Addressing Common Crosscutting Problems with Arcum*

Macneil Shonle William G. Griswold Sorin Lerner
Computer Science & Engineering, UC San Diego
La Jolla, CA 92093-0404
{mshonle, wgg, lerner}@cs.ucsd.edu

ABSTRACT

Crosscutting is an inherent part of software development and can typically be managed through modularization: A module's stable properties are defined in an interface while its likely-to-change properties are encapsulated within the module [19]. The crosscutting of the stable properties, such as class and method names, can be mitigated with automated refactoring tools that allow, for example, the interface's elements to be renamed [9, 18]. However, often the crosscutting from design idioms (such as design patterns and coding styles) are so specific to the program's domain that their crosscutting would not likely have been anticipated by the developers of an automated refactoring system.

The Arcum plug-in for Eclipse enables programmers to describe the implementation of a crosscutting design idiom as a set of syntactic patterns and semantic constraints. Arcum can process declarations of related implementations and infer the refactoring steps necessary to transform a program from using one implementation to its alternatives. As a result, automating refactoring for domain-specific crosscutting design idioms can be easy and practical. This paper presents a case study of how Arcum was used to mitigate four classic software engineering problems that are exacerbated by crosscutting: library migration, debugging, programmer-defined semantic checking, and architectural enforcement.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures—*Domain-specific architectures, languages, patterns*

General Terms

Languages, Design.

Keywords

Refactoring, design patterns, aspect-oriented programming.

*This work was supported in part by an Eclipse Innovation Award from IBM and NSF Science of Design grant CCF-0613845.

PASTE '08 Atlanta, Georgia USA

1. INTRODUCTION

Arcum is a framework for declaring and performing user-defined program checks and transformations, with the goal of increasing automated refactoring opportunities for the user [21]. By using Arcum, a programmer can view the implementation of a crosscutting design idiom as a form of module. Arcum uses a declarative language to describe the idiom's implementation, where descriptions are composed of Arcum interface and Arcum option constructs. An option describes one possible implementation of a crosscutting design idiom, and a set of options are related to each other when they all implement the same Arcum interface.

Arcum's declarative language uses a Java-like syntax for first-order logic predicate statements, including a special pattern notation for expressing Java code. Like most declarative languages, Arcum operates from a database of relations. Arcum's database contains relations associated with the program being analyzed, including relations like isA, hasMethod, and copiedTo, and with special relations to cover Java syntax, such as method invocations, field references, and type declarations.

This paper presents a case study of Arcum's use on its own codebase, where we encountered a variety of classical software engineering problems that are induced by crosscutting design idioms. This paper extends our previous work [21] by demonstrating several plausible ways crosscutting manifests itself in real-life programs, and how such crosscutting can be mitigated.

The Arcum project is an Eclipse plug-in written in Java, which comprises of over 150 public classes and 19,000 lines of code. We chose the Arcum codebase in part because we are familiar with it and we believe it uses idioms that can be assisted with Arcum.

We found that many of the software engineering problems encountered were not specific to the particulars of the project, yet also not general enough to be anticipated by tool developers or even the Java language designers. We believe there is a middle ground between problems so readily encountered they are addressed directly by the language, and problems so rarely encountered that you would only need to cope with them once. For our purposes, we refer to this middle ground as involving domain specific needs. The purpose of the case study is to demonstrate the frontier of possibility for an expert in the Arcum framework, leaving open for future user-studies to show 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. The first category covers problems related to code migration and the trade offs between making a program's implementation more standardized or more project specific (Section 2). The second category covers the process of debugging (Section 3) and the third category covers how needs specific to a domain differ from the general needs

```
01  interface IntegerBoxingOperation {
02    abstract boxing(Expr expr, Expr boxedValue) {
03      exists (DeclarationElement d) {
04        d == [Integer] && copiedTo(expr, d)
05      }
06      && isA(boxedValue, <int>)
07    }
08  }
09
10  option ImplicitBoxing implements IntegerBoxingOperation {
11    realize boxing(Expr expr, Expr boxedValue) { expr == [`boxedValue] }
12  }
13
14  option ExplicitBoxing implements IntegerBoxingOperation {
15    realize boxing(Expr expr, Expr boxedValue) { expr == [new Integer(`boxedValue)] }
16 }
```

Figure 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.

that a programming language covers (Section 4). The final category covers software architectural enforcements needs (Section 5).

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 [1]. 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 safely, 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 [24]. The pattern used to specify DeclarationElements in the program is a type pattern. Line 4 of Figure 1 shows a pattern to match all declarations of the Integer wrapper class.

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. The relation holds when the value of the expression is copied to a location declared by the declaration element.[1]

The code in Figure 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.

2.1 Canonicalization

One use we found for class library migration was when we started to employ the Google Collections library [8]. The library includes extra support for programming with generics in Java, including interfaces and operations to support functional programming styles.

For example, we had a need in Arcum to support operations that are lazily executed. This division of definition and execution allowed us to separate the knowledge of how to initialize an object from the knowledge of when to initialize it. We 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 we found that the Supplier interface fit our needs exactly. By using an interface defined in another library, we were able to make our own 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.

2.2 Removing Puns: De-Canonicalization

Of course, there are always trade offs with using standard libraries. One risk of transforming all similar-looking uses to 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 one be better typed as DelayedObject<String> while the other is typed as 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.

[1]Copy operations include: assignment, initialization, argument passing, and value returning.

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 meta-data annotations to them, creating a form of qualified types.[2] It could then 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 on 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.

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 [19], there are an infinite number of design futures—if you were to anticipate them all, development 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, we encountered an intermittent bug: Sometimes the program would halt with a NullPointerException and sometimes it would compute the expected result. Eventually, we discovered that the source of the problem was the iteration order of HashMaps. The hashCode used was the default identity code. On the VM we were 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, we wanted to see how we could use Arcum to help. One solution is related to the class library migration problem (see Section 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 we were 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 we weren't quite so lucky, we'd still have some options available: We could have added more test cases, in the hopes of finding the right code sequence to expose the bug again. Alternatively, we 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.

[2]However, as of Java 6, type arguments for parameterized types cannot have annotations. Thus, the solution is not complete, although some work-arounds exist.

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 2.2). This section covers additional examples where the user can benefit from extra checking and constraints.

4.1 When Simple Solutions are too General

Solutions can be considered too general in the context of using only special cases of standard methods. 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 Strings in Arcum, we discovered multiple uses of the pattern t.contains("."). Here, names of elements in the Java program analyzed were represented as Strings, 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, we were able to find all references to the special use of contains and transform them to use the static method. We then had a named entity that Eclipse could use to build a list of entity references. We 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 Strings 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, where Strings 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.

4.2 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. Had this method been declared as non-static we could have just made a simple call to it. Instead, we needed a mechanism to invoke a different static method depending on the type of the instance. By invoking getClass on the instance, we 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, we were able to use reflection in a more disciplined manner. Figure 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)

```
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 2: 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.

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

Figure 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 Strings.

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

Figure 4: A static solution for Figure 3 that doesn't require reflection.

All arguments passed to the invokeStatic method must match the number and type of the parameters specified 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.

4.2.1 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 anonymous inner classes. Figure 3 shows a method makeFunction that takes in a type and a method name and returns a Function object. When this idiom is encoded as an option the run-time 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 (Figure 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 particularly for methods belonging to immutable classes like BigInteger. In the implementation of Arcum, we use an immutable set-like construct; several times, we encountered a bug where results were inadvertently getting discarded, such as when we wrote:

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

instead of:

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

By checking calls to the union method we were 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.) 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();
```

We detected such a problem while debugging the Arcum project: We wondered why we couldn't see some debugging output, and then we realized that our debugging code came after a method that didn't return. The compiler accepted the code, even though for all purposes that code was unreachable.

5. ENFORCEMENT OF ARCHITECTURE AND STYLE

Section 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 5.1) and programming style (Section 5.2).

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, because external methods cannot access them; classes en-

capsulate their private fields, because 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 facade pattern [7]: The facade pattern can reduce the level of coupling between components and assist layering. We used the facade pattern in Arcum when interfacing with Eclipse's Java compiler. We utilized Eclipse's type checker to resolve variable bindings, but our syntax desugaring mechanism complicated where the bindings would be available. The solution was to write a facade that was a single point of interaction with Eclipse's resolver. Using Arcum for this solution helped in two ways: First, we were able to refactor each call to Eclipse's resolver to be a call to the facade instead. Second, we wrote new checks that prevented direct calls from being inadvertently made. This extra checking ensured that the facade pattern held and that layering was preserved.

On the smaller scale, we also utilized intra-class layering in our implementations. For example, we found that even the private access specifier was not strict enough for our needs when it came to reasoning about classes. In one case, we had two related fields in a class to which we only wanted the constructors and two tightly-coupled accessor methods to have direct access. We 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 the concept of friends in C++, with the ability to enable or disable read or read/write access.

5.2 Detecting Common Errors

There is a class of general programming errors that lend themselves well to automatic detection [12, 20], several of which can be checked with Arcum. One real-life bug we encountered was when we executed code that raised a particular RuntimeException, yet our exception handlers were not catching the exception.

The problem was related to how we *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 we 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 we 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, we added a check to find all cases of exception softening and made sure that RuntimeExceptions 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.

6. RELATED WORK

Arcum's most general philosophy is common to many other works: "Improve programming by letting programmers better express their intentions to the environment." Examples include Explicit Programming [5], Presentation Extension [6], Metaprogramming [26], and Intentional Programming [22]. Arcum takes a departure from these works because it does not extend the programming language itself. Instead, Arcum only applies checks to existing code, keeping exactly the same Java semantics, while in one form extending Java's type system through additional error messages the user can enable. The flexibility of the Arcum approach relies upon the expressiveness of refactoring transformations rather than upon the expressiveness of a new programming language.

On the implementation side, because Arcum has a declarative language, it is related to a large family of program query tools, such as JunGL [25], QL [17] and PQL [16]. Some of these systems support additional checks to apply to code, but do not use these checks to infer program transformations.

The iXj program transformation system for Java allows for pattern matching similar to Arcum's concept construct [4, 3]. The iXj system could assist the writing of Arcum concepts through its interactive features, while Arcum could complement iXj by providing a mean of expressing infrastructure related to concepts and by providing continuous checking of implementations.

DRIVEL is a program enhancement system using generative techniques on top of an aspect-oriented language [23]. DRIVEL offers a way to change the programming language such that code written using it is closer to what is intended. This technique is particularly well suited for design patterns, because the code that needs to be generated can be inferred from the context. For example, calls to methods related to the visitor pattern can be detected, with their definitions supplied at the byte-code level, eliminating the need to implement the methods manually.

Arcum is a departure from the role-based refactoring work of Hannemann et al. [11], which permits programmers to build macro-refactorings from micro-refactorings. AOP languages like AspectJ can manifest many crosscutting design idioms, including many design patterns, as modular abstractions [10].

Feature Oriented Refactoring (FOR) recognizes the crosscutting and non-modular nature of the implementation of software features, which are often crosscutting [15]. The REFINE system uses program templates, which can be used for both pattern matching and code transformation [13].

As a departure from REFINE, Kozaczynski et al. [14] employ semantic pattern matching to recognize concepts as part of a code transformation system for software maintenance. A more recent work in this area is the DMS system, which is similar to Kozaczynski et al. but has a much wider scope [2].

7. CONCLUSION

Our 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 idioms reach widespread use, the refactoring needs related to those idioms can be anticipated by tool developers. However, we 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.

Arcum lessens the liability of having certain kinds of crosscutting implementations present, and thus can change the development process itself. When properly employed, such extra freedom improves the quality of software or enhances other related economic utilities.

8. REFERENCES

[1] I. Balaban, F. Tip, and R. 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.

[2] I. D. Baxter, C. Pidgeon, and M. 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.

[3] M. Boshernitsan. *Program manipulation via interactive transformations*. PhD thesis, University of California at Berkeley, Berkeley, CA, USA, 2006. Adviser-Susan L. Graham.

[4] M. Boshernitsan and S. 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.

[5] A. Bryant, A. Catton, K. D. Volder, and G. 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. ACM.

[6] A. D. Eisenberg and G. 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.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

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

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

[10] J. Hannemann and G. 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.

[11] J. Hannemann, G. C. Murphy, and G. 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.

[12] D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, 2004.

[13] 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.

[14] W. Kozaczynski, J. Ning, and A. Engberts. Program concept recognition and transformation. *IEEE Trans. Softw. Eng.*, 18(12):1065–1075, 1992.

[15] J. Liu, D. Batory, and C. 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.

[16] M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using pql: a program query language. *SIGPLAN Not.*, 40(10):365–383, 2005.

[17] O. d. Moor, M. Verbaere, E. Hajiyev, P. Avgustinov, T. Ekman, N. Ongkingco, D. Sereni, and J. 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.

[18] W. F. Opdyke. *Refactoring object-oriented frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1992.

[19] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[20] N. Rutar, C. B. Almazan, and J. 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.

[21] M. Shonle, W. G. Griswold, and S. 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.

[22] C. Simonyi. The death of computer languages, the birth of intentional programming, 1995.

[23] E. Tilevich and G. 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.

[24] F. Tip, A. Kiezun, and D. 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.

[25] M. Verbaere, R. Ettinger, and O. 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.

[26] D. 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.