The Challenges of Writing Reusable and Portable Aspects in AspectJ: Lessons from *Contract4J*

Dean Wampler
Aspect Research Associates *and* New Aspects of Software
33 W. Ontario St., #29F
Chicago, IL 60610
dean@aspectprogramming.com

ABSTRACT

Contract4J is a developer tool written in AspectJ and Java that supports Design by Contract programming in these two languages. It is designed to be general purpose and to require minimal effort for adoption by users. For example, adoption requires little customization and prior experience with AspectJ. Writing Contract4J demonstrated several issues that exist when writing truly generic and reusable aspects using today's technologies. This paper discusses those experiences and comments on ways our understanding and tooling could improve to make it easier to write such aspects. In particular, I discuss the importance of migrating from syntax-based pointcut definitions to semantically-rich metaphors, similar to design patterns.

Categories and Subject Descriptors

D.1.5 [**Programming Techniques**]: Object-oriented Programming. Aspect-oriented Programming

General Terms

Design, Standardization, Languages, Theory.

Keywords

Aspect-oriented software development, object-oriented software development, design, AspectJ, Java, Contract4J, Design by Contract.

1. Introduction

Writing generic, reusable library software is difficult. This is no less true for aspect libraries, partly because of the relative immaturity of aspect design and programming techniques, but it also reflects the inherent nature of aspects themselves.

This paper discusses the lessons learned and challenges encountered while implementing *Contract4J* [3], a generic, reusable framework for *Design by Contract* (DbC) [6] in Java and AspectJ, which is written in AspectJ.

1.1 Design by Contract

All components have a "contract" for use, whether it is stated explicitly or not, DbC is an explicit formalism for describing the contract of a component and for automating contract enforcement during the test process. It is a tool for locating logic errors (as opposed to runtime errors like heap exhaustion). To remove the testing overhead, tests are turned off for production deployments.

A component's contract includes the input requirements that must be satisfied by clients who use the component, called the *preconditions,* and the constraints on the component's behavior (assuming the preconditions are satisfied), including *invariant conditions* and *postconditions* on the work done by the component (*e.g.,* method return values).

DbC also prescribes the rules for contract inheritance, based on the Liskov Substitution Principle (LSP [9]), which says that a class B is considered a subclass of class A if objects of B can be substituted for objects of A without program-breaking side effects. For DbC, this means that subclasses can only change the contract for their parents in particular ways. Invariants cannot be changed. Overridden preconditions can *relax* the constraints, because the client program will always meet a stricter subset of input constraints, namely the subset specified by the parent class. This is *contravariant* behavior, because while subclassing is a "narrowing" of sorts, the preconditions are "widened". In contrast, the postconditions can be narrowed, a *covariant* change, because the reduced subset of results will always satisfy the wider set of results expected by the client program, as stipulated by the parent class contract.

DbC was invented by Bertrand Meyer for the Eiffel language [6], which supports it natively. In addition to Contract4J, various toolkits have been invented that provide Java support through libraries or external tools. These include the XDoclet-based *Barter* package [2] and *jContract* [4].

Design by Contract complements Test Driven Development (TDD). Even if a developer relies exclusively on TDD, understanding the contractual nature of interfaces helps clarify design decisions.

2. Overview of Contract4J

2.1 Design Goals for Contract4J

Contract4J provides support for DbC in Java in an intuitive way and with minimal adoption effort. *Intuitive* means that users can specify component contracts using familiar Java features and they can do this efficiently and conveniently without obscuring the component's abstractions. Contract4J allows developers to embed contract information in the classes, aspects, and interfaces adjacent to the points where the contracts apply. This is a practical convenience for the developer and also keeps the contract portion of the component together with the component's methods and attributes, so clients have access to the full interface specification, of which the contract is an important part. The developer specifies the contract details in an intuitive format using familiar Java syntax, annotations or a JavaBeans-like convention that I call "ContractBeans".

Adoption includes straightforward build or load-time modifications and writing contracts as part of the usual development process. Hence, even developers without prior AspectJ experience can adopt Contract4J quickly.

2.2 How Contract4J Is Used

I illustrate using Contract4J with a simplistic bank account example. Figure 1 shows the basic interface.

```
interface BankAccount {
  float getBalance();

  float deposit(float amount);

  float withdraw(float amount);
  …
}
```

Figure 1: Simplified BankAccount Interface

There are methods for retrieving the current balance, depositing funds, and withdrawing funds. The balance-changing methods return the new balance. The interface is simple enough, but it leaves unanswered questions. What if the user tries to withdraw more money than the account currently has? What happens if the `amount` parameter in either the `deposit` or `withdraw` method is negative. Specifying answers to questions like these makes the full contract explicit. Consider Figure 2.

```
@Contract
interface BankAccount {
  @Post("$return >= 0")
  float getBalance();

  @Pre("amount >= 0")
  @Post("$this.balance ==
         $old($this.balance)+amount
         && $return == $this.balance")
  float deposit(float amount);

  @Pre("amount >= 0 &&
        $this.balance - amount >= 0")
  @Post("$this.balance ==
         $old($this.balance)-amount
         && $return == $this.balance")
  float withdraw(float amount);
  …
}
```

Figure 2: BankAccount with Contract Details

The `@Contract` annotation signals that this class has a contract specification defined. The `@Pre` annotation indicates a precondition test, such as a requirement on the `withdraw` method that the input amount must be greater than or equal to zero and it must be less than or equal to the balance, so that no overdrafts occur. Note that we can refer to the attribute `balance` that is implied by the JavaBean's accessor method `getBalance`, where the `$this` keyword tells Contract4J that `balance` refers to a field in the `BankAccount` instance being tested. The `@Post` annotation indicates a postcondition test, for example that the `deposit` or `withdraw` method must return the correct new balance and the new balance must be equal to the "old" balance (captured with the `$old(…)` expression) plus or minus the amount, respectively. Not shown is an example `@Invar` annotation for invariant conditions, which can be applied to fields, methods, or classes. The field and class invariants are tested before and after every non-private method, except for field accessor methods and constructors, where the invariants are evaluated after execution (to permit lazy evaluation, *etc.*). Method invariants are tested before and after the method executes.

The original interface plus annotations specifies the behavior more fully by explicitly stating the expected behavior.

This example shows the syntax supported by the latest version of Contract4J. In this version, Contract4J uses Jakarta Jexl (Java Expression Language) [5], a runtime expression evaluator, to evaluate the test strings in the annotations. This happens in the context of aspects that advice locations where the annotations are used. Typically, before advice is used for preconditions, after advice is used for postconditions, and around advice is used for invariants[1]. If a test fails, an error is reported and program execution halts.

A second experimental syntax uses a JavaBeans-style naming convention, which I call "ContractBeans". Using this format, the `BankAccount` interface is shown in Figure 3.

```
abstract class BankAccount {
  abstract public float getBalance();

  boolean postGetBalance(float result) {
    return result >= 0;
  }

  abstract public
  float deposit(float amount);

  public boolean preDeposit(float amount) {
    return amount >= 0;
  }
  public boolean postDeposit(float result,
                             float amount){
    return result >= 0 &&
           result == getBalance();
  }

  abstract public
  float withdraw(float amount);

  public boolean preWithdraw(
      float amount) {
    return amount >= 0 &&
           getBalance() - amount >= 0;
  }
  public boolean postWithdraw(
                           float result,
                           float amount) {
    return result >= 0 &&
           result == getBalance();
  }
  …
}
```

Figure 3: "ContractBeans" Format

This version does not support the "old" construct for remembering a previous data value, so the contract tests shown are slightly less precise than in the previous example (*e.g.*, `result >= 0`, instead of the more accurate `result = $old(result) + amount`). Also, I have switched to declaring an abstract class so that the tests, which are now defined as instance methods, can be defined "inline". An alternative

[1] Sometimes different types of advice are used in certain cases, for technical implementation reasons, as discussed later.

would be to use an aspect and intertype declarations to supply default implementations of the test methods for the original interface.

Following a JavaBeans-like convention, the postcondition test for the `withdraw` method is named `postWithdraw`. (Compare with the JavaBeans convention for defining a `getBalance` method for a `balance` instance field.) This method has the same argument list as `withdraw`, except for a special argument at the beginning of the list that holds the return value from `withdraw`. The `preWithdraw` method is similar, except that its argument list is identical to the `withdraw` argument list. All the test methods return `boolean`, indicating pass or fail.

This version of Contract4J uses runtime reflection to discover and invoke the tests. It was implemented as a way of eliminating issues with the original version of the annotation-based approach. However, the extensive reflection imposes significant runtime overhead and writing the tests is a more verbose process with a less "obvious" association between the tests and the elements they are testing.

The original version of the annotation-based approach did not use a runtime expression interpreter. Instead, it used a precompilation step to generate very specific aspects for each test with the test string converted to Java code. A custom plug-in for Sun's Annotation Processing Tool (APT) was used to find the annotations in the source code and to generate AspectJ aspects for each one, before compilation. This implementation is the simplest of the three versions, with excellent performance, but the precompilation step is a barrier to adoption. The expression interpreter version eliminates this issue, but the implementation is more complex internally, in part because it uses reflection, as I will discuss in detail below. Hence, it has a higher runtime overhead than the APT version. However, because Contract4J is a development/test tool, the performance is acceptable.

The following summary compares the strengths and weaknesses of the implementations. More details are provided in the subsequent sections. For completeness, I also include pros and cons for two alternative ways of doing DbC, simple Java `assert` statements and *ad hoc* aspects, such as those used as examples in some of the AspectJ literature.

"ContractBeans" Version

Pros

- Could be used with Java 1.4 and earlier code, since it doesn't use annotations.
- Tests are written as regular Java methods, which can be reused outside of Contract4J.
- Because tests are normal methods, they are checked by the compiler and IDE for typos and other bugs.
- If the tests are declared public, they are a visible part of the interface for clients and subclasses to see.
- The JavaBeans-like convention follows a metaphor familiar to developers.

Cons

- Significant runtime overhead for extensive reflection calls.
- Tests are somewhat verbose, because of the method "boilerplate", compared to annotations.
- If the tests are not declared public, they are not a visible part of the interface for clients.
- The JavaBeans-like convention has a few idiosyncrasies that can result in the tests being ignored. There is no mechanism to warn the user when this happens.

Annotations, Version 1 (APT Preprocessor)

Pros

- Most intuitive and succinct way of specifying contracts.
- Most flexible use of annotations, including tests on method parameters.
- Test inheritance follows correct behavior for Design by Contract, not the rules for Java 5 annotation inheritance, *i.e.,* method tests are inherited, even though method annotations are not.
- Contracts are properly part of the public interface for clients, including Javadocs.
- Fastest performance.
- Although tests are defined as strings, because they are converted to compiled AspectJ code, test syntax errors are caught by the compiler.

Cons

- Preprocessor step requires nontrivial build changes, which may not work well with IDEs and other tools.
- Since tests are defined in annotations, they are not easily reused in other ways.
- Although test syntax errors are caught by the compiler, the error messages point to the generated aspects, not the original annotations. The user must manually "map" the errors back to the original annotations.

Annotations, Version 2 (Jexl Interpreter)

Pros

- Most intuitive and succinct way of specifying contracts.
- Contracts are properly part of the public interface for clients, including Javadocs.
- Easiest adoption process; only minor build modifications required.
- Good performance.

Cons

- Can't use annotations on method parameters (not supported by AspectJ; but there are workarounds).
- Because test annotations are evaluated at runtime, tests defined on methods are not inherited automatically, following the inheritance and runtime-visibility rules for Java annotations. Subclass method overrides *must* include the same annotations *manually*. Class invariant annotations are inherited, although putting them on subclasses, for consistency, is harmless.
- Idiosyncrasies of Jexl expression interpreter complicate test writing slightly. (Read the examples and Contract4J unit tests!)
- A minor build change still required, *i.e.,* compiling or at least weaving with AspectJ.
- Since tests are defined in annotations, they are not easily reused in other ways.
- Since tests are defined as strings, they are not checked by the compiler or IDE for obvious test bugs. Buggy tests show up at runtime as Jexl expression failures with unintuitive error messages.

***Ad Hoc* Aspects** (Aspects hand written to test specific cases)

Pros

- Straightforward with no need to adopt a third-party toolkit, like Contract4J, if you are already using AspectJ.
- Complete flexibility to define arbitrarily complex tests and to define them in separate files, if desired.
- Tests are checked by the compiler and IDE.
- Optimal performance.

Cons

- Extensive, repetitive boilerplate code required that is handled automatically by Contract4J.
- Harder to present the complete interface specification to clients.
- Can clutter code being advised. Putting test aspects in separate files is possible, but that approach decouples the test "specifications" from the code, making the full interface specification obscure.
- Requires active use of and expertise in AspectJ.

Java Asserts

Pros

- Simplest way of specifying contracts.
- No AspectJ or other 3rd-party toolkits required.

Cons

- Slightly more invasive in the code.
- No coherent view of the contract.
- Not part of the client-visible interface.
- Not visible to other tools.

All three implementations of Contract4J share a common limitation; they only partially enforce the rules for contract inheritance discussed previously. Both the ContractBeans and APT annotation versions will invoke parent-class tests, unless overridden in subclasses. Because Java annotations on methods are not inherited, the Jexl annotation version cannot apply the tests for a parent-class method to a subclass override *unless* the override has the same annotations[2]. (However, the override can omit the test string; Contract4J will locate the parent's test string.) In contrast the Jexl/Annotation form does better at ensuring that invariant tests are not changed by subclasses. None of the three versions ensures that subclass preconditions are contravariant and postconditions are covariant.

Overall, the Jexl annotation version offers the best compromise of features and ease of use.

AspectJ is used in all three versions, but the aspects, while conceptually similar, are very different in the two versions. In the annotation-based version, since a precompilation step is used, all the aspects involved are generated during that step. They have very specific pointcuts, with no wildcards, that pick out just the join points for which a particular test is defined. These aspects are simple, although there can be a lot of them in a system with many DbC tests defined. However, because they are so specific and because they use no reflection, they have low runtime overhead[3].

For example, here is a simplified version of the generated precondition test aspect for the `withdraw` method.

```
public aspect BankAccount_pre_withdraw {
    before (BankAccount ba, float amount):
        execution (float BankAccount(float))
          && this(ba) && args(amount) {
        if (amount >= 0 &&
          ba.getBalance() - amount >= 0) {
        handleFailure("…");
    }
}
```

Figure 4: Example Aspect 1

In contrast, because the ContractBeans version eliminates the precompilation step, all logic has to be embedded in the runtime engine. This means that more complicated and comprehensive aspects are required to advise *all* possible join points for which a test might exist. The corresponding advice then uses runtime reflection to discover the tests, if any, and to invoke those that are found. Even if no tests are present for a particular join point, the overhead still exists.

All the pointcut definitions (PCDs) in this version are scoped by a marker interface (no annotations are used to permit use with pre-Java 5 source code). No class will be advised unless it implements this interface. Rather than explicitly adding this interface to all class and interface declarations, it is usually easier to write a custom aspect that uses intertype declaration (ITD) to add this interface into the classes of interest, as shown in the example in Figure 5.

```
public aspect EnableContracts {
  declare parents: com.foo.bar..*
    implements ContractMarker;
}
```

Figure 5: Aspect ITD of a Marker Interface

I discuss the two Annotation implementations and the ContractBeans implementation because each exposes different challenges for writing generic, reusable aspects that involve non-trivial interactions with the advised classes. However, for practical use, the ContractBeans implementation is considered experimental and is not recommended for normal use. I will explore the details and issues of these implementations in greater detail below.

3. Challenges in Aspect-Oriented Software Development with AspectJ

Most example AspectJ aspects you see are either very specific, using pointcuts that reference particular classes, methods, and fields (*e.g.,* Figure 4), or they are very general, using pointcuts that reference package hierarchies and/or class and method names with wildcards. Examples of the former tend to be tightly coupled to the advised classes, such as policy enforcement aspects to ensure proper usage of libraries, *etc.* The latter aspects usually implement *orthogonal* concerns, which means they have loose or

[2] This is a possible future extension. It could be implemented using reflection, but with significant overhead.

[3] Because DbC is primarily a development tool and the tests are (usually) removed from production builds, performance is not a serious concern anyway, as long as it is "reasonable".

no coupling to the classes they advise. Examples include tracing and authentication wrappers.

The main issue this paper addresses is the difficulty of writing closely-coupled aspects in a generic and portable way, *e.g.,* without embedding target-specific details in the pointcuts.

Let us delve into the issues in more detail, starting with a discussion of some general issues with Aspect-Oriented Software Development itself, which is still a young discipline, where many details of good design and coding practice need further development.

3.1 Conceptual Issues with Aspects

One of the interesting differences between aspects and objects is the scope of a "component" in each approach. Well-designed objects have a limited scope with minimal *coupling* to objects outside of their "namespace" or package. They also have high *cohesion,* a well defined and focused purpose and conformance to appropriate global and local conventions that contribute to system-wide "coherence" and consistency.

Well defined aspects should also have these properties internally, but because they are explicitly designed to support cross-cutting behavior, their coupling to other components is more complicated. Aspects that cause nontrivial changes of state and behavior to these components require new thinking about the nature of "interfaces" between the aspects and the components they advise.

When attempting to design generic, reusable aspects, this issue leads to a conundrum. For an aspect to offer fine-grained and powerful functionality, it needs some detailed information about the components it will advise. However, these details increase coupling to those components and reduce general applicability and reusability. Typical pointcut definitions written today rely on naming conventions and other syntax constructs used in the advised components, rather than relying on higher-level abstractions.

This leads to what I call a *concern semantics mismatch.* Component field, method, and class names reflect the primary concern of the component, the *dominant decomposition* [7], and they are likely to change as the problem domain understanding and/or the scope of the solution evolve. Pointcuts are part of a different domain, that of the cross-cutting concern, yet they are relying on the unrelated names and conventions in the components they advise, whose evolution will be "unexpected", from the concern's perspective, leading to fragile interdependencies.

The long-term solution is the development of higher-level design abstractions. The aspect-component relationship should be more of a "peer" relationship like the one that exists between objects today, rather than the approach commonly used where the aspect is "doing something" to another component. The noun "advice" and the concept of *obliviousness* reflect this bias, unfortunately.

Much of the research on aspect-oriented design (AOD) occurring now is moving away from this emphasis on oblivious insertion of advice and moving towards interface-based design approaches, *e.g.,* Aspect-Aware Interfaces [7] and Crosscutting Programming Interfaces (XPI) [8]. A compromise design strategy is emerging, where components will need to be "aspect aware", in the sense that they will need to expose state and behavior of potential interest to "clients", aspects as well as objects, without actually assuming particular details about those clients. The art of aspect-aware interface design will be to expose abstractions that are easily adapted by concerns that are different from the component's primary domain. I expect that most aspects will implement the *Bridge* pattern [10], connecting exposed interfaces with concern libraries. In fact, most aspects today follow this model, just in a more *ad hoc* fashion and with coupling to the fragile details of the advised classes, rather than coupling to more abstract and therefore stable interfaces. In other words, AOD is now expanding the established principles of object interfaces to support the new and unique needs of aspects.

3.1.1 Contract4J as a "Design Pattern"

You can view the annotation and the ContractBeans forms of Contract4J as *syntactically* different, yet *semantically* equivalent forms of an *ad hoc* "protocol", essentially a design pattern, which is used by a class to provide a design-pattern protocol for specifying the module's contract in a way that makes minimal assumptions about interested "clients" [6]. While invented for Contract4J, this protocol could be supported by a variety of other compile-time and run-time tools, including documentation tools and testing tools that generate unit tests from the annotations. The protocol is a mini domain-specific language (DSL) for DbC and it is conceptually consistent with the work on interface-based design in aspect systems [7-8]. In fact, a fruitful exercise would be to recast Contract4J in XPI formalism, for example.

3.2 Practical Challenges with AspectJ

Returning to AspectJ, its pointcut language is very powerful, but until recently, it has relied exclusively on concrete naming conventions, leading to the *concern semantics mismatch.* However, Java 5 annotations are a useful first step towards defining "interfaces" that support other concerns. Well-chosen annotations provide meaningful metadata about the element that tends to be more stable than naming idiosyncrasies of the element itself. Also, useful metadata will express information of interest to other concerns, implemented as aspects, in a more decoupled fashion. AspectJ 5 supports PCDs that match on annotations. Using annotations in Contract4J makes it unnecessary for it to know specific details about the classes it advises.

Put another way, most reusable aspects that have been documented to date are really reusable aspect *patterns.* They require customization of the PCDs to match on specific naming conventions for the project in question. The advices may also require modification. Truly generic PCDs that consist of almost all wildcards are often too broad, needlessly affecting far more join points than are really required.

However, having just made the argument that we need higher-level abstractions, it must be said that the lower-level join-point matching constructs currently available are still essential. Contract4J would not be possible without them. While annotations are used as "markers" for tests and for defining the test expressions, all the PCDs used in Contract4j still do matching on method and constructor calls or executions and field "gets" and "sets". This is in part an idiosyncrasy of Contract4J, since it supports detailed assertions about the component logic and those assertions have to be evaluated at very specific join points. Many other aspect-based tools and components will continue to require the lower-level constructs.

Let us consider the specific issues encountered in the three versions of Contract4J.

3.2.1 Contract4J Using Annotations, Version 1

Ironically, the original annotation-based version of Contract4J did not use any annotation-based PCDs. The precompilation step used a plug-in for Sun's Annotation Processing Tool (APT) to extract the annotation information and generate AspectJ code with PCDs that match on the specific classes, fields and methods with tests. The actual annotations are ignored in the PCDs, as they are no longer needed.

Figure 4 showed a simplified version of a typical aspect generated by this implementation. It uses the lower-level join point matching constructs, based on specific and explicit element names, because one aspect is generated for every annotation found (potentially creating a lot of aspects). This implementation proved to be the most straightforward to develop, because it did not require the more sophisticated PCDs needed in the subsequent Jexl aannotation version of Contract4J nor the more sophisticated introspection required in both the Jexl version and the ContractBeans version.

In fact, using a preprocessor tool (APT) avoided all the problems of the subsequent two versions of Contract4J, because using APT, a tool specific to the "annotation domain", if you will, handled all the dirty work of finding annotations and their context information.

3.2.2 Contract4J Using Annotations, Version 2

This is the most recent version of Contract4J and it is the one that will be maintained going forward. It uses annotation-based pointcuts to find the contracts and then uses the Jakarta Jexl expression interpreter [5] within advice to evaluate the test expressions at runtime.

Of the three implementations, this one has the most sophisticated aspects, combining nontrivial PCDs and construction of test context data that is passed to Jexl. The latter process uses Java's and AspectJ's reflection libraries to fill in information that can't be "bound" by the PCDs. In fact, the bulk of the code exists to support collecting context data and passing it to Jexl. The static typing of Java and the lack of "native" support for scripting (dynamic generation and evaluation of expressions) greatly complicated the implementation.

Consider two example aspects from this version. The first aspect implements method precondition tests and the second implements field invariants for field reads and writes.

3.2.2.1 Aspect for Method Precondition Tests

The PCD for this aspect is shown in Figure 6[4]

```
pointcut preMethod (                    // 1
  Pre pre, ContractMarker obj) :        // 2
    if (isPreEnabled()) &&              // 3
    !within_c4j() &&                    // 4
    execution (@Pre !static             // 5
      * ContractMarker+.*(..)) &&       // 6
    @annotation(pre) && this(obj);      // 7
```

Figure 6: PCD for Method Preconditions

Line 2 declares that two parameters will be bound, the annotation object, `pre`, which contains the test expression, and an object that implements the marker interface `ContractMarker`. This binding actually happens in line 7. The marker interface is injected into all types with the @Contract annotation (using a separate aspect), to make inheritance of tests easier to support; note the use of `ContractMarker+` in lines #4 and #7, to make sure that the join points in subclasses are matched. The `ContractMarker` object is the object under test.

Line 3 checks that preconditions tests are actually enabled, which can be configured globally through API calls and properties. (Postcondition and invariant tests can also be controlled this way.) Note that the preferred alternative for production deployments is to exclude the Contract4J aspects from the build, so that no DbC overhead is incurred at all. The referenced PCD in line 4 (not shown) is a typical PCD for excluding advising of the Contract4J code itself, to prevent infinite recursions, *etc.*

The key section of the PCD is in lines 5 and 6, highlighted in bold, where matching is done on execution join points of methods in `ContractMarker` and its subclasses. This PCD excludes constructors (handled separately) and only matches on nonstatic methods that have the `@Pre` annotation. Static methods are excluded because contracts focus on tests of instance state[5]. Note that since method annotations are not inherited in Java, we must require that the annotation appear on all method overrides[6]. If a subclass override does not have the same annotation, but the superclass implementation is invoked using `super…()`, the superclass method with the annotation will still be tested. However, even in this case Contract4J can't detect possible violations of the contract in the subclass method without the annotation and Contract4J can not currently detect that the annotation is missing.

Requiring the user to annotate all method overrides consistently is a design constraint reflecting a Java annotation limitation. However, even if method annotations where inherited, there is no way to write a pointcut that says "match a method in the class hierarchy if one of its ancestor methods has annotation A". Reflection could be used to handle this case (a possible enhancement), but it would be somewhat expensive to do.

An alternative would be to inject the missing annotation, if AspectJ's `declare parents` facility were generalized to support `declare method annotations`, for example, which could add an annotation to a method[7], assuming this is technically feasible. For this to be useful in the particular case discussed here, it would also be necessary for the declare statements to support a wider range of predicates, such as the pseudocode example suggested in Figure 7:

```
declare annotations:
  @Pre * ContractMarker+.method          // 1
```

[4] Some details have been altered for clarity and simplicity.

[5] However, you could argue that global (static) state could also be subject to testing. This may be supported in a future release.

[6] This was not a requirement for the original APT-based implementation, because the generated aspects no longer needed the annotations and would match on subclass overrides.

[7] Class annotations are already supported. Field annotation support is not needed in this case.

```
if (!@Pre * ContractMarker+.method  // 2
  && @Pre *ContractMarker.method)   // 3
```

Figure 7: "declare annotations" Extension

Here, the `@Pre` annotation is added to *`method`* in any subclass of `ContractMarker` (line 1) if it isn't already present (line 2), but it is present on the method in the top-level class or interface that defines the contract (line 3). How *`method`* is determined is intentionally left vague, but it would be the same method in all three lines. Note that Contract4J will locate the parent's test expression or generate a default expression, if no test expression is defined in a particular annotation.

At the very least, if this automatic mechanism can't be implemented (or the effort isn't otherwise justified), it would be useful if a mechanism exists to catch the user error of not annotating method overrides in subclasses.

In general, Contract4J's reliance on annotations points out some of the idiosyncrasies of Java 5 annotations, especially when used to represent a concept like DbC where expectations for inheritance behavior are different than for annotations.

3.2.2.2 Aspect for Field Invariant Tests When Fields Are Read or Written

Only invariant tests are supported for field reads and writes[8]. The lack of annotation inheritance that plagues method contract tests is not an issue here, since the field only "exists" in the class in which it is defined. Hence, if a field is annotated, all direct accesses will be correctly advised. However, field advice does have its own nuances.

3.2.2.2.1 Field "Gets"

Figure 8 shows the PCD for field "gets".

```
pointcut invarFieldGet (                // 1
  Invar invar, ContractMarker obj):     // 2
    if (isInvarEnabled()) &&            // 3
    !within_c4j()  &&                   // 4
    !cflowbelow (execution              // 5
       (ContractMarker+.new(..))) &&    // 6
    get(@Invar * ContractMarker+.*) &&  // 7
    @annotation(invar) && target(obj);  // 8
```

Figure 8: PCD for Field Get Invariants

The first four lines are very similar to those for the method precondition PCD in Figure 6, with `Invar` substituted for `Pre`. In lines 5 and 6, I exclude field accesses that occur inside constructors, since we shouldn't expect the field to be initialized properly until the end of constructor execution. A separate aspect handles this special case. It uses the `percflow` instantiation model and matches on the initialization join points. Another aspect records accesses of any annotated fields and then after advice on the constructor evaluates the corresponding field tests after construction completes.

Because the field invariant test is evaluated at the end of construction, such a contract specification is not appropriate for a field that will be initialized on demand. In this case, a `@Post` test on the corresponding `get` method should be used.

[8] In principle, field pre- and postconditions could also be supported, but these tests are best added to bean property `get` and `set` methods, instead.

Back to Figure 8; note that the pointcut does not declare an Object argument for the returned field value, which could then be bound in an after `returning advice`, as shown in Figure 9.

```
after (                                 // 1
  Invar invar, ContractMarker obj)      // 2
  returning (Object result):            // 3
    invarFieldGet(invar, obj, result){  // 4
    …
}
```

Figure 9: Possible After Returning Advice

In fact, `around` advice is used for this and most other `@Post` test cases because of a special test feature supported by Contract4J, namely the ability to capture "old" values of context data, such as the value of the field before it is changed, so that the old and new values can be compared in some way[9]. I used this feature in the Figure 2 example to check that a withdrawal or deposit changed the account balance appropriately.

If the test expression specifies any "old" data, it is captured first in the around advice. Then, `proceed` is called to execute the join point and the value it returns is saved as the new field value.

3.2.2.2.2 Field "Sets"

Figure 10 shows the PCD for field "sets".

```
pointcut invarFieldSet (                // 1
  Invar invar, ContractMarker obj,
      Object arg):                      // 2
    if (isInvarEnabled()) &&            // 3
    !within_c4j()  &&                   // 4
    !cflowbelow (execution              // 5
       (ContractMarker+.new(..))) &&    // 6
    set(@Invar * ContractMarker+.*) &&  // 7
    @annotation(invar) && target(obj)   // 8
      && args(arg);                     // 9
```

Figure 10: PCD for Field Get Invariants

The structure is very similar to the PCD in Figure 8 for field "gets", but now there is an extra `Object` parameter for the value being assigned to the field and of course `set(…)` join points replace `get(…)` join points.

Note that there is no way to actually bind an object to the field itself! Only the object being assigned to the field can be bound. Since Java variables are either references to objects or primitive values, this distinction is not important for Contract4j purposes, but it is possible that other applications using generic aspects may need to make this distinction. Perhaps AspectJ should support explicit binding to the field itself.

3.2.2.3 Advice

The advices used with these PCDs are all very similar. They use Java and AspectJ reflection APIs to fill in missing context information needed by the test expressions. They call support classes to create "default" test expressions when none is specified in the annotation. For invariant tests, they examine corresponding parent-class tests, if any, to ensure that the invariant tests are the same[10]. Finally, the advices call other support classes to package

[9] Only supported for primitives, Strings, and a few other classes.

[10] Only simple string comparison, ignoring white space, is currently supported, not true "semantic" equivalence. Hence "a==b" appears different from "b==a".

the information into the context structures required by Jexl and finally Jexl is invoked to execute the test. On failure, an error message is reported and program execution is stopped abruptly.

3.2.3 Contract4J "ContractBeans" Version

For completeness, I discuss the experimental *ContractBeans* (JavaBeans-like) version of Contract4J. The (PCDs) for this version are relatively simple, because most of the work must be done using reflection. Suppose I am testing the following class that uses the ContractBeans test approach.

```
class Foo (
  public int method(int i) {…}

  public boolean preMethod(int i) {…}

  public boolean
  postMethod(int result, int i) {…}
}
```

Figure 11: Foo Class Using *ContractBeans* Tests

Consider the precondition test case, where I could write a pointcut like the following.

```
pointcut pre(Foo foo, int i):
  call(int Foo.method(int)) &&
  hasMethod(boolean Foo.preMethod(int))
  && target(foo) && args(i);
```

Figure 12: Desired Pointcut for Precondition

The `hasMethod` pointcut specifier is a new undocumented experimental feature in AspectJ5 which tests for the existence of a method.

However, it is not possible to generalize this pointcut to arbitrary target classes and method signatures. It would require extending AspectJ to support a regular-expression syntax for matching strings, *e.g.*:

```
pointcut<T> pre(T t, Object[] args):
  call(* \(\T\)+.\(\M\)(\(\A\))) &&
  hasMethod(boolean $1.pre(cap($2)($3))
  && target(t) && args($3.values());
```

Figure 12: Possible Pointcut Regular Expression Syntax

In this contrived example, "\(…\)" indicates a capturing group, "\T" matches a type, "\M" matches a method name, "\A" matches the argument list, "$N" substitutes the value of the N[th] capturing group, and "$3.values()" returns the list of values corresponding to the argument list captured by "$3"[11]. The made-up method "cap" handles capitalization of the method name, *i.e.,* conversion of the the first letter in the method name to upper case.

However, this syntax is hard to read and would therefore be error prone to use. Also, the merits of implementing regular expression support may not outweigh the effort required to implement it.

Instead, the ContractBeans version of Contract4J uses relatively simple, wide-reaching pointcuts and extensive runtime reflection to locate the test methods. First, end user is required to declare a "scoping" aspect that uses ITD to insert a marker interface into all classes where tests exist (or might exist), *e.g.*,

```
aspect scope (
  declare parents: (com.foo.bar..*)
    implements ContractMarker;
}
```

Figure 13: "Scoping" Aspect

Straightforward pointcuts are used to locate *all* possible join points where tests might be evaluated, within the defined scope. For example, the method precondition pointcut is shown in Figure 14.

```
pointcut preMethod (ContractMarker obj):
    if (isPreEnabled()) &&           // 3
    !within_c4j() &&                 // 4
    execution (!static               // 5
      * ContractMarker+.*(..)) &&    // 6
    this(obj);                       // 7
```

Figure 13: ContractBeans Pointcut for Method Preconditions

The key section of the PCD is lines 5 and 6, shown in bold. The rest of the PCD is similar to the boilerplate seen before. In fact, the whole PCD looks very similar to the annotation-based PCD for method preconditions shown in Figure 6, except that there are no annotations involved here. The annotation-based PCD will match only those join points where tests are actually defined, whereas the PCD in Figure 13 will match on *every* non-static method in the `com.foo.bar` hierarchy, adding significant overhead.

The corresponding advice uses reflection to determine if there is a `pre`*`Method`* test method to go with every method *`method`* found. The logic must look for methods with the appropriate name, that return boolean and that have a matching argument list, as discussed previously. The reflection adds a significant amount of overhead. Found methods are cached, but there is a non-trivial amount of setup effort required to determine the "key" for such a cache, so only modest performance gains are realized. In this case, it would help if AspectJ had a way of programmatically removing advice at the current join point, when a test method is not found by reflection, so all futile searches are never repeated.

3.2.4 User Adoption Issues

Because aspects can potentially affect the entire system, almost all aspect libraries include some mechanism for scoping the PCDs to only those packages and classes of interest. The following approaches are the most common.

- Define an abstract scoping pointcut in the library aspect and require the user to implement a concrete version of it that defines the packages and specific classes of interest. This minimizes, but does not eliminate the knowledge of AspectJ required by the user and the customization required to adopt a library.
- Define a marker interface that all library pointcuts use as a scoping construct, then require the user to "implement" or "extend" this interface in all classes or interfaces, respectively, where the user wants the aspect to apply. This is invasive if done manually. Instead, the user can write an aspect that uses intertype declaration to apply the interface where desired. (See, *e.g.,* Figure 13) This approach imposes about the same adoption effort and skill on the user as the scoping aspect option.
- Define an annotation that can be used instead of a marker interface (for Java 5 projects). Annotations can also be

[11] I said this was contrived!

introduced with ITD. Contract4J defines a `@Contract` annotation for this purpose. The curious thing about Contract4J usage is that the user will typically add this annotation manually, because the user will also need to add the other test annotations anyway in order to define tests. Hence, in practice, the user of Contract4J never needs to write any AspectJ code, although it will be necessary to introduce AspectJ into the build process.

- Define abstract base aspects and require the user to implement a derived aspect that implements abstract methods, supplies required callbacks, *etc.* A variation of this approach is to have concrete aspects use regular Java interfaces that the user must implement and "wire" to the aspect. This approach requires some user effort, but uses only familiar Java techniques.

The annotation form of Contract4J uses all these techniques internally. For example, classes with class-level "@Invar" tests get the marker interface `ContractMarker` through ITD, even though the annotations themselves are inherited. This apparent redundancy makes it easier to write PCDs that pick out the same join points on subclasses, even when they don't have the same "@Invar" annotation.

4. Conclusions

AspectJ's pointcut language enables succinct, yet powerful aspects when advice is needed at specific join points in known packages and classes. However, it is hard to write generic aspects that don't assume specific signature conventions, yet need details of the join points where they match in order to interact with the join points in non-trivial ways. Such aspects must use reflection to determine the additional information that they need.

Contract4J demonstrates the issues encountered when implementing a generic, reusable aspect library. In fact, it uses many of the "types" of PCDs you might expect to write, at least those focused on a single class, including field accesses, method calls, and instantiation, where specific coupling and computations are required for each case. Hence, developers of other generic library aspects are likely to encounter one or more of the same issues encountered in Contract4J. These issues will be a barrier to widespread development of rich AspectJ libraries unless some enhancements are made that simplify the issues involved.

Note that AspectJ 5 configuration files can be used to define some explicit name dependencies, thereby removing them from aspect code. However, this mechanism is insufficient for the needs of tools like Contract4J.

One possible solution is to extend the join point model with regular expression-like constructs, so that more sophisticated join point matching can be done on signature conventions without requiring explicit knowledge of "irrelevant" naming details. The aspect developer would then be able to bind more information through the PCD arguments for use in the advice bodies, thereby reducing the amount of reflection code required[12].

[12] For most users, the relative runtime efficiencies of reflection *vs.* PCD binding, which may be similar, will be less important than the ease of development using either approach.

However, focusing on low-level constructs is probably the wrong enhancement strategy. Efforts to develop the theory and practice of aspect interfaces [7-8] are more important for the long-term evolution of AspectJ and AOSD in general. Components and aspects should be joined through interfaces that use the semantics of the concern, rather than being expressed through lower-level points of code execution, leading to the *concern semantics mismatch*.

Annotations that express meta-information about components are a first practical step in this direction. The Contract4J annotations form a design pattern that exposes key usage information about the component, in this case constraints on usage. Clients, including Contract4J aspects, IDEs, test generators, *etc.* interested in the "usage constaints concern" can work with the components in nontrivial ways through this "interface". However, even when matching on annotated join points in the Contract4J PCDs, the advice bodies still contain lots of low-level "plumbing", including calls to reflection APIs. Hence, annotations alone are not generally sufficient as an "aspect interface" to easily write powerful, yet generic aspects.

5. REFERENCES

[1] http://www.aspectj.org/

[2] http://barter.sourceforge.net/

[3] http://www.contract4j.org

[4] http://www.jcontract.org/

[5] http://jakarta.apache.org/commons/jexl/

[6] B. Meyer, *Object-Oriented Software Construction*, 2nd edition. Prentice Hall, Saddle River, NJ, 1997.

[7] G. Kiczales and M. Mezini, "Aspect-Oriented Programming and Modular Reasoning," *Proc. 27th Int'l Conf. Software Eng.* (ICSE 05), ACM Press, 2005, pp. 49-58.

[8] W. G. Griswold, *et al.,* "Modular Software Design with Crosscutting Interfaces", *IEEE Software*, vol. 23, no. 1, 2006, 51-60.

[9] Barbara Liskov, "Data Abstraction and Hierarchy," SIGPLAN Notices, vol. 23, no. 5, May, 1988.

[10] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns; Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, MA, 1995.