Elemental Design Patterns: A Formal Semantics for Composition of OO Software Architecture

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

Abstract

Design patterns are an important concept in the field of software engineering, providing a language and application independent method for expressing and conveying lessons learned by experienced designers. There is a large gap, however, between the aesthetic and elegance of the patterns as intended and the reality of working with an ultimately mathematically expressible system such as code. In this paper we describe a step towards meaningful formal analysis of code within the language of patterns, and discuss potential uses. The major contributions include: a compendium of Elemental Design Patterns (EDPs), a layer of seemingly simplistic relationships between objects that, on closer inspection, provide a critical link between the world of formal analysis and the realm of pattern design and implementation without reducing the patterns to merely syntactic constructs; an extension to the ς-calculus, termed ρ-calculus, a formal notation for expressing relationships between the elements of object oriented languages, and its use in expressing the EDPs directly. We discuss their use in composition and decomposition of existing patterns, identification of pattern use in existing code to aid comprehension, and future research directions, such as support for refactoring of designs, interaction with traditional code analysis systems, and the education of students of software architecture.

1. Problem Description

Programming has historically been an exercise in the creation of hierarchical abstractions to manage complexity. As programming techniques have progressed in the field, language designers have continued to push the envelope of producing explicit constructs for those conceptual lessons learned in the previous generation of languages, and software architects have continued to build ever more complex and powerful abstractions. At the same time that these abstractions have established methods for producing well designed systems, they have created problems for the pure theorist and the accomplished practitioner alike, resisting attempts at the formalizations that are necessary for many critical analyses of system architectures.

One of the current successful abstractions in widespread use is the design pattern, an approach that builds upon the nature of object oriented languages to describe 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 the practitioner with increasingly powerful constructs with which to work.

There is a class of problems, however, which is even more common, yet design patterns have in general ignored. These problems are usually considered too obvious to provide a description for, because they are in every good programmer's toolkit. The solutions to this class of problems we term *Elemental Design Patterns* (EDPs), and are the base concepts on which the more complex design patterns are built. Since they comprise the constructs which are used repeatedly within the more common patterns to solve the same problems, such as abstraction of interface and delegation of implementation, they exhibit some interesting properties for partially bridging the gap between the source code of everyday practice and the higher level abstractions of the larger patterns. The higher-level patterns are thus described in the language of elemental patterns, which fills an apparent missing link in the abstraction chain.

Design patterns also present an interesting set of problems for the theorist due to their dual nature[2], with both formally expressible and informally amorphous halves. 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 re-

ducible to a mathematically formal notation. The core concepts themselves, however, have to date evaded such formalization. We show here that such a formalization is in most cases possible, and in addition that it can meet certain critieria we deem essential.

We assert that such a formal solution should be implementation language independent, much as the design patterns are, if it is to truly capture universal concepts of programming methodology. We further assert that a formal denotation for pattern concepts should be a larger part of the formal semantics literature. Patterns are built on the theory and concepts of object-oriented programming, as surely as object-oriented approaches are built on procedural theory. A formal representation of patterns should reflect this, allowing for more detailed analysis of the code body if it is desired, or allowing an analysis at a very high level of abstraction.

We begin with the sigma (ς) calculus[1], an object-oriented analogue to lambda calculus. To this we add the ability to encode relationships between the constructs of ς-calculus though the use of *reliance operators*. We show how the combined calculus, the *rho (ρ) calculus*, can be used to express our EDPs directly and precisely, which in turn are used to express the more common class of design patterns. In essence, we build a well defined and formal chain from a basic denotational semantics for object based languages in general to the language of design patterns, providing a clear path between them, with well formed transformations and the opportunity for various interesting analyses of patterns and their applications within systems.

This ρ-calculus can be used as a framework to analyze source code, both legacy and newly implemented, and system architectures, for instances of design patterns, through the discovery of EDPs and their relationships. This discovery process is language, tool, and coding-style-independent, and relies only on a syntactic parsing of the source code, as any compiler can do. By providing engineers with such a tool of discovery and analysis, we aid in the comprehension of a system by bringing to light hidden, and perhaps unintended, uses of design patterns. The revelation of higher level abstractions such as design patterns gives the engineer important clues as to the operation and structure of the system.

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[8, 18, 26, 33]. The few researchers who have attempted to provide a truly formal basis for patterns have most commonly done so from a desire to perform refactoring of existing code, while others have attempted

the more pragmatic approach of identifying core components of existing patterns in use. Additionally, there has been an ongoing philosophical interest in the very nature of coding abstractions such as patterns, and their relationships.

2.1. Refactoring Approaches

Refactoring[12] has been a frequent target of formalization techniques, with fairly good success to date[9, 20, 22]. The primary motivation is to facilitate tool support for, and validation of, transformation of code from one form to another while preserving behaviour. This is an important step in the maintenance and alteration of existing systems, and patterns are seen as the logical next abstraction upon which they should operate.

Such techniques include fragments, as developed by Florijm, Meijers, and van Winsen[11], Eden's work on LePuS[10], Ó Cinnéide's work in transformation and refactoring of patterns in code[21] through the application of minipatterns.

2.2. Structural Analyses

An analysis of the 'Gang of Four' (GoF) patterns from the Design Pattens text [13] reveals many shared structural and behavioural elements, such as the similarities between Composite and Visitor, for instance[13]. The relationships between patterns, such as inclusion or similarity, have been investigated by various practitioners, and a number of meaningful examples of underlying structures have been described. [4, 8, 26, 31, 32, 33]

The Objectifier pattern[33] is one such example of a core piece of structure and behaviour that is 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 the Objectifier as a 'basic pattern' in the construction of several other GoF patterns, such as Builder, Observer, Bridge, Strategy, State, Command and Iterator. It is a simple yet elegantly powerful structural concept that is used repeatedly in other patterns.

Woolf takes this pattern one step further, adding a behavioural component, and naming it Object Recursion[32]. The class diagram in Figure 2 is extremely similar to Objectify, with an important difference, namely the behaviour in the leaf subclasses of *Handler*. Exclusive of this method behaviour, however, it looks to be an application of Objectify
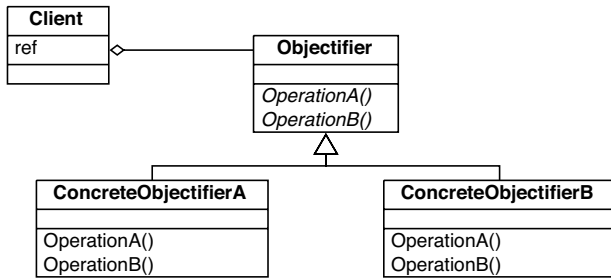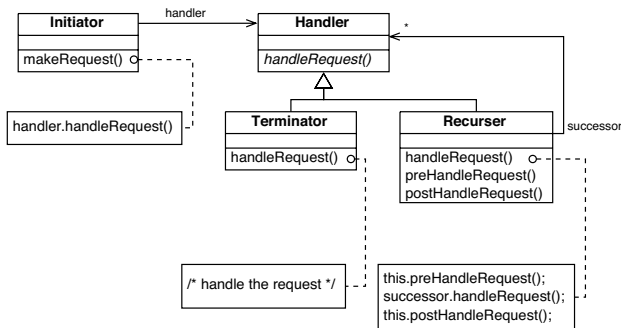
Figure 1. Objectifier class structure

Figure 2. Object Recursion class structure

in a more specific use. Note that Woolf compares Object Recursion to the relevant GoF patterns and deduces that: Iterator, Composite and Decorator can, in many instances, be seen as containing an instance of Object Recursion; Chain of Responsibility and Interpreter do contain Object Recursion as a primary component.

2.3. Conceptual Relationships

Taken together, the above instances of analyzed pattern findings seem to comprise two parts of a larger chain: Object Recursion contains an instance of Objectify, and both in turn are used by larger patterns. 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, 8, 19], and Pree's metapatterns[25] all support this viewpoint. It will become evident that these relationships between *concepts* are a core piece of allowing great flexibility to the practitioner implementing patterns in design, through constructs we term *isotopes*. A full discussion of isotopes is beyond the scope of this paper, but can be found in [29].

This paper will answer the following questions: how far can we take this decomposition and recomposition of patterns in a meaningful way? Is it possible to continue to identify useful solutions to common problems *within* the

patterns literature? Is it further possible to identify relationships between these components? How finely can patterns be deconstructed? Is to do so useful, or merely a theoretical exercise?

3. The EDP Catalog

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[13]. 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. Of these eight, five involved method invocation, leading us to investigate method interactions from a more abstract approach.

The method calls involved in the GoF patterns were classifiable 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

The remaining component abstractions from the GOF patterns consisted of abstractions related to object creation, abstract interface of methods, and object retrieval semantics. When it is understood that these comprise, respectively, object instantiation, method slot fulfillment, and object referencing, it becomes clear that when combined with our method call invocation styles and type subsumption through inheritance, all of object-oriented programming becomes available for analysis using this technique.

We present in this paper just a listing of the identified Elemental Design Patterns. A more complete discussion of their derivation can be found at [28]. 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.

Object Element EDPs

CreateObject	AbstractInterface
Retrieve	

Type Relation EDPs

Inheritance

Method Invocation EDPs

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

At first glance, these EDPs seem highly 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. Patterns are, to be precise, descriptions of relationships between objects, according to Alexander[2], and method invocations and typing are the process through which objects interact. We believe that we have captured the elemental components of object oriented languages, and the salient relationships used in the vast majority of software engineering. If patterns are the frameworks on which to create large understandable systems, these are the nuts and bolts that comprise the frameworks.

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

4. Formalization

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

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

4.1. Sigma Calculus

An analysis of desired traits for an intermediate formalization language includes that it be mathematically sound, consist of simple reduction rules, have enough expressive power to directly encode object-oriented concepts, and have the ability to flexibly encode relationships between code constructs. Given these constraints, there are few options. The most obvious possible solution is lambda calculus or one of its variants [30], but lambda calculus cannot directly encode object-oriented constructs. Various exten-

sions which would enable lambda calculus to do so have been proposed, but they invariably produce a highly cumbersome and complex rule set in an attempt to bypass apparently fundamental problems with expressing typed objects with a typed functional calculus [1]. One final candidate, sigma calculus, meets this requirement easily.

ς-calculus[1] is not an extension of λ-calculus. Attempts to produce such a hybrid have been made, but none has been particularly successful. A prime motivation for working to graft OO technologies onto λ-calculus is a desire to leverage off of the extremely large body of well done literature in that area. By starting anew, Abadi and Cardelli at first glance seem to have disposed of that body of work. On the contrary, they correctly recognize that the entirety of λ-calculus can be subsumed within the method calls of OOP. They even provide a mapping from λ-calculus to ς-calculus, resulting in "a simple and direct reduction semantics, instead of an indirect semantics involving both λ-abstraction and application." [1, p. 66]

While ς-calculus is a rich and important work in formalization of object oriented languages, it does not meet our needs for formalization of design patterns. Not only is it extremely unwieldly, but it also suffers from a complete rigidity of form, and does not offer any room for interpretation of the implementation description, or any necessary fungibility that may be required for a specific application. This lack of adaptiveness means that there would be an explosion of definitions for just 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.

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) are direct, quantifiable expressions of whether one element, (an object, method, or field) in any way relies or depends on the existance of another for its own definition or execution, and to what extent it does so.

This approach provides more detail than the formal description provided by UML, for instance. The calculus comprised of ς-calculus and these reliance operators, or *rho (ρ)calculus*, maps nicely to the concepts of IsA, HasA, HoldsA, UsesA, and so on that exist within UML, indicating that a simple mapping between the two should exist. Unlike UML, however, reliance operators encode entire

paths of reliances in a concise notation. All the reliances and relationships in the UML graphing system are encoded within the element that is under scrutiny, reducing the need for extended, and generally recursive, analysis for each element when needed.

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

For the purposes of this paper we need only two reliance operators: First, $\ll_m$, indicating a method invocation call reliance. Given the expression $a.f \ll_m b.g$, it indicates that within the body of method f in object a, a call is made to method g of object b. Secondly, $<:$, the traditional inheritance (or more properly subsumption of type,) showing a type reliance.

4.3. Example: RedirectInFamily

Consider the class diagram for the structure of the Method Invocation EDP *RedirectInFamily*, in Figure 3. Taken literally, it specifies that a class wishes to invoke a similar method (where, again, similarity is evaluated based on the function types of the methods) to the one currently being executed, and it wishes to do so on an object of its parent class' type. This sort of open-ended structural recursion is a part of many patterns.
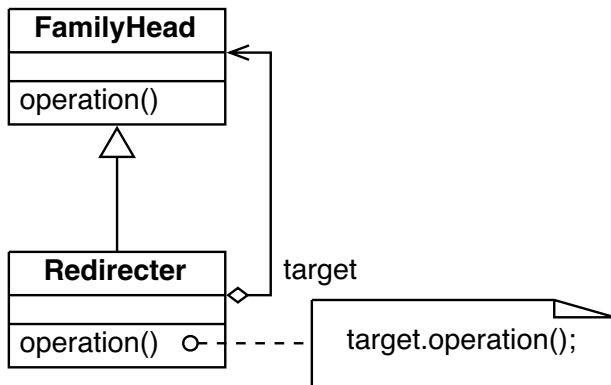

Figure 3. RedirectInFamily class structure

If we take an example Participants specification of RedirectInFamily, described as a pattern, we can state:

- 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:

$$\begin{array}{rcll} FamilyHead & \equiv & [operation : A] & (1) \\ Redirecter & <: & FamilyHead & (2) \\ Redirecter & \equiv & [target : FamilyHead, & \\ & & operation : A = \varsigma(x_i)\{target.operation\}] & (3) \\ r & : & Redirecter & (4) \\ fh & : & FamilyHead & (5) \\ r.target & = & fh & (6) \end{array}$$

This is a concrete implementation of the RedirectInFamily structure, but fails to capture the reliance of Redirecter.operation on FamilyHead.operation's behaviour. So, we introduce our reliance operator $\ll_m$:

$$r.operation \ll_m r.target.operation \qquad (7)$$

We can reduce one level of indirection...

$$\frac{r.target = fh, r.operation \ll_m r.target.operation}{r.operation \ll_m fh.operation} \qquad (8)$$

...and now we can produce a necessary and sufficient set of clauses at this point to represent RedirectInFamily:

$$\frac{\begin{array}{c} Redirecter <: FamilyHead, \\ r : Redirecter, \\ fh : FamilyHead, \\ r.operation \ll_m fh.operation, \\ r.operation : A, \\ fh.operation : A \end{array}}{RedirectInFamily(r, fh, operation)} \qquad (9)$$

5. Reconstruct known patterns

We can now adequately 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 the GOF pattern Decorator, building and using two intermediate patterns from the literature along the way.

5.1. AbstractInterface

AbstractInterface is, simply put, ensuring that the method in a base class is truly abstract, forcing subclasses to override and provide their own implementations. The exceedingly simple class diagram for this is given in Figure 4. The ρ-calculus definition can be given by simply using the **trait** construct of ς-calculus:

$$\frac{A \equiv [new : [l_i : A \to B_i{}^i \in 1...n], operation : A \to B]}{AbstractInterface(A, operation)} \qquad (10)$$
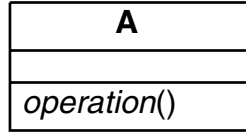
Figure 4. AbstractInterface

5.2. Objectifier

It should be obvious by now that 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 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:

$$
\frac{
\begin{array}{c}
Objectifier : [l_i : B_i{}^{i \in 1 \dots n}], \\
AbstractInterface(Objectifier, l_i{}^{i \in 1 \dots n}), \\
ConcreteObjectifier_j <: Objectifer^{j \in 1 \dots m}, \\
Client : [obj : Objectifier]
\end{array}
}{
Objectifier(Objectifier, ConcreteObjectifier_j{}^{j \in 1 \dots m}, Client)
}
$$
(11)

5.3. Object Recursion

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 Objectifier and RedirectInFamily patterns, as illustrated in Figure 5. The annotations indicate which roles of which patterns the various components of Object Recursion play. A formal EDP representation is:

$$
\frac{
\begin{array}{c}
Objectifier(Handler, Recurser_i{}^{i \in 1 \dots m}, Initiator), \\
Objectifier(Handler, Terminator_j{}^{j \in 1 \dots n}, Initiator), \\
Initiator \ll_m obj.handleRequest, \\
obj : Handler, \\
RedirectInFamily(Recurser, Handler, handleRequest), \\
!RedirectInFamily(Terminator, Handler, handleRequest)
\end{array}
}{
\begin{array}{c}
ObjectRecursion(Handler, Recurser_i{}^{i \in 1 \dots m}, \\
Terminator_j{}^{j \in 1 \dots n}, Initiator)
\end{array}
}
$$
(12)

5.4. ExtendMethod

The ExtendMethod EDP is used to extend, not replace, the functionality of an existing method in a superclass. Figure 6 shows the structure of such a pattern, illustrating the use of the abstraction **super**. A formal definition can be given by:
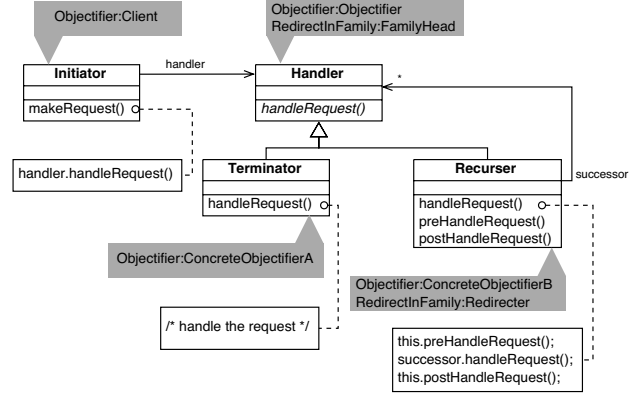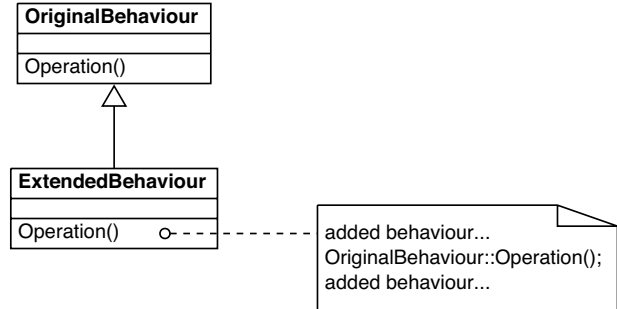

Figure 5. Object Recursion, annotated to show roles

Figure 6. ExtendMethod class structure

$$
\frac{
\begin{array}{c}
OriginalBehaviour : [l_i : B_i{}^{i \in 1 \dots m}, operation : B_{m+1}], \\
ExtendedBehaviour <: OriginalBehaviour, \\
eb : ExtendedBehaviour, \\
eb.operation \ll_m \textbf{super}.operation
\end{array}
}{
\begin{array}{c}
ExtendMethod(OriginalBehaviour, \\
ExtendedBehaviour, operation)
\end{array}
}
$$
(13)

5.5. Decorator

Now we can finally produce a pattern directly from the GoF text, the Decorator pattern. It is simple enough to be composed from the ground up, illustrating our technique of using fully formal methods entrenched in ς- and ρ-calculus coupled with the elemental design patterns catalog to create rich and conceptually true formal descriptions of useful design patterns. It is complex enough, however, to present a bit of a challenge, adding a bit of behavioural elegance to a primarily structural pattern.

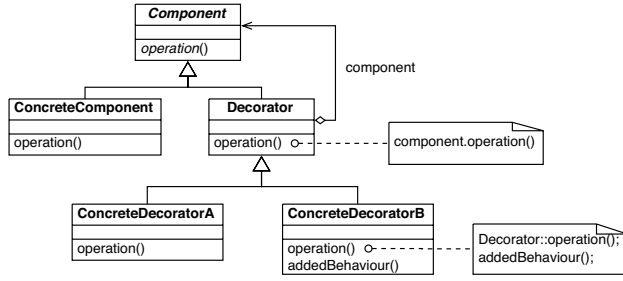Figure 7 is the standard class diagram for Decorator. Fig-
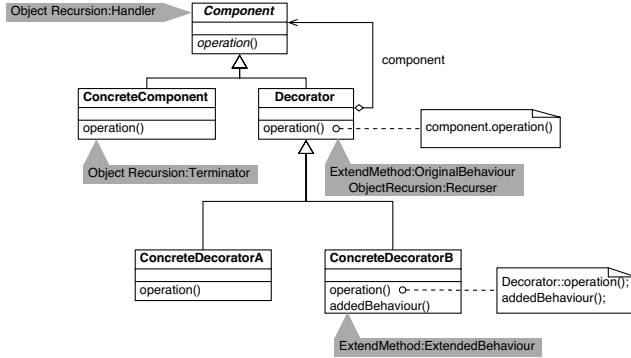
Figure 7. Decorator class structure

Figure 8. Decorator annotated to show EDP roles

ure 8 shows the same diagram, but annotated to show how the ExtendMethod and Object Recursion patterns interact. Again, we provide a formal definition:

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

(14)

The keyword **any** indicates that any object of any class may take this role, as long as it conforms to the definition of Object Recursion.

6. Discussion

Consider what we have just done - 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, and,

following our example of pattern composition, can be used as a building block for larger, even more intricate patterns that are *incrementally* comprehensible. At the same time, we believe we have retained the flexibility of implementation 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[5, 14, 15, 16, 27], slice metrics production[17, 23], and other traditional code analysis techniques[7, 9, 24] 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 to always resort to the full ρ-calculus for all analysis. One of the key contributions of this system is that the practitioner can *choose* on which level to operate, and perform the analyses and tasks which are suitable without losing the 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. Such analysis will be enabled by a toolset currently under production, the System for Pattern Query and Recognition (SPQR), which allows for design patterns to be found in existing C++ code.

The future research possibilities range across the full spectrum from formal analysis through human comprehension assistance, much as the ρ-calculus and elemental design patterns do, including the educational use of EDPs, support for refactoring, and comprehension of code during maintenance.

Education of software design best practices is problematic, since the student is being given solutions to problems they likely have never encountered. The solutions frequently seem overly cumbersome and are quickly forgotten. By providing the student with small, comprehensible steps in the EDP system, they can quickly see the incremental pieces that solve the components of a larger problem. This provides a context in which they can learn effectively. Similarly, refactoring is frequently approached as a series of small, well-formed steps in transforming code. We have provided a strong foundation on which to analyze existing code, and on which to provide assistance during the refactoring process.

Code comprehension, our initial driving problem, looks to benefit the most from our research. Revealing hidden clues of design and behaviour of a system to an engineer is expected to greatly increase the efficiency of practical work-

ing comprehension, allowing for effective maintenance of the system. The discovery process is fully automatable, using a system such as SPQR, and should provide a large benefit at a minimum of cost.

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

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

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

[9] 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.

[10] 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.

[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] M. Hitz and B. 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.

[15] B.-K. Kang and J. M. Bieman. Design-level cohesion measures: Derivation, comparison, and applications. In *Proc. 20th Intl. Computer Software and Applications Conf. (COMPSAC'96)*, pages 92–97, Aug. 1996.

[16] B.-K. Kang and J. 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.

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

[18] 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.

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

[20] 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.

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

[22] 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.

[23] L. 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.

[24] L. M. Ott and J. 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.

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

[26] 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.

[27] 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, Mar. 1994. Mar 8-10, 1994.

[28] 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.

[29] J. M. Smith and D. Stotts. Elemental design patterns: A logical inference system and theorem prover support for flexible discovery of design patterns. Technical Report TR-02-038, Univ. of North Carolina, 2002.

[30] R. Stansifer. *The Study of Programming Languages*. Prentice Hall, 1995.

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

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

[33] 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.

IEEE
COMPUTER
SOCIETY