When Refactoring Acts like Modularity

Keeping Options Open with Persistent Condition Checking

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

Oftentimes the changes required to improve the design of code are crosscutting in nature and thus easier to perform with the assistance of automated refactoring tools. However, the developers of such refactoring tools cannot anticipate every practical transformation, particularly those that are specific to the program's domain. We demonstrate Arcum, a declarative language for describing and performing both general and domain-specific transformations.

Because Arcum works directly with declarative descriptions of crosscutting code it can ensure that code written or modified after the transformation also satisfies the design's requirements. As a result, preconditions and postconditions are persistently checked, making the crosscutting code (such as the use of a design idiom or programming style) behave more like a module with respect to checkability and substitutability. Bringing such capabilities into the IDE allows for code to be decomposed closer to the programmer's intentions and less coupled to specific implementations.

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

General Terms Languages, Design.

Keywords Refactoring, Design patterns, Arcum.

1. Introduction

When employed, agile programming changes the way programs are developed (Beck and Andres 2004). In agile programming, programmers use advanced IDEs and write code according to their current knowledge; keeping open the chance to make modifications later, when requirements change or when more information is known. Modifications that operate on crosscutting code can be less tedious and error prone with the assistance of automated tools. However, the helpfulness of such automated tools is limited by the kinds of changes the tool can perform.

Arcum is a framework for scripting user-defined program checks and transformations, with the goal of increasing automated refactoring opportunities for the user (Shonle et al. 2007). In this paper, we show how the presence of a tool like Arcum can change how design decisions are handled. Following the spirit of agile programming, the programmer can use Arcum to record design decisions and later revisit them, even when the decision's implementation is crosscutting, such as when it implements a design pattern or programming style.

Using Arcum, a programmer can envision a crosscutting design idiom's implementation as a form of a module. Arcum uses a declarative language to describe the idiom's implementation, where descriptions are composed with Arcum `interface` and Arcum `option` constructs. An `option` describes one possible implementation of a crosscutting design idiom, and a set of `options` are related to each other when they all implement the same Arcum `interface`.

Arcum's declarative language uses a Java-like syntax for first-order logic predicate statements, including a special pattern notation for expressing Java code. Arcum infers from the predicates the conditions necessary for a proper implementation of the idiom, allowing for programming errors to be caught. Additionally, when the implementation of an alternative implementation is described, Arcum can infer the preconditions, transformation steps, and postconditions necessary to refactor between the two implementations.

This paper introduces the notion that Arcum can change the development process itself, and presents a new way to postpone the decisions behind some programming trade offs.

2. Development Example

In this section, we present an example of a design decision made in the process of development and how Arcum can

WRT'08, October 19, 2008, Nashville, Tennessee, USA.
ACM 978-1-60558-339-6/08/10.

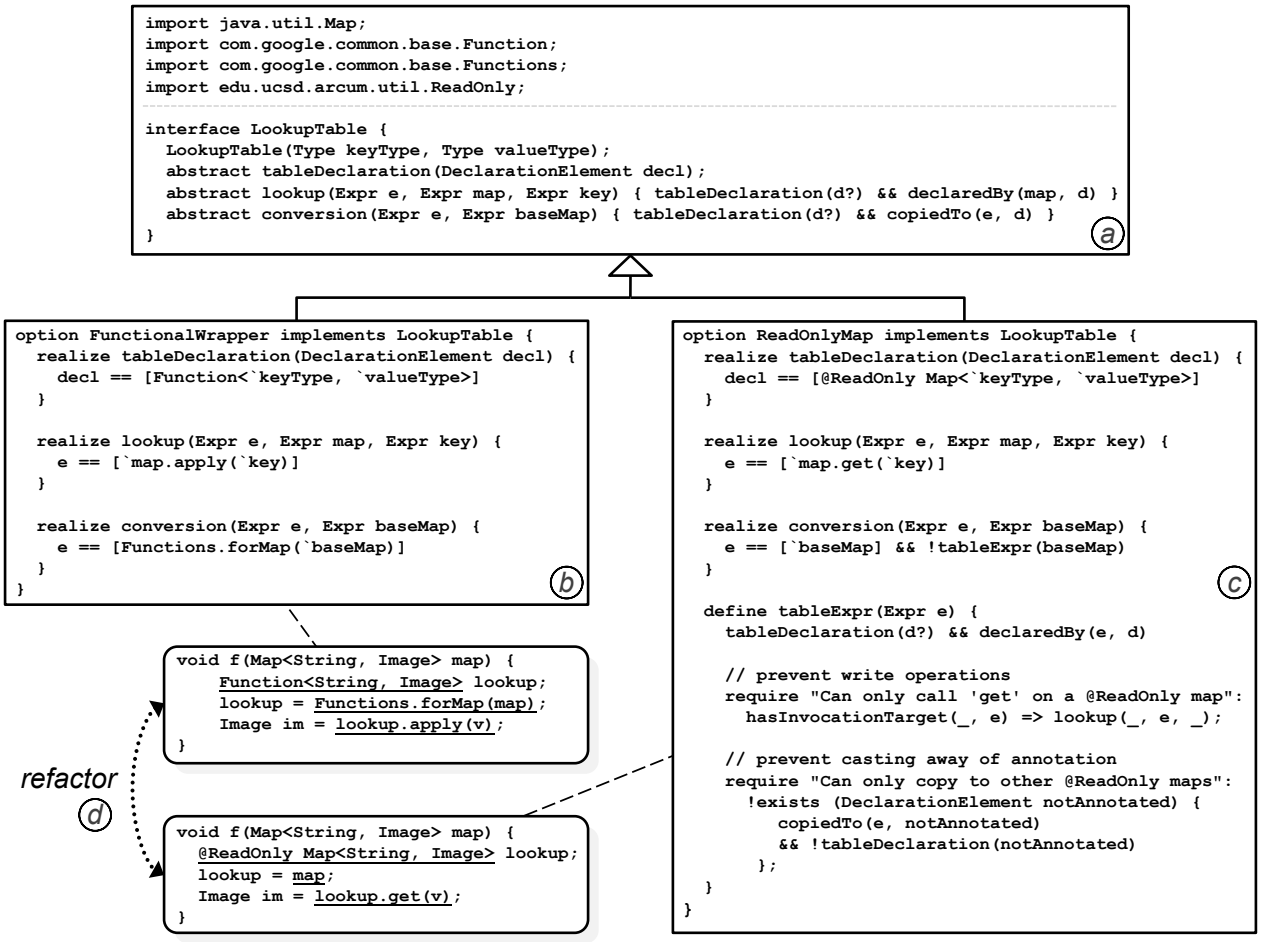

Figure 1. The Arcum interface for a lookup table (box a) is implemented in two ways: As a `Function` object that associates a range to a domain (box b), or as a `Map` annotated as read-only (box c). The sample code (d) shows these instantiations of `LookupTable` (with `keyType` bound to `String`, and `valueType` bound to `Image`). The refactoring between the two different implementations is bi-directional.

both keep track of this decision and allow for the refactoring to alternative implementations.

Consider a system where an association is made between members of a set of objects and a set of their definitions, such as in the implementation of a symbol table. For example, the `HashMap` class is sufficient for expressing such a relationship. Whether the `HashMap` class or `Map` interface is used for parameter types is one design decision that the programmer has to make. In this case, it is recommended that the `Map` interface be used, because it allows for more flexibility—this recommendation appears as items 40 and 52 of *Effective Java* (Bloch 2008).

However, even the `Map` interface itself might be broader than necessary. For example, methods taking instances of the symbol table as parameters might only need to perform lookup operations, and not modify the table. Thus, at the conceptual level, the programmer's requirements are for a lookup table, and the `Map` interface is just one possible implementation of such a mechanism. By coding with a focus on the direct requirements, programmers can code closer to their original intentions and preserve the option of using a different implementation later.

2.1 Example Arcum Interface

Figure 1, box a, shows an Arcum interface definition named `LookupTable`, which explicitly documents the requirements of a lookup table. The definition specifies how a `LookupTable` is parameterized and describes three abstract *concepts* related to the proper implementation of a `LookupTable`:

- The `LookupTable` has two parameters, both types: the key type and the value type.
- The first concept, `tableDeclaration`, defines when a declaration element belongs to a `LookupTable` imple-

mentation.[1] Arcum `options` that implement `LookupTable` must provide a complete definition for `tableDeclaration`.

- The second concept, `lookup`, defines when an expression is considered a lookup operation. A lookup operation must have both a `map` expression and a `key` expression. The concept is partially defined, restricting as valid `map` expressions only those whose type has been specified by a `tableDeclaration`. The rest of `lookup`'s definition must be provided by the implementing `option`.
- The third concept, `conversion`, defines when a regular `Map` is converted to an expression that yields a lookup table. This concept is also partially defined, restricting valid `conversion` operations to only those where the new lookup table is stored into a location specified as a `tableDeclaration`.

Any particular instance of these three concepts can be present any number of times in the program.

Both the `lookup` and `conversion` concepts reference the `tableDeclaration` concept, using it as a predicate. The `lookup` concept uses the built-in `declaredBy` predicate, which relates expressions to the declaration of its static type. The `conversion` concept uses the built-in `copiedTo` predicate, which relates copy expressions to the declaration associated with the memory location copied to. (Copy expressions includes expressions that appear as arguments, assignment values, or return values.)

2.2 Example Arcum Options

The `LookupTable` interface shown in Figure 1 has two Arcum options that implement it: `FunctionalWrapper` (b) and `ReadOnlyMap` (c).

The `FunctionalWrapper` implementation of `LookupTable` utilizes the `Function` interface, which defines a single method named `apply`. A `Function` can be created from a `Map` via the static `forMap` method. The advantage of using this `Function` implementation is that it requires the implementation of only the `lookup` operation (here, `apply`). This implementation includes three uses of the pattern notation, all enclosed in square brackets (`[...]`), one pattern specifies a type, while the two other patterns specify Java expressions.

The `ReadOnlyMap` implementation of `LookupTable` uses the standard `Map` interface, but with a twist: All declarations of lookup tables are labeled with the `@ReadOnly` annotation. This annotation signals to clients that only the `get` operation is required. Such annotations can be more than just documentation with the help of additional checks: The `tableExpr` concept defined in Figure 1, box c, does so by using two `require` statements. A `require` statement takes in an error message string and a predicate. If the predicate's condition is violated by any member of the `tableExpr` relation, Arcum reports the error to the IDE.

These two options, together with their common interface, can be used to refactor code between the two different implementations. When a programmer realizes that he or she needs a `LookupTable`, and not specifically a `Map`, he or she can record this design decision by making an instantiation for each use of it in the IDE. Such instantiations are kept in separate text files, also processed by Arcum, that serve to both document design decisions made and to ensure that the implementation doesn't deviate from the programmer's original intentions. The area marked "d" in Figure 1 shows two example code snippets that can be automatically transformed to and from each other with Arcum. The user performs a refactoring by opening up the instantiation file in the IDE, locating the current `option`'s instantiation and supplying the name of the `option` to replace it. The details of the algorithm are discussed in our earlier paper (Shonle et al. 2007).

Although the example presented in Figure 1 is complete, it can be extended to include further checking. For example, checks can be made to ensure that any type used for the `keyType` overrides the `equals` method whenever it also overrides the `hashCode` method. Such checks can prevent coding errors and help ensure bi-directionality.

3. Applications of Arcum

In the previous section, we saw an example showing how a programmer can code according to their original intentions and not have to commit to certain implementation decisions. The specification of such intentions can introduce new restrictions on the code, thus extending the type system in domain and program-specific ways.

Often, programmers must manage multiple design trade offs: *Should this be implemented with the fast method; or the one that takes less memory?, Should this be implemented in an expressive, but terse, manner; or in a way more familiar?* For many cases, such design decisions can be modularized with the programming language: What will most likely change is encapsulated in a module, and what will most likely remain stable is exported as the module's interface (Parnas 1972). But sometimes the crosscutting of design idioms are unavoidable—by their nature, they are based on the interaction of multiple participants. Additionally, there are also opportunities within the module where other implementation decisions and trade offs must be made (VanHilst and Notkin 1996): Further modularization can increase flexibility, but the indirection can also create code that is harder to reason about and more tedious to write.[2]

With Arcum, there's an opportunity for many kinds of trade offs that, although not modularized, can be explicitly documented in the form of `options` and their instantiations. In the same supplemental Arcum files, possible alternative implementations can also be expressed. When design re-

[1] A *declaration element* specifies the type in the following kinds of declarations: fields, local variables, parameters, casts, and method return types.

[2] Imagine a program that used the Factory pattern for all object instantiations; and only utilized `interfaces`, which cannot contain any method definitions, thus making control flow particularly hard to reason about.

quirements change, the documentation serves as a guide describing other options to explore. The Arcum code written to describe decisions encountered routinely can be reused in other projects.

One powerful example is the trade off between using Java's reflection libraries instead of coding connections directly. The reflection style lends itself well to rapid prototyping, while the direct style is safer due to its extra type checking. For example, the traditional Visitor Pattern (Gamma et al. 1995) can be implemented using reflection with the DJ library (Orleans and Lieberherr 2001). DJ performs the depth-first traversal over the object structure via reflection, using a sparse description of the traversal strategy as a guide. Thus, to get the functionality of the Visitor Pattern, a programmer simply needs to include the DJ library and specify a brief description of the strategy; he or she can then invoke the traversal wherever it is needed.

In addition to being quicker to write, the DJ solution is also more expressive: Because the class structure does not need to be repeated to implement any `accept` methods, the program is less redundant. However, the traditional implementation of the Visitor Pattern executes faster. We found that this trade off can be managed by Arcum with some encouraging preliminary results: We were able to code an `interface` and two `options` to describe these two different implementations of the Visitor Pattern and refactor between them. One benefit of using Arcum for this task is that the traditional Visitor Pattern implementation benefits from extra checking. For example, Arcum can detect when a branch of the traversal is missing, and generate an error. Due to the non-local reachability effects of the object graph, such checks can locate subtle bugs.[3]

Granting the IDEs such power can change the nature of trade offs, because techniques previously viewed as costly can become cheap. For example, a programmer can code in a rapid prototyping style when changes need to be performed, but then refactor back to more traditional implementation styles whenever the program's performance is critical. In this way the view of the program presented to the programmer is the one best suited to his or her needs.

4. Related Work

This section briefly covers some related works not discussed in the body of this paper. A longer discussion of related works can be found in our earlier paper (Shonle et al. 2007). The iXj program transformation system allows for pattern matching similar to Arcum's concept construct (Boshernitsan and Graham 2004). The iXj system could assist the writing of Arcum concept implementations through its interactive features.

[3] For example, in a compiler, the AST structures might support an expanded syntax that forces the type checking operation to traverse over branches it wouldn't have needed to before.

The Feature-Oriented Refactoring (FOR) work recognizes the crosscutting nature of the implementation of software features (Liu et al. 2006). With FOR, modules with optional features (such as bounds checking) can be refactored into a base module with separate delegator sub-modules that can be composed together. Using FOR, the best-suited variant of a module's implementation can be used according to the circumstances.

The example discussed in Section 2 is related to the class library migration problem, which is encountered when code needs to be ported to use a different library (Balaban et al. 2005). Finally, Arcum is related to the role-based refactoring project, which permits programmers to build macro-refactorings from micro-refactorings (Hannemann et al. 2005).

Acknowledgments

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

References

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

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

Joshua Bloch. *Effective Java*. Addison-Wesley, 2nd edition, 2008.

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

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

Jan Hannemann, Gail C. Murphy, and Gregor Kiczales. Role-based refactoring of crosscutting concerns. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 135–146, New York, NY, USA, 2005.

Jia Liu, Don Batory, and Christian Lengauer. Feature oriented refactoring of legacy applications. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 112–121, New York, NY, USA, 2006.

Doug Orleans and Karl J. Lieberherr. Dj: Dynamic adaptive programming in java. In *REFLECTION '01: Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, pages 73–80, London, UK, 2001. Springer-Verlag.

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

Macneil Shonle, William G. Griswold, and Sorin Lerner. Beyond refactoring: a framework for modular maintenance of crosscut-

ting 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.

Michael VanHilst and David Notkin. Decoupling change from design. In *SIGSOFT '96: Proceedings of the 4th ACM SIGSOFT symposium on Foundations of software engineering*, pages 58–69, New York, NY, USA, 1996. ISBN 0-89791-797-9.