Technical Report TR03-007

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

SPQR: Use of a First-Order Theorem Prover for Flexibly Finding Design Patterns in Source Code

Jason McC. Smith and David Stotts

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

smithja@cs.unc.edu
stotts@cs.unc.edu

March 21, 2003

SPQR: Use of a First-Order Theorem Prover for Flexibly Finding Design Patterns in 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 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. 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*). We then use a logical inference system 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. We describe our approach in this paper with a concrete example to drive the discussion, accompanied by formal treatment of the foundational topics.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications—*design languages, object-oriented languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*patterns*; F.4.1 [**Mathematical Logic**]: [lambda calculus and related systems]; D.2.11 [**Software Engineering**]: Software Architecture—*patterns*; D.2.7 [**Software Engineering**]: Distribution, Maintenance, Enhancement—*restructuring, reverse engineering, and reengineering*; D.3.1 [**Programming Languages**]: Formal Definition and Theory

General Terms

design, languages, measurement, theory

Keywords

design patterns, elemental design patterns, sigma calculus, rho calculus, pattern decomposition, pattern identification, refactoring, education

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 [17]. 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 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 describing our driving problem and provide a concrete example system. We then discuss related work in the fields of pattern decomposition and automated pattern extraction, leading into an introduction of 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* to form the ρ-calculus. 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. We conclude with a discussion of future research directions, and provide detailed discussions of the formalisms involved.

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 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 1. The first phase involved the File system having a MeasuredFile metric gathering suite wrapped around it. Secondly, multiple file handling was added by the FilePile abstraction, and lastly, a bug fix was added in the FilePileFixed 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.

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 [11, 24, 35, 43]. 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

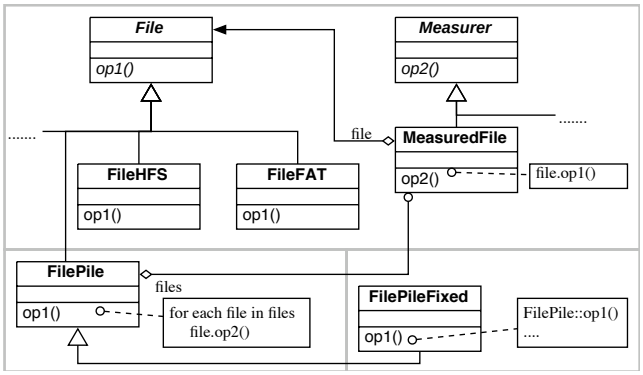

Figure 1: Killer Widget

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 [16] exist, and have met with fairly good success to date[12, 28, 31]. 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 [15], Eden's work on LePuS [13], and Ó Cinnéide's work in transformation and refactoring of patterns in code [29] through the application 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 [17] reveals many shared structural and behavioural elements, such as the similarities between Composite and Visitor [17]. 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 [5, 11, 35, 41, 42, 43].

Objectifier: The Objectifier pattern [43] 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

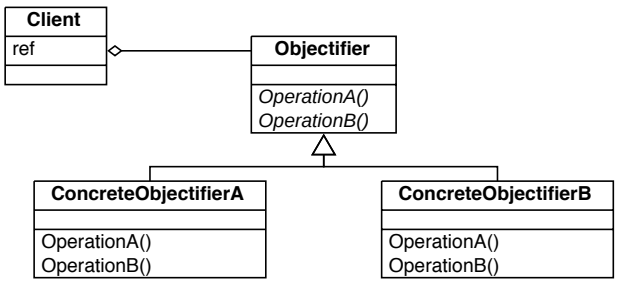

Figure 2: Objectifier

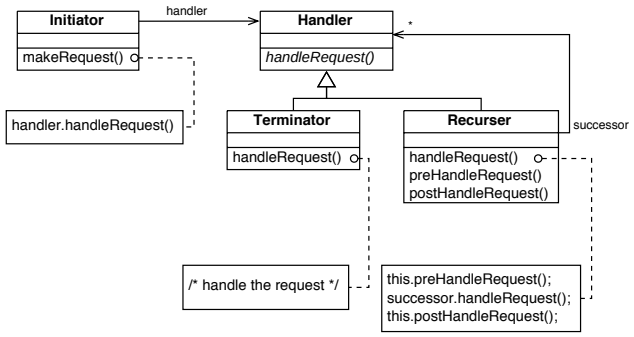

Figure 3: Object Recursion

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 [42]. The class diagram in Figure 3 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.

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 [8], Coplien and others' idioms [4, 11, 26], and Pree's metapatterns [34] all support this viewpoint. Shull, Melo and Basili's BACKDOOR's [38] 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. A related, but type-based approach that works instead on UML expressed class designs, is Egyed's UML/Analyzer system [14] which uses abstraction inferences to help guide engineers in code discovery.

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 4: Elemental Design Patterns

4. THE EDP CATALOG

In this paper we present our sixteen identified EDPs, in Figure 4, with a discussion of their generation and traits in Section 11.1. We do not claim that this list covers all the possible permutations of interactions, but that these are the core catalog of EDPs upon which others will be built. These EDPs can be found in complete detail as design pattern presentations in [39].

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.

Sigma calculus 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.

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 inheritnace, then $T' <: T$.

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; 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 Section 11.4 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 and discussed in detail in Section 11.1.1.

5.3 Example: RedirectInFamily

Consider the class diagram for the structure of the EDP **RedirectInFamily** [40], in Figure 5. 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 [4]: equivalent signature are

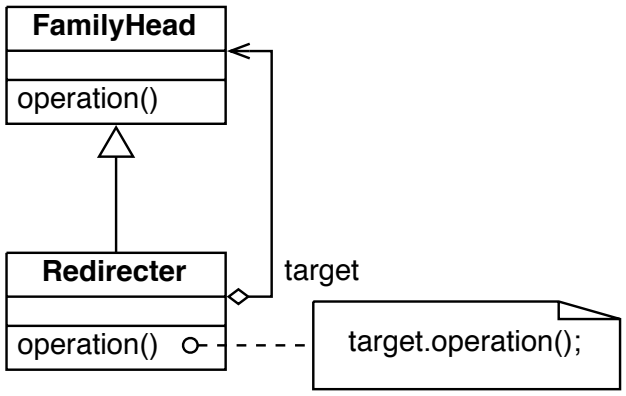

Figure 5: RedirectInFamily class structure

'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:

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

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

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

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

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

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

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...

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

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

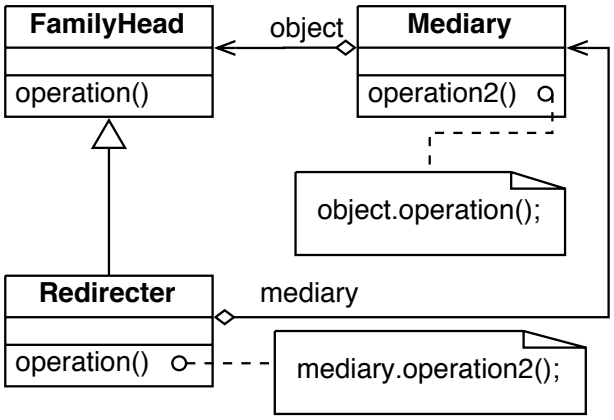

Figure 6: RedirectInFamily Isotope

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

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

5.4 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 6, 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 [16].

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.

$$Redirecter <: FamilyHead \quad (12)$$

$$r : Redirecter \quad (13)$$

$$fh : FamilyHead \quad (14)$$

$$r.mediary = m \quad (15)$$

$$m.object = fh \quad (16)$$

$$r.operation <_{\mu-} r.mediary.operation2 \quad (17)$$

$$m.operation2 <_{\mu-} m.object.operation \quad (18)$$

$$r.operation <_{\phi} r.mediary \quad (19)$$

$$m.operation <_{\phi} m.object \quad (20)$$

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

$$\frac{\begin{array}{c} r.operation <_{\mu-} r.mediary.operation2, \\ r.mediary = m \end{array}}{r.operation <_{\mu-} m.operation2} \quad (21)$$

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

$$\frac{\begin{array}{c} r.operation <_{\mu-} m.operation2, \\ m.operation2 <_{\mu-} m.object.operation \end{array}}{r.operation <_{\mu+} m.object.operation} \quad (22)$$

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

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

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

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 **RedirectInFamily**, as per Equation 11. This alternate structure is an example of an *isotope* of the **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 [8], 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.

6. RECONSTRUCT KNOWN PATTERNS

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

6.1 AbstractInterface

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:

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

6.2 Objectifier

Objectifier is simply a class structure applying the Inheritance EDP to an instance of **AbstractInterface** pattern, where the **AbstractInterface** applies to all methods in a class. This is equivalent to what Woolf calls an Abstract Class pattern. Referring back to Figure 2 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:

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

6.3 Object Recursion

We briefly described Object Recursion in section 3.2, and gave its class structure in Figure 3. We now show that this is a melding of the **Objectifier** and **RedirectInFamily** patterns, as illustrated in Figure 7. The annotations indicate which roles of which patterns the various components of **ObjectRecursion** play. A formal EDP representation is given in Equation 27.

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

6.4 ExtendMethod

The **ExtendMethod** EDP is used to extend, not replace, the functionality of an existing method in a superclass. Figure 8 shows the structure of the pattern, illustrating the use of **super**, formalized in Equation 28.

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

6.5 Decorator

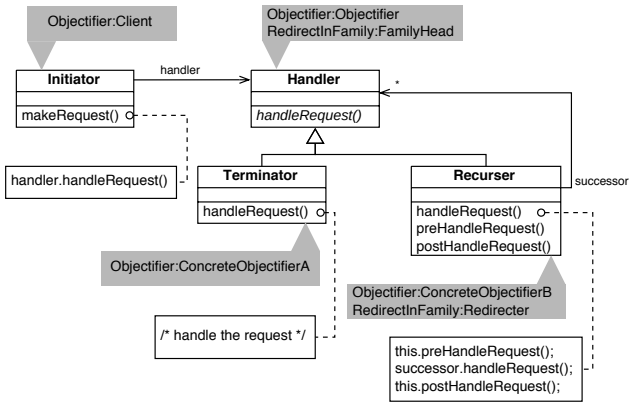

Figure 7: Object Recursion, annotated to show roles

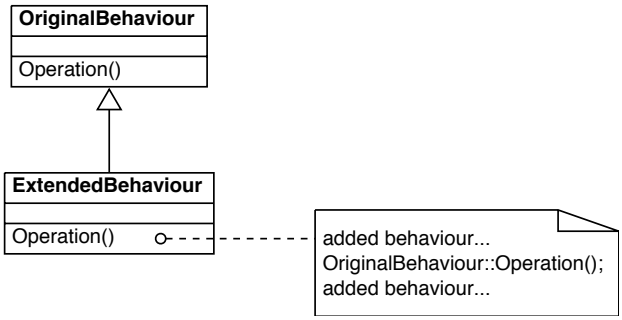

Figure 8: ExtendMethod

Now we can produce a pattern directly from the GoF text, the **Decorator** pattern. Figure 9 is the standard class diagram for **Decorator** annotated to show how the EDP **ExtendMethod** and **ObjectRecursion** pattern 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 **ObjectRecursion**.

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. Furthermore, by building this approach on an existing denotational semantics for object oriented programming we continue to be able to process the same system at an extremely low level. Cohesion and coupling analysis[6, 19, 21, 22, 36], slice metrics production[23, 32], and other traditional code analysis tech-

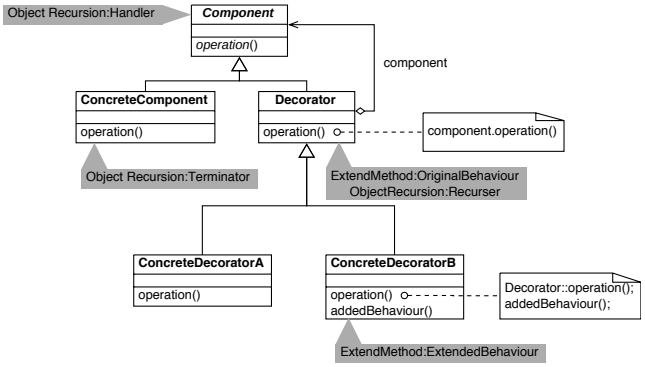

Figure 9: Decorator annotated to show EDP roles

niques[10, 12, 33] are still completely possible within the ρ-calculus. We have provided the link between patterns, as conceptual entity descriptions, to the formal semantics required and used by compilers and other traditional tools, without losing the flexibility of implementation required by the patterns. We do not, however, see an explicit need always to resort to the full ρ-calculus for all analyses. One of the key contributions of this system is that the practitioner can *choose* on which level to operate, and perform the analyses and tasks which are suitable without losing the flexibility of integrating other layers of analysis at a later date. Most importantly, we have created a system which enables the analysis of existing source code to extract the architecture, expressed as design patterns.

7. A PRACTICAL EXAMPLE

So how does this help our intrepid engineers at Widget, Inc? Let us start with their source code, salient features of which are shown as pseudo-C++ in Figure 10, 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 45.

We can quickly see that our **AbstractInterface** rule is fulfilled for class $File$, method $op1$ by Equation 30. Furthermore, $File$ and $FilePile$ fulfill the requirements of the **Objectifier** pattern, assuming, as we will here assert, that

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

```
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 10: Killer Widget pseudo-code

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

Figure 11: Killer Widget as ρ-calculus

the remainder of $File$'s methods are likewise abstract.

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

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

Finding an instance of **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 **RedirectInFamily**:

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

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.

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

ExtendMethod is a simple derivation as well:

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

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

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

Similarly, we can uncover the latent **Composite** pattern in the architecture. Both GoF pattern implementations are annotated in Figure 12. The intermediate patterns have been

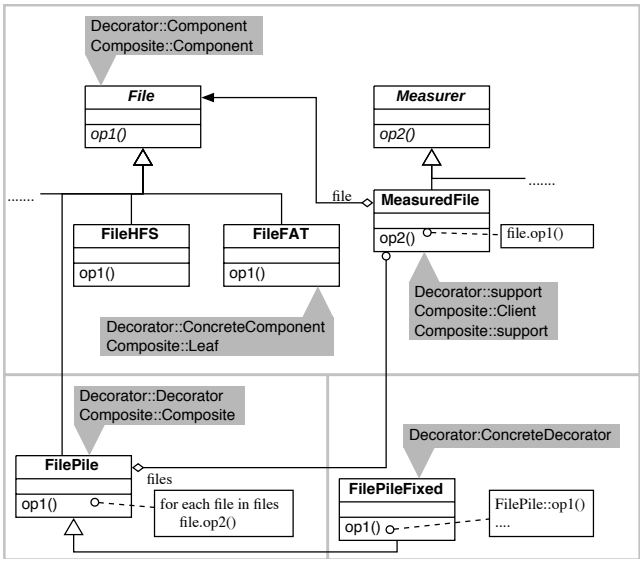

Figure 12: Discovered pattern roles

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.

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.

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 systems such as OTTER, to produce explicit illustrations of pattern implementation.

8. AUTOMATION OF APPROACH - SPQR

The above mechanism for design pattern extraction, as with many formal methods, is cumbersome for manual use. An engineer would find such a task about as useful as manual compilation of source code to assembler. Analogously, this approach was designed to be incorporated into an automatic toolset that performs the analysis directly on source code, much as a compiler.

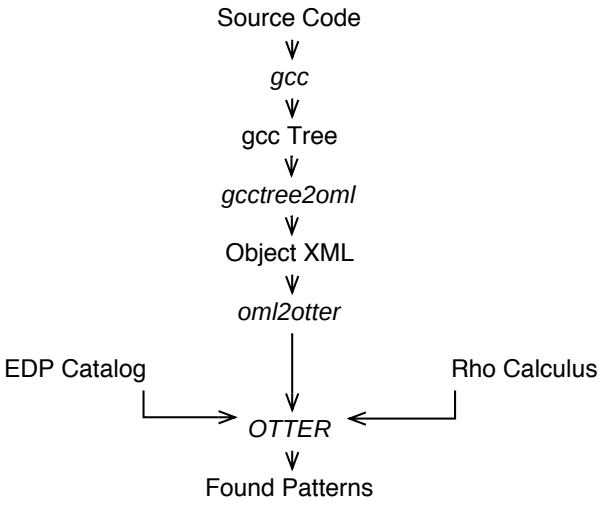

Figure 13: SPQR Outline

This toolset, the System for Pattern Query and Recognition, is comprised of several components, shown in Figure 13. 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 representation of the object structures. We chose an intermediary step so that various back ends could be used to input source semantics to SPQR. A second tool, *oml2rho* then reads a file of this format and produces an 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, and a report is compiled that can be used for further analysis, such as the production of UML diagrams.

The Killer Widget example has been successfully analyzed and the salient **Decorator** pattern was found using the OTTER ATP and the generated ρ-calculus rule set shown in Figure 14. Note the distinct similarity to our initial set of facts shown in Equations 30 through 45. The inputs to OTTER include the set of facts of the system under consideration, the necessary elements of ρ-calculus encoded as OTTER rules, and the design patterns of interest, including the EDPs, similarly encoded. For example, the **RedirectInFamily** pattern is shown encoded in Figure 15, illustrating the correspondence to our ρ-calculus definition in Equation 11. Our preliminary results indicate that scaling to larger systems in production code will be effective.

It is important to note that this is an application of our assertion that, by providing a rich formal foundation for this analysis, a practitioner can choose the level of detail in which to work. The *gcctree2oml* tool could, if we chose, do a raw conversion of the abstract syntax tree to pure ς-calculus, with all instances of ρ-calculus reliance operators being derived from basic principles. We find that it is, however, in many cases much faster to perform what static analysis we

```
%%% 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 14: Killer Widget as OTTER Input

```
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 15: RedirectInFamily as OTTER input

can to directly generate ρ-calculus. This results in a proof system that is not only simpler and more efficient, but easier for a human to read and follow if desired. If, however, a more formal and first principles approach were desired, it could easily be accomodated.

9. 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, education of best practices in object oriented programming, and guiding the selection of future language design.

9.1 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 construct from simpler forms.

9.1.1 Delegation

A recent and highly touted example of such a language construct is the *delegate* feature found in C#[27]. 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.

9.1.2 ExtendMethod

Most languages have some support for this EDP, through the use of either static dispatch, as in C++, or an explicit keyword, such as Java and Smalltalk's **super**. Others, such as BETA[25], 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.

9.1.3 AbstractInterface

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 programming, and more importantly, OO design. Future languages may benefit from a more explicit construct.

9.2 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, increasing the distribution of the information. This parallels the suggestions put forth by Goldberg in 1994[18]. We are putting this into practice, incorporating the EDPs into the 2003-04 curriculum for software engineering at the University of North Carolina at Chapel Hill, to investigate the effectiveness of such an approach.

9.3 Semi-automated support for refactoring

Refactoring is not likely to ever be, in our opinion, a fully automatable process. At some point the human engineer must make decisions about the architecture in question, and guide the transformation of code from one design to another. Several key pieces, however, may benefit from the work outlined in this paper. Our isotope example in Section 5.4 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[16, 31, 28]. Ó Cinnéide's minitransformations 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[29, 30]. Furthermore, we believe the fragments-based systems such as LePuS can now be integrated back into the larger domain of denotational semantics.

9.4 Comprehension of code

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*[37], or KT[7], that operate on a higher level of abstraction than just "class, object and method interactions"[37]. 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.

10. 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.

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.

11. FORMAL FOUNDATIONS

We present in this section more complete discussions of the generation of our elemental design patterns, and a formal presentation of the ρ-calculus.

11.1 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[17]. 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[41], 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[11].

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, extremely similar to Ó Cinnéide's Encapsulate Construction minipattern[29]. 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.

11.1.1 Method calls

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

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

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. The result, shown in Table 2, is a list of eight method calling styles. Note that four of these are simply variations on whether the called method is abstract or not. By identifying this as an instance of the AbstractInterface component from above, we can simplify this list further to our final collection of the six primary method invocation styles in the GoF text, shown in Table 3. 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 [4], it becomes obvious that this is an important concept we dub *similarity*.

11.2 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]

11.2.1 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 3:

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]

[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.

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

Table 2: Reduced method calling styles in Gang of Four patterns

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

Table 3: Final method calling styles in Gang of Four patterns

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.

11.2.2 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 in turn.

1. Self ($a = b$)

   (a) Self ($a = this$)

i. Same ($F = G$) ..... Recursion
ii. Different ($F \neq G$) ..... Conglomeration

(b) Super ($a = super$)

i. Same ($F = G$) ..... ExtendMethod
ii. Different ($F \neq G$) ..... RevertMethod

2. Other ($a \neq b$)

(a) Unrelated

i. Same ($F = G$) ..... Redirect
ii. Different ($F \neq G$) ..... Delegate

(b) Same ($A = B$)

i. Same ($F = G$) ..... RedirectedRecursion
ii. Different ($F \neq G$) DelegatedConglomeration

(c) Parent ($A <: B$)

i. Same ($F = G$) ..... RedirectInFamily
ii. Different ($F \neq G$) ..... DelegateInFamily

(d) Sibling ($A <: C, B <: C, A \not<: B$)

i. Same ($F = G$) ..... RedirectInLimitedFamily
ii. Different ($F \neq G$) . 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.

11.3 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 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, such as in Emerald[20], or cloning mechanisms in prototype based languages such as Cecil[9] or NewtonScript[3].

Inheritance Enforces a structural relationship for subtyping.

11.4 Rho Fragment

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.

11.4.1 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

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

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: $<_\mu$, $<_\phi$, $<_\gamma$

11.4.2 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 another object $\mathcal{O}'$, calling method μ', then we have a method reliance between the two, indicated by the μ form reliance operator ($<_\mu$):

$$\frac{\mathcal{O}.\mu \equiv [\mu = \varsigma()\mathcal{O}'.\mu']}{\mathcal{O}.\mu <_\mu \mathcal{O}'.\mu'} \tag{51}$$

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:

$$\frac{\mathcal{O}.\mu \equiv [\mu = \varsigma()\mathcal{O}']}{\mathcal{O}.\mu <_\phi \mathcal{O}'} \tag{52}$$

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:

$$\frac{\mathcal{O} : \tau, \tau = [\mathcal{O}' : \tau']}{\mathcal{O} <_\phi \mathcal{O}'} \tag{53}$$

11.4.3 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:

$$\frac{\mathcal{O}.\mu <_\mu \mathcal{O}'.\mu', \mu = \mu'}{\mathcal{O}.\mu <_{\mu+} \mathcal{O}'.\mu'} \tag{54}$$

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:

$$\frac{\mathcal{O}.\mu <_\mu \mathcal{O}'.\mu', \mu \neq \mu'}{\mathcal{O}.\mu <_{\mu-} \mathcal{O}'.\mu'} \tag{55}$$

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:

$$\frac{A <_\phi \mathcal{O}', \mathcal{O} : \tau, \mathcal{O}' : \tau', \tau = \tau'}{A <_{\phi+} \mathcal{O}'} \tag{56}$$

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

$$\frac{A <_\phi \mathcal{O}', \mathcal{O} : \tau, \mathcal{O}' : \tau', \tau \neq \tau'}{A <_{\phi-} \mathcal{O}'} \tag{57}$$

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.

11.4.4 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.

$$\frac{A <_x A', A' <_{x'} A''}{A <_x A''} \quad iff \quad x = x' \tag{58}$$

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.

$$\frac{A <_x A', A' <_{x'} A''}{A <_\gamma A''} \quad iff \quad x \neq x' \tag{59}$$

11.4.5 Generalizations

These are generalizations of relops, the opposite of the specialization rules earlier. Each of them generalizes out some

piece of information of 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.

$$\frac{A <_{x \overset{\pm}{\circ}} \mathcal{O}'.\mu'}{A <_{x \overset{\pm}{\circ}} \mathcal{O}'} \tag{60}$$

$$\frac{\mathcal{O}.\mu <_{x \overset{\pm}{\circ}} A'}{\mathcal{O} <_{x \overset{\pm}{\circ}} A'} \tag{61}$$

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).

$$\frac{A <_x A'}{A <_\gamma A'} \tag{62}$$

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

$$(x = \mu, \phi) \qquad \frac{A <_{x\pm} A'}{A <_x A'} \tag{63}$$

11.5 Objects vs. Types

It would seem natural to use types instead of actual objects in the reliance operators, but there is a fundamental incompatibility between $\Delta\rho$ and subsumption of types which makes this approach difficult for most practical applications of our analysis technique. Instead, since we are directly analyzing source code, we have the opportunity to use more information regarding object instances than would be evident in class diagramming notations such as UML.

11.5.1 LHS Typing

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

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

On the face of it, this seems reasonable. Assuredly the definition of the object type τ includes the same invocation call to $\mathcal{O}'.\mu'$. If, however, we include type subsumption:

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

Now we have a problem, since $\tau'.\mu$ may in fact replace the method body defined in τ with one that does **not** include the call to $\mathcal{O}'.\mu'$. We have no way of asserting that the above is true.

A similar problem appears with $<_\phi$ created with Method Field Reliance:

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

Here again this seems self-consistent. But once we find that after adding type subsumption:

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

we arrive at the same problem. The method body of $\tau'.\mu$ may eliminate the reference to the object, if for instance, it was a local variable.

Any occurrence of μ on the left hand side is incompatible with subsumption. If we look at the Object Field Reliance created instances of $<_\phi$, we find no problem:

$$\frac{\mathcal{O} <_\phi \mathcal{O}', \mathcal{O} : \tau}{\tau <_\phi \mathcal{O}'}$$

$$\frac{\tau <_\phi \mathcal{O}', \tau' <: \tau}{\tau' <_\phi \mathcal{O}'}$$

Since method/field extraction is prohibited in the ς-calculus, we can be assured that any subtypes of τ will still include the reference we found in τ. A subtype cannot remove references defined in a supertype. It may hide them from external access, ignore them, or otherwise make them less than useful, but it cannot outright delete them, so our final inference holds.

LHS Generalization combined with left hand side type replacement is fundamentally unsound when combined with the type subsumption ($\Delta_{<:}$) fragment from ς-calculus.

11.5.2 RHS Typing

RHS Generalization is not type-unsafe under subsumption, but instead leads to a natural expression of polymorphism.

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

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

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

What we find is that this is a *possible* reliance; it is an example of polymorphism, and it *may* be an inferable fact, but it also may never occur. It is interesting that we have come to a state that is not directly contradictable, yet not directly verifiable.

Right hand side type replacement is sound under subsumption and appears to provide a basis for polymorphic analysis, but in our experience this can be handled more easily through other means.

12. ACKNOWLEDGMENTS

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

13. REFERENCES

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

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

[3] Apple. *The NewtonScript programming language*. Apple Computer, Inc., 1993.

[4] Kent Beck. *Smalltalk Best Practice Patterns*. Prentice Hall, 1997.

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

[6] L.C. Briand and J.W. Daly. A unified framework for cohesion measurement in object-oriented systems. In *Proc. of the Fourth Conf. on METRICS'97*, pages 43–53, November 1997.

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

[8] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented System Architecture: A System of Patterns*. John Wiley & Sons, 1996.

[9] Craig Chambers. The cecil language: Specification and rationale. Technical Report TR-93-03-05, University of Washington, 1993.

[10] Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994. cohesion/LCOM.

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

[12] Serge Demeyer, Stéphane Ducasse, and Oscar 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.

[13] Amnon H. Eden. *Precise Specification of Design Patterns and Tool Support in their Application*. PhD thesis, Tel Aviv University, Tel Aviv, Israel, 1999. Dissertation Draft.

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

[15] 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.

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

[17] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison Wesley, 1995.

[18] Adele 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.

[19] Martin Hitz and Behzad Montazeri. Measuring coupling and cohesion in object-oriented systems. In *Proceedings of ISACC'95*, pages 10–21, Insitut für Angewandte Informatik und Informationssysteme, Uni versity of Vienna, Rathaustraße 1914, A-1010 Vienna, Austria, 1995.

[20] Eric Jul, Rajendra K. Raj, Ewan D. Tempero, Henry M. Levy, Andrew P. Black, and Norman M. Hutchinson. Emerald: A general-purpose programming language. *Software Practice and Experience*, 21(1):91–118, January 1991.

[21] Byung-Kyoo Kang and James M. Bieman. Design-level cohesion measures: Derivation, comparison, and applications. In *Proc. 20th Intl. Computer Software and Applications Conf. (COMPSAC'96)*, pages 92–97, August 1996.

[22] Byung-Kyoo Kang and James M. Bieman. Using design cohesion to visualize, quantify and restructure software. In *Eighth Int'l Conf. Software Eng. and Knowledge Eng., SEKE '96*, June 1996.

[23] Sakari Karstu. An examination of the behavior of slice-based cohesion measures. Master's thesis, Minnesota Technological University, 2999.

[24] Bent Bruun 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.

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

[26] Scott Meyers. *Effective C++*. Addison-Wesley, 1992.

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

[28] Ivan 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.

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

[30] Mel Ó Cinnéide and Paddy 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.

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

[32] Linda M. Ott. Using slice profiles and metrics during software maintenance. In *Proceedings of the 10th Annual Software Reliability Symposium, Denver, June 25-26, 1992*, June 1992.

[33] Linda M. Ott and Jefferey J. Thuss. Slice based metrics for estimating cohesion. In *Proceedings of the IEEE-CS International Software Metrics Symposium, Baltimore, May 21-22 1993*, May 1993.

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

[35] Dirk 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.

[36] M. H. Samadzadeh and S. J. Khan. Stability, coupling and cohesion of object-oriented software systems. In *Proc. 22nd Ann. ACM Computer Science Conf. on Scaling Up*, pages 312–319, March 1994. Mar 8-10, 1994.

[37] Mohlalefi Sefika, Aamod Sane, and Roy 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.

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

[39] Jason McC. Smith. An elemental design pattern catalog. Technical Report TR-02-040, Univ. of North Carolina, 2002.

[40] Jason McC. Smith and David Stotts. Elemental design patterns: A link between architecture and object semantics. Technical Report TR-02-011, Univ. of North Carolina, 2002.

[41] Bobby Woolf. The abstract class pattern. In Neil Harrison, Brian Foote, and Hans Rohnert, editors, *Pattern Languages of Program Design 4*. Addison-Wesley, 1998.

[42] Bobby Woolf. The object recursion pattern. In Neil Harrison, Brian Foote, and Hans Rohnert, editors, *Pattern Languages of Program Design 4*. Addison-Wesley, 1998.

[43] Walter Zimmer. Relationships between design patterns. In James O. Coplien and Douglas C. Schmidt, editors, *Pattern Languages of Program Design*, pages 345–364. Addison-Wesley, 1995.