Technical Report TR03-032

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

Elemental Design Patterns and the ρ-calculus: Foundations for Automated Design Pattern Detection in SPQR

Jason McC. Smith and David Stotts

Dept of Computer Science
Univ. of North Carolina at Chapel Hill
Chapel Hill, NC 27599-3175

smithja@cs.unc.edu

September 23, 2003

Elemental Design Patterns and the ρ-calculus: Foundations for Automated Design Pattern Detection in SPQR

Jason McC. Smith
University of North Carolina at Chapel Hill
Sitterson Hall CB #3175
Chapel Hill, NC 27599-3175
smithja@cs.unc.edu

David Stotts
University of North Carolina at Chapel Hill
Sitterson Hall CB #3175
Chapel Hill, NC 27599-3175
stotts@cs.unc.edu

Abstract

Finding design patterns in source code helps in maintenance, comprehension, refactoring and design validation during software development. SPQR (System for Pattern Query and Recognition) is a toolset for the automated discovery of design patterns in source code. SPQR uses a logical inference system to reveal large numbers of patterns and their variations from a small number of definitions. A formal denotational semantics is used to encode fundamental OO concepts (which we term elemental design patterns*), and a small number of rules (which we call* reliance operators*) for combining these concepts into larger patterns. These reliance operators, when combined with the ς-calculus[1], provide a formal foundation we call the* rho(ρ)-calculus*. In this paper we present both the formal semantics of SPQR and a discussion of other practical applications for elemental design patterns.*

1 Introduction

The System for Pattern Query and Recognition (SPQR) [24] improves on previous approaches for finding design patterns in source code. Other systems have been limited by the difficulty of describing something as abstract as design patterns. A single design pattern when reduced to concrete code can have myriad realizations, all of which have to be recognized as instances of that one pattern. Other systems have encountered difficulty in spanning these possible implementation variations, due to their reliance on static definitions of patterns and variants. SPQR overcomes this problem by using an inference system based on core concepts and semantic relationships. The formal foundation of SPQR defines base patterns and rules for how variation can occur, and the inference engine is free to apply variation rules in an unbounded manner. A finite number of definitions in SPQR can match an unbounded number of implementation variations.

This foundation is composed of two parts: the fundamental concepts of object-oriented programming and design (Elemental Design Patterns), and the rules for their variation and composition (ρ-calculus). The EDPs were deduced through careful analysis of the Gang of Four (GoF) design patterns [11] for use of core object-oriented language concepts (such as inheritance, delegation, recursion, etc); this analysis produced eleven EDPs from which the GoF patterns can be composed. It also produced a design space which was filled out to produce sixteen comprehensive EDPs. We speculate that these additional EDPs will prove useful in defining design patterns from other sources.

In the remainder of this paper we first discuss related work, and informally introduce our Elemental Design Patterns. We then provide several practical applications for these concepts, including a summary of the SPQR toolset and its use in finding GoF design patterns. We next present a rationale for the claim that EDPs capture the fundamental OO concepts needed for expression and composition of larger design patterns. We conclude with the full ρ-calculus, the formal semantics of our SPQR tools.

2 Related work

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 [6, 13, 22, 27]. The few researchers who have attempted to provide a formal basis for patterns have most commonly done so from a desire to perform refactoring of existing code, while others have attempted the more pragmatic approach of identifying core components of existing patterns in use.

2.1 Refactoring approaches

Attempts to formalize refactoring [10] exist, and have met with fairly good success to date[7, 18, 20]. The primary motivation is to facilitate tool support for, and validation of, the transformation of code from one form to another while preserving behaviour. 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. Such techniques include fragments, as developed by Florijn, Meijers, and van Winsen [9], Eden's work on LePuS [8], and Ó Cinnéide's work in transformation and refactoring of patterns in code [19] through the application of minipatterns. These approaches have one missing piece: appropriate flexibility of implementation.

2.2 Structural analyses

An analysis of the 'Gang of Four' (GoF) patterns [11] reveals many shared structural and behavioural elements, such as the similarities between Composite and Visitor [11]. 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 [4, 6, 22, 26, 27].

Objectifier: The Objectifier pattern [27] is one such example of a core piece of structure and behaviour shared between many more complex patterns. Zimmer uses Objectifier as a 'basic pattern' in the construction of several other GoF patterns, such as Builder, Observer, Bridge, State, Command and Iterator. It is a simple yet elegantly powerful structural concept that is used repeatedly in other patterns.

Object Recursion: Woolf takes Objectifier one step further, adding a behavioural component, and naming it Object Recursion [26]. The class diagram is extremely similar to Objectifier, with an important difference, namely the behaviour in the leaf subclasses of *Handler*. Exclusive of this method behaviour, however, it seems to be an application of Objectifier 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.

Taken together, the above instances of analyzed pattern findings comprise two parts of a larger chain: Object Recursion contains an instance of Objectifier, and both in turn are used by larger patterns. This indicates that there are meaningful relationships between patterns, yet past work has shown that there are more primary forces at work. Buschmann's variants [5], Coplien and others' idioms [3, 6, 16], and Pree's metapatterns [21] all support this viewpoint. Shull, Melo and Basili's BACKDOOR's

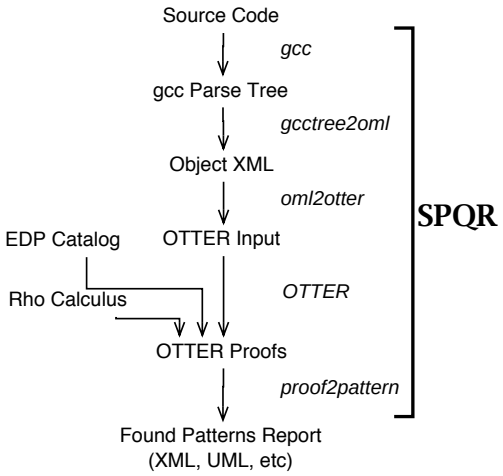

Figure 1. SPQR Tool Chain

[23] dependency on relationships is exemplary of the normal static treatment that arises. It will become evident that these relationships between *concepts* are a core piece which grant great flexibility to the practitioner implementing patterns in design, through constructs we term *isotopes*, which will be treated in Section 5.4.1.

2.3 Elemental Design Patterns

Informally, Elemental Design Patterns are design patterns that cannot be decomposed into smaller patterns - they sit one level above the primitives of object-oriented programming, such as objects, fields, and methods. As such, they occupy much of the same space as idioms, but are language independent, relying only on the core concepts of object-oriented theory. They perform the same conceptual task as the more common design patterns however, in that they provide solutions to common programming situations, and do so in orthogonal ways, differing only in scope. EDPs are the foundational nuts and bolts from which conceptual design frameworks are created. Also, relying only on the theoretical basis OO, they have one distinct advantage over other approaches directly involving design patterns: they are formalizable. We have chosen to extend the sigma-calculus [1] with a small set of relationship operators that provide a simple but solid basis, the ρ-calculus, from which to perform interesting analyses.

3 Practical Applications

Elemental Design Patterns and the ρ-calculus would be interesting academic exercises only, unless there were good reasons for performing this work. We present here three applications, including as an education tool, a set of concepts

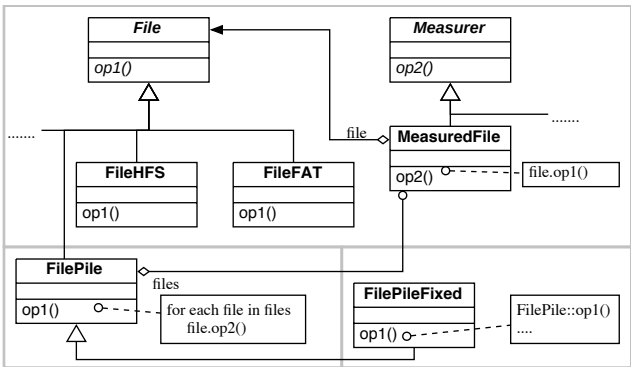

Figure 2. Grown system

for programming language design, and a formal basis for source code analysis tools.

3.1 SPQR

Our original driving problem is a ubiquitous one in software engineering - how best to aid an engineer in comprehending a large or complex system. One such way is to provide automated tools to extract conceptual information and present it to the user in a meaningful way. Our System for Pattern Query and Recognition [24] (SPQR) performs this task, finding instances of known design patterns within source code and alerting the engineer to their presence. Obviously, this has many direct applications, such as system inspection during education, ensuring that intended patterns exist in the final code, and investigation of unintended pattern instances that may provide cues for refactoring.

SPQR relies on the formal nature of EDPs and the ρ-calculus to perform the bulk of the discovery, by using an automated theorem prover to *infer* the existence of patterns in the source code under scrutiny. The SPQR toolchain comprises of several components, shown in Figure 1. From the engineer's point of view, SPQR is a single fully automated tool that performs the 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*. Compiler output of the syntax tree of a codebase (one current input source is gcc) is transformed into our formal notation encoded as input to the OTTER automated theorem prover [15]. This fact set, combined with the pre-defined encodings for the EDP Catalog and the relationships rules of ρ-calculus are operated on by OTTER to find instances of design patterns which are then reported to the user. SPQR is language and domain independent, and would be impractical without the formal foundations of our EDPs and ρ-calculus.

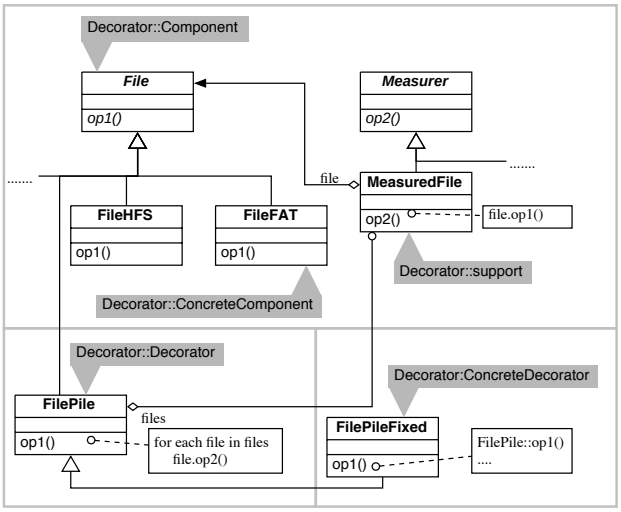

Figure 3. Primary discovered pattern roles

As an example, Figure 2 is a system that was designed as many are: through growth. Three distinct libraries are interacting in well-formed ways, each under the control of a different engineering group. Source code is unavailable to the other groups, and design issues are partitioned as well. After analysis by SPQR, however, it becomes clear, as in Figure 3, that there is a hidden instance of the Decorator design pattern that provides an intriguing clue to the engineers how to refactor the three libraries to work more effectively together. This example is illustrated in complete detail, and SPQR discussed more thoroughly, in prior publication.

3.2 EDPs as language design hints

While some will see the EDPs as truly primitive, we would point out that the development of programming languages has been a reflection of directly supporting features, concepts, and idioms that practioners of the previous generations of languages found to be useful. Cohesion and coupling analysis of procedural systems gave rise to many object oriented concepts, and each common OO 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 make from simpler forms.

A recent and highly touted example of such a language construct is the *delegate* feature found in C#[17]. 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 behaviours and structures for which languages provide support.

Most languages have some support for **ExtendMethod**, 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[14], 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.

The **AbstractInterface** EDP is, admittedly, one of the simplest in the collection. Every OO 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 behaviour such as Smalltalk's exception throwing for an unimplemented method. It should be noted though that the above are either composite constructs (*virtual foo() = 0;* in C++) or a non construct runtime behaviour (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 OO design. Future languages may benefit from a more explicit construct.

3.3 Educational uses of EDPs

We believe the EDPs 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 canon requires some non-trivial amount of design understanding 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. This parallels the suggestions put forth by Goldberg in 1994[12].

4 Elemental Design Patterns

The Elemental Design Patterns were deduced through analysis of the existing design pattern literature, and then extended to comprehensively cover the most interesting ways in which objects can interact in object-oriented programming. We believe that the sixteen EDPs comprise the core of any design space.

4.1 Examination of design patterns

Our first task was to examine the existing canon of design pattern literature, and a natural place to start is the ubiquitous Gang of Four text[11]. Instead of a purely structural inspection, we 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 - you wish to enforce polymorphic behaviour by requiring all subclasses to implement a method. Equivalent to Woolf's Abstract Class pattern[25], 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[6].

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 behaviour. Used in Decorator.

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

Iteration A runtime behaviour 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 behaviour 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, very similar to Ó Cinnéide's Encapsulate Construction minipattern[19]. 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.

On inspection, five of these possible patterns are centered around some form of method invocation. This led us to investigate what the critical forms of method calling truly are, and whether they could provide insights towards producing a comprehensive collection of EDPs. We assume, for the sake of this investigation, a dynamically bound language environment and make no assumptions regarding features of implementation languages. Categorizing the various forms of method calls in the GoF patterns can be summarized as in Table 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 behaviour 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.

If we eliminate the ownership attribute, we find that 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, we are free to reintroduce such traits later. Similarly, other method invocation attributes could be assigned, but do not fit within our 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 our CreateObject component, but we can place it aside for now to concentrate on the typing variations of method calls.

At this time, we can reorganize Table 1 slightly, removing the Mediator and Flyweight entry on the last line, as no typing attributable method invocations occur within those patterns. We can also merge State and Bridge into the appropriate calling styles, then note that four of our remaining list are simply variations on whether the called method is abstract or not. By identifying this as an instance of the AbstractInterface component from above, we can simplify this list further to our final collection of the six primary method invocation styles in the GoF text, shown in Table 2. We 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

This last item recurs often in our analysis, and once it is realized that it is the application of Beck's Intention Revealing Message best practice pattern [3], it becomes obvious that this is an important concept we dub *similarity*.

4.2 Method call EDPs

The first axis in the above list is simply a dichotomy between $Self$ and $Other$.[1] The second describes the relationship between A and B, if any, and the third 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.

It is illustrative at this point to attempt creation of a comprehensive listing of the various permutations of these axes, and see where our identified invocation styles fall into place. For the possible relationships between A and B, we have started with our list items of 'Parent', where $A <: B$,[2] '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 we add 'Same', or $A = B$, as an obvious simple type relation between the objects.[3]

[1] $Child$ is another possibility here, and a call to $Same$ maps to BETA's *inner*, for example.

[2] The notation is taken from Abadi and Cardelli's sigma calculus[1]. $A <: B$ reads 'A is a subtype of B'

[3] $Child$ is possible here as an addition as well, although we do not do so at this time.

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 1. 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, Bridge Adapter (object), Observer, Command, Memento

Table 2. Final method calling styles in Gang of Four patterns

4.2.1 Initial list

We start by filling in the invocation styles from our final list from the GoF patterns, mapping them to our six categories in Table 2:

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 ($A <: C, B <: C, A \not<: B$)
      i. Same ($F = G$) RedirectInLimitedFamily[5]
      ii. Different ($F \neq G$) .....

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[2]. We will treat these as meeting the definition of design patterns, and present them as such.

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

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

ExtendMethod A subclass wishes to extend the behaviour 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 namespace clash in the case of Adapter (class).

Redirect A method wishes to redirect some portion of its functionality to an extremely similar method in another object. We choose 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 behaviour 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.

4.2.2 The full list

We can now begin to see where the remainder of the method call EDPs will take us. Again, we will present the listing, and briefly discuss each new item 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. Diff ($F \neq G$) . . . DelegatedConglomeration
   (c) Parent ($A <: B$)
      i. Same ($F = G$) . . . . . RedirectInFamily
      ii. Different ($F \neq G$) . . . . . DelegateInFamily
   (d) Sibling ($A <: C, B <: C, A \not<: B$)
      i. Same ($F = G$) . . . RedirectInLimitedFamily
      ii. Diff ($F \neq G$) . . . . DelegateInLimitedFamily

Recursion Quite obvious on examination, this is a concrete link between primitive language features and our EDPs.

RedirectedRecursion A form of object level iteration.

DelegatedConglomeration Gathers behaviours from external instances of the current class.

DelegateInFamily Gathers related behaviours from the local class structure.

DelegateInLimitedFamily Limits the behaviours selected to a particular base definition.

4.3 Object Element EDPs

At this point we have a fairly comprehensive array of method/ object invocation relations, and can revisit our original list of concepts culled from the GoF patterns. Of the original eight, three are absorbed within our method invocations list: DelegatedImplementation, ExtendMethod, and AggregateAlgorithm. Of the remaining five, two are some of the more problematic EDPs to consider: Iteration, and Invariance. These can be considered sufficiently difficult concepts at this stage of the research that they are beyond the scope of this paper.

Our remaining three EDPs, CreateObject, AbstractInterface, and Retrieve, deal with object creation, method implementation, and object referencing, respectively. These are core concepts of what objects and classes are and how they are defined. CreateObject creates instances of classes, AbstractInterface determines whether or not that instance contains an implementation of a method, and Retrieve is the mechanism by which external references to other objects are placed in data fields. These are the elemental creational patterns and they provide the construction of objects, methods, and fields. Since these are the three basic physical elements of object oriented programming[1], we feel that these are a complete base core of EDPs for this classification.[4]

CreateObject Constructs an object of a particular type.

AbstractInterface Indicates that a method has *not* been implemented by a class.

Retrieve Fetches objects from outside the current object, initiating external references.

The method invocation EDPs from the previous section are descriptions of how these object elements interact, defining the relationships between them. One further relationship is missing, however: that between types. Subtyping is a core relationship in OO languages, usually expressed through an inheritance relation between classes. Subclassing, however, is *not* equivalent to subtyping[1], and

[4] Classes, prototypes, traits, selectors, and other aspects of various object oriented languages are expressible using only the three constructs identified.[1]

should be noted as a language construct extension to the core concepts of object-oriented theory. Because of this, we introduce a typing relation EDP, Inheritance, that creates a structural subtyping relationship between two classes. Not all languages directly support inheritance, it may be pointed out, instead relying on dynamic subtyping analysis to determine appropriate typing relations.

Inheritance Enforces a structural subtyping relationship.

5 Rho Calculus

Our EDPs are useful in many areas as they stand, but for formal analysis of source code, we needed to create an approach that would provide a semantic basis for logical inferences. Rho-calculus is the formal foundation of the EDP catalog. It allows us to encode facts about a codebase into a simple yet powerful notation that can be directly input to automated theorem provers, such as the OTTER system used in SPQR. Without this, the EDPs would be conceptually useful but impractical to use in an automated tool system.

This section defines the rho fragment (Δ_ρ) of the ρ-calculus which results when this fragment is added to the ς-calculus. By defining this as a calculus fragment, we allow researchers to add it to the proper mix of other fragments defined in [1] to create the particular formal language they need to achieve their goals.

5.1 Definitions

Let us define O as the set of all objects instantiated within a given system. Then $\mathcal{O} \in O$ is some object in the system. Similary, let M be the set of all method signatures within the system. Then $\mu \in M$ is some method signature in the system. $\mathcal{O}.\mu$ is then the selection of some method signature imposed on some object. We make no claim here that this is a well-formed selection, and in fact we have no need to - the underlying ς-calculus imposes that construct for us. τ is some type in the set of all types T defined in the system such that if $\mathcal{O}$ is of type τ, then $\mathcal{O} : \tau$.

$$\mathcal{O} \in O, \mu \in M, \tau \in T$$

Let A be either an object $\mathcal{O}$ or a method selection $\mathcal{O}.\mu$. Let A' be another such set for distinct object and method selections. (By convention, the base forms of the symbols will appear on the left side of the reliance operator (relop), and the prime forms will appear on the right hand side to indicate distinct items.) x is a signifier that a particular reliance operator may be one of our three variants: $\{\mu, \phi, \gamma\}$. μ is a method selection reliance, ϕ is a field reliance, and γ is a 'generalized' reliance where a reliance is known, but the exact details are not. (It is analogous to more traditional forms of coupling theory.) $\overset{\pm}{\circ}$ is an operator trait indicator, allowing for the three types of reliance specialization ($+$, $-$, $\circ$) to be abstracted in the following rules. The appearance of this symbol indicates that any of the three may exist there.

$$A = \{\mathcal{O}, \mathcal{O}.\mu\}$$
$$A' = \{\mathcal{O}', \mathcal{O}'.\mu'\}$$
$$x = \{\mu, \phi, \gamma\}$$
$$\overset{\pm}{\circ} = \{+, -, \circ\}$$

The basic reliance operator symbol, $<$, was selected to be an analogue to the inheritance/subsumption of types indicator in sigma calculus, $<:$, which can be interpreted to mean a reliance of type. Since the typing symbol is :, this leaves $<$ as a natural for the concept of 'reliance on'. This, combined with our three symbols from x above, gives rise to our three reliance operators: $<_\mu$, $<_\phi$, $<_\gamma$

5.2 Creation

We have three rules that create instances of our reliance operators. First, we have the Method Invocation Relop rule, which states that given a method μ invoked on object $\mathcal{O}$, if that method contains a method invocation call to method μ' of another object $\mathcal{O}'$, we have a method reliance between the two, indicated by the μ form reliance operator ($<_\mu$):

$$\frac{\mathcal{O}.\mu \equiv [\mu = \varsigma()\mathcal{O}'.\mu']}{\mathcal{O}.\mu <_\mu \mathcal{O}'.\mu'} \quad (1)$$

We have a similar rule for deriving an instance of a field reliance operator. This one states that if an object's method $\mathcal{O}.\mu$ contains a reference to another object $\mathcal{O}'$, then there is a reliance between the two based on reference access of the field, indicated by the ϕ form reliance operator ($<_\phi$). This is the Method Field Relop rule:

$$\frac{\mathcal{O}.\mu \equiv [\mu = \varsigma()\mathcal{O}']}{\mathcal{O}.\mu <_\phi \mathcal{O}'} \quad (2)$$

Similarly, if an object $\mathcal{O}'$ is referenced as an instance variable data field of an object $\mathcal{O}$, then we can use the Object Field Relop rule:

$$\frac{\mathcal{O} : \tau, \tau = [\mathcal{O}' : \tau']}{\mathcal{O} <_\phi \mathcal{O}'} \quad (3)$$

5.3 Similarity Specializations

We can pin down further details of the relationships between the operands of the reliance operators by inspecting the method signatures or the object types for μ and ϕ form

relops, respectively, reflecting the *similarity* trait found in the EDP catalog.

If the method signatures on both sides of a μ form relop match, then we have a similarity invocation, and append a $+$ to the relop symbol to indicate this trait:

$$\frac{\mathcal{O}.\mu <_{\mu} \mathcal{O}'.\mu', \mu = \mu'}{\mathcal{O}.\mu <_{\mu+} \mathcal{O}'.\mu'} \tag{4}$$

If, on the other hand, we know for a fact that the two method signatures do not match, then we have a dissimilarity invocation, and we append a $-$ to the relop:

$$\frac{\mathcal{O}.\mu <_{\mu} \mathcal{O}'.\mu', \mu \neq \mu'}{\mathcal{O}.\mu <_{\mu-} \mathcal{O}'.\mu'} \tag{5}$$

We follow a similar approach with the inspection of the object types of the operands in a ϕ form relop. If the two types are equal, then we have a similarity reference:

$$\frac{A <_{\phi} \mathcal{O}', \mathcal{O} : \tau, \mathcal{O}' : \tau', \tau = \tau'}{A <_{\phi+} \mathcal{O}'} \tag{6}$$

And if the two types are known to be unequal, then we have a dissimilarity reference:

$$\frac{A <_{\phi} \mathcal{O}', \mathcal{O} : \tau, \mathcal{O}' : \tau', \tau \neq \tau'}{A <_{\phi-} \mathcal{O}'} \tag{7}$$

In both the μ and ϕ form relops, if the above information is not known with certainty, then the relop remains unappended in a more general form.

5.4 Transitivity

Transitivity is the process by which large chains of reliance can be reduced to simple facts regarding the reliance of widely separated objects in the system. The three forms of relop all work in the same manner in these rules. The specialization trait of the relop ($\pm$) is not taken into consideration, and in fact can be discarded during the application of these rules - appropriate traits can be re-derived as needed.

Given two relop facts, such that the same object or method invocation appears on the *rhs* of the first and the *lhs* of the second, then the *lhs* of the first and *rhs* of the second are involved in a reliance relationship as well. If the two relops are of the same form, then the resultant relop will be the same as well.

$$\frac{A <_{x} A', A' <_{x'} A''}{A <_{x} A''} \ \mathrm{i}ff \ \ x = x' \tag{8}$$

If, however, the two relops are of different forms, then the resultant relop is our most general form, γ. This indicates that while a relationship exists, we can make no hard connection according to our definitions of the μ or ϕ forms. Note that this is the only point at which γ form relops are created.

$$\frac{A <_{x} A', A' <_{x'} A''}{A <_{\gamma} A''} \ \mathrm{i}ff \ \ x \neq x' \tag{9}$$

5.4.1 Isotopes

This is the key element of our *isotopes*, which allow design patterns to be inferred in a ***flexible*** manner. We do not require each and every variation of a pattern to be statically encoded, instead the transitivity in the ρ-calculus allows us to simply encode the relationships between elements of the pattern, and an automated theorem prover can infer as many possible situations as the facts of the system provide. In this way a massive search space can be created automatically from a small number of design pattern definitions.

5.5 Generalizations

These are generalizations of relops, the opposite of the specialization rules earlier. Each of them generalizes out some piece of information from the system that may be unnecessary for clear definition of certain rules and situations. Information is not lost to the system, however, as the original statements remain.

The first two generalize the right hand side and left hand sides of the relop, respectively, removing the method selection but retaining the object under consideration. They are RHS Generalization and LHS Generalization.

$$\frac{A <_{x^{\pm}_{\circ}} \mathcal{O}'.\mu'}{A <_{x^{\pm}_{\circ}} \mathcal{O}'} \tag{10}$$

$$\frac{\mathcal{O}.\mu <_{x^{\pm}_{\circ}} A'}{\mathcal{O} <_{x^{\pm}_{\circ}} A'} \tag{11}$$

This is a Relop Generalization. It indicates that the most general form of reliance (γ) can always be derived from a more specialized form (μ, ϕ).

$$\frac{A <_{x} A'}{A <_{\gamma} A'} \tag{12}$$

Similarly, the Similarity Generalization states that any specialized similarity trait form of a relop implies that the more general form is also valid.

$$(x = \mu, \phi) \qquad \frac{A <_{x\pm} A'}{A <_{x} A'} \tag{13}$$

6 Conclusion

We have presented the foundations for the System for Pattern Query and Recognition (SPQR), comprised of the

Elemental Design Patterns and matching formalizations in the *ρ-calculus* for composition into larger, more useful and abstract design patterns as usually found in software architecture. These EDPs were identified initially through inspection of the existing literature on design patterns, establishing which solutions appeared repeatedly within the same contexts, mirroring the development of the more traditional design patterns. Further, they are formally describable in the ρ-calculus, a notation that builds upon the ς-calculus, but adds the key concept of *reliance* to the base notation. These extensions, the *reliance operators* provide a large degree of flexibility to formally stating the relationships embodied in design patterns as *isotopes*, without locking them into any one particular implementation.

These contributions will allow for new approaches to analyzing software systems, education regarding design patterns and best practices in object-oriented architecture, and may help guide future language design by indicating which design elements are most commonly used by software architects.

Acknowledgments

The authors would like to acknowledge the contributions of our readers and the financial support of EPA Project # R82 - 795901 - 3.

References

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag New York, Inc., 1996.

[2] C. W. Alexander. *Notes on the Synthesis of Form*. Oxford Univ Press, 1964. Fifteenth printing, 1999.

[3] K. Beck. *Smalltalk Best Practice Patterns*. Prentice Hall, 1997.

[4] J. Bosch. Design patterns as language constructs. *Journal of Object Oriented Programming*, 1(2):18–52, May 1998.

[5] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented System Architecture: A System of Patterns*. John Wiley & Sons, 1996.

[6] J. Coplien. C++ idioms. In *Proceedings of the Third European Conference on Pattern Languages of Programming and Computing*, July 1998.

[7] S. Demeyer, S. Ducasse, and O. Nierstrsz. Finding refactoring via change metrics. In *Proceedings of the conference on Object-oriented programming, systems, languages, and applications*, pages 166–177. ACM Press, nov 2000.

[8] A. H. Eden. *Precise Specification of Design Patterns and Tool Support in their Application*. PhD thesis, Tel Aviv University, Tel Aviv, Israel, 2000.

[9] G. Florijn, M. Meijers, and P. van Winsen. Tool support for object-oriented patterns. In M. Askit and S. Matsuoka, editors, *Proc. of the 11th European Conf. on Object Oriented Programming - ECOOP'97*. Springer-Verlag, Berlin, 1997.

[10] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley, 1995.

[12] A. Goldberg. 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, 1995.

[13] B. B. Kristensen. 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, 1994.

[14] O. L. Madsen, B. Møller-Pederson, and K. Nygaard. *Object-oriented Programming in the BETA language*. Addison-Wesley, 1993.

[15] W. McCune. Otter 2.0 (theorem prover). In M. E. Stickel, editor, *Proc. of the 10th Intl Conf. on Automated Deduction*, pages 663–664, July 1990.

[16] S. Meyers. *Effective C++*. Addison-Wesley, 1992.

[17] Microsoft Corporation, editor. *Microsoft Visual C# .NET Language Reference*. Microsoft Press, 2002.

[18] I. Moore. 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, 1996.

[19] M. Ó Cinnéide. *Automated Application of Design Patterns: A Refactoring Approach*. Ph.D. dissertation, University of Dublin, Trinity College, 2001.

[20] W. F. Opdyke and R. E. Johnson. Creating abstract superclasses by refactoring. In *Proc. of the Conf. on 1993 ACM Computer Science*, page 66, 1993. Feb 16-18, 1993.

[21] W. Pree. *Design Patterns for Object-Oriented Software Development*. Addison-Wesley, 1994.

[22] D. Riehle. Composite design patterns. In *Proceedings of the 1997 ACM SIGPLAN conference on Object-oriented programming systems, languages and applications*, pages 218–228. ACM Press, 1997.

[23] F. Shull, W. L. Melo, and V. R. Basili. An inductive method for discovering design patterns from object-oriented software systems. Technical Report CS-TR-3597, University of Maryland, 1996.

[24] J. M. Smith and D. Stotts. SPQR: Flexible automated design pattern extraction from source code. In *18th IEEE Intl Conf on Automated Software Engineering*, oct 2003.

[25] B. Woolf. The abstract class pattern. In N. Harrison, B. Foote, and H. Rohnert, editors, *Pattern Languages of Program Design 4*. Addison-Wesley, 1998.

[26] B. Woolf. The object recursion pattern. In N. Harrison, B. Foote, and H. Rohnert, editors, *Pattern Languages of Program Design 4*. Addison-Wesley, 1998.

[27] W. Zimmer. Relationships between design patterns. In J. O. Coplien and D. C. Schmidt, editors, *Pattern Languages of Program Design*, pages 345–364. Addison-Wesley, 1995.