

Beyond Refactoring: A Framework for Modular Maintenance of Crosscutting Design Idioms*

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

Despite the automated refactoring support provided by today's IDEs many program transformations that are easy to conceptualize—such as improving the implementation of a design pattern—are not supported and are hence hard to perform. We propose an extension to the refactoring paradigm that provides for the modular maintenance of crosscutting design idioms, supporting both substitutability of design idiom implementations and the checking of essential constraints. We evaluate this new approach through the design and use of Arcum, an IDE-based mechanism for declaring, checking, and evolving crosscutting design idioms.

Categories and Subject Descriptors

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

General Terms

Languages, Design.

Keywords

Refactoring, design patterns.

1. INTRODUCTION

One of the benefits of modular programming is that the implementation of a module can be improved or replaced by another without requiring changes to, or extensive retesting of, other parts of the system. However, many change tasks are crosscutting in nature and thus outside of modular bounds [9]. For example, not every future change can be anticipated, meaning that the existing abstractions may not modularize the given change. Sometimes, the language's abstraction mechanisms are not powerful enough to permit an efficient modularization. Other times, an agile development process like XP may intentionally delay the introduction of such abstractions [4].

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ESEC/FSE'07, September 3–7, 2007, Cavtat near Dubrovnik, Croatia.
Copyright 2007 ACM 978-1-59593-811-4/07/0009 ...\$5.00.

In this paper we introduce the Arcum framework.¹ Arcum allows for certain forms of crosscutting design idioms to be transformed into alternative implementations. Arcum expands the opportunities for modular substitution and reasoning through *options*. An Arcum option declares the implementation details of a crosscutting entity, including any required supporting code and infrastructure. 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 design idiom 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 can be written in a generic fashion and then instantiated for a specific case, 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, not just during refactoring. In this respect, the benefits of static checking of classes extend to crosscutting entities. Continuous checking also 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 design idiom and expressing the programmer's intentions for its implementation. Finally, because Arcum declarations are supplements, the core source code remains unchanged, and in pure Java. The program is only changed 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.

In the next section we introduce the Arcum approach with a comparative example. We then review the Arcum language in greater detail and describe our algorithm for transforming between two op-

¹Arcum's general goals were first proposed in a short paper at the ICSE 2007 Doctoral Symposium [24].

```

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

```

Figure 1: Internal field implementation of the *altText* attribute.

```

15 public class Image {
16     static Map<Image, String> altText
17         = new IdentityHashMap<Image, String>();
18     // ...
19     public Image(String alternative) {
20         this();
21         Image.altText.put(this, alternative);
22     }
23     public String toString() {
24         if (Image.altText.get(this) == null)
25             return defaultAltText();
26         else
27             return Image.altText.get(this);
28     }
29 }

```

Figure 2: Static map implementation of the *altText* attribute.

tions. Next, to provide an evaluation of our approach, we describe the design of the Arcum refactoring engine, report on our development of a related group of options, and compare several change tasks versus the state of the art in Eclipse and AspectJ [13]. We close with a discussion of future work, related work, and a few concluding remarks.

2. EXAMPLE: ATTRIBUTE STORAGE

In this section we illustrate the Arcum framework with 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 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.

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 end result is that refactoring and software evolution in general is a common occurrence in the development of large software systems.

To give an overview of our approach, we 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 1 has only two reads (lines 9 and 12) and one write (line 6), in a realistic code base 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: (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 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 $f(\text{this.altText}=y)$ into $f(\text{Image.altText.put}(\text{this},y))$ 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 to ensure that new or modified code also satisfies the transformation’s pre- and post-conditions. Figure 3 shows the Arcum plug-in for Eclipse performing the refactoring.

Figure 5 shows how the *Image* constructor from our 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).

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 4. 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 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* “trait” (Figure 5, boxed), which represents all locations in the Java program where the attribute’s value is set. The parameters to the trait, 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

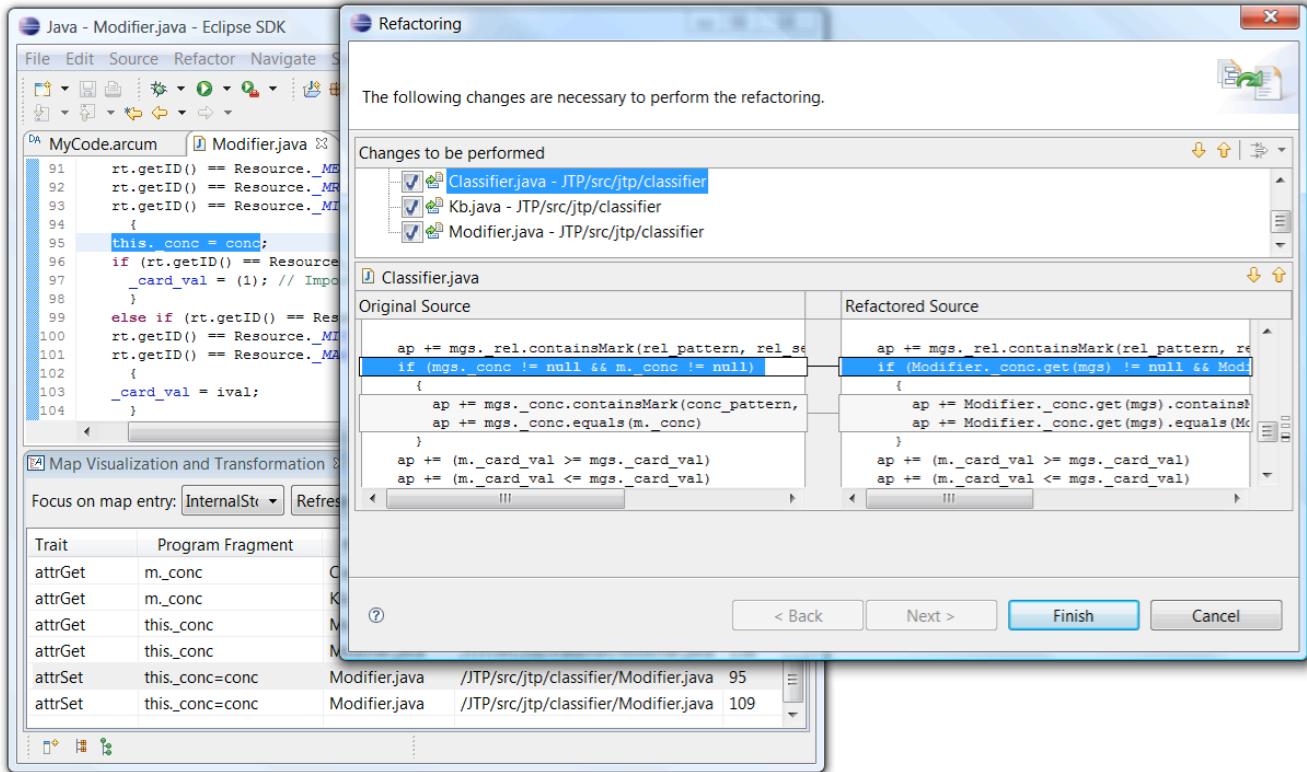


Figure 3: Refactoring in Arcum for Eclipse: The front-most window is a preview that shows the changes the refactoring would perform. In the background is the Eclipse environment itself, with an Arcum view at the bottom that shows a compressed view of the implementation’s scattered code fragments.

```
map {
  InternalField(targetType=Image, attrType=String,
    attrName="altText");
}
```

Figure 4: Map file 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.

object whose attribute is being set, valExpr is the value to which the attribute is being set, and setExpr is the entire expression that performs the set operation. Arcum variables, such as setExpr store references (i.e., pointers) to code fragments, and 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 *realization* of the attrSet trait. A trait is realized by providing a pattern that identifies the fragments of Java code that are instances of the trait. For example, the boxed pattern in the InternalField option shows how a regular assignment to a field becomes an instance of the attrSet trait. In our Image constructor example, line 6 of the Java code matches this pattern, and therefore becomes an instance of the trait. Similarly, the boxed pattern in the StaticMap option states which map operations become instances of the attrSet trait.

Patterns in the Arcum framework declaratively state the mapping between a crosscutting design idiom and the various fragments of Java code that implement an option. A key feature of Arcum is that these mappings are *bi-directional*: not only are the patterns used to build trait instances from Java code, but they are also used in the

other direction, to generate Java code from trait instances.

The directionality of the mapping is determined by how the mapping is instantiated and later changed. As declared in Figure 4, the InternalField implementation is the prevailing option at the beginning of our 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 our scenario, the altText field is initially implemented as a simple class field and the conceptual flow of information in Figure 5 goes in the clockwise direction, following the solid arrows. However, this refactoring can be run in either direction. The field assignment on line 6 in the original code is pattern matched into a trait instance, at which point the references setExpr, targetExpr and valExpr are bound. The newly constructed trait 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 trait patterns, the mapping can be changed later to perform the refactoring in the other direction. Because our approach explicitly and persistently identifies substitutable crosscutting entities and their prevailing options, their consistency properties can be continuously checked. This makes future transformations easier to perform because the checking aids code compliance. For instance, in our example, we would like to prevent the incorrect application of the refactoring to transform `f(this.altText=y)` into `f(Image.altText.put(this,y))`, because, as mentioned previously, the put method returns the *previous* value in the map. This requirement is checked by the interface with the requires clause (where `isSubExpression(e)` checks whether `e` is embedded in another expression). This requires clause can be checked continu-

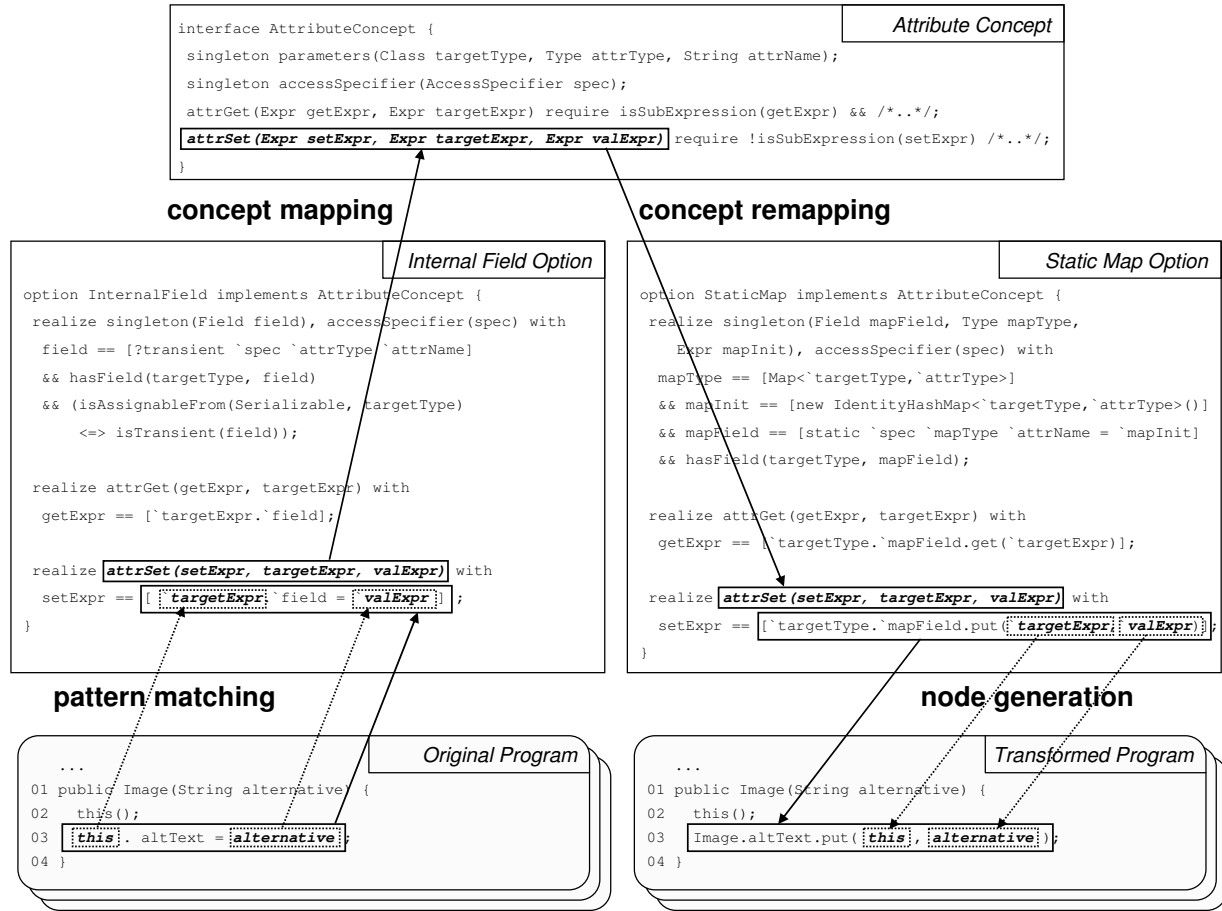


Figure 5: Overview of how Arcum transforms code into an alternative implementation. By changing the mapping, which describes the current option deployed, the code is automatically transformed in accordance with the substituted option.

ously, to make sure that developers don’t accidentally change code in a way that prevents the entity 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 slavishly preserving every detail of language-semantics-level behavior. We 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.

3. THE ARCUM LANGUAGE

The key construct of the Arcum language is the *trait*, which describes a tuple of Java program fragments (Section 3.1). All Arcum code appears in one of three declarations: an interface, an option, or a map. An interface (3.2) specifies the names and types of the traits common to all options that implement the interface. An option (3.3) provides concrete definitions of the traits in the interface by realizing them via *patterns* (3.3.1). Finally, a map (3.4) allows options to be parameterized for a particular application.

3.1 The Trait Construct

A trait is used to describe one distinct role, structure, or operation that occurs in a crosscutting design implementation (a concept

similar to a trait is referred to as a sub-concept in [15]). A trait can either be a tuple (called a *singleton trait*) or a set of tuples (potentially the empty set). Traits can have boolean *requirements* associated with them (described below, Section 3.1.1), which allow optional user-readable error messages to describe what the requirement expects.

The `AttributeConcept` in Figure 5 specifies four traits: `parameters`, `accessSpecifier`, `attrGet`, and `attrSet`. Because an Arcum interface is abstract, its traits are, too. Only a trait’s name, tuple member types, requirements clause, and optional error message may be specified here. These abstract traits are given concrete definitions via patterns specified in the options that implement an Arcum interface.

The `attrGet` and `attrSet` traits represent, respectively, the abstract operations of getting and setting the attribute, where both operations can occur in the program multiple times. The two singleton traits, `parameters` and `accessSpecifier`, represent unique program fragments that occur exactly once in the implementation. The unique program fragments in a singleton may sometimes be reifications of entities that are only abstractly present in the program.

Each trait 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 at-

<i>Predicate Name</i>	<i>Description</i>
isAssignableFrom(t_1, t_2)	Is t_2 equal to or a subtype of t_1 ?
hasMethod(c, m)	Does class c have a method m ?
isTransient(f)	Is field f declared as transient?
isJavaIdentifier(s)	Is the string s a valid identifier?
isReferenceType(t)	Is t a reference type?
isSubExpression(e)	Does e exist as a sub-expression?

Table 1: Example built-in predicates.

trSet trait 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 trait’s root fragment is an expression or statement it represents an operation. But traits can also express structural properties of code and other crosscutting forms. For example, one could declare a trait that represents “all declarations of type Vector.” Such a trait 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. [1]). Structural examples of traits include methods, fields, and annotations.

3.1.1 The Requires Clause

Programmers can add multiples requirement clauses with error messages to a trait. For example, the abstract parameters trait in Figure 5 can additionally declare:

```
require isJavaIdentifier(attrName)
catch error("'s' must be a valid Java ID", attrName);
```

The above requirement states that the name provided for the attribute must be a valid Java identifier.

In general, the Boolean condition provided in a require clause is evaluated for each trait instance. Conditions are expressed using a simple propositional logic, augmented with built-in primitives, examples of which are shown in Table 1. A require clause can have an optional error message associated with it, to provide a compile-time error message to the user in the event that the condition cannot be satisfied.

3.1.2 The Parameters Trait

The special parameters trait is a trait whose members are specified in the map file 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 5 has three parameters: a Class named targetType (the class that has the attribute), a Type named attrType (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, e.g., being hard-coded only for Strings 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 List could permit it.

3.2 The Interface Construct

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 design idiom’s interface and to centrally specify requirements that apply equally to all options that implement it.

Traits specified in the Arcum interface are abstract and all traits except for the special parameters trait must be concretely realized in all options that implement the interface. The special parameters trait is always abstract and is only allowed in Arcum interfaces (however, its options inherit it).

The accessSpecifier trait in Figure 5 is an abstract singleton with a single member, spec. This singleton trait is used by the AttributeConcept to simplify the parameters list. The spec member is used to 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 singleton depends on the current and expected options that the interface will modularize.

3.3 The Option Declaration

An option describes one complete implementation of a crosscutting design idiom 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 next section (3.3.1) describes how options use patterns to concretely realize the abstract traits specified in the interface.

3.3.1 Patterns

Options realize traits 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 back tick marks to identify Arcum variables inside the pseudo-code. For example, the InternalField option in Figure 5 uses the following pattern to match (or generate) all valid set expressions:

```
realize attrSet(setExpr, targetExpr, valExpr) with
  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 4. 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 realize an anonymous singleton trait to describe a program fragment that is specific to the implementation of the option. For example, the anonymous singleton in the InternalField option in Figure 5 has the single member field:

```
realize singleton(Field field) ... with
  field == [?transient 'spec 'attrType 'attrName] ...
```

Because this singleton is local to the option this field does not have to be present in any of the alternative options. The question-mark in the above pattern is a convenience notation for specifying that the transient modifier may or may not be present. It is equivalent to the union of two versions of field-declaration pattern: one with the keyword and one without. Section 4.2 discusses how Arcum determines which pattern to use for code generation.

3.3.2 Option Invariants

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

```
realize attrGet(getExpr, targetExpr) with
  getExpr == ['targetExpr.'field]
require
  isAssignableFrom(attrType, TypeOf(getExpr)) &&
  isAssignableFrom(targetType, TypeOf(targetExpr)) &&
  isSubExpression(getExpr);
```

Consider the `isSubExpression` requirement, for example. In Java, you cannot use a field access as a statement. Therefore, the expression statement `'this.altText;'` is rejected even though the expression statement `'Image.altText.get(this);'` is accepted. The `isSubExpression` requirement prevents the latter from being inadvertently written.

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

Options may also define static traits purely for checking implementation-specific details. As an example, in our substitution scenario from Figure 5, once the code has been refactored to the `StaticMap` option, we want to prevent programmers 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 static trait 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:

```
realize static anyAccess(Expr expr) with
  expr == ['targetType.'mapField...]
require attrSet(expr, _, _) || attrGet(expr, _);
```

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

3.4 Maps

Arcum maps are used to state which options are implemented in a program. Figure 4 shows a sample Arcum map. In general, an Arcum map 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 map format is that there is a separate file that documents some of the architecture of the program.

4. 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 traits and interface-level traits. 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 singleton traits. For example, when refactoring from the `InternalField` option to the `AttributeConcept` option in Figure 5, the `altText` field gets removed from the program because its AST node was pattern matched into an option-local trait. AST nodes that match into option-local traits are removed during transformation because by design these traits are option-specific—otherwise, the traits would be specified at the interface-level.
4. Construct new AST nodes using the patterns from the destination option's local traits and insert them into the program. In the refactoring scenario from Figure 5, the singleton trait `mapField` in the `StaticMap` option will add a declaration of a static `Map` to the program.
5. Replace each trait instance with a new AST node generated from the destination option's trait 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). We describe these two uses of patterns in Section 4.1 and Section 4.2, respectively.

4.1 Using Patterns for 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). Predicate expressions can be conjoined to the pattern to filter the matches further.

4.2 Using Patterns for 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 trait instances, but they are also used in the other direction, to generate Java code fragments from trait instances. As an example, the following pattern in the `StaticMap` option is used in our refactoring scenario to insert a call to the `put` method of the static map:

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

A trait 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 trait 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 Section 3.3.1 (and in Figure 5) shows two possibilities for the field-declaration: either the transient modifier is present or not.

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

```
&& (isAssignableFrom(Serializable, targetType)
    <=> 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, we use a generate-and-test approach: we 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 we 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.

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

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

into:

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

5. EVALUATION

To evaluate the Arcum framework we (1) implemented a prototype Arcum refactoring system for Java; (2) developed a complete `AttributeConcept` with three options; and (3) applied Arcum to three different change tasks and compared the ease of change with the state of the art in Eclipse and AspectJ [13].

5.1 Arcum Prototype Implementation

The Arcum prototype supports most of Java 5 except for generics with wildcards, and some aspects of anonymous inner classes. The prototype 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. Arcum comprises 11 KLOC of Java code. Arcum uses Eclipse’s parsers and type checkers for pattern matching and Eclipse’s factories for node generation.

5.2 Development of the Attribute Concept

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

We 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 design idiom. 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, we found that it was hard to get these right the first time. We also found that it wasn’t necessary. Our first version of the `AttributeConcept` interface was correctly parameterized, but its internals omitted all requirement checks. When we wrote an option, which described one particular implementation, we added explicit checks to the interface—the requirement invariance nature of Arcum drove this process (see Section 3.3.2). 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 we came across some serialization code, we realized that a general-purpose `InternalField` specification would have to account for this special case. If we had been just developing the related options for our own use, we 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 might also not think of this corner case—we added the necessary checks to insure that the attribute will be appropriately serialized in all options.

Examples of checks added have been given throughout the paper, 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 trait 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.

5.3 Change Tasks

We considered 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.

Task 1: Application of `AttributeConcept`. To get a preliminary feeling for whether Arcum imparts some of the benefits of modular substitution to crosscutting idioms, we put `AttributeConcept` to work. We chose the Polyglot framework [20], since we understand it well. Polyglot is an extensible compiler that heavily uses delegators and extension objects. To support these, 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.

We first externalized the storage of the `del` field. We instantiated an `InternalField` mapping for `del` and Arcum’s checks for conformance passed. We then changed it to instantiate the `StaticMap` option, thus triggering the refactoring and resulting in 13 substitutions. To measure the program’s new memory usage we compared its previous memory footprint from compiling an 80K line program to the modified version. In our 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 we 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 100KB for a program run that used 11MB total. Consequently, we changed the mapping to reinstantiate `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.

Task 2: Message Log Redirection. For a second test, we edited the source for the Arcum plug-in itself. The plug-in has a mode for sending to `System.out` debugging information. We considered a scenario where we 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 the console output from all other plug-ins as well.

The solution requires that the our plug-in uses one stream, while all other plug-ins continue to use `System.out`. Eclipse can perform this change more thoroughly than a standard textual find and replace: Eclipse allowed us at the semantic level to find all references to the field (which numbered just over 1,000) regardless of the whitespace formatting or static imports used. However, despite Eclipse's semantic level search, all modifications still needed to be made with a textual find and replace. This global change caught most instances, but another 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 trait with only one member (of type `Expr`) in it. Then, an option was written to realize this trait with all accesses to `System.out`. To test the option, we used Arcum's search view to display all program fragments that belonged to the trait. Once we were convinced the results were as expected, we 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, instead of only modifying the program's semantics through byte-code 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 when the need arises.

Task 3: Remove Control Coupling. For the final test, we 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. Our 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 we were able to

match all special null argument cases and change them into calls to the new method instead.

6. DISCUSSION AND FUTURE WORK

The foregoing has only scratched the surface of the Arcum approach. In this section we discuss some novel ways that Arcum might be usefully employed in software development. These uses suggest additional kinds of crosscutting design idioms that might be checked and refactored with Arcum, and are directions for future work.

Rapid Prototyping. The DJ library for Java makes extensive use of reflection to support a dynamic form of the visitor pattern [21]. The traditional visitor pattern only partially modularizes depth-first traversals over objects connected by the "has-a" relationship. In DJ, programmers describe a high-level strategy that specifies the source and destination types of the traversal. As a result, all of the method infrastructure required for the intermediate types can be omitted. Because the high-level strategy specifies only what is important about the traversal, it is less redundant and more flexible when the class graph changes.

However, the use of reflection to descend down object graphs entails a noticeable runtime overhead. Both implementations of the visitor pattern satisfy the same specification: traversing an object graph. Thus, this traversal should be encoded in an Arcum interface, with each implementation being a separate option. With both options in place, it is possible to begin development with the more flexible and convenient reflection-based visitor, and then use the normal visitor implementation once application development has settled down. Should development return to a rapid-prototyping phase, the options can be flipped again.

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 data-base 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.

Porting and Retargeting. The Arcum approach can also be useful in porting code bases. For instance, class library migration is an important problem addressed by Balaban et al. [1]: 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 it 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 should be explicitly synchronized.

In order for the Arcum framework to support such refactorings, its constraint checking system would need to be extended to include forms of data-flow analysis. This is a subject of future work.

Design Pattern Checking. Design patterns are another class of implementation techniques that could benefit from checking. 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. Looking further ahead, it may be possible for such errors to drive an automated refactoring that fixes the im-

plementation. Related work that addresses this problem includes Spine, a declarative language for checking design patterns [6].

Other implementation styles can also be checked. For example, an internationalization strategy could have rigid requirements for the ways string literals are used (such as always wrapping them around method calls). Arcum code can be written to check and enforce these rules. In the process, the potential exists later on for providing an alternative implementation.

Bug finding tools check for common coding errors and have the potential for finding bugs that code reviews and test cases miss [23]. 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.

Aspects and Fluid AOP. Extending Arcum to support AspectJ 5 would allow for more sophisticated kinds of refactorings, such as suggested by Relationship Aspects [22]. Due to Arcum’s bi-directional semantics, fluid aspect-oriented programming [12] is possible: code could be edited and modified in a crosscutting implementation, but refactored as needed into a localized aspect to enable changes to the concern to be expressed locally; after the change is made, the code could be refactored back into the scattered implementation.

Hybridized Options. 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.

7. RELATED WORK

Arcum is a departure from the role-based refactoring work of Hannemann et al., which permits programmers to build macro-refactorings from micro-refactorings [11]. The basic idea is to support the refactoring of crosscutting entities 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. Abstract roles are essentially the same as Arcum’s abstract traits. The approach is a traditional refactoring approach in that it does not externally specify the underlying interface or implementation options, thus not providing continuous checking or bi-directional refactorings. Marin et al. take a similar approach, although they assemble macro-refactorings from micro-concerns rather than roles [17].

AOP languages like AspectJ can manifest many crosscutting design idioms, including many design patterns, as modular abstractions [10]. However, when dealing with existing tangled code, this requires refactoring the existing code to modularize the tangled code into an aspect. Arcum can specify and check implementations without having to modify the code in any way. Of course, when Arcum is extended to support AspectJ, Arcum can assist programmers with refactoring crosscutting code into aspects (and back again).

Balaban et al. employ declarative semantic notations for automatically retargeting code libraries in large code bases [1]. 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 crosscutting and non-modular nature of the implementation of software features, which are often crosscutting [16]. 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, allowing for better-suited variants.

Simonyi’s Intentional Programming (IP) [25] is related in spirit to the goals of Arcum.² IP 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 [14]. The code transformations discussed were not bi-directional in nature and are 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, stopping short of a notion of a (persistent) option. 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 [15]. Concepts were not persistent like Arcum options, however, and transformations were explicitly defined as predicate-action pairs, not inferred from patterns, thus limiting transformations to being one-way. In many respects this supports user-programmable role-based refactoring. A more recent work in this area is the DMS system, which is similar to Kozaczynski et al. but has a much wider scope [3].

Arcum’s ability to express program properties, both in patterns and in require clauses, is related to the work on static program analysis tools, such as SLAM [2], and PDL [19]. Better support for static analysis in Arcum could lead both to richer source matching and error detection.

The bi-directional nature of Arcum relates to the area of multiple-views into a program, for example the recent work of Black et al. [5], Eisenberg et al. [8], and Mens et al. [18]. Different views demonstrate different aspects of the program and can thus be more expressive than the other in different circumstances. A programmer using Arcum can transform a crosscutting design idiom implementation into the option where the change can be most easily expressed. Once the change is made, the implementation can be changed back to the previous option, bringing all changes with it.

8. CONCLUSION

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

²The 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” [7, p.333].

Arcum expands the opportunities for modular analysis and substitution for such idioms. 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 design idiom into an interface and an option. An option uses semantic patterns that correspond to traits 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.

To evaluate the Arcum approach we developed a prototype Arcum's transformation engine, completely developed an interface with three options, and applied it to the Polyglot code base. We found that Arcum provided the feel of substitutability to the crosscutting design idiom.

9. ACKNOWLEDGMENTS

We would like to thank Andrew P. Black and the anonymous reviewers, who reviewed an earlier draft of this paper. Our thanks also go to the Programming Systems Group at UCSD, for lively discussions about Arcum.

10. 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] 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.
- [3] 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.
- [4] K. Beck and C. Andres. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, second edition, November 2004.
- [5] A. P. Black and M. P. Jones. The case for multiple views. In *Workshop on Directions in Software Engineering Environments, ICSE 2004*.
- [6] A. Blewitt, A. Bundy, and I. 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.
- [7] D. C. Dennett. *Consciousness Explained*. Back Bay Books, 1992.
- [8] 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.
- [9] W. 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.
- [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] G. Kiczales. Aspect-oriented programming: The fun has just begun. In *Software Design and Productivity Coordinating Group – Workshop on New Visions for Software Design and Productivity: Research and Applications*, Nashville, Tennessee, Dec. 2001.
- [13] 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.
- [14] 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.
- [15] W. Kozaczynski, J. Ning, and A. Engberts. Program concept recognition and transformation. *IEEE Trans. Softw. Eng.*, 18(12):1065–1075, 1992.
- [16] 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.
- [17] M. Marin, L. Moonen, and A. 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.
- [18] K. Mens, T. Mens, and M. Wermelinger. Maintaining software through intentional source-code views. In *SEKE '02: Proceedings of the 14th international conference on Software engineering and knowledge engineering*, pages 289–296, New York, NY, USA, 2002. ACM Press.
- [19] C. Morgan, K. D. Volder, and E. Wohlstadtter. 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.
- [20] N. Nystrom, M. Clarkson, and A. Myers. Polyglot: An extensible compiler framework for java, 2003.
- [21] D. Orleans and K. 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.
- [22] D. J. Pearce and J. Noble. Relationship aspects. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, pages 75–86, New York, NY, USA, 2006. ACM Press.
- [23] 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.
- [24] M. 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.
- [25] C. Simonyi. The death of computer languages, the birth of intentional programming, 1995.