Instrumenting Contracts with Aspect-Oriented Programming to Increase Observability and Support Debugging

Lionel C. Briand [§¶] W. J. Dzidek [¶] Yvan Labiche [§]
§ *Software Quality Engineering Laboratory, Systems and Computer Eng. Dept., Carleton University, Ottawa, ON, K1S 5B6, Canada, +1 613 520 2600 ext. {2471,5583}, {briand, labiche}@sce.carleton.ca*
¶ *Simula Research Laboratory, Martin Linges v 17, Fornebu, P.O. Box 134, 1325 Lysaker, Norway, +47 67 82 82 01, jamesdz@simula.no*

Abstract

In this paper we report on how Aspect-Oriented Programming (AOP), using AspectJ, can be employed to automatically and efficiently instrument contracts and invariants in Java. The paper focuses on the templates to instrument preconditions, postconditions, and class invariants, and the necessary instrumentation for compliance-checking to the Liskov Substitution Principle.

1. Introduction

Analysis and design by contract (DbC) allows the definition of a formal agreement between a class and its clients, expressing each party's rights and obligations. Operation contracts and class invariants are known to be a useful technique to specify the pre- and postcondition of operations and the legal states of class instances in an object-oriented (OO) context, making the definition of OO analysis or design elements more precise [8]. Furthermore, it is also useful to check such contracts and invariants at run time in order to help testing and debugging during corrective maintenance [11]. Indeed, experiments report a substantial gain when relying on instrumented contracts during those two activities [3]. However, the instrumentation of such contracts is a time consuming activity as this is usually performed manually. Our work focused on the automation of this instrumentation process with two main objectives: (1) to work at the bytecode level so that constraint-related code (assertions) and the program's source-code are kept separate (avoid polluting the source code and facilitate configuration management and maintenance); and (2) to propose an instrumentation strategy that is suited in a context where checking that inheritance hierarchies conform to the Liskov Substitution Principle (LSP) is required and exception handling mechanisms are used.

Note that throughout the article, three related, but distinct, words are used: *contract* (description of the services that are provided by an operation using pre- and postconditions), *constraint* (a pre-/postcondition, or an invariant), and *assertion* (the implementation language, e.g. Java, translation of a constraint that has to be instrumented). Furthermore, we assume that the reader is familiar with the basic concepts of Aspect-Oriented Programming (AOP) [4] and the following terminology: aspect, join point, pointcut, and advice. The rest of the article is structured as follows. Section 2 discusses related work. The strategy we follow to instrument constraints, accounting for inheritance, is described in Section 3. The AspectJ templates used for the instrumentation of contracts are detailed in Section 4. An example is given in Section 5. Finally, conclusions are drawn in Section 6.

2. Related Work

There exist two main strategies for automatic instrumentation of contracts in Java: source-code and bytecode manipulation. There exist nine DbC tools for Java (that the authors are aware of); six of these are compared in [9] (iContract, Jass, jContract, jContractor, JML, Handshake); [5] discusses two more: JMSAssert and Kopi; finally, there is the Dresden OCL Toolkit [13].

In order to compare these approaches we identified seven criteria, namely: (a) whether the approach is based on bytecode or source code manipulation (possibly with coding conventions) or on an extension of the Java Virtual Machine (JVM); (b) whether it supports the LSP [7]; (c) whether it supports separate compilation (i.e., allowing modifications of the application source code without recompiling assertion code or vice-versa); (d) whether contract checking in the presence of exceptions is supported; (e) the ability for assertion code to use private members; (f) the option to use either compile-time or load-time instrumentation (with load-time instrumentation constraint checking code can be installed or removed without requiring recompilation); and (g) the ability to add assertions to classes for which the source-code is not available. Our aim, in this paper, is a solution that provides the best alternative for all seven criteria: instrumenting byte code, checking the LSP, separate compilation, proper handling of exceptions, access to

Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)
1063-6773/05 $20.00 © 2005 **IEEE**

private members, instrumentation flexibility, and no source code needed. None of the nine surveyed tools were able to provide all these functionalities.

The idea of using AOP as the instrumentation technology for constraints checking is not new, yet the topic has never been given a thorough analysis. The AspectJ manual [1] gives a small example of code to check preconditions. In [10] AspectJ is used to check invariants. In [6] the authors discuss the topic of checking preconditions and postconditions using AOP, without offering a complete solution (e.g., the authors do not take into account inheritance hierarchies).

Note that this related work section only focused on instrumentation. There also exist approaches for the automatic translation of contracts expressed in high level languages (such as OCL) to implementation languages (such as Java). The interested reader is referred to [2].

3. Constraint Checking

Instrumenting a constraint requires that we identify where the corresponding assertion needs to be checked, the *insertion point*. The insertion points are summarized in Table 1 (which is adapted from [5]). For example, the insertion point for an assertion checking a precondition is right before the execution of the corresponding method. Table 1 also shows what is checked when an exception is thrown during the execution of a constrained method.

Table 1. Constraint Checking

		public (UML)	not public (UML)	constructor
pre	entry	X	X	X
post	regular exit	X	X	X
	exception			
inv	entry	X		N/A
	regular exit	X		X
	exception	X		

Furthermore, LSP [7] provides a theoretical framework for the definition of constraints in inheritance hierarchies, distinguishing subtyping from subclassing. Meyer probably captured LSP best with his contract-oriented paraphrase that "a subtype must require no more and promise no less than its supertype." In instrumentation terms, this means that ancestor classes' invariants must be checked for descendent classes (to check at run time whether the implementation, and not only the model, complies with the LSP). For similar reasons, when a method overrides another, both postconditions must be checked at the end of the execution of the overriding method. However, since the precondition of the overriding method does not imply that of the overridden method (that is exactly the contrary), only the overriding method's precondition is checked at the beginning of its execution.

Some authors promote the use of the LSP as it results in a safe use of inheritance [12]. It is, however, a reality that the LSP does not always hold in inheritance hierarchies. Ideally it should be allowed to specify inheritance hierarchies where the LSP does or does not hold so that constraint inheritance is only enforced where it makes sense to do so.

4. Aspects Checking Constraints

In this section, we present our aspect templates. The AspectJ code specifying the assertions and insertion points for several constraints can reside in one (aspect) file, although each class in the instrumented system can have its own file. The latter solution is more efficient from a compilation perspective as only one small aspect file has to be recompiled when a constraint changes.

The aspects used for constraint checking are `privileged` aspects (as shown below), meaning that the code in the aspect has access to any class' private/protected attributes and methods. This is necessary as constraint checking may require such access.

```
privileged aspect aClassConstraints {
    … // Advice code }
```

The templates for checking preconditions, invariants and postconditions are described in Sections 4.1 to 4.3, respectively. For each template, bold face text shows what is variable (e.g., parameter names) and square brackets denote optional parts. A complete aspect example is presented in Section 5.

4.1 Checking Preconditions

The advice code template for checking preconditions for a non-static method is shown in Figure 1.

```
before(aClass self [, method parameters]):
  execution(method_return_type
      aClass.aMethod([method parameter types]))
 && target(self) [&& args(parameter names)]
 && within(aClass) {… //Check the precondition.}
```

Figure 1. Precondition template

A `before` advice executes before the specified pointcut executes (i.e., the constrained method). The `before` keyword exposes variable names (with types) that can be used in the advice code: `self` of type `aClass`, and any method parameter (name and type) that the advice should use. (These will be used in the pointcut.) In the pointcut: **`execution(…)`** specifies, using a method signature, that any execution of method `aMethod` on any instance of class `aClass` is intercepted; **`target(self)`** maps `self` (defined in `before(…)`) to the object executing the intercepted method (on which the constraint is being evaluated). In the advice code, variable `self` will then be a reference to the object executing the intercepted method execution; **`args(…)`** maps names appearing in parenthesis (and defined in

before(…[params])) to the parameters of aMethod so that its arguments can be referred to in the advice code; **within(aClass)** specifies that the version of the executing method must be declared in class aClass. This is to prevent the interception of aMethod's execution on subclasses of aClass that override aMethod (and thus likely have a precondition different from the one of aMethod in class aClass).

The AspectJ code template for checking the precondition of a constructor is very similar. The only difference is the execution(…) part that reads:

```
execution(aClass.new([parameter types]))
```

4.2 Checking Invariants

The AspectJ code template for checking invariants is shown in Figure 2 as several code fragments.

The code fragment (1), located in the aspect itself, adds the method invariant() to class aClass (AspectJ allows us to add methods to an existing class). It checks aClass's invariant (including the parent's invariant if any – call super.invariant()) and is invoked in the advice bodies of fragments (2-4).

Placing the invariant() method inside the context class is an elegant solution to the check of invariants in an inheritance hierarchy, that relies on polymorphism and dynamic binding. Recall that when a child class invariant is checked, its parent class invariant must be checked as well. This is achieved by calling super.invariant() in method invariant(). The call to super.invariant() is optional since (a) the superclass may not have an invariant, and (b) constraint inheritance may not be desired (Section 3).

Next, fragment (2) ensures that the invariant is checked before the execution of any public method on any instance of the context class: aClass.*(..) specifies any method (*) with any parameter list (..) in class aClass. Likewise, fragment (3) ensures that the invariant is checked after those methods executions. Finally, fragment (4) ensures that the invariant is checked after the execution of all constructors. (Note that if a class implements Cloneable, an aspect intercepting calls to clone() is necessary since this method creates an instance of the class without invoking a constructor.)

!within(aClassConstraints) is used to ensure that the execution of invariant() is not intercepted by the aspect that triggered it (recall from the beginning of Section 4 that aClassConstraints contains all aspects related to a class). We thus avoid infinite recursion that would result in the aspect trying to check the invariant before and after the execution of invariant().

Keyword returning in fragment (4) specifies that the after advice only executes when the intercepted method execution completes successfully, i.e., no exception is thrown. This way, the invariant is only checked on successful termination of the intercepted constructor execution, as discussed in Section 3. The after advice used in fragment (3) is not affected by an abnormal termination of the intercepted execution. This results in checking the invariant even after an exception is raised during the execution of a public method (Section 3).

1	`void aClass.invariant() {`
	`  [super.invariant();]`
	`  …// Check the invariant. }`

2	`before(aClass self) : execution(public *`
	`aClass.*(..)) && target(self) &&`
	`within(aClass) && within(aClassConstraints)`
	`{ self.invariant(); }`

3	`after(aClass self) : execution(public *`
	`aClass.*(..)) && target(self) &&`
	`within(aClass) && within(aClassConstraints)`
	`{ self.invariant(); }`

4	`after(aClass self) returning :`
	`execution(aClass.new(..)) && target(self)`
	`&& within(aClass) &&`
	`within(aClassConstraints)`
	`{ self.invariant(); }`

Figure 2. Invariant template

Finally, optional enforcement of the LSP is controlled by the optional call to the super class invariant() method: no call leads to the LSP not being enforced.

4.3 Checking Postconditions

Figure 3 shows the postconditions checking template. The around advice is used to: intercept a method, perform some activity, and continue with the execution. This advice lets us gain access to old data (to support OCL's @pre) and the method's result (to support OCL's result) in the postcondition assertion.

Note that in the case the intercepted method throws an exception the postcondition is not verified, and adding the statement within(aClass) to the pointcut amounts to not enforcing constraint inheritance.

```
method_return_type around(aClass self [, method
parameters]) : execution( method_return_type
aClass.aMethod([method parameter types]))
&& target(self) && args(parameter names) {
  … // Create any necessary @pre variables
  // Necessary if the OCL keyword 'result'
  // is used in the postcondition.
  [method_return_type result;]
  // Let the execution of the method proceed.
  [result =] proceed(self [, method params]);
  … // Check the postcondition. }
```

Figure 3. Postcondition template

The following is the template checking a postcondition on a constructor. The keyword returning ensures our compliance with the discussion in Section 3:

```
after(aClass self [, constructor parameters])
returning : execution(public
aClass.new([constructor parameter types]))
   [&& args(parameter names)] && target(self)
   { … // Check the postcondition. }
```

5. Example

Our example (Figure 4) consists of `Person` (no parent class) with attributes `age`, `salary`, and `maxSalary`, all of type `Integer`. Additionally, class `Person` has an operation called `implementRaise(raise:Integer)` that raises the person's salary. The following is the class invariant for `Person` and the precondition and postcondition for the `raise():int` operation:

- `context Person inv: self.age >= 18`
- `context Person::raise(raise:Integer)`
  `pre:self.salary + raise <= self.maxSalary`
  `post:self.salary= self.salary@pre + raise`

Also, a modest case study was used to perform a first feasibility analysis and to evaluate the instrumentation overhead, it yielded promising results. The interested reader is referred to [2].

6. Conclusions

In this paper we present Aspect-Oriented Programming (AOP) AspectJ templates for automatic and efficient instrumentation of contracts and invariants in Java. Our main motivation, based on past studies [3], is that checking constraint assertions at run-time is extremely valuable during testing to detect failures and during maintenance to help locating faults (debugging).

Our instrumentation strategy consists in manipulating the bytecode (and does not require coding conventions) instead of the source code (no source code pollution). As a consequence, the user can work on the source code without having to regenerate the constraint assertions before each compile, resulting in large time savings. Furthermore, the strategy addresses: contract checking in the presence of exceptions, the ability for assertion code to use private members, the option to use either compile-time or load-time instrumentation (on the fly), the ability to add assertions to classes for which the source-code is not available, and the option to enforce the checking of the Liskov Substitution Principle in inheritance hierarchies.

References

[1] AspectJ-Team, The AspectJ Programming Guide, www.eclipse.org/aspectj/, (Last accessed March 2005)

[2] L. C. Briand, W. Dzidek and Y. Labiche, "Using Aspect-Oriented Programming to Instrument OCL Contracts in Java," Carleton University, Technical Report SCE-04-03, www.sce.carleton.ca/squall, 2004.

[3] L. C. Briand, Y. Labiche and H. Sun, "Investigating the Use of Analysis Contracts to Improve the Testability of Object-Oriented Code," *Software - Practice and Experience*, vol. 33 (7), pp. 637-672, 2003.

[4] T. Elrad, R. E. Filman and A. Bader, "Aspect-Oriented Programming: Introduction," *Communications of the ACM*, vol. 44 (10), pp. 29-32, 2001.

[5] M. Lackner, A. Krall and F. Puntigam, "Supporting Design by Contract in Java," *Journal Of Object Technology*, vol. 1 (3), 2002.

[6] M. Lippert and C. V. Lopes, "A Study on Exception Detection and Handling Using Aspect-Oriented Programming," *Proc. International Conference on Software Engineering*, pp. 418-427, 2000.

[7] B. H. Liskov and J. M. Wing, "A Behavioral Notion of Subtyping," *ACM Transactions on Programming Languages and Systems*, vol. 16 (6), pp. 1811-1841, 1994.

[8] B. Meyer, *Object-Oriented Software Construction*, Prentice Hall, 2nd Edition, 1997.

[9] R. Plösch, "Evaluation of Assertion Support for the Java Programming Language," *Journal of Object Technology*, vol. 1 (3), 2002.

[10] R. Van Der Straeten and M. Casanova, "Stirred but not Shaken: Applying Constraints in Object-Oriented Systems," *Proc. NetObjectDays*, pp. 138-150, 2001.

[11] J. M. Voas and L. Kassab, "Using Assertions to Make Untestable Software More Testable," *Software Quality Professional*, vol. 1 (4), pp. 31-40, 1999.

[12] J. Warmer and A. Kleppe, *The Object Constraint Language*, Addison-Wesley, 1999.

[13] R. Wiebicke, *Utility Support for Checking OCL Business Rules in Java Programs*, Diploma Thesis, Dresden University of Technology, 2000


```
privileged aspect PersonConstraints {
static void constraintFailed(String constraint) {
  … // Logic to notify of broken constraint. }
// (Fragment A) (instance of Figure 2, fragment 1)
void Person.invariant() { if
(!(self.age.intValue()>=18)){
  constraintFailed("self.age >= 18"); } }
// (Fragment B) (instance of Figure 2, fragment 2)
before(Person self): execution(public * Person.*(..)) &&
  target(self) && within(Person) &&
  !within(PersonConstraints) { self.invariant(); }
// (Fragment C) (instance of Figure 2, fragment 3)
after(Person self): execution(public * Person.*(..)) &&
  target(self) && within(Person) &&
  !within(PersonConstraints) { self.invariant(); }
// (Fragment D) (instance of Figure 2, fragment 4)
after(Person self) returning: execution(Person.new(..))
  && target(self) && within(Person) &&
  !within(PersonConstraints) { self.invariant(); }
```

```
// (Fragment E) (instance of Figure 1)
before(Person self, int raise) : execution(void
  Person.raise(int)) && target(self)
  && args(raise) && within(Person) {
  if (!(self.salary.intValue() + raise <=
  self.maxSalary.intValue())) {
      constraintFailed("self.salary + raise" +
                  " <= self.maxSalary"); } }
// (Fragment F) (instance of Figure 3)
void around(Person self, int raise): execution(public
  void Person.raise(int)) && target(self) && args(raise){
    int oldSalary = self.salary.intValue(); // Old
    salary
    proceed(self, raise); // Continue executing.
    // Check the postcondition.
   if (!(self.salary.intValue() == (oldSalary + raise)))
    {constraintFailed("self.salary = self.salary@pre +
    raise"); } }
}
```

Figure 4. Complete AspectJ code on an example.