An Aspect for Design by Contract in Java

Sérgio Agostinho[1], Pedro Guerreiro[2] and Hugo Taborda[1]

[1]Universidade Nova de Lisboa 2829-516 Caparica, Portugal

[2]Universidade do Algarve 8005-139 Faro, Portugal

Abstract. Several techniques exist for introducing Design by Contract in languages providing no direct support for it, such as Java. One such technique uses aspects that introduce preconditions and postconditions by means of before and after advices. For using this, programmers must be knowledgeable of the aspect language, even if they would rather concentrate on Design by Contract alone. On the other hand, we can use aspects to weave in preconditions, preconditions and invariants that have been programmed in the source language, as regular Boolean functions. In doing this, we must find ways to automatically "inherit" preconditions and postconditions when redefining functions in subclasses and we must be able to record the initial state of the object when calling a modifier, so that it can be observed in the postconditions. With such a system, during development, the program will be compiled together with the aspects providing the Design by Contract facilities, using the compiler for the aspect language, and the running program will automatically check all the weaved in assertions, raising an exception when they evaluate to false. For the release build, it suffices to compile using the source language compiler, ignoring the aspects, and the assertions will be left out.

1 Introduction

Design by Contract (DbC) [18] is a well-established methodology, which aims to guide the software development from analysis to implementation. It relies on systematic use of assertions that record pre/postconditions associated to each method. Not many languages provide direct support for DbC, however. Remarkable exceptions are Eiffel [17], the language on which DbC was invented, and most recently the D language. If one wants to follow the discipline of DbC with another language, some kind of compromise that involves the use of an add-on must be accepted.

The original specification of Java featured DbC, but it did not prevail [8]. Since 2001, DbC support is in the top Request for Enhancements at the Sun Developer Network. As such, DbC seems to be relatively well accepted and several tools exist that adjust Java for it, typically using generative programming techniques.

Another approach is to use aspects that introduce preconditions and postconditions using aspect-oriented programming, by means of `before` and `after` advices [2] [14]. That is, the assertions are written in an aspect and weaved in the executable program, firing exceptions when the contract is violated. This is interesting but somewhat defeats the core idea that DbC is also an important part of the documenta-

Agostinho S., Guerreiro P. and Taborda H. (2008).
An Aspect for Design by Contract in Java.
In *Proceedings of the 6th International Workshop on Modelling, Simulation, Verification and Validation of Enterprise Information Systems*, pages 119-128
DOI: 10.5220/0001743001190128

tion of each module, and, as such, should be part of the module itself [18]. Indeed, we agree that contracts are not crosscutting concerns [3].

We pursue a different way of using aspects for DbC, which more closely resembles the standard approach found in Eiffel: for each method we explicitly program, in Java, two Boolean methods, one for the precondition and another for the postcondition. In order to keep things simple, the names of these methods follow strict rules and they have the same arguments as the original method or an initial sub-list thereof. As such, in the Java program, these methods can indeed be understood as documentation, and usually they are not called in the program itself. Then, when we compile the class with the DbC aspect, the Boolean methods are weaved in automatically and are evaluated before and after each method call.

This paper has six sections. In section 2, we discuss the features of our system, including assertions and contract polymorphism. In section 3, we evaluate our results, through the use of third-party metrics. In section 4, we discuss performance. In section 5, we present related work. We wrap up, on section 6, with the main conclusions on this experience.

This work was partly supported by the Portuguese *Fundação para a Ciência e Tecnologia*, in the context of the SOFTAS project (POSI/EIA/60189/2004). The authors would like to thank Dr Ana Moreira for her support in this work.

2 Features

DbC advocates software reliability through the use of assertions [11], namely preconditions, postconditions and invariants. Method preconditions are assertions that must hold when the execution of the method starts. Method postconditions are assertions that must hold immediately after the execution of the method ends. Class invariants are assertions that must be held during the lifetime of any object of the class, between method calls. A precondition failure is a fault in the *client*, i.e., in the method that issued the call, while a postcondition failure is a fault in the *supplier*, i.e., in the failing method itself. Invariants must be upheld by the supplier after the constructor execution.

A classic analogy of DbC is a business contract. In a business, there are two interested parties: the client and the supplier. In order to be able to provide a service properly, the supplier expects that some conditions are met: the preconditions (an obligation to the client, but a benefit to the supplier). Conversely, at the end of the service, the client expects that some results are achieved: the postconditions (an obligation to the supplier, but a benefit to the client). Additionally, the contract can have clauses that specify conditions that both parties must respect throughout the duration of the service: the invariants.

2.1 Preconditions, Postconditions and Invariants

In our system, preconditions are Boolean methods with the same name as the original method, but prefixed with `pre`, using "lower camel case". Postconditions are also

Boolean methods with the same name as the original method, but prefixed with `post`. If the method returns a value, an extra first argument is used, representing the result of the method, to be used in the postcondition:

```
public boolean preFactorial(int n) { return n >= 0; }
public long factorial(int n) { /* ... */ }
public boolean postFactorial(long result, int n)
          { if (n == 0) return (result == 1);
          else return (result >= 1); }
```

Each class may have a Boolean invariant method, named `invariant`, representing the invariant of the class. Invariants are checked before and after any public methods are called and after any of the constructors are called:

```
public class Simple2DCoordinates {
          private int latitude, longitude;  // degrees
          public boolean invariant() {
            return latitude >= -90 && latitude <= 90 &&
            longitude >= -180 && longitude <= 180; } }
```

Under the hood, AspectJ [2] advices are executed before and after every non-private method execution or class constructor invocation. `before` advices check invariant first and then precondition; `after` advices check invariant first and then postcondition. We use three types of pointcuts to implement contract assertions: constructor executions, static method executions, and non-static method executions. Each pointcut has two advices: `before` and `after` return. Exceptional terminations of a method or constructor are not evaluated. We need those three types of pointcuts because in some situations class invariants are not checked. For non-static methods, invariants are evaluated before preconditions and after postconditions. Before constructor execution the invariant is not yet set, so it can not be evaluated. Invariants do not apply to static methods.

Assertions that fail raise an exception, with type according to assertion type (precondition, postcondition, or invariant). These exceptions extend Java's `RuntimeException`, so that they do not need to be handled through a `try`/`catch` block.

2.2 The "old" Construct

In postconditions, it is sometimes necessary to compare the final state of the object, when the method returns, with its state when it started. This is known as the "old" mechanism. Languages with native DbC support implement this via a special `old` expression, but in our approach that is not feasible. For imitating the old mechanism, we introduce a class `ContractMemory`, with a method `old()`, which returns a copy of the object before the execution of the method where it appears. Since storing a copy of the object before each method execution could create an unacceptable overhead, we let the programmer explicitly make that request in the precondition, using the `remember()` method from class `ContractMemory`.

Sometimes we need to introduce artificial preconditions, only to call remember, so that old can be used in the postcondition. This is a compromise for simplicity that we choose to accept. The second issue is the explicit reference to a particular class: `Con-`

tractMemory. This creates coupling between the application and the library, but there is a workaround that lets us avoid it (see section 2.5).

Finally, there is the cloning issue. Classes using the "old" construct must implement the `Cloneable` interface, and therefore must supply the `clone()` method. Here we meet a well-know Java complication. The `Object` class (which every class extends) does implement the `clone()` method, but as `protected` visibility. As such, one cannot "force" an object cloning without its correspondent class implementing the actual method. This is illustrated as follows:

```
public class Point2D implements Cloneable {
          private int x, y;
public void incX() { /* … */ }
public boolean preIncX() {
            ContractMemory.remember();
            return true; }
public boolean postIncX() {
          Point2D old = (Point2D) ContractMemory.old();
          return this.getX() == old.getX() + 1; } }
```

Having this in consideration, and the fact that in many occasions the old construct is used for accessing a primitive type, we have added an alternative. The `ContractMemory` class supplies another pair of methods: the `observe()` and `attribute()` [10]. These methods do the same as `remember()` and `old()`, but store primitive types identified by a "tag" string. This alternative is less expensive, and allows the arbitrary storage of variables. With this, the `Point2D` example could be presented as follows:

```
public class Point2D {
          private int x, y;
public boolean preIncX() {
          ContractMemory.observe("Point2D.x", getX());   re-
turn true; }
public void incX() { /* … */ }
public boolean postIncX() {
          int oldX = (Integer)
          ContractMemory.attribute("Point2D.x");
          return this.getX() == oldX + 1; } }
```

The `ContractMemory` class has only empty stub methods. The actual behavior is inserted by an aspect. The aspect stores data in two stacks of hash tables, one for cloned objects and the other for primitive types. The hash tables provide a constant search time and the stacks support context for method calls and recursion.

2.3 Contract Polymorphism

How does DbC cope with class inheritance? The Liskov substitution principle [16] (also known as the "behavioral subtyping principle") states that if class B is a subtype of class A, then, in any situation that an instance of A can be used, it can be replaced by an instance of B. Consequently, the precondition of a method in B can only be the same or weaker than the corresponding precondition in A, and cannot be stronger. Conversely, the postcondition of a method in B can only be the same or stronger than the corresponding one in A, and cannot be weaker. Invariants from A must be res-

pected in B. Following our analogy with business, inheritance can be seen as subcontracting. If the original supplier subcontracts the service, then the subcontractor cannot impose further restrictions to the contract and cannot offer fewer benefits than the original supplier.

In our system, the implementation of the substitution principle is achieved by composing preconditions with logical disjunction; and by composing postconditions with logical conjunction. The invariant preservation is implemented in the same way as postconditions. This means that actual pre/postconditions are compositions of the current and inherited pre/postconditions. However, the Eiffel standard [6] recently relaxed the postcondition evaluation rule: it is not necessary that all partial postconditions are fulfilled, only the ones correspondent to the partial preconditions evaluated as true. Recent work by Finder *et al* [7] claims that disjunction/conjunction composition rules are necessary, but insufficient, since they do not prevent programmers from writing covariant preconditions or contravariant postconditions. As such, they propose additional rules, namely that an evaluation with true value in a method precondition must imply a true value in the subclasses overridden preconditions (for postconditions, the rule is inversed).While we adhere to the Eiffel algorithm, we currently do not support Findler's proposal on our prototype.

Assertions are discovered through reflection. Starting with the object (static) type, assertions are searched and evaluated bottom-up until a direct subclass of `Object`. For each class, the algorithm looks for a method with the given prefix with no arguments, and adds an argument to the signature until it gets a match. This algorithm may find the wrong assertion if method (and assertion) overloading is used, with ambiguous assertion signatures.

Preconditions are evaluated bottom-up and postconditions are evaluated top-down. In order to ensure the correctness of the side-effect `ContractMemory` calls, every partial pre/postcondition must be evaluated. Class invariants are lazily checked bottom-up, since no further restrictions apply to this type of assertions.

Since assertions represent a contract, they should be `public`. This poses a problem for evaluating inherited contracts through reflection, because public methods are always evaluated with respect to their runtime type [9]. Thus, if both class A and B have a public `preFoo()` method, given an instance of B, it is not possible to execute the `A.preFoo()`. The exceptions to this rule are the `static` and `private` methods [15]. Hence, we must resort to `private` methods and use the `AccessibleObject.setAccessible(boolean)` method [23], which breaks Java's visibility protection. Although AspectJ's `privileged` keyword allows an aspect to access private members of a class, it does not work with reflection. As an alternative to private methods, our system also supports public contracts in the presence of inheritance, for classes that have a *copy constructor*.

2.4 Contract Rules

Some authors argue that contract validation should not be disabled in production builds, or at least not completely [18] [19]. In conformity, our library supplies a mechanism which we call contract rules, allowing us to define which assertions should be checked and in which classes. This can be used in two ways: statically or at run-

time. Static rules are written in a properties file, in the LACE format [17] [18]. With this file it is possible to specify the default assertion check level and the specific classes in which specific check levels are to be used. These levels are the following: `no`: no checks; `pre`: only preconditions; `post`: both preconditions and postconditions; `all`: all assertions are checked.

The file is loaded at the library start-up, and if the file is not found, a conservative configuration is assumed (all default checks, and no specific classes). There are also runtime rules available, through an API with which we can modify the start-up configuration. In some situations it is useful to temporarily disable assertions within a public method, in order to execute a few simple calls that momentarily break the contract. The "expose" mechanism is used for that purpose. It is supported by the Rules class via two methods: `expose()` and `unexpose()`. The `ContractRules` class is similar to `ContractMemory`. It supplies only one method, for retrieving the `Rules` *singleton* object. The `Rules` object is loaded from a file (or otherwise initialized with default values) at startup, and stored by the main aspect. Inside that object, two hash tables exist for storing the rules for specific classes, as well as the exposed classes, respectively.

2.5 Unplugging

Our current prototype implementation has three aspects. The core aspect is responsible for most of the work, including checking the contracts and loading the configuration rules. The other two introduce the actual behavior of `ContractMemory` and `ContractRules` classes, which are simply "mock" classes. Therefore, if the programmer is using core features only, unplugging is simply removing the library from the project, and recompiling as a regular Java project. The same is true if the contracts are specified in aspects, using AspectJ's *intertype declarations* which allow adding new methods to classes or interfaces. If the project has references to the `ContractMemory` and `ContractRules`, or if the programmer still wants to continue using AspectJ, a different solution is required. Since removing all the contracts manually is not very practical, we supply a stub version of our library that contains only a subset of the public classes used. This way, plugging the library consists only in replacing the stub version with the full version.

3 Evaluation Metrics

Plösh [21] proposed a set of metrics to evaluate assertion support in Java. In order to provide a third-party evaluation, we shall submit our solution to this criteria proposal. These are the evaluation results:

- Basic assertions: Since Java 1.4 and later support the assertion facility, we do not feel it is necessary to introduce a specific mechanism for such feature.
- Preconditions and Postconditions: In this area, we fully support the assertions, but do not guarantee side-effect free assertions.
- Invariants: Same as above.

- Enhanced assertion expressions: We support access to the original state of the object or arbitrary (primitive type) variables, but not arbitrary expressions. We also support access to the parameter values in postconditions.
- Operations on collections: No support for first-order logic other than the trivial implication operator.
- Additional expressions: We support the "expose" mechanism, as well as a run-time API for configurability.
- Interfaces: It is possible to add contracts to interfaces, but only through the inter-type declaration feature of AspectJ. This means it does not follow the subtyping principle.
- Correctness I: We impose precondition weakening and postcondition strengthening in subcontracts.
- Correctness II: The rules for behavioral subtyping are the same as specified in the Eiffel standard.
- Contract violations: We use runtime exceptions. While there are no monitoring features (we believe it is up to the programmer to explicitly decide what to do in these situations), it would be straightforward to implement this.
- Configurability: Assertion enabling/disabling is supported, by assertion type and class. It would be interesting to support package level granularity, but method granularity seems "overkill".
- Efficiency: There is a significant overhead of memory and (specially) processing usage. This is discussed in detail in Section 4.

4 Performance

The experiment subject is a small text-mode Java application, which reads a comma-separated values text file, and inserts a pair of values into a hash table. The application is in two files: "Dictionary.java" (the supplier class) and "Main.java" (the client class). Three versions of the program were written: the first version is an optimistic implementation, in which contracts are only comments ("foobar"); in the second version contracts are in-lined assertions using Java's 1.4 Assert Facility ("foobar-jaf"); the third version writes the contracts using our prototype – DbC4J ("foobar-dbc"). While the second solution may seem pointless, we brought it to our comparison because it corresponds to what could be obtained automatically by using a specification language for contract writing and a code transformation tool. The experiment was performed with J2SE JDK 1.5.0 and AspectJ 1.5.3, under Fedora Core 4 GNU/Linux. 12 runs were made; the best and worst were removed; the resulting time value is the average of the 10 remaining values. The time values are measured using the `currentTimeMillis()` method. The quantitative results of the experiment are summarized in Table 1. The measurement units used are: milliseconds (ms), lines of code (LOC), and bytes (B).

Table 1. Performance experiment quantitative results.

	foobar	foobar-jaf	foobar-dbc
Execution time (ms)	7	7	45082
Source code size (LOC)	177	185	226
Source code size (B)	4396	4759	5406
Bytecode size (B)	4533	4755	23819

From this table, it is possible to observe some interesting facts. First, the DbC4J version performance is about 6000 times slower than the original version. An extra test was performed, running the DbC4J version with all assertions disabled: the execution time was 49 milliseconds, about 7 times slower than the original. This confirms that performance degradation is essentially caused by the infrastructure Java code (mostly reflection) rather than by the AspectJ code. Secondly, execution time in the JAF version is similar to the original version. This shows that the `assert` feature is optimized by the compiler and virtual machine, and poses no real overhead. Finally, the size of the code increased when using contracts. What the table doesn't show is that while in the JAF version the increase is on the client side, in the DbC version it is on the supplier side. This means the DbC version would scale better.

Why is DbC4J performance so low? The largest bottleneck is the use of reflection, a taxing mechanism. Thus, we can improve performance by decreasing the use of reflection, either by algorithm optimizations, stricter pointcuts, or simply by changing the library interface. The second bottleneck is AspectJ's overhead. In spite of continuous performance improvements in AspectJ's releases, there is little that can be done here, other than limiting the scope of the pointcut expressions. Finally, it is worth mentioning that one of the biggest advantages of incorporating DbC in the Java language instead of using an *ad-hoc* solution, is the possibility of using first-order artifacts, which could be used to perform low-level compiler/JVM optimizations (similar to those we observed in the assert facility).

5 Related Work

Behavioral specification languages such as Z [1], and more recently UML's OCL [20] presented a great influence in DbC solutions. Several extensions to abstract syntax tree of Java have been proposed [5], [7]. Extending the language is the most straightforward technique, but makes the code non-valid under a regular Java compiler. As such, several generative programming techniques have been employed.

Using source code preprocessing techniques, such as in iContract [13], contracts are written in source code comments, which are processed by a tool that generates the equivalent Java code to verify such contracts. These solutions need to use a specific language for contracts, which implies an extra effort on the software programmer. Also, preprocessing is a "one way" technique: traceability is discarded, as well as interoperability with development tools, such as debuggers and IDEs.

Bytecode instrumentation is a technique in which contract checking code is inserted at bytecode level [12]. This technique makes possible the writing of contracts in the form of executable code, but it is very intrusive, as it interferes with the Java

compile/runtime behavior, which may change in different releases, platforms or implementations, as well as cause unpredictable side effects.

A more recent technique is to use the compiler annotation plug-in API made available in Java 1.6, such as in ModernJass [22]. This facility enables seamless integration of third-party plug-ins with standard tools, like compilers and IDEs.

Using AOP for implementing DbC is not a novel technique. For instance, Contract4J [24] is an AspectJ solution that, unlike our work, uses Java annotations to specify the contract, instead of Java code. While this makes contracts less wordy and simplifies the introduction of the old and result mechanisms, annotations are metadata, which means they are not validated by the Java compiler, but interpreted by a third-party interpreter.

Specification languages such as the Java Modeling Language (JML) [4] allow richer specifications than executable code, namely by incorporating static checking and theorem-proving techniques. However, JML specifications are not executable code, and as such, the full use of this language requires the adoption of specific tools (compiler, documentation generator, etc.), besides the JML language itself.

6 Conclusions

Ad-hoc mechanisms that enable Design by Contract in languages that do not have native support for the methodology tend to be clumsy and unnatural. Ours is no exception, but at least it does not introduce any additional language complexity. The contracts are written in the source language, the assertions are indeed compiled (they are not mere comments) using the language compiler, and if there was nothing else, they would have some merit as documentation. With the addition of the DbC aspect and the aspect compiler, the contracts are transparently weaved, and enforced in the executable program. Note that the programmer does not have to be knowledgeable of aspect technology: he merely invokes a tool that somehow "switches on" the contracts. From this perspective, the essential idea is similar to the one underlying one of the killer applications for aspects: profiling. In order to profile a program using an aspect, one only has to add the aspect: no more interaction is required.

Design for Contract is a methodology for developing software starting from the specification of the methods by means of their pre and postconditions. Once we achieve that, it is only natural to think of automatic generation of test cases, and, even better, of automatically proving that the contracts cannot be violated during the execution of the program [4]. These are ambitious tasks that are better handled, as DbC itself, with some kind of "native" support. Nevertheless, we cannot exclude that an *ad-hoc* solution, in the vein of the one presented here, can be instrumental in bringing the techniques to wider audiences.

References

1. Abrial, J., Schuman, S., and Meyer, B. A Specification Language. On the Construction of Programs, Cambridge University Press, McNaughten, R., and McKeag, R. (editors), 1980.

2. AspectJ Team, The. The AspectJ Programming Guide. 2003.
3. Balzer, S., Eugster, P., and Meyer, B. Can Aspects Implement Contracts? Proceedings of Rapid Integration of Software Engineering techniques (RISE), Geneva, Switzerland, September 13-15, 2006.
4. Burdy, L., Cheon, Y., Cok, D., et al. An overview of JML tools and applications. International Journal on Software Tools for Technology Transfer, June 2005.
5. Duncan, A., and Hölzle, U. Adding Contracts to Java with Handshake. Technical report TRCS98-32, December 8, 1998.
6. Ecma. Eiffel: Analysis, Design and Programming Language (2nd ed.), June 2006.
7. Findler, R., and Felleisen, M. Contract Soundness for Object-Oriented Languages. Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA), Florida, USA, 2001.
8. First Person Inc. Oak Language Specification. 1994.
9. Gosling, J., Joy, B., Steele, G., and Bracha, G. The Java Language Specification (3rd edition). Prentice-Hall, 2005.
10. Guerreiro, P. Simple Support for Design by Contract in C++, TOOLS USA 2001, Proceedings, pages 24-34, IEEE, 2001.
11. Hoare, C. An Axiomatic Basis for Computer Programming. Communications of the ACM, Vol. 12, No. 10, October 1969.
12. Karaorman, M., and Abercrombie, P. jContractor: Introducing Design-by-Contract to Java Using Reflective Bytecode Instrumentation. Formal Methods in System Design, Vol. 27, No. 3, November, 2005.
13. Kramer, R. iContract - the Java design by contract tool. 26th Technology of Object-Oriented Languages and Systems (TOOLS), California, USA, 1998.
14. Laddad, R. AspectJ in Action: Pratical Aspect-Oriented Programming. Manning, 2003.
15. Lindholm, T., and Yellin, F. The Java Virtual Machine Specification. Prentice-Hall, 1999.
16. Liskov, B., and Wing, J. Family Values: A Behavioral Notion of Subtyping. Technical report MIT/LCS/TR-562b, Carnegie Mellon University, July 16, 1993.
17. Meyer, B. Eiffel: The Language. Prentice-Hall, 1991.
18. Meyer, B. Object-Oriented Software Construction (2nd ed.). Prentice-Hall, 1997.
19. Mitchell, R., and McKim, J. Design by Contract, by Example. Addison-Wesley, 2002.
20. OMG Unified Modeling Language (UML) 2.0 OCL convenience document. 2005.
21. Plösh, R. Evaluation of Assertion Support for the Java Programming Language. Journal of Object Technology, Vol. 1, No. 3, Special Issue: TOOLS USA 2002 proceedings.
22. Rieken, J. Design by Contract for Java - Revised (master thesis), Carl von Ossietzky Universität - Correct System Design Group, April 24th, 2007.
23. Sun Microsystems Java 2 Platform Standard Edition 5.0 API Specification. 2004.
24. Wampler, D. Contract4J for Design by Contract in Java: Designing Pattern-Like Protocols and Aspect Interfaces. Industry Track at AOSD 2006, Bonn, Germany, March 22, 2006.