Technical Report TR02-038

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

Elemental Design Patterns: A Logical Inference System and Theorem Prover Support for Flexible Discovery of Design Patterns

Jason McC. Smith and David Stotts

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

stotts@cs.unc.edu

September 9, 2002

Elemental Design Patterns: A Logical Inference System and Theorem Prover Support for Flexible Discovery of Design Patterns

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

Abstract

*Previous approaches to discovering design patterns in source code have suffered from a need to enumerate static descriptions of structural and behavioural relationships, resulting in a finite library of variations on pattern implementation. Our approach differs in that we do not seek to statically encode each pattern, and each variant, that we wish to find. Rather, we encode in a formal denotational semantics a small number of fundamental OO concepts (*elemental design patterns*), encode the rules by which these concepts are combined to form patterns(*reliance operators*), and encode the structural/behavioral relationships among components of objects and classes (*rho-calculus*). A logical inference system then is used to reveal large numbers of patterns and their variations from this small number of definitions. Our system finds patterns that were not* explicitly *defined, but instead are inferred dynamically during code analysis by a theorem prover, providing practical tool support for software construction, comprehension, maintenance, and refactoring.*

1. Introduction

Practical tool support has always lagged behind the development of important abstractions and theoretical concepts in programming languages. One current successful abstraction in widespread use is the design pattern, an approach describing portions of systems that designers can learn from, modify, apply, and understand as a single conceptual item [11]. Design patterns are generally, if informally, defined as common solutions to common problems which are of significant complexity to require an explicit discussion of the scope of the problem and the proposed solution. Much of the popular literature on design patterns is dedicated to these larger, more complex patterns, providing practitioners with increasingly powerful constructs with which to work.

Design patterns, however, are at such a level of abstraction that they have so far proven resistant to tool support. The myriad variations with which any one design pattern may be implemented makes them difficult to describe succinctly or find in source code. We have discovered a class of patterns that are small enough to find easily but composable in ways that can be expressed in the rules of a logical inference system. We term them *Elemental Design Patterns* (EDPs), and they are the base concepts on which the more complex design patterns are built. Because they comprise the constructs which are used repeatedly within more common patterns to solve the same problems, such as abstraction of interface and delegation of implementation, they exhibit interesting properties for partially bridging the gap between source code in everyday use and the higher-level abstractions of the larger patterns. The higher-level patterns are thus described in the language of elemental patterns, which fills an apparent missing link in the abstraction chain.

The formally expressible and informally amorphous halves of design patterns also present an interesting set of problems for the theorist due to their dual nature [2]. The concepts contained in patterns are those that the professional community has deemed important and noteworthy, and they are ultimately expressed as source code that is reducible to a mathematically formal notation. The core concepts themselves have to date evaded such formalization. We show here that such a formalization is possible, and in addition that it can meet certain essential critieria. We also show how the formalization leads to useful and direct tool support for the developer with a need for extracting patterns from an existing system.

We assert that such a formal solution should be implementation language independent, much as design patterns are, if it is to truly capture universal concepts of programming methodology. We further assert that a formal denotation for pattern concepts should be a larger part of the formal semantics literature. Patterns are built on the theory and concepts of object-oriented programming, as surely as object-oriented approaches are built on procedural theory.

We begin with describing our driving problem, providing a concrete example system. We then discuss the related work in the field of automated pattern extraction, leading into a brief introduction to our EDPs. We show how these EDPs can be formally expressed in a version of the sigma (ς) calculus [1], that we have extended with *reliance operators*. We illustrate our method with a chain of pattern composition from our EDPs to the Decorator pattern. We then show how to derive an instance of Decorator from our example scenario using automatable reduction rules that are processed by a theorem prover.

2. Problem scenario

At Widgets, Inc., there are many teams working on the next Killer Widget application. Each is responsible for a well-defined and segmented section of the app, but they are encouraged to share code and classes where possible. As is normal in many such situations, teams have write access only for their own code - they are responsible for it, and all changes must be cleared through regular code reviews. All other teams may inspect the code, but may not change it. Suggestions can be made to the team in charge, to be considered at the next review, but no one likes their time wasted, and internal changes take priority during such reviews.

A legacy library exists (call it BaseLib, shown in Figure 1) that is heavily used throughout Killer Widget. One of the core pieces of this library is the File hierarchy, a class tree for cross-platform file handling, which has been used since the 1.0 release of the product. The Measurer hierarchy is a class tree for gathering statistical data on the performance of key classes. Originally designed as a temporary fix using an Adapter pattern (with object compositing) to wrap existing classes, it quickly became used for several key pieces of new code when the 2.0 release deadline loomed. By the 4.0 release, the entire code base had been migrated to use the MeasuredFile class instead of the more natural File class, even though by that time it was obvious that the Measurer classes probably should have been implemented differently. The source code for the entire library is large, unwieldly, and, frankly, no one still at Widgets, Inc. is quite sure what is in it. The official policy, due to the critical nature of the library, is that no changes will be incorporated into BaseLib.

With the 5.0 release, Team A, in charge of core file access, has been tasked with making the File concept more flexible, to allow new UI behaviour. (Specifically, the UI team has determined that it would be highly useful for Killer Widget to be able to handle the dropping of a folder onto its icon, as well as a file.) They diligently add to the hierar-

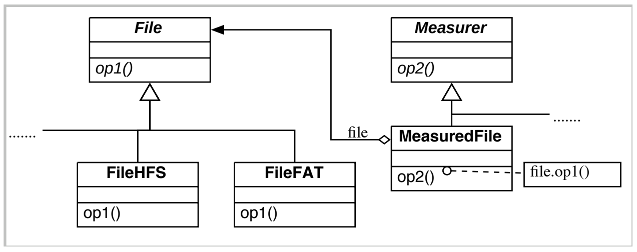


Figure 1. BaseLib

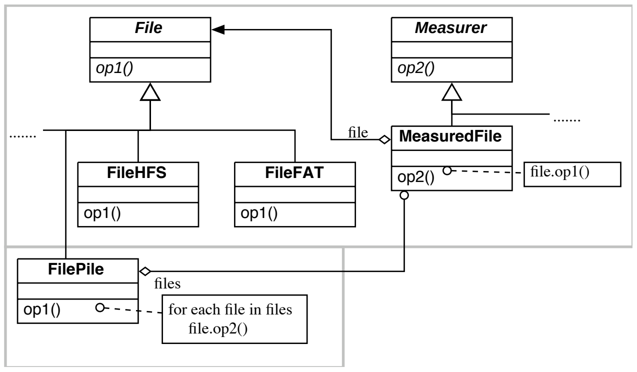


Figure 2. BaseLib + Team A's 5.0 additions

chy under File, and, in the process, include the concept of FilePile as a bundle for Files, to allow the application to handle collections of files internally. FilePile must, however, access File objects through MeasuredFile to gain the statistics gathering capability used for systematic testing, and to be consistent with the rest of the system. It is messy, but with the existing code using MeasuredFile in a pervasive manner and without the authorization to change the MeasuredFile class, they are limited in their options. Team A's efforts are shown with the BaseLib in Figure 2.

For 5.1, Team B, in charge of one of the Widget modules, has been assigned to fix a problem with their mishandling of FilePiles. They determine that the bad assumptions on how to use FilePile pop up in a tremendous number of places in their code and, instead of changing each and every location, they are best served by subclassing FilePile and making the changes there in a single method for their local use. They can then alter the very few places in their code where FilePile objects are created or passed in. Since the results of FilePile's operation are *nearly* correct, and they do not want to replicate FilePile's code (and thereby incur a new testing responsibility), they extend the method by calling FilePile's implementation, and performing a bit of data massaging on the information returned for each file. The system is now show in Figure 3.

For 6.0, it is decided that a massive code review is in order and a new group of developers is brought on to take over the responsibilities of the members of the Teams that have

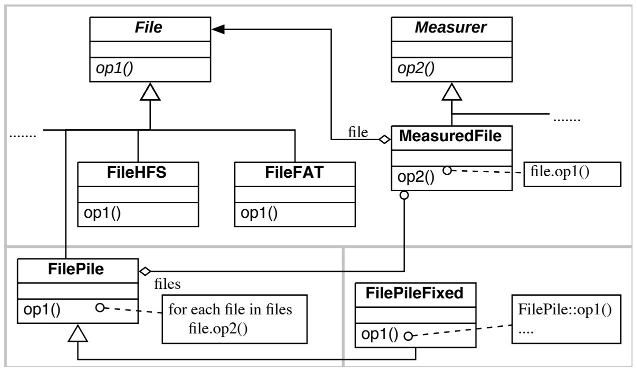


Figure 3. Killer Widget 5.1

been promoted to the code review team. The architecture in Figure 3 is just a small sample of the hundreds of classes in the system, sections of which are owned by various teams.

What insight into the behaviour of the codebase would help both the new engineers and the review board? Hidden patterns exist within the architecture that encapsulate the intent of the larger system, that would facilitate the comprehension of the novice developers, and would help point the architects towards a useful refactoring of the system. We will use this as our driving example.

3. 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, 12, 19, 24]. 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. Additionally, there is ongoing philosophical interest in the very nature of coding abstractions, such as patterns and their relationships.

3.1. Refactoring approaches

Attempts to formalize refactoring [10] exist, and have met with fairly good success to date[7, 15, 17]. The primary motivation is to facilitate tool support for, and validation of, 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 Florijm, Meijers, and van Winsen [9], Eden's work on LePuS [8], and Ó Cinnéide's work in transformation and refactoring of patterns in code [16] through the application

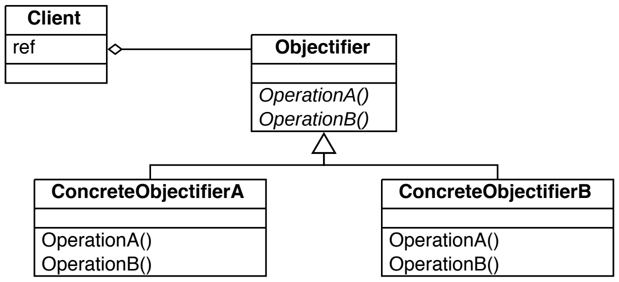


Figure 4. Objectifier

of minipatterns. These approaches have one missing piece: appropriate flexibility of implementation.

3.2. Structural analyses

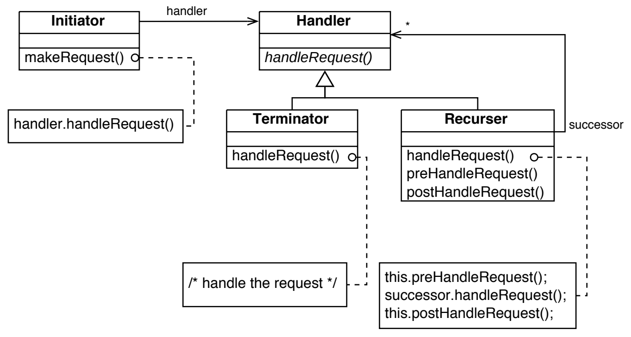
An analysis of the 'Gang of Four' (GoF) patterns [11] reveals many shared structural and behavioural elements, such as the similarities between Composite and Visitor [11]. The relationships between patterns, such as inclusion or similarity, have been investigated by various practitioners, and a number of meaningful examples of underlying structures have been described [4, 6, 19, 22, 23, 24].

Objectifier: The Objectifier pattern [24] is one such example of a core piece of structure and behaviour shared between many more complex patterns. Its Intent is to

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

Zimmer uses 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 Objectifier one step further, adding a behavioural component, and naming it Object Recursion [23]. The class diagram in Figure 5 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 looks 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.



```
handler.handleRequest()
```

```
/* handle the request */
```

```
this.preHandleRequest();
successor.handleRequest();
this.postHandleRequest();
```

Figure 5. Object Recursion

3.3. Conceptual relationships

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, 14], and Pree's metapatterns [18] all support this viewpoint. Shull, Melo and Basili's BACKDOOR's [20] 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 of allowing great flexibility to the practitioner implementing patterns in design, through constructs we term *isotopes*, which will be treated in Section 5.4.

4. The EDP catalog

Our first task was to examine the existing patterns starting with the Gang of Four [11]. Instead of a purely structural inspection, we identified common concepts used in the patterns, resulting in eight identified core concepts. Of these, five involved method invocation, leading us to investigate method interactions more abstractly. The abstractions found are classifiable along three orthogonal axes: the relationship between the calling object and the receiver of the method message; the relationship between the *types* of the calling and receiving objects; the relationship between the signatures of the calling and invoked methods.

The remaining component abstractions from the GoF patterns consisted of abstractions related to object creation, abstract interface of methods, and object retrieval semantics. These combined with our method call invocations and type subsumption result in our EDPs.

We present in this paper a listing of the identified EDPs, in Figure 6. We do not claim that this list covers all the possible permutations of interactions, but that these are the

Object Element EDPs

CreateObject	AbstractInterface
Retrieve	

Type Relation EDPs

Inheritance	

Method Invocation EDPs

Redirect	Delegate
Recursion	Conglomeration
ExtendMethod	RevertMethod
RedirectInFamily	DelegateInFamily
RedirectedRecursion	RedirectInLimitedFamily
DelegateInLimitedFamily	DelegatedConglomeration

Figure 6. Elemental Design Patterns

core catalog of EDPs upon which others will be built. A more complete discussion of the EDPs can be found in [21], but we will provide a detailed example of one in Section 5.3.

At first glance, these EDPs seem unlikely to be very useful, as they appear to be positively primitive... and they are. These are the core primitives that underlie the construction of patterns in general. According to Alexander [2] patterns are descriptions of relationships between entities, and method invocations and typing are the processes through which objects interact. We believe that we have captured the elemental components of object oriented languages, and the base relationships used in the vast majority of software engineering. If patterns are the frameworks on which we create large understandable systems, then these are the nuts and bolts that comprise the frameworks.

5. Formalization

Source code is, at its root, a mathematical symbolic language with well formed reduction rules. We strive to find an appropriately formal analogue for the formal side of patterns. 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 encode as well, in order to preserve their flexibility.

5.1. Sigma calculus

Desired traits of a formalization language include that it be mathematically sound, consist of simple reduction rules, have enough expressive power to encode directly object-oriented concepts, and have the ability to encode flexibly relationships between code constructs. The sigma calculus

[1] is our choice for a formal basis, given the above requirements. It is a formal denotational semantics that deals with objects as primary components of abstraction, and has been shown to have a highly powerful expressiveness for various language constructs.

It is not without its drawbacks, however. Not only is it extremely unwieldly, but also it suffers from a complete rigidity of form, and does not offer any room for interpretation of the implementation description. This lack of adaptiveness means that there would be an explosion of definitions for even a simple 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.

5.2. Reliance operators: the rho calculus

It is fortunate then, that ς-calculus is simple to extend. We propose a new set of rules and operators within ς-calculus to support directly relationships and reliances between objects, methods and fields.

These *reliance operators*, as we have termed them (the word 'relationship' is already overloaded in the current literature, and only expresses part of what we are attempting to deliver), are direct, quantifiable expressions of whether one element (an object, method, or field), in any way relies or depends on the existance of another for its own definition or execution, and to what extent it does so.

This approach provides more detail than the formal description provided by UML however, as the calculus comprised of ς-calculus and the reliance operators, or *rho calculus* encodes 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.

We would like to continue the general notation of ς-calculus, so we adopt the operator used for subsumption, $<:$, and provide a similar sign, $\ll$, that indicates a reliance relationship. If $a \ll b$, then a relies on b in some manner. It may be the interface, the implementation, a data member access, or a particular method call of b which is relied on by a for proper definition and operation. Differentiating between these paths of reliance is a bit more challenging.

For the purposes of this paper we give only a brief description of the needed reliance operators. First, $\ll_m$, indicating a method invocation reliance. Given the expression $a.f \ll_m b.g$, it indicates that within the body of method f in object a, a call is made to method g of object b. A refinement of this is $\ll_{m-}$, which denotes that methods f and g have no signature relationship (see [21] for a full explanation of the importance of this distinction), while $\ll_{m+}$ states that f and g have the same invocation signature.

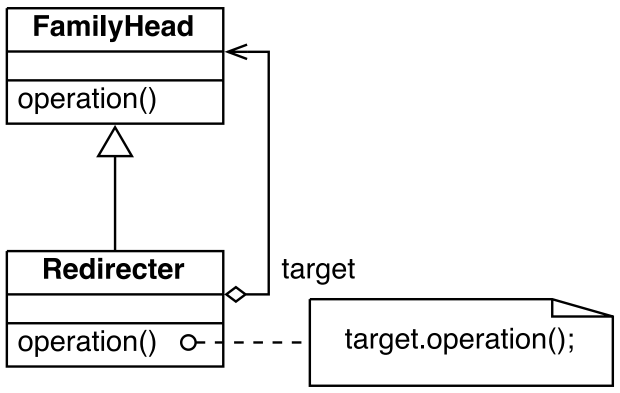


Figure 7. RedirectInFamily class structure

Our data field requirements for this paper are satisfied by the use of $\ll_f$. If $a.f \ll_f b$, then object a's method f uses the object b in some manner. All of the above have well defined transitivity properties, as well as hierarchical implications: if $a.f \ll b.g$, then obviously $a \ll b.g$, $a.f \ll b$, and $a \ll b$. Finally, we have $<:$, for inheritance (or more properly subsumption of type), showing a type reliance.

5.3. Example: RedirectInFamily

Consider the class diagram for the structure of the EDP $\mathbf{RedirectInFamily}$ [21], in Figure 7. Taken literally, it specifies that a class wishes to invoke a similar method (where similarity is evaluated based on the signature types of the methods, as hinted at by Beck's Intention Revealing Message best practice pattern [3]) to the one currently being executed, and it wishes to do so on an object of its parent class' type. This sort of open-ended structural recursion is a part of many patterns.

If we take the Participants specification of $\mathbf{RedirectInFamily}$, we find that:

- FamilyHead defines the interface, contains a method to be possibly overridden.

- Redirecter uses interface of FamilyHead through inheritance, redirects internal behaviour back to an instance of FamilyHead to gain polymorphic behaviour over an amorphous object structure.

We can express each of these requirements in ς-calculus:

$$FamilyHead \equiv [operation : A] \tag{1}$$

$$Redirecter <: FamilyHead \tag{2}$$

$$\begin{aligned} Redirecter \equiv [&target : FamilyHead, \\ &operation : A = \varsigma(x_i)\{target.operation\}] \end{aligned} \tag{3}$$

$$r : Redirecter \tag{4}$$

$$fh : FamilyHead \tag{5}$$

$$r.target = fh \tag{6}$$

This is a concrete implementation of the **RedirectInFamily** structure, but fails to capture the reliance of $Redirecter.operation$ on the behaviour of $FamilyHead.operation$. It also has an overly restrictive requirement concerning r's ownership of $target$, when compared to many implementations of this pattern. So, we introduce our reliance operators to produce a ρ-calculus definition:

$$r.operation \ll_{m+} r.target.operation \tag{7}$$

$$r \ll_f r.target \tag{8}$$

We can reduce two areas of indirection...

$$\frac{r.target = fh, r.operation \ll_{m+} r.target.operation}{r.operation \ll_{m+} fh.operation} \tag{9}$$

$$\frac{r \ll_f r.target, r.target = fh}{r \ll_f fh} \tag{10}$$

...and now we can produce a set of clauses to represent **RedirectInFamily**:

$$\frac{\begin{array}{l} Redirecter <: FamilyHead, \\ r : Redirecter, \\ fh : FamilyHead, \\ r.operation \ll_m fh.operation, \\ r \ll_f fh \end{array}}{\mathbf{RedirectInFamily}(Redirecter, FamilyHead, operation)} \tag{11}$$

5.4. Isotopes

Common wisdom holds that formalization of patterns in a mathematical notation will inevitably destroy the flexibility and elegance of patterns. An interesting side effect of expressing our EDPs in the ρ-calculus, however, is an *increased* flexibility in expression of code while conforming to the core *concept* of a pattern. We term variations of code expression that conform to the concepts and roles of an EDP *isotopes*.

Consider now Figure 8, which, at first glance, does not look much like our original specification. We have introduced a new class to the system, and our static criteria that the subclass' method invoke the superclass' instance has been replaced by a new calling chain. In fact, this construction looks quite similar to the transitional state while applying Martin Fowler's *Move Method* refactoring [10].

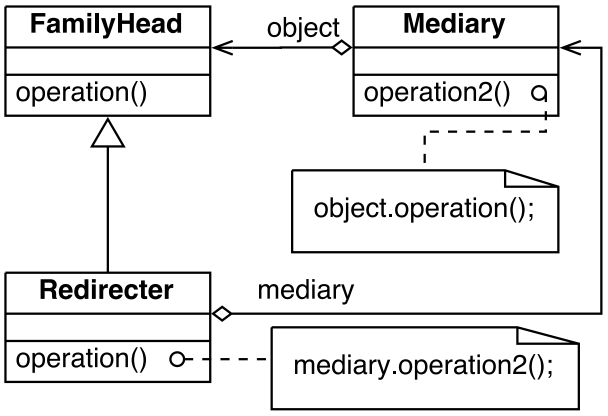


Figure 8. RedirectInFamily Isotope

We claim that this is precisely an example of a variation of **RedirectInFamily** when viewed as a series of formal constructs, as in Equations 12 through 20.

$$Redirection <: FamilyHead \tag{12}$$

$$r : Redirection \tag{13}$$

$$fh : FamilyHead \tag{14}$$

$$r.mediary = m \tag{15}$$

$$m.object = fh \tag{16}$$

$$r.operation \ll_{m-} r.mediary.operation2 \tag{17}$$

$$m.operation2 \ll_{m-} m.object.operation \tag{18}$$

$$r.operation \ll_f r.mediary \tag{19}$$

$$m.operation \ll_f m.object \tag{20}$$

If we start reducing this equation set, we find that we can perform an equality operation on Equations 15 and 17:

$$\frac{\begin{array}{c} r.operation \ll_{m-} r.mediary.operation2, \\ r.mediary = m \end{array}}{r.operation \ll_{m-} m.operation2} \tag{21}$$

We can now reduce this chain with Equation 18:

$$\frac{\begin{array}{c} r.operation \ll_{m-} m.operation2, \\ m.operation2 \ll_{m-} m.object.operation \end{array}}{r.operation \ll_{m+} m.object.operation} \tag{22}$$

$$\frac{r.operation \ll_{m+} m.object.operation, m.object = fh}{r.operation \ll_{m+} fh.operation} \tag{23}$$

Likewise, we can take Equations 15, 16, 19 and 20:

$$\frac{\begin{array}{c} r.operation \ll_f r.mediary, \\ m.operation \ll_f m.object, \\ r.mediary = m, \\ m.object = fh \end{array}}{r \ll_f fh} \quad (24)$$

If we now take Equations 12, 13, 14, 23, and 24 we find that we have satisfied the clause requirements set in our *original* definition of $\mathbf{RedirectInFamily}$, as per Equation 11. This alternate structure is an example of an *isotope* of the $\mathbf{RedirectInFamily}$ pattern, and required no adaptation of our existing rule. Our single rule takes the place of an enumeration of static pattern definitions. The concepts of *object relationships* and *reliance* are the key. It is worth noting that while this may superficially seem to be equivalent to the common definition of *variant*, as defined by Buschmann [5], there is a key difference: 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. Variants are not interchangeable without retooling the surrounding code, but isotopes are. This is an essential requirement of isotopes, and precisely why we chose the term. This flexibility in internal representation grants the implementation of the system a great degree of latitude, while still conforming to the abstractions given by design patterns.

6. Reconstruct known patterns

We can now demonstrate an example of using EDPs to express larger and well known design patterns. We begin with $\mathbf{AbstractInterface}$, a simple EDP, and build our way up to Decorator, visiting two other established patterns along the way.

6.1. AbstractInterface

$\mathbf{AbstractInterface}$ ensures that the method in a base class is truly abstract, forcing subclasses to override and provide their own implementations. The ρ-calculus definition can be given by simply using the trait construct of ς-calculus:

$$\frac{A \equiv [new : [l_i : A \rightarrow B_i{}^i \in 1...n], operation : A \rightarrow B]}{\mathbf{AbstractInterface}(A, operation)} \quad (25)$$

6.2. Objectifier

$\mathbf{Objectifier}$ is simply a class structure applying the Inheritance EDP to an instance of $\mathbf{AbstractInterface}$ pattern, where the $\mathbf{AbstractInterface}$ applies to all methods in a class. This is equivalent to what Woolf calls an Abstract Class pattern. Referring back to Figure 4 from our earlier discussion in section 3.2, we can see that the core concept is to create a family of subclasses with a common abstract ancestor. We can express this in ρ-calculus as:

$$\frac{\begin{array}{l} Objectifier : [l_i : B_i{}^{i \in 1...n}], \\ \mathbf{AbstractInterface}(Objectifier, l_i{}^{i \in 1...n}), \\ ConcreteObjectifier_j <: Objectifer^{j \in 1...m}, \\ Client : [obj : Objectifier] \end{array}}{\begin{array}{l} \mathbf{Objectifier}(Objectifier \\ \qquad ConcreteObjectifier_j{}^{j \in 1...m}, Client) \end{array}} \quad (26)$$

6.3. Object Recursion

We briefly described Object Recursion in section 3.2, and gave its class structure in Figure 5. We now show that this is a melding of the $\mathbf{Objectifier}$ and $\mathbf{RedirectInFamily}$ patterns, as illustrated in Figure 9. The annotations indicate which roles of which patterns the various components of $\mathbf{ObjectRecursion}$ play. A formal EDP representation is given in Equation 27.

$$\frac{\begin{array}{l} \mathbf{Objectifier}(Handler, Recurser_i{}^{i \in 1...m}, Initiator), \\ \mathbf{Objectifier}(Handler, Terminator_j{}^{j \in 1...n}, \\ \qquad Initiator), \\ init \ll_m obj.handleRequest, \\ init : Initiator, \\ obj : Handler, \\ \mathbf{RedirectInFamily}(Recurser, Handler, \\ \qquad handleRequest), \\ !\mathbf{RedirectInFamily}(Terminator, Handler, \\ \qquad handleRequest) \end{array}}{\begin{array}{l} \mathbf{ObjectRecursion}(Handler, Recurser_i{}^{i \in 1...m}, \\ \qquad Terminator_j{}^{j \in 1...n}, Initiator) \end{array}} \quad (27)$$

6.4. ExtendMethod

The $\mathbf{ExtendMethod}$ EDP is used to extend, not replace, the functionality of an existing method in a superclass. Figure 10 shows the structure of the pattern, illustrating the use of **super**, formalized in Equation 28.

$$\frac{\begin{array}{l} OriginalBehaviour : [l_i : B_i{}^{i \in 1...m}, operation : B_{m+1}], \\ ExtendedBehaviour <: OriginalBehaviour, \\ eb : ExtendedBehaviour, \\ eb.operation \ll_{m+} \mathbf{super}.operation \end{array}}{\begin{array}{l} \mathbf{ExtendMethod}(OriginalBehaviour, \\ \qquad ExtendedBehaviour, operation) \end{array}} \quad (28)$$

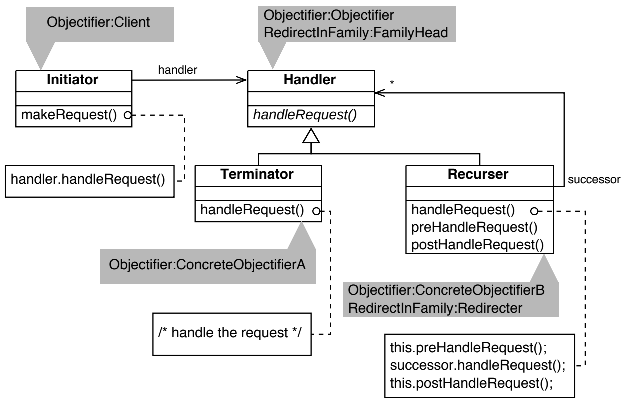


Figure 9. Object Recursion, annotated to show roles

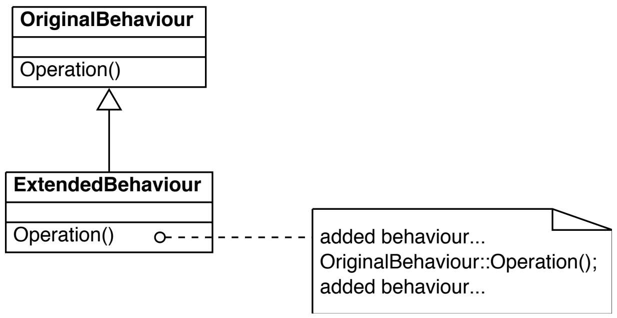


Figure 10. ExtendMethod

6.5. Decorator

Now we can produce a pattern directly from the GoF text, the $\mathbf{Decorator}$ pattern. Figure 11 is the standard class diagram for $\mathbf{Decorator}$ annotated to show how the $\mathbf{ExtendMethod}$ and $\mathbf{ObjectRecursion}$ patterns interact. Again, we provide a formal definition in Equation 29, although only for the method extension version (the field extension version is similar but unnecessary for our purposes here). The keyword **any** indicates that any object of any class may take this role, as long as it conforms to the definition of $\mathbf{ObjectRecursion}$.

We have now created a formally sound definition of how to solve a problem in software architecture design. This definition is now subject to formal analysis, discovery, and metrics, and, following our example of pattern composition, can be used as a building block for larger, even more intricate patterns that are *incrementally* comprehensible. At the same time, we believe we have retained the flexibility of implementation, and the conceptual semantics of the pattern, by making precise choices of abstraction at each stage of the composition. Furthermore, by building this approach on an existing denotational semantics for object oriented programming (ς-calculus), we continue to be able to process the same system at an extremely low level. One of the key contributions of this system, however, is that the practitioner can *choose* on which level to operate, and perform the analyses and tasks which are suitable without losing the flexibility of integrating other layers of analysis at a later date.

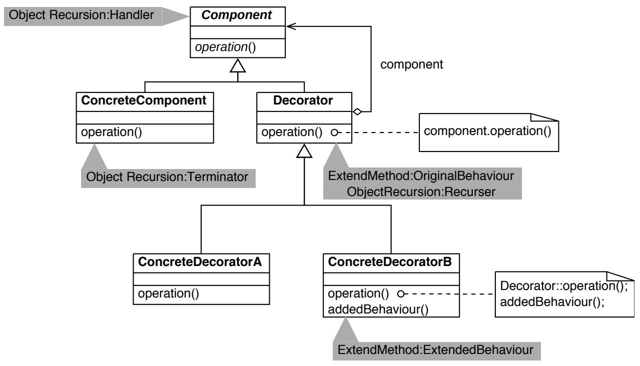


Figure 11. Decorator annotated to show EDP roles

7. Widget, Inc. revisited

So how does this help our intrepid engineers at Widget, Inc? Let us start with their source code, and assume that the Killer Widget compiler system can produce a diagnostic parse tree, as the GNU `gcc` system does. A syntactic analysis of the parse tree and translation into ρ-calculus gives us a large body of facts about the system, a very few of which are given in Equations 30 through 42.

At this point we can encode the above ρ-calculus facts into a form usable by one of several automated theorem provers, such as OTTER [13], and search for the most basic

$$\frac{\begin{array}{l}\mathbf{ObjectRecursion}(Component, Decorator_i^{\,i\in 1\ldots m},\\ \quad ConcreteComponent_j^{\,j\in 1\ldots n}, \mathbf{any}),\\ \mathbf{ExtendMethod}(Decorator,\\ \quad ConcreteDecoratorB_k^{\,k\in 1\ldots o}, operation_k^{k\in 1\ldots o}),\end{array}}{\begin{array}{l}\mathbf{Decorator}(Component, Decorator_i^{\,i\in 1\ldots m},\\ \quad ConcreteComponent_j^{\,j\in 1\ldots n},\\ \quad ConcreteDecoratorB_k^{\,k\in 1\ldots o},\\ \quad ConcreteDecoratorA_l^{\,l\in 1\ldots p},\\ \quad operation_k^{k\in 1\ldots o+p})\end{array}} \tag{29}$$

$$File \equiv [op1 : File \rightarrow []] \tag{30}$$
$$FileFAT <: File \tag{31}$$
$$fp : FilePile \tag{32}$$
$$FilePile <: File \tag{33}$$
$$fp.op1 <<_{m-} mfile.op2 \tag{34}$$
$$FilePile.mfiles <<_{f} MeasuredFile \tag{35}$$
$$mf : MeasuredFile \tag{36}$$
$$MeasuredFile.file <<_{f} File \tag{37}$$
$$MeasuredFile.op2 <<_{f} MeasuredFile.file \tag{38}$$
$$mf.op2 <<_{m-} file.op1 \tag{39}$$
$$fpf : FilePileFixed \tag{40}$$
$$FilePileFixed <: FilePile \tag{41}$$
$$FilePileFixed.op1 <<_{m+} \mathbf{super}.op1 \tag{42}$$

$$\frac{\begin{array}{l} FilePile <: File, \\ fp : FilePile, \\ f : File, \\ fp.op1 \ll_{m-} fp.mfile.op2, \\ fp.mfile = mf, \\ mf.op2 \ll_{m-} mf.file.op2, \\ mf.file = f, \\ fp.op1 \ll_{f} fp.mfile, \\ mf.op2 \ll_{f} mf.file \end{array}}{\mathbf{RedirectInFamily}(FilePile, File, op1)} \tag{44}$$

EDPs, then work our way up to more complex patterns, as in our $\mathbf{Decorator}$ example.

We can quickly see that Eq 30 fulfills our $\mathbf{AbstractInterface}$ rule for class $File$, method $op1$. Furthermore, $File$ and $FilePile$ fulfill the requirements of the $\mathbf{Objectifier}$ pattern, assuming, as we will here assert, that the remainder of $File$'s methods are likewise abstract.

$$\frac{\begin{array}{l} File : [op1 : []], \\ \mathbf{AbstractInterface}(File.op1), \\ FilePile <: File, \\ MeasuredFile <<_{f} File \end{array}}{\mathbf{Objectifier}(File, FilePile, MeasuredFile)} \tag{43}$$

$\mathbf{Objectifier}(File, FileFAT, MeasuredFile)$ and analogous instances of $\mathbf{Objectifier}$ for the other concrete subclasses of the File class, can be similarly derived.

Finding an instance of $\mathbf{RedirectInFamily}$ is a bit more complex, and requires the use of our isotopes. Following the example in Section 5.4, however, it becomes straight forward to derive $\mathbf{RedirectInFamily}$,

It can be shown also that one *cannot* derive $\mathbf{RedirectInFamily}$(FileFAT, File, op1). We now

$$\frac{\begin{array}{l} \mathbf{Objectifier}(File, FilePile, MeasuredFile), \\ \mathbf{Objectifier}(File, FileFAT, MeasuredFile), \\ mf : MeasuredFile, \\ mf \ll mfile.op1, \\ file : File, \\ \mathbf{RedirectInFamily}(FilePile, File, op1), \\ \mathbf{!RedirectInFamily}(FileFAT, File, op1) \end{array}}{\begin{array}{l} \mathbf{ObjectRecursion}(File, FilePile, \\ \qquad FileFAT, MeasuredFile) \end{array}} \tag{45}$$

see that $\mathbf{ObjectRecursion}$ derives cleanly from Equations 43 and 44 and their analogues, in Equation 45. $\mathbf{ExtendMethod}$ is a simple derivation as well:

$$\frac{\begin{array}{l} FilePile \equiv [op1 : \mathbf{any}], \\ FilePileFiled <: FilePile, \\ fpf : FilePileFixed, \\ fpf.op1 \ll_{m+} \mathbf{super}.op1 \end{array}}{\mathbf{ExtendMethod}(FilePile, FilePileFixed, op1)} \tag{46}$$

Finally, we arrive at the uncovering of a full $\mathbf{Decorator}$ pattern:

$$\frac{\begin{array}{l} \mathbf{ObjectRecursion}(File, FilePile, FileFAT, \\ \qquad MeasuredFile), \\ \mathbf{ExtendMethod}(FilePile, FilePileFixed, op1), \end{array}}{\begin{array}{l} \mathbf{Decorator}(File, FilePile, FileFAT, \\ \qquad FilePileFixed, op1) \end{array}} \tag{47}$$

Similarly, we can uncover the latent $\mathbf{Composite}$ pattern in the architecture. Both GoF pattern implementations are annotated in Figure 12. The intermediate patterns have been left out for clarity, as have finer granularity relationships. The annotations indicate which classes fulfill which roles in the pattern descriptions, as *Pattern::Role*. Note that a single class can fulfill more than one role in more than one pattern.

The revealed patterns are, to be honest, not hard to spot in this small example. Real life, however, tends to leave us with a lack of sufficient documentation, and even reverse engineering tools that extract architectural diagrams are not going to explicitly reveal the hidden patterns in a system of several hundred classes. In the cases where pattern recognition does occur, it frequently relies on the implementation of patterns to be an exact match to some predefined template. Isotopes remove this restriction, instead letting the relationships in the code reduce to reliance paths in a natural way. This formalized method is useful precisely because it is can be made automatic, deriving from syntactic analysis of the parse tree of the original source code a system of facts about the architecture, and then using theorem solving

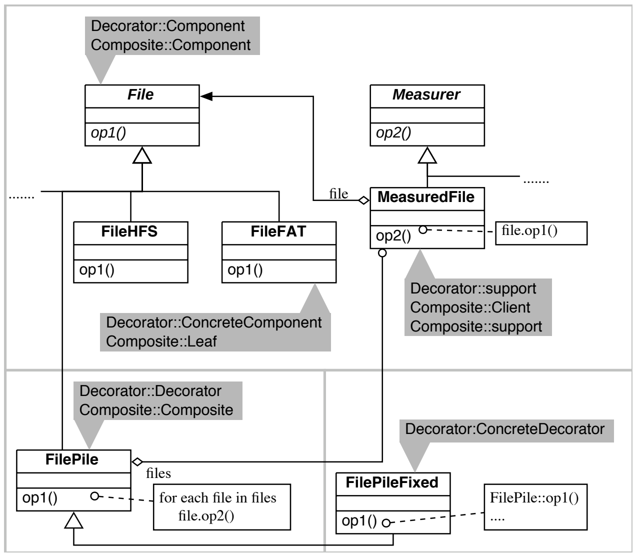


Figure 12. Discovered patterns

systems such as OTTER, to produce explicit illustrations of pattern implementation. Just such a tool set is currently under development and industry validation test cases are available.

At this point the review team at Widgets, Inc. can quickly see that there are areas that could use some conceptual cleaning, have been given pointers as to where the problem lies, and have also been shown which classes and methods are required for each pattern to continue working. This last point may direct the team to start refactoring in a meaningful and well-defined manner.

8. 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, 1999. Dissertation Draft.
[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] 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.
[13] 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.
[14] S. Meyers. *Effective C++*. Addison-Wesley, 1992.
[15] 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.
[16] M. Ó Cinnéide. *Automated Application of Design Patterns: A Refactoring Approach*. Ph.D. dissertation, University of Dublin, Trinity College, 2001.
[17] 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.
[18] W. Pree. *Design Patterns for Object-Oriented Software Development*. Addison-Wesley, 1994.
[19] 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.
[20] 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.
[21] J. M. Smith and D. Stotts. Elemental design patterns: A link between architecture and object semantics. Technical Report TR-02-011, Univ. of North Carolina, 2002.
[22] B. Woolf. The abstract class pattern. In N. Harrison, B. Foote, and H. Rohnert, editors, *Pattern Languages of Program Design 4*. Addison-Wesley, 1998.
[23] B. Woolf. The object recursion pattern. In N. Harrison, B. Foote, and H. Rohnert, editors, *Pattern Languages of Program Design 4*. Addison-Wesley, 1998.
[24] 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.