A Framework for the Checking and Refactoring of Concepts *

Macneil Shonle William G. Griswold Sorin Lerner

Computer Science & Engineering
UC San Diego
La Jolla, CA 92093-0114
{mshonle, wgg, lerner}@cs.ucsd.edu

Abstract

Programmers resort to design patterns, micro-architectures, and other idioms when their design ideas can't be expressed directly in the programming language. The crosscutting code that appears as a result makes it harder to ensure a correct implementation of the idiom, and complicates software evolution when the idiom's implementation cannot be modularly substituted or extended like a method or class.

In this paper we introduce *Concepts*, an IDE-based mechanism for declaring, checking, and evolving crosscutting design idioms (such as design patterns). Programmers code their design idioms as before, but also declare their fundamental properties in supplemental files. A concept's behavior and implementation are described separately. This separation permits describing a new implementation for a concept and then having the concept tool mechanically transform the concept's current implementation into the new one. As a consequence we get many of the same benefits for concepts that we get for classes: checking of key behaviors and substitutability.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Constraint and logic languages; D.2.11 [*Software Engineering*]: Software Architectures—Domain-specific architectures, languages, patterns

General Terms Languages, Design.

Keywords Aspect-oriented programming, refactoring, design patterns.

* This work was supported in part by an Eclipse Innovation Award from IBM.

1. Introduction

One of the benefits of modular programming is that the implementation of a module can be substituted without requiring changes to, or extensive retesting of, other parts of the system. For a variety of reasons, however, many change tasks are crosscutting rather than modular. For example, not every future change can be anticipated, meaning that the existing abstractions may not modularize the given change. Sometimes, the language's abstraction mechanisms are not powerful enough to permit an efficient modularization. Other times, an agile development process like XP may intentionally delay the introduction of such abstractions [2].

In this paper we introduce Arcum, a framework and tool for checking and refactoring the implementations of crosscutting concepts in Java programs. Arcum expands the opportunities for modular substitution and reasoning through two key constructs, Concepts and Options, which are declared as auxiliary supplements to a Java program. A Concept declaration states the essential abstract properties of a concept. An Option declaration states one way that a Concept may be implemented. The relationship between a Concept and its Options is similar to the relationship between a Java `interface` and the classes that implement that interface.

A programmer may be motivated to declare a Concept and one or more Options when the need arises for either refactoring of a crosscutting concept or better checking of the concept's implementation. Once declared, refactoring is merely a matter of specifying the replacement of the prevailing Option with an alternative Option. The correctness of such a replacement is managed by checks specified in the Concept and the Options. A Concept specifies behavioral constraints on its Options, and each Option specifies additional constraints specific to its implementation. Concepts and their Options can be written in a generic fashion and then instantiated for a specific case, enabling reuse and the development of Concept libraries.

There are several unique benefits of retaining Concepts and Options as persistent, supplemental descriptions to a Java program. For one, being persistent, unlike the typical refactoring operations invoked by a programmer via an IDE, a declared Concept and its currently deployed Option are

continuously checked, not just during refactoring. In this respect, the benefits of static checking of regular classes are being extended to crosscutting concepts. Continuous checking also ensures that the substitutability of the prevailing Option for an alternative Option is preserved. Two, due to their declarative nature, Concepts and Options provide a precise mechanism for documenting crosscutting concepts and expressing the programmer's intentions for crosscutting concepts. Finally, because Concept and Option declarations are supplements, the core source code remains unchanged, and in pure Java. The program is only changed when one concept is refactored to use an alternative implementation. Such refactoring is always done within the IDE at the programmer's discretion, by specifying a change in the prevailing Option. The separation of Arcum code and Java code reduces the cost and risk of initiating the use of Arcum, and enables late-stage adoption.

In the next section we introduce the Arcum approach through a comparative example. We then review the Arcum language in greater detail and describe our algorithm for refactoring between two Options. Next, we provide a preliminary evaluation of our approach by describing the design of the Arcum refactoring engine, reporting on our development of a complete Concept, and demonstrating their use on a real system. We close with a discussion of future work, related work, and a few concluding remarks.

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

Figure 1. *Example implementation of the* altText *attribute as a field.*

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

Figure 2. *Alternative implementation that uses a static table to sparsely store non-null '*alt*'-tag values.*

2. Example: Object Attribute Storage

In this section we illustrate our refactoring framework with a simple Java program that processes HTML image elements. Image elements in HTML have an optional 'alt'-tag attribute that specifies alternate text to display in place of the image. There are a variety of ways of implement this concept of "alternate text" in Java. For example, one can simply add a field named altText to the Image class that represents image elements, as shown in Figure 1. Alternatively, if one expects the alternate text to be absent often, meaning that it takes on a predefined default value, then storing the alternate text in an external table can save memory at the expense of runtime. Such an implementation is shown in Figure 2.

Although this intentionally simplistic problem might be easy to anticipate, in general, it is difficult, if not impossible, to design software abstractions that are flexible enough to support all future changes. Furthermore, some programming methodologies, such as Agile development, in fact favor rapid development of prototypes, with refactorings being applied as needed later in the development process. In either case, the end result is that refactoring transformations are a common occurrence in the development of large software systems.

To give an overview of our refactoring approach, we describe how a developer would refactor a large body of code from the internal field implementation of the "alternate text" Concept to the static map implementation, first using a regular IDE such as Eclipse (§2.1), and then using the proposed Arcum framework (§2.2).

2.1 Refactoring using Eclipse

Although the code shown in Figure 1 has only two reads (lines 9 and 12) and one write (line 6), in a realistic code base one would expect to encounter many references to the altText field that need to be modified.

A developer could use Eclipse's built-in refactorings: the "Encapsulate Field" refactoring could replace all references to the altText field in the original code with calls to getter and setter methods; these getter and setter methods would then be modified to call the appropriate map operations, after which the "Inline Method" refactoring would inline away calls to the getter and setter methods. Although these built-in refactorings make manual modification less onerous, the problem remains the same: refactorings generally require many changes to be made to the code, and the tool performing the transformations is simply not aware of the structure that is present in the code being manipulating. This lack of structure awareness results in a variety of drawbacks, including: (1) code refactoring is error-prone and tedious; (2) it is often difficult to switch back and forth from the original implementation to the refactored implementation; (3) subtle bugs can be introduced, for example transforming f(this.altText = y) into f(Image.altText.put(this, y)) is an incorrect transformation because the put method returns the

previous value in the map; and (4) little of the work done for refactoring the altText field can be reused for refactoring other fields.

2.2 Refactoring using Arcum

The Arcum approach addresses the above limitations by enabling the programmer to formally capture implicit structure in his or her code. Rather than directly applying refactoring transformations, the programmer first declares behavior (Concept) and implementation descriptions (Options) for the code that will be refactored. By specifying that the Concept's prevailing implementation Option is being replaced by another Option, the code is automatically refactored. The Concept and the chosen Option are retained and continue to impose checks to ensure that the Concept is employed as intended.

Figure 5 provides an overview of how the Image constructor from our example comes to be refactored using Arcum (see Appendix A for the listing of the actual Arcum code for this example). At the bottom left of the figure is the original Java code for the constructor, and at the bottom right is the desired target Java code. At the top of the figure is the Arcum code specifying the abstract Concept to be refactored. Below the Concept code is additional Arcum code that specifies two implementation Options for this Concept: one using an internal field (the InternalField Option), and another using a static map (the StaticMap Option).

Initially the programmer declares AttributeConcept and an Option corresponding to the current realization of the Concept, the InternalField Option. The programmer then declares that this Concept and Option should hold in the Java program by instantiating the Option with the mapping shown in Figure 3. Note that rather than designing a refactoring that applies only to the "attribute for alternate text" Concept, the programmer has designed a parameterized Concept so that it can be applied to any attribute of any class. In particular, the AttributeConcept from Figure 5 takes three parameters: targetType is the class for which the attribute is defined, attrType is the type of the attribute, and attrName is its name. Oftentimes a programmer might find the desired Concept and Options in a library, rather than having to implement them from scratch.

The relevant part of the Concept code for this example is the attrSet trait (Figure 5, line 32), which is meant to represent the locations in the Java program where the refactored attribute is being set. The parameters to the trait, in this case setExpr, targetExpr and valExpr, are fragments of Java code that are extracted from the locations in the code where the attribute is set. For example, targetExpr is the object whose attribute is being set, valExpr is the value to which the attribute is being set, and setExpr is the entire expression that performs the set operation. Arcum variables, such as setExpr store *references* (i.e., pointers) to code fragments, and so in this example the setExpr variable identifies the location and scope of the set operation, which is later used to determine

```
01  map {
02    InternalField(
03      targetType=Image, attrType=String, attrName="altText");
04  }
```

Figure 3. *Concept map for the original implementation of the* altText *attribute. It declares that the* Image *class's* altText *field satisfies the requirements of the* InternalField *option.*

```
01  map {
02    StaticMap(
03      targetType=Image, attrType=String, attrName="altText");
04  }
```

Figure 4. *Concept map for the sparse implementation of the* altText *attribute. By changing* InternalField *to* StaticMap*, the Java code is automatically refactored to make the revised Option mapping hold.*

what portion of the code gets transformed during refactoring.

Each Option specifies a different "realization" of the attrSet trait. A trait is realized by providing a pattern that identifies the fragments of Java code that are instances of the trait. For example, the pattern on line 39 in the InternalField Option shows how a regular assignment to a field becomes an instance of the attrSet trait. In our Image constructor example, line 42 in the Java code matches this pattern, and therefore becomes an instance of the trait. Similarly, the pattern on line 46 in the StaticMap Option states which map operations become instances of the attrSet trait.

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

The directionality of the mapping is determined by how the top-level Concept mapping is instantiated and later edited. As declared in Figure 3, the InternalField implementation is the prevailing Option at the beginning of our scenario. To refactor to the sparse implementation of the altText field, the programmer changes the named Option in the mapping to StaticMap as shown in line 2 of Figure 4, which triggers an automatic refactoring of the code

In our scenario, the altText field is initially implemented as a simple class field and the conceptual flow of information in Figure 5 goes in the clockwise direction, following the solid arrows. However, this refactoring can be run in either direction. The field assignment on line 42 in the original code is pattern matched into a trait instance, at which point the references setExpr, targetExpr and valExpr are bound. The newly constructed trait instance is lifted to the Concept level, and then pushed back down to the alternate StaticMap Option, at which point the pattern on line 49, along with

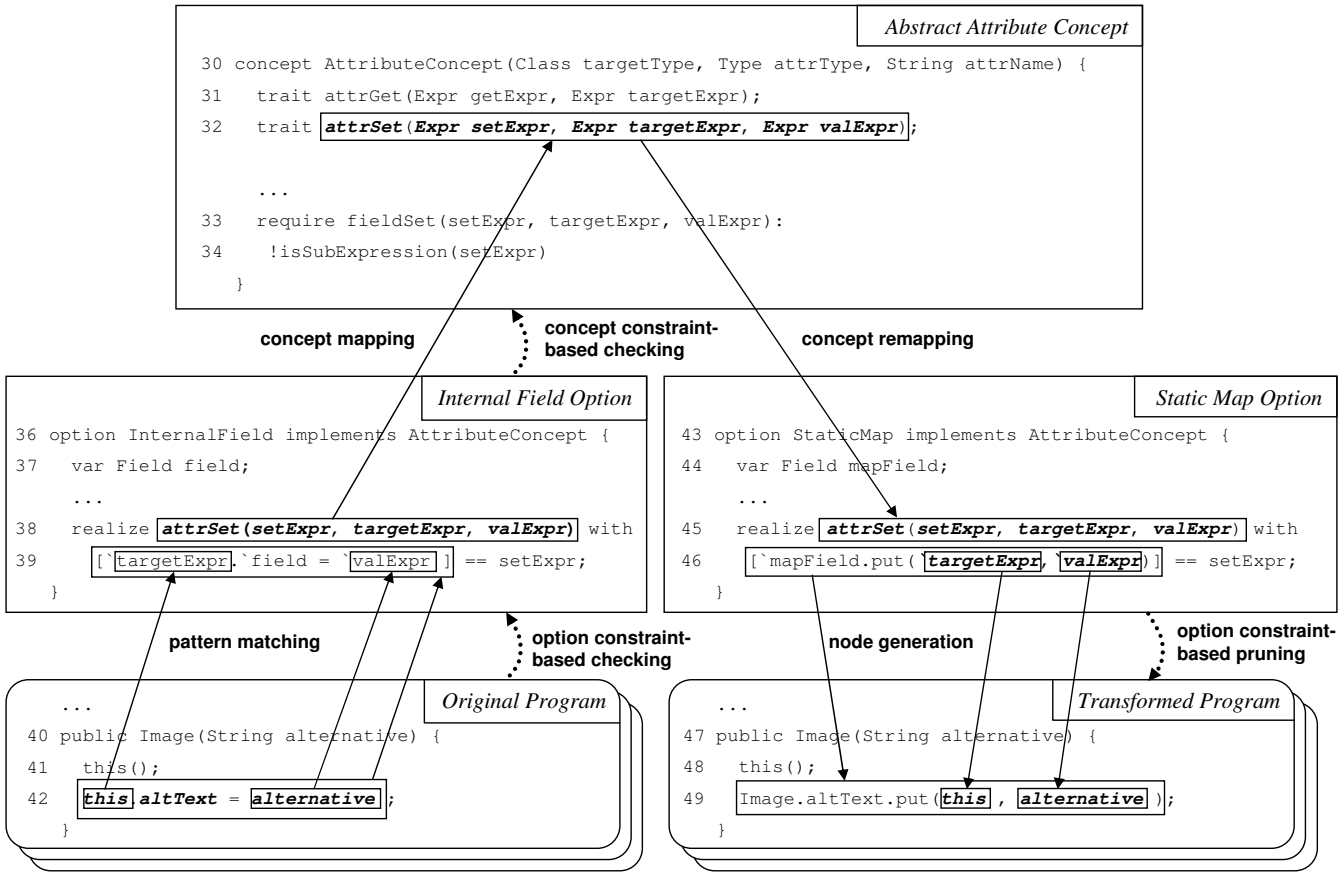


Figure 5. *Overview of how Arcum transforms code into an alternative implementation. By editing the concept mapping, which describes the current concept implementation, the code is transformed in accordance with the new concept mapping.*

setExpr, targetExpr and valExpr, are used to construct the replacement code.

Due to the bi-directional nature of trait patterns, the concept mapping can be changed later to perform the refactoring in the other direction. Because our approach explicitly and persistently identifies refactorable Concepts and their prevailing Options, their consistency properties can be continuously checked. This makes future possible refactorings easier to perform, since the checking enforces Concept compliance. For instance, in our example, we would like to prevent the incorrect application of the refactoring to transform f(this.altText = y) into f(Image.altText.put(this, y)), because, as mentioned previously, the put method returns the *previous* value in the map. This requirement is checked by the Concept with the requires clause on lines 33–34 of figure 5, where isSubExpression(e) checks whether e is embedded in another expression. This requires clause can be checked continuously, to make sure that developers don't accidentally change code in a way that prevents the Concept from being refactored.

3. The Arcum Language

Having seen an overview of Arcum through the example from Figure 5, we now describe the Arcum language in more detail. The Arcum language has three top-level constructs, each of which is described in turn: Concepts (§3.1), Options (§3.2), and Concept Maps (§3.3).

3.1 Concept Declarations

A Concept declares at the behavioral level what is common to all of its Options. This is achieved through four main syntactic constructs: (1) *parameters;* (2) *traits;* (3) *abstract variables;* and (4) *constraints*. These constructs name and manipulate Java program fragments, and the types of the program fragments that are currently supported in our implementation are listed in Table 1. We now describe each of the four syntactic constructs in detail.

3.1.1 Concept Parameters

The AttributeConcept in Figure 5 is parameterized by a Class named targetType (the class that has the attribute), a Type

Fragment Type	Valid as a Parameter?
Type Name	√
Type Method	√
Type Field	√
Annotation	
Declaration Element	
Expression	
Statement	
String	√
Boolean	√

Table 1. *Program fragment types that a Concept can manipulate. Those that can appear as a parameter to a concept are checked.*

named attributeType (the type of the attribute), and a String named attrName (the name of the attribute).[1]

The restriction that targetType must be a Java class is due to the internal field storage Option: only classes can hold fields, and so interface types are not allowed. This restriction is not an issue in the StaticMap Option, because instances of interfaces are valid objects to store in a HashMap. However, the Concept must be an abstract description of *all* Options, and so the AttributeConcept can only operate on classes.

Concept parameters (and other constructs for naming Java program fragments) should be as abstract as possible, sometimes anticipating likely variants from other possible Options. This is the reason why the attrName parameter is a String instead of a Field (which could be passed by explicitly qualifying it: Image.altText).

3.1.2 Traits

The AttributeConcept in Figure 5 specifies two *traits*: attrGet and attrSet. A trait describes a particular role or interaction that occurs in the context of a Concept (these are referred to as sub-concepts in [9]). The attrGet and attrSet traits represent, respectively, the abstract operations of getting and setting the attribute, either of which could occur in the program any number of times (including not at all). Traits have parameters, which are program fragments whose types can be drawn from the ones listed in Table 1. A trait can be thought of as a set of matching program fragments. For example, the attrSet trait represents the set of all program fragments in the Java code that set the attribute. The matching program fragments are trait *instances*. At the Concept level, traits only *declare* these sets. As shown in §3.2.1, the Options can then realize the traits by specifying, using pattern matching, exactly what program fragments in fact belong to the set.

When a trait is used to represent an operation, as is the case for attrGet and attrSet, it resembles a pointcut in AspectJ [7], in that the trait is essentially the set of all matching expressions. For example, in the InternalField Option, the attrSet trait is similar to AspectJ's set pointcut designator. In the StaticMap option, the attrSet trait is similar to AspectJ's call pointcut designator.

Traits can represent more than just operations: they can also express structural properties of code and other crosscutting forms. For example, one could declare in Arcum a trait that would represent "all declarations of type Vector". Such a trait would be useful for refactorings that port from one library to another, for example porting from the older Java Vector collection class to the newer ArrayList class [1].

3.1.3 Abstract Variables

Sometimes a program fragment identified by a Concept is unique in the program, in that there is exactly one occurrence of the program fragment for each instance of the Concept. Concept parameters are one example of such unique fragments. For unique program fragments that are *not* parameters to the Concept, Arcum developers can use *abstract variables*. Abstract variables are declared at the Concept level and can be used for performing Concept-level checks. Although it is also possible to bind variables in a Concept, abstract variables are bound through pattern matching in the individual Options.

To illustrate the use of abstract variables, consider the parameters to the AttributeConcept in Figure 3. It turns out that the parameters have some redundancies in them, which we can remove using abstract variables. In particular, the attrType parameter could be inferred from the remaining two parameters, targetType and attrName: a target class (the targetType) can only have one field named attrName, and so the type attrType could be initialized to the type of that field. Thus, to eliminate the redundancy, the attrType parameter could be turned into an abstract variable. All the checks that use attrType would be performed as before (the Arcum code in Figure 5 does not show any checks that use attrType, but our implementation has such checks) and the specification in the concept map (as in Figure 3) would have one less argument.

3.1.4 Concept-Level Constraints

After program fragments have been named (by the Concept) and bound (by either a map or an Option), the Concept can apply additional consistency constraints to the fragments. For example, the constraint on line 34 of Figure 5 says that all expressions belonging to the attrSet trait must not be embedded inside of other expressions. In general, the boolean condition provided in a require clause is evaluated for each trait instance, and an error is issued if any of the conditions evaluate to false. Conditions are expressed using a simple propositional logic, augmented with built-in primitives such as the ones shown in Table 2. Optionally, a require clause can have an error message associated with it, to provide a detailed message to the user in the event that the condition is not satisfied.

[1] The difference between Class and Type is that Type may include Java interfaces and enums as well as classes.

Predicate Name	Description
$a == b$	Are the two fragments equal?
isAssignableFrom(t_1, t_2)	Is t_2 equal to or a subtype of t_1?
hasMethod(c, m)	Does class c have a method m?
isTransient(f)	Is field f declared as transient?
isJavaIdentifier(s)	Is the string s a valid Java identifier?
isReferenceType(t)	Is t a reference type?
isSubExpression(e)	Does e exist as a sub-expression?
isStringLiteral(e)	Is e a string literal expression?

Function Name	Description
TypeOf(a)	Returns the static type of a (e.g. a could be an expression)
AsLiteral(a)	Returns a literal representing the value a
PositionOf(a)	Returns the filename and line number of element a, for providing error messages

Table 2. *Example built-in predicates and functions.*

3.2 Option Declarations

An Option describes one of the ways in which a Concept can be realized in a program. Options use the implements keyword to specify which concept they are an implementation of. Unlike classes, an Option can only implement one Concept.

Options use pattern matching (described in §3.2.1) as a way to locate the various program fragments needed by the Concept. Program fragments used in pattern matching can be given names by using local variables (§3.2.2). Additionally, Options can provide their own constraints for Option-specific checking (§3.2.3).

3.2.1 Patterns

Options realize traits using declarative patterns, which identify and construct program fragments. Patterns are expressed as Java-like pseudo-code inside square brackets, with back tick marks to identify Arcum variables inside the pseudo-code. For example, the InteralField option in Figure 5 uses the following pattern to match all set operations to the attribute field:

```
realize attrSet(setExpr, targetExpr, valExpr) with
  [‘targetExpr.‘field = ‘valExpr] == setExpr;
```

This pattern is similar in meaning to the following AspectJ code:

```
pointcut attrSet(Image targetExpr, String valExpr):
  set(String Image.altText) && target(targetExpr)
  && args(valExpr);
```

The key difference between the above AspectJ pointcut and Arcum's pattern is the way in which each is used: the pointcut is only used to match a set of join-points, whereas Arcum's pattern is used to *generate* code in addition to matching.

Arcum supports patterns to match expressions of all of the program fragment types listed in Table 1. For exposition purposes, we have used a simplified version of the Arcum pattern language in our examples. The implemented pattern language is slightly more elaborate, and strictly more expressive than the one presented in the examples. The algorithms that use patterns for matching and for generating new AST nodes are discussed in Section 4.

3.2.2 Option-Local Variables

Certain participants in a Concept are local to one particular Option. For example, the field in the InternalField option only makes sense in the context of that Option. *Option-local variables* can be used to capture such program fragments that are specific to a particular Option.

Option-local variables are bound using pattern matching in a way that is similar to trait realization. For example, the following pattern states how the field variable in the InternalField Option gets bound (this pattern was omitted from Figure 5 for clarity of exposition):

```
realize field with
  [class ‘targetType { ... ‘field ... }]
  && [private ‘attrType ‘attrName] == field
```

3.2.3 Option-Specific Constraints

Options can use require clauses to specify option-specific constraints. As an example, in our refactoring scenario from Figure 5, once the code has been refactored to the StaticMap Option, we want to prevent programmers from performing operations other than calling get and put on the altText map, since these method calls (such as altText.clear()) would not have an analogue in the InternaField Option. The following constraints check for this property:

```
trait access(Expr e): [‘targetType.‘mapField] == e;
trait putCall(Expr e): [‘e.put(..., ...)];
trait getCall(Expr e): [‘e.get(...)];
require access(e): in(e, putCall) || in(e, getCall);
```

Here we are declaring three local traits that are used only for the purposes of checking: access(e) represents all ASTs in the program that are accesses to the altText map, whereas putCall(e) and getCall(e) represent all ASTs in the program which are used as the target of a put or get call, respectively. The require clause then states that each access to the altText map must be in the putCall or getCall trait, guaranteeing that the map is only used to call put and get methods.

Another example of an Option-specific constraint is motived by a third Option that we have implement for the attribute Concept, ExternalLookup. This Option stores all attributes for all classes in a class called Lookup, which supports static get and set methods. The equivalent of line 42 from Figure 5 in this third Option would be: Lookup.set(this, "altText", alternative). Once code has been refactored using Arcum to use Lookup, manually inserted calls to Lookup.put in the Java code should not set altText to a value other than a String. The type system cannot enforce this constraint because Lookup.put takes an Object as the third parameter to accommodate all possible attributes. This check can, however, be expressed using Arcum's require clause.

3.3 Concept Maps

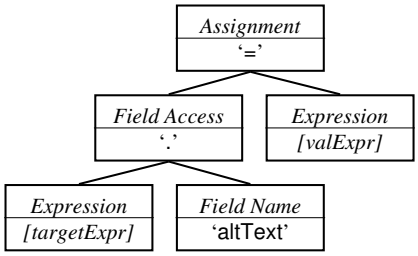
Concept maps in Arcum are used to state which Options are implemented in a program. Figures 3 and 4 have already showed some sample Concept maps. In general, a Concept map is a list of Option instantiations, where each instantiations states the Option's name, and a set bindings for all of the Option's parameters. One benefit of the Concept map format is that there is a separate file that documents some of the architecture of the program.

4. Arcum Refactoring Algorithm

The Arcum refactoring algorithm takes code that implements a source Option and translates it to new code that implements a destination Option. The following process performs such a refactoring:

1. Use the patterns specified in the source Option to bind Option-local variables, Concept-level abstract variables, and trait instances. The pattern matching identifies the program fragments, represented as AST nodes, that participate in the refactoring.
2. Perform any Option-specific and Concept-specific constraint checks, and stop with an error if any of the constraints don't hold.
3. Remove from the program all AST nodes that were pattern matched into the source Option's local variables. For example, when refactoring from the InternalField Option to the AttributeConcept Option in Figure 5, the altText field gets removed from the program because its AST node was pattern matched into an Option-local variable (for brevity, the pattern that performs this match was omitted from Figure 5, but it can be found in §3.2.2). AST nodes that match into Option-local variables are removed during refactoring because by design these variables are Option-specific—otherwise, the local variable could instead be pushed up to the Concept-level as an abstract variable or Concept parameter.
4. Construct new AST nodes using the patterns from the destination Option's local variables and insert them into the program. In the refactoring scenario from Figure 5, the local variable mapField in the StaticMap option will cause a declaration of a static map to be inserted into the program.
5. Replace each trait instance with a new AST node generated from the destination Option's trait patterns; construct the new AST node such that it satisfies the destination Option's constraints (if present).

The main challenge in the above algorithm lies in processing patterns both to perform pattern matching (in step 1), and to generate new AST nodes (in steps 4 and 5). We describes these two uses of patterns in §4.1 and §4.2, respectively.



(a)

```
[‘targetExpr.‘field = ‘valExpr]
```

(b)

Figure 6. ***(a)*** *A partially specified AST sub-tree with variables* targetExpr *and* valExpr*; and* ***(b)*** *the pattern that generated it.*

4.1 Using Patterns for Matching

Arcum patterns are represented using partially specified ASTs. For example, the pattern from Figure 6(b) is represented using the AST shown in Figure 6(a), which is a partially specified AST because some of its sub-trees are Arcum variables, rather than concrete values. A standard unification routine is used to perform the actual pattern matching. The concrete syntax of the program is canonicalized before the matching is performed, so that operations are closer to their semantic meaning. For instance, even though the pattern in Figure 6 uses the "dot" notation, it will also match program fragments that use the implicit this (without the dot). Side conditions can also be added to the pattern in the form of boolean expressions to narrow down the matches.

4.2 Using Patterns for Node Generation

One of the key features of Arcum is that patterns are bi-directional: not only are they used for matching Java code fragments into trait instances, but they are also used in the other direction, to generate Java code fragments from trait instances. As an example, the following pattern in the StaticMap Option is used in our refactoring scenario to insert a call to the put method of the static map:

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

An AST is generated from a trait instance and a pattern by taking the partially specified AST representing the pattern, and inserting into it the values of the Arcum variables from the trait instance. Because Arcum variables store references (pointers) to program fragments, in the above pattern, ‘mapField.put(‘targetExpr, ‘valExpr) creates an AST node representing the call to put, and the equality (==) makes the newly created AST node be placed at the location in the program specified by setExpr.

For some patterns, there are multiple possible AST nodes that could be generated. For example, the pattern shown in §3.2.2 for matching the field variable was in fact a simplified

version of the real pattern. The actual pattern in our implementation also takes into account the possibility of having a transient modifier:

```
realize field with
  [class ‘targetType { ... ‘field ... }]
  && [?transient private ‘attrType ‘attrName] == field
  && (isAssignableFrom(Serializable.class, targetType)
      <=> isTransient(field))
```

The transient modifier begins with a question mark, meaning that the modifier is optional. The conjoined constraint (which is not a pattern because it is not in square brackets) specifies when the field should be transient: the field must be transient exactly when the targetType class implements the Serializable interface.

When there are multiple possible AST nodes that could be generated from a pattern, we use a generate-and-test approach: we generate all the possible AST nodes, and then use the side constraints in the pattern to prune nodes out. This approach works well as long as there are a small number of possible AST nodes to generate. In the above example, because the transient modifier is underspecified, we would generated two ASTs, one with the modifier and one without. The side condition would then be used to determine which of the two AST nodes to use.

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

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

into:

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

5. Evaluation

We performed a preliminary evaluation of the Arcum framework by (1) implementing a prototype Arcum refactoring system for Java; (2) developing a complete AttributeConcept with three Options; and (3) applying Arcum and this Concept in refactoring a sizable Java application to observe whether the use of Concepts provides a feel of modular substitution. Although simplistic, the attribute example exhibits the qualities required for this evaluation. The next section touches on several compelling concept scenarios.

5.1 The Arcum Prototype

The Arcum prototype supports Java 5, except for enums, generics with wildcards, and some aspects of anonymous inner classes. The prototype is implemented largely with the Java-based Polyglot compiler front end [12]. Polyglot is used for both parsing and analysis on the Arcum-supplemented Java programs. A simple Eclipse IDE plug-in has also been implemented, which detects changes to a Concept mapping and refreshes any buffers impacted by the resulting refactoring.

Arcum is realized in Polyglot as three *passes* over the Java program, which is represented as a series of ASTs coupled with a symbol table. The details of what occurs in those passes are described in Section 4. Arcum comprises 9 KLOC of Java code; Polyglot itself is 80 KLOC.

5.2 Development of the Attribute Concept

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

We found that there were two key considerations in the design of a Concept and its Options. The first, of course, is that any resulting refactorings should be meaning preserving. Analogously, if the programmer edits code relating to a Concept in a way that violates the intended use of the Concept, then an error should be reported. The second is how to structure the Concept so that it admits a suitable range of Options without being so general as to unnecessarily complicate the implementation of Options.

In practice, we found that it was hard to get these right the first time. We also found that it wasn't necessary. Our first version of AttributeConcept was correctly parameterized, but its internals omitted all semantic checks. This version was perfectly adequate for the target programs we planned to apply the Concept; that is, their implementations would not have violated the omitted checks. For example, the target programs did not employ serialization of the target type, and as a result the previously discussed serialization checks were not required.

However, there was one design flaw in this initial design: the field name was made a parameter of the setter and getter traits, even though it was the same for all trait instances. This made the interface between AttributeConcept and the two initial Options unnecessarily wide, so we refactored the Concept and its Options to make the field name a local variable of the Concept and Options.

Later, when we came across some serialization code, we realized that a general-purpose meaning-preserving AttributeConcept would have to check for these conditions. If we had been just developing the Concept for our own use, we might have ignored this insight or simply added a check to prevent the application of this Concept to a serializable class. However, under the assumption that AttributeConcept might become part of a reusable library of Concepts, we added the necessary checks to insure that the attribute will be appropriately serialized in all Options.

Over time, several more checks were added, examples of which have been given throughout the paper, such as:

- Type consistency constraints
- Restrictions on the use of certain methods and fields.

- Checks for making sure that fields are appropriately labeled transient in the face of Serializable classes.

Over the history of AttributeConcept's development, its interface did not change, thus its external clients were unaffected by the numerous improvements. However, there were a variety of changes that involved an interaction between the Concept and an Option (such as the unnecessary parameter passed through the getter and setter traits). This is not surprising, as their interaction is analogous to the interplay between a superclass and its subclasses, which often collaborate intimately.

5.3 Application of AttributeConcept

To get a preliminary feeling for whether Arcum Concepts impart some of the benefits of modular substitution to crosscutting concepts, we put AttributeConcept to work. We chose the Polyglot framework itself, since we understand it well. Polyglot is an extensible compiler that uses delegators and extension objects extensively. To support these, the Node class in Polyglot has two protected fields: del and ext. A compiler extension to Polyglot uses these fields to extend the compiler's behavior. Based on their infrequent usage on a per-object basis, a sparse representation of these fields could conserve memory usage.

We first externalized the storage of the del field. We instantiated an InternalField mapping for del, and ran the Arcum engine to check for conformance, which succeeded. We then edited it to instantiate the StaticMap option, thus triggering the refactoring and resulting in 13 substitutions. In our performance measurements this modification actually required more memory than the InternalField option. One cause could be padding issues: removing one field might not generate gains in real space.

Thus, we then declared field ext as an InternalField and then specified the refactoring to the StaticMap's implementation, resulting in 12 substitutions. After this refactoring was performed, the total memory usage was less than the original, unmodified program. However, the gains in saved space were not large, for example saving 100KB for a program run that used 11MB total. Consequently, we re-edited the mapping to reinstantiate InternalField for del and for ext to reverse the refactoring.

In the above tests we found that running Arcum to perform these refactorings ran up to three times slower than running Polyglot as a simple Java language checker. Incremental compilation techniques, like those used by the AspectJ compiler, could substantially reduce this overhead. However, compared to the alternative of reverting the code base to an earlier version, this process is still lightweight by comparison.

A lesson from this experience is that the consequences of some refactorings are difficult to anticipate, and the ability to quickly and safely try and undo refactorings is valuable. The experience of switching implementations by editing the mappings was not unlike tweaking an object factory to return different object types under different conditions. It felt like modular substitution.

6. Discussion and Future Work

The foregoing has only scratched the surface of the Arcum approach. In the following we discuss some novel ways that Concepts might be usefully employed in software development. These uses suggest some additional types of Concepts that might be checked and refactored with the Arcum, as well as suggesting directions for future work.

Rapid Prototyping. The DJ library for Java makes extensive use of reflection to support a dynamic form of the visitor pattern [13]. The traditional visitor pattern only partially modularizes depth-first traversals over objects of different types connected by the "has-a" relationship. In DJ, programmers describe a high-level strategy that specifies the source and destination types of the traversal (and can optionally prune that implied path). As a result, all of the method infrastructure required for the intermediate types can be omitted. Because the high-level strategy specifies only what is important about the traversal, it is less redundant and more flexible when the class graph changes.

However, the use of reflection to descend down object graphs entails a noticeable runtime overhead. A visitor pattern implemented by reflection and a visitor pattern implemented by the method infrastructure realize the same concept, just implemented with different Options. By specifying the visitor as a Concept and these two implementations as Options, it is possible to begin development with the more flexible and convenient reflection-based visitor, and then refactor over to a normal visitor once the application development has settled down. Should development return to a rapid-prototyping phase, the Options can be flipped again. The refactoring to the more efficient Option can also be made a part of the compilation process.

Porting and Retargeting. The Concept approach can also be useful in porting code bases. For instance, class library migration is an important problem addressed by Balaban et al. [1]: one example is refactoring code using the old Java Vector class to use the more efficient ArrayList class. The Vector class is less efficient because all of its methods are synchronized by default, while the ArrayList class is only synchronized when explicitly requested. The concept methodology would be particularly well-suited for this task because it can be continuously checked. For example, if new code is written that allows an ArrayList instance to escape from the thread that created it, it should be explicitly synchronized.

As suggested by De Sutter et al. [17] the unsynchronized StringBuilder class is more efficient than the synchronized StringBuffer class. In the future, a concept could be implemented with a single option that checks for StringBuffers only being used when required.

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

Performance Experimentation. Some Option implementations lend themselves better for gathering performance statistics, but would be impractical in a release version of the software. In such cases, Arcum can be used to switch back and forth between two implementations. In particular, the program can be refactored to the implementation that best supports the collection of statistics and, within that implementation, revisions can be made to address any runtime issues discovered. Performing these revisions might be an iterative process where new issues are only discovered after addressing previous issues. When changes are complete, this modified version can be refactored to use the previous Option's shippable implementation again. Without two-way refactorings, changes made in the modified version would have to be repeated in the version with the original implementation.

Style Checking. Large software projects often have requirements that crosscut the program. For example, an in-house internationalization strategy could have rigid requirements for the ways string literals are used (such as always wrapping them around method calls). Concepts can be written to check and enforce these rules. In the process, the potential exists later on for providing an alternative implementation.

Bug finding tools check for common coding errors and have the potential for finding bugs that code reviews and test cases miss [15]. A project that uses a specialized library could benefit by having an accompanying Concept perform similar checks. For example, performance bugs or other common errors can be detected, providing junior-level programmers with extra assistance and knowledge.

Design Pattern Checking. Design patterns are another class of implementation techniques that could benefit from checking. For example, the initialization of a Singleton instance can be incorrect in the context of a multi-threaded system (for example, the instance might become initialized twice). Concepts can be written for these common error cases. Looking further ahead, it may be possible for such errors to drive an automated refactoring that fixes the implementation.

We are considering support for *concept mining*: given a program and a set of Concepts and Options, find the largest possible concept cover (concept mapping) for the program. In the context of concept mining, Options can be provided for incorrect or partial implementations, perhaps providing a pathway to refactoring to a correct Option.

Aspects and Fluid AOP. The prototype implementation of Arcum supports a subset of the Java 5 language. Supporting the full AspectJ 5 language would allow for more sophisticated refactorings of concepts, such as suggested by Relationship Aspects [14]. Due to Arcum's bi-directional semantics, fluid aspect-oriented programming [6] is possible: the code for a concept could be edited and modified in a crosscutting implementation, but refactored as needed into a localized aspect to enable changes to the concern to be expressed locally; after the change is made, the code could be refactored back into the scattered implementation.

Hybridized Concepts. Currently, an Arcum Option implements just a single Concept, akin to single inheritance. However, just as design patterns can be hybridized, a single Option could be used to realize two Concepts. For example, with support for Options to implement multiple Concepts, Concepts for the mediator pattern and the observer pattern could be realized by a single Option for the mediator-observer pattern, which could enforce that the mediator is also the observer.

Some Concepts might coincidentally hybridize, causing unexpected interactions that would be caught during checking. For example, a visitor Concept and an observer Concept might coincidentally share participant code fragments. If the visitor Concept were to be refactored, it could violate a constraint of the observer Concept, triggering a constraint violation.

7. Related Work

Arcum is a departure from the role-based refactoring work of Hannemann et al., which permits programmers to build macro-refactorings from micro-refactorings [5]. The basic idea is to support the refactoring of crosscutting entities like design patterns by separately recognizing the code for each role in a design pattern (with programmer interaction), and then applying micro-refactorings to each of those roles to achieve the macro-refactoring. Abstract roles are essentially the same as Arcum's abstract traits. The approach is a traditional refactoring approach in that it does not externally specify the underlying concept or implementation options, thus not providing continuous checking or bi-directional refactorings. Marin et al. take a similar approach, although they assemble macro-refactorings from micro-concerns rather than roles [11].

AOP languages like AspectJ can manifest many concepts, including many design patterns, as modular abstractions [4]. 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 concepts without having to modify the code in any way. Of course, when Arcum is extended to support AspectJ, Arcum can assist programmers with refactoring crosscutting code into aspects (and back again).

Balaban et al. employ declarative semantic notations for automatically retargeting code libraries in large code bases [1]. They use a rich type system and a constraint solver to enable finding correct library call replacements that other-

wise could not be found automatically (due to subtle issues like synchronization).

The Feature Oriented Refactoring (FOR) work of Liu et al. recognizes the crosscutting and non-modular nature of the implementation of software features, which are often crosscutting [10]. For example, adding bounds checking to a data-structure could crosscut that structure's implementation. With FOR, certain types of programs can be refactored into a base program and modular feature refinements. The features are refactored and composed through the application of advanced delegation techniques. The application of FOR allows optional features, such as bounds checking, to be removed, allowing for better-suited variants.

Simonyi's Intentional Programming (IP) [16] is related in spirit to the goals of Arcum.[2] IP aims to have programmers work at the level of their intentions, allowing for easier change to programs. Instead of being a refactoring system, IP utilizes a program-as-database approach: if any linked entry changes, the change follows all links backward.

The REFINE system also employs a program-as-database approach, in addition to program templates, which can be used for both pattern matching and code transformation [8]. The code transformations discussed were not bi-directional in nature and are directed at uses such as "eliminate redundant multiplies by 1" and code mutations for test suite validation. The source template and destination template are bound in the same transformation rule, stopping short of a notion of a (persistent) concept. Also, alternative transformations cannot be introduced without duplicating existing rules.

As departure from REFINE, by Kozaczynski et al. employs semantic pattern matching—including control-flow and data-flow—to recognize concepts as part of a code transformation system for software maintenance [9]. Concepts were not persistent, however, and transformations were explicitly defined as predicate–action pairs, not inferred from patterns, thus limiting transformations to being one-way. In many respects this supports user-programmable role-based refactoring.

8. Conclusion

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

Arcum expands the opportunities for modular analysis and substitution for such concepts. Based on a paradigm of declarative pattern matching and substitution, Arcum specifications are declarative supplements to the program, neither modifying the code nor its behavior. Only the substitution process changes the code.

Arcum separates the behavior and implementation of a concept into Concept and Option declarations. An Option specifies the implementation of a Concept by specifying semantic patterns that correspond to the traits of the Concept. Both are parameterized, supporting reuse and the development of Concept libraries.

When the programmer uses a concept mapping to specify that a given Option instantiation is expected to hold in the program, the Arcum engine can check this by matching the Option's patterns over the program and then checking the matched elements against the Concept's behavioral constraints. If the programmer specifies that a new, different Option should now hold, the Arcum engine not only performs these checks for the old Option, but then replaces the matched elements with the code specified in the patterns of the new Option. Due to the declarative nature of the language, as well as the fact that the current Option is continuously checked, the transformation process can be run in either direction.

We preliminarily evaluated the Arcum approach by prototyping Arcum's transformation engine, completely developing the attribute Concept with three Options, and applying it to the Polyglot code base. We found that Arcum provided the feel of substitutability to the crosscutting concept, although the Arcum engine, as currently implemented, is slower than compilation. Incremental compilation techniques should dramatically reduce runtime.

Future work will generalize the Arcum framework, supporting AspectJ 5 as well as constraint checking that uses data-flow information. With these, much more powerful concepts will be specifiable, as well as supporting fluid AOP. These extensions, with more complete IDE support, will enable case studies of the application of Arcum in complex software systems.

[2] The name "Arcum" is derived from the Latin phrase "*intendere arcum*," which means "to aim a bow and arrow at" and is the metaphorical root of the word "intention" [3, p.333].

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] K. Beck and C. Andres. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, second edition, November 2004.

[3] D. C. Dennett. *Consciousness Explained*. Back Bay Books, 1992.

[4] 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.

[5] 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.

[6] G. Kiczales. Aspect-oriented programming: The fun has just begun. In *Software Design and Productivity Coordinating Group – Workshop on New Visions for Software Design and Productivity: Research and Applications*, Nashville, Tennessee, Dec. 2001.

[7] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *15th European Conference on Object-Oriented Programming (ECOOP 2001)*, pages 327–353, June 2001.

[8] 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.

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

[10] 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.

[11] 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.

[12] N. Nystrom, M. Clarkson, and A. Myers. Polyglot: An extensible compiler framework for java, 2003.

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

[14] D. J. Pearce and J. Noble. Relationship aspects. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, pages 75–86, New York, NY, USA, 2006. ACM Press.

[15] N. Rutar, C. B. Almazan, and J. S. Foster. A comparison of bug finding tools for java. In *ISSRE '04: Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE'04)*, pages 245–256, Washington, DC, USA, 2004. IEEE Computer Society.

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

[17] B. D. Sutter, F. Tip, and J. Dolby. Customization of java library classes using type constraints and profile information. In *Proc. ECOOP'04*, pages 585–609, 2004.

A. Arcum Code Listing

```
////////////////////////////////////////////////////////////
concept AttributeConcept
  (Class targetType, Type attrType, String attrName) {
  // Interface
  trait attrGet(Expr getExpr, Expr targetExpr);
  trait attrSet(Expr setExpr, Expr targetExpr, Expr valExpr);

  // Constraint checks
  require attrGet(accessExpr, target)
      || attrSet(accessExpr, target, ...):
    isAssignableFrom(attrType, TypeOf(accessExpr))
    && isAssignableFrom(targetType, TypeOf(target))
  onError:
    message("%s: The expression '%s' must be of type %s and"
      + " the target must be of type %s", PositionOf(accessExpr),
      accessExpr, attrType, targetType);

  require attrSet(setExpr, any, valExpr):
    !isSubExpression(setExpr)
    && isAssignableFrom(attrType, TypeOf(valExpr));

  require:
    isJavaIdentifier(attrName) && isReferenceType(attrType);
}

////////////////////////////////////////////////////////////
option InternalField implements AttributeConcept {
  var Field field;

  realize field with
    [ClassMember: class `targetType { ... `field ... }]
    && [Field: ?transient default `attrType `attrName] == field
    && isAssignableFrom([Type: Serializable], targetType)
       <=> isTransient(field)
  onError:
    message("The class %s must have a default access field"
      + " named '%s'", targetType, attrName);

  realize attrGet(getExpr, targetExpr) with
    [Get: `targetExpr.`field] == getExpr;

  realize attrSet(setExpr, targetExpr, valExpr) with
    [Set: `targetExpr.`field = `valExpr] == setExpr;
}

////////////////////////////////////////////////////////////
option StaticMap implements AttributeConcept {
  var Field mapField;

  let mapType =
    [Type: WeakIdentityHashMap<`targetType, `attrType>];
  let mapInit =
  [InitExpr: new WeakIdentityHashMap<`targetType, `attrType>()];

  realize mapField with
    [ClassMember: class `targetType { ... `mapField ... }]
    && [Field: static `mapType `attrName = `mapInit] == mapField;

  realize attrGet(getExpr, targetExpr) with
    [Call: `mapField.get(`targetExpr)] == getExpr;

  realize attrSet(setExpr, targetExpr, valExpr) with
    [Call: `mapField.put(`targetExpr, `valExpr)] == setExpr;

  trait access(Expr e): [`targetType.`mapField] == e;
  trait putCall(Expr e): [`e.put(..., ...)];
  trait getCall(Expr e): [`e.get(...)];
  require access(e): in(e, putCall) || in(e, getCall);
  onError:
    message("Only the get and set operations can be performed"
      + " on the %s map.", mapField);
}
```