The Techniques Programmers use to Cope with Crosscutting using 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

At their most essential, aspect languages, program analysis tools, and refactoring tools attempt to give programmers mechanisms to make it more cost effective to manage the crosscutting behavior in their programs. Arcum is a tool to help manage crosscutting that lets programmers define custom program checks and program transformations, using a declarative language [22]. In this paper we present a study aimed at investigating how programmers use Arcum for managing the complexity of crosscutting. In particular, we recorded and transcribed three pairs of programmers performing a variety of tasks using Arcum. By informally analyzing the language in the transcript, we identify the metaphors that the participants used to think about crosscutting design idioms, and the development styles that they used to build solutions. Based on these observations, we reflect on how the use of Arcum relates to traditional programming and AOP approaches, and propose improvements to be made to the development environment to help programmers handle the challenges of crosscutting code. A key observation was that the programmers actively and ingeniously sought ways to force the tool to give early feedback, suggesting that AOP tools like Arcum could do more to support early feedback.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Modules and interfaces*; D.2.6 [**Software Engineering**]: Programming Environments—*Integrated environments*

General Terms

Design, Experimentation, Human Factors.

Keywords

Crosscutting, refactoring, user study.

1. INTRODUCTION

As a software project matures, it may require changes not anticipated by the system's designers nor the programming environment's developers. Thus, performing such changes requires either tedious and error-prone manual modifications or the ad hoc application of existing tools, such as program analysis, refactoring, and text searching tools. An alternative is to extend the programming environment's language capabilities, such that each change can be described once and then automatically applied when required. Many aspect-oriented tools and meta-programming systems take this approach.

One instance of this approach is embodied in Arcum [22], a tool for declaring crosscutting design idioms and their implementations. By allowing programmers to seamlessly switch between different implementations of the same crosscutting idiom, Arcum can modularize many crosscutting design idioms.

In this paper, we study how three pairs of experienced programmers performed a variety of tasks using Arcum. From our analysis of both the words used by the participants and the different approaches taken to solve each problem, we present the metaphors that the participants used to think about crosscutting code, and the development styles that they used to address the difficulties of crosscutting.

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

- The Arcum process is a successful way for programmers to reason about the set of entities comprising a crosscut in isolation. We show how the participants, by using previously existing Arcum examples and feedback from the IDE, were able to develop working Arcum code. We observed two distinct styles by which programmers arrived at their working solutions, one based on copying existing examples and another based on incrementally adding code to an always-executable form. The two styles are not mutually exclusive and we believe these styles were chosen in order to get feedback from the tool as soon as possible, assisting the formation of the mental model of the crosscutting design idiom.
- Not surprisingly, the process of writing programs that describe crosscutting carries with it not only some of the challenges of regular programming, but further challenges of its own. For example, programmers have to think about not just the crosscutting design idiom's implementation, but also the description of the implementation, and the different forms that alternative implementations may take. We show how these challenges of meta-level programming manifested themselves in our study, and how programmers used Arcum to address them. A common mistake was to confuse types and entities of those types.
- IDE support is essential for understanding crosscutting as it appears in real programs because the scattering and tangling inherently covers more code than comprehensible in

a glance. For example, we observed that participants relied upon Arcum's pattern matching visualizations, Arcum's transformation preview pane, and various error reporting capabilities in the Java compiler as well as the Arcum compiler.

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

After describing Arcum (Section 2) and the study itself (Section 3), we discuss how the test subjects approached crosscutting (Section 4). We then discuss the metaphors and techniques used by the subjects to help them understand instances of crosscutting design idioms, such as abstracting its essential features into a workable mental model (Section 5). In analyzing the subjects' activities, we observed two development styles that participants used to construct their solutions, one based on copying existing solutions, and one based on incrementally building a solution from scratch (Section 6). We discuss the challenges and techniques used when writing custom checks or custom refactorings—which requires thinking about code not just in the concrete form in which it exists, but also the form in which is may exist (Section 7). Finally, we conclude with some preliminary design decisions to help guide the development of aspect-oriented tools (Section 9).

2. BACKGROUND: THE ARCUM CONCEPT FRAMEWORK

Arcum is a framework for declaring and performing user-defined program checks and transformations for Java programs, with the goal of increasing automated refactoring opportunities for the user [22, 23, 24].

By using Arcum, a programmer can view the implementation of a crosscutting design idiom—such as a design pattern—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.

Given the declarative descriptions, Arcum can infer the transformation steps necessary to refactor from one option to a related option. Arcum options are parameterized so that they can be applied multiple times in different contexts. When not used for refactoring, Arcum options allow properties of the program to be checked: Because the option describes one correct implementation of the idiom, deviations from this correct form are automatically identified. Arcum allows the option writer to insert additional checks, with custom error messages, to guide the user of the option to a correct implementation.

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 using a database of relations. Arcum's database contains relations associated with the program being analyzed, including relations such as isA and hasField, and with special relations to cover Java syntax, such as method invocations, field references, and type declarations. These special relations are written in a pattern matching style in the form of quoted Java program fragments. Arcum variables can be used as placeholders in the quoted patterns through an escape mechanism. Similar to Prolog and other logic languages, new relations can be defined by the user. When these relations are present in both the interface and its implementing options, they are called *concepts*, because each program fragment that belongs to the relation represents an idea that exists at a high level.

Arcum is delivered as a plug-in for the Eclipse IDE and leverages the Java Development Tool's compilation and refactoring components. Figure 1 shows the "Fragments View," supplied by Arcum, which allows programmers to view the program fragments that match each concept of the crosscutting design idiom.

3. STUDY DESCRIPTION

We chose to perform a qualitative, exploratory study, because there is little experience with programmers using Arcum, and we wished to discover the basic phenomena and issues revolving around Arcum's use in modularizing crosscutting design idioms. Our expectation was that programmers with experience writing large programs could understand how to effectively use Arcum (Section 3.3).

For the study, we recruited six subjects (Section 3.1), who worked worked in pairs on tasks such as changing a program and writing checks to verify properties of the program. We provided to the subjects written instructions that described the sequence of tasks to perform (see Section 3.2) together with short reference materials for the Arcum language. The experimenter observed these subjects over two sessions, which took place in a quiet office environment.

We used pair programming in order to capture natural conversations, closer to what might occur outside of an experimental setting [17]. Pair programming is common in many real-world settings, especially on complex tasks like those that might be solved with Arcum. An alternative would have been to use individual sessions with each programmer, but that would have required either the less natural "please think out loud" technique, or constant questioning from the experimenter, which could introduce bias through tone of voice and other cues.

Each pair's audio was captured along with the contents of their computer screen and file system output. (TechSmith's Camtasia was used for the recording.) The audio component of the sessions were transcribed and we then analyzed the subject's use of language, in order to see the kinds of metaphors the subjects used and how they thought about the process. This analysis led to insights based on their expectations and intuitions, together with what kind of intellectual tools they use, such as abstraction, to cope with the change tasks.

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

3.1 Study Subjects

All six study subjects were graduate students in the computer sci-

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

Table 1: Study subject's industry experience, whether he or she has used Eclipse before, and programming languages known.

ence department and experienced programmers. Table 1 shows the backgrounds of the subjects. We chose experienced programmers because part of the intention of Arcum is to enable experienced programmers to write transformation and checking libraries that could be used by a wider audience.

3.2 Study Tasks

The study comprised two sessions for each pair of subjects. The first session was a tutorial that covered Eclipse and the Arcum plug-in, and included step-by-step guides for completing the tasks (Section 3.2.1). The second session was held on the following day and covered program transformation and checking tasks without any step-by-step guides (Section 3.2.2). The subjects were given 90 minutes to complete the first session, and 60 minutes to complete the second session.

3.2.1 Tutorial Session

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

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

The subjects were free to use Eclipse as they saw fit to perform the change. Even though the change was simple, we devised the code so two bugs would occur if the changes were made carelessly: a NullPointerException could occur if a corner case in the program was not identified (discussed in Section 4.1) and a semantic change was possible due a particular method call not being a perfect substitute for a Java operation (discussed in Section 7.1).

Arcum Training Task. The next task gave the subjects practice with executing Arcum code and provided the background for writing code in the language itself. A complete code example was provided that contained one interface representing the attribute idiom and two options representing the alternative implementations (internal field versus external map). The attribute interface has two concepts, attrGet and attrSet, which abstracts the attribute read and write operations. Arcum allows a programmer to switch between the two options, where one option, for example, represents an attribute read as a field reference, while the alternative option represents an attribute read as a method call. This attribute example—introduced in a previous paper [22]—automates the refactoring performed for the *Manual Transformation Task* and also demonstrates several features of the Arcum language while being a short example.

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

Custom Check Creation Task. After being given the step-by-step guide for inserting extra checks, the subjects were asked to insert another check. The purpose of this check was to automate the detection of the bug discussed in Section 7.1.

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

3.2.2 Advanced Session

The session using Arcum without step-by-step instructions used the HTML renderer component of the Lobo project [32]. Lobo is a complete web browser written in Java. Lobo was chosen because it was the top desktop application project available from SourceForge (a repository for open-source code) that was written entirely in Java and compilable with Eclipse. Lobo is also well-written and rich with crosscutting design idioms.

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

Change StringBuffer to StringBuilder Task. Next, the subjects were asked to migrate the Lobo codebase from using the always-synchronized java.lang.StringBuffer class to the more efficient java.lang.StringBuilder class (this is an instance of the class library migration problem [1]). Although this change could easily have been made with a global text-based find-and-replace (because the two classes have the same API and neither of them require Java import statements), we wanted to see how programmers would solve such a transformation with Arcum. Accomplishing this task with Arcum requires recognizing and replacing program fragments that belong to different syntactic categories (namely, type declarations and constructor call expressions).

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

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

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

This special usage can be considered a crosscutting design idiom: Any changes to the policy (e.g., of how the log is acquired, or which log is used) would require global changes. One simple property of this crosscutting design idiom that can be checked is if the argument given has, in fact, the correct name. (For example, a copy and paste error would lead to the logs of one class to be written to

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

Table 2: How each group performed the tasks over the two sessions. A '+' indicates when the subjects ran out of time and could not fully complete the task. The '*' indicates that the task was completed with minor assistance.

the log of the copied class.) The instructions for this task required the subjects write Arcum code that could check for this property.

After the second session, the subjects also participated in a separate post-study interview.

3.3 Performance of the Tasks

All three groups successfully completed the tutorial session in the time alloted, but no group fully completed the advanced session. Table 2 shows the time it took for each group to complete each task. All groups finished the tutorial session early but used all of the time alloted for the advanced session. Times for the advanced session do not add up to a full 60 minutes due to group start up delays.

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

Group C made the most progress on the *Check Logging-Idiom Task*. Perhaps it was not co-incident that Group C spent the most time on the task compared to the other two groups: Group C had almost twice the time, at 30 minutes versus 16 minutes, because they completed the previous two tasks relatively quickly and the session started on time. The specific challenges of this task are discussed in Section 7.2.

3.4 Threats to Validity

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

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

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

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

4. REASONING ABOUT CROSSCUTTING

In the strictest sense of the term, no module could utilize another module without some form of crosscutting, because the module's *interface* must be known by all modules that need to use it [2, 29]. But not all forms of crosscutting are equal: By their nature, well-written interfaces are stable [19], so when elements of the API (such as method names) crosscut the program, they do not become liabilities when that module's implementation needs to change. This section focuses on the kinds of crosscutting that do not naturally fit into stable interfaces and thus require reasoning over several different modules.

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

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

4.1 Using Build Errors as a Guide

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

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

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

However, the errors introduced from the change were type errors and did not have a perfect correspondence to references: The accesses to the `next` field had different types now that the `List` class's `next` field changed its type from `List` to `Map`. Yet, expressions with

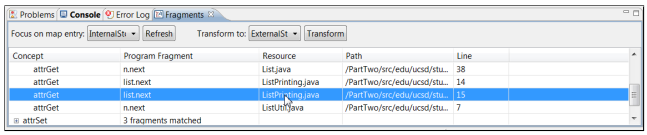

Figure 1: Arcum's Fragments View: Shown are four different matches in the program that represent instances of the attrGet (attribute access) operation.

values of these two types can exist in the same code context. For example, the following loop was in the sample program:

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

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

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

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

4.2 Making Direct Queries with Arcum

During the advanced session, the subjects were asked to reason about several instances of crosscutting, such as the scattered use of the StringBuilder class, or the scattered instances of the logging idiom. Figure 1 shows Arcum's Fragments View, which was utilized by the subjects in many of these instances to visualize the matches.

In the post-study interviews, one subject compared Arcum to a "semantic grep," a comparison that holds in several regards: The Fragments View provides programmers with a compressed view of one aspect of the crosscutting design idiom, much like the output of grep. Such compressed views can help programmers focus on areas of interest without having to read unrelated code [8]. Further, much like the grep command, Arcum can be used for pattern matching. However, Arcum's pattern matching is based on desugared AST nodes (instead of characters in a text file) and can take into account type information. The desugaring of Arcum's matcher was noticeable during the *Review Code Examples Task*:

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

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

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

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

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

> *A2: Oh, those are, oh we were looking at the wrong thing. Cool. But now we know there are ones we're not getting too, right, because...*
>
> *A1: [..] it can take various sorts of arguments*
>
> *A2: Right.*

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

4.3 Confusing Definition with Reference

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

> *B2: So what are we looking for an expression? Actually that's not even an expression. Right now we're just looking for a type.*
>
> *B1: I wonder if we can just hit 'type.' Sure, let's try it, see what happens.*
>
> *B2: Do you think that will give us occurrences of the name of that class or it'll just give us definitions of that class?*

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

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

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

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

As suggested in Section 4.1, the definition/reference relationship can be given more importance in the environment through added hyperlinks between the two. Such two-way links are already part of the AspectJ Development Tools support for viewing the relationship between join-points and advice. These guides could be taken a step further by creating a tool in the environment that suggests code (e.g., patterns) that will match the Java code currently highlighted by the user, and also include a link back to the full results of each proposed pattern. In the case of AspectJ, a user would select a method call in the program, and a separate view would generate code for the different pointcuts possible to match that join-point. The generated code could then be copied, or explored for the other matches it creates.

4.4 Using Reference Materials

Reference material is another source of information that participants used to help them reason about crosscutting idioms. In particular, we observed subjects using API documentation (Section 4.4.1) and forming models based on the texts discovered in the program (Section 4.4.2).

4.4.1 Using the Documentation

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

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

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

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

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

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

4.4.2 Using Error Message Texts

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

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

> *A1: Okay, so we have to take in some kind of arguments. Can we see where arguments are actually being given in the? Where it's being run? Cause that's like [...] Commandline args?*
>
> *A2: Yeah, so since it didn't say "panic no args given". There's.*
>
> *A1: Yeah, so it must be. It must be getting some sort of args.*

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

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

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

We also observed that a properly written error message aided the reasoning of the program. For example, the following check in Arcum was provided to the subjects:

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

Here, the value returned by the read operation on an attribute must be used, otherwise it is flagged as an error. Such a check is useful because it is likely an error if an attribute is read but not used. The subjects were asked in the *Custom Check Creation Task* to write a similar check, but this time to check that the value of a *write* to an attribute is *not* used. The purpose of this check is to prevent the case discussed in Section 4.4.1—where the put method returns the previous value in the table—by restricting all code forms to the lowest common denominator. Thinking at this high level made it easy for the subjects to produce the correct solution:

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

5. ABSTRACTIONS OF CROSSCUTTING

Through the process of reasoning about the crosscutting of an idiom, a mental model of the crosscutting is formed in the programmer's mind. Because Arcum's interface and option constructs are modeled after the concepts of modularity, we hypothesized that these constructs would provide a natural form for expressing the crosscutting. Arcum's notion of creating an interface for crosscutting code was partially inspired by our group's previous work on XPIs in AspectJ [9, 28]. We found some support for our hypothesis, but we also identified cases where bad habits in the context of modular design (such as poor naming choices) remained difficulties in the context of Arcum.

5.1 A Decompositional Model

Arcum enables a refactoring operation to be decomposed into two options with a common interface. As a result, the complete transformation can be broken down first by thinking about the option that describes the current implementation as a search:

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

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

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

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

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

> *C2: Know what we should do? We should write another Arcum file that transforms this Arcum file to the file we want.*
>
> *C1: But it's gonna be adding things so we can't really do that, it's not a refactoring it's an adding, so...*

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

5.2 An Overloading Model

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

> *A1: So the options I think are, basically, it's sort of an overloading.*
>
> *A2: Implementing the field or whatever it is.*

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

5.3 Patterns as Abstractions

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

> *A1: So we're going to import this...*
>
> *A2: Import it into what? Import it into Arcum? Oh yeah, I guess so.*

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

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

6. DEVELOPMENT STYLES

As indicated by Table 1, the participants of our study have diverse backgrounds in terms of their previous programming experience, their previous knowledge of Eclipse, and their familiarity

with Java. Despite this diversity in background, we noticed a common theme in their approach to dealing with crosscutting idioms: They all focused heavily on *getting feedback early*.

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

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

6.1 Reuse of Uses

The first development style we observed involved inspecting, copying and then modifying previous Arcum code in order to quickly get a solution that could be tried out immediately, with the possibility of later refinements. This idea of using already existing examples to guide the development of code with unfamiliar constructs has is known as reuse of uses [21].

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

> *A2: I wonder if we can like copy and paste.*
>
> *A1: Well we can certainly start with that.*

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

> *B2: How we wanna wrap in function call... so let's look at the ExternalStorage implementation. Where is that, farther down?*
>
> *B1: Oh, it's down here, yeah.*
>
> *B2: Right? It's almost like analogous to...*
>
> *B1: Yeah.*
>
> *B2: Internal/external thing.*

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

Figure 2 shows Group B in the process of editing a copied version of an option. One can see how the participants have split the window vertically, with the original code on the left (which they also wrote in this case), and the edited copy on the right.

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

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

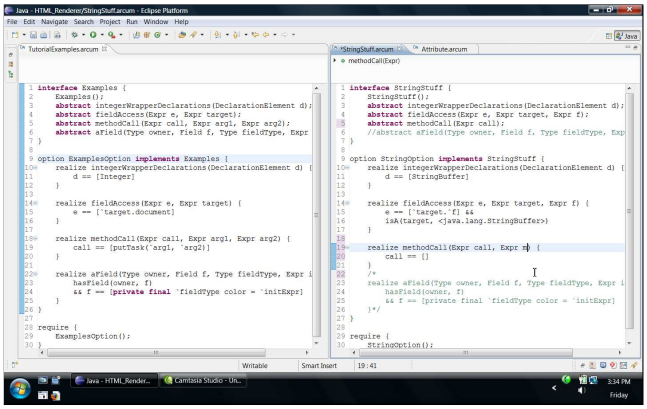

Figure 2: A direct example of reuse of uses by Group B.

6.2 Incremental Exploration

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

> *A1: Let's start the null case and see if we can build up from there.*
>
> *A2: Sure I think that sounds reasonable.*
>
> *A1: Okay so we'll have an option that realizes nothing. At least give us an interesting error message probably.*

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

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

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

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

> *A2: Eleven of one, I guess we can add the two rule now [..] and see if any match the two [..] or we could do the transformation and if it doesn't compile we can undo it*
>
> *A1: Yep! ... Let's take a look at what it actually turns them into.*

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

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

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

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

Furthermore, much like using the compiler warnings discussed in Section 4.1, a trial-and-error approach may not capture all problems. For example, if some important case is forgotten, but this case is decoupled from a type checking point of view from the other cases, then the Java type checker will not find the refactoring omission. Knowing what is important to refactor or not is similar to the challenges of modularity and knowing what is stable or not [19].

6.3 Improving Arcum Development Style

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

Pattern Tester. To give the developer early feedback on whether or not patterns work correctly, a pattern-testing tool would be useful. Such a testing tool could allow the developer to try patterns in the IDE and browse through the matches without having to build all the surrounding Arcum code. This tool would improve both development styles: in the reuse-of-uses style, it would allow the developer to test the patterns before putting them into the copied version of the Arcum code; in the incremental development style, it would allow the developer to try patterns out before having to write the boiler plate Arcum code.

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

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

7. REASONING ABOUT SEVERAL POSSIBILITIES

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

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

7.1 Thinking of Before and After

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

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

> *B2: I should have been commenting out the other stuff.*
>
> ...
>
> *B2: [typing] list.next.put(n, result)... and now we can put list.get, right? I'm just gonna comment this out.*
>
> *B1: OK, yeah.*
>
> *B2: Cause I don't know if I've gotten this right.*

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

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

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

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

Figure 3: (A) The closest code written by any of the groups to check proper log initialization; and (B) The change necessary to make it correct: moving the conjunct into a require.

> *A2: So nice. OK, and then we need [..]*
>
> *A1: Two options*
>
> *A2: Yeah one that will actually map what we have, and one that will map, match what we want.*

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

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

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

7.2 Thinking of Correct and Incorrect

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

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

Figure 3(A) shows the code of Group C, which got the closest to the correct implementation, and Figure 3(B) shows the correct one. The only difference is the location of the predicate starting with init ==. If the predicate is placed in the realize clause, then it becomes an additional *pattern matching constraint*, which narrows the set of matches that are found (without ever generating an error message), whereas if it is placed in the require clause, it becomes a *checked constraint*, which gets checked *after* the pattern matching has been performed (and leads to an error message if violated). The participants did not make this distinction.

One possible way of characterizing the problem is that pattern matching is more about the "before and after" metaphor, whereas the require clause is more about the "various incorrect versions" metaphor. The question then becomes: did the participants simply not distinguish between these two metaphors? or did they distinguish between the metaphors, but were not able to figure out how to express the distinction in Arcum?

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

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

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

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

8. RELATED WORK

Arcum is related to a host of IDE tools and user studies evaluating the affordances provided by such tools.

8.1 User Studies

Sillito et al. studied programming in Eclipse focused on the questions programmers ask when modifying programs [25]. Part of the study used pair programming in order record conversations to be later analyzed. This analysis gave insights into how programmers understand a system and what they need to know in order to make

modifications. Their study had a wide focus, intended to help guide the creation of software tools and tutorials, while our study was focused specifically as an evaluation of Arcum.

Robillard et al. investigated the process programmers use to understand code before they make changes to it and found that programmers who invested more time in making the most accurate model of the program were the most successful [20]. For example, the more lines of code a programmer examined (rather than skimmed) the higher the rate of success. Their study did not record the audio portion of programmer activities, and thus is was natural to use individual programmers instead of pairs of programmers. The focus of our study was the metaphors programmers use instead of a comparison of successful and unsuccessful programmers.

Ko et al. studied software changes performed in Eclipse and they found that much of the effort of reasoning about a maintenance task was navigating between scattered code dependencies and inspecting tangled code unrelated to the change [12]. The kinds of program changes examined were either bug fixes or adding additional features to the program, so the nature of the changes are separate from Arcum's focus on crosscutting design idioms. Storey et al. recognized in a large programmer study the different approaches programmers use to understand programs based on the different affordances available to them, and concluded that inspecting code dependencies was the most useful to programmers [27].

Murphy et al. argue for the structure of crosscutting tasks to have a concrete representation in the IDE to guide further changes [18]. Arcum's approach for creating structure is through the definition of options when the software system itself either does not or cannot modularize a design decision.

8.2 Languages

Arcum's general philosophy is common to many other works: "Improve programming by letting programmers better express their intentions to the environment." Examples include Explicit Programming [6], Presentation Extension [7], Metaprogramming [31], and Intentional Programming [26]. 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.

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.

DRIVEL is a program enhancement system using generative techniques on top of an aspect-oriented language [30]. 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.

8.3 Tools

The iXj program transformation system for Java allows for pattern matching similar to Arcum's concept construct [5, 4]. 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.

Some of the programming tasks given to the subjects are instances of the class library migration problem, which is encountered when code needs to be ported to use a different library [1].

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. Marin et al. take a similar approach, although they assemble macro-refactorings from micro-concerns rather than roles [16].

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 [3].

9. CONCLUSION

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

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

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

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

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] C. Y. Baldwin and K. B. Clark. *Design Rules: The Power of Modularity Volume 1*. MIT Press, Cambridge, MA, USA, 1999.

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] W. G. Griswold, K. Sullivan, Y. Song, M. Shonle, N. Tewari, Y. Cai, and H. Rajan. Modular software design with crosscutting interfaces. *IEEE Software*, 23(1):51–60, 2006.

[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] A. J. Ko, H. Aung, and B. A. Myers. Eliciting design requirements for maintenance-oriented ides: a detailed study of corrective and perfective maintenance tasks. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 126–135, New York, NY, USA, 2005. ACM.

[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. 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.

[17] N. Miyake. Constructive interaction and the iterative process of understanding. *Cognitive Science*, 10(2):151–177, 1986.

[18] G. C. Murphy, M. Kersten, M. P. Robillard, and D. Cubranic. The emergent structure of development tasks. In *ECOOP*, volume 3586 of *Lecture Notes in Computer Science*, pages 33–48. Springer, 2005.

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

[20] M. P. Robillard and W. Coelho. How effective developers investigate source code: An exploratory study. *IEEE Trans. Softw. Eng.*, 30(12):889–903, 2004.

[21] M. B. Rosson and J. M. Carroll. The reuse of uses in smalltalk programming. *ACM Trans. Comput.-Hum. Interact.*, 3(3):219–253, 1996.

[22] 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.

[23] M. Shonle, W. G. Griswold, and S. Lerner. Addressing common crosscutting problems with arcum. In *8th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, November 2008.

[24] M. Shonle, W. G. Griswold, and S. Lerner. When refactoring acts like modularity: Keeping options open with persistent condition checking. In *Second Workshop on Refactoring Tools*, October 2008.

[25] J. Sillito, G. C. Murphy, and K. D. Volder. Questions programmers ask during software evolution tasks. In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 23–34, New York, NY, USA, 2006. ACM.

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

[27] M.-A. D. Storey, K. Wong, and H. A. Müller. How do program understanding tools affect how programmers understand programs? *Sci. Comput. Program.*, 36(2-3):183–207, 2000.

[28] K. Sullivan, W. G. Griswold, Y. Song, Y. Cai, M. Shonle, N. Tewari, and H. Rajan. Information hiding interfaces for aspect-oriented design. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 166–175, New York, NY, USA, 2005.

[29] K. J. Sullivan, W. G. Griswold, Y. Cai, and B. Hallen. The structure and value of modularity in software design. In *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 99–108, New York, NY, USA, 2001. ACM.

[30] 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.

[31] 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.

[32] Lobo, 2008. http://lobobrowser.org/.