Using Metaphors from Natural Discussion to Improve the Design of Arcum

Macneil Shonle
Department of Computer Science
The University of Texas at San Antonio
San Antonio, TX 78249-0667
mshonle@cs.utsa.edu

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

Abstract

In this paper we present an exploratory pair-programming study aimed at investigating how programmers use a tool and language designed for performing crosscutting change tasks. Through a qualitative analysis of the pairs' discussions, we identify the metaphors that the participants used to think about crosscutting change tasks, which allowed us to infer their expectations. The metaphors of particular interest were the comparisons participants used to describe their approach in terms of other meta and non-meta programming tasks. From this analysis, we identified challenges the participants encountered in writing custom checks and refactorings.

1. Introduction

Arcum allows Java programmers to extend the development environment to support custom checks and refactorings for crosscutting design idioms [8, 9]. By using Arcum, a programmer can view the implementation of a crosscutting design idiom—such as a full design pattern or simply recurring programming idioms—as if it were a module.

The usability question for Arcum is if its meta nature can be approached by competent programmers. Arcum is meta in nature because programmers use the Arcum language to describe transformations to programs. Arcum's solution for dealing with this complexity is to use the principle of modularity to map programmers' high-level conceptions of crosscutting design idioms to concrete implementations. When two such mappings are provided, the transformation operations necessary to refactor from one implementation to another can be inferred by Arcum. Because modularity can help programmers decompose designs, our hypothesis is that modularity can also help decompose change tasks, reducing the burdens of its meta nature.

We chose to perform a qualitative, exploratory study, because there is little experience with programmers using Arcum. We used pair programming in order to capture natural conversations, closer to what might occur outside of an experimental setting [5]. From the natural discussion, we focused in particular on the metaphors participants used as a window into their thought processes and expectations, continuing from the work of previous studies on metaphors in programming [1, 3]. (Our analysis is preliminary and informal in nature. It is the subject of future work to apply Grounded Theory [12] to new Arcum studies.)

With this analysis, we make several observations: (1) Writing a program that describes crosscutting can carry with it not only some of the challenges of regular programming, but further challenges of its own—for example, a common mistake was to confuse types for entities of those types; (2) 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; and (3) There are opportunities to improve the Arcum system and related tools, based on the activities and areas of confusion we observed the participants of the study experience while coping with crosscutting concepts.

The remainder of this paper is structured as follows: In Section 2 we describe the Arcum system in more detail; in Section 3 we describe the study conducted; in Section 4 we discuss our findings; in Section 5 we discuss related works; and we conclude in Section 6.

2. The Arcum Concept Framework

Arcum employs a declarative language to let programmers describe an implementation of a design idiom. Descriptions in Arcum are composed of `interface` and `option` constructs. An `option` describes one possible implementation of a design idiom, and a set of `options` relate 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`, resulting in a form of modular substitution for non-modular program fragments. 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 users 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. Arcum supports relations associated with the program being analyzed—e.g., `isA` and `hasField`—including special relations to cover Java syntax, such as method invocations, field references, and type declarations. For example, one can write a simple pattern matching expression in Arcum in order to capture all references in the program to `System.err`:

```
expr == [System.err]
```

By writing an alternative pattern in another `option`, all of these references could be refactored to, for example, a call to access a custom error log (`ErrorLog.getLog()`). Note that this example is different from the Encapsulate Field refactoring because the `System.err` field itself is not changed, only the *references* to it are changed. The scope of changes made by Arcum can be fully specified, for example, by applying the change to only a particular `package` in the program.

Arcum is delivered as a plug-in for the Eclipse IDE and leverages the Java Development Tool's compilation and refactoring components.

3. Study Description

For the study, we recruited six participants, who worked in pairs on tasks, including changing a program and writing checks to verify properties of the program. We gave the participants written instructions that described the sequence of tasks to perform together with short reference materials for the Arcum language. We observed these participants over two sessions, which took place in a quiet office environment. All six study participants were graduate students in the computer science department and highly competent programmers. Table 1 shows the backgrounds of the participants.

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 participant's use of language, in order to see the kinds of metaphors the participants 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.

Group	Participant	Eclipse	Languages
A	A1	no	Java, Lisp
	A2	yes	Java, Lisp, ML
B	B1	no	Java, ML
	B2	yes	Java, ML
C	C1	yes	Java, Lisp, Prolog
	C2	no	Java, ML

Table 1. Study participant's programming language and Eclipse experience.

The study comprised two sessions for each pair of participants. The first session was a tutorial that covered Eclipse and the Arcum plug-in, and included step-by-step guides for completing the tasks. The second session was held on the following day and covered program transformation and checking tasks without any step-by-step guides.

3.1 Tutorial Session

The tutorial session used a small (83 line) Java project that has three classes implementing a simple linked-list and utility operations, including 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 participants were free to use Eclipse as they saw fit to perform the change.

Arcum Training Task. The next task gave the participants practice with executing Arcum code and provided the background for writing code in the language itself. A complete code example was provided that contained one `interface` representing the attribute idiom and two `options` representing the alternative implementations. This attribute example 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 participants were asked to insert another check.

Automate a Transformation Task. Finally, with the basics of Arcum covered, the participants 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 and 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 Advanced Session

The session using Arcum without step-by-step instructions used the HTML renderer component of the Lobo project, a complete web browser written in Java.[1] We chose Lobo because it was the top desktop application project available from SourceForge that was written 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. 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 participants were asked to migrate the Lobo codebase from using the always-synchronized `StringBuffer` to the more efficient `StringBuilder`. Although this change could easily have been made with a global text-based find-and-replace (because the two classes have the same API), 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.

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 (how the log is acquired, or which log is used) would require global changes. One simple property of this design idiom that can be checked is if the argument given has, in fact, the correct name (e.g., a copy and paste error would lead to the logs of one class to be written to the log of the copied class). The instructions for this task required the participants write Arcum code that could check for this property.

[1] http://lobobrowser.org/

3.3 Threats to Validity

There are some threats to the validity of our study. We identify here the main threats that could limit the generalizability of our results.

Use of graduate students. It was useful using graduate students because it is expected that they would be more open to new ideas as well as more inventive with this novel tool, whose characteristics are not well understood enough to provide training that would be typical with a well-understood commercial tool. Thus, we can't draw any strong conclusions from the participants' inventiveness. Rather, the insights derived here could be used in tutorial documentation for more typical programmers, letting them apply techniques similar to those discovered here.

Pair programming. It is quite typical for programmers to work in pairs on complex tasks or when learning a new tool. In this respect, the study has good internal and external validity. However, it is also reasonable to expect that individuals might work differently, because of the higher cognitive load, not having an assistant: the work would likely be slower, more prone to errors, and less inventive. However, this study did not investigate these as measures.

Simplicity of task. For the purposes of time, the tasks were kept relatively simple. Thus, the techniques we saw might not be comprehensive; additional techniques could be discovered on larger-scale tasks.

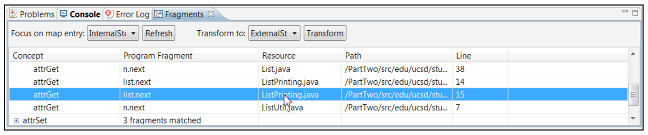
4. Discussion

By looking at the most difficult aspects of performing the change tasks, we can see where a participant's conceptual model breaks down, and from there see how the Arcum language and its supporting tools can be enhanced to better meet a user's expectations.

In this section, we discuss the following findings: (1) Participants reasoned about scattering in the program by writing queries to describe them, and then scanning the query results (Section 4.1); (2) The meta nature of Arcum was the most problematic when it came to separating references from definitions (Section 4.2); (3) Standard programming techniques, such as consulting reference materials, can make participants think of specific instances when the appropriate level of abstraction should be the most general conceptual level (Section 4.3); and (4) Reasoning over two versions of an implementation is far less problematic than reasoning over many possible incorrect implementations (Section 4.4).

4.1 Making Direct Queries with Arcum

During the advanced session, the participants were asked to reason about several instances of crosscutting, such as the scattered use of the `StringBuilder` class, or the scattered instances of the logging idiom. Arcum's Fragments View, which displays a list of program matches along with the file and line number, was used by the participants in many of these instances (see Figure 1).



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

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

In the post-study interviews, one participant 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 [2].

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*:

A2: 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 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 could ensure that the queries were tested at least once by following the practice of injecting known matches into the code, although such tests alone are not comprehensive.

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. While not Arcum-specific, this lead 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 participants a longer time to understand the crosscutting nature of the code: Not only did the participants 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. Drawing live connections in the IDE could be the next logical step to making such connections clearer.

4.2 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 participants are unclear what the Arcum type `Type` means. The reference sheet given to the participants defined a `Type` as: "A Java class, enum, or interface," and it remained unclear to the participants if this meant the unique definition for the type (the correct answer), or the many references to the type. When participants initially pattern matched for 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 participants 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 fragments, 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 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.

4.3 Using Reference Materials

Reference materials were another source of information that participants used to help them reason about crosscutting idioms.

When working on the *Manual Transformation Task* the participants 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 `Map` interface's `put` method, and the signature for the method was displayed as:

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

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

The participants discovered their error after executing the program and seeing how its output changed. The participants returned to the API documentation and scrolled down to reveal the explanation of 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 Reasoning about Several Implementations

One of the mental challenges of reasoning about checking and refactoring lies in the need to conceptualize different implementations of the same program. 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. However, for performing checks, participants had to think about both the correct program, and the infinite various possible *incorrect* versions of the program.

Refactoring is a simpler problem for participants to conceptualize, because they only needed to consider two versions, the before and after. A tool that assisted the before-and-after metaphor was Arcum's transformation preview pane, which showed the two different versions of the program side by side. The participants inspected the differences to get better confidence in their transformations. However, even the before-and-after conceptualization is not without problems: 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.

Our study shows 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. 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.

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?

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 development of Arcum, we have many times debated what the best choice of keywords would be, 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.

5. Related Work

Sillito et al. studied programming in Eclipse focused on the questions programmers ask when modifying programs [10]. 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.

Robillard et al. investigated the process programmers use to understand code before changing it, and found that programmers who invested more time in making more accurate models of the program were the most successful [7]. Arcum is a tool that aims to work at a programmer's conceptual level, so a future study could investigate how useful Arcum declarations are at helping programmers construct accurate models compared to code reading alone.

Ko et al. studied software changes performed in Eclipse and 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 [4]. Our study focused on scattering that is the result of well-structured idioms that lack programming language support, while Ko et al.'s study was concerned with the scattering of program features and their dependencies.

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 [11].

Murphy et al. argue for the structure of crosscutting tasks to have a concrete representation in the IDE to guide further changes [6]. 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.

6. Conclusion

Our user study shows that the Arcum approach to developing checks and refactorings for crosscutting idioms was natural to the participants and that they could leverage their existing knowledge of modularity. Additionally, we observed that the participants could apply general programming techniques to meta-programming.

By observing the metaphors that the participants used while addressing these challenges, we obtained a better understanding of the Arcum development processes. In doing so, we identified a few preliminary design recommendations to improve software maintenance tools. For example, keywords in programming languages should be made to match as closely as possible the metaphors that programmers will use in the development process. Choosing keywords in this way decreases the gap between the developer's mental model of programming idioms and how he or she expresses those idioms in the programming language. Further, environments for software maintenance tasks should include tools for pattern testing, visualization, and generation. These tools would help programmers by providing them with immediate feedback about their crosscutting queries.

Acknowledgments

We'd like to thank the anonymous participants for the time they put into the study and for their feedback. This work was supported in part by an Eclipse Innovation Award from IBM and NSF Science of Design grant CCF-0613845.

References

[1] A. F. Blackwell. Metaphors we program by: Space, action and society in java. In *18th Workshop of the Psychology of Programming Interest Group*, pages 7–21, September 2006.

[2] W. G. Griswold. Coping with crosscutting software changes using information transparency. In *REFLECTION '01*, pages 250–265. Springer-Verlag, 2001.

[3] W. G. Griswold, J. J. Yuan, and Y. Kato. Exploiting the map metaphor in a tool for software evolution. In *ICSE '01*, pages 265–274. IEEE Computer Society, 2001.

[4] 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*, pages 126–135, 2005.

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

[6] 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.

[7] M. P. Robillard and W. Coelho. How effective developers investigate source code: An exploratory study. *IEEE Trans. Softw. Eng.*, 30(12):889–903, 2004. ISSN 0098-5589.

[8] M. Shonle, W. G. Griswold, and S. Lerner. Beyond refactoring: a framework for modular maintenance of crosscutting design idioms. In *ESEC-FSE '07*, pages 175–184, 2007.

[9] M. Shonle, W. G. Griswold, and S. Lerner. Addressing common crosscutting problems with arcum. In *PASTE '08*, November 2008.

[10] J. Sillito, G. C. Murphy, and K. D. Volder. Questions programmers ask during software evolution tasks. In *SIGSOFT '06/FSE-14*, pages 23–34, 2006. ISBN 1-59593-468-5.

[11] 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.

[12] A. Strauss and J. Corbin. *Basics of qualitative research: Techniques and procedures for developing grounded theory*. Thousand Oaks, CA, SAGE Publications, Inc., 1998.