameter
% phi.A.i: Parameter use (raw call)
all context calledmethod keyword input (
  context iscontextfor mapin(calledmethod, keyword, input) |
  context iscontextfor copyin(calledmethod, keyword, input) ->
  context phi input
).

% Phi Parameter (Update)
% phi.A.ii: Parameter use (call in update rhs)
all context calledmethod keyword input lhs (
  context iscontextfor updatebycallparam(lhs, mapin(calledmethod, keyword, input)) |
```

```
  context iscontextfor updatebycallparam(lhs, copyin(calledmethod, keyword, input)) ->
  context phi input
).

% phi.B: update
% Phi Update1
% phi.B.i: update -> lhs field reliance on field
all context lhs rhs (
  context iscontextfor updatebyfield(lhs, rhs) ->
  context phi lhs
).

% Phi Update2
% phi.B.ii: update -> rhs field reliance
all context lhs rhs (
  context iscontextfor updatebyfield(lhs, rhs) ->
  context phi rhs
).

% Phi Update1
% phi.B.iii: update -> lhs field reliance on call
all context lhs rhs (
  context iscontextfor updatebycall(lhs, rhs) ->
  context phi lhs
).

% phi.C: kappa -> phi
% Phi Kappa1
% phi.C.i: lhs
all context lhs rhs (
  context iscontextfor (lhs kappa rhs) ->
  context phi lhs
).

% Phi Kappa2
% phi.C.ii: rhs
all context lhs rhs (
  context iscontextfor (lhs kappa rhs) ->
  context phi rhs
).

% Phi Sigma
% phi.D: sigma -> phi
all context lhs rhs (
  context iscontextfor (lhs sigma rhs) ->
  context phi lhs
).

%%%%%%%%
% sigma
% Sigma Update
% sigma.A.i: update w/ mapped return value
all context target calledmethod retval (
```

```
  context iscontextfor updatebycall(target, calledmethod) &
  mapout(calledmethod, retval) ->
  context iscontextfor (target sigma calledmethod)
).

% Sigma Update
% sigma.A.ii: update w/ copied return value
all context target calledmethod retval (
  context iscontextfor updatebycall(target, calledmethod) &
  copyout(calledmethod, retval) ->
  context iscontextfor (target sigma calledmethod)
).

% Sigma Call-by-Name
% sigma.B.i: updated mapped field parameter
all context calledmethod keyword input localvar newval (
  context iscontextfor mapin(calledmethod, keyword, input) &
  receive(calledmethod, keyword, localval) &
  calledmethod iscontextfor updatebyfield(localval, newval) ->
  context iscontextfor (input sigma calledmethod)
).

% Sigma Call-by-Name
% sigma.B.ii: updated mapped call parameter
all context calledmethod keyword input localvar secondarymethod (
  context iscontextfor mapin(calledmethod, keyword, input) &
  receive(calledmethod, keyword, localval) &
  calledmethod iscontextfor updatebycall(localval, secondarymethod) ->
  context iscontextfor (input sigma calledmethod)
).

%%%%%%%%
% kappa
% Kappa Return Value
% kappa.A.i: update w/ mapped return value
all context target calledmethod retval (
  context iscontextfor updatebycall(target, calledmethod) &
  mapout(calledmethod, retval) ->
  context iscontextfor target kappa retval
).

% Kappa Return Value
% kappa.A.ii: update w/ copied return value
all context target calledmethod retval (
  context iscontextfor updatebycall(target, calledmethod) &
  copyout(calledmethod, retval) ->
  context iscontextfor target kappa retval
).

% Kappa Update
% kappa.B: update by field -> kappa
all context target val (
  context iscontextfor updatebyfield(target, val) ->
```

```
  context iscontextfor (target kappa val)
).

% Kappa Transitivity
% kappa.C: transitivities
all context lhs bridge rhs (
  context iscontextfor lhs kappa bridge &
  context iscontextfor bridge kappa rhs ->
  context iscontextfor lhs kappa rhs
).

% Kappa Input Parameter
% kappa.D: mapping in param
all context calledmethod kw param input target (
  context iscontextfor updatebycallparam(target, mapin(calledmethod, kw, input) ) &
  receive(calledmethod, kw, param) ->
  calledmethod iscontextfor param kappa input
).

% Inter-relop transitivities:
% Phi Mu/Phi
% IR.mpp: mu + phi => phi
all lhs rhs bridge (
  lhs mu bridge &
  bridge phi rhs ->
  lhs phi rhs
).

% Phi Phi/Kappa
% IR.pkp: phi + kappa => phi
all lhs rhs context bridge (
  lhs phi bridge &
  context iscontextfor bridge kappa rhs ->
  lhs phi rhs
).

% Mu Phi/Kappa
% IR.pkm: phi + kappa => mu (context)
all lhs rhs context somefield (
  lhs phi rhs &
  context iscontextfor rhs kappa somefield ->
  lhs mu context
).

% Mu Phi/Sigma1
% IR.psm.i: phi + sigma => mu
all lhs rhs context somefield (
  lhs phi rhs &
  context iscontextfor rhs sigma calledmethod ->
  lhs mu calledmethod
).

% Mu Phi/Sigma2
```

```
% IR.psm.ii: phi + sigma => mu (context)
all lhs rhs context somefield (
  lhs phi rhs &
  context iscontextfor rhs sigma calledmethod ->
  lhs mu context
).

% Sigma Sigma/Mu
% IR.sms: sigma + mu => sigma
all context target calledmethod secondarymethod (
  context iscontextfor target sigma calledmethod &
  calledmethod mu secondarymethod ->
  context iscontextfor target sigma secondarymethod
).

% Kappa Sigma/Phi
% IR.spk: sigma + phi => kappa
all context target calledmethod somefield (
  context iscontextfor target sigma calledmethod &
  calledmethod phi somefield ->
  context iscontextfor target kappa somefield
).

% Sigma Kappa/Sigma
% IR.kss: kappa + sigma => sigma
all context target bridge calledmethod (
  context iscontextfor target kappa bridge &
  context iscontextfor bridge sigma calledmethod ->
  context iscontextfor target sigma calledmethod
).
end_of_list.
```

G.6 INH and SUB blocks

Python tool for generating the INH and SUB blocks on demand. The depth of the dotright scoping chains is estimated prior to SPQR runs and given as an input.

```
# Generates supplementary blocks of rules for rhocalc as implemented in Otter.
# Most of these rules are highly similar, with just minor permutations.
# Also, the blocks themselves are quite similar.  It made sense to auto-generate
# these as necessary.

import sys

# If True, emit the production of sigma binding rules for Otter, and tweak the other
# rule blocks to support this binding as well.
enableSigmaBindings = False


class ObjDesc:
  """Base class for types of objects being used in the rules"""
```

```
  def __init__(self):
    self.pre = ""
    self.post = ""
    self.desc = ""
    self.quals = []
    self.excl = ""

class External(ObjDesc):
  """Elements that are not part of object instances - no 'this'"""
  def __init__(self, tag, src):
    self.pre = tag
    self.post = tag
    self.quals = [tag]
    self.desc = "ext" + tag
    self.excl = "\n\t$NOT($OCCURS(this, " + tag + ")) & "

class Raw(ObjDesc):
  """Objects that are raw, no enclosing scope"""
  def __init__(self, tag, src):
    self.pre = "class dot method dot this"
    self.post = src.postraw
    self.quals = ['class', 'method']
    self.desc = "raw" + tag
    self.excl = ""

class Norm(Raw):
  """Normal elements - those that are scoped and in instance objects"""
  def __init__(self, tag, depth, src):
    Raw.__init__(self, tag, src)
    for i in range(depth):
      arg = tag + str(i+1)
      self.pre += " dot " + arg
      self.post += " dot " + arg
      self.quals.append(arg)
    self.desc = "n" + tag + str(depth)
    self.excl = ""

def NormList(tag, depth, src):
  """Generate a list of Norm elements to a depth as given"""
  rtnlist = []
  for i in range(depth):
    rtnlist.append(Norm(tag, i+1, src))
  return rtnlist

def uniquify(l):
  d = {}
  for item in l:
    d[item] = {}
  return d.keys()

class ValTypes:
  """These are the values provided for the two elements, fields and methods.  Scoped
   in this class to allow access by getattr"""
```

```
  class FieldTypes(list):
    def __init__(self, tag, depth, src):
      self.extend(NormList(tag, depth, src))
      self.append(Raw(tag, src))
      self.append(External(tag, src))

  class MethodTypes(list):
    """No Raw elements since methods cannot be outside all objects"""
    def __init__(self, tag, depth, src):
      self.extend(NormList(tag, depth, src))
      self.append(External(tag, src))


class RuleBlock:
  """Generator of a rule block.  Uses subclassing to set the required strings for use
   in the emission methods."""
  def __init__(self):
    self.blockname = ""
    self.precontext = ""
    self.postcontext = ""
    self.basequals = []
    self.bridgerules = "\t$T"

  def simple_block(self, depth):
    """Allows blocks to emit rules that are simple and don't require the below
     machinations."""
    pass

  # Used for sigma and kappa
  def two_op_context_infix_block(self, desc):
    """Two operands, in a context, infix operator"""
    op1types = getattr(ValTypes, desc['op1']['type'])(desc['op1']['tag'], \
           desc['op1']['depth'], self)
    op2types = getattr(ValTypes, desc['op2']['type'])(desc['op2']['tag'], \
           desc['op2']['depth'], self)
    for op1 in op1types:
      for op2 in op2types:
        quals = self.basequals[:]
        quals.extend(op1.quals)
        quals.extend(op2.quals)
        quals = uniquify(quals)
        print "%", ".".join([self.blockname, desc['op'], op1.desc, op2.desc])
        print 'all ' + " ".join(quals) + ' ('
        print "\t" + self.precontext + " (" + op1.pre + ") " + desc['op'] + \
          " (" + op2.pre + ') &' + op1.excl + op2.excl
        print self.bridgerules + " ->"
        print "\t" + self.postcontext + " (" + op1.post + ") " + desc['op'] + \
          " (" + op2.post + ') '
        print ").\n"

  # used for mu and phi
  def two_op_infix_block(self, desc):
    """Two operand infix operator"""
```

```
  op1types = getattr(ValTypes, desc['op1']['type'])(desc['op1']['tag'], \
        desc['op1']['depth'], self)
  op2types = getattr(ValTypes, desc['op2']['type'])(desc['op2']['tag'], \
        desc['op2']['depth'], self)
  for op1 in op1types:
    for op2 in op2types:
      quals = self.basequals[:]
      quals.extend(op1.quals)
      quals.extend(op2.quals)
      quals = uniquify(quals)
      print "%", ".".join([self.blockname, desc['op'], op1.desc, op2.desc])
      print 'all ' + " ".join(quals) + ' ('
      if enableSigmaBindings:
        print "\t", op1.pre, 'proto' + desc['op'], op2.pre, '&' + op1.excl + \
          op2.excl
      else:
        print "\t", op1.pre, desc['op'], op2.pre, '&' + op1.excl + op2.excl
      print self.bridgerules + " ->"
      print "\t", op1.post, desc['op'], op2.post
      print ").\n"

# Used for updatebycall and updatebyfield
def two_op_context_prefix_block(self, desc):
  print '%%'
  print '%  updateby{call,field}'
  print '%'
  if enableSigmaBindings:
    print '%  context protoiscontextfor updateby{call,field}( TARGET, NEWVAL )'
  else:
    print '%  context iscontextfor updateby{call,field}( TARGET, NEWVAL )'
  print '%\n'

  op1types = getattr(ValTypes, desc['op1']['type'])(desc['op1']['tag'], \
        desc['op1']['depth'], self)
  op2types = getattr(ValTypes, desc['op2']['type'])(desc['op2']['tag'], \
        desc['op2']['depth'], self)
  for op1 in op1types:
    for op2 in op2types:
      quals = self.basequals[:]
      quals.extend(op1.quals)
      quals.extend(op2.quals)
      quals = uniquify(quals)
      print "%", ".".join([self.blockname, desc['op'], op1.desc, op2.desc])
      print 'all ' + " ".join(quals) + ' ('
      print "\t" + self.precontext,  desc['op'] + "(" + op1.pre + ", " +  \
        op2.pre + ') &' + op1.excl + op2.excl
      print self.bridgerules + " ->"
      print "\t" + self.postcontext,  desc['op'] + "(" + op1.post + ", " + \
        op2.post + ') '
      print ").\n"

# Mapout/copyout
def allout_block(self, depth):
```

```
  op1types = ValTypes.FieldTypes('retval', depth, self)
  for style in ['mapout', 'copyout']:
    print '%%'
    print '%  ' + style
    print '%'
    print '%  ' + style + '(method, VAL)'
    print '%\n'

    for op1 in op1types:
      quals = self.basequals[:]
      quals.extend(op1.quals)
      quals = uniquify(quals)
      print "%", ".".join([self.blockname, style, op1.desc])
      print 'all ' + " ".join(quals) + ' ('
      print "\t" + style + "(", self.premethod, ",", op1.pre, ') &' + op1.excl
      print self.bridgerules + " ->"
      print "\t" + style + "(", self.postmethod, ",", op1.post, ')'
      print ").\n"

# various updatebycallparameter
def ubcp_block(self, depth):
  print '%%'
  print '%  updatebycallparam'
  print '%'

  op1types = ValTypes.FieldTypes('target', depth, self)
  op2types = ValTypes.MethodTypes('called', depth, self)
  op3types = ValTypes.FieldTypes('input', depth, self)
  for style in ['copyin', 'mapin']:
    print '%  ' + style
    if enableSigmaBindings:
      print '%  context protoiscontextfor updatebycallparam( TARGET,', \
        style + '( CALLED, kw, INPUT ) )'
    else:
      print '%  context iscontextfor updatebycallparam( TARGET,', \
        style + '( CALLED, kw, INPUT ) )'
    print '%\n'
    for op1 in op1types:
      for op2 in op2types:
        for op3 in op3types:
          quals = self.basequals[:]
          quals.extend(op1.quals)
          quals.extend(op2.quals)
          quals.extend(op3.quals)
          quals = uniquify(quals)
          print "%", ".".join([self.blockname, 'ubcp', style, op1.desc, \
                  op2.desc, op3.desc])
          print 'all ' + " ".join(quals) + ' kw ('
          print "\t" + self.precontext + " ( updatebycallparam( " + op1.pre + \
            ", " + style + "( " + op2.pre + ', kw, ' + op3.pre + ' ))) &' + \
            op1.excl + op2.excl + op3.excl
          print self.bridgerules + " ->"
          print "\t" + self.postcontext + " ( updatebycallparam( " + op1.post + \
```

```
              ", " + style + "( " + op2.post + ', kw, ' + op3.post + ' ))) '
            print ").\n"


# Sigma bindings block - deprecated
# -If you look back up the list of *_block methods in RuleBlock, you'll see a lot
# -of commented out lines with 'proto' in them.  They all go with sigma binding.
# -If for any reason this needs to be turned back on, turn those on as well.
# Creates rules governing sigma binding from a type to an object.
class SBlock(RuleBlock):
  def __init__(self):
    RuleBlock.__init__(self)
    self.blockname = 'S'
    self.basequals = ['class', 'method', 'obj', 'cs']
    self.premethod = 'class dot method'
    self.postmethod = 'obj dot method'
    self.precontext = 'class dot method protoiscontextfor'
    self.postcontext = 'obj dot method iscontextfor'
    self.bridgerules = "\tCreateObject(cs, class, obj) &\n\tclass defines method "
    self.postraw = 'obj'

  def simple_block(self, depth):
    print '%%%%%'
    print '%%'
    print '%% Sigma binding - this is the magic trick'
    print '%% There are several places where we can perform this, and each is a'
    print '%% different form.'
    print '%% Each rule block contains a template for the rule with variant points', \
      'given as all caps.'
    print '%% Naming conventions for rule names, based on variant points:'
    print "%%   no modifier:  'normal' form: Class.method.this.something"
    print '%%   raw:      raw this: Class.method.this'
    print '%%   ext:      something (no this, indicates an external object)'
    print '%%   del:      something (no this, indicates a delegated call to',\
      'external object)'
    print '%% As a general rule, methods are norm or del, fields are norm, ext, or',\
      'raw, unless otherwise'
    print '%% noted.'
    print '%%'
    print '%% First up, method declaration/definition'

    print '%  S.dec'
    print 'all class method obj cs ('
    print '%\tCreateObject(cs, class, obj) &'
    print '\tobj : class &'
    print '\tclass declares method ->'
    print '\tobj declares method'
    print ').\n'

    print '%  S.def'
    print 'all class method obj cs ('
    print '%\tCreateObject(cs, class, obj) &'
    print '\tobj : class &'
```

```
print '\tclass defines method ->'
print '\tobj defines method'
print ').\n'

# Bring forward any methods that were inherited from superclasses unchanged
# Technically not needed: INH.def + S.def do the same thing in two steps
print '%  S.inhdef'
print 'all class method subclass cs obj ('
print '\tCreateObject( cs, subclass, obj ) &'
print '\tclass defines method &'
print '\tsubclass inherits class dot method ->'
print '\tobj defines method'
print ').\n'

print '%% CreateObject propagation'
print '%'
print '%  S.CreateObj (normal)'
print 'all cs class obj classobj ctor class2 obj2 ('
print '\tCreateObject(cs, class, obj) &'
print '\tCreateObject(classobj dot ctor, class2,',\
  'classobj dot ctor dot this dot obj2) &'
print '\tclassobj isclassobjfor class &'
print '\tobj : class &'
print '\t$NOT( dciseq( leftdot(cs), class2 ) ) &'
print '\t$NOT( dciseq( leftdot(cs), classobj ) ) &'
print '\t$NOT( dciseq( class, class2 ) ) ->'
print '\tCreateObject(cs, class2, obj dot obj2)'
print ').\n'

print 'all cs class obj classobj ctor class2 obj2 blah ('
print '\tCreateObject(someclass dot m, class, obj) &'
print '\tCreateObject(classobj dot ctor, class2,',\
  'classobj dot ctor dot this dot blah dot obj2) &'
print '\tclassobj isclassobjfor class &'
print '\t$NOT( dciseq( someclass, class2 ) ) &'
print '\t$NOT( dciseq( someclass, classobj ) ) &'
print '\tobj : class &'
print '\t$NOT( dciseq( class, class2 ) ) ->'
print '\tCreateObject(cs, class2, obj dot blah dot obj2)'
print ').\n'

print '%  S.CreateObjSelf (cycle (incl. Self))'
print '%  Instead of just having the same class name, using OCCURS grabs'
print '%  instances where a class is creating an instance of a *subclass* - the'
print '%  same issue arises. Possible to eliminate the leftdot in the dciseq'
print '%  clause, and instead have CrObj(class dot cs, class, obj) initially?'
print 'all cs class obj classobj ctor class2 obj2 ('
print '\tobj : class &'
print '\tCreateObject(cs, class, obj) &'
print '\tCreateObject(classobj dot ctor, class2,',\
  'classobj dot ctor dot this dot obj2) &'
print '\tclassobj isclassobjfor class &'
print '\t( $TRUE(dciseq( leftdot(cs), class2 )) |'
```

```
    print '\t  $TRUE(dciseq( leftdot(cs), classobj )) |'
    print '\t  $TRUE(dciseq( class, class2)) ) ->'
    print '\tCreateObjectSelf(cs, class2, obj dot obj2)'
    print ').\n'

    print 'all cs class obj classobj ctor class2 obj2 blah ('
    print '\tobj : class &'
    print '\tCreateObject(cs, class, obj) &'
    print '\tCreateObject(classobj dot ctor, class2,',\
      'classobj dot ctor dot this dot blah dot obj2) &'
    print '\tclassobj isclassobjfor class &'
    print '\t( $TRUE(dciseq( leftdot(cs), class2 )) |'
    print '\t  $TRUE(dciseq( leftdot(cs), classobj )) |'
    print '\t  $TRUE(dciseq( class, class2)) ) ->'
    print '\tCreateObjectSelf(cs, class2, obj dot blah dot obj2)'
    print ').\n'

    print '%  S.receive'
    print '%% No other form can exist - a parameter is guaranteed to be bounded '
    print '%% by this, and guaranteed to be wrapped by the method'
    print 'all class method param cs obj keyword ('
    print '\treceive(class dot method, keyword,',\
      'class dot method dot this dot param) &'
    print '\tclass defines method &'
    print '\tCreateObject(cs, class, obj) ->'
    print '\treceive(obj dot method, keyword, obj dot param)'
    print ').\n'

# Inheritance block
# These rules make sure that methods defined in a superclass are also accessible
# to the subclass
class INHBlock(RuleBlock):
  def __init__(self):
    RuleBlock.__init__(self)
    self.blockname = 'INH'
    self.basequals = ['class', 'method', 'subclass']
    self.premethod = 'class dot method'
    self.postmethod = 'subclass dot method'
    if enableSigmaBindings:
      self.precontext = 'class dot method protoiscontextfor'
      self.postcontext = 'subclass dot protomethod iscontextfor'
    else:
      self.precontext = 'class dot method dot this dot method iscontextfor'
      self.postcontext = 'subclass dot method dot this dot method iscontextfor'
    self.bridgerules = "\tsubclass inherits class dot method &\n" + \
      "\tclass defines method "
    self.postraw = 'subclass dot method dot this'

  def simple_block(self, depth):
    print '%  INH.def'
    print 'all class subclass method ('
    print '\tclass defines method &'
    print '\tsubclass inherits class dot method ->'
```

```
    print '\tsubclass defines method'
    print ').\n'

    print '%  INH.receive'
    print 'all class subclass method param keyword ('
    print '\treceive(class dot method, keyword,',\
      'class dot method dot this dot param) &'
    print '\tclass defines method &'
    print '\tsubclass inherits class dot method ->'
    print '\treceive(subclass dot method, keyword,',\
      'subclass dot method dot this dot param)'
    print ').\n'


# Super block
# These rules ensure that an instance of a superclass' method that is brought
# down to a subclass intact (in the presence of an overriding subclass
# definition of the same method) is correctly renamed, and that all rules
# associated with the superclass' definition get copied to the subclass appropriately.
class SUPBlock(RuleBlock):
  def __init__(self):
    RuleBlock.__init__(self)
    self.blockname = 'SUP'
    self.basequals = ['class', 'method', 'subclass', 'newmethodname']
    self.premethod = 'class dot method'
    self.postmethod = 'subclass dot newmethodname'
    if enableSigmaBindings:
      self.precontext = 'class dot method protoiscontextfor'
      self.postcontext = 'subclass dot newmethodname protoiscontextfor'
    else:
      self.precontext = 'class dot method iscontextfor'
      self.postcontext = 'subclass dot newmethodname iscontextfor'
    self.bridgerules = "\tdirectlycallfrom_to_as(subclass, class dot method," + \
      " newmethodname) "
    self.postraw = 'subclass dot newmethodname dot this'

  def simple_block(self, depth):
    pass


def gen_block(block, depth):
  block.simple_block(depth)

  rule_data = {'sigma': {'type': 'two_op_context_infix_block', \
          'op1': {'type': 'FieldTypes', 'tag': 'lhs', 'depth': depth}, \
          'op2': {'type': 'MethodTypes', 'tag': 'rhs', 'depth': depth} }, \
        'kappa': {'type': 'two_op_context_infix_block', \
          'op1': {'type': 'FieldTypes', 'tag': 'lhs', 'depth': depth}, \
          'op2': {'type': 'FieldTypes', 'tag': 'rhs', 'depth': depth} }, \
        'mu': {'type': 'two_op_infix_block', \
          'op1': {'type': 'MethodTypes', 'tag': 'lhs', 'depth': depth}, \
          'op2': {'type': 'MethodTypes', 'tag': 'rhs', 'depth': depth} }, \
        'phi': {'type': 'two_op_infix_block', \
```

```
          'op1': {'type': 'MethodTypes', 'tag': 'lhs', 'depth': depth}, \
          'op2': {'type': 'FieldTypes', 'tag': 'rhs', 'depth': depth} }, \
        'updatebycall': {'type': 'two_op_context_prefix_block', \
          'op1': {'type': 'FieldTypes', 'tag': 'target', 'depth': depth}, \
          'op2': {'type': 'MethodTypes', 'tag': 'called', 'depth': depth} }, \
        'updatebyfield': {'type': 'two_op_context_prefix_block', \
          'op1': {'type': 'FieldTypes', 'tag': 'target', 'depth': depth}, \
          'op2': {'type': 'FieldTypes', 'tag': 'newval', 'depth': depth} } \
      }

  for rule in rule_data:
    desc = rule_data[rule]
    desc.update({'op': rule})
    getattr(block, desc['type'])(desc)

  block.ubcp_block(depth)
  block.allout_block(depth)

depth = int(sys.argv[1])
print "% Generated on "
print "% Depth =", depth
print "formula_list(usable).\n"
if enableSigmaBindings:
  blocks = [INHBlock(), SUPBlock(), SBlock()]
else:
  blocks = [INHBlock(), SUPBlock()]

for b in blocks:
  gen_block(b, depth)
print "end_of_list."
```

Appendix H

XSLTs

H.1 POML2Otter.xsl

```
<xsl:stylesheet version="1.0" xmlns:cl="http://www.cs.unc.edu/~smithja/changelog"
  xmlns:dc="http://purl.org/dc/elements/1.1/" xmlns:poml="http://www.cs.unc.edu/~smithja/spqr"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="text"/>

<!--  <xsl:variable name="debug"></xsl:variable> -->
  <xsl:variable name="debug">1</xsl:variable>

  <xsl:variable name="delim"> dot </xsl:variable>
  <xsl:variable name="underscore">_</xsl:variable>

  <xsl:template match="xsl:documentation">
      <xsl:text>% </xsl:text>
      <xsl:value-of select="."/>
<!--    Convert to this with XSLT 2.0 aware engines.
      <xsl:value-of select="replace(node(), '\n', '\n%')"/> -->
  </xsl:template>

  <!-- The scopeable names that are likely to be inside named spaces -->
  <xsl:template match="poml:method/poml:name | poml:field/poml:name | poml:object/poml:name |
      poml:class/poml:name | poml:parameter/poml:name">
    <xsl:choose>
      <!-- If a scope exists, use it.  Otherwise, get the parent's name. -->
      <xsl:when test="poml:scope">
        <xsl:apply-templates select="poml:scope"/>
      </xsl:when>
      <xsl:otherwise>
        <!-- If the parent name exists, grab it.  Added to stop spurious $delim
           prepended to top level objects/classes -->
        <xsl:if test="../../poml:name">
          <xsl:apply-templates select="../../poml:name"/>
          <xsl:if test="../poml:static">
            <xsl:value-of select="$staticobjtag"/>
          </xsl:if>
          <xsl:value-of select="$delim"/>
        </xsl:if>
      </xsl:otherwise>
    </xsl:choose>
    <xsl:value-of select="normalize-space(node())"/>
  </xsl:template>

  <!-- The scopeable names that are *unable* be inside named spaces -->
  <xsl:template match="poml:fulfilledBy | poml:type | poml:name | poml:objectname |
      poml:fieldname | poml:methodname | poml:lhs | poml:rhs | poml:isclassobjfor">
    <xsl:call-template name="process-scoped-name"/>
```

```
</xsl:template>

<!-- Pulling this out lets us call it from elsewhere (like handling parent and
   inheritsmethod) -->
<xsl:template name="process-scoped-name">
  <xsl:apply-templates select="poml:scope"/>
  <xsl:value-of select="normalize-space(node())"/>
  <xsl:if test="poml:static">
    <xsl:value-of select="$staticobjtag"/>
  </xsl:if>
</xsl:template>

<xsl:template name="flatten-scoped-name">
  <xsl:apply-templates select="poml:scope">
    <xsl:with-param name="usedelim" select="$underscore"/>
  </xsl:apply-templates>
  <xsl:value-of select="normalize-space(node())"/>
</xsl:template>

<xsl:template match="poml:scope">
  <xsl:param name="usedelim" select="$delim"/>
  <xsl:choose>
    <xsl:when test="poml:scope">
      <xsl:apply-templates select="poml:scope"/>
      <xsl:value-of select="normalize-space(node())"/>
      <xsl:if test="./poml:static">
        <xsl:value-of select="$staticobjtag"/>
      </xsl:if>
      <xsl:value-of select="$usedelim"/>
    </xsl:when>
    <xsl:otherwise>
      <xsl:value-of select="normalize-space(node())"/>
      <xsl:if test="./poml:static">
        <xsl:value-of select="$staticobjtag"/>
      </xsl:if>
      <xsl:value-of select="$usedelim"/>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>

<xsl:template match="poml:object">
  <xsl:if test="$debug">
    <xsl:text>%-----  Object </xsl:text>
    <xsl:apply-templates select="poml:name"/>
    <xsl:text> file=</xsl:text>
    <xsl:value-of select="@sourcefile"/>
    <xsl:text> line=</xsl:text>
    <xsl:value-of select="@line"/>
    <xsl:text>&#xA;</xsl:text>
  </xsl:if>
  <xsl:value-of select="$pretag"/>
  <xsl:apply-templates select="poml:name"/>
  <xsl:text> : </xsl:text>
  <xsl:apply-templates select="poml:type"/>
  <xsl:value-of select="$posttag"/>
  <!-- Tie classobj to class -->
    <xsl:if test="poml:isclassobjfor">
      <xsl:value-of select="$pretag"/>
```

```
      <xsl:apply-templates select="poml:name"/>
      <xsl:text> isclassobjfor </xsl:text>
      <xsl:apply-templates select="poml:isclassobjfor"/>
      <xsl:value-of select="$posttag"/>
    </xsl:if>
  <!-- Methods -->
    <xsl:apply-templates select="poml:method"/>
  <!-- Fields -->
  <xsl:apply-templates select="poml:field"/>
</xsl:template>

<xsl:template match="poml:class">
  <!-- Create a class-level object for ctors/dtors/static elements -->
  <!-- Handled by POML generation tool.  Really. -->
  <xsl:if test="$debug">
    <xsl:text>%-----  Class </xsl:text>
    <xsl:apply-templates select="poml:name"/>
    <xsl:text> file=</xsl:text>
    <xsl:value-of select="@sourcefile"/>
    <xsl:text> line=</xsl:text>
    <xsl:value-of select="@line"/>
    <xsl:text>&#xA;</xsl:text>
  </xsl:if>
  <!--  No longer needed since object *IS* handled by POML generation tool.
  <xsl:value-of select="$pretag"/>
  <xsl:apply-templates select="poml:name"/>
  <xsl:value-of select="$staticobjtag"/>
  <xsl:text> : </xsl:text>
  <xsl:apply-templates select="poml:name"/>
  <xsl:value-of select="$statictypetag"/>
  <xsl:value-of select="$posttag"/>
  -->
  <!-- Inheritance items -->
  <xsl:for-each select="poml:parent">
    <xsl:value-of select="$pretag"/>
    <xsl:apply-templates select="../poml:name"/>
    <xsl:text> inh </xsl:text>
    <xsl:call-template name="process-scoped-name"/>
    <xsl:value-of select="$posttag"/>
  </xsl:for-each>
  <xsl:for-each select="poml:inheritedmethod">
    <xsl:value-of select="$pretag"/>
    <xsl:apply-templates select="../poml:name"/>
    <xsl:text> inherits </xsl:text>
    <xsl:call-template name="process-scoped-name"/>
    <xsl:value-of select="$posttag"/>
  </xsl:for-each>
  <xsl:for-each select="poml:directlycalls">
    <xsl:value-of select="$pretag"/>
    <xsl:text>directcallfrom_to_as( </xsl:text>
    <xsl:apply-templates select="../poml:name"/>
    <xsl:text>, </xsl:text>
    <xsl:call-template name="process-scoped-name"/>
    <xsl:text>, </xsl:text>
    <xsl:value-of select="poml:as"/>
    <xsl:text> )</xsl:text>
    <xsl:value-of select="$posttag"/>
  </xsl:for-each>
```

```
    <!-- Methods -->
    <xsl:apply-templates select="poml:method">
      <xsl:with-param name="inclass" select="true()"/>
    </xsl:apply-templates>
    <xsl:apply-templates select="poml:field"/>
  </xsl:template>

  <xsl:template match="poml:method">
    <xsl:param name="inclass" select="false()"/>
    <xsl:if test="$debug">
      <xsl:text>     %-----   Method </xsl:text>
      <xsl:apply-templates select="poml:name"/>
      <xsl:if test="poml:static">
        <xsl:text> (Static)</xsl:text>
      </xsl:if>
      <xsl:text> file=</xsl:text>
      <xsl:value-of select="@sourcefile"/>
      <xsl:text> line=</xsl:text>
      <xsl:value-of select="@line"/>
      <xsl:text>&#xA;</xsl:text>
    </xsl:if>

<!--
     If the method is abstract, emit an AbstractInterface pattern.
       If it turns out that this is a double with an explicitly described
       pattern in the original POML, that's okay with Otter.
    -->
    <xsl:choose>
      <xsl:when test="poml:abstract">
        <xsl:value-of select="$pretag"/>
        <xsl:text>AbstractInterface( </xsl:text>
        <xsl:apply-templates select="../poml:name"/>
        <xsl:text>, </xsl:text>
        <xsl:value-of select="poml:name"/>
        <xsl:text> )</xsl:text>
        <xsl:value-of select="$posttag"/>
      </xsl:when>
      <xsl:otherwise>
        <xsl:value-of select="$pretag"/>
        <xsl:apply-templates select="../poml:name"/>
        <xsl:if test="./poml:static">
          <xsl:value-of select="$staticobjtag"/>
        </xsl:if>
        <xsl:text> defines </xsl:text>
        <xsl:value-of select="normalize-space(poml:name/node())"/>
        <xsl:value-of select="$posttag"/>

        <!-- Emit the parameters -->
        <xsl:apply-templates select="poml:parameter"/>

        <!-- Emit the body rules in the order they're encountered -->
        <xsl:apply-templates select="poml:field | poml:calls | poml:uses | poml:update |
            poml:result">
          <xsl:with-param name="inclass" select="$inclass"/>
        </xsl:apply-templates>
      </xsl:otherwise>
    </xsl:choose>
```

```
  </xsl:template>

  <xsl:template match="poml:parameter">
    <!-- If the parameter is unnamed, just skip -->
    <xsl:if test="poml:name">
      <xsl:value-of select="$pretag"/>
      <xsl:text>receive( </xsl:text>
      <xsl:apply-templates select="../poml:name"/>
      <xsl:text>, </xsl:text>
      <xsl:value-of select="poml:keyword"/>
      <xsl:text>, </xsl:text>
      <xsl:apply-templates select="poml:name"/>
      <xsl:text> ) </xsl:text>
      <xsl:value-of select="$posttag"/>
    </xsl:if>
  </xsl:template>

  <xsl:template match="poml:calls">
    <xsl:param name="inclass" select="false()"/>
    <xsl:value-of select="$pretag"/>
    <xsl:text>( </xsl:text>
    <xsl:apply-templates select="../poml:name"/>
<!--
    <xsl:if test="$inclass">
      <xsl:value-of select="$delim"/>
      <xsl:text>this</xsl:text>
      <xsl:value-of select="$delim"/>
      <xsl:value-of select="../poml:name"/>
    </xsl:if>
-->
    <xsl:text> ) </xsl:text>
<!--
    <xsl:if test="$inclass">
      <xsl:text>proto</xsl:text>
    </xsl:if>
-->
    <xsl:text>mu ( </xsl:text>
    <xsl:apply-templates select="poml:objectname"/>
    <xsl:value-of select="$delim"/>
    <xsl:apply-templates select="poml:methodname"/>
    <xsl:text> )</xsl:text>
    <xsl:value-of select="$posttag"/>
    <xsl:apply-templates select="poml:callingparameter">
      <xsl:with-param name="inclass" select="$inclass"/>
    </xsl:apply-templates>
  </xsl:template>

  <xsl:template match="poml:uses">
    <xsl:param name="inclass" select="false()"/>
    <xsl:value-of select="$pretag"/>
    <xsl:text>( </xsl:text>
    <xsl:apply-templates select="../poml:name"/>
    <xsl:if test="$inclass">
      <xsl:value-of select="$delim"/>
      <xsl:text>this</xsl:text>
      <xsl:value-of select="$delim"/>
      <xsl:value-of select="../poml:name"/>
    </xsl:if>
```

```
    <xsl:text> ) </xsl:text>
<!--
    <xsl:if test="$inclass">
      <xsl:text>proto</xsl:text>
    </xsl:if>
-->
    <xsl:text>phi ( </xsl:text>
    <xsl:apply-templates select="poml:objectname"/>
    <xsl:value-of select="$delim"/>
    <xsl:apply-templates select="poml:fieldname"/>
    <xsl:text> )</xsl:text>
    <xsl:value-of select="$posttag"/>
  </xsl:template>

  <xsl:template match="poml:callingparameter">
    <xsl:param name="inclass" select="false()"/>
    <xsl:value-of select="$pretag"/>
    <xsl:apply-templates select="../../poml:name"/>
    <xsl:if test="$inclass">
      <xsl:value-of select="$delim"/>
      <xsl:text>this</xsl:text>
      <xsl:value-of select="$delim"/>
      <xsl:value-of select="../../poml:name"/>
    </xsl:if>
    <xsl:text> </xsl:text>
<!--
    <xsl:if test="$inclass">
      <xsl:text>proto</xsl:text>
    </xsl:if>
-->
    <xsl:text>iscontextfor </xsl:text>
    <xsl:value-of select="poml:callby"/>
    <xsl:text>in( </xsl:text>
    <xsl:apply-templates select="../poml:objectname"/>
    <xsl:value-of select="$delim"/>
    <xsl:apply-templates select="../poml:methodname"/>
    <xsl:text>, </xsl:text>
    <xsl:value-of select="poml:keyword"/>
    <xsl:text>, </xsl:text>
    <xsl:apply-templates select="poml:name"/>
    <xsl:text> )</xsl:text>
    <xsl:value-of select="$posttag"/>
  </xsl:template>

  <xsl:template match="poml:update">
    <xsl:param name="inclass" select="false()"/>
    <xsl:value-of select="$pretag"/>
    <xsl:apply-templates select="../poml:name"/>
    <xsl:if test="$inclass">
      <xsl:value-of select="$delim"/>
      <xsl:text>this</xsl:text>
      <xsl:value-of select="$delim"/>
      <xsl:value-of select="../poml:name"/>
    </xsl:if>
    <xsl:choose>
    <xsl:when test="poml:fromcall">
      <xsl:text> </xsl:text>
<!--
```

```
      <xsl:if test="$inclass">
        <xsl:text>proto</xsl:text>
      </xsl:if>
-->
      <xsl:text>iscontextfor updatebycall( </xsl:text>
      <xsl:apply-templates select="poml:lhs"/>
      <xsl:text>, </xsl:text>
      <xsl:apply-templates select="poml:rhs/poml:objectname"/>
      <xsl:value-of select="$delim"/>
      <xsl:value-of select="poml:rhs/poml:methodname"/>
      <xsl:text> )</xsl:text>
      <xsl:value-of select="$posttag"/>
      <xsl:for-each select="poml:rhs/poml:parameter">
        <xsl:value-of select="$pretag"/>
        <xsl:apply-templates select="../poml:objectname"/>
        <xsl:value-of select="$delim"/>
        <xsl:value-of select="../poml:methodname"/>
        <!-- Consider reusing callingparameter up above, but with params for
           updatebycallparam et al-->
        <xsl:text> </xsl:text>
<!--
        <xsl:if test="$inclass">
          <xsl:text>proto</xsl:text>
        </xsl:if>
-->
        <xsl:text>iscontextfor updatebycallparam( </xsl:text>
        <xsl:apply-templates select="../../poml:lhs"/>
        <xsl:text>, </xsl:text>
        <xsl:value-of select="poml:callby"/>
        <xsl:text>in( </xsl:text>
        <xsl:apply-templates select="../poml:objectname"/>
        <xsl:value-of select="$delim"/>
        <xsl:apply-templates select="../poml:methodname"/>
        <xsl:text>, </xsl:text>
        <xsl:value-of select="poml:keyword"/>
        <xsl:text>, </xsl:text>
        <xsl:apply-templates select="poml:name"/>
        <xsl:text> )</xsl:text>
        <xsl:value-of select="$posttag"/>
       </xsl:for-each>
<!--
       <xsl:apply-templates select="poml:rhs"/>
       <xsl:text> )</xsl:text>
-->
    </xsl:when>
    <xsl:otherwise>
      <xsl:text> </xsl:text>
<!--
      <xsl:if test="$inclass">
        <xsl:text>proto</xsl:text>
      </xsl:if>
-->
      <xsl:text>iscontextfor updatebyfield( </xsl:text>
      <xsl:apply-templates select="poml:lhs"/>
      <xsl:text>, </xsl:text>
      <xsl:apply-templates select="poml:rhs"/>
      <xsl:text> )</xsl:text>
      <xsl:value-of select="$posttag"/>
```

```
    </xsl:otherwise>
    </xsl:choose>
  </xsl:template>

<!--
  <xsl:template match="poml:lhs | poml:rhs">
    <xsl:apply-templates select="poml:name"/>
  </xsl:template>
-->

  <xsl:template match="poml:result">
    <xsl:value-of select="$pretag"/>
    <xsl:value-of select="poml:passby"/>
    <xsl:text>out( </xsl:text>
<!--    <xsl:apply-templates select="../../poml:method/poml:name"/>      -->
    <xsl:apply-templates select="../poml:name"/>
    <xsl:text>, </xsl:text>
    <xsl:apply-templates select="poml:name"/>
    <xsl:text> )</xsl:text>
    <xsl:value-of select="$posttag"/>
  </xsl:template>

  <xsl:template match="poml:field">
    <xsl:if test="$debug">
      <xsl:text>     %-----   Field </xsl:text>
      <xsl:apply-templates select="poml:name"/>
      <xsl:if test="poml:static">
        <xsl:text> (Static)</xsl:text>
      </xsl:if>
      <xsl:text> file=</xsl:text>
      <xsl:value-of select="@sourcefile"/>
      <xsl:text> line=</xsl:text>
      <xsl:value-of select="@line"/>
      <xsl:text>&#xA;</xsl:text>
    </xsl:if>
    <xsl:value-of select="$pretag"/>
    <xsl:apply-templates select="poml:name"/>
    <xsl:text> : </xsl:text>
    <xsl:apply-templates select="poml:type"/>
    <xsl:value-of select="$posttag"/>
  </xsl:template>

  <xsl:template match="poml:pattern">
    <xsl:value-of select="$pretag"/>
    <xsl:value-of select="poml:name"/>
    <xsl:text>( </xsl:text>
    <xsl:for-each select="poml:role">
      <xsl:apply-templates select="poml:fulfilledBy"/>
      <xsl:if test="position() != last()">
        <xsl:text>, </xsl:text>
      </xsl:if>
    </xsl:for-each>
    <xsl:text> )</xsl:text>
    <xsl:value-of select="$posttag"/>
  </xsl:template>

  <xsl:template match="poml:resultpattern">
    <xsl:value-of select="poml:name"/>
```

```
    <xsl:text>( </xsl:text>
    <xsl:for-each select="poml:role">
      <xsl:apply-templates select="poml:name"/>
      <xsl:if test="position() != last()">
        <xsl:text>, </xsl:text>
      </xsl:if>
    </xsl:for-each>
    <xsl:text> )&#xA;).&#xA;</xsl:text>
  </xsl:template>

  <xsl:template match="poml:atprule">
    <xsl:value-of select="$pretag"/>
    <xsl:value-of select="node()"/>
    <xsl:value-of select="$posttag"/>
  </xsl:template>

</xsl:stylesheet>
```

H.2 POML2OtterFacts.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:poml="http://www.cs.unc.edu/~smithja/spqr"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform" xmlns:dc="http://purl.org/dc/elements/1.1/">
    <xsl:output method="text"/>

<!--
    <xsl:variable name="pretag">( </xsl:variable>
    <xsl:variable name="posttag"> ).&#xA;</xsl:variable> -->
    <xsl:variable name="pretag"></xsl:variable>
    <xsl:variable name="posttag">.&#xA;</xsl:variable>
    <xsl:variable name="staticobjtag">__ClassObj</xsl:variable>
    <xsl:variable name="statictypetag">__ClassObjType</xsl:variable>

    <xsl:include href="POML2Otter.xsl"/>

    <xsl:template match="/poml:system">
        <xsl:text>%%%%&#xA;%&#xA;</xsl:text>
        <xsl:text>%&#x9;</xsl:text>
        <xsl:value-of select="dc:title"/>
        <xsl:text>&#xA;</xsl:text>
        <xsl:text>%&#x9;</xsl:text>
        <xsl:value-of select="dc:creator"/>
        <xsl:text>&#xA;</xsl:text>
        <xsl:text>%&#x9;</xsl:text>
        <xsl:value-of select="dc:date"/>
        <xsl:text>&#xA;</xsl:text>
        <xsl:text>%&#x9;</xsl:text>
        <xsl:value-of select="dc:description"/>
<!--  Convert to this for XLST 2.0 aware engines:
        <xsl:value-of select="replace(dc:description, '\n', '\n%')"/> -->
        <xsl:text>&#xA;</xsl:text>
        <xsl:text>%&#x9;Generated by POML2OtterFacts.xsl&#xA;</xsl:text>
        <xsl:text>%&#xA;%%%&#xA;&#xA;&#xA;</xsl:text>
        <xsl:text>list(sos).&#xA;&#xA;</xsl:text>
        <xsl:apply-templates select="poml:class | poml:object | poml:pattern"/>
        <xsl:text>&#xA;&#xA;end_of_list.&#xA;</xsl:text>
    </xsl:template>
```

```
</xsl:stylesheet>
```

H.3 POML2OtterSearch.xsl

```
<xsl:stylesheet version="1.0" xmlns:cl="http://www.cs.unc.edu/~smithja/changelog"
  xmlns:dc="http://purl.org/dc/elements/1.1/" xmlns:poml="http://www.cs.unc.edu/~smithja/spqr"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:output method="text"/>

    <xsl:variable name="prepend">x</xsl:variable>
    <xsl:variable name="delim"> dot </xsl:variable>

    <xsl:template match="/poml:system">
        <xsl:text>%%%%&#xA;%&#xA;</xsl:text>
        <xsl:text>%&#x9;</xsl:text>
        <xsl:value-of select="dc:title"/>
        <xsl:text>&#xA;</xsl:text>
        <xsl:text>%&#x9;</xsl:text>
        <xsl:value-of select="dc:creator"/>
        <xsl:text>&#xA;</xsl:text>
        <xsl:text>%&#x9;</xsl:text>
        <xsl:value-of select="dc:date"/>
        <xsl:text>&#xA;</xsl:text>
        <xsl:text>%&#x9;</xsl:text>
        <xsl:value-of select="dc:description"/>
        <xsl:text>&#xA;</xsl:text>
        <xsl:text>%&#x9;Generated by POML2OtterSearch.xsl&#xA;</xsl:text>
        <xsl:text>%&#xA;%%%%&#xA;&#xA;&#xA;</xsl:text>
        <xsl:text>formula_list(usable).&#xA;&#xA;</xsl:text>
        <xsl:apply-templates select="poml:resultpattern"/>
        <xsl:text>&#xA;&#xA;end_of_list.&#xA;</xsl:text>
    </xsl:template>
    <xsl:template match="poml:resultpattern">
        <xsl:text>all </xsl:text>
        <!-- Once with only spaces -->
        <xsl:apply-templates select="poml:quantifiers/poml:quantifier"/>
        <xsl:text>(-(</xsl:text>
        <xsl:value-of select="poml:name"/>
        <xsl:text>(</xsl:text>
        <!-- Once with separating commas -->
        <xsl:for-each select="poml:role">
            <xsl:apply-templates select="poml:fulfilledBy"/>
            <xsl:if test="position() != last()">
                <xsl:text>, </xsl:text>
            </xsl:if>
        </xsl:for-each>
        <xsl:text>))).</xsl:text>
    </xsl:template>

    <xsl:template match="poml:fulfilledBy">
        <xsl:apply-templates select="poml:scope"/>
        <xsl:value-of select="$prepend"/>
        <xsl:value-of select="normalize-space(node())"/>
    </xsl:template>

    <xsl:template match="poml:quantifier">
```

```
        <xsl:apply-templates select="poml:scope"/>
        <xsl:value-of select="$prepend"/>
        <xsl:value-of select="normalize-space(node())"/>
        <xsl:text> </xsl:text>
    </xsl:template>

    <xsl:template match="poml:scope">
        <xsl:choose>
            <xsl:when test="poml:scope">
                <xsl:apply-templates select="poml:scope"/>
                <xsl:value-of select="$prepend"/>
                <xsl:value-of select="normalize-space(node())"/>
                <xsl:value-of select="$delim"/>
            </xsl:when>
            <xsl:otherwise>
                <xsl:value-of select="$prepend"/>
                <xsl:value-of select="normalize-space(node())"/>
                <xsl:value-of select="$delim"/>
            </xsl:otherwise>
        </xsl:choose>
    </xsl:template>


</xsl:stylesheet>
```

Bibliography

Abadi, M. and Cardelli, L. (1996). *A Theory of Objects*. Springer-Verlag New York, Inc.

Abrahams, D., Dimov, P., Gregor, D., Hinnant, H. E., Hommel, A., and Meredith, A. (2005). Impact of rvalue reference on the library. ISO C++ committee paper ISO/IEC JTC/SC22/WG21/N1771.

Accredited Standards Committee X3 (Information Processing Systems) – Working Group 21 (1996). Working paper for draft proposed international standard for information systems – programming language C++. Technical Report X3J16/96-0225 WG21/N1043, American National Standards Institute.

Agerholm, S. (1991). Mechanizing program verification in HOL. In Archer, M., Joyce, J. J., Levitt, K. N., and Windley, P. J., editors, *Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and its Applications*, pages 208–222. IEEE Computer Society Press.

Agesen, O., Bak, L., Chambers, C., Chang, B.-W., Hölze, U., Maloney, J., Smith, R. B., Ungar, D., and Wolczko, M. (1993). *The Self 3.0 Programmer's Reference Manual*. Sun Microsystems.

Aho, A. V., Sethi, R., and Ullman, J. D. (1986). *Compilers: Principles, Techniques and Tools*. Addison-Wesley.

Alexander, C., Ishikawa, S., and Silverstein, M. (1977). *A Pattern Language: Towns, Building, Construction*. Oxford Univ Press.

Alexander, C. W. (1964). *Notes on the Synthesis of Form*. Oxford Univ Press. Fifteenth printing, 1999.

Alexander, C. W. (1979). *A Timeless Way of Building*. Oxford Univ Press.

Allen, R. and Garlan, D. (1997). A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249.

ANSI/IEEE, editor (1983). *An American National Standard IEEE Standard Glossary of Software Engineering Terminology*. ANSI/IEEE Standard 729.

Apple (1993). *The NewtonScript programming language*. Apple Computer, Inc.

Arango, G. (1986). TMM: Software maintenance by transformation. *IEEE Software*, 3(3):27–39.

Banker, R. D., Datar, S. M., Kemerer, C. F., and Zweig, D. (1993). Software complexity and maintenance costs. *Communications of the ACM*, 36(11):81–94.

Barnard, C. I. (1960). *The Functions of the Executive*. Harvard University Press, Cambridge, MA. First published 1938, 14th printing.

Basili, V. R., Briand, L. C., and Melo, W. L. (1996). A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761.

Beck, K. (1997). *Smalltalk Best Practice Patterns*. Prentice Hall.

Benbasat, I. and Weber, R. (1996). Research commentary: Rethinking "diversity" in information systems research. *Information Systems Research*, 7(4):389–399.

Benbasat, I. and Zmud, R. W. (1999). Empirical research in information systems: The practice of relevance. *Management Information Sciences Quarterly*, 23(1):3–16.

Bergin, J. (2003). Teaching polymorphism with elementary design patterns. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 167–169. ACM Press.

Beyer, D., Noack, A., and Lewerentz, C. (2005). Efficient relational calculation for software analysis. *IEEE Transactions on Software Engineering*, 31(2):137–149.

Bieman, J. M. and Kang, B.-K. (1995). Cohesion and reuse in an object-oriented system. In *Proc. of the ACM Symposium on Software Reusability, SSR'95*, pages 259–262. Reprinted in ACM Software Engineering Notes, Aug 1995.

Bieman, J. M. and Kang, B.-K. (1998). Measuring design-level cohesion. *IEEE Transactions on Software Engineering*, 24(2):111–124.

Bieman, J. M. and Ott, L. (1994). Measuring functional cohesion. *IEEE Transactions on Software Engineering*, 20(8):644–657.

Bosch, J. (1996). Language support for design patterns. In *Proc. TOOLS Europe '96*.

Bosch, J. (1998). Design patterns as language constructs. *Journal of Object Oriented Programming*, 1(2):18–52.

Briand, L. and Daly, J. (1997). A unified framework for cohesion measurement in object-oriented systems. In *Proc. of the Fourth Conf. on METRICS'97*, pages 43–53.

Brooks Jr., F. P. (1982). *The Mythical Man-Month*. Addison-Wesley.

Brown, K. (2000). Design reverse-engineering and automated design pattern detection in smalltalk. Master's thesis, North Carolina State University.

Brown, W. J., Malveau, R. C., Brown, W. H., Hays III, W. M., and Mowbray, T. J. (1998). *Anti-Patterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons.

Budinsky, F., Finnie, M., Vlissides, J., and Yu, P. (1996). Automatic code generation from design patterns. *IBM Systems Journal*, 35(2).

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. (1996). *Pattern-Oriented System Architecture: A System of Patterns*, volume 1 of *Wiley Series in Software Design Patterns*. John Wiley & Sons.

Chambers, C. (1993). The cecil language: Specification and rationale. Technical Report TR-93-03-05, University of Washington.

Chan, T., Chung, S. L., and Ho, T. H. (1996). An economic model to estimate software rewriting and replacement times. *IEEE Transactions on Software Engineering*, 22(8).

Charette, R. N., Adams, K. M., and White, M. B. (1997). Managing risk in software maintenance. *IEEE Software*, 14(3):43–50.

Charlton, C. C., Leng, P. H., and Wilkinson, D. M. (1988). A microprogram meta-disassembler. *Microprocessing and Microprogramming*, 22(4):255–262.

Chen, J. and Lu, J. (1993). A new metric for object-oriented design. *Information and Software Technology*, pages 232–240.

Chidamber, S. R. and Kemerer, C. F. (1991). Towards a metrics suite for object-oriented design. In *Proc. OOPSLA '91*, pages 197–211. ACM.

Chidamber, S. R. and Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493. cohesion/LCOM.

Cifuentes, C. and Gough, K. J. (1995). Decompilation of binary programs. *Software - Practice and Experience*, 25(7):811–829.

Clarke, E. M., Emerson, E. A., and Sistla, A. P. (1986). Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263.

Clarke, E. M., Grumberg, O., and Peled, D. A. (2000). *Model Checking*. The MIT Press.

Coplien, J. (1998). C++ idioms. In *Proceedings of the Third European Conference on Pattern Languages of Programming and Computing*.

Coplien, J. O. (1996a). Code patterns. *The Smalltalk Report*.

Coplien, J. O. (1996b). Idioms and patterns as architecture literature. *IEEE Software*. Special Issue on Objects, Patterns and Architectures.

Davidson, J. D. and Inc., A. C. (2002). *Learning Cocoa with Objective-C*. O'Reilly & Associates, 2nd edition.

DeMarco, T. (1996). The role of software development methodologies: past, present, and future. In *Proc. of the 18th Intl. Conf. on Software Engineering*, page 2. Mar 25-29, 1996.

Demeyer, S., Ducasse, S., and Nierstrsz, O. (2000). Finding refactoring via change metrics. In *Proceedings of the conference on Object-oriented programming, systems, languages, and applications*, pages 166–177. ACM Press.

Deransart, P., Ed-Dbali, A., and Cervoni, L. (1996). *Prolog: The Standard: Reference Manual*. Springer-Verlag Berlin, Inc.

Dimov, P., Dawes, B., and Colvin, G. (2003). A proposal to add general purpose smart pointers to the library technical report. ISO C++ committee paper ISO/IEC JTC/SC22/WG21/N1450.

Eden, A. H. (1997). Giving "the quality" a name. *Journal of Object Oriented Programming*.

Eden, A. H. (1999). Symbolic logic specification of design patterns: Part 2: The language. Presented at Formal design Techniques Group, SICS – Swedish Institute of Computer Science, March 24, 1999.

Eden, A. H. (2000). *Precise Specification of Design Patterns and Tool Support in their Application*. PhD thesis, Tel Aviv University, Tel Aviv, Israel.

Eden, A. H. (2001). Formal specification of object-oriented design. In *Proc. Int'l. Conf. Multidisciplinary Design in Engineering CSME-MDE*.

Eden, A. H., Gil, J., Hirshfeld, Y., and Yehudai, A. (1999). Towards a mathematical foundation for design patterns. Technical report, Department of Information Technology, Uppsala University.

Eden, A. H. and Hirshfeld, Y. (1999). Specification of recurring motifs in o-o software architecture. Presentation, August 24, 1999. Programming Technology Lab, Department of Computer Science, Vrije Universiteit Brussels, Belgium.

Eden, A. H., Yehudai, A., and Gil, J. (1997). Precise specification and automatic application of design patterns. In *1997 International Conference on Automated Software Engineering (ASE '97)*. IEEE.

Elmasri, R. and Navathe, S. B. (2000). *Fundamentals of Database Systems*. Addison-Wesley, third edition.

Elrad, T., Filman, R. E., and Bader, A. (2001). Aspect-oriented programming: Introduction. *Communications of the ACM*, 11(10):29–32.

Emmerik, M. and Waddington, T. (2004). Using a decompiler for real-world source recovery. In *Proc. of 11th Working Conf on Reverse Engineering*, pages 27–36.

Ernst, M. D., Cockrell, J., Griswold, W. G., and Notkin, D. (2001). Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123.

Etzkorn, L., Bansiya, J., and Davis, C. (1999). Design and code complexity metrics for oo classes. *Journal of Object Oriented Programming*.

Fenton, N. (1994). Software measurement: A necessary scientific basis. *IEEE Transactions on Software Engineering*, 20(3):199–206.

Feynman, R. P. (1988). *Between Quantum and Cosmos: Studies and Essays in Honor of John Archibald Wheeler*, chapter Quantum Mechanical Computers, pages 523–+. Princeton University Press.

Feynman, R. P. (1999). Simulating physics with computers. *Feynman and computation: exploring the limits of computers*, pages 133–153.

Fischer, C. and LeBlanc, R. (1991). *Crafting A Compiler with C*. Addison-Wesley.

Florijn, G., Meijers, M., and van Winsen, P. (1997). Tool support for object-oriented patterns. In Askit, M. and Matsuoka, S., editors, *Proc. of the 11th European Conf. on Object Oriented Programming - ECOOP'97*. Springer-Verlag, Berlin.

Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

French, A. P. and Taylor, E. F. (1978). *An Introduction to Quantum Physics*. The M.I.T. Introductory Physics Series. W. W. Norton & Company, Inc.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns*. Addison Wesley.

Goldberg, A. (1995). What should we teach? In *Addendum to the proceedings of the 10th annual conference on Object-oriented programming systems, languages, and applications (Addendum)*, pages 30–37. ACM Press.

Goldberg, A. and Robson, D. (1983). *Smalltalk-80: The Language and its Implementation*. Addison-Wesley.

Goodhue, D. L., Wybo, M. D., and Kirsch, L. J. (1992). The impact of data integration on the costs and benefits of information systems. *Management Information Sciences Quarterly*, pages 293–311.

Gosling, J., Joy, B., Steele, G., and Bracha, G. (2005). *The Java^{TM} Language Specification*. Addison-Wesley Professional, 3rd edition.

Grossman, D., Morrisett, G., and Zdancewic, S. (2000). Syntactic type abstraction. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(6):1037–1080. multi-agent calculus.

Group, O. M. (2005). MOF 2.0/XMI mapping specification, v2.1. Technical report, Object Management Group.

Grünwald, P. (2005). A tutorial introduction to the Minimum Description Length principle. In Grünwald, P., Myung, I. L., and Pitt, M., editors, *Advances in Minimum Description Length: Theory and Applications*. MIT Press.

Gupta, B. S. (1997). A critique of cohesion measures in the object-oriented paradigm. Master's thesis, Michigan Technological University.

Hansen, M. H. and Yu, B. (2001). Model selection and the principle of minimum description length. *Journal of the American Statistical Association*, 96(454):746–774.

Hatton, L. (1997). Reexamining the fault density-component size connection. *IEEE Software*, 14(2):89–97.

Hayduk, L. A. (1987). *Structural Equation Modeling with LISREL*. Johns Hopkins Press.

Hecht, M. S. (1977). *Flow Analysis of Computer Programs*. Elsevier Science Inc., New York, NY.

Hitz, M. and Montazeri, B. (1995). Measuring coupling and cohesion in object-oriented systems. In *Proceedings of ISACC'95*, pages 10–21, Insitut für Angewandte Informatik und Informationssysteme, Uni versity of Vienna, Rathaustraße 1914, A-1010 Vienna, Austria.

Hitz, M. and Montazeri, B. (1996). Chidamber and Kemerer's metrics suite: A measurement theory perspective. *IEEE Transactions on Software Engineering*, 22(4).

Jordan, H. and Alaghband, G. (2002). *Foundations of Parallel Computing*. Prentice Hall.

Jul, E., Raj, R. K., Tempero, E. D., Levy, H. M., Black, A. P., and Hutchinson, N. M. (1991). Emerald: A general-purpose programming language. *Software Practice and Experience*, 21(1):91–118.

Kalman, J. A. (2001). *Automated Reasoning with Otter*. Rinton Press.

Kang, B.-K. and Bieman, J. M. (1996a). Design-level cohesion measures: Derivation, comparison, and applications. In *Proc. 20th Intl. Computer Software and Applications Conf. (COMPSAC'96)*, pages 92–97.

Kang, B.-K. and Bieman, J. M. (1996b). Using design cohesion to visualize, quantify and restructure software. In *Eighth Int'l Conf. Software Eng. and Knowledge Eng., SEKE '96*.

Karstu, S. (2999). An examination of the behavior of slice-based cohesion measures. Master's thesis, Minnesota Technological University.

Kay, M. (2004). *XSLT 2.0 Programmer's Reference*. Wrox.

Kemerer, C. (1987). An empirical validation of software cost estimation models. *Communications of the ACM*, 30(5):416–429.

Kennedy, K. (1981). *Program Flow Analysis: Theory and Applications*, chapter A Survey of Data Flow Analysis Techniques, pages 5–54. Prentice-Hall, Englewood Cliffs, New Jersey.

Klarlund, N., Koistinen, J., and Schwartzbach, M. I. (1996). Formal design constraints. In *Proc. of the 11th Ann. Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pages 370–383.

Knuth, D. E. (1984). Literate programming. *The Computer Journal*, 27(2):97 –111.

Kristensen, B. B. (1994). Complex associations: abstractions in object-oriented modeling. In *Proc of the ninth annual conference on Object-oriented programming systems, language, and applications*, pages 272–286. ACM Press.

Kruchten, P. (2003). *The Rational Unified Process: An Introduction*. Addison-Wesley Professional, 3rd edition.

Lakhotia, A. (1993). Rule-based approach to computing module cohesion. In *Proc. of the 15th Int'l Conf. Software Eng.*, pages 35–44. May 17-21, 1993.

Lakos, J. (1996). *Large-Scale Object-Oriented Software Design in C++*. Addison-Wesley.

LaLonde, W. and Pugh, J. (1994). Gathering metric information using metalevel facilities. *Journal of Object Oriented Programming*, 7(1):33–37.

Leitsch, A. (1997). *The Resolution Calculus*. Springer-Verlag.

Lientz, B. P. and Swanson, E. B. (1980). *Software Maintenance Management*. Addison-Wesley, Reading, MA.

Lovatt, H. C., Sloane, A. M., and Verity, D. R. (2005). A Pattern Enforcing Compiler (PEC) for Java: Using the compiler. In Hartmann, S. and Stumptner, M., editors, *Conferences in Research and Practice in Information Technology*, volume 43. Appeared at The Second Asia-Pacific Conference on Conceptual Modeling (APCCM2005).

Lutz, R. (2002). Recovering high-level structure of software systems using a minimum description length principle. In *AICS '02: Proceedings of the 13th Irish International Conference on Artificial Intelligence and Cognitive Science*, pages 61–69, London, UK. Springer-Verlag.

Madsen, O. L., Møller-Pederson, B., and Nygaard, K. (1993). *Object-oriented Programming in the BETA language*. Addison-Wesley.

Maletic, J. I. and Marcus, A. (2000). Using latent semantic analysis to identify similarities in source code to support program understanding. In *12th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'00)*, page 46.

Maletic, J. I. and Valluri, N. (1999). Automatic software clustering via latent semantic analysis. In *14th IEEE International Conference on Automated Software Engineering (ASE'99)*, page 251.

McComb, D. and Smith, J. Y. (1991). System project failure: The heuristics of risk. *Journal of Informations Systems Management*, 8(1).

McCune, W. (1990). Otter 2.0 (theorem prover). In Stickel, M. E., editor, *Proc. of the 10th Intl Conf. on Automated Deduction*, pages 663–664.

McMillan, K. L. (1992). *Symbolic model checking - an approach to the state explosion problem*. PhD thesis, SCS, Carnegie Mellon University.

Microsoft Corporation, editor (2002). *Microsoft Visual C# .NET Language Reference*. Microsoft Press.

Microsystems, S. (2005). Javadoc 5.0. http://java.sun.com/j2se/1.5.0/docs/guide/javadoc/index.html.

Mikkonen, T. (1998). Formalizing design patterns. In *The 20th Intl Conf on Software Engineering*, pages 115–124. April 19-25.

Miller, G. A. (1957). The magical number 7 plus or minus two: Some limits on our capacity for processing information. *Psychological Review*, 63:81–97.

Minker, J., editor (1988). *Foundations of deductive databases and logic programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Monson-Haefel, R., Burke, B., and Labourey, S. (2004). *Enterprise JavaBeans*. O'Reilly & Associates, 4th edition.

Moore, I. (1996). Automatic inheritance hierarchy restructuring and method refactoring. In *Proc. of the eleventh annual conference on Object-oriented programming systems, languages, and applications*, pages 235–250. ACM Press.

Muthen, L. K. and Muthen, B. O. (2001). *Mplus: Statistical Analysis with Latent Variables: User's Guide*. Muthen & Muthen, 2nd edition.

Niere, J., Schäfer, W., Wadsack, J. P., Wendehals, L., and Welsh, J. (2002). Towards pattern-based design recovery. In *Proc. of the* 24^{th} *International Conference on Software Engineering (ICSE), Orlando, Florida, USA*, pages 338–348. ACM Press.

Niere, J., Wadsack, J. P., and Wendehals, L. (2003a). Handling large search space in pattern-based reverse engineering. In *Proc. of the* 11^{th} *International Workshop on Program Comprehension (IWPC), Portland, USA*, pages 274–279. IEEE Computer Society Press.

Niere, J., Wendehals, L., and Zündorf, A. (2003b). An interactive and scalable approach to design pattern recovery. Technical Report tr-ri-03-236, University of Paderborn, Paderborn, Germany.

NIST (1999). *Algorithms and Theory of Computation Handbook*, chapter Dictionary of Algorithms and Data Structures. CRC Press LLC.

Ó Cinnéide, M. (2001). *Automated Application of Design Patterns: A Refactoring Approach*. Ph.D. dissertation, University of Dublin, Trinity College.

Ó Cinnéide, M. and Nixon, P. (1998). Program restructuring to introduce design patterns. In *Proceedings of the Workshop on Experiences in Object-Oriented Re-Engineering, European Conference on Object-Oriented Programming, Brussels*.

Opdyke, W. F. and Johnson, R. E. (1993). Creating abstract superclasses by refactoring. In *Proc. of the Conf. on 1993 ACM Computer Science*, page 66. Feb 16-18, 1993.

Ott, L. M. (1992). Using slice profiles and metrics during software maintenance. In *Proceedings of the 10th Annual Software Reliability Symposium, Denver, June 25-26, 1992*.

Ott, L. M., Bieman, J. M., Kang, B.-K., and Mehra, B. (1995). Developing measures of class cohesion for object-oriented software. In *7th Annual Oregon Workshop on Software Metrics*.

Ott, L. M. and Thuss, J. J. (1989). The relationship between slices and module cohesion. In *Proceedings of the 11th International Conference on Software Engineering, May 15-18, 1989*.

Ott, L. M. and Thuss, J. J. (1993). Slice based metrics for estimating cohesion. In *Proceedings of the IEEE-CS International Software Metrics Symposium, Baltimore, May 21-22 1993*.

Pace, J. A. D. and Campo, M. R. (2001). Analyzing the role of aspects in software design. *Communications of the ACM*, 44(10):66–73.

Partsch, H. and Steinbrueggen, R. (1983). Program transformation systems. *ACM Computing Surveys*, 15(3):199–236.

Patel, S., Chu, W., and Baxter, R. (1992). A measure for composite module cohesion. In *Int'l Conf. Software Eng.*, pages 38–48.

Pericas-Geertsen, S. M. (2001). Sigma. http://types.bu.edu/ool-mini-seminar/sigma.html Boston University.

Perkins, J. H. and Ernst, M. D. (2004). Efficient incremental algorithms for dynamic detection of likely invariants. In *Proceedings of the ACM SIGSOFT 12th Symposium on the Foundations of Software Engineering (FSE 2004)*, pages 23–32. Newport Beach, CA.

Plaisted, D. A. (1999). *Wiley Encyclopedia of Electrical and Electronics Engineering*, volume 21, chapter Theorem Proving, pages 662–682. Wiley & Sons.

Plaisted, D. A. and Zhu, Y. (2000). Ordered semantic hyper linking. *Journal of Automated Reasoning*, 25(3):167–217.

Quatrani, T. (2002). *Visual Modeling with Rational Rose 2002 and UML*. Addison-Wesley Professional, 3rd edition.

Ramamohanarao, K. and Harland, J. (1994). An introduction to deductive database languages and systems. *The VLDB Journal*, 3(2):107–122.

Riehle, D. (1997). Composite design patterns. In *Proceedings of the 1997 ACM SIGPLAN conference on Object-oriented programming systems, languages and applications*, pages 218–228. ACM Press.

Rumbaugh, J., Jacobson, I., and Booch, G. (2004). *The Unified Modeling Language Reference Manual*. Addison-Wesley Professional, 2nd edition.

Samadzadeh, M. H. and Khan, S. J. (1994). Stability, coupling and cohesion of object-oriented software systems. In *Proc. 22nd Ann. ACM Computer Science Conf. on Scaling Up*, pages 312–319. Mar 8-10, 1994.

Santaun, P. and Prakash, A. (1994). A framework for source code search using program patterns. *IEEE Transactions on Software Engineering*, 20(6):463–475.

Schneidewind, N. F. (1987). The state of software maintenance. *IEEE Transactions on Software Engineering*, 13(3):303–310.

Sefika, M., Sane, A., and Campbell, R. H. (1996). Architecture-oriented visualization. In *Proc. of the eleventh annual conference on Object-oriented programming systems, languages, and applications*, pages 389–405. ACM Press.

Shalit, A. (1996). *The Dylan Reference Manual*. Addison-Wesley, Cambridge, MA.

Smith, J. M. (2002). An Elemental Design Pattern catalog. Technical Report TR-02-040, Univ. of North Carolina.

Smith, J. M. and Stotts, D. (2002). Elemental Design Patterns: A formal semantics for composition of OO software architecture. In *Proc. of 27th Annual IEEE/NASA Soft. Engineering Workshop*, pages 183–190.

Smith, J. M. and Stotts, D. (2003). SPQR: Flexible automated design pattern extraction from source code. In *18th IEEE Intl Conf on Automated Software Engineering*, pages 215–224.

Stansifer, R. (1995). *The Study of Programming Languages*. Prentice Hall.

Steane, A. (1988). Quantum computing. *Reports on Progress in Physics*, 61(2):117–173.

Steele, G. L. (1984). *Common LISP*. Elsevier Digital Press / Butterworth-Heinemann, 2nd (1990) edition.

Stotts, D. and Smith, J. M. (2002). Semi-automated hyperlink markup for archived video. In *Proc of ACM Hypertext 2002*, pages 105–106. ACM.

Stotts, D., Smith, J. M., and Gyllstrom, K. (2004). Distributed pair programming in facetop. In *Proceedings of XP/Agile Universe 2004*.

Stotts, D., Smith, J. M., and Jen, D. (2003). The vis-a-vid transparent video facetop. In *Proc of UIST 2003*, pages 57–58 and demo.

Sutcliffe, G. and Suttner, C. (1998). The TPTP Problem Library: CNF Release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203.

Swanson, E. B. (1999). IS maintainability: Should it reduce the maintenance effort? In *Proc. of the 1999 ACM SIGCPR Conf. on Computer Personnel Research*, pages 164–173. Apr 8-10, 1999.

Team, C. P. (2002). Capability maturity model-d integration (CMMI^{SM}), version 1.1. Technical Report CMU/SEI-2002-TR-029, Carnegie Mellon Software Engineering Institute.

Turley, R. T. and Bieman, J. M. (1995). Competencies of exceptional and non-exceptional software engineers. *Journal of Systems and Software*, 28(1):19–38.

van Rossum, G. (2003). *Python Language Reference Manual - Release 2.3*. Network Theory Limited, Bristol, UK.

van Winsen, P. (1996). (Re)engineering with object oriented design patterns. Master's thesis, Utrecht University.

Vlissides, J. M. (1998). Notation, notation, notation. *C++ Report*, pages 48–51.

von Mayrhauser, A. and Vans, A. M. (1995). Program comprehension during software maintenance and evolution. *IEEE Computer*, 28(8).

von Mayrhauser, A. and Vans, A. M. (1996a). Identification of dynamic comprehension processes during large scale maintenance. *IEEE Transactions on Software Engineering*, 22(6):424–437.

von Mayrhauser, A. and Vans, A. M. (1996b). On the role of hypotheses during opportunistic understanding while porting large scale code. In *Proc. of the 4th Intl. Workshop on Progam Comprehension (IWPC'96)*. March 1996.

Wall, L., Christiansen, T., and Orwant, J. (2000). *Programming Perl*. O'Reilly & Associates, 3rd edition.

Wallace, L. G. (1999). *The Development of an Instrument to Measure Software Project Risk*. Ph.D. dissertation, Georgia State University.

Wallace, L. G., Keil, M., and Rai, A. (2004). How software project risk affects project outcomes: An investigation of the dimensions of risk and an exploratory model. *Decision Sciences*, 35(2):289–321.

Weiderhold, G., Wegner, P., and Ceri, S. (1992). Towards megaprogramming. *Communications of the ACM*, 35(11):89–99.

Weiser, M. (1984). Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357.

Woolf, B. (1998a). The abstract class pattern. In Harrison, N., Foote, B., and Rohnert, H., editors, *Pattern Languages of Program Design 4*. Addison-Wesley.

Woolf, B. (1998b). The object recursion pattern. In Harrison, N., Foote, B., and Rohnert, H., editors, *Pattern Languages of Program Design 4*. Addison-Wesley.

Wos, L. and Pieper, G. W. (2003). *Automated Reasoning and the Discovery of Missing and Elegant Proofs*. Rinton Press.

Yourdon, E. and Constantine, L. (1979). *Structured Design*. Prentice Hall, Englewood Cliffs, N.J.

Zimmer, W. (1995). Relationships between design patterns. In Coplien, J. O. and Schmidt, D. C., editors, *Pattern Languages of Program Design*, pages 345–364. Addison-Wesley.