UNIVERSITY OF CALIFORNIA, SAN DIEGO

A Framework for the Checking and Refactoring of Crosscutting Concepts

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Computer Science

by

Macneil Charles Shonle

2009

The dissertation of Macneil Charles Shonle is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

Co-Chair

Co-Chair

University of California, San Diego

2009

DEDICATION

With love to my parents, my wife Anna, and my son Ansel.

And in loving memory to

Steven John Metsker, Arthur W. Chou, and Joseph Amadee Goguen.

EPIGRAPH

Yes we can.
—Barack Obama

The secret of genius is to carry the spirit of the child into old age,
which means never losing your enthusiasm.
—Aldous Huxley

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

Thanks first go to my wife, Anna, because she gave me the idea for this work when I asked her what “deconstructive programming” would be like. She was also the most important catalyst in my transition into becoming a full time Ph.D. student.

I am also grateful that I had the opportunity to work with both Bill Griswold and Sorin Lerner. I can still remember the research meeting at Bill’s house when the ideas behind this dissertation really came together. The results were impressive enough for us to rope Sorin into joining the team. I look forward to working with you both again. And also thanks go to Karl Lieberherr, at Northeastern University, for helping me publish my first conference paper.

I’d like to thank Kevin Li for comparing notes as we both ventured into the job market together. Similarly, thanks to Cynthia Bailey Lee for sharing our experiences when we both taught our first classes during the Summer 2007 session. I did have some time for a social life, and one of the richest moments was the group dinners known as the Veggie Chow Brigade, organized by Rob Fargo and then Cynthia Taylor.

Thanks must also go to Bill’s UbiComp group and to Ranjit Jhala and Sorin’s Programming Languages group. Also, here are some particular individuals I’d like to thank, in autobiographical order: Darren Atkinson, John Stephenson, Patricia Shanahan, Dana Dahlstrom, Neil McCurdy, Charles Lucas, Jack Sampson, Roshni Malani, Lisa Cowan, Pat Rondon, Elizabeth Bales, Krista Davis, Michael Stepp, Jan Voung, Ravi Chugh, Zach Tatlock, and Jeff Meister.

Ultimate thanks go to Kevin Sullivan (University of Virginia), James D. Hollan, Ingolf Krueger, and Jens Palsberg (UCLA). Thank you for reading the drafts of my work and providing crucial feedback. And I would like to thank Andrew P. Black and the anonymous reviewers from *PASTE ’08*, *WRT ’08*, *ESEC-FSE ’07*, and the *ICSE ’07 Doctoral Symposium*, who reviewed earlier drafts of my work.

This work was supported by an Eclipse Innovation Award from IBM, a Cal-(IT)^2 fellowship, and an NSF Science of Design grant (CCF-0613845).

Most importantly, I want to thank my family. This work is dedicated to you.

Sections 1.1.1–1.1.2, and 4.1, and Chapters 2–3 and 6–7, in part, are a reprint of the material as it appears in Beyond refactoring: a framework for modular maintenance of crosscutting design idioms. 2007. Shonle, M., Griswold, W., and Lerner, S. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering* (Dubrovnik, Croatia, September 03–07, 2007). ESEC-FSE '07. ACM, New York, NY, 175–184, © 2007 ACM, Inc. http://doi.acm.org/10.1145/1287624.1287650. The dissertation author was the primary investigator and author of this paper.

Chapter 4, in part, is a reprint of the material as it appears in Addressing Common Crosscutting Problems with Arcum. 2008. Shonle, M., Griswold, W., and Lerner, S. In *8th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 64–69, © 2008 ACM, Inc. http://doi.acm.org/10.1145/1512475.1512489. The dissertation author was the primary investigator and author of this paper.

Section 4.4.4, in part, is a reprint of the material as it appears in When Refactoring Acts like Modularity: Keeping Options Open with Persistent Condition Checking. 2008. Shonle, M., Griswold, W., and Lerner, S. In *Second ACM Workshop on Refactoring Tools (WRT)*. The dissertation author was the primary investigator and author of this paper.

Chapter 5, in whole, is a reprint of the material as it appears in The techniques programmers use to cope with crosscutting using Arcum. 2008. Shonle, M, Griswold, W., and Lerner, S. UCSD CS2008-0933, December 5, 2008. The dissertation author was the primary investigator and author of this paper.

VITA

2000	Bachelor of Arts, Clark University, Worcester, Massachusetts
2006	Master of Science, University of California, San Diego
2009	Doctor of Philosophy, University of California, San Diego

PUBLICATIONS

Shonle, M, Griswold, W., and Lerner, S. 2008. The techniques programmers use to cope with crosscutting using Arcum. UCSD CS2008-0933, December 5, 2008, 12 pages.

Shonle, M., Griswold, W., and Lerner, S. 2008. Addressing Common Crosscutting Problems with Arcum. In *8th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 64–69, 2008.

Shonle, M., Griswold, W., and Lerner, S. 2008. When Refactoring Acts like Modularity: Keeping Options Open with Persistent Condition Checking. In *Second ACM Workshop on Refactoring Tools (WRT)*, 4 pages, 2008.

Shonle, M., Griswold, W., and Lerner, S. 2007. Beyond refactoring: a framework for modular maintenance of crosscutting design idioms. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering* (Dubrovnik, Croatia, September 03–07, 2007). ESEC-FSE '07. ACM, New York, NY, 175–184.

Shonle, M. 2007. Modular-Like Transformations and Style Checking for Crosscutting Programming Concepts. In *Companion to the Proceedings of the 29th international Conference on Software Engineering* (May 20–26, 2007). IEEE Computer Society, Washington, DC, (ICSE Doctoral Symposium), pages 95–96, 2007.

ABSTRACT OF THE DISSERTATION

A Framework for the Checking and Refactoring of Crosscutting Concepts

by

Macneil Charles Shonle

Doctor of Philosophy in Computer Science

University of California, San Diego, 2009

Professor William G. Griswold, Co-Chair
Professor Sorin L. Lerner, Co-Chair

Modularity is a fundamental technique used for the composition of large software systems. Under modularity, design decisions that are likely to change are encapsulated within individual modules. However, programmers also employ crosscutting concepts, such as design patterns and programming idioms, which cannot be effectively modularized. Consequently, implementations of these crosscutting concepts can be expensive to change, even when the code is well-structured.

In this dissertation, I describe an extension to the refactoring paradigm that provides for the modular maintenance of crosscutting concepts, supporting both substitutability of implementations and the checking of essential constraints. This extension was realized through the Arcum framework, a plug-in for Eclipse that allows programmers to describe the use of their crosscutting programming concepts with a declarative language.

I present the conceptual underpinnings of the Arcum approach, and show how Arcum can be used to address several classical software engineering problems. I also present evidence from a user study of three pairs of programmers showing that Arcum can be easy to learn and use.

Chapter 1

Introduction

The difficultly of creating software on schedule, error free, and within estimated costs is frequently cited as part of a “software crisis” [Gla02]. And, until there is a revolution in software development that changes the essential complexity of software, these difficulties will remain [Bro87]. However, software developers do have tools to help manage the complexities that they encounter; the most important of which is modularity. Modularity is a regime that allows programmers to decompose programs into smaller units, called modules, which have interfaces that define how the other modules in the system can interact with them.

Modularity enables abstraction, information hiding [Par72], modular substitution, and interface checking, among several other benefits [BC99]. The benefit of modular substitution is that the implementation of a module can be improved (or even replaced by another module) without requiring changes to, or extensive retesting of, other parts of the system. As a result, the cost of performing experiments with a module’s implementation are reduced, which enables more value to be added to the system. The benefit of interface checking is that many programming errors, such as type errors, can be detected by the compiler.

However, due to limitations of the programming language, not every logical unit of a program, called a *concern*, can easily be encapsulated into a module. When a concern’s implementation cuts across the implementation of other concerns, it is referred to as a *crosscutting concern*. For example, a technique known as the visitor pattern is employed when a heterogeneous set of data-structures needs to be traversed in a depth-first order [GHJV95]. Because the visitor pattern is a traversal over heterogeneous types, its traditional implementation is scattered over several modules in an object-oriented language.

Crosscutting often leads to bugs and longer development times because the concern's implementation is harder to reason about. In the case of the visitor pattern, for example, a programmer adding a single field to a class can inadvertently require new code to be introduced, or existing code to be modified, potentially affecting many separate classes. Additionally, such scattering makes the process of modifying crosscutting code tedious and error prone. Unfortunately, such crosscutting is not rare. Crosscutting programming concepts, such as design patterns and programming idioms, are a common form of crosscutting encountered in virtually every large program. Such crosscutting concepts might be general in nature—such as the visitor pattern—or domain-specific in nature, applying only to a particular family of programs.

Even a well-designed program might require change tasks that are crosscutting in nature, and thus outside of modular bounds [Gri01]. For example, it's impossible for every future change to be anticipated, and so a program's existing abstractions may not modularize a given change. Sometimes, the language's abstraction mechanisms are not powerful enough to permit an efficient modularization. Other times, an agile development process like Extreme Programming may intentionally delay the introduction of such abstractions [BA04].

1.1 Solutions for Crosscutting

The problems introduced by, or exacerbated by, crosscutting have been addressed in several different solution spaces, spanning language-based solutions (Section 1.1.1) and tool-based solutions (Section 1.1.2).

1.1.1 Language-Based Solutions

A language-based solution for crosscutting code is one that introduces a new programming language that can provide better expressions of the kind of designs that lead to crosscutting. Because crosscutting can lead to bugs and increased development times, language solutions aim to reduce the amount of code that needs to crosscut in the first place.

The paradigm of aspect-oriented software development (AOSD) has introduced a host of aspect-oriented programming (AOP) languages. Notable examples of AOP languages are AspectJ [KHH$^+$01] and HyperJ (derived from the work on Multi-Dimensional Separation of Concerns [OT99, TOHS99, TOS02]).

AspectJ addresses crosscutting through new modular abstractions, called *aspects*. Program reasoning is improved with the help of aspects, because code related to one concern, which would otherwise be crosscutting, can now be reasoned about in isolation. For example, Hannemann has shown that the design of many crosscutting concepts, including design patterns, can be improved when written as aspects [HK02], resulting in greater flexibility and expressiveness, and improved reasoning. Even though the possibilities for new abstractions with this technique are promising, AspectJ in practice still has many limitations in its ability to fully modularize a concern. For example, if an aspect is not written carefully, although it can have some implementation details encapsulated, the knowledge of likely to change design decisions might still crosscut the rest of the program [SGS$^+$05, GSS$^+$06].

Another limitation of the AspectJ approach is that it is a solution focused on the development of new code: A Java program can be converted into an AspectJ program (because AspectJ is a superset of the Java language), but none of the benefits of AspectJ will be available until new aspects are written (or existing aspect libraries used). For example, a Java program that has tangled code (a symptom of crosscutting) will first need to be rewritten into a more flexible aspect form.

The aspect-oriented solution is only one angle of attack for a language-based solution for crosscutting. Other solutions instead focus on means for programmers to better express their intentions to the programming environment. For example, Explicit Programming with Elide extends the Java language to allow user-defined modifiers to be applied to Java program elements, such as classes, fields and methods [BCVM02]. The modifiers can change the semantics of the program element to which it is attached. For example, a class marked with appropriate serialization modifiers can have serialization methods automatically added to it. Such a system allows programmers to focus on what is needed instead of on implementation details. More expressive systems in general have less redundancy than less expressive systems, and redundancy is one source of crosscutting. For example, once a class's field changes, the serialization method of the class will also need to be changed.

The static metaprogramming system of Dincklage [vD03] is similar to Elide but works with the Common Lisp system. The goal of the metaprogramming system is to provide direct support for constructs and design patterns that object-oriented programmers routinely use.

Presentation Extension [EK07] is similar to the Elide and Metaprogramming work, but it seeks to allow changes to the semantics of a program through presentation extensions. A presentation is a rendering of a program in the programming environment, which can take several forms (for example, representing a call to a square root function using the mathematical symbol). A presentation extension is similar to a syntax extension, but all necessary syntax extensions are achieved through Java 5 style metadata annotations.

DRIVEL is a program enhancement system using generative techniques on top of an aspect-oriented language [TB08]. What differentiates DRIVEL from the other language extension systems discussed is that it does not need to extend the syntax or presentation of the programming language. Instead, programmers write Java programs that assume that certain program elements are already defined. If the element is not present, DRIVEL will generate it automatically. For example, a call to a visitor function that is not defined will signal to DRIVEL to generate the infrastructure necessary for the visitor pattern. This technique is particularly well suited for design patterns, because the code that needs to be generated can be inferred from the context based on role usage. Even though code developed with DRIVEL or Presentation Extension are saved as syntactically valid Java code, both must be processed with special compilers, effectively making both of these techniques extensions to the programming language.

All of the programming language extensions discussed thus far in this section are general purpose in their aims. That is, programmers should be able to be naturally decomposed programs into a paradigm dictated by the language, and, as a result, the problems of crosscutting are directly addressed. However, such generality can be hard to achieve, because some software engineering problems are domain-specific: that is, the problems do not easily allow for programming language constructs to be generalized such that the constructs are widely applicable. The typical course of action is that—instead of making languages more complex by having infrequently used, but significant, features for these special cases—programming languages are kept simpler. The result is that opportunities can be missed: There are known solutions to a problem, but they do not occur regularly enough to be addressed by a general purpose language.

In this space is where domain-specific solutions shine: For example, there are extensions to Java that deal directly with the visitor pattern, such as DemeterJ [LO97], which has support for the many variations of the visitor pattern and provides for the customization and optimization of traversal paths (a feature absent from the other tools mentioned, even those that provided some support for the visitor pattern). The MultiJava language [CLCM00] can reduce the need for the visitor pattern through the use of multi-methods. MultiJava is a general purpose language, but it provides a direct solution for the domain-specific problem of multiple method dispatch. My own work in this space was the XAspects [SLS03] project, which aimed to make it easier for developers to create domain-specific solutions, which could integrate with other solutions. To paraphrase an old saying, the creed of the project was: “Generally, you won’t need this structure. But when you do need it, you really need it!”

One drawback of language solutions that extend the programming language (or introduce an entirely new language) is the risks associated with tool adoption: Compilers, debuggers, and other tools will need to be changed to accommodate the added features of the new language. If a feature of a tool that supports only the original language becomes too valuable to discard, the project will need to be converted back. Indeed, some of these tools for extensions to Java provide an “export to Java” feature that reduces this risk, but the translated code often cannot later be translated back into the original form again.

Often with language extensions the finished executable is the result of several compilation steps, usually with the final step being on the base language that was extended. A drawback with this approach is the traceability of the code that the programmer directly works on. For example, a programmer focusing on one method in AspectJ might need to know that the method is advised by a separate aspect. Extensions to the programming environment can improve the traceability of aspect-oriented programs, which leads to another solution for crosscutting: tool-based solutions.

1.1.2 Tool-Based Solutions

Tool-based solutions complement language-based solutions. Instead of aiming to change the programming language, the problems of crosscutting are addressed through tools: namely, what can be learned about the program and how the program can be changed.

A long line of programming tools are available to cope with crosscutting: In addition to text editors and the Unix grep command are tools like CScope [Ste85] and Integrated Development Environments (IDEs), such as IBM's Eclipse. Eclipse is a modern IDE that provides automated refactoring transformations, derived from the Refactoring Browser for Smalltalk [Rob99]. Refactoring is a meaning-preserving program transformation performed in order to improve the design of a program [Gri91, Opd92, Fow99]. An automated refactoring system like Eclipse can enable large scale changes to be made to a program through repeated application of smaller refactoring operations.

Refactoring tools also have the potential to improve code by reducing the liabilities associated with using programming language extensions. For example, certain design patterns implemented in Java can be refactored to more flexible AspectJ equivalents through the role-based refactoring tool by Hannemann et al. [HMK05]. With the role-based refactoring tool, programmers can build macro-refactorings from micro-refactorings. The basic idea is to support the refactoring of crosscutting concepts like design patterns by separately recognizing the code for each role in a design pattern (with programmer interaction), and then applying micro-refactorings to each of those roles to achieve the macro-refactoring. Marin et al. take a similar approach, although they assemble macro-refactorings from micro-concerns rather than roles [MMvD05].

Just as there were domain-specific language-based solutions for the problems of crosscutting, refactoring tools can address domain-specific problems. For example, one common crosscutting problem encountered is the use of libraries: The choice of one library over an alternative is a decision that gets scattered over the program wherever the library is used. Thus, when using an alternative library is desired, all of the existing code must be migrated. To automatically assist class library migration tasks, Balaban et al. employ declarative semantic notations for automatically retargeting code libraries in large codebases [BTF05]. They use a rich type system and a constraint solver to enable finding correct library call replacements that otherwise could not be found automatically (due to subtle issues like synchronization).

The Feature Oriented Refactoring (FOR) work of Liu et al. recognizes the cross-cutting and non-modular nature of the implementation of software features, which are often crosscutting [LBL06]. For example, adding bounds checking to a data-structure could crosscut that structure's implementation. With FOR, certain types of programs can be refactored into a base program and modular feature refinements. The features are refactored and composed through the application of advanced delegation techniques. The application of FOR allows optional features, such as bounds checking, to be removed, enabling the deployment of better-suited variants.

Other tools allow programmers to better express changes by understanding the program better. For example, Simonyi's Intentional Programming (IP) [Sim95] aims to have programmers work at the level of their intentions, allowing for easier change to programs. Instead of being a refactoring system, IP utilizes a program-as-database approach: If any linked entry changes, the change follows all links backward.

The REFINE system also employs a program-as-database approach, in addition to program templates, which can be used for both pattern matching and code transformation [KM89]. The code transformations discussed were directed at uses such as "eliminate redundant multiplies by 1" and code mutations for test suite validation. The source template and destination template are bound in the same transformation rule. Also, alternative transformations cannot be introduced without duplicating existing rules.

As a departure from REFINE, Kozaczynski et al. employ semantic pattern matching—including control-flow and data-flow—to recognize "concepts" as part of a code transformation system for software maintenance [KNE92]. A more recent work in this area is the DMS system, which is similar to Kozaczynski et al. but has a much wider scope [BPM04].

The iXj program transformation system for Java allows for pattern matching of code related to a crosscutting concept [BG04, Bos06]. The iXj system is interactive and allows for programs to be changed at a level more sophisticated than a text editor's find and replace feature.

Another wide class of tools that can address crosscutting are static program analysis tools, such as SLAM [BMMR01], ESP [DLS02], and PDL [MVW07], which can be used to find bugs, which are often crosscutting in nature.

One shortcoming with tool-based solutions for crosscutting is that they are task focused: the tool only executes when requested by the user. As a result, the tool itself is not "part of the loop" of the development cycle: Whatever important design information that is provided to the tool is lost until the tool used again (often with the user repeating the high-level design information again). For example, a programmer using Eclipse might have the intention of encapsulating a field. After running the 'Encapsulate Field' refactoring, the code transformation is complete, yet there is nothing to prevent other programmers on the project from unencapsulating the field (for example, by making a private method directly access the field's value instead of using a getter method). The programmer's intentions have been forgotten by the development environment, and so the programmer must communicate this information by documenting it in a comment or a separate document.

1.2 Overview of the Arcum Approach

My hypothesis is that *The benefits of modularity afforded to the tasks of software evolution and constraint checking can be extended to crosscutting programming concepts—such as design patterns and programming idioms—through additions to the programming environment instead of the programming language.* Even when the programming language itself cannot completely modularize a crosscutting concept's implementation, the well-structured form of that implementation makes it possible for the programming environment to be able to check and transform code related to it. In this dissertation, I present my work on the Arcum framework [Sho07, SGL07, SGL08b, SGL08a]. Arcum allows for certain forms of crosscutting concepts that are implemented in the Java programming language to be automatically transformed into alternative implementations. Arcum's goal is to be a means for programmers to express their intentions to the development environment. In that spirit, the name 'Arcum' is derived from the Latin phrase *intendere arcum*, which means "to aim a bow and arrow at" and is the metaphorical root of the word *intention* [Den92, p. 333].

Arcum expands the opportunities for modular substitution and reasoning through *options*. An Arcum option declares the implementation details of a crosscutting concept. Such implementation details include any required supporting code and infrastructure to make a complete and correct implementation. For example, one crosscutting concept could employ an idiom that uses a wrapper function; the wrapper function itself would then be a required component of a complete solution.

A group of options are related to each other when they all implement the same Arcum *interface*. An Arcum interface states the stable properties that are common to all options that implement it. The relationship between an Arcum interface and its options is similar to the relationship between a Java interface and the classes that implement that interface.

Arcum declarations are auxiliary supplements to Java programs. A programmer may be motivated to declare one or more options when the need arises for either transforming a crosscutting concept or for better checking of a particular implementation. Once declared, transformation is merely a matter of specifying the replacement of the prevailing option with an alternative option. The correctness of such a replacement is aided by checks specified in the Arcum declarations. An Arcum interface specifies behavioral constraints on its options, and each option specifies additional constraints specific to its implementation. Arcum declarations are written in a generic fashion so that they can be applied to multiple cases, enabling reuse of Arcum declarations.

There are several unique benefits of retaining Arcum declarations as persistent, supplemental descriptions to a Java program. For one, being persistent, unlike the typical refactoring operations invoked by a programmer via an IDE, an instantiated option is continuously checked (for example, every time the program is compiled), not just during refactoring. Continuous checking ensures that the ability to replace the prevailing option for an alternative option is preserved. Two, due to its declarative nature, an option provides a precise mechanism for documenting a crosscutting concept and expressing the programmer's intentions for its implementation. Finally, because Arcum declarations are supplements, the core source code remains unchanged, so that programs retain exactly the same Java semantics they had before. The program is changed only when one implementation is transformed to one of the alternative options. Such transformations are always done within the IDE at the programmer's discretion, by specifying a change in the prevailing option. The separation of Arcum code and Java code reduces the cost and risk of initiating the use of Arcum, and enables late-stage adoption.

Even though Arcum does not directly extend the Java programming language, its checks applied to Java code is a form of extending Java's type system (achieved through additional error messages enabled by the user). Hence, the flexibility of the Arcum approach relies upon the expressiveness of programmer specified declarations rather than upon the expressiveness of a new programming language.

Key to my hypothesis is the claim that programmers employ crosscutting concepts in their code. But there is the alternative view that programmers will code primarily to the affordances of their programming language. This view can be called the Whorfian hypothesis for programming languages. The Whorfian hypothesis, named after the linguist Benjamin Whorf, claims that "the language people speak controls how they think" [Pin07, p. 124]. As an example of its influence, Bjarne Stroustrup quotes the Whorfian hypothesis in preface to the definitive book on C++ [Str97].

Are programmers caught in a trap set by the Whorfian hypothesis? Programming languages have limitations when it comes to expressing certain concepts, so such a trap could mean many programming opportunities are missed. But the existence of the design patterns movement shows that programmers are not limited in their thinking by the language they choose. The cost of using a technique not directly supported by the programming language must be balanced by the expected benefit of using the technique. One possible explanation for the belief in the Whorfian hypothesis for programming languages is that programmers decompose their programs according to the paradigm of the language they are using. But this explanation reverses cause and effect [Pin07, p. 125]: Programmers have a choice of programming languages, and thus choose the language that fits the decompositions they have been trained to use and are familiar with.

1.3 Outline

Chapter 2 provides an overview of the Arcum approach and shows how constructs that mimic modules (i.e., interfaces and options) can be practical in addressing crosscutting concerns. In Chapter 3, I present the design of the Arcum language in greater detail and describe the algorithm for transforming between two options. A case study in Chapter 4 shows a breadth of applications of Arcum and demonstrates more of Arcum's capabilities, including performing transformations that are many-many instead of 1-1. Chapter 5 discusses a user study conducted on the use of Arcum for elementary coding and change tasks. I discuss possibilities for future work in Chapter 6, and I close with a few concluding remarks in Chapter 7.

Sections 1.1.1–1.1.2, in part, is a reprint of the material as it appears in Beyond refactoring: a framework for modular maintenance of crosscutting design idioms. 2007. Shonle, M., Griswold, W., and Lerner, S. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering* (Dubrovnik, Croatia, September 03–07, 2007). ESEC-FSE '07. ACM, New York, NY, 175–184. The dissertation author was the primary investigator and author of this paper.

Chapter 2

The Arcum Approach

This chapter illustrates the Arcum approach with an example centered on a simple Java program that processes HTML image elements. Image elements in HTML have an optional '`alt`'-tag attribute that specifies alternate text to display in place of the image. There are a variety of ways of implement this concept of "alternate text" in Java. For example, one can simply add a field named `altText` to the `Image` class that represents image elements, as shown in Figure 2.1. Alternatively, if one expects the alternate text to be absent often (meaning that it takes on a predefined default value), then storing the alternate text in an external table can save memory at the expense of processor cycles. Such an implementation is shown in Figure 2.2.

In this example, it would probably have been easy to anticipate the need to change the implementation of the alternate text attribute. As a result, the developer may have chosen to use getter and setter methods for the alternate text field, making the refactoring easier. However, I keep this overview example simple for the purposes of exposition.

Although this intentionally simplistic problem might be easy to anticipate, it is difficult to design software abstractions that are flexible enough to support all future changes. Furthermore, some programming methodologies, such as Agile development, in fact favor rapid development of prototypes, with refactorings being applied later in the development process, as needed. In either case, the reality is that refactoring and software evolution in general is a common occurrence in the development of large software systems.

```
01 public class Image {
02   String altText;
03   /* ... */
04
05   public Image(String alternative) {
06     this();
07     this.altText = alternative;
08   }
09
10   public String toString() {
11     if (altText == null)
12       return defaultAltText();
13     else
14       return altText;
15   }
16 }
```

Figure 2.1: A simple internal field implementation of the `altText` attribute.

To give an overview of my approach, I describe how a developer would refactor a large body of code from the internal field implementation of the “alternate text” concept to the static `Map` implementation, first using a regular IDE such as Eclipse (Section 2.1), and then using the Arcum framework (Section 2.2).

2.1 Refactoring using Eclipse

Although the code shown in Figure 2.1 has only two reads (lines 11 and 14) and one write (line 7), in a realistic codebase one would expect to encounter many references to the `altText` field that need to be modified.

A developer could use Eclipse’s built-in refactorings in the following way to transform the code in Figure 2.1 into the code in Figure 2.2: (1) Replace all references to the `altText` field in the original code with calls to getter and setter methods with the ‘Encapsulate Field’ refactoring; (2) Manually edit these getter and setter methods to call the appropriate `Map` operations instead; And, optionally, (3) inline away calls to the getter and setter methods with the ‘Inline Method’ refactoring.

Although these built-in refactorings make manual modification less onerous, the

```
public class Image {
  static Map<Image,String> altText = new IdentityHashMap<Image,String>();
  /* ... */

  public Image(String alternative) {
    this();
    Image.altText.put(this, alternative);
  }

  public String toString() {
    if (Image.altText.get(this) == null)
      return defaultAltText();
    else
      return Image.altText.get(this);
  }
}
```

Figure 2.2: Static `Map` implementation of the `altText` attribute.

problem remains the same: refactorings generally require many changes to be made to the code, and the tool performing the transformations is simply not aware of the structure that is present in the code being manipulated. This lack of structure awareness results in a variety of drawbacks, including: (1) Code refactoring is error-prone and tedious—it is error-prone, for example, because the manual editing of the getter and setter methods by the programmer occurs outside of Eclipse's meaning-preserving operations; (2) It is often difficult to switch back and forth from the original implementation to the refactored implementation; (3) Subtle bugs can be introduced, for example transforming

```
f(this.altText = y)
```

into

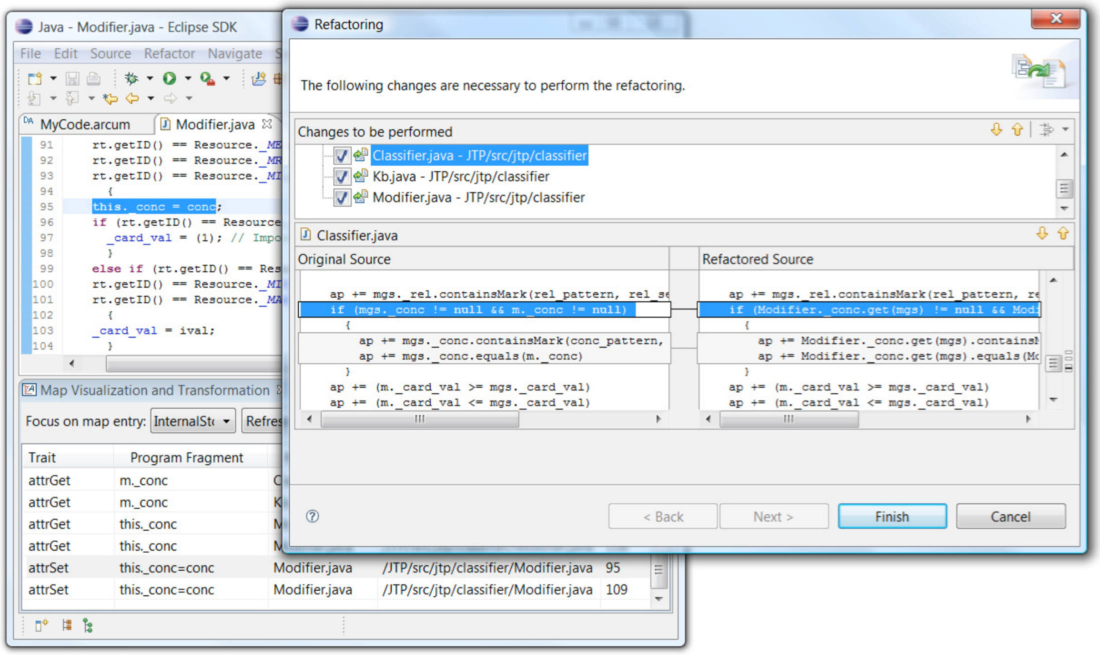
```
f(Image.altText.put(this, y))
```

is an incorrect transformation because the `put` method returns the *previous* value in the `Map`; And, (4) little of the work done for refactoring the `altText` field can be reused for refactoring other fields.

2.2 Refactoring using Arcum

The Arcum approach addresses the above limitations by enabling the programmer to formally capture implicit structure in his or her code. Rather than directly applying refactoring transformations, the programmer first declares behavior (in the interface) and implementation descriptions (in the options) for the code that will be changed. After the options and their interface have been described, the prevailing implementation can be replaced by any other related option. The chosen option is retained and continues to impose checks every time the program is compiled, to ensure that new or modified code also satisfies the transformation's pre- and post-conditions. Figure 2.3 shows the Arcum plug-in for Eclipse performing the refactoring.

Initially the programmer declares the `InternalField` option to be the current realization of the `AttributeConcept` interface. This specification is made concrete with the *mapping* shown in Figure 2.4. The refactoring transformation occurs when



```
    rt.getID() == Resource._ME
    rt.getID() == Resource._MR
    rt.getID() == Resource._MI
      {
    this._conc = conc;
    if (rt.getID() == Resource
      _card_val = (1); // Impo
      }
    else if (rt.getID() == Res
    rt.getID() == Resource._MI
    rt.getID() == Resource._MA
      {
    _card_val = ival;
      }
```

The following changes are necessary to perform the refactoring.

- Classifier.java - JTP/src/jtp/classifier
- Kb.java - JTP/src/jtp/classifier
- Modifier.java - JTP/src/jtp/classifier

Original Source:

```
  ap += mgs._rel.containsMark(rel_pattern, rel_se
  if (mgs._conc != null && m._conc != null)
    {
      ap += mgs._conc.containsMark(conc_pattern,
      ap += mgs._conc.equals(m._conc)
    }
  ap += (m._card_val >= mgs._card_val)
  ap += (m._card_val <= mgs._card_val)
```

Refactored Source:

```
  ap += mgs._rel.containsMark(rel_pattern, re
  if (Modifier._conc.get(mgs) != null && Modi
    {
      ap += Modifier._conc.get(mgs).containsM
      ap += Modifier._conc.get(mgs).equals(Mc
    }
  ap += (m._card_val >= mgs._card_val)
  ap += (m._card_val <= mgs._card_val)
```

Trait	Program Fragment			
attrGet	m._conc	C		
attrGet	m._conc	K		
attrGet	this._conc	M		
attrGet	this._conc	M		
attrSet	this._conc=conc	Modifier.java	/JTP/src/jtp/classifier/Modifier.java	95
attrSet	this._conc=conc	Modifier.java	/JTP/src/jtp/classifier/Modifier.java	109

Figure 2.3: Refactoring in Arcum for Eclipse. The front-most window is a preview that shows the changes the refactoring would perform. In the background is the Eclipse environment itself, with an Arcum view at the bottom that shows a compressed view of the implementation's scattered code fragments.

```
check {
  InternalField(targetType: Image, attrType: String, attrName: altText);
}
```

Figure 2.4: Mapping for the original `InternalField` implementation of the `altText` attribute. By using Eclipse to change ‘`InternalField`’ to ‘`StaticMap`,’ the Java code is automatically transformed to make the revised mapping hold.

Eclipse is used to change this mapping.

Figure 2.5 shows how the `Image` constructor from the example is transformed with Arcum. At the bottom left of the figure is the original Java code for the constructor, and at the bottom right is the desired target Java code. At the top of the figure is the Arcum code specifying the interface that both options implement. The declarations of the two options are directly below the interface: one using an internal field (the `InternalField` option), and another using a static `Map` (the `StaticMap` option).

Note that rather than designing a refactoring that applies only to the “attribute for alternate text” concept, the programmer has designed a parameterized interface so that it can be applied to any attribute of any class. In particular, the `AttributeConcept` from Figure 2.5 takes three parameters: `targetType` is the class for which the attribute is defined, `attrType` is the type of the attribute, and `attrName` is its name. Oftentimes a programmer might find the desired set of options in a library, rather than having to implement them from scratch.

The relevant part of the `AttributeConcept` code for this example is the `attrSet` *concept* (Figure 2.5, boxed), which represents all locations in the Java program where the attribute’s value is set. The members of the concept, in this case `setExpr`, `targetExpr` and `valExpr`, are fragments of Java code that are extracted from the locations in the code where the attribute is set. For example, `targetExpr` is the object whose attribute is being set, `valExpr` is the value that the attribute is being set to, and `setExpr` is the entire expression that performs the set operation. Arcum variables, such as `setExpr`, keep track of the source code location of the code fragment they represent. So, in this example, the `setExpr` variable identifies the location and scope of the set operation, which is later used to determine what portion of the code gets transformed or preserved.

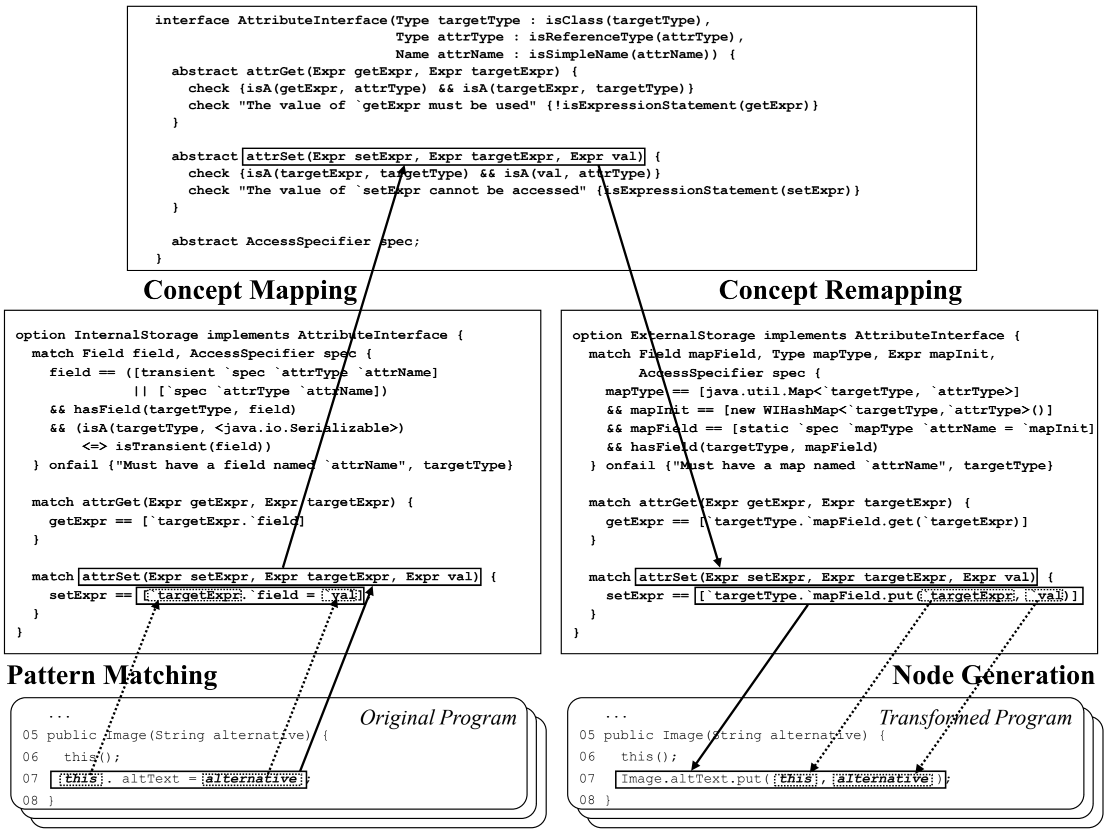
Each option specifies a different implementation of the `attrSet` concept. An abstract concept is implemented by an option when it provides a pattern that identifies the fragments of Java code that are instances of the concept. For example, the boxed pattern in the `InternalField` option shows how a regular assignment to a field becomes an instance of the `attrSet` concept. In the `Image` constructor example, line 7 of the Java code matches this pattern, and therefore becomes an instance of the concept. Similarly, the boxed pattern in the `StaticMap` option states which `Map` operations become instances of the `attrSet` concept.

Arcum patterns declaratively state the association between a crosscutting concept and the various fragments of Java code that implement an option. A key feature of Arcum is that these associations are *bi-directional:* Not only are the patterns used to build concept instances from Java code, but they are also used in the other direction, to generate Java code from concept instances.

The directionality of the association is determined by how the mapping is instantiated and later changed. As declared in Figure 2.4, the `InternalField` implementation is the prevailing option at the beginning of the scenario. To refactor to the sparse implementation of the `altText` field, the programmer changes the named option in the mapping to `StaticMap`, which triggers a refactoring of the code.

In the scenario, the `altText` field is initially implemented as a simple class field and the conceptual flow of information in Figure 2.5 goes in the clockwise direction, following the direction of the arrows. For example, the field assignment on line 7 in the original code is pattern matched into a concept instance, at which point the references `setExpr`, `targetExpr` and `valExpr` are bound. The newly constructed concept instance is lifted to the interface level, and then pushed back down to the alternate `StaticMap` option, at which point the pattern along with `setExpr`, `targetExpr` and `valExpr`, are used to construct the replacement code.

Due to the bi-directional nature of patterns, the mapping can be changed later to perform the refactoring in the other direction. Because my approach explicitly and persistently identifies substitutable crosscutting concepts and their prevailing options, their consistency properties can be continuously checked. This makes



```
interface AttributeInterface(Type targetType : isClass(targetType),
                             Type attrType : isReferenceType(attrType),
                             Name attrName : isSimpleName(attrName)) {
  abstract attrGet(Expr getExpr, Expr targetExpr) {
    check {isA(getExpr, attrType) && isA(targetExpr, targetType)}
    check "The value of `getExpr must be used" {!isExpressionStatement(getExpr)}
  }

  abstract attrSet(Expr setExpr, Expr targetExpr, Expr val) {
    check {isA(targetExpr, targetType) && isA(val, attrType)}
    check "The value of `setExpr cannot be accessed" {isExpressionStatement(setExpr)}
  }

  abstract AccessSpecifier spec;
}
```

```
option InternalStorage implements AttributeInterface {
  match Field field, AccessSpecifier spec {
    field == ([transient `spec `attrType `attrName]
              || [`spec `attrType `attrName])
    && hasField(targetType, field)
    && (isA(targetType, <java.io.Serializable>)
        <=> isTransient(field))
  } onfail {"Must have a field named `attrName", targetType}

  match attrGet(Expr getExpr, Expr targetExpr) {
    getExpr == [`targetExpr.`field]
  }

  match attrSet(Expr setExpr, Expr targetExpr, Expr val) {
    setExpr == [`targetExpr.`field = `val]
  }
}
```

```
option ExternalStorage implements AttributeInterface {
  match Field mapField, Type mapType, Expr mapInit,
        AccessSpecifier spec {
    mapType == [java.util.Map<`targetType, `attrType>]
    && mapInit == [new WIHashMap<`targetType,`attrType>()]
    && mapField == [static `spec `mapType `attrName = `mapInit]
    && hasField(targetType, mapField)
  } onfail {"Must have a map named `attrName", targetType}

  match attrGet(Expr getExpr, Expr targetExpr) {
    getExpr == [`targetType.`mapField.get(`targetExpr)]
  }

  match attrSet(Expr setExpr, Expr targetExpr, Expr val) {
    setExpr == [`targetType.`mapField.put(`targetExpr, `val)]
  }
}
```

```
  ...
public Image(String alternative) {
  this();
  this . altText = alternative;
}
```

```
  ...
public Image(String alternative) {
  this();
  Image.altText.put(this, alternative);
}
```

Figure 2.5: Overview of how Arcum transforms code into an alternative implementation. By changing the mapping, which describes the current option deployed, the code is automatically transformed in accordance with the substituted option.

future transformations easier to perform because the checking aids code compliance. For example, as mentioned previously, a programmer would like to prevent the incorrect application of the refactoring to transform `f(this.altText=y)` into `f(Image.altText.put(this,y))`, because the `put` method returns the *previous* value in the `Map`. This requirement is checked by the interface in Figure 2.5 with the `check` clause (where `isExpressionStatement(e)` tests whether the expression `e` is used directly as a statement). This `check` clause is checked continuously, to make sure that developers don't accidentally change code in a way that prevents the crosscutting concept from being transformed.

Much of the power of the Arcum approach arises from the fact that its transformations are focused on preserving the requirements as asserted in the interface, rather than preserving every detail of language-semantics-level behavior. I still call this refactoring, in deference to the programmer's intent that an Arcum interface specifies the important behaviors that need to be preserved during transformation to another implementation.

2.3 The Arcum Language

The key construct of the Arcum language is the *concept*, which describes a collection of Java code fragments (Section 2.3.1). All Arcum code appears in one of three declarations: an interface, an option, or a mapping. An interface (2.3.2) specifies the names and types of the concepts common to all options that implement the interface. An option (2.3.3) provides concrete definitions of the concepts in the interface by using *patterns*. Finally, a mapping (2.3.4) allows options to be parameterized for a particular application.

2.3.1 The Concept Construct

A *concept* is used to describe one distinct role, structure, or operation that occurs in a crosscutting concept's implementation (a similar construct is referred to as a sub-concept by Kozaczynski et al. [KNE92]). A concept can either represent a single program fragment or a set of tuples of program fragments (potentially the empty set).

Concepts can have boolean conditions associated with them, which allow optional user-readable error messages to describe what code was expected.

The `AttributeConcept` in Figure 2.5 specifies three abstract concepts: `attrGet`, `attrSet`, and `accessSpecifier`. Because an Arcum interface is abstract, its concepts must be too. A concept's name, tuple member types, `check` clauses, optional error messages, and a partial specification may be specified in an interface. These abstract concepts are given concrete definitions via patterns specified in the options that implement its interface. Any partial specifications provided at the interface level are conjoined with the complete specification at the option level. Such partial specifications are allowed to help prevent code duplication for common restrictions.

The `attrGet` and `attrSet` concepts represent, respectively, the abstract operations of getting and setting the attribute, where both operations can occur in the program multiple times.

Each concept tuple has a root program fragment, of which all other members in the tuple are sub-members. In syntactic terms, the root program fragment is an AST node that is directly or indirectly the parent of all other AST nodes in the tuple. For example, the `attrSet` concept has three tuple members, each of which are expressions: `setExpr`, `targetExpr`, and `valExpr`. The `targetExpr` is the expression whose value is the object that has the attribute and `valExpr` is the new value for the attribute. Both of these are sub-expressions of `setExpr`.

When a concept's root fragment is an expression or statement it represents an operation. But concepts can also express structural properties of code and other cross-cutting forms. For example, one could declare a concept that represents "all declarations of type `Vector`." Such a concept would be useful for porting from one library to another (for example, changing uses of the `Vector` collection class to the newer `ArrayList` class, as done by Balaban et al. [BTF05]). Structural examples of concepts include methods, fields, and annotations.

The Check Clause

Programmers can add multiples conditions with error messages to a concept. For example, the abstract `attrGet` concept from Figure 2.5 declares:

Table 2.1: Example built-in predicates.

Predicate Name	Description
isA(t_1, t_2)	Is t_1 equal to or a subtype of t_2?
hasMethod(c, m)	Does class c have a method m?
isTransient(f)	Is field f declared as transient?
isReferenceType(t)	Is t a reference type?
isExpressionStatement(e)	Is e the contents of a statement?

```
check "The value of `getExpr must be used" {
  !isExpressionStatement(getExpr)
}
```

The above `check` tests that the expression specified by `getExpr` cannot have its value discarded. The error message in the programming environment is associated with the root program fragment. Other fragments mentioned in the text of the message (escaped with a backticks "`") can help focus the user's attention to the specific cause of the problem.

In general, the Boolean condition provided in a `check` clause is evaluated for each concept instance. Conditions are expressed using a simple propositional logic, augmented with built-in primitives, examples of which are shown in Table 2.1. A `check` clause's error message is optional, to provide a compile-time error message to the user in the event that the condition cannot be satisfied. When an error message is not specified a default error message describes the condition that failed.

Interface Parameters

An interface's parameters is a special concept whose members are specified in the mapping instead of by the options that implement the interface. Parameters are typically required for an Arcum interface declaration to be instantiated for multiple uses. The `AttributeConcept` in Figure 2.5 has three parameters: a `Class` named `targetType` (the class that has the attribute), a `Type` named `attributeType` (the type of the attribute), and a `String` named `attrName` (the name of the attribute). This parameterization permits the `AttributeConcept` to be applied to several different attributes instead of, for example, being hard-coded only for `String`s in the `Image` class.

The type of the parameter tuple in this example demonstrates how interface properties must hold for all options. Because the `InternalField` option assumes a field can be added to the `targetType` it must assume the `targetType` is a Java class and not a Java interface, and thus all other options must make the same assumption as well. This is an example of how the interface must accept the least common denominator—an essential property of modular substitution in general. For example, a `List` interface would not allow random access, even though some implementations of a `List` structure could permit it.

2.3.2 The Interface Declaration

An Arcum interface declares, at the behavioral level, what is common to all of its implementing options. The interface's primary purpose is to document the crosscutting concept's interface and to centrally specify requirements that apply equally to all options that implement it.

Concepts specified in the Arcum interface are abstract and all concepts must be concretely implemented by the options that implement the interface.

The abstract `spec` concept in Figure 2.5 represents a single program fragment, of type `AccessSpecifier`. This concept is used by the `AttributeConcept` to simplify the interface's parameters list. Instances of the `AccessSpecifier` type specify one of the modifiers `private`, `public`, `protected` or the implicit "package" modifier. Both the `InternalField` and `StaticMap` options infer this specifier by defining it to be the access specifier of the field named `attrName`. This same kind of inference can be used to remove the `attrType` member from the parameters list as well. The decision on whether to let a program fragment be a parameter or an inferred concept depends on the current and expected options that the interface will modularize. For example, it may always be safe to assume that the `spec` can be inferred but not safe to assume that the `attrType` can always be inferred—for instance, an implementation that did not use the Java 5 generic `Map` class would not have this type information available in the Java source code, and thus would have to be specified in the Arcum mapping.

2.3.3 The Option Declaration

An option describes one complete implementation of a crosscutting concept specified by an Arcum interface. Options use the `implements` keyword to specify which interface they implement. Unlike classes, an option can only implement one interface. The constructs used to implement options are patterns and invariant condition checks.

Patterns

Options specify the implementation of concepts using declarative patterns, which are used to both identify and construct program fragments. Patterns are expressed as Java-like pseudo-code inside square brackets, with backtick marks to identify Arcum variables inside the pseudo-code. For example, the `InternalField` option in Figure 2.5 uses the following pattern to match (or generate) all valid set expressions:

```
match attrSet(Expr setExpr, Expr targetExpr, Expr valExpr) {
  setExpr == [`targetExpr.`field = `valExpr]
}
```

Arcum supports patterns to match different kinds of Java program fragments. The algorithms that use patterns for matching and for generating new AST nodes are discussed in Section 3.2. An Arcum variable can be associated with either a single pattern expression or a union of several pattern expressions (combined with the `||`-operator).

An option can match program fragments that are specific to the implementation of the option. For example, a concept in the `InternalField` option in Figure 2.5 has the single member `field`:

```
match Field field, AccessSpecifier spec {
  field == ([transient `spec `attrType `attrName]
            || [`spec `attrType `attrName])
  /* ... */
}
```

Because the `field` concept is local to the option this field does not have to be present in any of the alternative options. Here, a disjunction is used to reflect that the `transient` modifier may or may not be present; Section 3.2.2 discusses how Arcum determines which pattern to use for code generation.

Invariant Condition Checks

An option can place additional requirements on an abstract concept with the `check` clause. This is needed, for instance, for the field access pattern shown in `InternalField`'s implementation of the `attrGet` concept, which requires constraints that are equivalent to:

```
match attrGet(getExpr, targetExpr) {
  getExpr == [‘targetExpr.‘field]

  check {
    isA(attrType, getExpr)
    && isA(targetType, targetExpr)
    && !isExpressionStatement(getExpr)
  }
}
```

Consider the `!isExpressionStatement(...)` requirement, for example. In Java, you cannot use a field access as a statement. Therefore, the expression statement

```
this.altText;
```

is rejected by the Java compiler, even though the expression statement

```
Image.altText.get(this);
```

is accepted. The `!isExpressionStatement(...)` requirement prevents the latter from being inadvertently written.

This condition must hold over all declared options for the interface, otherwise substitutability is not preserved. Hence, Arcum considers all of the checks that options place on abstract concepts, and continuously checks them. This is a slight departure from typical interface semantics, which doesn't allow implementations to automatically impose constraints on their alternatives. Such constraints are best stated in the interface declaration, but the Arcum approach allows for the emergent conditions to be stated either in an option or explicitly pushed up to the interface.

Options may also define static concepts purely for checking implementation-specific details. As an example, in the substitution scenario from Figure 2.5, once the code has been refactored to the `StaticMap` option, programmers are prevented from performing operations other than calling `get` and `put` on the `altText Map`, because these method calls (such as `altText.clear()`) would not have an analogue in the `InternalField` option. A definition can be used to match these illegal uses of the `Map` to prevent a programmer from writing new code that violates the `AttributeConcept`'s specifications:

```
define anyAccess(Expr expr) {
  expr == [`targetType.`mapField...]
  check {
    attrSet(expr, _, _) || attrGet(expr, _)
  }
}
```

Here, the special underscore variable is used to accept any binding that matches. This example also demonstrates how already matched concepts (like `attrSet` and `attrGet`) can be used as predicates.

2.3.4 Mappings

Arcum mappings are used to state which options are implemented in a program. Figure 2.4 shows a sample Arcum mapping. In general, an Arcum mapping is a list of option instantiations, where each instantiation states the option's name, and a set bindings for all of the option's parameters.

One benefit of the mapping format is that there is a separate file that documents some of the architecture of the program and the design decisions made. Such a record provides programmers with a different view into the decision space of the program: When changes need to be considered, the list of mappings shows the decisions made, and thus provides insight into the possible alternatives to consider.

2.4 The Arcum Development Process

This section describes some of the design decisions that came up in the process of writing the Arcum code for `AttributeConcept`. In doing so, I bring to the forefront common issues that Arcum developers will think about and have to address when they write Arcum code.

I found that there were two key considerations in the design of a group of related options. The first, of course, is that any resulting transformations should satisfy the specifications for the crosscutting concept. Analogously, if the programmer edits Arcum code in a way that violates the intended use of the specification, then an error should be reported. The second is how to structure the interface so that it admits a suitable range of options without being so general as to unnecessarily complicate the implementation of options.

In practice, I found that it was hard to get these right the first time. But I also found that it wasn't necessary. My first version of the `AttributeConcept` interface was correctly parameterized, but its internals omitted extra condition checking. When I wrote an option, which described one particular implementation, I added explicit checks to the interface—the invariance of condition checks drove this process (see Section 2.3.3). As a result, Arcum handles some of the burden of knowing and applying Java language rules.

The target program did not employ serialization of the target type, and, as a result, the previously discussed serialization checks were not required. Later, when I came across some serialization code, I realized that a general-purpose `InternalField` specification would have to account for this special case. If I had been just developing the related options for my own use, I might have ignored this insight or simply added a check to prevent the application of the `AttributeConcept` to a serializable class. However, under the assumption that `AttributeConcept` might become part of a reusable library—and thus clients could benefit if this corner case was addressed—I added the necessary checks to insure that the attribute will be appropriately serialized in all options.

Other examples of the kinds of checks written have been discussed previously, such as:

- Type consistency constraints;
- Restrictions on the use of certain methods and fields; and
- Checks for making sure that fields are appropriately labeled `transient` in the face of `Serializable` classes.

Over the history of `AttributeConcept`'s development, the concept signatures did not change, thus its external clients were unaffected by the numerous improvements. However, there were a variety of changes that involved an interaction between the interface and an option. This is not surprising, as their interaction is analogous to the interplay between a superclass and its subclasses, which often collaborate intimately.

Chapter 2, in part, is a reprint of the material as it appears in Beyond refactoring: a framework for modular maintenance of crosscutting design idioms. 2007. Shonle, M., Griswold, W., and Lerner, S. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering* (Dubrovnik, Croatia, September 03–07, 2007). ESEC-FSE '07. ACM, New York, NY, 175–184. The dissertation author was the primary investigator and author of this paper.

Chapter 3

Arcum's Design and Implementation

The prototype implementation of Arcum is a plug-in for the Eclipse IDE and is built on top of Eclipse's Java Development Tools plug-in and uses the Language Toolkit API to perform AST transformations. The source for the Arcum plug-in comprises 19,000 lines of Java code. Arcum uses Eclipse's parsers and type checkers for pattern matching and Eclipse's factories for node generation. Arcum's predicate solver is based off of similar solvers from declarative languages, such as Prolog [CR93].

The Arcum language by design is special purpose and not Turing complete—a restriction to make Arcum code simpler to debug, because termination of the solver can be guaranteed. The details of Arcum's solver are discussed in Section 3.1. Because Arcum has a declarative language that can be used to query Java programs, it is related to a large family of program query tools, such as JunGL [VEdM06], QL [MVH$^+$07] and PQL [MLL05]. Some of these systems support additional checks to apply to code, but do not use these checks to infer program transformations.

Arcum's other design criteria is that transformation steps are always inferred instead of explicitly written. For example, in the development of the language, I considered expressing transformations as finding a match and then specifying what the match should be translated into. In several ways, such a mechanism would have been comparable to the current Arcum implementation, but it would lack the parallels with modularity: mainly, stable interfaces and implementations for those interfaces. This modular view was achieved by using the transformation algorithm discussed in Section 3.2.

```
define hasA(Type t, Type u) {
  directlyHasA(t, u)
  || hasAListOf(t, u)
  || (hasA(t, v?) && isSubtypeOf(v, u))
}

define directlyHasA(Type t, Type u) {
  hasField(t, field?)
  && !isStatic(field)
  && isA(field, u)
}

define hasAListOf(Type t, Type u) {
  exists (Type v : hasA(t, v)) {
    v == <‘u[]>
    || (isA(c?, <Collection>) && v == <‘c<‘u> >)
  }
}
```

Figure 3.1: The `hasA` relation: Defined in terms of itself, `directlyHasA` and `hasAListOf`, where `hasAListOf` is defined in terms `hasA`.

3.1 Monotonic Fixed-Point Solver

The Arcum solver for predicates guarantees termination by requiring a strict monotonicity requirement: At each step of the evaluation, the set of tuples of program elements that belong to a predicate is unioned with a set of new tuples. The solver terminates once a fixed-point is found; i.e., when the existing set does not change after a complete iteration. Once a tuple is added to the set, it will always be a member. Thus, the solver always reaches a fixed-point because the size of the set is bounded by the number of elements in the *n*-ary crossproduct of all program fragments, where *n* is the number of members of each tuple.

Predicates defined in Arcum can be (mutually) recursive. Figure 3.1 is an example with two mutually defined relations: The `hasA` relation depends on itself and `hasAListOf`, and `hasAListOf` depends on `hasA`. The `directlyHasA` relation is not recursive and depends only on the built-in relations: `hasField`, `isStatic`, and `isA`.

The case of solving the `directlyHasA` is the simplest: Initially, the set of tuples for the `directlyHasA(`*t*`,` *u*`)` relation is empty, with no bindings for any *t* or *u*. The fixed-point solver runs through the following steps to try to satisfy the expression (a conjunction of three subexpressions):

`hasField(`*t*`, `*field*`?) && !isStatic(`*field*`) && isA(`*field*`, `*u*`)`

1. `hasField(`*t*`, `*field*`?)`: Because `hasField` is a built-in relation (and defined according to the compiled program) every solution for *t* and *field* are returned as results. The question-mark notation used here ("*field*?") declares a new binding for *field*.

2. `!isStatic(`*field*`)`: The solver then tries to satisfy the next conjunct: Because there are already bindings for *field*, solving this conjunct will act as a filter. In this case, solving it will filter out all fields that are not static.

3. `isA(`*field*`, `*u*`)`: Given this potentially shorter list of bindings for *field* (and the bindings for *t* associated with those *field* bindings), the last conjunct is solved. Because `isA` is a built-in predicate the solver knows all of the *u* bindings that correspond to the set of `field` bindings.

4. Finally, a projection of the tuples `(`*t*`, `*field*`, `*u*`)` is taken, keeping only *t* and *u*.

Because the set of predicates that the `directlyHasA` predicates depends on are all solved (that is, the set cannot gain new elements, since every candidate in the finite program has been exhausted), the solution arrived at after this first iteration of the algorithm is the complete solution, and the predicate can be marked as solved.

If the first two conjuncts in the above expression were swapped, forming:

`!isStatic(`*field*`?) && hasField(`*t*`, `*field*`) && isA(`*field*`, `*u*`)`

then the evaluation process is similar, but potentially longer to evaluate, because the *field* is implicitly bound to every field in the program, before the `!isStatic(`*field*`?)` filter is applied.

Next in the process is the solution of the `hasA` and `hasAListOf` predicates. Because these two predicates are mutually dependent, the iterations of their solution must be interleaved until the contents of their solution sets don't change.

```
01 define doesNotPrint(Method m) {
02   forall (Method n : invokes(m, n)) {
03     !hasSignature(<java.io.PrintStream>, n)
04     && doesNotPrint(n)
05   }
06 }
```

Figure 3.2: A broken specification for the `doesNotPrint` predicate. The predicate is intended to be true when method *m* performs a printing operation or makes a call to another method that does.

Because the fixed-point solver starts with the empty set as a solution, and then builds the set up, monotonically increasing its size, programmers must be careful in how declarations are solved. For example, consider a `doesNotPrint(`*m*`)` predicate, which is supposed to hold for a method *m* if *m* does not make any `PrintStream` calls and also doesn't call any methods that directly or indirectly make `PrintStream` calls. Figure 3.2 is one attempt at coding this predicate. On the first iteration of the solver, the set is empty and thus the conjunct `doesNotPrint(`*n*`)` on line 4 will not find any solutions for *n*. As a result, no methods *m* that satisfy the predicate will be found.

The problem with this attempted solution is that its base-case (the solution after the first iteration) does not contain instances that would satisfy a final solution. To resolve this issue, the solution must be conceived in terms of starting with known good solutions and building the set up from there. Figure 3.3 shows one such resolution: The condition is negated logically and used to define a `mayPrint` predicate.

The `mayPrint` predicate works because under its base-case it collects all methods *m* that invoke a `PrintStream` method. Under the next iteration, it collects all methods *m* that call those methods, and so on, until a fixed-point is reached. Once the `mayPrint` predicate is solved, it is simply a matter of writing its negation (on lines 8–10). Here, there is a filter applied to bindings for *m*, but at the start of the solver, no bindings for *m* have been found. This is a special case, in which all possible bindings for *m* are considered (that is, all possible `Methods`). This special rule is intended for situations like the above, and only applies when there is a single variable under consideration. In order to get similar functionality with multiple variables an `exists` expression must

```
define mayPrint(Method m) {
  exists (Method n : invokes(m, n)) {
    hasSignature(<java.io.PrintStream>, n)
    || mayPrint(n)
  }
}

define doesNotPrint(Method m) {
  !mayPrint(m)
}
```

Figure 3.3: A correct specification for the `doesNotPrint` predicate.

be used, which makes the presence of the (often very large) cross-product explicit.

Because of the monotonicity of the solver there are some special cases to consider when the topological ordering of predicates to solve is computed. In particular, if a predicate depends on another predicate via a negated clause, there is the potential for incorrect results to be added to the solution set because the intermediate solution is incomplete. For example, suppose that `mayPrint` and `doesNotPrint` in Figure 3.3 are solved simultaneously: Then the `mayPrint` solution will be correct, but the `doesNotPrint` solution will end up with the set of all methods. The solution for this case is to consider `doesNotPrint` depending on a solved `mayPrint` set before being evaluated.[1]

This change of dependencies in terms of the order of when predicates are solved is natural due to the monotonicity of the solver: Clauses would have to be written very carefully so that no extra results are counted as part of the solution. Instead of reasoning about such cases, the solver linearizes these dependencies. In this way, the relations solved by the fixed-point solver are done so in the same order as a topological sort, with the mutually dependent relations treated as a single node in the topology. In addition to dependencies involving negation, the solver also linearizes dependencies intertwined with uses of the if and only if operator (‘<=>’) and the `exists` expression.

[1]That is, users can write “may analyses,” and thus a “must” analysis must be written as the negation of a “may not.”

3.2 Transformation Algorithm

The Arcum transformation algorithm takes code that implements a source option and translates it to new code that implements a destination option. The following process performs such a transformation:

1. Use the patterns specified in the source option to bind option-local concepts and abstract concepts. The pattern matching identifies the program fragments, represented as AST nodes, that participate in the refactoring.

2. Perform all option-specific and interface-level constraint checks, and stop with an error if any of the checks fail.

3. Remove from the program all AST nodes that were pattern matched into the source option's local concepts. For example, when refactoring from the `InternalField` option to the `AttributeConcept` option in Figure 2.5, the `altText` field gets removed from the program because its AST node was pattern matched into an option-local concept. AST nodes that match into option-local concept are removed during transformation because by design these concepts are option-specific—otherwise, the concepts would be abstract and specified at the interface-level.

4. Construct new AST nodes using the patterns from the destination option's local concepts and insert them into the program. In the refactoring scenario from Figure 2.5, the concept `mapField` in the `StaticMap` option will add a declaration of a static `Map` to the program.

5. Replace each concept instance with a new AST node generated from the destination option's patterns; construct the new AST node such that it satisfies the destination option's constraints (if present).

The main challenge in the above algorithm lies in processing patterns both to perform pattern matching (in step 1), and to generate new AST nodes (in steps 4 and 5). These two uses of patterns are described in Section 3.2.1 and Section 3.2.2, respectively.

3.2.1 Pattern Matching

Arcum patterns are represented using ASTs that can have variable nodes for sub-trees in addition to concrete AST nodes. A standard unification routine is used to perform the pattern matching. The concrete syntax of the program is canonicalized before the matching is performed, so that operations are closer to their semantic meaning. For instance, even though the pattern

```
setExpr == [`targetExpr.`field = `valExpr]
```

uses the "dot" notation, it will also match program fragments that use the implicit `this` (without the dot).

One feature of the pattern matcher is that the type of pattern in square brackets must be determined before AST node used for matching is parsed. In the above case, for example, the type of `setExpr` is an `Expr`, and thus the code in the brackets is parsed as a Java expression. For this reason, all uses of the unification operator ('==') must have an Arcum variable on the left-hand side. Thus, in order to unify two program fragment patterns, an intermediate variable (for example, one created from an `exists` clause) must be introduced.

3.2.2 Node Generation

One of the key features of Arcum is that patterns are bi-directional: not only are they used for matching Java code fragments into concept instances, but they are also used in the other direction, to generate Java code fragments from concept instances. As an example, the following pattern in the `StaticMap` option is used in the refactoring scenario (from Figure 2.5) to insert a call to the `put` method of the static map:

```
setExpr == [`targetType.`mapField.put(`targetExpr, `valExpr)]
```

A concept instance and a destination pattern generate an AST node by taking the partially specified AST representing the pattern, and inserting into it the values of the Arcum variables from the concept instance. Because Arcum variables store references to program fragments the above pattern creates an AST node representing the call to put, and the equality (==) makes the newly created AST at the location in the program specified by setExpr.

For some patterns, there are multiple possible AST nodes that could be generated. For example, the field pattern shown in Figure 2.5 shows two possibilities for the field-declaration, where either the transient modifier is present or not:

```
field == ([transient `spec `attrType `attrName]
          || [`spec `attrType `attrName])
&& hasField(targetType, field)
&& (isA(targetType, <java.io.Serializable>)
    <=> isTransient(field))
```

Its conjoined constraint specifies when the field should be transient: the field must be transient exactly when the targetType class implements the Serializable interface:

```
isA(targetType, <java.io.Serializable>) <=> isTransient(field)
```

Here, the <=> represents the logical if and only if operator. When there are multiple possible AST nodes that could be generated from a pattern or union of patterns, I use a generate-and-test approach: I generate all of the possible AST nodes, and then use the conjoined constraints to prune nodes out. This approach works well as long as there are a small number of possible AST nodes to generate. In the above example, Arcum would generate both field-declarations: one with the modifier and one without. The evaluation of the constraint determines which of the two AST nodes to use.

One special case is when the "don't care" variable ('_') appears in patterns: This is allowed for node generation cases only when the choice of the its value would not affect the construct's semantics. For example, the parameters of the methods in a Java interface must be named, even though the name does not affect the meaning of the method's signature. In such cases, if a reasonable default name can be generated, it will be allowed. Currently, the interface case is the only one allowed by Arcum.

The replacement algorithm uses a top-down ordering to replace nodes once they have been generated, to allow for sub-nodes of concepts to be replaced by other concepts. By using this top-down order, Arcum is able to correctly transform:

```
anImage.altText = defaultImage.altText;
```

into:

```
  altText.put(anImage, altText.get(defaultImage));
```

Chapter 3, in part, is a reprint of the material as it appears in Beyond refactoring: a framework for modular maintenance of crosscutting design idioms. 2007. Shonle, M., Griswold, W., and Lerner, S. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering* (Dubrovnik, Croatia, September 03–07, 2007). ESEC-FSE '07. ACM, New York, NY, 175–184. The dissertation author was the primary investigator and author of this paper.

Chapter 4

Case Study of Arcum

This chapter presents a case study of Arcum's use on primarily its own codebase, where I encountered a variety of classical software engineering problems that are induced by crosscutting concepts. This chapter demonstrates several plausible ways crosscutting manifests itself in real-life programs, and how such crosscutting can be mitigated. The aim of the case study is to demonstrate the frontier of possibility for an expert in the Arcum framework. (In contrast, the aim of the user study in Chapter 5 is to explore how well a non-expert can employ Arcum for intermediate-level tasks.)

Each category of software engineering problem covered has a working example developed and tested against the codebase. To provide context for these examples, Section 4.1 compares the use of Arcum to the use of Eclipse's refactoring capabilities and AspectJ's ability to advise crosscutting code. I start with the category of software engineering problems related to code migration and the trade-offs between making a program's implementation more standardized or more project specific (Section 4.2). Next I address the process of debugging as a programming task that requires reasoning over crosscutting code, which sometimes result in making crosscutting changes to fix the bug or to narrow down its root cause (Section 4.3). Then, I cover how needs specific to a particular domain differ from the general needs that a programming language covers (Section 4.4). Software architectural design enforcement follows domain-specific checks, but with a more general focus (Section 4.5). Finally, I describe in detail several variants of a design pattern, and how different implementations provide different views into the program (Section 4.6).

4.1 Comparisons to Eclipse and AspectJ

In this section, I consider three different change tasks that could plausibly be required in a large program, each with different degrees of crosscutting. A discussion of each change task follows. I also compare the ease of each change task with the state of the art in Eclipse and AspectJ [KHH$^+$01].

4.1.1 Task 1: Application of AttributeConcept

To get a preliminary feeling for whether Arcum imparts some of the benefits of modular substitution to crosscutting concepts, I put the `AttributeConcept` code (from Chapter 2) to work. I chose the Polyglot framework [NCM03] because it uses advanced design idioms which could potentially benefit from enhancement. Polyglot is an extensible compiler through its use of delegators and extension objects. To support such extension, the `Node` class in Polyglot has two fields: `del` and `ext`. A compiler extension to Polyglot uses these fields to extend the compiler's behavior. Based on their infrequent usage on a per-object basis, a sparse representation of these fields could conserve memory usage.

I first externalized the storage of the `del` field. I created an `InternalField` mapping for `del`, which subjected the code to checking, and the checks for conformance passed. I then changed the mapping to the `StaticMap` option, via a refactoring, which resulted in 13 substitutions. To measure the program's new memory usage, I compared its previous memory footprint from compiling an 80K line program to the modified version. In my measurements the `StaticMap` version actually required more memory than the `InternalField` version. One cause could be padding issues: removing one field might not generate gains in real space.

To see if this was the case, I externalized the `ext` attribute as well, resulting in 12 substitutions. After this refactoring was performed, the total memory usage was less than the original (unmodified) program. However, the gains in saved space were not large, for example saving 100 KB for a program run that used 11 MB total. Consequently, I changed the mapping to an `InternalField` for `del` and for `ext` to reverse both refactorings.

A lesson from this experience is that the consequences of some refactorings are difficult to anticipate, and the ability to quickly and safely try and undo refactorings is valuable. The equivalent steps necessary to perform the same changes in Eclipse (covered in Section 2.1) would not make this kind of exploration as easy. The experience of switching implementations by editing the mappings was similar to modular substitution, such as tweaking an object factory to return different object types under different conditions.

4.1.2 Task 2: Message Log Redirection

For a second test, I edited the source for the Arcum plug-in itself. The plug-in has a mode for sending debugging information to `System.out`. I considered a scenario where I wanted this output to be redirected to a different stream. This change can easily be accomplished by redirecting `System.out` itself. However, redirection is too blunt of a solution in Eclipse, because it redirects all console output, including from other plug-ins.

The solution requires that the my plug-in uses one stream, while any other plug-ins continue to use `System.out`. Eclipse can perform this change more thoroughly than a standard textual find and replace: Eclipse can locate all references to the field (which numbered just over 1,000) regardless of the whitespace formatting or scope issues (such as when static imports are used). However, despite Eclipse's semantic level search, all source code modifications still needed to be made using a textual find and replace. This global replace operation caught most instances, but repeating the semantic search revealed the rare syntactic exceptions missed by the search pattern.

AspectJ was better suited at performing the change than just Eclipse alone. AspectJ has a `get` pointcut similar to Arcum's field-access pattern: it can match all references to a specific field. The pointcut can also be narrowed down to match classes contained in a specific set of packages. A simple `around` advice applied to this pointcut can replace the value used for `System.out` in every location in the project. The AspectJ solution also had the advantage that the stream returned by the advice could be either the value of a stored stream or the result of a method call.

Finally, the same change was made using Arcum. A simple interface was written that had one abstract concept with only one member (of type `Expr`) in it. Then, an option was written to match this concept with all accesses to `System.out`. To test the option, I used Arcum's search view to display all program fragments that belonged to the concept. Once I was convinced the results were as expected, I wrote an alternative option that accessed a different field instead. Similar to the AspectJ solution, it was a simple matter to change the stream used to the result of a method call instead, just by adding yet another option.

One advantage of the Arcum solution over the AspectJ solution is that it refactors the program itself, as opposed to only modifying the program's semantics through bytecode weaving. As a result, design decisions are more visible and can better reflect the nature of the program. The prior disadvantages of such crosscutting code are no longer disadvantages with Arcum: for example, even though the calls to a helper method may crosscut the program, this crosscutting is not a liability because the code can easily be changed back should the need arise.

4.1.3 Task 3: Remove Control Coupling

For another change task, I considered refactoring calls to a method that took on the duty of two different operations: the operation to perform was determined based on whether or not one of the arguments was `null`. This argument essentially became a flag, and a usage pattern emerged as a result where calls to this method would pass `null` into this argument. Such a usage pattern increases the coupling between modules and it would be better to refactor the source code so that these calls would invoke a separate method that just performs the expected operation. My goal was to perform this refactoring using Arcum.

The Eclipse 'Change Signature' feature is useful for introducing a new argument or removing one, but it's not suited for the task of changing some but not all method calls. In Arcum I was able to match all special `null` argument cases and change them into calls to the new method instead. One limitation with the current Arcum approach (and similarly for AspectJ) is that a dataflow analysis is not performed. For example, instead of syntactically matching when `null` is an argument it would be more flexible to categorize the argument into one of three sets: (1) The set of calls where the argument is known to be `null`; (2) The set of calls where the argument is known to be non-`null`; or (3) The set of calls where the argument may or may not be `null` (for example, when it can only be determined at runtime).

4.2 Class Library Migration

Class library migration is a general software evolution need: Often it is desirable to remove the use of a legacy library and directly use its replacement instead [BTF05]. The solution for this problem devised by Balaban et al. includes a type constraint solver that finds the largest set of code locations that can be safely changed, even in the presence of synchronized methods. For example, uses of the always synchronized `Vector` class can be changed to uses of the more efficient `ArrayList` class, where the `ArrayList` instances are synchronized only when necessary.

Arcum supports a complementary variation of class library migration. Instead of determining the largest set of code locations that can be safely migrated, Arcum's approach is to have the location set explicitly described; for example, one description might be "all uses of `Vector` in package `p` or by class `C`." When the set cannot be transformed, the starting option presents the user with static error messages. All errors have to be resolved before transformation is allowed. In some instances, the location set specified might match more code than expected, requiring the user to narrow the set's definitions; in other instances, the set definition is correct but modifications need to be made to the code itself first, to bring it into conformance; in yet other cases, the user might realize, by the nature of the errors, that he or she needs to take a completely different approach. In this way, Arcum gives the programmer an opportunity to interact with the tool, helping to ensure that his or her conception of the system matches the actual implementation.

The key to Arcum's support for matching uses of class libraries is the `DeclarationElement` type, derived from the same term used by Tip et al. to describe all local variables, fields, return types, and cast expressions [TKB03]. The pattern used to specify `DeclarationElements` in the program is a type pattern. Figure 4.1 shows a pattern to match all declarations of the `Integer` wrapper class.

```
interface IntegerBoxingOperation {
  abstract boxing(Expr expr, Expr boxedValue) {
    exists (DeclarationElement d) {
      d == [Integer] && copiedTo(expr, d)
    }
    && isA(boxedValue, <int>)
  }
}

option ImplicitBoxing implements IntegerBoxingOperation {
  match boxing(Expr expr, Expr boxedValue) {
    expr == [`boxedValue]
  }
}

option ExplicitBoxing implements IntegerBoxingOperation {
  match boxing(Expr expr, Expr boxedValue) {
    expr == [new Integer(`boxedValue)]
  }
}
```

Figure 4.1: The `ImplicitBoxing` option matches all `int` expressions that get implicitly boxed into an instance of `Integer`. When this option is transformed into the `ExplicitBoxing` operation the `Integer` constructor is explicitly called.

In addition to matching all `DeclarationElements` in the program, support for migration must allow for the description of operations such as: class instantiations, method invocations, and conversion operations. Instantiations and method invocations can be matched using various `Expr` patterns. To match conversion operations, Arcum defines a `copiedTo` relation in the database: `copiedTo` relates expressions to declaration elements, and the relation holds when the value of the expression is copied to a location declared by the declaration element. Copy operations include: assignment, initialization, argument passing, and value returning.

The code in Figure 4.1 demonstrates how implicit boxing can be made explicit, which could be used as the first step to replacing uses of the `java.lang.Integer` class with an alternative class.

4.2.1 Canonicalization

One use I found for class library migration was when I started to employ the Google Collections library [Goo07]. The library includes extra support for programming with generics in Java, including interfaces and operations to support functional programming styles.

For example, I had a need in Arcum to support operations that are lazily executed. This division of definition and execution allowed me to separate the knowledge of how to initialize an object from the knowledge of when to initialize it. I initially defined a parameterized `Thunk<T>` interface to achieve this, which declared a single, no-argument method with a return type of `T`. Looking into the Collections library, I found that the `Supplier` interface fit my needs exactly. By using an interface defined in another library, I was able to make my use of the interface less mysterious compared to the original solution. By writing code that conforms to a more standardized interface there is a better chance to integrate independently developed code that followed the same standards.

4.2.2 Removing Puns: De-Canonicalization

There are always trade-offs with using standard libraries. One risk of transforming similar looking code to all use the same canonical form is that two uses that only accidentally look similar could be mistaken to belong to the same concern. Such similar uses could be called “puns.” For example, if there are two methods that accept an instance of `Supplier<String>`, are they related? Or, would it be better for one to be named `DelayedObject<String>` with the other named `DefaultValue<String>`? The answer depends on your circumstances.

Providing different class definitions to prevent puns can assist modularization. For example, while the current needs of the interface might result in the same structure, a programmer may anticipate additional operations that will need to be added later; and the additional operations will only make sense in one context but not the other. Similarly, the semantics of the operations might shift over time. For example, `DelayedObject<String>` might have caching semantics, while

`DefaultValue<String>` would be better served re-computing the result for each request.

One middle ground that can also be employed with Arcum is to extend the richness of types via annotations. For example, two structurally similar, but conceptually different, uses of the same interface could be separated by applying different Java metadata annotations to them, creating a form of qualified types. (However, as of Java 6, type arguments for parameterized types cannot have annotations. Thus, the solution is not complete, although some work-arounds exist.) Under this strategy, it could be up to the option author to decide if assignments are allowed between variables belonging to the same type but with different annotations. Or, alternatively, to allow conversion, but only when explicitly exposed through a static method.

Key to the Arcum style is that programming decisions like this do not have to be made immediately: There is always the freedom to change your mind later, when your needs are clearer. Using Arcum, a "best guess" for a design decision can be made, with that decision documented as an option, to be revisited as needed.

4.3 Debuggability

The considerations made while debugging a program are different than the considerations during the design and implementation processes. For instance, while a well-designed program is modularized based on the criteria of what is likely to change [Par72], there are an infinite number of design futures—an attempt to anticipate them all would get no where. This lack of anticipation is pronounced when it comes to software bugs: The kind of bugs that trouble programmers weren't known to them when they first designed the system. Thus, in the process of debugging, a programmer may need to make changes across the decomposition of the system, including changes made to help find the cause of the bug, and then to fix it.

At one stage in the development of Arcum, I encountered an intermittent bug: Sometimes the program would halt with a `NullPointerException` and sometimes it would compute the expected result. Eventually, I discovered that the source of the problem was the iteration order of `HashMap`s. The `hashCode` used was the default identity code. On the VM I was using, this identity code was related to the bookkeeping records of the garbage collector. The location in memory where objects were initially allocated was important in determining this identity code and thus objects would be hashed to different sections of the hash table (and, hence, be iterated in a different order) on various executions of the program.

After the cause of the non-determinism was found, I wanted to see how I could use Arcum to help. One solution is related to the class library migration problem (see Section 4.2): The program can be refactored from using the `HashMap` class to using the `LinkedHashMap` class instead. `LinkedHashMap` is a sub-class of `HashMap` that has a predictable iteration order; it maintains a parallel linked-list to keep track of the order in which entries are added to the table.

By changing the program to use a deterministic order, I was able to reliably reproduce the bug, making it easier to locate the root cause of the problem and then to fix it. Part of this was luck, because the iteration order just happened to execute the operations in the order necessary to reproduce the bug. If I wasn't quite so lucky, I'd still have some options available: I could have added more test cases, in the hopes of finding the right code sequence to expose the bug again. Alternatively, I could have written a variant of the `LinkedHashMap` class that placed the elements in an arbitrary, but predictable, ordering based on a hard-coded seed.

The examples of making a program more deterministic are a special case of a more general problem: Oftentimes what was assumed to be stable during development time might need to be changed to assist debugging. It is not surprising that a programmer would not decompose a program to easily detect a bug, because, by definition, he or she did not see the bug coming. By using Arcum, defensive techniques can be employed, even those previously considered impractical (such as making every `HashMap` a `Linked-HashMap`), because there is always the option to change the code back again.

4.4 User-Defined Semantics

Types are sometimes used by a program in ways that need to be more restrictive than the type system itself requires (see also Section 4.2.2). This section covers additional examples where the user can benefit from extra checking and constraints.

4.4.1 When Simple Solutions are too General

Solutions using standard methods can be too general when only special cases of those methods are required. A balance must be made between using a library in an idiomatic style and making the intentions of the code clear.

For example, when working on `String`s in Arcum, I discovered multiple uses of the pattern `t.contains(".")`. Here, names of elements in the program analyzed were represented as `String`s, and the `contains` test was used to determine if the name was a qualified name. An alternative to this idiom is to direct all such tests to a static method instead: `isQualifiedName(t)`. Using Arcum, I was able to find all references to the special use of `contains` and transform them to use the static method. I then had a named entity that Eclipse could use to build a list of entity references. I reviewed this list to determine if any of those uses of `contains` were puns: That is, checks for the presence of dots that had nothing to do with qualified names—one reasonable case would be when the `String`s represented numbers, and the check would be better written as `isFloatingPoint(t)`.

One limitation of the transformation technique is that it could not detect typos. For example, a use might inadvertently have the `String` literal `".."`. One way to help find these cases would be to employ a type qualifier strategy [GF07], where `String`s that represent Java element names are marked with an annotation. Such a strategy could be useful as an intermediate step toward modularizing that use of `String` so that it's encapsulated in a wrapper class.

4.4.2 Checking Uses of Reflection

Reflection in Java is powerful but needs to be employed carefully. Once reflection is used, opportunities for static checking by the Java compiler are missed, even when only a subset of Java's reflection capabilities are necessary.

One area where reflection was employed in Arcum was in accessing a static method that was defined for each concrete implementation of Eclipse's `ASTNode` class.

```
public @ClassInterface interface PropertyDescriptorsMethod {
  List propertyDescriptors(int apiLevel);
}

public static StructuralPropertyDescriptor[]
getProperties(@ClassDefines(PropertyDescriptorsMethod.class) ASTNode n) {
  /* ... */
  proxy = ClassProxy.make(n.getClass(), PropertyDescriptorsMethod.class);
  list = proxy.propertyDescriptors(AST.JLS3);
}
```

Figure 4.2: An example restricted use of reflection: Instances of `ASTNode` are passed to `getProperties`, but each concrete implementation of the `ASTNode` class must have a static `propertyDescriptors` method defined with the same signature. Such a restriction can be checked by Arcum with a description of the `@ClassDefines` annotation's intended use. A proxy is employed together with the interface to make invocation more convenient.

Had this method been declared as non-static, I could have just made a simple call to it. Instead, I needed a mechanism to invoke a different static method depending on the type of the instance. By invoking `getClass` on the instance, I reflectively dispatched to the right method. This particular use of reflection had to make assumptions about the presence of the method. But such assumptions can be error prone and leave essential parts about the program's structure obscured.

However, by using annotations together with predicate checks in Arcum, I was able to use reflection in a more disciplined manner. Figure 4.2 demonstrates the use of a technique for making the requirements explicit in the code and allowing Arcum to check it. The `ClassDefines` annotation takes in a single value, a type token that references an `interface` that declares exactly one method; in this case, it declares the `propertyDescriptors` method.

Several properties are checked when the `ClassDefines` annotation is used. If any of the properties don't hold, then Arcum generates an error at compile time. Here are some example properties: (1) The annotation's argument must be an `interface` token that describes exactly one method; (2) All concrete sub-classes of the annotated type must implement a static method with the same signature; (3) All arguments passed to the `invokeStatic` method must match the number and type of the parameters specified

```
Function<FormalParameter, String> getIdentifier =
  Accessor.makeFunction(FormalParameter.class,
                        String.class,
                        "getIdentifier");
```

Figure 4.3: An accessor method, `getIdentifier`, exported as a `Function` object. This accessor can be used for functional style programming, such as transforming a list of `FormalParameter` instances into a list of `String`s.

```
static final Function<FormalParameter, String> getIdentifier =
  new Function<FormalParameter, String>() {
    public String apply(FormalParameter formal) {
      return formal.getIdentifier();
    }
  };
```

Figure 4.4: A static solution for Figure 4.3 that doesn't require reflection.

in the method. Checking all of these properties together brings back static type checking to this use of reflection. Errors that otherwise would only have been available during testing are made clear during development.

One assumption of reflection checking is that the program under analysis fits the *closed world model*. That is, during development time, the tool has access to all of the source code that is relevant to the use of reflection. Exceptional cases where the reflective calls must be unguarded can be marked, so that Arcum does not identify them as errors. (The presence of such markings can also serve as valuable documentation.)

Reflection as a Shortcut

Sometimes reflection is useful to employ in situations where it's a short cut for equivalent, but more verbose, static techniques. For example, Java lacks support for function pointers, but a workaround can be achieved using interfaces and reflection or anonymous inner classes. Figure 4.3 shows a method `makeFunction` that takes in a type and a method name and returns a `Function` object. When uses of this idiom are encoded as an option, the runtime typing checks can also be made at compile time. For example, if the method name was misspelled, was not visible, or if the return type was improperly specified, the user would see a static error message from Arcum.

Checking uses of reflection makes reflective techniques more practical. Arcum also encourages use of these reflective techniques by not forcing the developer to *commit* to them. At any time in the process, the user can automatically refactor to the static form; for example, having a static field declared in the class that performs the access instead, as shown in Figure 4.4.

4.4.3 Detecting Library-Specific Errors

Some constraints that apply to Java language constructs cannot be applied to abstractions meant to replace them. For example, the result of the '`+`' operator must be used in an assignment or argument (e.g., '`a+b`' is not a valid statement, but '`x=a+b`' is).

The gap between the Java language constructs and library abstractions is that methods have no way to specify that their return value must be used. Such a requirement is common for methods belonging to immutable classes, like `BigInteger`. In the implementation of Arcum, I use an immutable set construct; several times, I encountered a bug where results were inadvertently discarded, such as when I wrote:

```
result.union(sat)
```

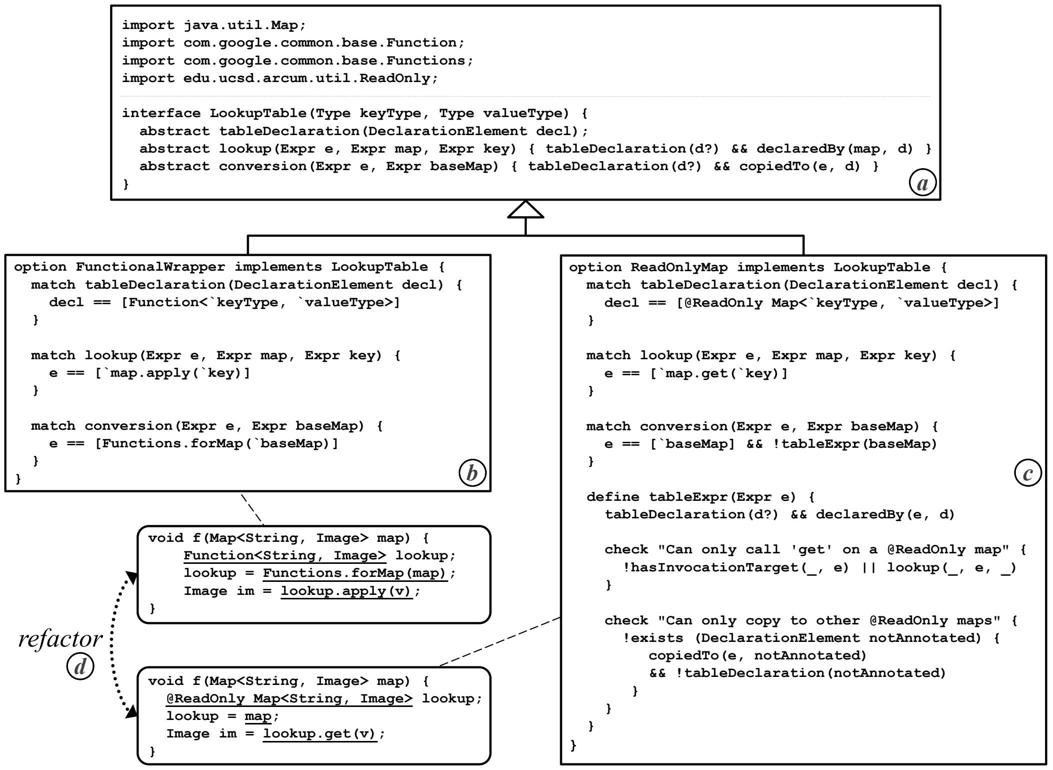
instead of:

```
result = result.union(sat);
```

By checking calls to the `union` method, I was able to prevent future bugs.

Another group of methods that benefit from extra checking are methods that do not return. For example, methods that raise an exception or call `exit` do not return. When a method is marked with a `@DoesNotReturn` annotation, an Arcum option can find all calls to the method and ensure that the next statement after the call is either a `return` or a `throw` statement. This way, the Java compiler will prevent code from following it, because such code would be considered unreachable. For example, the `fatalError` method called below could be marked as not returning:

```
fatalError("A fatal error has occurred");
throw new Unreachable();
```



```
import java.util.Map;
import com.google.common.base.Function;
import com.google.common.base.Functions;
import edu.ucsd.arcum.util.ReadOnly;

interface LookupTable(Type keyType, Type valueType) {
  abstract tableDeclaration(DeclarationElement decl);
  abstract lookup(Expr e, Expr map, Expr key) { tableDeclaration(d?) && declaredBy(map, d) }
  abstract conversion(Expr e, Expr baseMap) { tableDeclaration(d?) && copiedTo(e, d) }
}
```

```
option FunctionalWrapper implements LookupTable {
  match tableDeclaration(DeclarationElement decl) {
    decl == [Function<`keyType, `valueType>]
  }

  match lookup(Expr e, Expr map, Expr key) {
    e == [`map.apply(`key)]
  }

  match conversion(Expr e, Expr baseMap) {
    e == [Functions.forMap(`baseMap)]
  }
}
```

```
option ReadOnlyMap implements LookupTable {
  match tableDeclaration(DeclarationElement decl) {
    decl == [@ReadOnly Map<`keyType, `valueType>]
  }

  match lookup(Expr e, Expr map, Expr key) {
    e == [`map.get(`key)]
  }

  match conversion(Expr e, Expr baseMap) {
    e == [`baseMap] && !tableExpr(baseMap)
  }

  define tableExpr(Expr e) {
    tableDeclaration(d?) && declaredBy(e, d)

    check "Can only call 'get' on a @ReadOnly map" {
      !hasInvocationTarget(_, e) || lookup(_, e, _)
    }

    check "Can only copy to other @ReadOnly maps" {
      !exists (DeclarationElement notAnnotated) {
         copiedTo(e, notAnnotated)
         && !tableDeclaration(notAnnotated)
       }
    }
  }
}
```

```
void f(Map<String, Image> map) {
    Function<String, Image> lookup;
    lookup = Functions.forMap(map);
    Image im = lookup.apply(v);
}
```

```
void f(Map<String, Image> map) {
  @ReadOnly Map<String, Image> lookup;
  lookup = map;
  Image im = lookup.get(v);
}
```

Figure 4.5: Two descriptions of a lookup table: The Arcum interface for a lookup table (see a) is implemented as a `Function` object that associates a range to a domain (b); or as a `Map` annotated as read-only (c). The sample code (d) shows these instantiations of `LookupTable` (with `keyType` bound to `String`, and `valueType` bound to `Image`).

I detected such a problem while debugging the Arcum project: I wondered why I couldn't see some debugging output, and then I realized that my debugging code came after a method that didn't return. The compiler accepted the code, even though in practice that code was unreachable.

4.4.4 Java Metadata Annotations

The `@DoesNotReturn` annotation from the previous section is an example of how Java metadata annotations can be used as a means to document and keep track of design decisions that would otherwise not be directly represented in the source of a program.

For example, consider a system where an association is made between members of a set of objects and a set of their definitions, such as in the implementation of a symbol table. The `HashMap` class is sufficient for expressing such a relationship. Whether the `HashMap` class or `Map` interface is used in parameter specifications is one design decision that the programmer has to make. In this case, it is recommended that the `Map` interface be used, because it allows for more flexibility (this recommendation appears as items 40 and 52 of *Effective Java* [Blo08]).

However, even the `Map` interface itself might be broader than necessary. For example, methods taking instances of the symbol table as parameters might only need to perform lookup operations, and not modify the table. Thus, at the conceptual level, the programmer's requirements are for a lookup table, and the `Map` interface is just one possible implementation of such a mechanism.

Figure 4.5 shows an Arcum interface and two options to explicitly document the requirements of a lookup table. The definition specifies how a `LookupTable` is parameterized and describes three abstract concepts:

1. The `LookupTable` has two parameters, both `Type`s: the key type and the value type.

2. The first concept, `tableDeclaration`, defines when a declaration element belongs to a `LookupTable` implementation.

3. The second concept, `lookup`, defines when an expression is considered a lookup operation. A lookup operation must have both a `map` expression and a `key` expression. The concept is partially defined, restricting as valid `map` expressions only those whose type has been specified by a `tableDeclaration`. The rest of lookup's definition must be provided by the implementing option.

4. The third concept, `conversion`, defines when a regular `Map` is converted to an expression that yields a lookup table. This concept is also partially defined, restricting valid `conversion` operations to only those where the new lookup table is stored into a location specified as a `tableDeclaration`.

Both the `lookup` and `conversion` concepts reference the `tableDeclaration` concept, using it as a predicate. The `lookup` concept uses the built-in `declaredBy` predicate, which relates expressions to the declaration of its static type.

The `LookupTable` interface shown in Figure 4.5 has two Arcum options that implement it: `FunctionalWrapper` (b) and `ReadOnlyMap` (c). The `FunctionalWrapper` implementation of `LookupTable` utilizes the `Function` interface, which defines a single method named `apply`. A `Function` can be created from a `Map` via the static `forMap` method. The advantage of using this `Function` implementation is that it requires the implementation of only the `lookup` operation (here, `apply`).

The `ReadOnlyMap` implementation of `LookupTable` uses the standard `Map` interface, but with a twist: All declarations of lookup tables are labeled with the `@ReadOnly` annotation. This annotation lets programmers know that only the `get` operation will be accessed. Such annotations can be more than just documentation with the help of additional checks: The `tableExpr` concept defined in Figure 4.5, box c, does so with two `check` clauses.

The area marked "d" in Figure 4.5 shows two example code snippets that can be automatically transformed to and from each other with Arcum. Although the example presented in Figure 4.5 is complete, it can be extended to include further checking. For example, checks can be made to ensure that any type used for the `keyType` overrides the `equals` method whenever it also overrides the `hashCode` method. Such checks can prevent coding errors and help ensure bi-directionality.

4.5 Software Architectural Design Enforcement

Section 4.4.3 covers domain-specific checking of concepts at the micro scale. This section covers domain-specific checks for larger scales, in particular, for access control (Section 4.5.1) and programming style (Section 4.5.2).

4.5.1 Finer-Grained Access Control

Encapsulation allows for the detection of violations of the *knows-about* relation. For example, methods encapsulate their local variables, and thus external methods cannot access them; classes encapsulate their private fields, and thus external classes cannot access them. The knows-about relation is important, because knowing even about the presence of a separate entity creates a liability: When that entity is subject to change, so too are all elements that know about it.

One example of controlling what software components need to be aware of is the Façade pattern [GHJV95]: The Façade pattern can reduce the level of coupling between components and assist layering. I used the Façade pattern in Arcum when interfacing with Eclipse's Java compiler. I utilized Eclipse's type checker to resolve variable bindings, but my syntax desugaring mechanism made the process of finding the bindings more complex. The solution was to write a Façade that was a single point of interaction with Eclipse's resolver. Using Arcum for this solution helped in two ways: First, I was able to refactor each call to Eclipse's resolver to be a call to the Façade instead. Second, I wrote new checks that prevented direct calls from being inadvertently made. This extra checking ensured that the Façade pattern held and that layering was preserved.

On the smaller scale, I also utilized intra-class layering in my implementations. For example, I found that even the `private` access specifier was not strict enough for my needs when it came to reasoning about classes. In one case, I had two related fields in a class to which I only wanted the constructors and two tightly-coupled accessor methods to have direct access. I labeled these fields with an annotation that specified the group of methods that were allowed both read and write access to these fields. Using this annotation as a guide, extra checks were able to ensure that only the methods defined in the group had access.

The nature of the method group solution can apply to inter-class layering as well: A family of methods cutting across several classes might be accessible to each other, but inaccessible to other methods, even those methods that are defined in the same scope. Such a solution is similar to `friend`s in C++, with the added ability to enable or disable read or read/write access.

4.5.2 Detecting Common Errors

There is a class of general programming errors that lend themselves well to automatic detection [HP04, RAF04], several of which can be checked with Arcum. I encountered one bug when I executed code that raised a particular `RuntimeException`, yet my exception handlers were not catching the exception.

The problem was related to how I *softened* checked exceptions. Checked exceptions can sometimes violate layering principles in code because they force `throws` declarations on methods that neither know how to detect the exception nor know how to handle it. Thus, at times I would soften a checked exception type by wrapping it in a `RuntimeException` and throwing it. That `RuntimeException` could then be unwrapped later, at the level that is able to address the error. The bug was that I softened all exceptions, not just the checked ones, so the specific `RuntimeException` subtypes were replaced by the generic `RuntimeException` type.

Given such dangers of using exception softening, I added a check to find all cases of exception softening and made sure that `RuntimeException`s were not included:

```
catch (RuntimeException e) {
    throw e;
}
catch (Exception e) {
    // soften only non-runtime exceptions
    throw new RuntimeException(e);
}
```

Such a check can be made syntactically by making sure that exception softening always fits the format shown above.

4.6 Extended Example: The Visitor Pattern

This section uses the Visitor Pattern [GHJV95] to show how Arcum can be used to improve the ability to comprehend and evolve crosscutting concepts. This particular pattern was chosen for this extended example because it has many interesting variants, including a code transformation that is many-many instead of 1-1.

The visitor pattern is a depth-first traversal over a heterogeneous collection of objects. One popular example of the visitor pattern is a language interpreter with an abstract syntax tree that is enhanced with a type-checking operation to be performed over AST nodes. Even this canonical use of the visitor pattern presents difficulties with respect to comprehension and evolution. In particular, it is hard to detect bugs in the visitor pattern implementation and it is difficult to perform crosscutting changes.

Conceptually, the visitor pattern can be understood as an operation involving two graphs: a *class graph* and a *traversal graph* [Lie96]. A class graph abstracts the has-a relationships of a group of classes: Classes are represented as graph nodes, and fields are the edges that connect nodes. For the purposes of exposition, I consider class graphs formed from a single root class. Figure 4.6 defines a `classGraph` relation, where `rootType` is an externally specified `Type`.

The traversal graph is a subgraph of the class graph that is defined in terms of a *traversal strategy*. The traversal strategy is specified by a set of *target types* and a set of edges (fields) to bypass. Starting from the root of the class graph, an edge is included in the traversal graph only when (1) taking the edge will eventually lead to one of the target types, and (2) the edge is not in the bypass set. Example of edges to bypass include fields representing cached computations, and fields that are merely back-links to parent nodes. The relation in Figure 4.6 defines a `traversalGraph` relation, where `targetType` and `bypassEdge` are externally specified predicates.

The definition of `traversalGraph` can be viewed as starting at the bottom of the class graph and moving up: Initially, the only valid bindings for the `to` variable are those from the `targetType` relation. So, the first time the equation is solved the only members of the relation are those nodes from the `classGraph` that connect directly to one of the target types. The next iteration of the constraint solver is then able to

```
define classGraph(Type from, Type to, Field edge) {
  (from == rootType || classGraph(_, from, _))
  && hasA(from, to, edge)
}

define traversalGraph(Type from, Type to, Field edge) {
  classGraph(from, to, edge)
  && !bypassEdge(edge)
  && (targetType(to) || traversalGraph(to, _, _))
}
```

Figure 4.6: Two recursively defined relations: `classGraph`, stated in terms of the `hasA` relation and itself; and `traversalGraph`, stated in terms of the `classGraph` relation and itself. The `hasA` relation used here is a three-tuple form that exposes the `Field` that makes the relation hold.

use these new bindings from the recursive clause of the equation, adding to the set all nodes that connect directly to a node that connects directly to a target type. The process continues until no additional edges can be considered as traversal edges, at which point a fixed-point is found.

The purpose of the traversal graph is to work in terms closer to the conceptual problem: *What are the classes visited?* As the program evolves, the class graph and the traversal graph may change while the set of desired `targetType`s is more stable. When coding is shifted to focus efforts on the stable definition, the development process is more adaptive to change [Lie96]. The implications of this shift in focus in terms of bug detection is discussed in Section 4.6.1, while a more dynamic approach altogether is discussed in Section 4.6.3.

4.6.1 Visitor Arcum Option Implementation

A general purpose `VisitorConcept` can be parameterized by the variables that occurred free in the code from Figure 4.6: the `rootType` variable, the `targetType` relation, and the `bypassEdge` relation. The `VisitorConcept` can additionally be parameterized by: a Java interface that all visitor objects must implement, and a name for the traversal. Figure 4.7 declares these parameters on lines 2–7.

Lines 12–17 of Figure 4.7 examines each type given in the `targetType` relation and checks to see if the given interface has a corresponding `visit` method defined for that type. When such a method signature is not present, the user is given the error message on line 15, which indicates what was expected.

This visitor interface check is useful when the members of the `targetType` set changes: For example, a user could parameterize the use of the visitor pattern with a `targetType` set defined in terms of all subclasses of an abstract class. When a new subclass is created, either the corresponding visitors should change to accommodate this new type, or the expression defining the `targetType` set should be revised to exclude the case. In both cases, the introduction of the new subclass is an important event that requires modification to either the program or the supplemental description of the program's implementation. The implication can be checked in the other direction as well: If a visit method signature is present in the interface, then the programmer probably

```
01 interface VisitorConcept(
02   Name traversalName,
03   Type visitorInterface : isInterface(visitorInterface),
04   Type rootType : isClass(rootType),
05   targetType(Type type),
06   viaEdge(Field edge) default isField(edge),
07   bypassEdge(Field edge) default false)
08 {
09   abstract visit(Expr root, Expr target, Expr visitor);
10   define classGraph(Type from, Type to, Field edge) { /* ... */ }
11   define traversalGraph(Type from, Type to, Field edge) { /* ... */ }
12   check {
13     forall (Type t : targetType(t)) {
14       hasSignature(visitorInterface, <public boolean visit(`t `_)>)
15       onfail {"Missing visit method of type `t", visitorInterface}
16     }
17   }
18 }
```

Figure 4.7: Code listing for a Visitor Pattern Arcum interface.

intended it to be included in the `targetType` set.

Line 9 of Figure 4.7 specifies an abstract concept that captures all expressions (`root`) that represent invocations that start the traversal. Each traversal is applied to some base object (`target`), and is given a visitor instance (`visitor`). Lines 10–11 of Figure 4.7 are placeholders for the complete definitions of the `classGraph` and `traversalGraph` relations, specified in Figure 4.6.

One possible implementation of the `VisitorConcept` is the traditional "Gang of Four" implementation [GHJV95]. Figure 4.8 shows `GoFVisitor`, an `option` that implements this interface. Details related to collections and Java interfaces are elided for expository purposes.

The `visit` concept is matched starting on line 2 of Figure 4.8. Line 3 binds the visit expression to a call to the traversal method, and line 4 restricts the matched program fragments to only those traversal calls related to the given `rootType`. Line 5 is an additional filter: Code that implements the visitor pattern infrastructure itself should not count as visit operations.

The `acceptMethod` definition starting on line 8 of Figure 4.8 describes a correct

```
01 option GoFVisitor implements VisitorConcept {
02   match visit(Expr root, Expr target, Expr visitor) {
03     root == [`target.`traversalName(`visitor)]
04     && isA(rootType, target)
05     && !exists (Method m : acceptMethod(m)) { within(root, m) }
06   }
07
08   match acceptMethod(Method m) {
09     traversalGraph(c?, _, _) && isClass(c)
10     && hasMethod(c, m)
11     && m == [public void `traversalName(`visitorInterface v) {
12               `[Statement stmt : acceptMethodStmt(c, stmt)]
13              }]
14   }
15
16   define acceptMethodStmt(Class owner, Statement stmt) {
17     traversalGraph(owner, _, edge?)
18     && stmt == select {
19          targetType(target?) && declaredAs(edge, target):
20            <visitor.visit(this.`edge);>,
21          default:
22            <this.`edge.`traversalName(visitor);>
23        }
24   }
25 }
```

Figure 4.8: Code listing for `GoFVisitor`, the traditional Gang of Four implementation of the `VisitorConcept` interface.

implementation of the visitor pattern method infrastructure. It serves as a guide to identify missing statements or methods when the class graph changes. For example, there are situations where a programmer may add a new field to a class, creating another node that should be visited. The addition of a new field can require global changes when it enables a new subgraph of the class graph to be reachable. I found this check to be helpful in driving the process of implementing new visitors and for finding bugs in existing code. In the source control history of the Arcum project one bug recurred multiple times: certain sub-expressions were not being type-checked, because the visitor infrastructure left out those cases.

Line 9 of Figure 4.8 takes a projection of the `traversalGraph`, taking only types that are classes (as opposed to interfaces). Lines 10–13 ensure that each of these classes has the appropriately defined `accept` methods. Line 12 makes use of `acceptMethod-Stmt`, a helper relation defined on lines 16–24. This helper relation relates a class with a series of statements that the class's `accept` method should contain. (Not shown in this figure is the definition for `acceptSignature`, which is similar to `acceptMethod` but applied to Java interfaces instead.)

4.6.2 Variant: Cycle Risks

The implementation presented so far is fairly simple, but this simplicity burdens the visitor instances themselves. The burden is that each visitor object is responsible for cycle detection. When the object graph is cycle free—as found with tree structures like ASTs—the visitor object can be oblivious of this concern. But what options are available when the program evolves and cycles are possible? A cycle-aware visitor implementation isn't conceptually different from the existing `option` implementation. Thus, an alternative `CycleAwareGoFVisitor` option can implement the `VisitorConcept` and maintain a (thread local) `Set` of objects that keep track of what has already been visited, preventing instances from being visited more than once.

Given this new definition, an alternative `option` would be available, allowing the programmer to refactor between the two implementations. The visitors already written for the cycle-free case could then be used as-written without risk of infinite loops.

An alternative solution to the introduction of cycles would be to directly modify the classes that define the visitor objects themselves, so that each visitor is in charge of marking the visit history. Such an implementation change can also be guided with the help of Arcum by make the visitor object implementations the target of refactorings instead.

4.6.3 Variant: Reflection using the DJ Library

The complexity of the visitor pattern leads some developers to chose to avoid it altogether [MW06, p. 338]. The benefits from using the visitor pattern should outweigh its costs. For example, authors of an API can benefit greatly by allowing a mechanism for external clients to write extensions without modifying the codebase. But in an agile

```
option DJLibrary implements VisitorConcept {
  match Field strategy, Expr init {
    init == [Strategy.create(new TypeLiteral<`rootType>() {})
        .targets(`anyOrder:[Expr e: targetType(t?)
                                    && e == <new TypeLiteral<`t>() {}>])]
    && strategy == [public static Strategy `traversalName = `init]
    && hasField(rootType, strategy)
  } onfail {"Must have a static field named `traversalName", rootType}

  match visit(Expr root, Expr target, Expr visitor) {
    root == [`rootType.`strategy.traverse(`target, `visitor)]
  }
}
```

Figure 4.9: Code listing for `DJVisitor`, a dynamic implementation of the `VisitorConcept` interface.

or rapid prototyping context, it might be better to consider alternatives to the visitor pattern first.

Yet, by its general nature, the traversal operation itself is a valuable algorithmic tool; the kind of tool that should be available to use out of the box. Fortunately, there is an alternative implementation of the visitor pattern that allows for just that: The DJ library uses reflection in Java to offer a dynamic implementation of the visitor pattern [OL01]. The programmer describes the set of target types and the set of edges to bypass and then the DJ library performs the depth-first traversal on the object graph.

The DJ library offers multiple benefits: Because DJ reduces the amount of code that needs to be written, the program's class structure is easier to change. This reduction in code makes it easier for alternative designs to be explored, increasing the software's quality. Additionally, the bugs discussed in Section 4.6.1 do not occur since the traversal logic is contained in the DJ library.

Figure 4.9 shows an Arcum option that implements the `VisitorConcept` by using the DJ library. The code listing comprises the entire Arcum code necessary to describe the implementation. The use of the `TypeLiteral` class instead of the simpler `ClassName.class` notation for creating type literals is due to a technical limitation with Java's generics; the workaround was devised by Gafter [Gaf06].

The downside to the DJ implementation is its extra performance costs: Reflective operations can be expensive and are harder for VMs to optimize. But, due to the fluid nature of Arcum refactorings, programmers can get the best of both worlds. When performance is critical, the traditional implementation can be used. Then, when changes need to be made to the class structure, the implementation can be refactored to use the DJ library instead. After the changes are made, the program can be tested and then refactored back to the tradition implementation. In this way, the view of the program presented to the user is the one best suited for his or her needs.

4.6.4 Refactoring Special Cases

The alternative implementations presented in Section 4.6.1 and Section 4.6.3 differ in one dimension: static versus dynamic. In addition to cycle risks (Section 4.6.2), there are other implementation considerations. Sometimes the visitor pattern is used even when its full power is not needed. For example, if the `targetTypes` set contains only a single type, then the iterator pattern might be more suitable.

An accumulation can also be considered as a special case of the visitor pattern. An accumulation operation gathers information from heterogeneous classes [Ker05, p. 320]; for example, by taking the sum of all instances of a `Price` class. In this example, an accumulation implementation can take several forms, such as: (1) *An object-oriented decomposition:* A visitor pattern-style traversal where an object visits `Price` instances and accumulates the summation in a field, to be retrieved later. Or, (2) *A control-flow-based decomposition:* A single procedure that directly traverses the object structure, handling cases through a sequence of conditionals with `instanceof` checks.

Figure 4.10 shows a hypothetical sequence of refactoring steps that transforms code from using the DJ library to using a control flow-based decomposition: Initially, the code, and its associated mapping, show that the summation of `Price` instances is achieved by using the DJ library. The advantage with the DJ solution is that it encapsulates all code related to `Price` summation into a single location. This code is then translated into another implementation of the `VisitorConcept` interface, using the Gang of Four implementation. This translation leads to a more efficient version, but one that lacks the complete encapsulation of the DJ solution. Because the Gang of Four implementation of an accumulation operation is common, the `Accumulation` interface has a

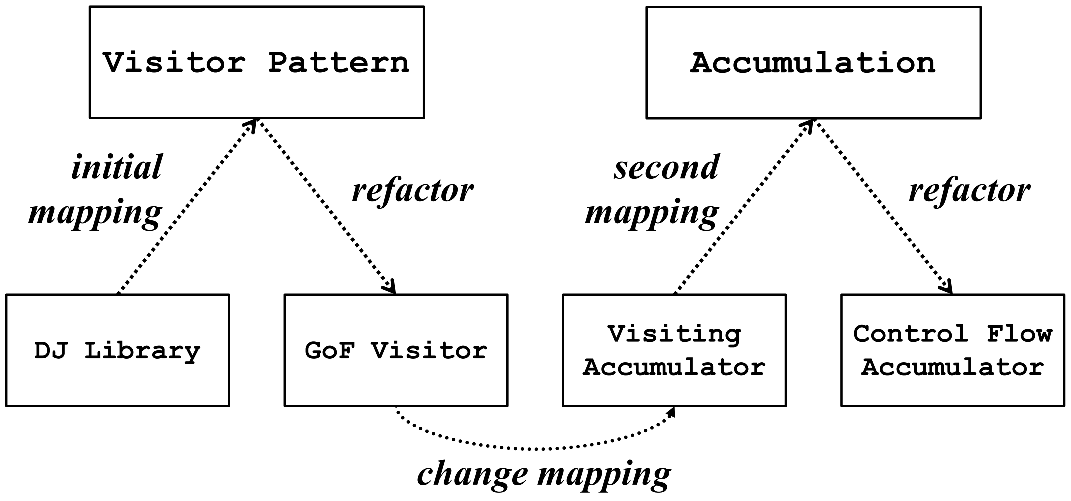


Figure 4.10: A single implementation can be “factored” in different ways.

`VisitingAccumulator` option that can match it. Such a matching can be made by now changing the mapping instead. When the mapping is changed, the Java portion of the program remains the same, but the checks performed on the program and the program fragments that get matched change.

Such a change in mapping is a *factoring* close in spirit to the mathematical sense of the term: The code remains the same, but the focus of the analysis changes. The code can then be refactored to a control flow decomposition implementation that also implements the `Accumulation` interface. Thus, with a change of a mapping and two refactorings, it’s possible to transform code from one option to another even when the options don’t share a common interface.

The use of a control flow-based decomposition is not just limited to the accumulator case. For example, consider an evaluator for an interpreter. The programmer has a choice in how such evaluators are decomposed: Should each member of the class have an `evaluate` method that performs evaluations on its sub-expressions? Or, should there be a single (recursive) `evaluate` method that directly traverses the object structure by performing `instanceof` checks on the expressions?

The former is a traditional object-oriented decomposition, while the latter is a procedural decomposition. The object-oriented decomposition is more efficient and in some respects cleaner, while the procedural decomposition arranges related code in the same location and provides a view that makes the logic easier to change. The procedural decomposition is also more error prone: In the case of deep tree structures, it's possible for a conditional that is more general to appear before a conditional that is more specific, thus silently masking the more specific operation. The Java language only provides such checking for exception `catch` blocks—i.e., checking if a more general `catch` block makes a later one unreachable. Future support for expressing partial orders in Arcum could allow for checks that can detect bugs in `if-else` sequences. Thus, the extra checking provided by systems like Arcum could allow for practices that were once considered error prone to be practical.

4.6.5 Discussion of the Development Process

The development of the `VisitorConcept` and its related options required very concrete and precise thinking about the implementation issues. In some regards, the declarative nature of the Arcum language helped because it made certain relations in the program explicit. For example, once the `hasA` relation was used, I could then ask the questions "Should this be an `isA` relation instead? Why not?" and "Or perhaps this should be a `mayHaveA` relation?" Looking at the descriptions of implementations makes some corner cases more obvious, because the better known corner cases of relations are easier to recognize.

Determining what to put into the Arcum interface was also a coding consideration: The `classGraph` and `traversalGraph` relations were only used by the `GoFVisitor` option, but were general enough to be defined in the interface. To determine that the `visit` concept also needed to be in the interface I had to look in sample source code for real occurrences of the traditional visitor pattern and the DJ library. Without having these concrete code samples to look at it was too easy to forget about special cases in the pattern's implementation itself.

The development process I found most effective was to write a mini-program that demonstrated the pattern and then writing the first `option` to match the demonstration. I could then test the Arcum code against the mini-program before applying it to the real life use and testing it again.

Section 4.1, in whole, is a reprint of the material as it appears in Beyond refactoring: a framework for modular maintenance of crosscutting design idioms. 2007. Shonle, M., Griswold, W., and Lerner, S. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering* (Dubrovnik, Croatia, September 03–07, 2007). ESEC-FSE '07. ACM, New York, NY, 175–184. The dissertation author was the primary investigator and author of this paper.

Chapter 4, in part, is a reprint of the material as it appears in Addressing Common Crosscutting Problems with Arcum. 2008. Shonle, M., Griswold, W., and Lerner, S. In *8th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 64–69, 2008. The dissertation author was the primary investigator and author of this paper.

Section 4.4.4, in part, is a reprint of the material as it appears in When Refactoring Acts like Modularity: Keeping Options Open with Persistent Condition Checking. 2008. Shonle, M., Griswold, W., and Lerner, S. In *Second ACM Workshop on Refactoring Tools (WRT)*. The dissertation author was the primary investigator and author of this paper.

Chapter 5

User Study of Arcum

This chapter describes how three pairs of experienced programmers performed a variety of tasks using Arcum. From my analysis of both the words used by the participants and the different approaches taken to solve each problem, I present the metaphors that the participants used to think about crosscutting code, and the development styles that they used to address the difficulties of crosscutting.

With this understanding of how experienced programmers use Arcum, I make several observations:

- The Arcum process is a successful way for programmers to reason about several concepts in isolation. I show how the participants, by using previously existing Arcum examples and feedback from the IDE, were able to develop working Arcum code. I observed two distinct styles by which programmers arrived at their working solutions, one based on copying existing examples and another based on incrementally adding code to an always-executable form. The two styles are not mutually exclusive and I believe these styles were chosen in order to get feedback from the tool as soon as possible, assisting the formation of the mental model of the crosscutting concept.

- Not surprisingly, the process of writing programs that describe crosscutting carries with it not only some of the challenges of regular programming, but further challenges of its own. For example, programmers have to think about not just the crosscutting concepts's implementation, but also the description of the implementation, and the different forms that alternative implementations may take. I show how these challenges of meta-level programming manifested themselves in my study, and how programmers used Arcum to address them. A common mistake was to confuse types and entities of those types with each other.

- IDE support is essential for understanding crosscutting as it appears in real programs because its scattering and tangling inherently covers more code than comprehensible in a glance. For example, I observed that participants relied upon Arcum's pattern matching visualizations, Arcum's transformation preview pane, and various error reporting capabilities in the Java compiler as well as the Arcum compiler.

- There are opportunities to improve the Arcum system and related Aspect-Oriented Software Development (AOSD) tools, based on the activities and areas of confusion I observed the participants use to cope with crosscutting concepts. For example, I believe the IDE can make definition/use relationships more explicit to the user. As another example, I noticed a disconnect between certain keywords in Arcum and the metaphors that the participants used when coding with those keywords. This disconnect suggests a metaphor-based approach to keyword naming, something that I believe may help make Arcum a better language for novices. Another improvement that could improve productivity is a means for the environment to assist the inference of larger patterns in the code, such as by generating queries (and showing their matches) when the focus is placed on one particular code instance.

After describing the study itself (Section 5.1), I discuss how the study participants approached crosscutting (Section 5.2). I then discuss the metaphors and techniques used by the participants to help them understand instances of crosscutting concepts, such as abstracting its essential features into a workable mental model (Section 5.3). In analyzing the participants' activities, I observed two development styles that participants used to construct their solutions, one based on copying existing solutions, and one based on incrementally building a solution from scratch (Section 5.4). Next, I discuss the challenges and techniques used when writing custom checks or custom refactorings—which requires thinking about code not just in the concrete form in which it exists, but also the form in which it may exist (Section 5.5). Then, I discuss related studies, which either evaluated tools similar to Arcum, or had similar evaluation techniques (Section 5.6). Finally, I end this chapter with preliminary design guidelines for the development of aspect-oriented tools (Section 5.7).

5.1 Study Description

I chose to perform a qualitative, exploratory study, to document the experience of programmers using Arcum, and I wished to discover the basic phenomena and issues revolving around Arcum's use in modularizing crosscutting concepts. My expectation was that programmers with experience writing large programs could understand how to effectively use Arcum (Section 5.1.3).

For the study, I recruited six participants (Section 5.1.1), who worked in pairs on tasks; such as changing a program and writing checks to verify properties of the program. I provided the participants with written instructions that described the sequence of tasks to perform (Section 5.1.2) together with short reference materials for the Arcum language. Appendix C shows the complete materials provided. I observed these participants over two sessions, which took place in a quiet office environment.

I used pair programming in order to capture natural conversations, closer to what might occur outside of an experimental setting [Miy86]. Pair programming is common in many real-world settings, especially on complex tasks like those that might be solved with Arcum. One alternative would have been to use individual sessions with each programmer, but that would have required either the less natural "please think out loud" technique, or constant questioning from the experimenter, which could introduce bias through tone of voice and other cues.

Each pair's audio was captured along with the contents of their computer screen and file system output. TechSmith's Camtasia was used for the recording. After the audio component of the sessions were transcribed, I analyzed the participants' use of language, in order to see the kinds of metaphors they used and how they thought about the process. This analysis led to insights based on their expectations and intuitions,

Table 5.1: Participant demographics: Participant's industry experience, whether he or she has used Eclipse before, and programming languages known.

Group	Participant	Experience (months)	Eclipse	Languages Known
A	A1	6	no	Java, Lisp
	A2	6	yes	Java, Lisp, ML
B	B1	6	no	Java, ML
	B2	0	yes	Java, ML
C	C1	12	yes	Java, Lisp, Prolog
	C2	6	no	Java, ML

together with what kind of intellectual tools they use, such as abstraction, to cope with the change tasks.

This study was a follow-up to a pilot study, consisting of the tutorial session, where I determined the original language syntax used was less natural and could be refined to be more Java-like and conform closer to the participants' expectations. I also found in the pilot study some confusion with Arcum's scoping rules, so for this study I addressed the rules in the written tutorial.

During the study, the participants would occasionally ask me a general question about Eclipse, Java, or Arcum, and I provided answers. Also, in the process of using Arcum, the participants would sometimes encounter known limitations with its type checking of incorrect code. In these cases, I compensated by alerting the participants when errors were made by describing the error message that a complete (non-prototype) version of Arcum would have given.

5.1.1 Study Participants

All six study participants were graduate students in the computer science department and experienced programmers. Table 5.1 shows their backgrounds. I chose experienced programmers because part of the intention of Arcum is to enable experienced programmers to write transformation and checking libraries that could be used by a wider audience.

5.1.2 Study Tasks

The study comprised two sessions for each programming pair. The first session was a tutorial that covered Eclipse and the Arcum plug-in, and included step-by-step guides for completing the tasks. The second session was held on the following day and covered program transformation and checking tasks without the help of step-by-step guides. The pairs were given 90 minutes to complete the first session, and 60 minutes to complete the second session.

Tutorial Session

The tutorial session used a small (83 line) Java project that has three classes implementing a simple linked-list with associated utility operations, including a `main` method that performed a unit test.

Manual Transformation Task. The first task required making a simple conceptual change to the program without using Arcum: Change the storage of a value associated with an object from an internal (field) representation into an external (sparse) representation. What these two implementations have in common is that both are ways to implement the common practice of associating attributes with objects. (This is same transformation used as the main example of Chapter 2).

The participants were free to use Eclipse as they saw fit to perform the change. Even though the change was simple, I devised the code so that two bugs would occur if the changes were made carelessly: a `NullPointerException` could occur if a corner case in the program was not identified (discussed in Section 5.2.1) and a semantic change was possible due a particular method call not being a perfect substitute for a Java operation (discussed in Section 5.5.1).

Arcum Training Task. The next task gave the participants practice with executing Arcum code and provided the background for writing code in the language itself. A complete code example was provided that contained one `interface` representing the attribute idiom and two `option`s representing the alternative implementations (internal field versus external map). The attribute interface has two concepts, `attrGet` and `attrSet`, which abstracts the attribute read and write operations. Arcum allows a programmer to switch between the two `option`s, where one option, for example, represents an attribute read as a field reference, while the alternative option represents an attribute read as a method call. This attribute example (shown in Figure 2.5) automates the refactoring performed for the *Manual Transformation Task* and also demonstrates several features of the Arcum language while being a short example.

The training was split into three sub-tasks: (1) Learn the concepts of the Arcum language; (2) Run a sample transformation; and (3) Follow a step-by-step guide to insert an additional check into the provided Arcum code.

Custom Check Creation Task. After being given the step-by-step guide for inserting extra checks, the participants were asked to insert another check. The purpose of this check was to automate the detection of the bug discussed in Section 5.5.1.

Automate a Transformation Task. Finally, with the basics of Arcum covered, the pairs were asked to create two Arcum options that implement the same interface, thus allowing a transformation to be made. This task had three sub-tasks: (1) Create an option (with its required interface) that recognizes all references to `System.err`; (2) Write an alternative option to recognize references to an error log accessing function; and (3) Perform a refactoring using Arcum to transform the uses of `System.err` into calls to the log accessor method.

Advanced Session

The session using Arcum without step-by-step instructions used the HTML renderer component of the Lobo project [Lob08]. Lobo is a web browser written in Java. Lobo was chosen because it was the top desktop application project available from SourceForge (a repository for open-source code) that was written entirely in Java and compilable with Eclipse. Lobo is also well-written and rich with crosscutting concepts.

Review Code Examples Task. The first task of the second session was to review example Arcum code and explore the results of the provided Arcum queries. These queries were applied to the Lobo project and provided many results and different cases to explore. The example code given only had one option, so no transformations were possible. Instead, the purpose of the option was to demonstrate several pattern syntaxes (and their matches).

Change StringBuffer to StringBuilder Task. Next, the pairs were asked to migrate the Lobo codebase from using the always-synchronized `java.lang.String-Buffer` class to the more efficient `java.lang.StringBuilder` class (this is an instance of the class library migration problem [BTF05] and inspired by a suggestion from De Sutter et al. [STD04]). Although this change could easily have been made with a global text-based find-and-replace (because the two classes have the same API and neither of them require Java `import` statements), I wanted to see how programmers would solve such a transformation with Arcum. Accomplishing this task with Arcum requires recognizing and replacing program fragments that belong to different syntactic categories (namely, type declarations and constructor call expressions).

Check Logging-Idiom Task. Finally, the participants were asked to consider the following code snippet:

```
public class DocumentBuilderImpl /* ... */ {
  private static final Logger logger =
    Logger.getLogger(DocumentBuilderImpl.class.getName());
  /* ... */
}
```

Here, a `logger` instance is used by the class `DocumentBuilderImpl`, to log activities related to the execution of the class. This pattern repeated itself in the project, where the argument to the `getLogger` call is the name of the class that defines the static field.

This special usage can be considered a crosscutting concept: Any changes to the policy (e.g., of how the log is acquired, or which log is used) would require global changes. One simple property of this crosscutting concept that can be checked is if the correct argument is passed. For example, a copy and paste error would lead to the logs of one class to be written to the log of the copied class. The instructions for this task required the pairs to write Arcum code that could check for this property. After the second session, the pairs also participated in a separate post-study interview.

5.1.3 Performance of the Tasks

All three groups successfully completed the tutorial session in the time alloted, but no group fully completed the advanced session. Table 5.2 shows the time it took for each group to complete each task. All groups finished the tutorial session early but used

Table 5.2: How each group performed the tasks over the two sessions. A '+' indicates when the pairs ran out of time and could not fully complete the task. The '*' indicates that the task was completed with minor assistance.

	Time to Complete (minutes)		
Task	Group A	Group B	Group C
Manual Transformation	11	15	11
Arcum Training	22	18	19
Custom Check Creation	10	19	6
Automate Transformation	35	21	21
Total for Tutorial Session	78	73	57
Review Code Examples	6	5	4
Change StringBuffer	30	*29	24
Check Logging-Idiom	+16	+16	+30
Total for Advanced Session	52	50	58

all of the time alloted for the advanced session. Times for the advanced session do not add up to a full 60 minutes due to group start up delays.

During the *Change StringBuffer Task*, Group B planned a solution that would have required a significant amount of code to fully complete. In the process, the group was blocked by a bug in Arcum's evaluator, which halted their progress. It's conceivable that Group B could have made this alternative technique work, but the blocking bug could not be immediately resolved. Instead, I hinted that the solution to the task could be simpler and reminded the group to look at the task instructions again.

Group C made the most progress on the *Check Logging-Idiom Task*. Perhaps it was not just a coincidence that group C spent the most time on the task compared to the other two groups: Group C had almost twice the time, at 30 minutes versus 16 minutes, because they completed the previous two tasks relatively quickly and the session started on time. The specific challenges of this task are discussed in Section 5.5.2.

5.1.4 Threats to Validity

As with any user study, there are some threats to the validity of the study. I identify here the main threats.

Graduate student participants. The participants in my study were computer science graduate students from two areas: programming languages and architecture. Graduate students in general have more experience in seeing new ideas and exploring non-conventional ways of solving problems. As such, they may be better equipped to quickly understand and use a new tool like Arcum. Furthermore, programming languages graduate students have even more experience with adapting to new programming models, and many of them would already be comfortable with the idea of programs processing other programs.

Pair programming. My use of pair programming was instrumental in identifying what the participants were thinking about while they were performing tasks. However, it also brings up the question of whether or not my observations generalize to individual programming.

Instructions causing bias. The study instructions given to the participants contained explanations of how Arcum works, and as a result contained language that may bias the choice of words used by participants in the study.

5.2 Reasoning About Crosscutting

In the strictest sense of the term, no module could utilize another module without some form of crosscutting, because the module's *interface* must be known by all modules that need to use it [BC99, SGCH01]. But not all forms of crosscutting are equal: By their nature, well-written interfaces are stable [Par72], so when elements of the API (such as method names) crosscut the program, they do not become liabilities when that module's implementation needs to change. This section focuses on the kinds of crosscutting that do not naturally fit into stable interfaces and thus require reasoning over several different modules.

I discuss the strategies that the pairs used to cope with this crosscutting, the pitfalls they encountered, and I suggest possible improvements to methodology or the environment to assist non-modular reasoning. Examples of such reasoning include identifying all references made to a single program element, such as a method or a field. Even though the Eclipse IDE, AspectJ, and Arcum are all well-equipped for finding such references, I found their use involved several pitfalls.

In the case of searching, I identified instances where the participants misunderstood the information provided by the environment, and other cases where the participants searched with the wrong query. In addition, I observed pitfalls in how programmers reason about documentation and other artifacts written in English.

5.2.1 Using Build Errors as a Guide

To successfully complete the *Manual Transformation Task*, in which Arcum was not used, all three groups first deleted (or commented out) the field to be stored externally (the field was named `next`) and replaced it with a static `java.util.Map` declaration (also named `next`). The following discussion is representative of the discussions or actions of all three pairs:

> *A2: So everything should be broken.*
>
> *A1: Yeah it's broken now we have to go through and find all the instances where the next is accessed.*
>
> ...
>
> *A1: Okay so let's just search for all instances of next right?*
>
> *A2: Well I think that all of these little red things will help us out.*

Here, the "little red things" are Eclipse's error markers associated with syntax errors, type errors, or other problems. With the `next` field now being stored externally, all reads from and writes to that field must instead pass through the static `Map` as `get` (lookup) or `put` (store) calls. The common mistake made was believing that all of the code locations flagged by the compiler were all of the locations that needed to be changed to either `get` or `put` calls.

However, the errors introduced from the change were type errors and did not have a perfect correspondence to references: The accesses to the `next` field had different types now that the `List` class's `next` field changed its type from `List` to `Map`. Yet, expressions with values of these two types can exist in the same code context. For example, the following loop was in the sample program:

```
while (list.next != null) {
  list = list.next;
  /* ... */
}
```

Here, with the `next` field made static, both accesses of `list.next` should be changed to `List.next.get(list)`. The second access (after the change is made) is a type error, because it would be assigning a `Map` to a `List`. However, the first access is not a type error, because instances of `Map` can be compared to `null`. Note that because `next` is a static member, the notation `list.next` is still valid, but generates a warning in Eclipse because it is a non-static reference to a static member.

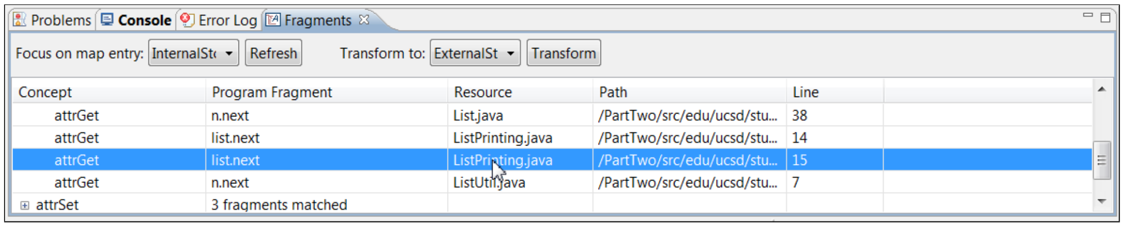
Thus, the compiler errors issued could not be used reliably as a guide for all references to `next`. All pairs identified the `while` loop conditional as needing to be changed, perhaps because of its proximity to another line of code explicitly marked as an error, or because the Eclipse Java editor highlighted it with yellow (to represent the warning). Had the change not been caught, eventually a `NullPointerException` would have been detected during testing.

One way to improve the compiler as a guide would be for the IDE to identify trends among the error messages it creates. In particular, when several errors have a single declaration in common, the IDE can include links to that declaration, and then backward links to *all* references to the declaration, flagging the ones that have the errors, to let programmers notice patterns and consider other cases that need to be addressed.

5.2.2 Making Direct Queries with Arcum

During the advanced session, the participants were asked to reason about several instances of crosscutting, such as the scattered use of the `StringBuilder` class, or the scattered instances of the logging idiom. Figure 5.1 shows Arcum's Fragments View, which was utilized by the pairs in many of these instances to visualize the matches.

In the post-study interviews, one participant compared Arcum to a "semantic grep," a comparison that holds in several regards: The Fragments View provides pro-



Focus on map entry: InternalSt Transform to: ExternalSt

Concept	Program Fragment	Resource	Path	Line
attrGet	n.next	List.java	/PartTwo/src/edu/ucsd/stu...	38
attrGet	list.next	ListPrinting.java	/PartTwo/src/edu/ucsd/stu...	14
attrGet	list.next	ListPrinting.java	/PartTwo/src/edu/ucsd/stu...	15
attrGet	n.next	ListUtil.java	/PartTwo/src/edu/ucsd/stu...	7
⊞ attrSet	3 fragments matched			

Figure 5.1: Arcum's Fragments View: Shown are four different matches in the program that represent instances of the `attrGet` (attribute access) operation.

grammers with a compressed view of one aspect of the crosscutting concept, much like the output of grep. Such compressed views can help programmers focus on areas of interest without having to read unrelated code [Gri01]. Further, much like the grep command, Arcum can be used for pattern matching. However, Arcum's pattern matching is based on desugared AST nodes (instead of characters in a text file) and can take into account type information. The desugaring of Arcum's matcher was noticeable during the *Review Code Examples Task*:

> *A1: fieldAccess [..] take a look, pick one... OK, pick another one. Are they all "this.document"?*
>
> *A2: I bet we can find out by looking at the Arcum file... "target.document", so in this case [..] target must always be "this"?*
>
> *A1: Let's scroll through the [Fragments View] — "document" and "this.document"*
>
> *A2: Oh I see, so "target" could be like the null expression*

When considering the *Change StringBuffer Task*, Group A realized there was a corner case with replacing uses of the `StringBuffer` class with the `StringBuilder` class: If an external library returned a `StringBuffer` then the library itself could not be changed, so some conversion operation would be necessary. The group considered making a query to determine if such calls were present:

> *A1: Maybe it's not something we can actually fix with this because it's not our code, it's a bad library dependence.*
>
> *A2: Well what we can do is detect where it happens.*

One possible pitfall with this approach is what happens when the query declaration does not match the programmer's intentions: If there is an error in the query's construction, it can create false confidence about the properties of the program. A defensive programming approach might ensure that the queries were tested by injecting known matches into the code, but such tests would not be complete.

The flip side to this problem is that sometimes the query is complete and correct, but the user looks at the search results from a different query, also leading to false impressions of the code:

A2: Oh, those are, oh we were looking at the wrong thing. Cool. But now we know there are ones we're not getting too, right, because...

A1: [..] it can take various sorts of arguments

A2: Right.

In this case, it took the pair a longer time to understand the crosscutting nature of the code: Not only did they have to reason about the program itself, but they also had to reason about the correctness of the queries. This difficultly is partially addressed by Arcum through its pattern syntax: When programmers are reasoning about the Arcum code, it becomes a model of their understanding of the crosscutting code, and even looks like the crosscutting code. AspectJ's pointcut language takes an approach different than Arcum's by focusing on semantic joinpoints instead of desugared syntactic patterns. Arcum's approach of having the patterns look like the code being searched for can be intuitive for reading and understanding the patterns; however, the desugaring adds an extra level of abstraction which can be deceptive when the semantics of the desugaring are not fully understood.

5.2.3 Confusing Definition with Reference

In Arcum, the Java program fragments that are computed on are typed according to their syntactic category. For example, an `Expr` (expression) fragment is something that could be found in a `Statement` fragment, just like the corresponding Java grammar rules. However, I observed instances where Arcum's types became a source of confusion:

B2: So what are we looking for an expression? Actually that's not even an expression. Right now we're just looking for a type.

B1: I wonder if we can just hit 'type.' Sure, let's try it, see what happens.

B2: Do you think that will give us occurrences of the name of that class or it'll just give us definitions of that class?

Here, the participants are unclear what the Arcum type `Type` means. The reference sheet given to the participants defined a `Type` as: "A Java class, enum, or interface," and it remained unclear to the participants if this meant the unique definition for the type (the correct answer), or the many references to the type. When participants initially pattern matched for `java.lang.StringBuffer` they were surprised to see only one result listed (one without an accessible source line, because it is in a compiled binary). The same pair clearly desired a more direct relationship:

B2: Yeah, is there like a kind of predicate that is "isUses"...

The above confusion about what `Type` would match is in fact a meta-programming problem: Arcum types refer to syntactic categories of Java code, and thinking at this meta-level requires additional care and attention from the programmer.

This meta-level confusion suggests two possibilities to explore: (1) Arcum's type system could become richer, having `-Use` and `-Definition` suffixes for each type, to make the desired choice explicit. For example, a `FieldDefinition` type would refer to the syntactic field declaration that appears inside its defining type, while a `FieldUse` type would refer to an expression. Or, (2) Arcum could have a relaxed type system, where the type of the program fragment named depends upon how it is used. Alternatively, the definition/reference confusion could merely be a part of Arcum's learning curve, making language guides and tutorials the areas to improve.

As suggested in Section 5.2.1, the definition/reference relationship can be given more importance in the environment through added hyperlinks between the two. Such two-way links are already part of the AspectJ Development Tools support for viewing the relationship between join-points and advice. These guides could be taken a step further by creating a tool in the environment that suggests code (e.g., patterns) that will match the Java code currently highlighted by the user, and also include a link back to the full results of each proposed pattern. Such a feature, in the case of AspectJ, would allow a user to select a method call in the program, and a separate view would generate code for the different pointcuts possible to match that join-point. The generated code could then be copied, or explored for the other matches it creates.

5.2.4 Using Reference Materials

Reference material is another source of information that participants used to help them reason about crosscutting concepts. In particular, I observed participants using API documentation and forming models based on the texts discovered in the program.

Using the Documentation

When working on the *Manual Transformation Task* the pairs needed to know what was returned by the `Map`'s `put` method. Eclipse displays Javadoc documentation when the mouse hovers over a method:

> *A2: There's some way that it will give you the type. There you go. You do need to mouseover it.*
>
> *A1: It's "value."*
>
> *A2: So it gives you the value. So in this case we can use it. Just like this one. So we can just put this guy.*

During the above discussion, the participants placed their mouse over a call to the `put` method, and the signature for the method was displayed as:

```
V put(K key, V value)
```

Noticing that the return type was the same as the type of the `value` argument, the participants assumed that `put` would return the same value it was given. This was a natural assumption to make given its similarity with the Java assignment operator, yet what the `put` method actually returns is the *previous* value that was stored in the table. This type/value confusion is another example of a meta-level complexity: the participants above mistakenly thought that value equality could be deduced from the type equality.

The participants discovered their error after executing the program and seeing how its output had changed. The participants returned to the API documentation and scrolled down to reveal the explanation for the return value. Thus, one pitfall of thinking about operations on a higher-level, where multiple correct implementations for the operation are possible, is that details known about one specific implementation might lead to incorrect generalizations about all implementations.

Using Error Message Texts

The sample Java and Arcum programs from the tutorial session contained error handling code with associated error messages. The Java program checked to see if the `args` array given to `main` was `null`, and the Arcum program checked to see if a function call was used as intended.

The error messages printed by these checks became an essential part of how the participants worked to understand the program. By virtue of being visible by the user, such messages relay information at the program requirements level. For example, the Java program had a line in `main` that printed the following error message, under some conditions: "panic! no args given," which lead to the following discussion:

A1: Okay, so we have to take in some kind of arguments. Can we see where arguments are actually being given in the? Where it's being run? Cause that's like [...] Command-line args?

A2: Yeah, so since it didn't say "panic no args given". There's.

A1: Yeah, so it must be. It must be getting some sort of args.

The participants in the above discussion saw that the message was *not* printed at run-time, and so they assumed that some arguments must be passed to `main`. However, this conclusion is incorrect, because the condition under which the error message is printed tests for `args` being null, and so it's possible that the error message is not printed, and still there are no `args` (if `args` is the empty list). The participants in the end realized this:

A1: Or uh, no. Hold on. Can you close that? That's checking that they're null, not that they're an empty. And it's probably an empty string.

Thus, care must be taken when writing the contents of error messages, because programmers can sometimes interpret them semantically. In the above example, a more accurate error message would be "args is null!".

I also observed that a properly written error message aided the reasoning of the program. For example, the following check in Arcum was provided to the pairs:[1]

```
require "The value of `getExpr must be used":
  !isExpressionStatement(getExpr);
```

Here, the value returned by the read operation on an attribute must be used, otherwise it is flagged as an error. Such a check is useful because it is likely an error if an attribute is read but not used. The pairs were asked in the *Custom Check Creation Task* to write a similar check, but this time to check that the value of a *write* to an attribute is *not* used. The purpose of this check is to prevent the case discussed previously—where the `put` method returns the previous value in the table—by restricting all code forms to the lowest common denominator. Thinking at this high level made it easy for the participants to produce the correct solution:

> *A1: Well so we can just probably use "isExpressionStatement" right? Cause here it's "this must be used." And here it's "it can't be used."*

5.3 Abstractions of Crosscutting

Through the process of reasoning about the crosscutting of an idiom, a mental model of the crosscutting is formed in the programmer's mind. Because Arcum's `interface` and `option` constructs are modeled after the concepts of modularity, I hypothesized that these constructs would provide a natural form for expressing the crosscutting. Arcum's notion of creating an interface for crosscutting code was partially inspired by my previous work on XPIs in AspectJ [GSS$^+$06, SGS$^+$05]. I found some support for my hypothesis, but I also identified cases where bad habits in the context of modular design (such as poor naming choices) remained difficulties in the context of Arcum.

5.3.1 A Decompositional Model

Arcum enables a refactoring operation to be decomposed into two options with a common interface. As a result, a transformation can be broken down first by thinking

[1]Note: The preliminary version of the Arcum language used in the study had the keyword `require` instead of `check`. See Section 5.5.2 for a discussion of this change.

about the option that describes the current implementation as a search:

A1: First let's see if we can just find them, and then if we can replace them

I found that this divide-and-conquer strategy accomplishes the task, but does not encourage the creation of an effective abstraction. For example, Group A had given their concept the name '`search`,' which described what they wanted to write the concept for, but did not describe what the program fragments captured by the concept represented. The `interface` associated with the `option` was named `FindSysErr`, after the first task the participants were given, and the `option` itself was named the abbreviation `FSE`. Similar problems occur in OOP, for example, when classes are named after verbs instead of nouns. In the case of Arcum, with its meta perspective, the effect is more misleading. When it came time to write the second `option`, the participants noticed the trouble with the names picked:[2]

A1: Realize search. Our naming has gotten fairly horrible because we're doing replace with search.

The issue of giving Arcum options and interfaces meaningful names is related to the meta-level aspects of Arcum: The entities being named are not in Java, but rather one level up from the Java code. As a result, these naming difficulties could in part be attributed to the intellectual difficulties of understanding and conceptualizing meta-level constructs.

However, at other times, the different levels of abstraction and the elements of meta-programming required were very natural for the participants:

C2: Know what we should do? We should write another Arcum file that transforms this Arcum file to the file we want.

C1: But it's gonna be adding things so we can't really do that, it's not a refactoring it's an adding, so...

The fact the participants are entertaining the idea of applying Arcum to itself shows that they have gotten comfortable with the idea of developing code that manipulates other code.

[2]Similar to the `require`/`check` keyword change, the preliminary version of the Arcum language used in the study had the keyword `realize` instead of `match`. See Section 5.5.2 for a discussion of this change.

5.3.2 An Overloading Model

I observed an alternate metaphor for reasoning about Arcum `interface`s based on overloading. In overloading, a group of methods that are "the same" in some sense can have the same name, even though they are applied to objects of different types.

A1: So the options I think are, basically, it's sort of an overloading.

A2: Implementing the field or whatever it is.

The comment by A2 follows the metaphor further: Although the `interface` is about attributes in an abstract form, a field is one valid implementation of that attribute idiom. The idea of a field is overloaded, because attributes can be thought of as fields, even when they're implemented as external lookup tables instead.

5.3.3 Patterns as Abstractions

Abstraction, in the general sense of the term, is what makes Arcum's Java pattern syntax intuitive and useful. A necessary part of this usefulness is to gloss over subtle differences between otherwise similar fragments of code. For example, participants would write patterns for the Java elements they were searching for, writing them in their most familiar forms. Arcum would then desugar both the pattern and its internal representation of the program to perform the matching. This desugaring process led to actions unexpected by the participants. For example, when Arcum performs a transformation, it adds `import` statements as required:

A1: So we're going to import this...

A2: Import it into what? Import it into Arcum? Oh yeah, I guess so.

B1: There's the imports, ah they didn't even import star, they imported only what they needed to.

Thus, through its desugaring abstraction, Arcum freed the participants from thinking about details of the transformation that they did not (initially) consider. However, Arcum's desugaring was not completely seamless, as reflected by their surprise.

5.3.4 Other Metaphors Observed

String Substitution. Solving change tasks with Arcum involves thinking about transformations. One group used their knowledge of string substitutions and applied it to the refactoring context where direct accesses to `System.err` are replaced with indirect method calls, given that some of those method calls already exist:

> *A1: So here is the interesting experiment then, would be what happens if you put both a systemerror and an errorlog.*
>
> ...
>
> *A2: It's like you have abb, you change all b's to a's and you change all a's to c's... or change all b's to c's. Or whatever. But that makes sense.*

Participant A2 here is essentially answering A1's question using a string replacement metaphor. One participant also made a direct comparison to a find and replace tool:

> *C2: We could have just like, in EX replaced all instances of the word builder with the word buffer, or opposite*

Here, "ex" is the command the participant uses to access string replacements in the vi text editor.

Types in Programming Languages. Because Arcum contains types named directly for syntactic categories, the participants could reason about these categories better:

> *A2: Oh, wait a second, but the arguments. The list of arguments is not an expression, it's a list of expressions. A list of expressions isn't an expression.*

5.4 Development Styles

As indicated by Table 5.1, the participants of my study have diverse backgrounds in terms of their previous programming experience, their previous knowledge of Eclipse, and their familiarity with Java. Despite this diversity in background, I noticed a common theme in their approach to dealing with crosscutting concepts: They all focused heavily on getting feedback early.

When starting on a new task, each group would invariably strive to quickly get to a point where the Arcum tool could give them feedback on their approach. Although getting early feedback is a common approach for mitigating the cost of mistakes in regular programming (for example with the use of type-checking), my study confirms that getting early feedback is also important (and possibly more so) when developers are dealing with crosscutting.

Even though all groups had the same goal of getting feedback early, I observed two different development styles for attaining this goal: (1) A copy/paste/modify approach that makes heavy use of previously written Arcum interface and options; and (2) A bottom-up approach guided by trial-and-error. I describe each of these two development styles in turn.

5.4.1 Reuse of Uses

The first development style I observed involved inspecting, copying and then modifying previous Arcum code in order to quickly get a solution that could be tried out immediately, with the possibility of later refinements. This idea of using already existing examples to guide the development of code with unfamiliar constructs is known as the Reuse of Uses [RC96].

As a concrete example, when group A started the tutorial task of changing the error-log stream from `System.err` to a custom stream, they had to write a new option for finding all references to `System.err`. In order to do this, group A looked at the previously provided options for storing attributes, chose one of them to copy-and-paste, and subsequently went on to edit the copied option:

A2: I wonder if we can like copy and paste.

A1: Well we can certainly start with that.

Using previously written Arcum code to guide the development of new Arcum code was also prevalent when writing pattern expressions:

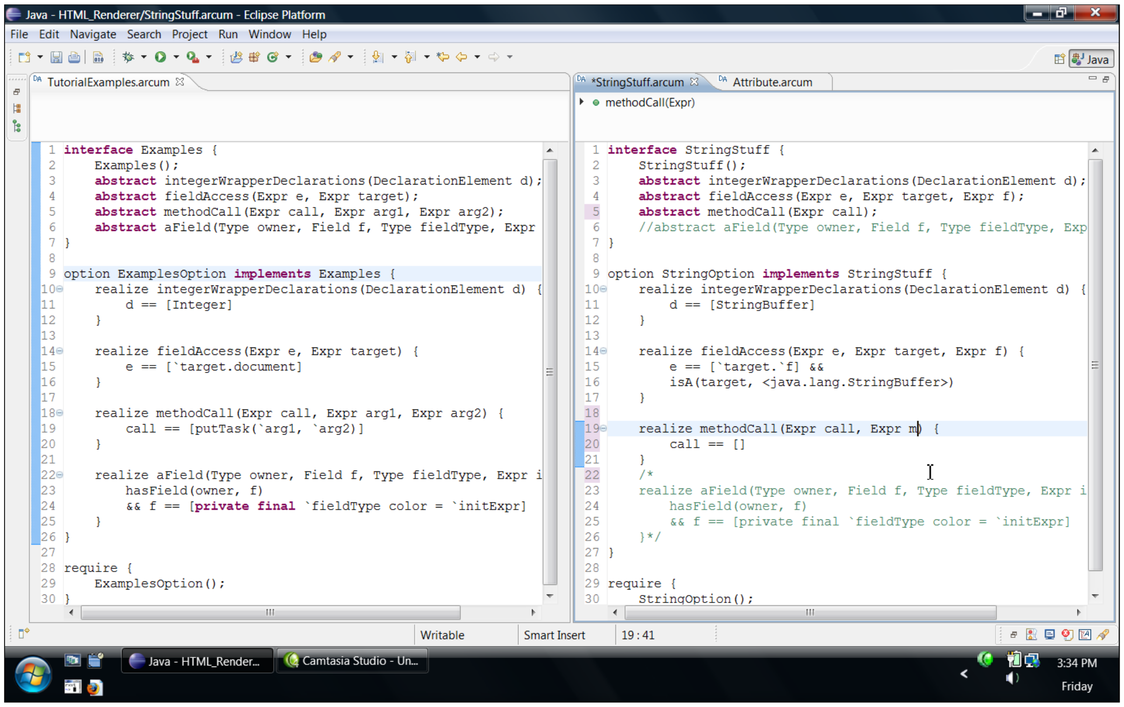
B2: How we wanna wrap in function call... so let's look at the ExternalStorage implementation. Where is that, farther down?

B1: Oh, it's down here, yeah.

B2: Right? It's almost like analogous to...

B1: Yeah.

B2: Internal/external thing.



```
interface Examples {
    Examples();
    abstract integerWrapperDeclarations(DeclarationElement d);
    abstract fieldAccess(Expr e, Expr target);
    abstract methodCall(Expr call, Expr arg1, Expr arg2);
    abstract aField(Type owner, Field f, Type fieldType, Expr
}

option ExamplesOption implements Examples {
    realize integerWrapperDeclarations(DeclarationElement d) {
        d == [Integer]
    }

    realize fieldAccess(Expr e, Expr target) {
        e == [`target.document]
    }

    realize methodCall(Expr call, Expr arg1, Expr arg2) {
        call == [putTask(`arg1, `arg2)]
    }

    realize aField(Type owner, Field f, Type fieldType, Expr i
        hasField(owner, f)
        && f == [private final `fieldType color = `initExpr]
    }
}

require {
    ExamplesOption();
}
```

```
interface StringStuff {
    StringStuff();
    abstract integerWrapperDeclarations(DeclarationElement d);
    abstract fieldAccess(Expr e, Expr target, Expr f);
    abstract methodCall(Expr call);
    //abstract aField(Type owner, Field f, Type fieldType, Exp
}

option StringOption implements StringStuff {
    realize integerWrapperDeclarations(DeclarationElement d) {
        d == [StringBuffer]
    }

    realize fieldAccess(Expr e, Expr target, Expr f) {
        e == [`target.`f] &&
        isA(target, <java.lang.StringBuffer>)
    }

    realize methodCall(Expr call, Expr m) {
        call == []
    }
    /*
    realize aField(Type owner, Field f, Type fieldType, Expr i
        hasField(owner, f)
        && f == [private final `fieldType color = `initExpr]
    }*/
}

require {
    StringOption();
```

Figure 5.2: A direct example of the Reuse of Uses by group B.

Here again, the participants are referring back to the previously provided attribute storage example in order to write a new option.

Figure 5.2 shows group B in the process of editing a copied version of an option. The participants split the window vertically, with the original code on the left (provided for them as an example of Arcum's various constructs), and the edited copy on the right.

I observed yet another example of the Reuse of Uses approach, although in slightly different context: to build patterns, some of the participants copied Java code in a pattern, and then added Arcum variables to it by adding backticks and revising the expression.

The Reuse of Uses development style, with its copy/paste/modify model, allowed participants to quickly build a solution on which they could immediately get feedback. However, in the case where the copied Arcum code is large (say, if it includes both the interface and the options), this approach requires the participants to customize many places in the copied code before getting something that is testable, thus delaying the time to feedback.

5.4.2 Incremental Exploration

The other approach that participants used to get early feedback was to construct a solution bottom-up, using incremental trial-and-error exploration to guide the construction. Instead of copying a complete solutions and modifying it, in this case, the participants would start with an empty file and incrementally populate it with constructs that they could easily test along the way. For example, in the following excerpt, group A clearly uses language that evokes a bottom-up metaphor:

A1: Let's start the null case and see if we can build up from there.

A2: Sure I think that sounds reasonable.

A1: Okay so we'll have an option that realizes nothing. At least give us an interesting error message probably.

Later in the discussion, group A uses language that is indicative of the trial-and-error metaphor, in particular when discussing how to identify constructors with zero or one arguments:

A2: But the other thing is, if we do something wrong... here's another thing... here's a way to know. So let's just do it for the zero case and the one case, we'll just have two rules, which is ugly but it'll work and then [..] if we miss something the compiler will complain because we'll be trying to put a string buffer into a string builder. So let's do it for the zero and the one case, and then ...

The participants here are proposing to only identify constructor calls with zero or one arguments, and see what happens. As it turns out, this is enough for the given task, since `StringBuilder` doesn't have constructors taking more than one argument.

The groups that used incremental exploration also used the "undo" metaphor in their language. This indicates that, not surprisingly, Arcum's undo feature (which undoes all the refactoring changes made in one step) gave programmers the confidence to even entertain the idea of trial-and-error. For example, here is a discussion in which group A realizes that undo allows them the freedom to experiment:

A2: Eleven of one, I guess we can add the two rule now [..] and see if any match the two [..] or we could do the transformation and if it doesn't compile we can undo it

A1: Yep! ... Let's take a look at what it actually turns them into.

Another feature of the Arcum plug-in that helped participants perform experiments was the transformation preview window, which displays the transformations Arcum would make before they are committed. I observed the participants using this preview pane as an exploration mechanism, often looking at the results and then canceling the transformations to further change their Arcum code.

The above examples of using a bottom-up incremental approach points to an important way in which participants managed the intellectual complexity of reasoning about crosscutting concerns: the bottom-up approach allowed participants to build *custom solutions* that were specific to their needs. These custom solutions were easier to develop and to reason about than generally reusable solutions. Furthermore, the fact that new users to Arcum were able to build these kinds of custom case-by-case solutions is a good indicator that Arcum supports incremental adoption: users can start by creating custom solutions as they did in my study, and as they become more comfortable with Arcum, they can make their solutions more general and reusable.

The specialized nature of the solutions developed by the participants also highlights one of the key advantages of Arcum over general and reusable solutions as embodied in IDE refactoring tools. In particular, because IDE refactoring tools are intended to be broadly applicable, they cater to the common case, and as a result may not work for special circumstances. In contrast, Arcum allows the developer to build customized application-specific solutions.

When compared to the Reuse of Uses approach, the bottom-up incremental approach allows programmers to test each pattern individually, which means that they can test the first pattern without having to write all of them down. In contrast, the Reuse of Uses approach uses a more monolithic “change all patterns and test” paradigm. One may, as a result, be tempted to conclude that the bottom-up approach gives feedback earlier. However, this is not necessarily the case, since the bottom-up approach requires building a lot of Arcum boiler plate code to test the first pattern, and that boiler plate can take time for a novice user to develop.

Furthermore, much like using the compiler warnings discussed in Section 5.2.1, a trial-and-error approach may not capture all problems. For example, if some important case is forgotten, and this case is decoupled, from a type checking point of view, from the other cases, then the Java type checker will not find the omission. Knowing what is important to refactor or not is similar to the challenges of modularity and knowing what is stable or not [Par72].

5.4.3 Improving Arcum Development Style

My observations about the above two development styles, and the lengths to which the participants went to get immediate feedback, points to a variety of possible improvements to the Arcum tool. These improvements would, in turn, give the programmers more flexibility in their development styles.

Pattern Tester. A pattern-testing tool could give developers early feedback on whether or not patterns work correctly; allowing developers to try patterns in the IDE and browse through the matches, without having to build any surrounding Arcum code. This tool would improve both development styles: in the Reuse of Uses style, it would allow developers to test the patterns before putting them into the copied version of the Arcum code; in the incremental development style, it would allow developers to try patterns out before having to write the boiler plate Arcum code.

Patterns from Java Code. Another improvement that would help users develop patterns is a pattern generator. Such a tool would allow the user to select a set of expressions in a Java program, and from this set automatically generate a pattern that captures the structure of the selected expressions. Once the pattern is generated, the user would be able to observe the pattern's other matches too (beyond the selected expressions), and refine the pattern as needed. Such a pattern generator and tester would be useful in other AOSD environments.

Better Undo. My observations about the incremental development style show that experimentation is a useful form of feedback for refactoring tasks that involve crosscutting concepts. Furthermore, it is the ability to undo that gave programmers the chance to make changes they weren't certain about. Expanding the capabilities of undo could further lower the cost of experimentation. For example, the undo system could be extended into a light-weight, local revision control similar to repository systems. Such a system could also include the ability to create tags and save the undo history in the form of a tree (rather than a simple list).

5.5 Reasoning about Several Possibilities

One of the mental challenges of reasoning about refactoring lies in the need to conceptualize different versions of the same program, for example the version before the refactoring, and the version after. In the context of refactoring crosscutting concepts, the intellectual burden of tracking multiple possibilities is compounded further by the need to mentally account for the various crosscutting aspects of the program being refactored.

In my study, participants had to think about several versions of a program in two contexts: (1) they had to think about the program before and after the refactoring and (2) when performing checks, they had to think about both the correct program, and various possible incorrect versions of the program. I describe each in turn.

5.5.1 Thinking of Before and After

The most straightforward case where a developer has to conceptualize multiple versions of a program stems directly from the refactoring metaphor: an *original program* is transformed to a *refactored program*, and the developer must mentally model both of these programs when designing the refactoring.

While performing the refactoring manually, participants often kept the original code as comments in order to help them think about the before and after state of the program:

B2: I should have been commenting out the other stuff.

...

B2: [typing] list.next.put(n, result)... and now we can put list.get, right? I'm just gonna comment this out.

B1: OK, yeah.

B2: Cause I don't know if I've gotten this right.

When using Arcum, however, this kind of commenting was not necessary, since Arcum provides its own tools for the before-and-after metaphor, namely the `option` construct. There is evidence in the vocabulary used by the participants to indicate that they identified the `option` construct with the refactoring metaphor of before-and-after, for example:

A2: So nice. OK, and then we need [..]

A1: Two options

A2: Yeah one that will actually map what we have, and one that will map–match what we want.

Another Arcum tool that allowed participants to reason about their code in the before-and-after metaphor was Arcum's transformation preview pane, which showed the two different versions of the program side by side. The participants inspected the differences to get better confidence in their transformations. However, when the changes to be performed affected many files, sometimes the participants would only inspect a sampling of the files to see at least one example for each pattern. This suggests an opportunity to improve Arcum by adding to the preview window a summary of the transformations based on pattern coverage.

Despite the prevalence of the before-and-after metaphor, the goal of Arcum is not merely to be a refactoring tool. Whereas refactoring tools are often unidirectional, Arcum is meant to allow for switching between options seamlessly, regardless of the direction. Therefore, the notion of "before vs. after" becomes "one option vs. another option," where the options are not ordered in any way. Here again, the words used by the participants indicate that they understood the bi-directionality of Arcum, for example:

A1: Because certainly at this point we can just transform it back. Actually why don't we try transforming it back. Make sure it reverts properly. It should.

5.5.2 Thinking of Correct and Incorrect

Participants also had to think about multiple versions of the same program when they were writing additional checks using Arcum. These additional checks, which are performed continuously, capture the invariants necessary to ensure that all the options of a given interface are applicable all of the time.

My study shows evidence that writing proper checks to detect incorrect code is difficult. None of the three groups were able to complete the task of writing the check in the study, even though all the groups got close. For example, group A was in the process of devising one solution that would have caught only a subset of the possible errors. Had they finished the solution, it would have given them the false confidence that the check was being fully performed, when in fact it would only apply to a subset of the intended cases. A similar problem can occur in AspectJ: A '`declare warning`' applied to a pointcut that is improperly constructed creates the impression that a given property is fully checked, when instead only a subset of the cases are checked. These observations confirm that checks themselves need to be tested and debugged thoroughly, particularly because programmers rely on them to reason about the crosscutting in their programs.

Figure 5.3.A shows group C's code, which got the closest to the correct implementation, and Figure 5.3.B shows one correct solution. The only difference is the location of the predicate starting with '`init ==`.' If the predicate is placed in the `realize` clause, then it becomes an additional *pattern matching constraint*, which narrows the set of matches that are found (without ever generating an error message), whereas if it is placed in the `require` clause, it becomes a *checked constraint*, which gets checked *after* the pattern matching has been performed (and leads to an error message if violated). The participants did not make this distinction.

One possible way of characterizing the problem is that pattern matching is more

```
realize checkInit(Type owner, Field f, Expr init) {
  f == [private static final Logger logger = `init]
  && init == [Logger.getLogger(`owner.class.getName())]
  && hasField(owner, f);
}
```

(A)

```
realize checkInit(Type owner, Field f, Expr init) {
  f == [private static final Logger logger = `init]
  && hasField(owner, f);

  require "`init: The log file must use the class's name":
    init == [Logger.getLogger(`owner.class.getName())];
}
```

(B)

Figure 5.3: Implementations for `checkInit`: (A) The closest code written by any of the groups to check proper log initialization; and (B) The change necessary to make it correct: moving the conjunct into a `require`.

about the “before and after” metaphor, whereas the `require` clause is more about the “various incorrect versions” metaphor. The question then becomes: Did the participants simply not distinguish between these two metaphors? Or, did they distinguish between the metaphors, but were not able to figure out how to express the distinction in Arcum?

Looking at the word choices of the three groups, I conclude that the groups did in fact make the distinction, as shown in the following excerpt:

> *B1: It matched it but it didn’t tell us anything. So we need to do something that detects the error. So we have to capture this in a variable and check that it’s of that form. Or something like that. Or maybe not.*

Excerpts such as the one above lead me to conclude that the problem in fact lies with the participants not being able to *express* the distinction in Arcum, rather than not *seeing* the distinction. The root of this confusion may very well lie in the participants’ lack of experience with previous checking examples. However, another contributing factor may be the choice of keywords in the Arcum language: the words `realize` and `require`, unfortunately, do not reflect the metaphors that the participants were using when reasoning about `realize` and `require`. In particular, the metaphors used by participants were the “pattern matching” metaphor and the “error reporting” metaphor. Consequently, I conjecture that a better choice for the `realize` keyword is `match`, and a better choice for `require` is `check`, which, in addition to bringing the error metaphor into the keyword, also makes the temporal ordering of matching and error checking more clear.

A more general lesson could be drawn from my study about the choice of keywords in a language. Over the course of the project, I have many times debated what the best choice of keywords would be in Arcum, but I did not seriously look at the keywords from the point of view of the metaphors or models that a novice programmer might have in mind when thinking about the constructs. This metaphor-based approach to keyword selection provides a useful way of choosing keywords that could make languages more approachable to novices and experts alike.

5.6 Related Studies

Sillito et al. studied programming in Eclipse focused on the questions programmers ask when modifying programs [SMV06]. Part of the study used pair programming in order record conversations to be later analyzed. This analysis gave insights into how programmers understand a system and what they need to know in order to make modifications. Their study had a wide focus, intended to help guide the creation of software tools and tutorials, while my study was focused specifically as an evaluation of Arcum.

Robillard et al. investigated the process programmers use to understand code before they make changes to it and found that programmers who invested more time in making the most accurate model of the program were the most successful [RC04]. For example, the more lines of code a programmer examined (rather than skimmed) the higher the rate of success. Their study did not record the audio portion of programmer activities, and thus is was natural to use individual programmers instead of pairs of programmers. The focus of my study was the metaphors programmers use instead of a comparison of successful and unsuccessful programmers.

Ko et al. studied software changes performed in Eclipse and they found that much of the effort of reasoning about a maintenance task was navigating between scattered code dependencies and inspecting tangled code unrelated to the change [KAM05]. The kinds of program changes examined were either bug fixes or adding additional features to the program.

Storey et al. recognized in a large programmer study the different approaches programmers use to understand programs based on the different affordances available to them, and concluded that inspecting code dependencies was the most useful to programmers [SWM00].

Murphy et al. argue for the structure of crosscutting tasks to have a concrete representation in the IDE to guide further changes [MKRC05]. Arcum's approach for creating structure is through the definition of Arcum options when the software system itself does not (or cannot) modularize a design decision.

5.7 Design Recommendations

My user study shows that the Arcum approach to developing checks and refactorings for the crosscutting concepts in a program was natural to the programmers and that they could leverage their existing knowledge of modularity. However, the meta nature of Arcum code development carries difficulties. By observing the metaphors that the developers used while addressing these challenges I obtained a better understanding of the Arcum development processes. In doing so, I identify a few preliminary design recommendations to improve AOSD tools.

First, adding better undo functionality to current environments is a promising way to lower the costs of experimenting with design alternatives. For example, a tree-based undo history would allow developers to make multiple changes while allowing easy comparison, back and forth, among a set of options.

Second, keywords in programming languages should be made to match as closely as possible the metaphors that programmers will use in the development process. Choosing keywords in this way decreases the gap between the developer's mental model of programming idioms and how he or she expresses those idioms in the programming language.

Finally, environments for aspect-oriented software development should include tools for pattern testing, visualization and generation. These tools would help programmers by providing them with immediate feedback about their crosscutting queries.

Chapter 5, in whole, is a reprint of the material as it appears in The techniques programmers use to cope with crosscutting using Arcum. 2008. Shonle, M, Griswold, W., and Lerner, S. UCSD CS2008-0933, December 5, 2008. The dissertation author was the primary investigator and author of this paper.

Chapter 6

Future Work

The Arcum approach for the modular maintenance of crosscutting concepts can be extended into several lines of future work. The Arcum approach can be applied to a wide and rich source of software development experience (Section 6.1). Encoding such knowledge as reusable options can be a way to passively share programming knowledge from experts to novices.

The Arcum language itself can become more expressive and powerful with the addition of new language features (Section 6.2). In addition to language extensions, the Arcum approach can also benefit by having new operations available in the programming environment (Section 6.3).

6.1 Future Applications

The many Design Patterns identified after the seminal work by the Gang of Four [GHJV95] is a wide class of implementations techniques that could benefit from the kind of static checking Arcum provides. For example, the initialization of a Singleton instance can be incorrect in the context of a multi-threaded system (for example, the instance might become initialized twice). Checks can be written for these common error cases with transformations that serve as suggested fixes. Related work that addresses this problem includes Spine, a declarative language for checking design patterns [BBS05]. A wide awareness of different programming techniques and their interactions with other technologies will have increased importance as more technologies become available.

In general, any implementation style can benefit from extra checking. For example, internationalization strategies have rigid requirements for the ways string literals are used in programs (such as always wrapping them around method calls). A wide survey of these styles should be performed, with Arcum code written to check and enforce these rules. In the process, the potential exists later on for providing alternative implementations.

To complement such a large survey of practice, bug finding tools can be used as a rich source for common coding errors. Some of those tools are good for finding bugs that code reviews and test cases miss [RAF04]. A project that uses a specialized library could benefit by having an accompanying option perform similar checks. For example, performance bugs or other common errors can be detected, providing junior-level programmers with extra assistance and knowledge.

Product line architecture is a special case of an interesting design need. Under product line architecture, components can be used to describe a family of applications that may have, for example, different scalability or security needs. There are opportunities for Arcum to allow a mixing of implementation styles. Implementation requirements can be specified in the form easiest to express (for example, a Java field) and transformed at compile-time to the target product's needs (for example, a database access). A special case would be an in-house product line instrumented with performance measurement and debugging support that would not be part of the release version of the software. Any modifications to the software in the process of improving and measuring performance would automatically be applied to all versions.

6.2 Extensions to the Arcum Language

One of the research goals behind Arcum was to see how far a syntactic and type-based approach to code matching and generation could be applied. One obvious omission from the language was support for dataflow analysis constructs. Such dataflow constructs could be added to the language in one of two forms: (1) New primitives in the language for accessing common, pre-computed dataflow facts (e.g., if an object allocated at a particular site can escape its thread); and (2) A means for programmers to provide their own dataflow analysis constructs, either through a domain-specific language or through a programmable plug-in architecture. Additionally, the pattern matching of program fragments could benefit from dataflow analysis, such as matching equivalent code fragments beyond the basic desugaring done by Arcum's pattern matcher.

Dataflow analysis is available in refactoring systems similar to Arcum. For example, class library migration is an important problem addressed by Balaban et al. [BTF05]: One example is refactoring code using the old Java `Vector` class to use the more efficient `ArrayList` class. The `Vector` class is less efficient because all of its methods are synchronized by default, while the `ArrayList` class is only synchronized when explicitly requested. The Arcum methodology would be particularly well-suited for this task because the synchronization guarantees can be continuously checked. For example, if new code is written that allows an `ArrayList` instance to escape from the thread that created it, it would be flagged by the compiler as needing to be explicitly synchronized.

Currently, an Arcum option implements just a single interface, akin to single inheritance. However, just as design patterns can be hybridized, a single option could be used to implement two interfaces. For example, specifications for the mediator pattern and the observer pattern could be realized by a single option for the mediator-observer pattern, which could enforce that the mediator is also the observer.

Some implementations might coincidentally hybridize, causing unexpected interactions that would be caught during checking. For example, a visitor implementation and an observer implementation might coincidentally share participant code fragments. If the visitor code were to be refactored, it could violate a constraint of the observer code, triggering a constraint violation.

6.3 New Operations for the Environment

Operating in the other direction, a series of Arcum options can be queried against a program inorder for programmers to learn more about the program. *Concept mining* is a way to help programmers determine what crosscutting concepts may already be present. In particular, the problem of concept mining is to find (given an option library) the largest number of instantiation of the given options that can be found in a given program, thus automatically generating a mapping.

Concept mining would reduce the burden on the programmer by eliminating the need to write mappings in the first place. An automatically generated mapping also makes the task of identifying potential refactorings mechanical and therefore less error prone. Further still, a series of programs could be queried as a way to anticipate Arcum options: Common patterns could be identified, suggesting either new library abstractions or new Arcum options to develop.

Chapter 6, in part, is a reprint of the material as it appears in Beyond refactoring: a framework for modular maintenance of crosscutting design idioms. 2007. Shonle, M., Griswold, W., and Lerner, S. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering* (Dubrovnik, Croatia, September 03–07, 2007). ESEC-FSE '07. ACM, New York, NY, 175–184. The dissertation author was the primary investigator and author of this paper.

Chapter 7

Conclusion

One of the benefits of traditional class and method abstraction is modular substitution of their implementations. However, the implementations of some concepts are naturally crosscutting or are intentionally scattered across other code. Design patterns are typical examples.

Arcum expands the opportunities for modular analysis and substitution for such crosscutting concepts. Based on a paradigm of declarative pattern matching and substitution, Arcum specifications are declarative supplements to the program, neither modifying the code nor its behavior. Only the substitution process changes the code.

Arcum separates the behavior and implementation of a crosscutting concept into an interface and an option. An option uses semantic patterns that correspond to abstract concepts in the interface to provide a concrete implementation of the specification. Interfaces may be parameterized, supporting reuse and the development of Arcum refactoring libraries.

When the programmer uses a mapping to specify that a given option instantiation is expected to hold in the program, the Arcum engine can check this by matching the option's patterns over the program and then checking the matched elements against the interface's behavioral constraints. If the programmer specifies that a new, different option should now hold, the Arcum engine not only performs these checks for the old option, but then replaces the matched elements with the code specified in the patterns of the new option. Due to the declarative nature of the language, as well as the fact that the current option is continuously checked, the transformation process can be run in either direction.

My case study covered examples of crosscutting encountered “in the wild” and showed that such crosscutting can be managed through declarations written with Arcum. As more design patterns and programming idioms reach widespread use, the refactoring needs related to those crosscutting concepts can be anticipated by tool developers. However, I conjecture that real software has quirks and even very familiar idioms will not all be implemented in the same way. What is necessary is that these variants of a common idiom theme be implemented consistently. Once that step is taken, the implementation’s crosscutting nature can be managed through the use of tools like Arcum. Additionally, Arcum can help even when codebases have inconsistent implementations of idioms, because Arcum can express extra checks to catch non-conforming code, simplifying what sometimes must be a manual process.

7.1 Contributions

The primary contribution of this dissertation is an extension to the programming environment that allows for some of the benefits of modularity to be extended to crosscutting programming concepts. To achieve this, several other contributions were necessary, or naturally followed:

- The Arcum approach unifies refactoring and program checking, so that the knowledge a programmer shares with the programming environment for one case can be preserved and applied for multiple purposes. Such a unification has symmetrical benefits: the programmer can be primarily specifying a transformation, but as a side-effect benefit from additional checking; or the programmer can be primarily specifying a check for one particular implementation, leaving the program structured enough to be transformed into using alternative implementations in the future.
- Arcum provides an additional means for the programmer’s conception of the program to be expressed. Not only does this give programmers the opportunity to document design decisions (and their alternatives) when a modular solution is not

possible, it also enables automatic checking of implementations, some of which may be specific to only a small family of programs. Such checking constrains what kind of operations can be performed: For example, the `private` access specifier in the Java programming language can be too permissive, allowing more methods access to a data field than necessary. Such permissiveness can be reined in through additional checks, reducing the likelihood of programming errors.

- In addition to limiting the programming language when necessary, the Arcum approach also enhances the programming language when it is not powerful enough. There are often programming trade-offs between expressiveness and efficiency (such as the many dynamic/static trade-offs in Java), and the Arcum approach enables the best of both worlds: The program can be written in the more expressive form for the current task, and then effortlessly transformed into the more efficient form when necessary.

- A preliminary user study of Arcum has demonstrated the feasibility of the Arcum approach, showing that the Arcum language can be use effectively for reasoning about several different concepts in isolation.

- The presence of a system like Arcum can assist with the implementation of cross-cutting concepts and thus creates more opportunities to use advanced programming techniques. By reducing such costs Arcum can create more opportunities for using design patterns and programming idioms—such as defensive programming (e.g. making code easier to debug) and reflection. This is accomplished through a technique that is an alternative to aspect-oriented programming [KH01], which is different by not requiring the programmer to switch programming languages. Programmers do not have to commit to these advanced techniques, because they can retain the option to refactor the program back to alternative implementations.

Appendix A

Syntax, Types and Built-in Predicates

This appendix presents a grammar for the Arcum language (Section A.1) and a reference for Arcum's types and built-in predicates (Section A.2).

A.1 Syntax

Arcum's lexical structure is a superset of Java's, allowing for the backtick character ('`') to be recognized. Some of Arcum's syntactic structure embeds portions of Java code. Given such constructs, the terminals and non-terminals that are used, but not defined, here can be found in Chapters 3 and 18 of *The Java Language Specification, Third Edition* [GJSB05].[1] Following that book's conventions:

- $[x]$ denotes zero or one occurrences of x
- $[x]^*$ denotes zero or more occurrences of x
- $x \mid y$ denotes one of either x or y

```
ArcumCompilationUnit ::= [ImportDeclaration]* [ArcumDeclaration]*

ArcumDeclaration ::=
   InterfaceDeclaration
 | OptionDeclaration
 | MappingDeclaration
```

[1]The rules imported from Java are: *ImportDeclaration*, *Identifier*, *Type*, *StringLiteral*, and *Literal*.

```
InterfaceDeclaration ::=
   interface Identifier [InterfaceParams] { [InterfaceMember]* }

InterfaceParams ::= ( ) | ( InterfaceFormal [, InterfaceFormal]* )

InterfaceFormal ::=
   Type Identifier [: Expr] [default Expr]
 | Identifier Formals [: Expr] [default Expr]

InterfaceMember ::=
   AbstractConcept
 | PredicateDefinition
 | StandaloneConstraint

AbstractConcept ::=
   abstract Type Identifier ConceptConstraints
 | abstract Identifier Formals ConceptConstraints

ConceptConstraints ::=
   ;
 | { Expr [CheckClause]* }

CheckClause ::= check [StringLiteral] { Expr }

Formals ::= ( [FormalsList] )

FormalsList ::= Type Identifier [, Type Identifier]*

PredicateDefinition ::= define Identifier Formals { Expr }

StandaloneConstraint ::= check [StringLiteral] { Expr }

OptionDeclaration ::=
   option Identifier implements Identifier { [OptionMember]* }

OptionMember ::=
   MatchDeclaration
 | PredicateDefinition
 | StandaloneConstraint

MatchDeclaration ::=
   match Identifier Formals { Expr }
 | match FormalsList { Expr } [OnFailClause]

OnFailClause ::= onfail { StringLiteral [, Identifier] }
```

```
MappingDeclaration ::= check { [MappingOrPredicateDefinition]* }

MappingOrPredicateDefinition ::= Mapping | PredicateDefinition

Mapping ::= Identifier OptionArguments ;

OptionArguments ::= ( ) | ( NameValuePair [, NameValuePair]* )

NameValuePair ::=
   Identifier : Literal
 | Identifier : Type
 | Identifier : Identifier
 | Identifier : Type . Identifier

Expr ::= Term | Expr || Term

Term ::= Factor | Term && Factor

Factor ::=
   Identifier ( PredicateArgument [, PredicateArgument]* )
 | Identifier == VariableValue
 | ! Factor
 | ( Expr [<=> Expr]* )
 | ExistsExpression
 | ForallExpression
 | true
 | false

PredicateArgument ::=
   _
 | Identifier
 | Identifier ?
 | ImmediatePattern

VariableValue ::=
   Identifier
 | SelectExpression
 | Pattern
 | ( Pattern [|| Pattern]* )

ExistsExpression ::= exists QuantifiedVars { Expr }

ForallExpression ::= forall QuantifiedVars { Expr [OnFailClause] }

QuantifiedVars ::= ( FormalsList : Expr )
```

SelectExpression ::=
  select { *SelectCase* [, *SelectCase*]* **default :** *VariableValue* **}**

SelectCase ::= *Expr* **:** *VariableValue*

Pattern ::= *SearchPattern* | *ImmediatePattern*

EmbeddedArcumExpression ::=
  ` [*OrderingSpecification* :] **[** *Type Identifier* **:** *Expr* **]**
 | *UnquotedVariable*

OrderingSpecification ::= **anyOrder** | **strictOrder**

UnquotedVariable ::=
 | **`** *Identifier*
 | **`** _
 | **`** ...

A *SearchPattern* is a sequence of Java tokens with matched brackets and parentheses contained within a pair of square brackets ([...]). Identifiers in the sequence can be preceded with a backtick "`", which is the escape mechanism used to refer to Arcum variables. The special variable "_" matches anything, and the special variable "..." matches any sequence of constructs (such as arguments in a method call, or statements in a block). An *ImmediatePattern* follows the same rules as *SearchPattern*, except the sequence is contained within a pair of angle brackets (<...>). A *SearchPattern* may have *EmbeddedArcumExpression*s within it, but an *ImmediatePattern* can only have *UnquotedVariable*s.

A.2 Types and Built-In Predicates

Table A.1 lists the types available in Arcum for matching against program fragments. Once such program fragments are matched, their properties can be explored using the predicates specified in Table A.2.

Table A.1: Arcum Types

Type	Program Fragment
AccessSpecifier	A subtype of Modifiers: One of either public, private, protected, or the default ‘package access.’
Annotation	A Java metadata annotation.
DeclarationElement	The declaration of a local variable, field, return type, or a cast expression. Specified as a type.
Expr	An expression.
Field	A subtype of DeclarationElement: A field.
Method	A method.
Modifiers	A possibly empty set of modifiers that are applied to methods, classes, and fields (e.g., static, public, final)
Name	A name for a type, method, variable, or package.
Signature	The name and parameter types of a method.
Statement	A statement.
Type	A Java type, either primitive or reference.

Table A.2: Built-In Arcum Predicates. The following abbreviations are used for types: *A: Annotation, D: DeclarationElement, E: Expr, F: Field, M: Method, N: Name, S: Signature, T: Type, Any: (any type).*

Predicate	Meaning
copiedTo(E *e*, D *d*)	The value of expression *e* is copied to a location declared by the declaration element *d*. Copy operations include: assignment, initialization, argument passing, and value returning.
declaredBy(E *e*, D *d*)	The static type of *e* is determined by declaration *d*.
hasAnnotation(T|M|D *p*, A *a*)	The program fragment *p* is marked with the annotation *a*.
hasField(T *t*, F *f*)	Type *t* has field *f* as a member.
hasInvocationTarget(E *e*, E *t*)	Expression *e* is a method invocation, and *t* is the target of the invocation. This can be pattern matched with: `e == [`t.`_(`...)]`
hasMethod(T *t*, M *m*)	Type *t* has method *m* as a member.
hasSignature(T *t*, S *s*)	Type *t* has a method or abstract method with signature *s*.
invokes(E|M *p*, M *m*)	The expression or method *p* invokes method *m*.
isA(E|T *p*, T *t*)	The type of expression *p*, or the type *p*, is equal to or a subtype of type *t*.
isAbstract(F|M|T *p*)	The program fragment *p* is abstract.
isAnnotationType(T *t*)	The type *t* is an annotation type.
isClass(T *t*)	The type *t* is a class.
isEnum(T *t*)	The type *t* is an enum.
isExpressionStatement(E *e*)	The expression *e* is the contents of a statement.
isFinal(F|M|T *p*)	The program fragment *p* is final.
isInterface(T *t*)	The type *t* is an interface.
isPublic(F|M|T *p*)	The program fragment *p* is public.
isPrivate(F|M|T *p*)	The program fragment *p* is private.
isProtected(F|M|T *p*)	The program fragment *p* is protected.
isQualifiedName(N *n*)	The name *n* is a qualified name.
isReferenceType(T *t*)	The type *t* is a reference type.
isSimpleAssignment(E *e*)	The expression *e* is a non-compound assignment operation.
isSimpleName(N *n*)	The name *n* is a simple name.
isStatic(F|M|T *p*)	The program fragment *p* is static.
isSynchronized(M *m*)	Method *m* is synchronized.
isTransient(F *f*)	Field *f* is transient.
within(Any *a*, Any *b*)	The text for program fragment *a* is nested within the text for program fragment *b*.

Appendix B

Visitor Concept Implementation

```
import edu.ucsd.mshonle.*;
import com.google.inject.TypeLiteral;

interface VisitorConcept(
        Name traversalName,
        Type visitorInterface : isInterface(visitorInterface),
        Type rootType : isClass(rootType),
        targetType(Type type),
        viaEdge(Field edge) default isField(edge),
        bypassEdge(Field edge) default false)
{
    check {
        forall (Type t : targetType(t)) {
            hasSignature(visitorInterface, <public boolean visit(`t `_)>)
            onfail {"Missing visit method of type `t", visitorInterface}
        }
        && forall (Signature s : hasSignature(visitorInterface, s)) {
            s == <public boolean visit(`t `_)>
            && targetType(t)
            onfail {"Spurious visit method of type `t", s}
        }
    }

    abstract visit(Expr root, Expr target, Expr visitor) {
        check "The target must be of type `rootType" {
            isA(target, rootType)
        }
        check "The visitor must be of type `visitorInterface" {
            isA(visitor, visitorInterface)
        }
    }

    // Relation holds when instances of ’t’ have an instance of ’u’
    // via some field ’field’
    define hasA(Type t, Type u, Field field) {
        hasField(t, field)
        && !isStatic(field)
        && pointsToA(field, u)
        || exists (Type v : hasA(t, v, field)) { isSubtypeOf(v, u) }
    }

    define pointsToA(Field f, Type t) {
        declaredAs(f, t) ||
        exits (Type u : pointsToA(f, u)) {
            u == <‘t[]>
            || (isSubtypeOf(listType?, <Collection>)
                && u == <‘listType<‘t> >)
        }
    }

    define classGraph(Type fromType, Type toType, Field edge) {
        ((fromType == rootType) || classGraph(_, fromType, _))
        && hasA(fromType, toType, edge) && viaEdge(edge)
    }

    define traversalGraph(Type fromType, Type toType, Field edge) {
        classGraph(fromType, toType, edge) && !bypassEdge(edge)
        && (targetType(toType) || traversalGraph(toType, _, _))
    }
}

option GoFVisitor implements VisitorConcept {
    realize visit(Expr root, Expr target, Expr visitor) {
        root == [‘target.‘traversalName(‘visitor)] && isA(target, rootType)
        && !exists (Method m : acceptMethod(m, _)) { within(root, m) }
    }

    realize acceptMethod(Method m, Type c) {
        traversalGraph(c, _, _) && isClass(c) && hasMethod(c, m)
        && m == [public void ‘traversalName(‘visitorInterface visitor) {
            ‘strictOrder:[Statement s : acceptMethodStmt(c, _, s)]
        }]
    }

    realize acceptSignature(Signature s, Type i) {
        traversalGraph(i, _, _) && isInterface(i) && hasSignature(i, s)
        && s == [public void `traversalName(`visitorInterface visitor)]
    }

    define acceptMethodStmt(Type fromType, Field edge, Statement stmt) {
        traversalGraph(fromType, toType?, edge) && isClass(fromType)
        && stmt == select {
            targetType(toType) && traversalGraph(toType, _, _):
                <if (visitor.visit(this.`edge)) {
                     this.`edge.`traversalName(visitor);
                 }>
            targetType(toType) && !traversalGraph(toType, _, _):
                <visitor.visit(this.`edge);>,
            default:
                <this.`edge.`traversalName(visitor);>
        }
    }
}

option DJLibrary implements VisitorConcept {
    match Field strategy, Expr init {
        init ==
            [Strategy.create(new TypeLiteral<`rootType>() {})
                .targets(`anyOrder:[Expr e :
                  targetType(t?) && e == <new TypeLiteral<`t>() {}>])]
        && strategy == [public static Strategy `traversalName = `init]
        && hasField(rootType, strategy)
    } onfail {"Must have a static field named `traversalName", rootType}

    match visit(Expr root, Expr target, Expr visitor) {
        root == [`rootType.`strategy.traverse(`target, `visitor)]
    }
}

check {
    GoFVisitor(
        traversalName: visitBooks,
        visitorInterface: IBookVisitorWithDJ,
        rootType: LibraryWithDJ,
        targetType(Type type): publicationTargets);

    define publicationTargets(Type t)
        type == (<BookWithDJ> || <PaperWithDJ> || <MakeWithDJ>)

}
```

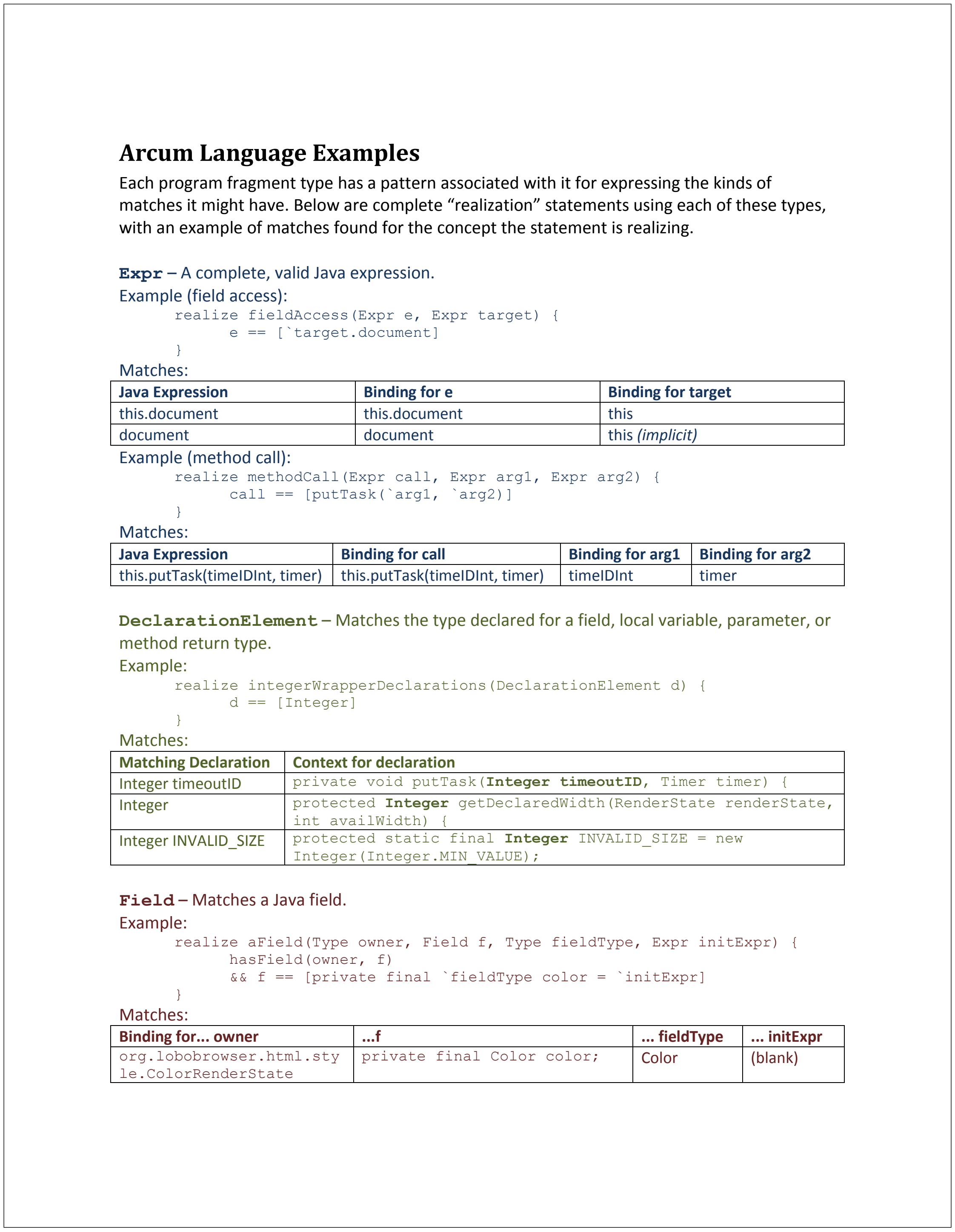
Appendix C

User Study Materials

I provided hard copies of user study instructions and language reference materials to all participants in the user study. Figure C.1 shows examples of using the Arcum types necessary to complete the study tasks, and Figure C.2 is a brief language reference.

Figures C.3–C.9 are the pages of the instructions from the first day of the user study, and Figures C.10–C.11 are the pages of the instructions from the second day of the user study.

All participants were requested to fill out a questionnaire form (Figures C.12–C.13) in order to find out each participant's previous experience with programming languages, development environments, and design patterns.



Arcum Language Examples

Each program fragment type has a pattern associated with it for expressing the kinds of matches it might have. Below are complete "realization" statements using each of these types, with an example of matches found for the concept the statement is realizing.

`Expr` – A complete, valid Java expression.

Example (field access):

```
	realize fieldAccess(Expr e, Expr target) {
		e == [`target.document]
	}
```

Matches:

Java Expression	Binding for e	Binding for target
this.document	this.document	this
document	document	this *(implicit)*

Example (method call):

```
	realize methodCall(Expr call, Expr arg1, Expr arg2) {
		call == [putTask(`arg1, `arg2)]
	}
```

Matches:

Java Expression	Binding for call	Binding for arg1	Binding for arg2
this.putTask(timeIDInt, timer)	this.putTask(timeIDInt, timer)	timeIDInt	timer

`DeclarationElement` – Matches the type declared for a field, local variable, parameter, or method return type.

Example:

```
	realize integerWrapperDeclarations(DeclarationElement d) {
		d == [Integer]
	}
```

Matches:

Matching Declaration	Context for declaration
Integer timeoutID	`private void putTask(`**`Integer timeoutID`**`, Timer timer) {`
Integer	`protected `**`Integer`**` getDeclaredWidth(RenderState renderState, int availWidth) {`
Integer INVALID_SIZE	`protected static final `**`Integer`**` INVALID_SIZE = new Integer(Integer.MIN_VALUE);`

`Field` – Matches a Java field.

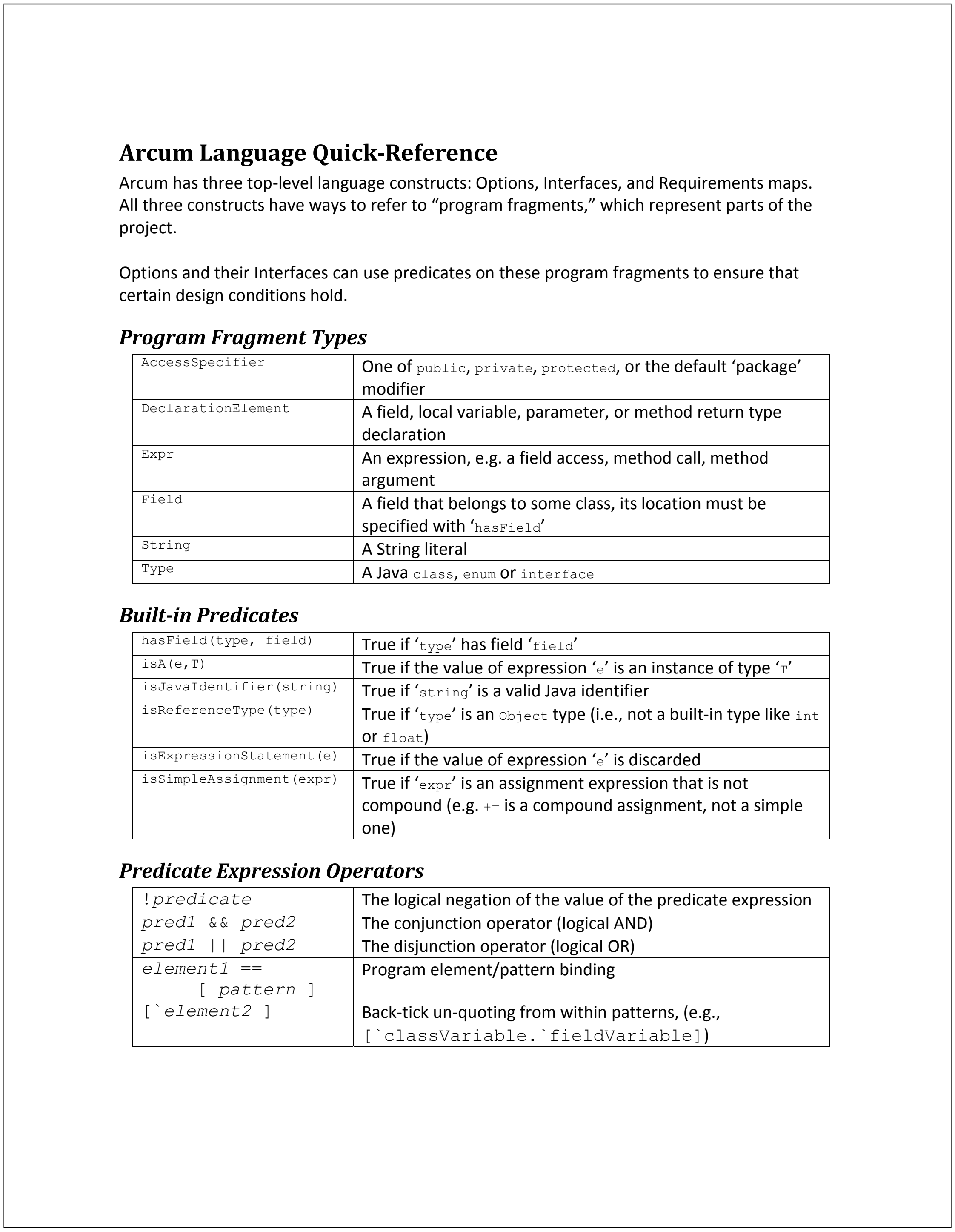
Example:

```
	realize aField(Type owner, Field f, Type fieldType, Expr initExpr) {
		hasField(owner, f)
		&& f == [private final `fieldType color = `initExpr]
	}
```

Matches:

Binding for... owner	...f	... fieldType	... initExpr
`org.lobobrowser.html.style.ColorRenderState`	`private final Color color;`	Color	(blank)

Figure C.1: Language examples document provided to users.



Arcum Language Quick-Reference

Arcum has three top-level language constructs: Options, Interfaces, and Requirements maps. All three constructs have ways to refer to "program fragments," which represent parts of the project.

Options and their Interfaces can use predicates on these program fragments to ensure that certain design conditions hold.

Program Fragment Types

`AccessSpecifier`	One of `public`, `private`, `protected`, or the default 'package' modifier
`DeclarationElement`	A field, local variable, parameter, or method return type declaration
`Expr`	An expression, e.g. a field access, method call, method argument
`Field`	A field that belongs to some class, its location must be specified with '`hasField`'
`String`	A String literal
`Type`	A Java `class`, `enum` or `interface`

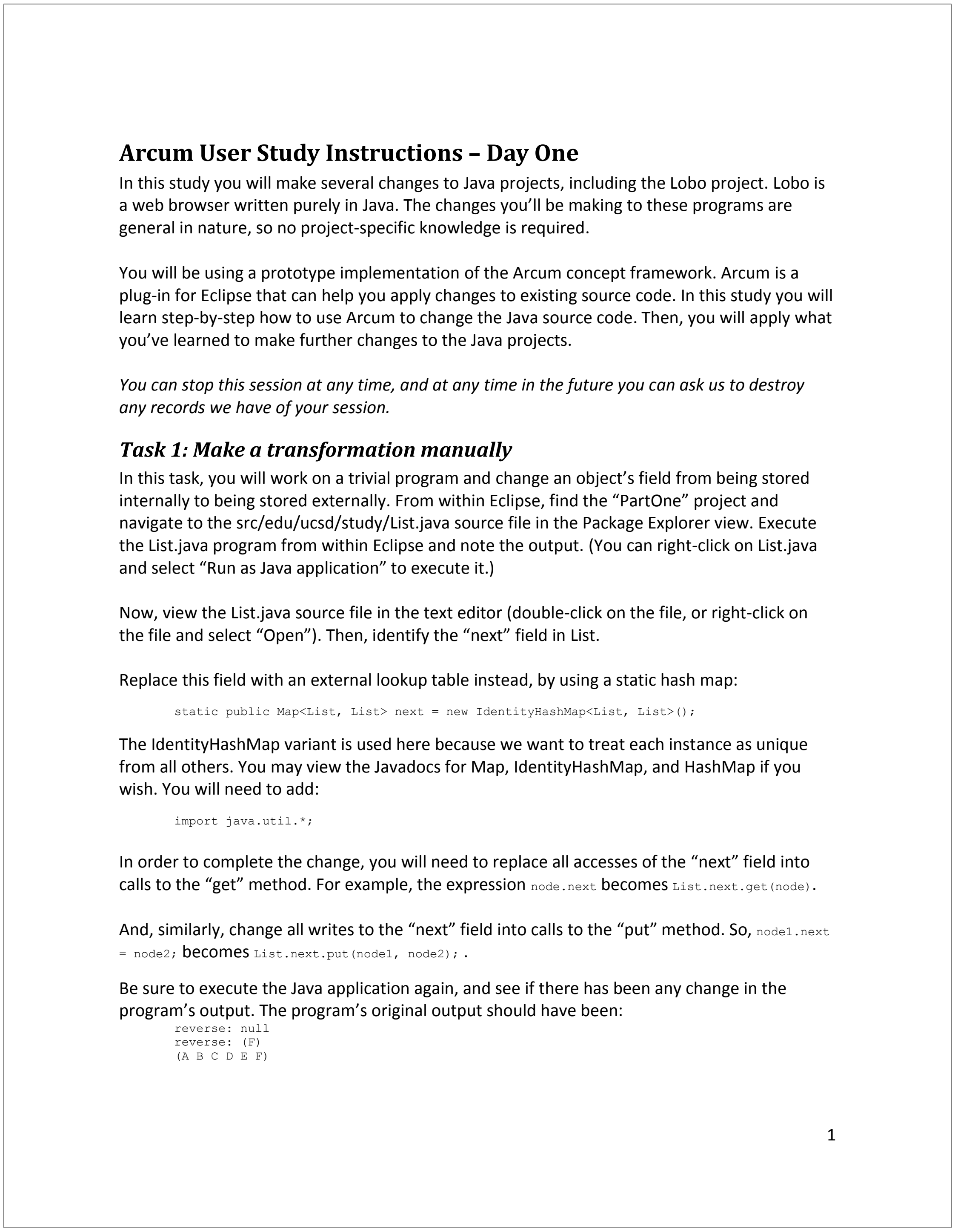
Built-in Predicates

`hasField(type, field)`	True if '`type`' has field '`field`'
`isA(e,T)`	True if the value of expression '`e`' is an instance of type '`T`'
`isJavaIdentifier(string)`	True if '`string`' is a valid Java identifier
`isReferenceType(type)`	True if '`type`' is an `Object` type (i.e., not a built-in type like `int` or `float`)
`isExpressionStatement(e)`	True if the value of expression '`e`' is discarded
`isSimpleAssignment(expr)`	True if '`expr`' is an assignment expression that is not compound (e.g. `+=` is a compound assignment, not a simple one)

Predicate Expression Operators

!*predicate*	The logical negation of the value of the predicate expression
pred1 && *pred2*	The conjunction operator (logical AND)
pred1 || *pred2*	The disjunction operator (logical OR)
element1 == [*pattern*]	Program element/pattern binding
[`*element2*]	Back-tick un-quoting from within patterns, (e.g., [`classVariable.`fieldVariable])

Figure C.2: Language Reference



Arcum User Study Instructions – Day One

In this study you will make several changes to Java projects, including the Lobo project. Lobo is a web browser written purely in Java. The changes you'll be making to these programs are general in nature, so no project-specific knowledge is required.

You will be using a prototype implementation of the Arcum concept framework. Arcum is a plug-in for Eclipse that can help you apply changes to existing source code. In this study you will learn step-by-step how to use Arcum to change the Java source code. Then, you will apply what you've learned to make further changes to the Java projects.

You can stop this session at any time, and at any time in the future you can ask us to destroy any records we have of your session.

Task 1: Make a transformation manually

In this task, you will work on a trivial program and change an object's field from being stored internally to being stored externally. From within Eclipse, find the "PartOne" project and navigate to the src/edu/ucsd/study/List.java source file in the Package Explorer view. Execute the List.java program from within Eclipse and note the output. (You can right-click on List.java and select "Run as Java application" to execute it.)

Now, view the List.java source file in the text editor (double-click on the file, or right-click on the file and select "Open"). Then, identify the "next" field in List.

Replace this field with an external lookup table instead, by using a static hash map:

```
static public Map<List, List> next = new IdentityHashMap<List, List>();
```

The IdentityHashMap variant is used here because we want to treat each instance as unique from all others. You may view the Javadocs for Map, IdentityHashMap, and HashMap if you wish. You will need to add:

```
import java.util.*;
```

In order to complete the change, you will need to replace all accesses of the "next" field into calls to the "get" method. For example, the expression `node.next` becomes `List.next.get(node)`.

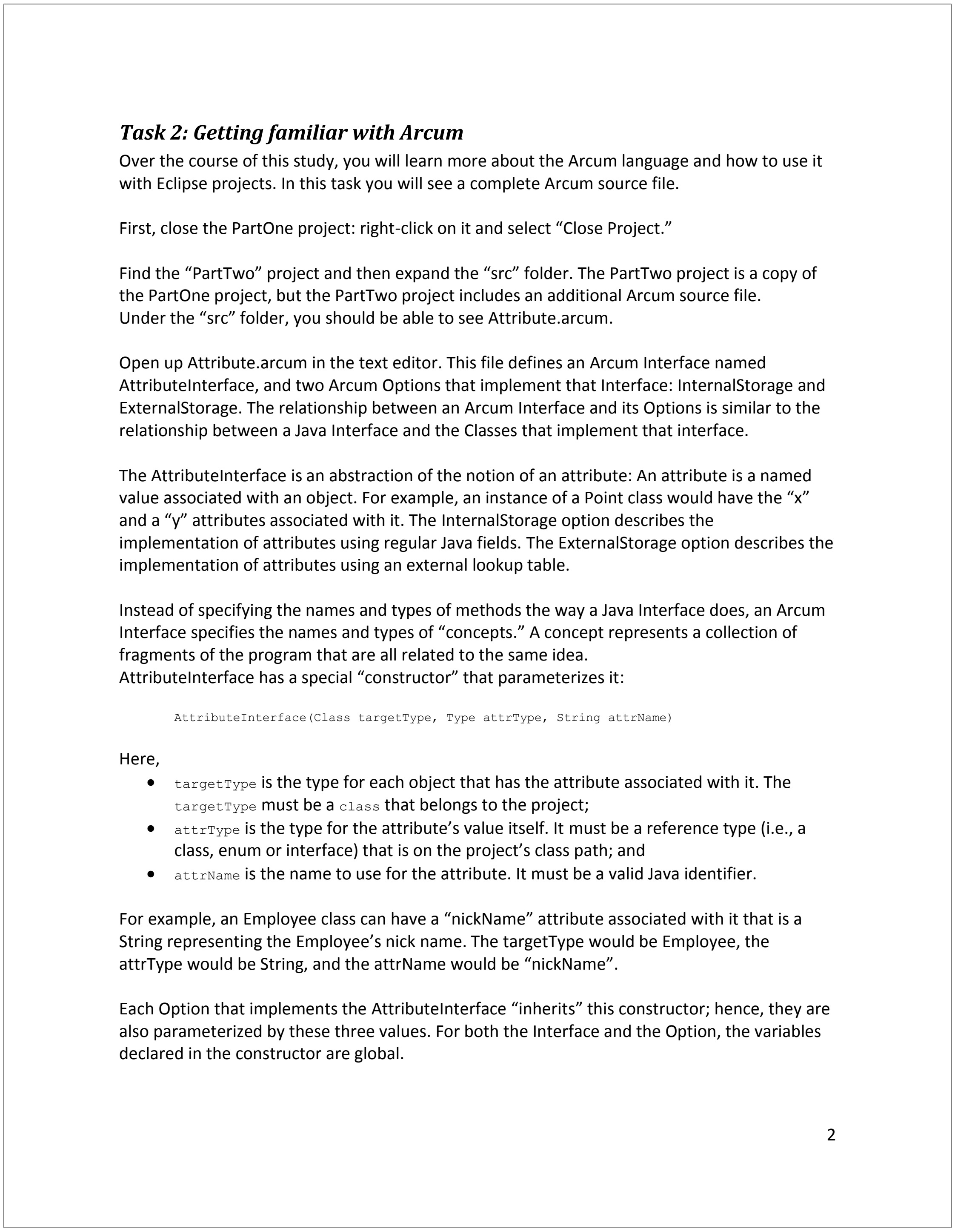
And, similarly, change all writes to the "next" field into calls to the "put" method. So, `node1.next = node2;` becomes `List.next.put(node1, node2);` .

Be sure to execute the Java application again, and see if there has been any change in the program's output. The program's original output should have been:

```
reverse: null
reverse: (F)
(A B C D E F)
```

1

Figure C.3: Arcum User Study Instructions — Day One, page 1



Task 2: Getting familiar with Arcum

Over the course of this study, you will learn more about the Arcum language and how to use it with Eclipse projects. In this task you will see a complete Arcum source file.

First, close the PartOne project: right-click on it and select “Close Project.”

Find the “PartTwo” project and then expand the “src” folder. The PartTwo project is a copy of the PartOne project, but the PartTwo project includes an additional Arcum source file.
Under the “src” folder, you should be able to see Attribute.arcum.

Open up Attribute.arcum in the text editor. This file defines an Arcum Interface named AttributeInterface, and two Arcum Options that implement that Interface: InternalStorage and ExternalStorage. The relationship between an Arcum Interface and its Options is similar to the relationship between a Java Interface and the Classes that implement that interface.

The AttributeInterface is an abstraction of the notion of an attribute: An attribute is a named value associated with an object. For example, an instance of a Point class would have the “x” and a “y” attributes associated with it. The InternalStorage option describes the implementation of attributes using regular Java fields. The ExternalStorage option describes the implementation of attributes using an external lookup table.

Instead of specifying the names and types of methods the way a Java Interface does, an Arcum Interface specifies the names and types of “concepts.” A concept represents a collection of fragments of the program that are all related to the same idea.
AttributeInterface has a special “constructor” that parameterizes it:

```
AttributeInterface(Class targetType, Type attrType, String attrName)
```

Here,

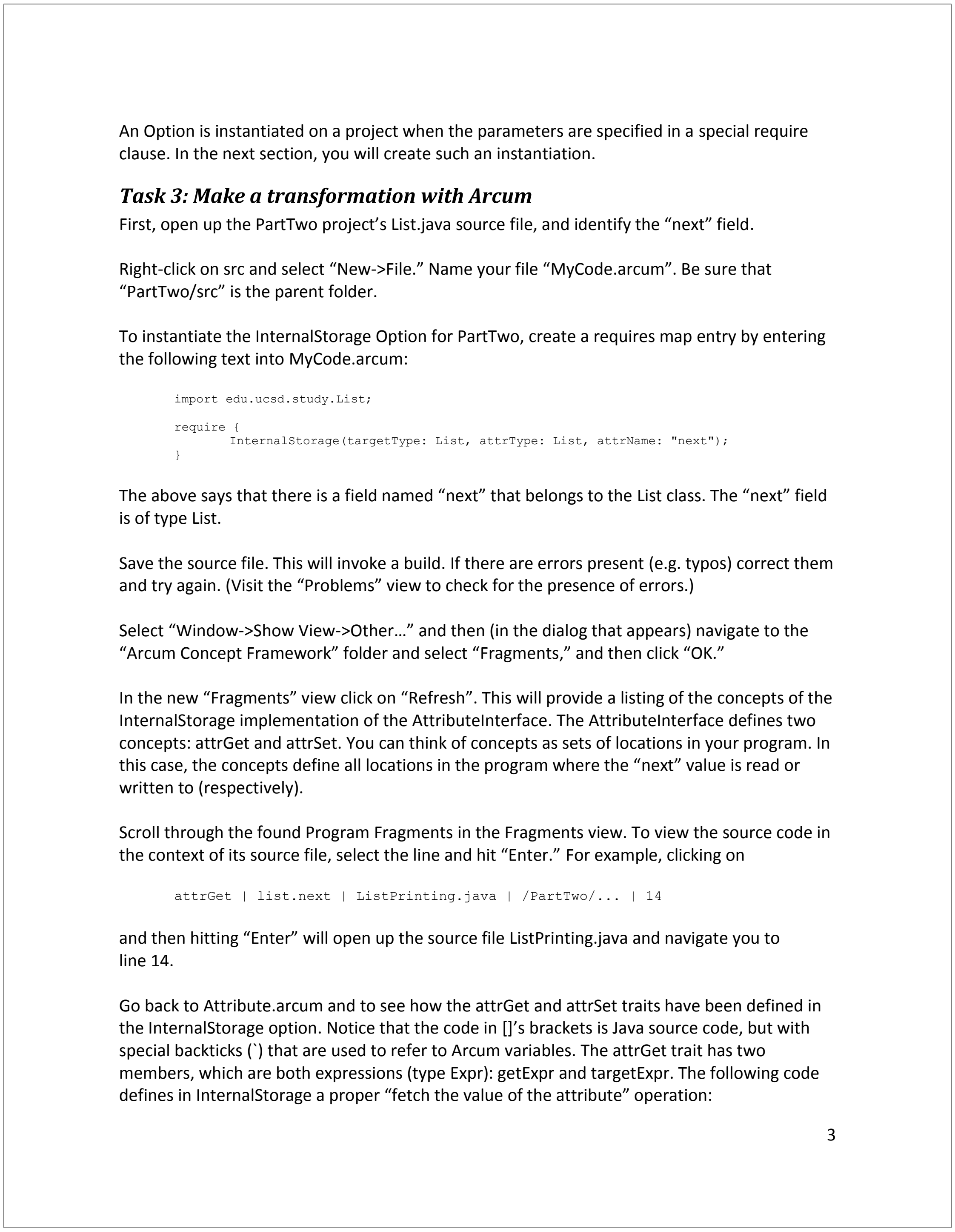
- `targetType` is the type for each object that has the attribute associated with it. The `targetType` must be a `class` that belongs to the project;
- `attrType` is the type for the attribute’s value itself. It must be a reference type (i.e., a class, enum or interface) that is on the project’s class path; and
- `attrName` is the name to use for the attribute. It must be a valid Java identifier.

For example, an Employee class can have a “nickName” attribute associated with it that is a String representing the Employee’s nick name. The targetType would be Employee, the attrType would be String, and the attrName would be “nickName”.

Each Option that implements the AttributeInterface “inherits” this constructor; hence, they are also parameterized by these three values. For both the Interface and the Option, the variables declared in the constructor are global.

2

Figure C.4: Arcum User Study Instructions — Day One, page 2



An Option is instantiated on a project when the parameters are specified in a special require clause. In the next section, you will create such an instantiation.

Task 3: Make a transformation with Arcum

First, open up the PartTwo project's List.java source file, and identify the "next" field.

Right-click on src and select "New->File." Name your file "MyCode.arcum". Be sure that "PartTwo/src" is the parent folder.

To instantiate the InternalStorage Option for PartTwo, create a requires map entry by entering the following text into MyCode.arcum:

```
import edu.ucsd.study.List;

require {
       InternalStorage(targetType: List, attrType: List, attrName: "next");
}
```

The above says that there is a field named "next" that belongs to the List class. The "next" field is of type List.

Save the source file. This will invoke a build. If there are errors present (e.g. typos) correct them and try again. (Visit the "Problems" view to check for the presence of errors.)

Select "Window->Show View->Other…" and then (in the dialog that appears) navigate to the "Arcum Concept Framework" folder and select "Fragments," and then click "OK."

In the new "Fragments" view click on "Refresh". This will provide a listing of the concepts of the InternalStorage implementation of the AttributeInterface. The AttributeInterface defines two concepts: attrGet and attrSet. You can think of concepts as sets of locations in your program. In this case, the concepts define all locations in the program where the "next" value is read or written to (respectively).

Scroll through the found Program Fragments in the Fragments view. To view the source code in the context of its source file, select the line and hit "Enter." For example, clicking on

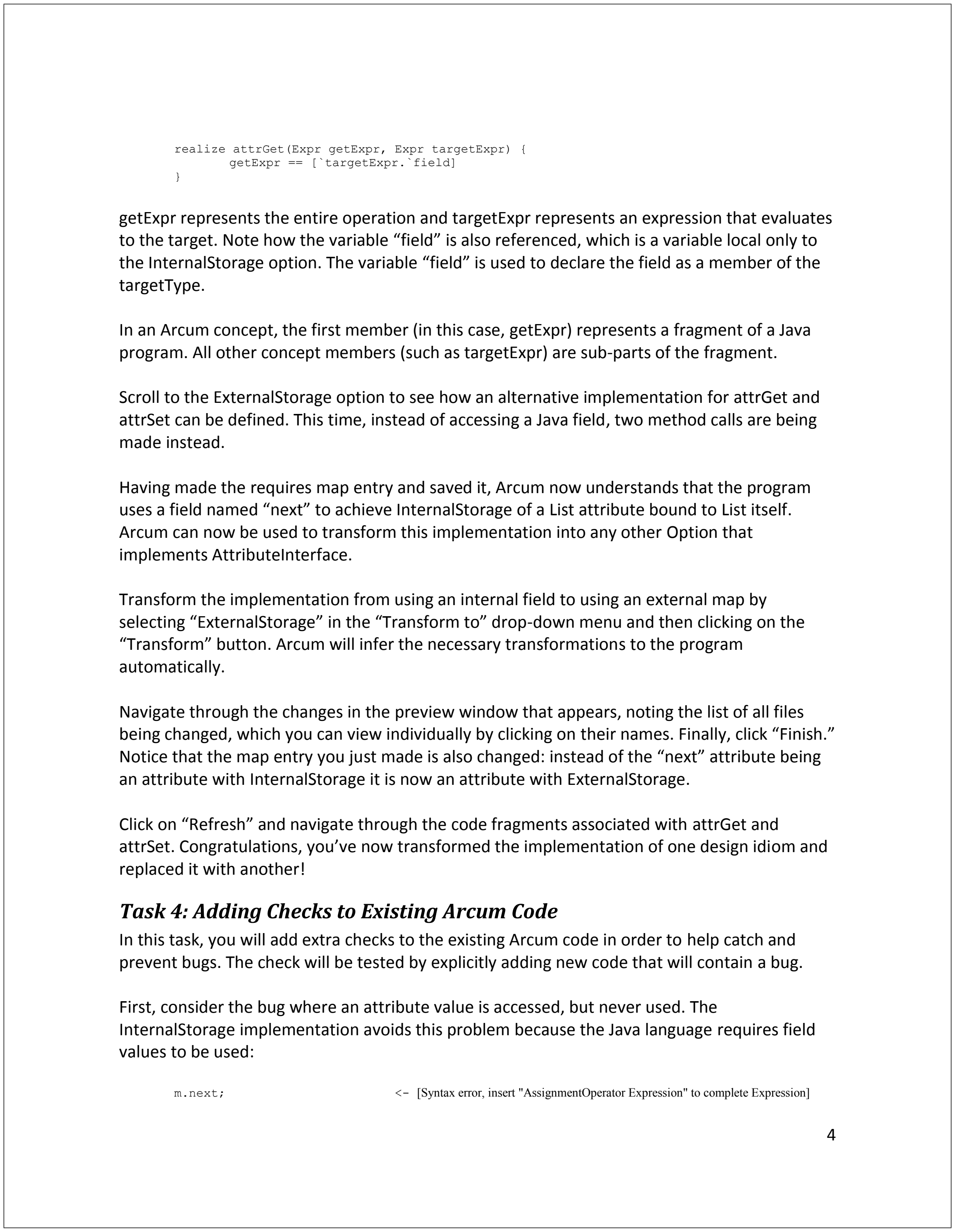
```
attrGet | list.next | ListPrinting.java | /PartTwo/... | 14
```

and then hitting "Enter" will open up the source file ListPrinting.java and navigate you to line 14.

Go back to Attribute.arcum and to see how the attrGet and attrSet traits have been defined in the InternalStorage option. Notice that the code in []'s brackets is Java source code, but with special backticks (`) that are used to refer to Arcum variables. The attrGet trait has two members, which are both expressions (type Expr): getExpr and targetExpr. The following code defines in InternalStorage a proper "fetch the value of the attribute" operation:

3

Figure C.5: Arcum User Study Instructions — Day One, page 3



```
realize attrGet(Expr getExpr, Expr targetExpr) {
        getExpr == [`targetExpr.`field]
}
```

getExpr represents the entire operation and targetExpr represents an expression that evaluates to the target. Note how the variable "field" is also referenced, which is a variable local only to the InternalStorage option. The variable "field" is used to declare the field as a member of the targetType.

In an Arcum concept, the first member (in this case, getExpr) represents a fragment of a Java program. All other concept members (such as targetExpr) are sub-parts of the fragment.

Scroll to the ExternalStorage option to see how an alternative implementation for attrGet and attrSet can be defined. This time, instead of accessing a Java field, two method calls are being made instead.

Having made the requires map entry and saved it, Arcum now understands that the program uses a field named "next" to achieve InternalStorage of a List attribute bound to List itself. Arcum can now be used to transform this implementation into any other Option that implements AttributeInterface.

Transform the implementation from using an internal field to using an external map by selecting "ExternalStorage" in the "Transform to" drop-down menu and then clicking on the "Transform" button. Arcum will infer the necessary transformations to the program automatically.

Navigate through the changes in the preview window that appears, noting the list of all files being changed, which you can view individually by clicking on their names. Finally, click "Finish." Notice that the map entry you just made is also changed: instead of the "next" attribute being an attribute with InternalStorage it is now an attribute with ExternalStorage.

Click on "Refresh" and navigate through the code fragments associated with attrGet and attrSet. Congratulations, you've now transformed the implementation of one design idiom and replaced it with another!

Task 4: Adding Checks to Existing Arcum Code

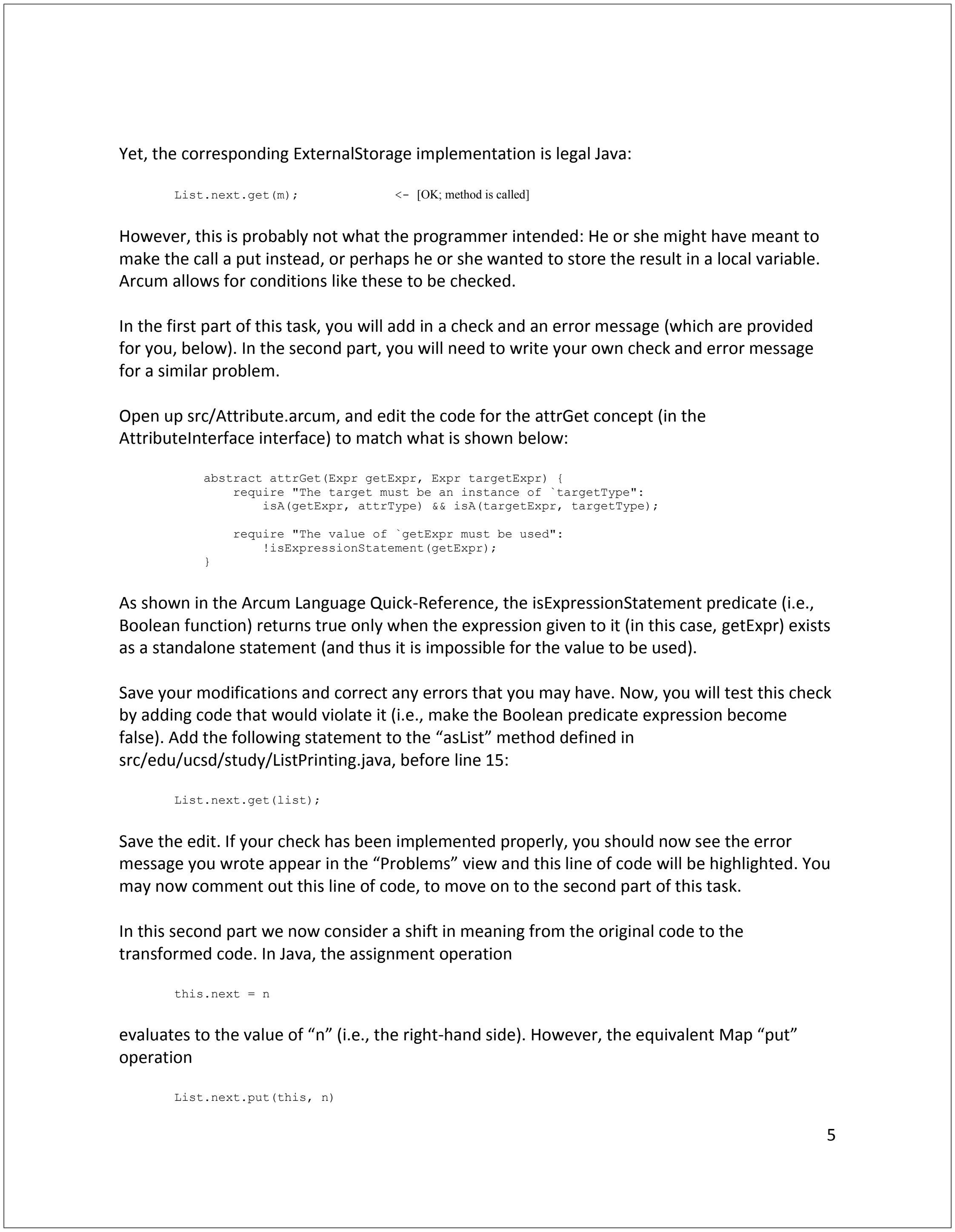
In this task, you will add extra checks to the existing Arcum code in order to help catch and prevent bugs. The check will be tested by explicitly adding new code that will contain a bug.

First, consider the bug where an attribute value is accessed, but never used. The InternalStorage implementation avoids this problem because the Java language requires field values to be used:

```
m.next;                    <- [Syntax error, insert "AssignmentOperator Expression" to complete Expression]
```

4

Figure C.6: Arcum User Study Instructions — Day One, page 4



Yet, the corresponding ExternalStorage implementation is legal Java:

```
        List.next.get(m);                 <-  [OK; method is called]
```

However, this is probably not what the programmer intended: He or she might have meant to make the call a put instead, or perhaps he or she wanted to store the result in a local variable. Arcum allows for conditions like these to be checked.

In the first part of this task, you will add in a check and an error message (which are provided for you, below). In the second part, you will need to write your own check and error message for a similar problem.

Open up src/Attribute.arcum, and edit the code for the attrGet concept (in the AttributeInterface interface) to match what is shown below:

```
abstract attrGet(Expr getExpr, Expr targetExpr) {
    require "The target must be an instance of `targetType":
        isA(getExpr, attrType) && isA(targetExpr, targetType);

    require "The value of `getExpr must be used":
        !isExpressionStatement(getExpr);
}
```

As shown in the Arcum Language Quick-Reference, the isExpressionStatement predicate (i.e., Boolean function) returns true only when the expression given to it (in this case, getExpr) exists as a standalone statement (and thus it is impossible for the value to be used).

Save your modifications and correct any errors that you may have. Now, you will test this check by adding code that would violate it (i.e., make the Boolean predicate expression become false). Add the following statement to the “asList” method defined in src/edu/ucsd/study/ListPrinting.java, before line 15:

```
        List.next.get(list);
```

Save the edit. If your check has been implemented properly, you should now see the error message you wrote appear in the “Problems” view and this line of code will be highlighted. You may now comment out this line of code, to move on to the second part of this task.

In this second part we now consider a shift in meaning from the original code to the transformed code. In Java, the assignment operation

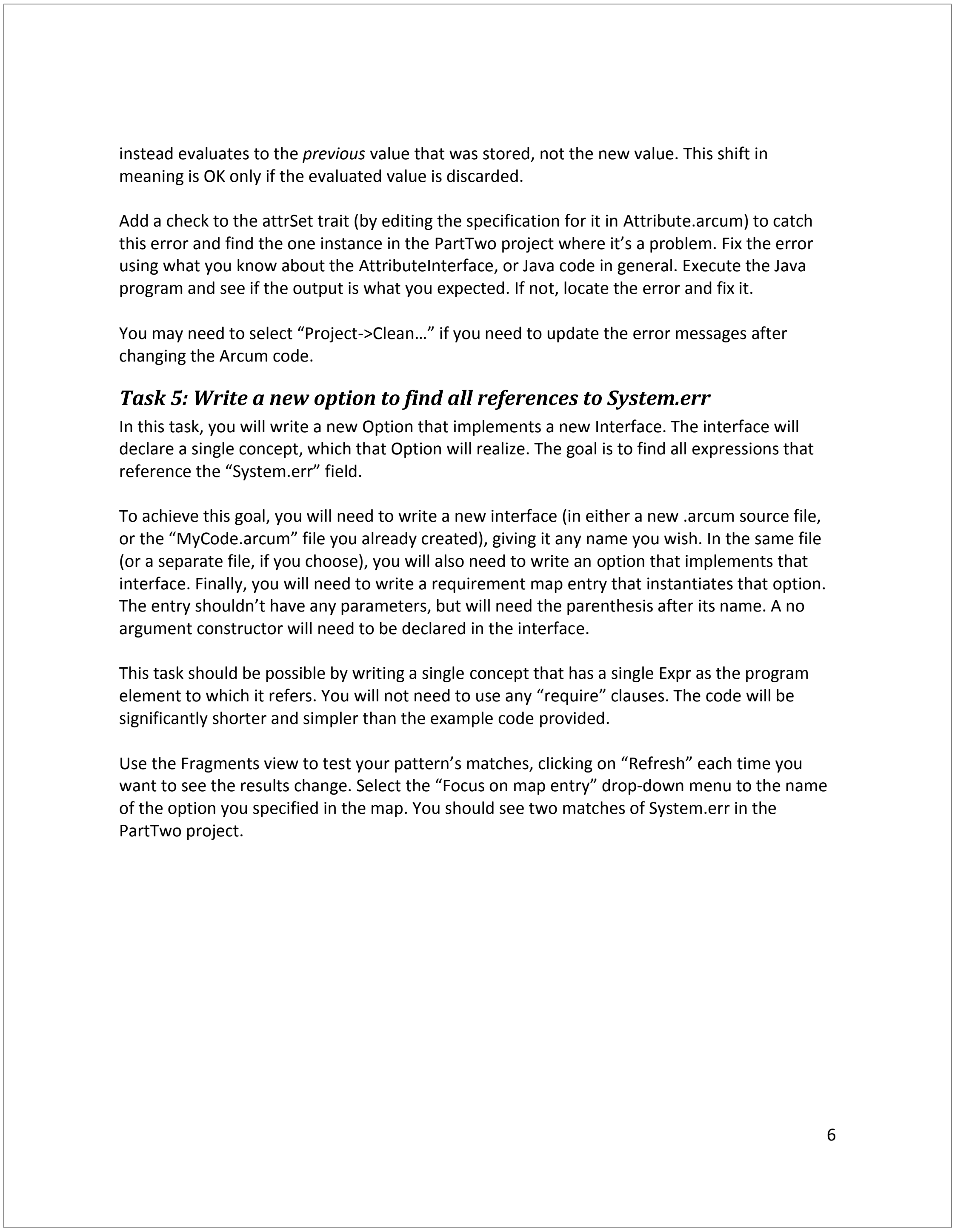
```
        this.next = n
```

evaluates to the value of “n” (i.e., the right-hand side). However, the equivalent Map “put” operation

```
        List.next.put(this, n)
```

5

Figure C.7: Arcum User Study Instructions — Day One, page 5



instead evaluates to the *previous* value that was stored, not the new value. This shift in meaning is OK only if the evaluated value is discarded.

Add a check to the attrSet trait (by editing the specification for it in Attribute.arcum) to catch this error and find the one instance in the PartTwo project where it's a problem. Fix the error using what you know about the AttributeInterface, or Java code in general. Execute the Java program and see if the output is what you expected. If not, locate the error and fix it.

You may need to select "Project->Clean…" if you need to update the error messages after changing the Arcum code.

Task 5: Write a new option to find all references to System.err

In this task, you will write a new Option that implements a new Interface. The interface will declare a single concept, which that Option will realize. The goal is to find all expressions that reference the "System.err" field.

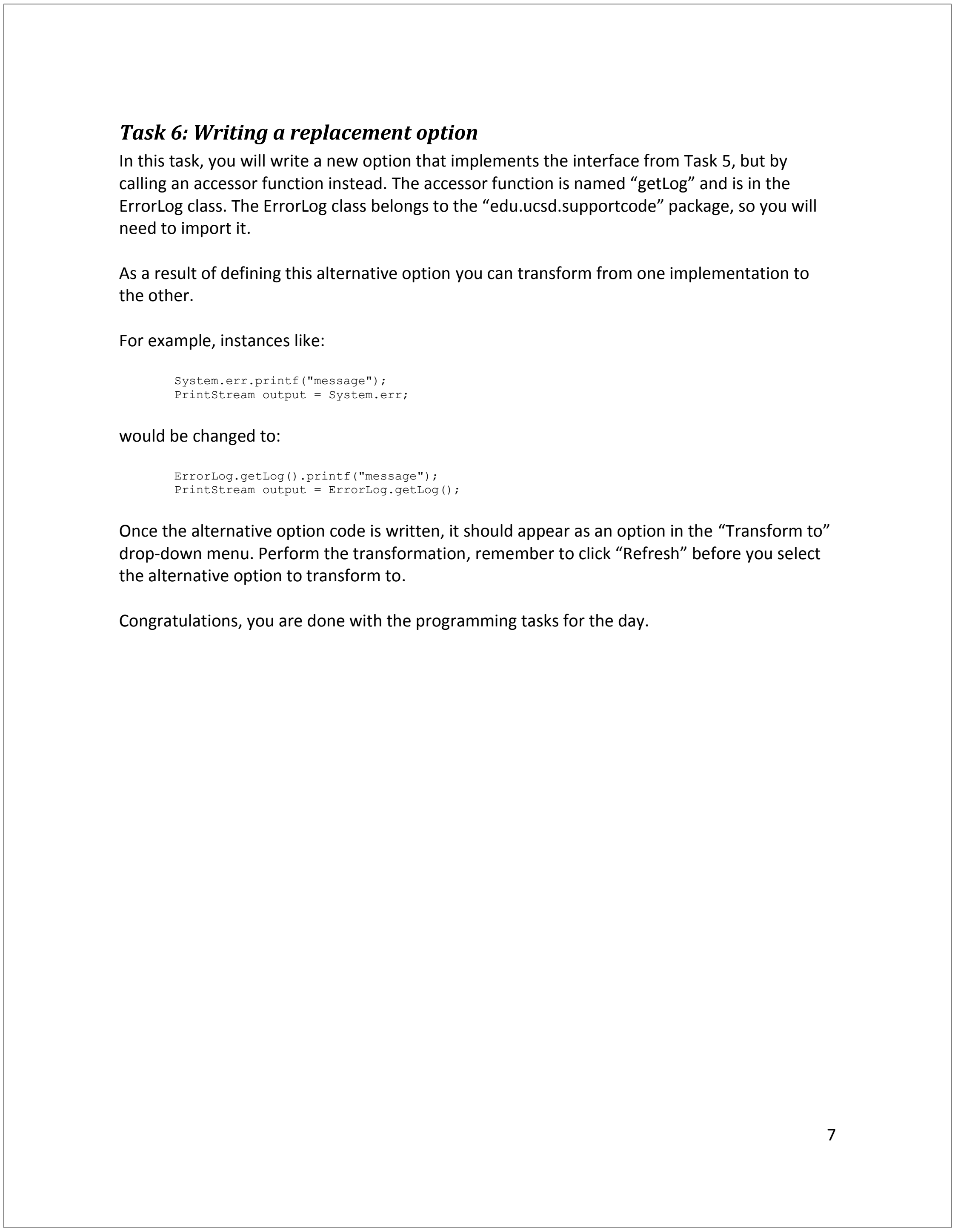
To achieve this goal, you will need to write a new interface (in either a new .arcum source file, or the "MyCode.arcum" file you already created), giving it any name you wish. In the same file (or a separate file, if you choose), you will also need to write an option that implements that interface. Finally, you will need to write a requirement map entry that instantiates that option. The entry shouldn't have any parameters, but will need the parenthesis after its name. A no argument constructor will need to be declared in the interface.

This task should be possible by writing a single concept that has a single Expr as the program element to which it refers. You will not need to use any "require" clauses. The code will be significantly shorter and simpler than the example code provided.

Use the Fragments view to test your pattern's matches, clicking on "Refresh" each time you want to see the results change. Select the "Focus on map entry" drop-down menu to the name of the option you specified in the map. You should see two matches of System.err in the PartTwo project.

6

Figure C.8: Arcum User Study Instructions — Day One, page 6



Task 6: Writing a replacement option

In this task, you will write a new option that implements the interface from Task 5, but by calling an accessor function instead. The accessor function is named “getLog” and is in the ErrorLog class. The ErrorLog class belongs to the “edu.ucsd.supportcode” package, so you will need to import it.

As a result of defining this alternative option you can transform from one implementation to the other.

For example, instances like:

```
        System.err.printf("message");
        PrintStream output = System.err;
```

would be changed to:

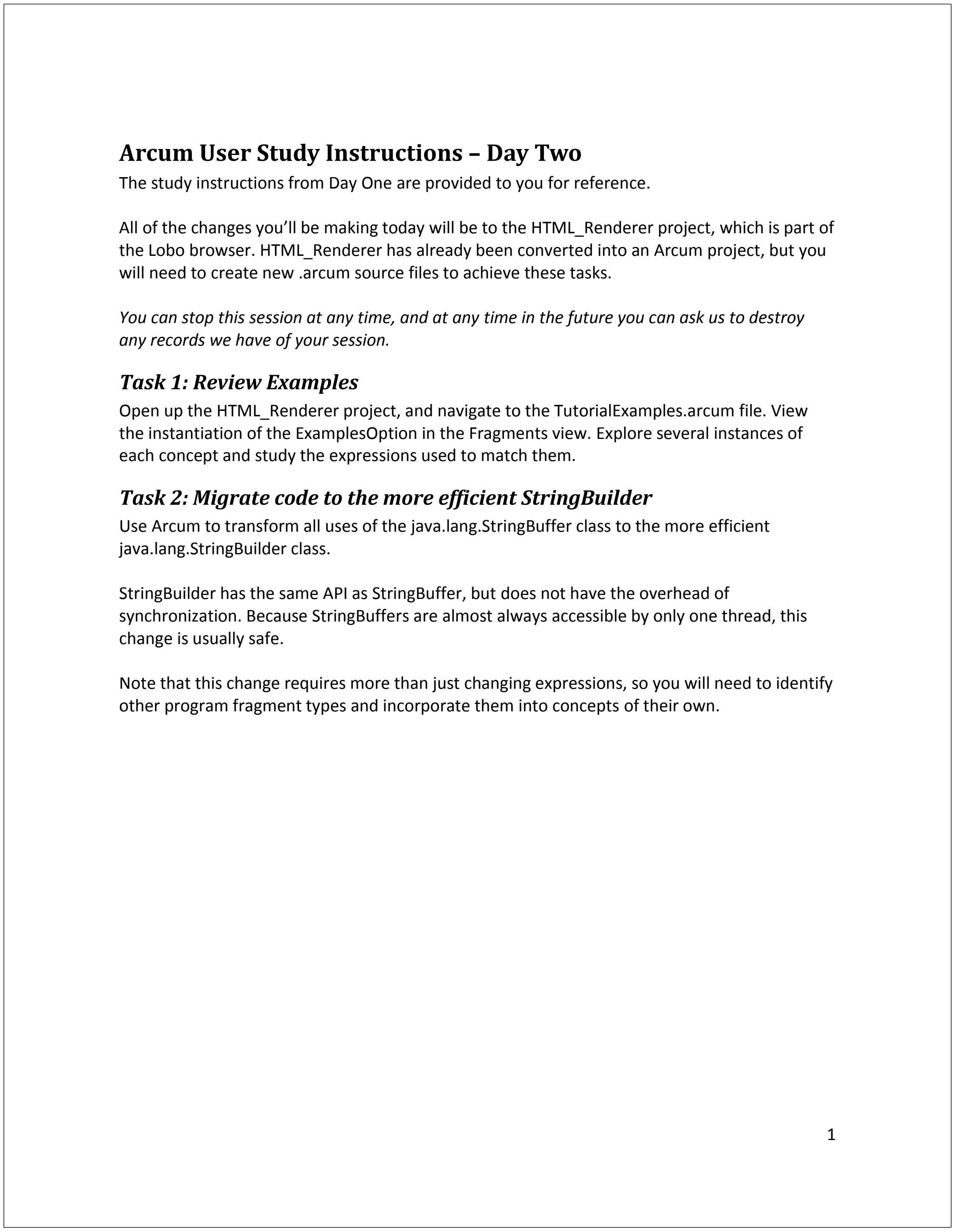
```
        ErrorLog.getLog().printf("message");
        PrintStream output = ErrorLog.getLog();
```

Once the alternative option code is written, it should appear as an option in the “Transform to” drop-down menu. Perform the transformation, remember to click “Refresh” before you select the alternative option to transform to.

Congratulations, you are done with the programming tasks for the day.

7

Figure C.9: Arcum User Study Instructions — Day One, page 7



Arcum User Study Instructions – Day Two

The study instructions from Day One are provided to you for reference.

All of the changes you'll be making today will be to the HTML_Renderer project, which is part of the Lobo browser. HTML_Renderer has already been converted into an Arcum project, but you will need to create new .arcum source files to achieve these tasks.

You can stop this session at any time, and at any time in the future you can ask us to destroy any records we have of your session.

Task 1: Review Examples

Open up the HTML_Renderer project, and navigate to the TutorialExamples.arcum file. View the instantiation of the ExamplesOption in the Fragments view. Explore several instances of each concept and study the expressions used to match them.

Task 2: Migrate code to the more efficient StringBuilder

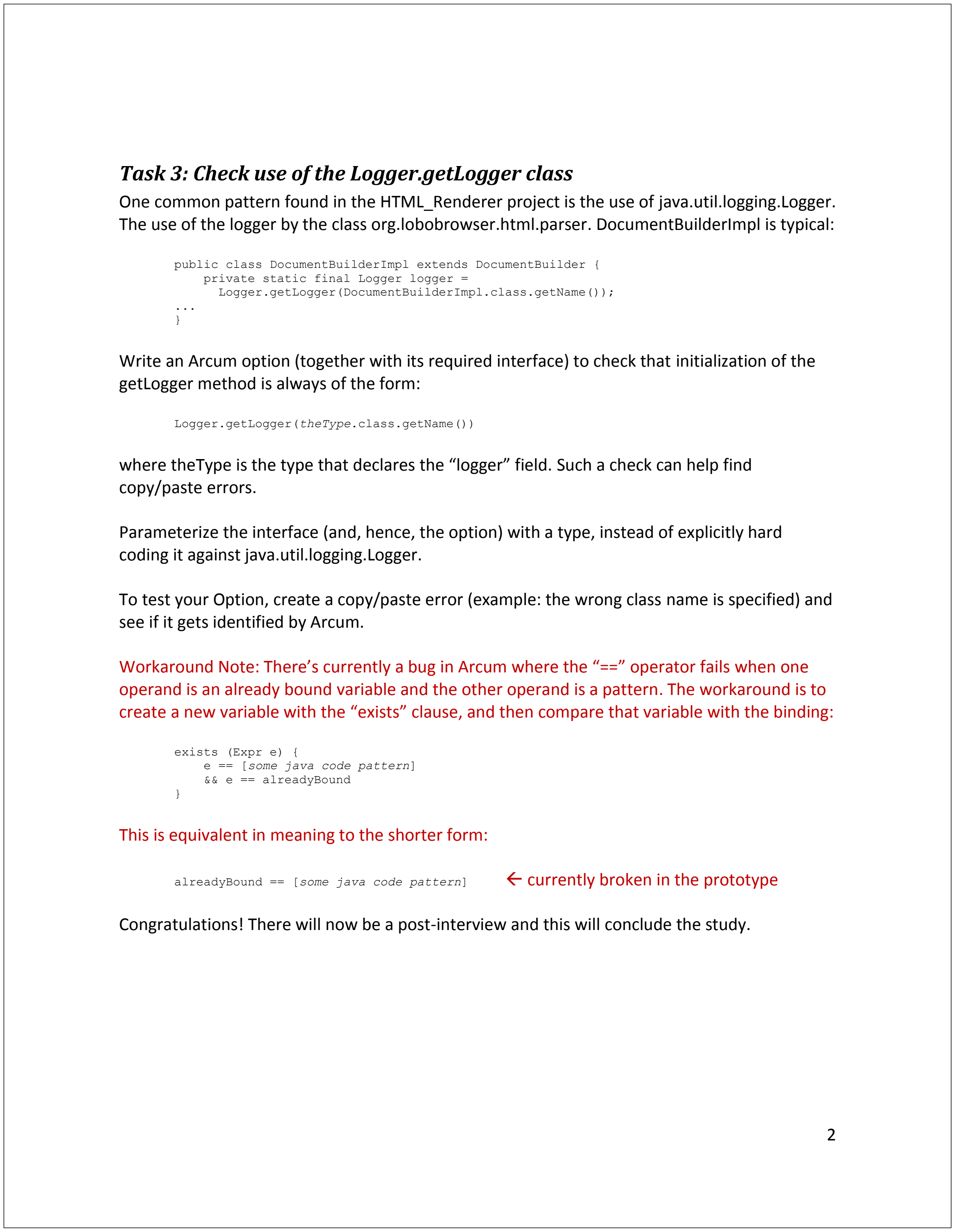
Use Arcum to transform all uses of the java.lang.StringBuffer class to the more efficient java.lang.StringBuilder class.

StringBuilder has the same API as StringBuffer, but does not have the overhead of synchronization. Because StringBuffers are almost always accessible by only one thread, this change is usually safe.

Note that this change requires more than just changing expressions, so you will need to identify other program fragment types and incorporate them into concepts of their own.

1

Figure C.10: Arcum User Study Instructions — Day Two, page 1



Task 3: Check use of the Logger.getLogger class

One common pattern found in the HTML_Renderer project is the use of java.util.logging.Logger. The use of the logger by the class org.lobobrowser.html.parser. DocumentBuilderImpl is typical:

```
public class DocumentBuilderImpl extends DocumentBuilder {
    private static final Logger logger =
      Logger.getLogger(DocumentBuilderImpl.class.getName());
...
}
```

Write an Arcum option (together with its required interface) to check that initialization of the getLogger method is always of the form:

```
Logger.getLogger(theType.class.getName())
```

where theType is the type that declares the “logger” field. Such a check can help find copy/paste errors.

Parameterize the interface (and, hence, the option) with a type, instead of explicitly hard coding it against java.util.logging.Logger.

To test your Option, create a copy/paste error (example: the wrong class name is specified) and see if it gets identified by Arcum.

Workaround Note: There’s currently a bug in Arcum where the “==” operator fails when one operand is an already bound variable and the other operand is a pattern. The workaround is to create a new variable with the “exists” clause, and then compare that variable with the binding:

```
exists (Expr e) {
    e == [some java code pattern]
    && e == alreadyBound
}
```

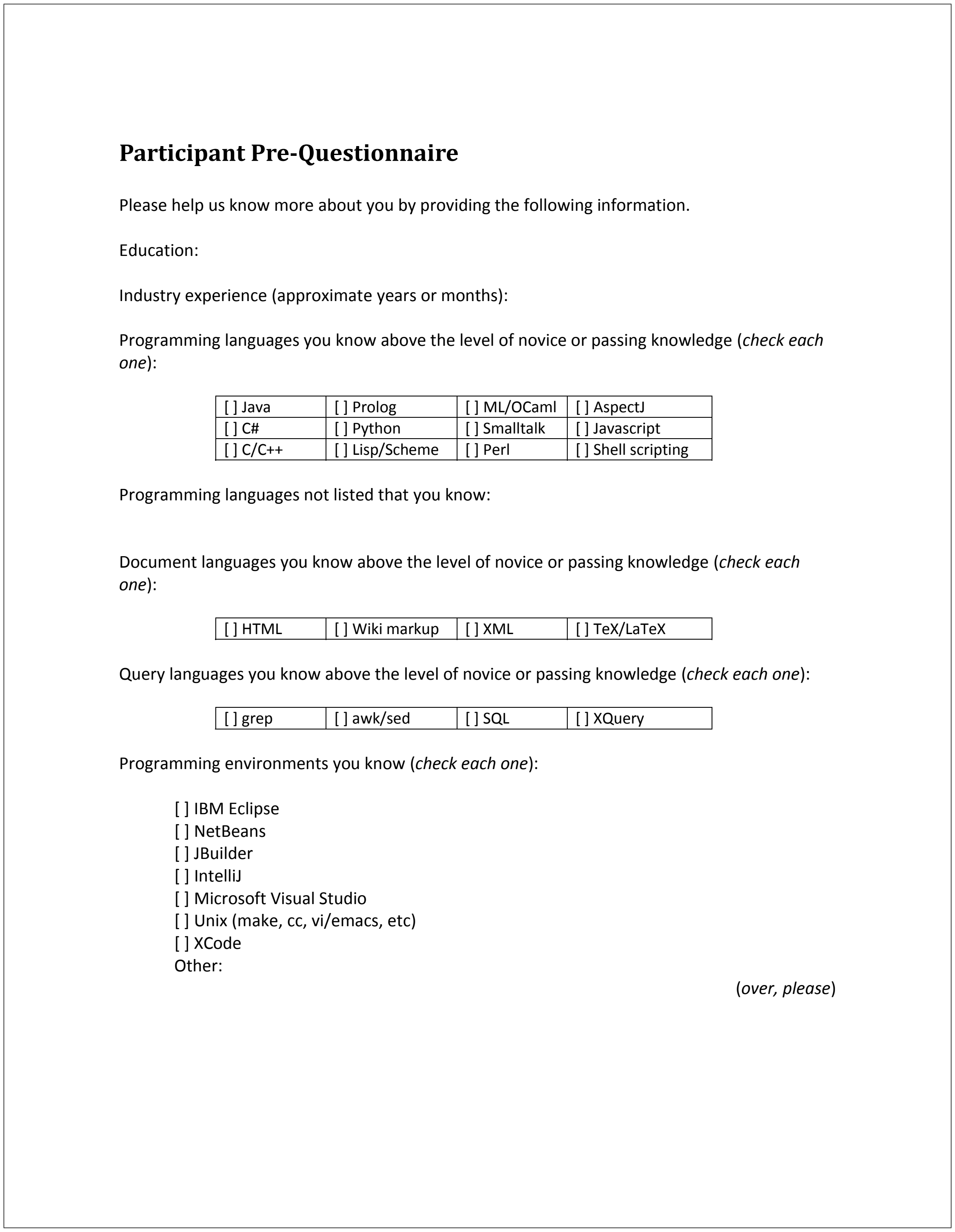
This is equivalent in meaning to the shorter form:

```
alreadyBound == [some java code pattern]
```
← currently broken in the prototype

Congratulations! There will now be a post-interview and this will conclude the study.

2

Figure C.11: Arcum User Study Instructions — Day Two, page 2



Participant Pre-Questionnaire

Please help us know more about you by providing the following information.

Education:

Industry experience (approximate years or months):

Programming languages you know above the level of novice or passing knowledge (*check each one*):

[ ] Java	[ ] Prolog	[ ] ML/OCaml	[ ] AspectJ
[ ] C#	[ ] Python	[ ] Smalltalk	[ ] Javascript
[ ] C/C++	[ ] Lisp/Scheme	[ ] Perl	[ ] Shell scripting

Programming languages not listed that you know:

Document languages you know above the level of novice or passing knowledge (*check each one*):

[ ] HTML	[ ] Wiki markup	[ ] XML	[ ] TeX/LaTeX

Query languages you know above the level of novice or passing knowledge (*check each one*):

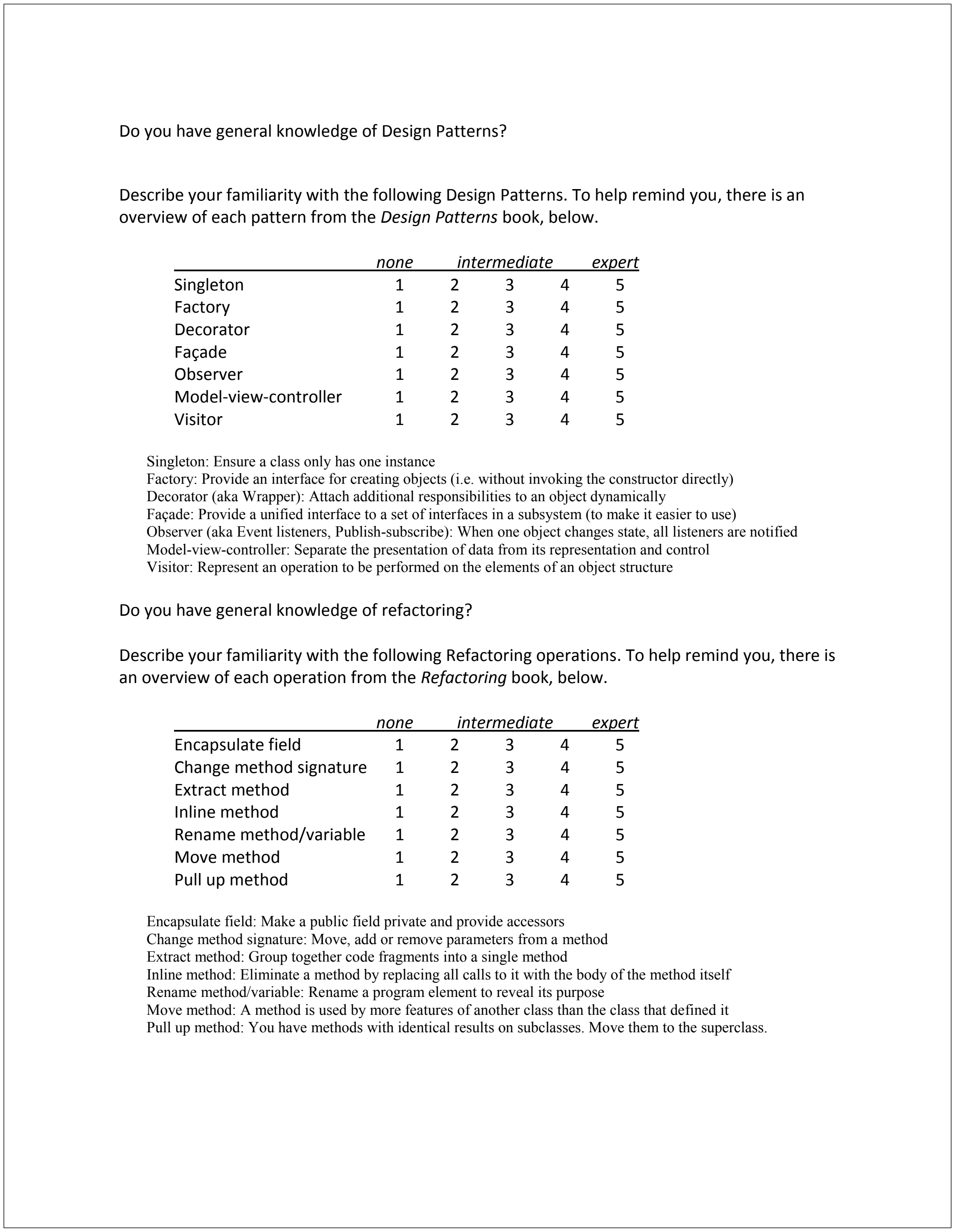
[ ] grep	[ ] awk/sed	[ ] SQL	[ ] XQuery

Programming environments you know (*check each one*):

[ ] IBM Eclipse
[ ] NetBeans
[ ] JBuilder
[ ] IntelliJ
[ ] Microsoft Visual Studio
[ ] Unix (make, cc, vi/emacs, etc)
[ ] XCode
Other:

(*over, please*)

Figure C.12: Participant Pre-Questionnaire — page 1



Do you have general knowledge of Design Patterns?

Describe your familiarity with the following Design Patterns. To help remind you, there is an overview of each pattern from the *Design Patterns* book, below.

	none		*intermediate*		*expert*
Singleton	1	2	3	4	5
Factory	1	2	3	4	5
Decorator	1	2	3	4	5
Façade	1	2	3	4	5
Observer	1	2	3	4	5
Model-view-controller	1	2	3	4	5
Visitor	1	2	3	4	5

Singleton: Ensure a class only has one instance
Factory: Provide an interface for creating objects (i.e. without invoking the constructor directly)
Decorator (aka Wrapper): Attach additional responsibilities to an object dynamically
Façade: Provide a unified interface to a set of interfaces in a subsystem (to make it easier to use)
Observer (aka Event listeners, Publish-subscribe): When one object changes state, all listeners are notified
Model-view-controller: Separate the presentation of data from its representation and control
Visitor: Represent an operation to be performed on the elements of an object structure

Do you have general knowledge of refactoring?

Describe your familiarity with the following Refactoring operations. To help remind you, there is an overview of each operation from the *Refactoring* book, below.

	none		*intermediate*		*expert*
Encapsulate field	1	2	3	4	5
Change method signature	1	2	3	4	5
Extract method	1	2	3	4	5
Inline method	1	2	3	4	5
Rename method/variable	1	2	3	4	5
Move method	1	2	3	4	5
Pull up method	1	2	3	4	5

Encapsulate field: Make a public field private and provide accessors
Change method signature: Move, add or remove parameters from a method
Extract method: Group together code fragments into a single method
Inline method: Eliminate a method by replacing all calls to it with the body of the method itself
Rename method/variable: Rename a program element to reveal its purpose
Move method: A method is used by more features of another class than the class that defined it
Pull up method: You have methods with identical results on subclasses. Move them to the superclass.

Figure C.13: Participant Pre-Questionnaire — page 2

Bibliography

[BA04] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2nd edition, 2004. 2

[BBS05] Alex Blewitt, Alan Bundy, and Ian Stark. Automatic verification of design patterns in java. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 224–232, New York, NY, USA, 2005. ACM Press. 96

[BC99] Carliss Y. Baldwin and Kim B. Clark. *Design Rules: The Power of Modularity Volume 1*. MIT Press, Cambridge, MA, USA, 1999. 1, 73

[BCVM02] Avi Bryant, Andrew Catton, Kris De Volder, and Gail C. Murphy. Explicit programming. In *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*, pages 10–18, New York, NY, USA, 2002. 3

[BG04] Marat Boshernitsan and Susan L. Graham. ixj: interactive source-to-source transformations for java. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 212–213, New York, NY, USA, 2004. ACM. 7

[Blo08] Joshua Bloch. *Effective Java*. Addison-Wesley, 2nd edition, 2008. 51

[BMMR01] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, Snowbird, Utah, June 2001. 8

[Bos06] Marat Boshernitsan. *Program manipulation via interactive transformations*. PhD thesis, University of California at Berkeley, Berkeley, CA, USA, 2006. 7

[BPM04] Ira D. Baxter, Christopher Pidgeon, and Michael Mehlich. Dms: Program transformations for practical scalable software evolution. In *ICSE*

'04: Proceedings of the 26th International Conference on Software Engineering, pages 625–634, Washington, DC, USA, 2004. IEEE Computer Society. 7

[Bro87] Frederick P. Brooks, Jr. No silver bullet: Essence and accidents of software engineering. *Computer*, 20(4):10–19, 1987. 1

[BTF05] Ittai Balaban, Frank Tip, and Robert Fuhrer. Refactoring support for class library migration. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 265–279, New York, NY, USA, 2005. ACM Press. 7, 20, 41, 71, 98

[CLCM00] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. Multijava: modular open classes and symmetric multiple dispatch for java. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 130–145, New York, NY, USA, 2000. ACM. 5

[CR93] Alain Colmerauer and Philippe Roussel. The birth of prolog. In *HOPL-II: The second ACM SIGPLAN conference on History of programming languages*, pages 37–52, New York, NY, USA, 1993. ACM. 28

[Den92] Daniel C. Dennett. *Consciousness Explained*. Back Bay Books, 1992. 9

[DLS02] Manuvir Das, Sorin Lerner, and Mark Seigle. Esp: Path-sensitive program verification in polynomial time. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, Berlin, Germany, June 2002. 8

[EK07] Andrew D. Eisenberg and Gregor Kiczales. Expressive programs through presentation extension. In *AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development*, pages 73–84, New York, NY, USA, 2007. ACM Press. 4

[Fow99] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999. 6

[Gaf06] Neal Gafter. Super type tokens. http://gafter.blogspot.com/2006/12/super-type-tokens.html, December 2006. 60

[GF07] David Greenfieldboyce and Jeffrey S. Foster. Type qualifier inference for java. *SIGPLAN Not.*, 42(10):321–336, 2007. 46

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. 2, 53, 54, 57, 96

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, third edition, 2005. 103

[Gla02] Robert L. Glass. Sorting out software complexity. *Commun. ACM*, 45(11):19–21, 2002. 1

[Goo07] Google. Google collections library 0.5 (alpha). http://code.google.com/-p/google-collections/, October 2007. 43

[Gri91] William G. Griswold. *Program Restructuring as an Aid to Software Maintenance*. PhD thesis, Department of Computer Science and Engineering, University of Washington, July 1991. 6

[Gri01] William G. Griswold. Coping with crosscutting software changes using information transparency. In *REFLECTION '01: Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, pages 250–265, London, UK, 2001. Springer-Verlag. 2, 76

[GSS$^+$06] William G. Griswold, Kevin Sullivan, Yuanyuan Song, Macneil Shonle, Nishit Tewari, Yuanfang Cai, and Hridesh Rajan. Modular software design with crosscutting interfaces. *IEEE Software*, 23(1):51–60, 2006. 3, 81

[HK02] Jan Hannemann and Gregor Kiczales. Design pattern implementation in java and aspectj. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 161–173, New York, NY, USA, 2002. ACM Press. 3

[HMK05] Jan Hannemann, Gail C. Murphy, and Gregor Kiczales. Role-based refactoring of crosscutting concerns. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 135–146, New York, NY, USA, 2005. ACM Press. 6

[HP04] David Hovemeyer and William Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, 2004. 53

[KAM05] Andrew J. Ko, Htet Aung, and Brad A. Myers. Eliciting design requirements for maintenance-oriented ides: a detailed study of corrective and perfective maintenance tasks. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 126–135, New York, NY, USA, 2005. ACM. 94

[Ker05] Joshua Kerievsky. *Refactoring to Patterns*. Addison-Wesley, 2005. 61

[KH01] Gregor Kiczales and Erik Hilsdale. Aspect-oriented programming. In *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, page 313, New York, NY, USA, 2001. ACM. 102

[KHH^{+}01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *15th European Conference on Object-Oriented Programming (ECOOP 2001)*, pages 327–353, June 2001. 3, 38

[KM89] G. Kotik and L. Markosian. Automating software analysis and testing using a program transformation system. In *TAV3: Proceedings of the ACM SIGSOFT '89 third symposium on Software testing, analysis, and verification*, pages 75–84, New York, NY, USA, 1989. ACM Press. 7

[KNE92] Wojtek Kozaczynski, Jim Ning, and Andre Engberts. Program concept recognition and transformation. *IEEE Trans. Softw. Eng.*, 18(12):1065–1075, 1992. 7, 19

[LBL06] Jia Liu, Don Batory, and Christian Lengauer. Feature oriented refactoring of legacy applications. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 112–121, New York, NY, USA, 2006. ACM Press. 7

[Lie96] Karl J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996. 55, 56

[LO97] Karl J. Lieberherr and Doug Orleans. Preventive program maintenance in Demeter/Java (research demonstration). In *International Conference on Software Engineering*, pages 604–605, Boston, MA, 1997. ACM Press. 5

[Lob08] Lobo, 2008. http://lobobrowser.org/. 70

[Miy86] N. Miyake. Constructive interaction and the iterative process of understanding. *Cognitive Science*, 10(2):151–177, 1986. 67

[MKRC05] Gail C. Murphy, Mik Kersten, Martin P. Robillard, and Davor Cubranic. The emergent structure of development tasks. In *ECOOP*, volume 3586 of *Lecture Notes in Computer Science*, pages 33–48. Springer, 2005. 94

[MLL05] Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding application errors and security flaws using pql: a program query language. *SIGPLAN Not.*, 40(10):365–383, 2005. 28

[MMvD05] Marius Marin, Leon Moonen, and Arie van Deursen. An approach to aspect refactoring based on crosscutting concern types. In *MACS '05: Proceedings of the 2005 workshop on Modeling and analysis of concerns in software*, pages 1–5, New York, NY, USA, 2005. ACM Press. 6

[MVH$^+$07] Oege de Moor, Mathieu Verbaere, Elnar Hajiyev, Pavel Avgustinov, Torbjorn Ekman, Neil Ongkingco, Damien Sereni, and Julian Tibble. Keynote address: .ql for source code analysis. *Source Code Analysis and Manipulation, 2007. SCAM 2007. Seventh IEEE International Working Conference on*, pages 3–16, 2007. 28

[MVW07] Clint Morgan, Kris De Volder, and Eric Wohlstadter. A static aspect language for checking design rules. In *AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development*, pages 63–72, New York, NY, USA, 2007. ACM Press. 8

[MW06] Steven Metsker and William C. Wake. *Design Patterns in Java*. Addison-Wesley, 2006. 59

[NCM03] N. Nystrom, M. Clarkson, and A. Myers. Polyglot: An extensible compiler framework for java, 2003. 38

[OL01] Doug Orleans and Karl J. Lieberherr. Dj: Dynamic adaptive programming in java. In *REFLECTION '01: Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, pages 73–80, London, UK, 2001. Springer-Verlag. 60

[Opd92] William F. Opdyke. *Refactoring object-oriented frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1992. 6

[OT99] Harold Ossher and Peri Tarr. Using subject-oriented programming to overcome common problems in object-oriented software development/evolution. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 687–688, New York, NY, USA, 1999. ACM. 3

[Par72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972. 1, 44, 73, 89

[Pin07] Steven Pinker. *The Stuff of Thought: Language as a Window into Human Nature*. Viking, 2007. 10

[RAF04] Nick Rutar, Christian B. Almazan, and Jeffrey S. Foster. A comparison of bug finding tools for java. In *ISSRE '04: Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE'04)*, pages 245–256, Washington, DC, USA, 2004. IEEE Computer Society. 53, 97

[RC96] Mary Beth Rosson and John M. Carroll. The reuse of uses in smalltalk programming. *ACM Trans. Comput.-Hum. Interact.*, 3(3):219–253, 1996. 85

[RC04] Martin P. Robillard and Wesley Coelho. How effective developers investigate source code: An exploratory study. *IEEE Trans. Softw. Eng.*, 30(12):889–903, 2004. 94

[Rob99] Donald Bradley Roberts. *Practical analysis for refactoring*. PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1999. Adviser-Johnson, Ralph. 6

[SGCH01] Kevin J. Sullivan, William G. Griswold, Yuanfang Cai, and Ben Hallen. The structure and value of modularity in software design. In *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 99–108, New York, NY, USA, 2001. ACM. 73

[SGL07] Macneil Shonle, William G. Griswold, and Sorin Lerner. Beyond refactoring: a framework for modular maintenance of crosscutting design idioms. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 175–184, New York, NY, USA, 2007. ACM. 8

[SGL08a] Macneil Shonle, William G. Griswold, and Sorin Lerner. Addressing common crosscutting problems with arcum. In *8th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, November 2008. 8

[SGL08b] Macneil Shonle, William G. Griswold, and Sorin Lerner. When refactoring acts like modularity: Keeping options open with persistent condition checking. In *Second Workshop on Refactoring Tools*, October 2008. 8

[SGS$^+$05] Kevin Sullivan, William G. Griswold, Yuanyuan Song, Yuanfang Cai, Macneil Shonle, Nishit Tewari, and Hridesh Rajan. Information hiding interfaces for aspect-oriented design. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 166–175, New York, NY, USA, 2005. 3, 81

[Sho07] Macneil Shonle. Modular-like transformations and style checking for crosscutting programming concepts. In *ICSE COMPANION '07: Companion to the proceedings of the 29th International Conference on Software*

Engineering, pages 95–96, Washington, DC, USA, 2007. IEEE Computer Society. 8

[Sim95] C. Simonyi. The death of computer languages, the birth of intentional programming, 1995. 7

[SLS03] Macneil Shonle, Karl Lieberherr, and Ankit Shah. Xaspects: an extensible system for domain-specific aspect languages. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 28–37. ACM Press, 2003. 5

[SMV06] Jonathan Sillito, Gail C. Murphy, and Kris De Volder. Questions programmers ask during software evolution tasks. In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 23–34, New York, NY, USA, 2006. ACM. 94

[STD04] B. De Sutter, F. Tip, and J. Dolby. Customization of java library classes using type constraints and profile information. In *Proc. ECOOP'04*, pages 585–609, 2004. 71

[Ste85] J. Steffen. Interactive examination of a c program with cscope. In *Proceedings of the 1985 Winter Usenix Conference, Dallas, TX.*, 1985. 6

[Str97] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, third edition, 1997. 10

[SWM00] M.-A. D. Storey, K. Wong, and H. A. Müller. How do program understanding tools affect how programmers understand programs? *Sci. Comput. Program.*, 36(2-3):183–207, 2000. 94

[TB08] Eli Tilevich and Godmar Back. "Program, enhance thyself!" – demand-driven pattern-oriented program enhancement,". In *AOSD '08: Proceedings of the 7th international conference on Aspect-oriented software development*, April 2008. 4

[TKB03] Frank Tip, Adam Kiezun, and Dirk Bäumer. Refactoring for generalization using type constraints. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 13–26, New York, NY, USA, 2003. ACM. 41

[TOHS99] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton, Jr. N degrees of separation: multi-dimensional separation of concerns. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 107–119, New York, NY, USA, 1999. ACM. 3

[TOS02] Peri Tarr, Harold Ossher, and Stanley M. Sutton, Jr. Hyper/j^{TM}: multi-dimensional separation of concerns for java^{TM}. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 689–690, New York, NY, USA, 2002. ACM. 3

[vD03] Daniel von Dincklage. Making patterns explicit with metaprogramming. In *GPCE '03: Proceedings of the 2nd international conference on Generative programming and component engineering*, pages 287–306, New York, NY, USA, 2003. Springer-Verlag New York, Inc. 4

[VEdM06] Mathieu Verbaere, Ran Ettinger, and Oege de Moor. Jungl: a scripting language for refactoring. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 172–181, New York, NY, USA, 2006. ACM. 28