Contract4J for Design by Contract in Java: Design Pattern-Like Protocols and Aspect Interfaces

Dean Wampler
Aspect Research Associates *and*
New Aspects of Software

dean@aspectprogramming.com

ABSTRACT

Recent trends in Aspect-oriented Design (AOD) have emphasized interface-based modularity constructs that support noninvasive advising of components by aspects in a robust and flexible way. We show how the AspectJ-based tool *Contract4J* supports *Design by Contract* in Java using two different forms of a design pattern-like protocol, one based on Java 5 annotations and the other based on a JavaBeans-like method naming convention. Neither form resembles conventional Java-style interfaces, yet each promotes the same goals of abstraction and minimal coupling between the cross-cutting concern of contract enforcement and the components affected by the contracts. Unlike traditional implementations of design patterns, which tend to be *ad hoc* and require manual enforcement by the developer, the Contract4J protocol offers at least partial support for compile time enforcement of proper usage. This example suggests that the concept of an interface in AOD should more fully encompass usage protocols and conversely, aspects give us ways to enforce proper usage programmatically.

Categories and Subject Descriptors

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

General Terms

Design, Theory.

Keywords

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

1. INTRODUCTION

Much of the current research work on Aspect-Oriented Design (AOD) is focused on understanding the nature of component and aspect interfaces with the goal of improving modularity, maintainability, *etc.,* just as pure object-oriented systems emphasize interface-based design [1-2]. Because aspects are cross-cutting, they impose new challenges for design approaches that don't exist for objects, but they also offer new capabilities that can be exploited.

Griswold, *et al.* [2] have demonstrated one approach to designing interfaces, called *Crosscutting Programming Interfaces* (XPIs), which resemble conventional object-oriented interfaces with additions such as pointcut declarations (PCDs) that hide component join point details behind abstractions. The interfaces are implemented by components that wish to expose state or behavior to interested "clients", *e.g.,* aspects. The aspects then use the interfaces' PCDs only, rather than specifying PCDs that rely on specific details of the components. This approach effectively minimizes coupling between the components and the aspects, while still supporting nontrivial interactions. For our purposes, it is interesting to note that the XPI PCDs also impose contract-like constraints on what the components, which are the interface implementers, and the aspects are allowed to do. For example, a component may be required to name all state-changing methods with a prefix like "set" or "do", while an aspect may be required to make no state changes to the component.

Contract4J [3] is an open-source tool that supports *Design by Contract* (DbC) [4] in Java using AspectJ for the implementation. It demonstrates an alternative approach to aspect-aware interface design with two syntax forms supporting a common protocol that will be suitable for many situations. Instead of specifying methods to implement, like a traditional Java interface, one form uses Java 5 annotations and the other uses a method naming convention. Both approaches are essentially a design pattern [5] that must be followed by components that wish to expose key information, in this case a usage contract, and by clients interested in the usage contract, which they can "read" from the components if they understand the protocol, without having to know other details about the components.

2. DESIGN BY CONTRACT AND CONTRACT4J

Briefly, Design by Contract (DbC) [4] is a way of describing the contract of a component interface in a programmatically-testable way. *Preconditions* specify what inputs the client must provide in order for the component to successfully perform its work. *Postconditions* are guarantees made by the component on the results of that work, assuming the preconditions are satisfied. Finally, *invariant conditions* can be specified that hold before and after any publicly-visible component operation. The test conditions are written as executable code fragments to support runtime evaluation of the tests. When the tests fail, program execution stops immediately, forcing the developer to fix the problem immediately. Hence, DbC is a development tool for finding and fixing logic errors. It complements Test-Driven Development [6].

Contract4J supports DbC in Java as follows. The component developer annotates classes, fields, and methods with annotations that define the condition tests as Java boolean expressions. Aspects advice the join points of these annotations and evaluate the expressions at runtime to ensure that they pass. If not, program execution is halted.

Here is an example using a simplistic `BankAccount` interface.

```
@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 1: BankAccount with Contract Details

The Contract4J annotations are shown in **bold.** The `@Contract` annotation signals that this class has a DbC specification defined. The other annotations are ignored unless this annotation is present. Contract4J also includes aspects that will generate compile-time warnings if the other annotations are present without the `@Contract` annotation, an example of partial programmatic enforcement of proper usage[1]. The `@Pre` annotation indicates a precondition test, which is evaluated before the join point executes. The `withdraw` method has a requirement that the input `amount` must be greater than or equal to zero and the `amount` must be less than or equal to the balance, so that no overdrafts occur. The expression `$this.balance` refers to an instance field that is implied by the JavaBean's accessor method `getBalance` defined in the interface. The `@Post` annotation indicates a postcondition test, which is evaluated after the join point executes. The `deposit` or `withdraw` method must return the correct new balance (specified with the `$return` keyword) and the new balance must be equal to the "old" balance (captured with the `$old(..)` expression) plus or minus the input amount, respectively. Not shown is an example `@Invar` annotation for invariant specifications, which can be applied to fields and classes, as well as methods. The field and class invariants are tested before and after every non-private method executes, 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 test expressions are strings, since Java 5 annotations can't contain arbitrary objects. Aspects match on join points where the annotations are present. The corresponding advice evaluates the test strings as Java expressions using a runtime evaluator, the Jakarta Jexl interpreter [7].

So, including the annotations specifies the behavior of the component more fully, by explicitly stating the expected behavior and eliminating ambiguities about what would happen if, for example, `amount` were less than 0.

A separate, experimental implementation of Contract4J, called *ContractBeans,* supports an alternative way of defining test conditions, where the component implementer writes the tests as special methods that follow a JavaBeans-like [8] signature convention,. Here is the same `BankAccount` interface expressed using this approach[2].

```
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 2: "ContractBeans" Format

An abstract class is used, rather than an interface, so that the tests, which are now defined as instance methods, can be defined "inline". (An alternative would be to use an aspect with 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 `getFoo` method for a `foo` 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

[1] The @Contract annotation is not strictly necessary, but it makes the Contract4J implementation more efficient and it makes the code more self-documenting. A future release may drop the requirement for it to be present.

[2] Actually, this implementation doesn't support the "old" keyword for comparing against a previous value, so the tests shown reflect this limitation.

identical to the `withdraw` argument list. All the test methods must return `boolean`, indicating pass or fail. Invariant methods follow similar conventions.

This version of Contract4J advices *all* the fields and methods in a class, then uses runtime reflection to discover and invoke the tests, when present. Hence, this implementation has significant runtime overhead and the method based approach has several drawbacks compared to the annotation approach, including greater verbosity when writing tests and a less obvious connection between the tests and the methods or fields they constrain. It is discussed here because it demonstrates the approach of using naming patterns to convey meta-information of interest to other concerns. However, as a practical tool, the annotation form of Contract4J is more usable.

More details on the design and usage of Contract4J, as well as a discussion of general issues encountered during its development, can be found in [9] and [3].

3. THE CONTRACT4J PROTOCOL AND ASPECT INTERFACES

The two approaches are different syntax forms for a design pattern, in the sense that they define a protocol for interaction between a component and clients of its contract, with minimal coupling between them. In this case, the protocol is used by aspects to find tests and execute them before and/or after a method invocation or field access. Compare this to the well-known *Observer* pattern where a state change in a subject triggers notifications to observers. In fact, many of the better-known patterns could be implemented, at least in part, using similar techniques. A simple implementation of *Observer* would be to annotate state-changing methods in potential subjects with an annotation, *e.g.,* @ChangesState. Aspects could advice these methods with `after` advice to retrieve the new state and react accordingly.

Using either form of the Contract4J protocol in an interface or class enhances the declaration with additional usage constraints that complete the specification of the component's contract, both the usage requirements that clients must satisfy, and the results the component guarantees to produce. DbC also promotes runtime evaluation of these constraints.

Consistent with the general goals of interface-based design, including the recent work on aspect-aware interfaces, the component remains agnostic about how the contract information is used. Of course, the two syntax forms were designed with Contract4J's needs in mind. However, other aspect-based tools could exploit these protocols for other purposes. Just as IDEs exploit JavaBeans conventions, the Contract4J protocol could be used, *e.g.,* by code analysis tools, automated test generation tools, and the code-completion features offered by most IDEs (*e.g.,* to warn when method arguments are clearly invalid). Hence, for a class of situations like DbC, a protocol like the two in Contract4J effectively provides an interface used by aspects for cross-cutting concerns. Specifically, if an aspect needs to locate all methods, constructors and fields meeting certain criteria that can be indicated through an annotation (or naming convention) and some action needs to be taken in the corresponding advice and that action can be expressed through the same syntax conventions, then this concern can be implemented in a similar way.

For example, the EJB 3 specification defines annotations that indicate how to persist the state of the annotated "Plain Old Java Objects" (POJOs) [12]. Actually, this example could be considered a case of tangling an explicit concern in the POJO; the annotations should ideally be more generic, perhaps conveying general lifetime information (*e.g.,* must live beyond the life of the process) and indicating what properties are required for uniquely specifying the state (*vs.* those that are transient properties). They should convey enough information that a persistence aspect could "infer" the correct the behavior (and similarly, other, unrelated aspects could infer what they need to know for their needs). Finding the right balance between explicitness and generality is clearly an art that needs to be developed.

Similar examples of using annotations to integrate with infrastructure services and containers include annotations that flag "sensitive" methods that trigger authentication processes before execution, annotations that provide "hints" to caching systems, and annotations that indicate potentially long-running activities that could be wrapped in a separate thread, *etc*.

4. DISCUSSION AND FUTURE WORK

The two Contract4J protocol forms demonstrate an idiomatic "interface"-based approach to aspect-component collaboration that are more like a design pattern than conventional interfaces, including the XPI form discussed by Griswold, *et al.* [2]. It is interesting that [2] calls for an explicit statement of the contract of behavior implied for both the components that implement the XPI and the aspects that use them to advise components. The unique characteristics of aspects make an explicit contract specification more important, because the risk of breaking existing behavior is greater, yet no programmatic enforcement mechanism currently exists. In contrast, Contract4J is specifically focused on contract definition *and* enforcement. Hence, Contract4J could be used to enforce the contract section of an XPI. Conversely, Contract4J could be extended to support the XPI-style of contract specification.

In general, Contract4J's protocol and the inclusion of contract details in XPI both suggest that good aspect-aware interfaces can go beyond the limitations of conventional interfaces, which provide a list of available methods and public state information, but usually offer no guidance on proper usage and never offer programmatic enforcement of that usage. Aspects offer the ability to monitor and enforce proper usage, which can then be specified as part of the interface.

With this support, design patterns can be programmatically enforceable and not just *ad hoc* collaboration conventions that require manual enforcement. The enforcement can be implemented as aspects and specified using a new kind of interface [10]. This would raise the level of abstraction in aspect design from focusing on points in a program's execution to designing nontrivial component collaborations at a higher level. Another example of "aspects as protocols" is the work of Jacobson and Ng on use cases as aspects [13]. Use cases are units of work involving collaborations of components. It is necessary for use case implementers to understand the usage constraints of the components and for components to expose suitable programmatic abstractions that permit them to be used in use cases. Currently, use cases are more like "extended" design

patterns, relying on *ad hoc* conventions and manually-enforced usage compliance.

For Contract4J itself, future work will focus on completing the partial support for enforcing proper usage of the protocol. For example, while Contract4J now warns the user if contract annotations are used without the class @Contract annotation, it contains few additional enforcement mechanisms. Enhancing this support will further clarify how design pattern-like protocols can be specified in aspect-aware interfaces in a way that is programmatically executable and enforceable. This capability will be necessary to remove the current *ad hoc* nature of design patterns and make them as rigorously precise as classes and aspects themselves.

Possible extensions to Contract4J's functionality include extending the notion of a "contract" beyond the domain of DbC. For example, there is no built-in support currently for enforcing contracts *between* components, as opposed to enforcing a component's contract in isolation. Another possible enhancement is to support time-ordered event specifications, similar to [11].

Intercomponent contracts could be supported in several ways. First, it is already possible for one component to make assertions about another one, if the latter component is a bean property of the former or invocations of static (class) methods are sufficient. The test expression can simply call methods on the other component and assert conditions on the results. Without an explicit property connection, it would be necessary for Contract4J to support some sort of lookup scheme for locating instances. The bean management facilities of a framework like Spring could be leveraged, for example.

Annotations could be added to Contract4J to support temporal constructs [11]. Constraints on the order of invocation of API calls are perhaps the most common usage scenario for intercomponent contracts. However, temporal annotations could also be used to drive events in a state machine, notify observers, *etc*., which would extend these annotations beyond the contract role to more general purposes.

An alternative approach is to bypass Contract4J and to write *ad hoc* aspects that test the component relationships. The AspectJ literature is full of practical examples. This approach gives the developer maximum flexibility, but it requires AspectJ expertise and the developer doesn't get the other advantages of Contract4J, discussed previously and elsewhere [9].

The annotation approach has its limits. It is certainly not a "complete" conception of aspect-aware interfaces. It provides a convenient and terse mechanism for expressing information, which can be used to drive nontrivial aspects behind the scenes. However, it has limited expressive power and "hard-coding" an annotation in source code undermines the separation of concerns advantage of aspects, if not used judiciously.

5. CONCLUSIONS

Contract4J defines a design pattern-like protocol with two different syntax forms for specifying the contract of a component. The protocol is essentially a non-conventional "interface" that could be used by other tools as well, such as code analysis tools and IDEs. Like good interfaces, the protocol minimizes coupling between aspects and components. For some situations, this approach provides a terse, yet intuitive and reasonably-expressive way to specify metainformation about the component that can be exploited by aspects behind the scenes to support nontrivial component interactions. However, this approach has its limits and it does not replace a more complete concept of aspect interfaces.

Consistent with the work on crosscutting programming interfaces (XPIs) [2], Contract4J shows that aspect-aware interfaces, whatever their form, should contain more precise contract specifications for proper use and not just lists of methods and state information. Otherwise, the risk of breaking either the aspects or the components is great, especially as they evolve. However, programmatic enforcement of the interfaces is also essential. Fortunately, aspects also make such enforcement possible in ways that reduce the *ad hoc* nature of most pattern implementations seen today. Taken together, these facts suggest that a fruitful direction for AOP research is to explore how aspects can elevate *ad hoc* patterns of collaboration to programmatic constructs.

6. ACKNOWLEDGMENTS

My thanks to Eric Bodden, Kevin Sullivan, and Ron Bodkin for stimulating discussions. The comments of the anonymous reviewers were very helpful in clarifying the ideas in this paper.

7. REFERENCES

[1] 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.

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

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

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

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

[6] http://en.wikipedia.org/wiki/Test_driven_development

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

[8] http://java.sun.com/products/javabeans/index.jsp

[9] D. Wampler, "The Challenges of Writing Portable and Reusable Aspects in AspectJ: Lessons from *Contract4J*", Industry Track, AOSD 2006, *forthcoming.*

[10] J. Hannemann and G. Kiczales, Design Pattern Implementation in Java and AspectJ. *Proc. OOPSLA '02,* ACM Press, 2002, pp. 161-173.

[11] http://www.bodden.de/studies/publications/DiplomaThesis/body_diplomathesis.php

[12] JSR-220: http://java.sun.com/products/ejb/docs.html

[13] I. Jacobson and Pan-Wei Ng, *Aspect-Oriented Software Development with Use Cases,* Addison-Wesley, Upper Saddle River, NJ, 2005.