Technical Report TR03-016

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

SPQR: Flexible Automated Design Pattern Extraction from Source Code

Jason McC. Smith and David Stotts

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

stotts@cs.unc.edu

May 21, 2003

SPQR: Flexible Automated Design Pattern Extraction From Source Code

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

Previous automated 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, System for Pattern Query and Recognition*, or* SPQR*, differs in that we do not seek statically to encode each pattern and each variant that we wish to find. 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. We use a logical inference system to reveal large numbers of patterns and their variations from a small number of definitions by encoding 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 chain of fully automated tools provides a path from source code to revealed patterns. We describe our approach in this paper with a concrete example to drive the discussion, accompanied by formal treatment of the foundational topics.*

1 Introduction

Practical tool support consistently lags 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 [13]. 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)[24, 25], and they are the base concepts on which 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. 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 evaded such formalization to date. We show here that such a formalization is possible, and in addition that it can meet certain essential criteria. We also show how our 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 truly to 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 discussing related work in the fields of pattern decomposition and automated pattern extraction. We then describe our driving problem and provide a concrete example system. We show how to derive an instance of the Decorator design pattern from our example scenario using SPQR, then illustrate how this is accomplished using automatable reduction rules that are processed by a theorem prover. We manually illustrate the mechanism underlying our method with a chain of pattern composition from our EDPs to the Decorator pattern. We then show how these EDPs can be formally expressed in a version of the sigma (ς) calculus [1], that we have extended with *reliance operators* to form the ρ-calculus, by which we justify the use of an automated theorem prover. We conclude with a discussion of future research directions, and provide detailed discussions of the formalisms involved.

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 [7, 14, 21, 30]. 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.

2.1 Refactoring approaches

Attempts to formalize refactoring [12] exist, and have met with fairly good success to date[8, 16, 19]. 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 [11], Eden's work on LePuS [9], and Ó Cinnéide's work in transformation and refactoring of patterns in code [17] through the application

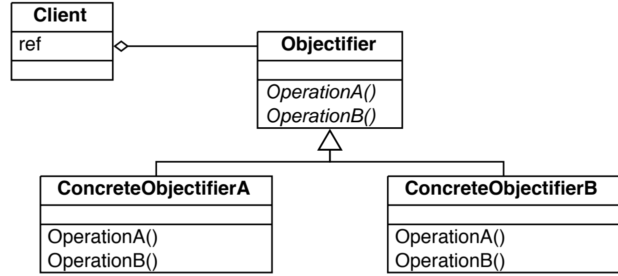

Figure 1. Objectifier

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 [13] reveals many shared structural and behavioural elements, such as the similarities between Composite and Visitor [13]. 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, 7, 21, 28, 29, 30].

Objectifier: The Objectifier pattern [30] 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 [29]. The class diagram in Figure 2 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.

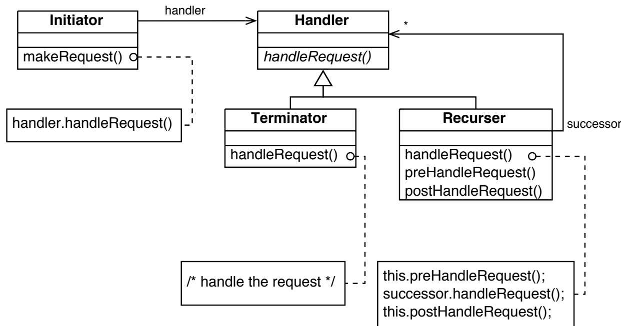

Figure 2. Object Recursion

2.3 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 [6], Coplien and others' idioms [3, 7, 15], and Pree's metapatterns [20] all support this viewpoint. Shull, Melo and Basili's BACKDOOR's [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 4.4. A related, but type-based approach that works instead on UML expressed class designs, is Egyed's UML/Analyzer system [10] which uses abstraction inferences to help guide engineers in code discovery.

3 Using SPQR

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 often normal in 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.

Three main phases of development by three different teams have taken place on a core library used by the application, resulting in a conceptually unclear system, shown in Figure 3. The first phase involved the File system having a MeasuredFile metric gathering suite wrapped around it.

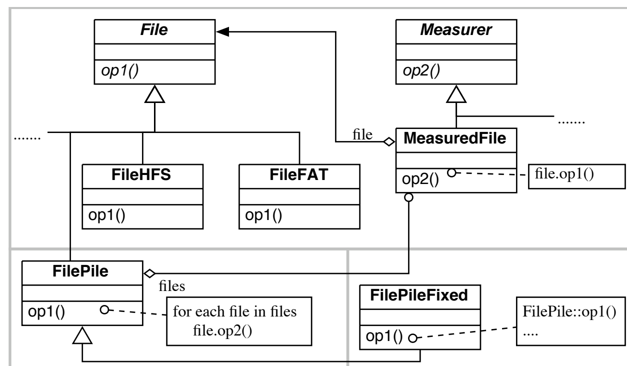

Figure 3. Killer Widget

```
class File {
        virtual void op1();
};

class MeasuredFile {
        File*   file;
        void op2() {    file.op1(); };
};

class FileFAT : File {
        void op1();
};

class FilePile : File {
        MeasuredFile*   mfile;
        void op1() {    foreach file in mfile:
                                file.op2(); };
};

class FilePileFixed : FilePile {
        void op1() {    FilePile::op1();
                        fixTheProblem(); };
};
```

Figure 4. Killer Widget pseudo-code snippet

Secondly, multiple file handling was added by the FilePile abstraction, and lastly, a bug fix was added in the FilePile-Fixed class to work around an implementation error that become ubiquitously assumed. A review of the design is called for the next development cycle.

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

We describe here our chain of tools from the viewpoint of a practitioner using them on Killer Widget. This toolset, the System for Pattern Query and Recognition, is comprised of several components, shown in Figure 5. From the engi-

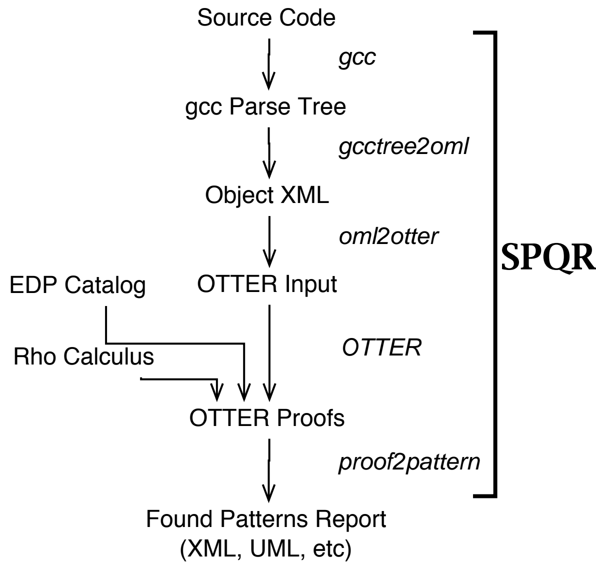

Figure 5. SPQR Outline

neer's point of view, SPQR is a single 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*.

We describe here the current set of tools in the SPQR toolset. First, source code is analyzed for particular syntactic constructs that correspond to the ρ-calculus concepts we are interested in. It turns out that the ubiquitous `gcc` has the ability to emit an abstract syntax tree suitable for such analysis. Our first tool, *gcctree2oml*, reads this tree file and produces an XML description of the object structure features. We chose an intermediary step so that various back ends could be used to input source semantics to SPQR. A second tool, *oml2otter* then reads this Object XML file and produces a feature-rule input file to the automated theorem prover, in the current package we are using Argonne National Laboratory's *OTTER*. OTTER finds instances of design patterns by inference based on the rules outlined in this paper. Finally, *proof2pattern* analyzes the OTTER proof output and produces an Object XML pattern description report that can be used for further analysis, such as the production of UML diagrams.

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, perhaps local to a specific institution or workgroup, they can be added to the catalog used for query.

The Killer Widget example has been successfully analyzed and a salient **Decorator** pattern was found using SPQR. The inputs to OTTER include the set of facts of the system under consideration (shown in Figure 6, as would be output by *gcctree2oml* and *oml2otter* from the code snippet in Figure 4), the necessary elements of ρ-calculus encoded as OTTER rules, and the design patterns of interest, including the EDPs, similarly encoded (both of which are provided as part of SPQR). For example, the **RedirectInFamily** pattern is shown in Figure 7.

```
%%% Current environment
list(sos).
File declares op1.
FileFAT inh File.
fp : FilePile.
FilePile inh File.
(fp dot op1) relmd ((fp dot mfile) dot op2).
fp.mfile = mf.
mf : MeasuredFile.
(mf dot file) relf f.
f : File.
(mf dot op2) relf (mf dot file).
(mf dot op1) relmd (file dot op1).
(mf dot file) = f.
fpf : FilePileFixed.
FilePileFixed inh FilePile.
(fp dot op1) relms ((fp dot super) dot op1).
(fp dot op1) relf ((fp dot m) dot file).
end_of_list.
```

Figure 6. Killer Widget as OTTER Input

The work required of the developer is to simply request SPQR to perform the analysis, and the resultant found patterns are reported by *proof2pattern* as an Object XML snippet, such as:

```
<pattern name="Decorator">
    <role name="Component"> "File" </role>
    <role name="Decorator"> "FilePile" </role>
    <role name="ConcreteComponent">
        "FileFAT" </role>
    <role name="ConcreteDecorator">
        "FilePileFixed" </role>
    <role name="operation"> "op1" </role>
</pattern>
```

Such information can then be used to produce diagrams such as Figure 8, done by hand. 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, such as *Pattern::Role*. Note that a single class can fulfill more than one role in more than one pattern.

Our preliminary speed results indicate that scaling to larger systems in production code should be effective. Our tolerance threshhold is whether or not the SPQR analysis is roughly equivalent to compilation of the same source code, and to date this has held true in informal tests.

```
all Redirecter FamilyHead r fh operation (
    (Redirecter inh FamilyHead) &
    (r : Redirecter) &
    (fh : FamilyHead) &
    ((r dot operation) relm (fh dot operation)) &
    (r relf fh) ->

    (RedirectInFamily(Redirecter, FamilyHead,
                                operation))
).
```

Figure 7. RedirectInFamily as OTTER input

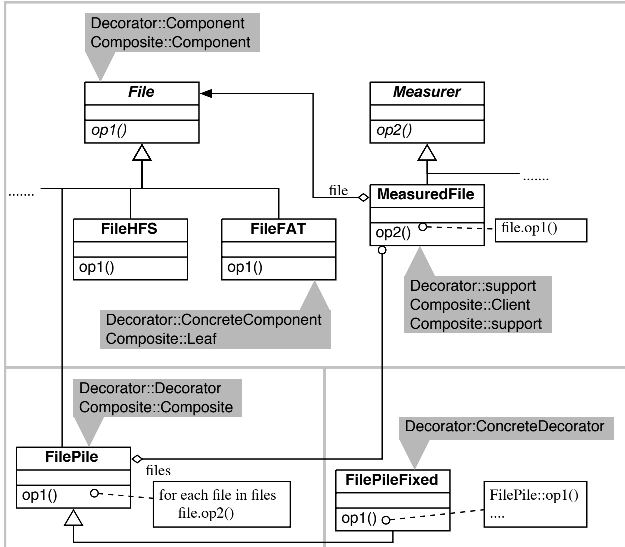

Figure 8. Primary discovered pattern roles

We 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.

4 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.

4.1 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.

We will only need describe a small subset of ς-calculus for the purposes of this paper. Specifically, we will need the concepts of type definition, object typing, and type subsumption (inheritance). A type T is defined by $T \equiv [...]$, where the contents of the brackets are method and field definitions. An object $\mathcal{O}$ is shown to be of type T by $\mathcal{O} : T$. If type T' is a subtype of type T, such as it would be under inheritance, then $T' <: T$.

4.2 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; likewise the word ‘dependency’ has many complementary definitions already in use), are direct, quantifiable expressions of whether one element (an object, method, or field), in any way relies or depends on the existence 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 other notation systems such as 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.

See [27] for a formal treatment of the ρ-calculus and its definition. Informally, the reliance operator $<$ has three forms: a method invocation reliance ($<_\mu$), a field access reliance ($<_\phi$), and a generalized reliance ($<_\gamma$). The $<_\mu$ has an optional annotation to indicate a similarity association $(+)$, or a dissimilarity association $(-)$, as defined below.

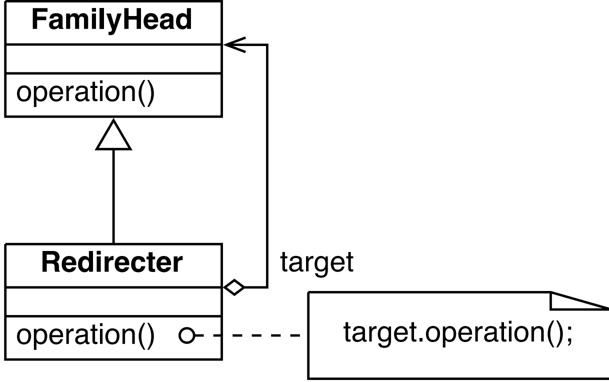

Figure 9. RedirectInFamily class structure

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

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

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

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

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

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

4.3 Example: RedirectInFamily

Consider the class diagram for the structure of the EDP **RedirectInFamily** [26], in Figure 9. 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]: equivalent signature are 'similar', inequivalent signatures are 'dissimilar') 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 **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, as in Equations 1 through 6.

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

$$r.operation <_{\mu+} r.target.operation \tag{7}$$

$$r <_{\phi} r.target \tag{8}$$

We can reduce two areas of indirection...

$$\frac{r.target = fh, r.operation <_{\mu+} r.target.operation}{r.operation <_{\mu+} fh.operation} \tag{9}$$

$$\frac{r <_{\phi} r.target, r.target = fh}{r <_{\phi} fh} \tag{10}$$

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

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

4.4 Isotopes

Conventional 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 10, 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 [12].

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.

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

$$\frac{r.operation <_{\mu-} r.mediary.operation2, \; r.mediary = m}{r.operation <_{\mu-} m.operation2} \tag{21}$$

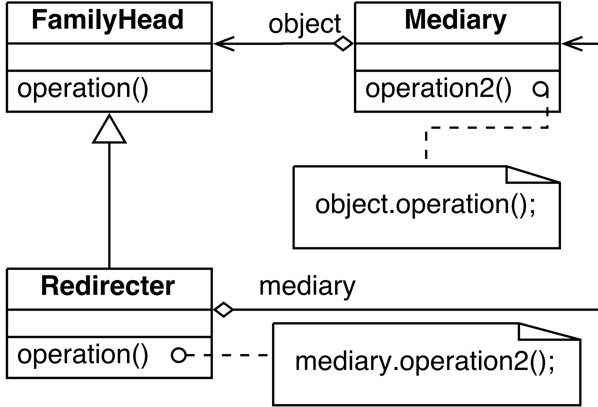

Figure 10. RedirectInFamily Isotope

We can now reduce this chain under transitivity with Equation 18:

$$\frac{r.operation <_{\mu-} m.operation2, \quad m.operation2 <_{\mu-} m.object.operation}{r.operation <_{\mu+} m.object.operation} \tag{22}$$

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

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

$$\frac{r.operation <_{\phi} r.mediary, \quad m.operation <_{\phi} m.object, \quad r.mediary = m, \quad m.object = fh}{r <_{\phi} fh} \tag{24}$$

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*

$$Redirecter <: FamilyHead \tag{12}$$
$$r : Redirecter \tag{13}$$
$$fh : FamilyHead \tag{14}$$
$$r.mediary = m \tag{15}$$
$$m.object = fh \tag{16}$$
$$r.operation <_{\mu-} r.mediary.operation2 \tag{17}$$
$$m.operation2 <_{\mu-} m.object.operation \tag{18}$$
$$r.operation <_{\phi} r.mediary \tag{19}$$
$$m.operation <_{\phi} m.object \tag{20}$$

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 [6], there is a key difference: encapsulation. Isotopes may differ from 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.

5 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 $\mathbf{Decorator}$, visiting two other established patterns along the way.

$\mathbf{AbstractInterface}$ 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:

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

$\mathbf{Objectifier}$ 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 1 from our earlier discussion in Section 2.2, 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:

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

We briefly described Object Recursion in section 2.2, and gave its class structure in Figure 2. We now show that this

is a melding of the $\mathbf{Objectifier}$ and $\mathbf{RedirectInFamily}$ patterns.

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

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

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

Now we can produce a pattern directly from the GoF text, the $\mathbf{Decorator}$ pattern. 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 $\mathbf{ObjectRecursion}$.

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

We have created a formally sound definition of a description of how to solve a problem of software architecture design. This definition is now subject to formal analysis, discovery, and metrics. Following our example of pattern composition, this definition 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 that patterns demand. Also, we believe that we have retained the conceptual semantics of the pattern by intelligently and diligently making precise choices at each stage of the composition.

$$File \equiv [op1 : File \rightarrow []] \tag{30}$$
$$FileFAT <: File \tag{31}$$
$$fp : FilePile \tag{32}$$
$$FilePile <: File \tag{33}$$
$$fp.op1 <_{\mu-} fp.mfile.op2 \tag{34}$$
$$fp.mfile = mf \tag{35}$$
$$mf : MeasuredFile \tag{36}$$
$$mf.file <_\phi f \tag{37}$$
$$f : File \tag{38}$$
$$mf.op2 <_\phi mf.file \tag{39}$$
$$mf.op2 <_{\mu-} mf.file.op1 \tag{40}$$
$$mf.file = f \tag{41}$$
$$fpf : FilePileFixed \tag{42}$$
$$FilePileFixed <: FilePile \tag{43}$$
$$fp.op1 <_{\mu+} \mathbf{super}.op1 \tag{44}$$
$$fp.op1 <_\phi fp.mfile \tag{45}$$

Figure 11. Killer Widget as ρ-calculus

6 Killer Widget analysis in detail

We can now build a picture of what SPQR is doing within OTTER's inference system. The equations in Figure 11 are the equivalent of the *oml2otter* output we saw in Figure 6.

We can quickly see that our $\mathbf{AbstractInterface}$ rule is fulfilled for class $File$, method $op1$ by Equation 30. 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.

$$\frac{\begin{array}{l}File : [op1 : []],\\ \mathbf{AbstractInterface}(File.op1),\\ FilePile <: File,\\ mfile <_\phi file,\\ file : File\end{array}}{\mathbf{Objectifier}(File, FilePile, MeasuredFile)} \tag{46}$$

$\mathbf{Objectifier}(File, FileFAT, MeasuredFile)$ 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. Follow-

ing the example in Section 4.4, however, it becomes straight forward to derive $\mathbf{RedirectInFamily}$:

$$\frac{\begin{array}{l} FilePile <: File, \\ fp : FilePile, \\ f : File, \\ fp.op1 <_{\mu-} fp.mfile.op2, \\ fp.mfile = mf, \\ mf.op2 <_{\mu-} mf.file.op2, \\ mf.file = f, \\ fp.op1 <_{\phi} fp.mfile, \\ mf.op2 <_{\phi} mf.file \end{array}}{\mathbf{RedirectInFamily}(FilePile, File, op1)} \tag{47}$$

It can also be shown that one simply *cannot* derive the fact **RedirectInFamily**(FileFAT, File, op1). We now see that **ObjectRecursion** derives cleanly from Equations 46 and 47 and their analogues, in Equation 48.

$$\frac{\begin{array}{l} \mathbf{Objectifier}(File, FilePile, MeasuredFile), \\ \mathbf{Objectifier}(File, FileFAT, MeasuredFile), \\ mf : MeasuredFile, \\ mf <_{\mu} file.op1, \\ file : File, \\ \mathbf{RedirectInFamily}(FilePile, File, op1), \\ !\mathbf{RedirectInFamily}(FileFAT, File, op1) \end{array}}{\begin{array}{c} \mathbf{ObjectRecursion}(File, FilePile, \\ FileFAT, MeasuredFile) \end{array}} \tag{48}$$

ExtendMethod is a simple derivation as well:

$$\frac{\begin{array}{l} FilePile \equiv [op1 : \mathbf{any}], \\ FilePileFiled <: FilePile, \\ fpf : FilePileFixed, \\ fpf.op1 <_{\mu+} \mathbf{super}.op1 \end{array}}{\mathbf{ExtendMethod}(FilePile, FilePileFixed, op1)} \tag{49}$$

Finally, we arrive at the uncovering of a full **Decorator** pattern:

$$\frac{\begin{array}{l} \mathbf{ObjectRecursion}(File, FilePile, FileFAT, \\ \quad MeasuredFile), \\ \mathbf{ExtendMethod}(FilePile, FilePileFixed, op1), \end{array}}{\begin{array}{c} \mathbf{Decorator}(File, FilePile, FileFAT, \\ FilePileFixed, op1) \end{array}} \tag{50}$$

7 Future Work

Several branches of future research are natural advances to build on the foundation we have outlined here. They cover continued source code analysis and comprehension assistance, cued support for re-factoring, and continued SPQR tool production.

SPQR was designed to be modular, and easily extended. Future possibilities include further language/compiler integration, perhaps through IDE plugins for systems such as Eclipse, .NET, or ProjectBuilder. As design patterns continue to be discovered and described, they can be easily added to the current catalog of ρ-calculus rules. Post-processing could include automated UML diagram production, integrated report production, or perhaps enhancement of IDE workflow.

Validation must continue on large codebases from industry to establish correct heuristics to manage the level of detail of found patterns. A report that indicated all instances of **AbstractInterface**, for example, would be unwieldy and cumbersome. We envision allowing the developer to choose a level of complexity on the pattern dependency graph, below which discovered patterns will be culled from the final report.

Refactoring is not likely to ever be, in our opinion, a fully automatable process. Several key pieces, however, may benefit from the work outlined in this paper. Our isotope example in Section 4.4 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[12, 19, 16]. Ó Cinnéide's mini-transformations likewise could be formally verified and applied not only to existing patterns, but also perhaps to code that is not yet considered pattern-ready, as key relationships are deduced from a formal analysis[17, 18]. Furthermore, we believe the fragments-based systems such as LePuS can now be integrated back into the larger domain of denotational semantics.

Finally, we revisit the original motivation for this research, to reduce the time and effort required for an engineer to comprehend a system's architecture well enough to guide the maintenance and modification thereof. We believe that the approach outlined in the paper, along with the full catalog of EDPs and ρ-calculus, can form a formal basis for some very powerful source code analysis tools such as *Choices*[22], or KT[5], that operate on a higher level of abstraction than just "class, object and method interactions"[22]. Discovery of patterns in an architecture should be become much more possible than it is today, and we 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.

8 Conclusion

We have presented a System for Pattern Query and Recognition (SPQR), a toolset for the support of a suite of simple design patterns, 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.

9 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] K. Brown. Design reverse-engineering and automated design pattern detection in smalltalk. Master's thesis, North Carolina State University, 2000.

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

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

[8] 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.

[9] 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.

[10] A. Egyed. Automated abstraction of class diagrams. *ACM Transactions on Software Engineering and Methodology*, 11(4):449–491, Oct. 2002.

[11] 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.

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

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

[14] 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.

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

[16] 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.

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

[18] M. Ó Cinnéide and P. Nixon. 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*, July 1998.

[19] 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.

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

[21] 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.

[22] M. Sefika, A. Sane, and R. H. Campbell. Architecture-oriented visualization. In *Proc. of the eleventh annual conference on Object-oriented programming systems, languages, and applications*, pages 389–405. ACM Press, 1996.

[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. An elemental design pattern catalog. Technical Report TR-02-040, Univ. of North Carolina, 2002.

[25] J. M. Smith and D. Stotts. Elemental design patterns: A formal semantics for composition of oo software architecture. In *Proc. of 27th Annual IEEE/NASA Software Engineering Workshop*, dec 2002.

[26] 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.

[27] J. M. Smith and D. Stotts. Spqr: Use of a first-order theorem prover for flexibly finding design patterns in source code. Technical Report TR-03-007, Univ. of North Carolina, 2003.

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

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

[30] 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.