ezContract: Using Marker Library and Bytecode Instrumentation to Support Design by Contract in Java

Yu Chin Cheng, Chien-Tsun Chen, and Chin-Yun Hsieh
Department of Computer Science and Information Engineering
National Taipei University of Technology
Taipei, Taiwan 106
{yccheng, s1669021, hsieh}@ntut.edu.tw

Abstract

Several approaches have been proposed to support Design by Contract in Java. In this paper, through the use of markers which are predefined dummy methods and attributes, a new approach to annotate contracts is presented. The annotated programs can be directly compiled by standard Java compilers. A bytecode instrumentor is developed to manipulate the bytecode to inject contract evaluation instructions and make the contracts executable at runtime. The marker approach avoids two primary problems found in the existing practices: source compatibility that depends on language extension and symbolic barrier that leaves contracts and their targets unrelated. It also facilitates streamlined integration with IDEs and improves readability as well as writability of the contract-annotated programs.

1. Introduction

Introduced by Bertrand Meyer [15][16], Design by Contract (DBC) has been generally acknowledged as a useful and pragmatic technique to build reliable software systems. Contracts are also regarded as an effective mechanism for debugging because they can help pinpointing the error as early as possible when a program deviates from its normal state. Although Java does not have full built-in support for DBC [17], many attempts have been made. Key design issues of supporting DBC in Java include:

i. How to specify and annotate programs with contracts,

ii. How to translate the program with annotated contracts into executable runtime checking statements, and

iii. How to evaluate and to minimize the runtime overheads of contract checking.

In this paper, we argue that the first design issue primarily determines whether a DBC tool is likely to be more widely used or not. We then present our ongoing research on the development of a DBC tool called ezContract by focusing on the first two issues. Shortly, the proposed approach to annotate contracts with markers is a new one, though the contract translation concept which makes use of low-level bytecode instrumentation or high-level aspect-oriented technologies is relatively well-known (e.g., see [1][6]).

According to our literature review, five contract specification mechanisms are used in existing approaches:

i. Introducing new keywords into the host language such as require and ensure to form syntactic constructs for contracts. The host language is then used as a contract specification language to write assertion statements [3][12].

ii. Annotating contracts in comments. Generally, a new contract specification language which may or may not be similar to the host language is designed to specify contracts [4][11][13].

iii. Using naming conventions or naming patterns to form syntactic constructs for contracts. The host language is used to specify contracts [1][7].

iv. Annotating contracts in separated contract files. The contract specification language could be the host language [1] or a specially designed language [8]. Note that this approach enables specifying contracts even if the source code is not available.

v. Documenting contracts as string literals in language-aware meta-data such as annotations in Java or custom attributes in C#. The documented

1 Published in the 14th Asia-Pacific Software Engineering Conference (APSEC 07), Dec 5-7, 2007, Nagoya, Japan.
string literals usually follow the syntax of the host language, although it is not required [6].

Regardless of the detailed implementations of these approaches, a DBC tool alone is not sufficient to support effective DB-based software development in the real world. Streamlined integration with modern integrated development environments (IDEs) is a critical issue for any DBC tool to be widely adopted. In general, developers have to balance the need to use the customary IDE to ensure productivity and the need to adopt a new DBC tool to develop robust software. If applying a particular DBC tool unjustifiably changes the developers’ development habits or reduces their coding speed, such a tool is unlikely to be widely used. Therefore, from the adoptability aspect, the existing approaches have one of the two major drawbacks that possibly prevent them from becoming widely used practices:

i. **Breaking source compatibility:** Introducing new keywords is an ideal approach to supporting DBC in Java or other languages. By so doing, Java becomes an Eiffel-like language regarding contracts. However, unless such a language extension becomes a standard part of the Java language specification, developers have to use a proprietary compiler and/or IDE to manipulate the source code, which does not seem to be encouraged by most of the software development companies. Even if the proprietary compiler is open source, unless it is constantly and consistently evolved with the Java language specification, programs that use new Java language features cannot be compiled by the proprietary compiler. Such a language evolution problem might become a maintenance nightmare for developers. Moreover, since useful tool supports such as refactoring, testing, and static code analysis only work with standard Java programs, breaking source compatibility denies the contract-annotated programs of such supports.

ii. **Inducing symbolic barrier:** Although the approaches of using comments, naming conventions, contract files, and language-aware meta-data for contract annotation avoid the source compatibility problem, they unavoidably cause the symbolic barrier problem [5]. That is, contracts specified with these mechanisms are not actually correlated to the annotated targets from the compiler’s perspective. As a result, changes of the targets are not automatically propagated or reflected to the contracts, and vice versa. For example, suppose that you make use of modern IDEs such as Eclipse, NetBean, or Visual Studio .NET to develop software. You habitually conduct automatic refactoring to rename a parameter of a method. In this situation, contracts that are annotated in comments and in Java annotations become invalid because the refactoring tool does not have the syntactical information to reflect such a change. Also, renaming a method and a class respectively invalidate contracts documented with naming conventions and documented in separate files. Although it can be argued that such an interoperability problem is irrelevant to DBC on its own, it is indeed becoming increasingly a problem because refactoring is intensively conducted in programming nowadays.

To avoid the two primary drawbacks discussed above, we propose a novel approach to annotate contracts using markers. Specifically, markers are predefined dummy methods and attributes which are used to imitate contract defining keywords, such as require, ensure, old, result, and invariant in Eiffel [15]. The Java assert keyword is used to write assertions in pre-, post-conditions, and class invariants defined by the markers. Once contracts are specified in source code, the program can be compiled by standard Java compilers. Consequently, the contract specifications are embedded in the compiled bytecode. To correctly evaluate the annotated contracts at runtime, the produced bytecode must be further processed by a bytecode instrumentor, which injects contract evaluation statements into the bytecode.

By using the proposed approach, software developers do not have to give up their favorite IDEs to apply DBC. All they have to do is to import the marker library and to invoke the bytecode instrumentor after compilation, perhaps by writing a make file to do so. Refactoring and other existing tools which work for standard Java programs should also work for the annotated programs.

The rest of this paper is organized as follows. Section 2 presents the use of markers to annotate contracts in Java. In Section 3, the design and implementation of a bytecode instrumentor is illustrated. Several DBC tools are surveyed in Section 4 and compared to the proposed approach. Section 5 concludes this paper and indicates possible future research directions.

2. Annotating Contracts
2.1 Contracts for classes

Fig. 1 shows a DBC version of a Stack example using the proposed approach. Contracts, including preconditions, postconditions, and class invariants (lines 19, 23-26, and 33, respectively), are specified by the `assert` statement introduced in Java 1.4. However, the semantics of `assert` is insufficient to represent different types of contracts [17]. Thus, each of these assertions is surrounded by a named block using markers so that these assertions can be bound to a particular type of contract.

Markers are classified into three categories. In the first category, markers are used in pairs to form a block for preconditions or postconditions. For example, in Fig. 1 the `Require.begin()` and the `Require.end()` are used together to form a precondition block (lines 18-20). Postconditions are placed between the `Ensure.begin()` and the `Ensure.end()` (lines 22-27).

Markers in the second category include `Old.value()` and `Result.value` (line 23). The former is used to preserve the value of an expression before entering a method and the latter represents the return value of a method. The third category has only one kind of marker, namely, the `classInvariant()` method (lines 32-34). It is predefined in the interface `IContract` and classes that specify invariants must implement this interface (lines 12).

It should be noted that since contracts are specified by the `assert` statement, writing contracts is just like writing regular Java programs except that they are enclosed with markers. The major advantage of using the Java language as the contract specification language is that the developer can be assisted by modern IDEs with sophisticated coding supports such as incremental compilation, code assist (auto-completion), code navigation, and refactoring. In other words, our approach provides streamlined integration with DBC into development environments.

2.2 Contracts for interfaces

Contracts in Fig. 1 are called inside contracts because they are defined in the same file of the targets they annotate. Eiffel uses this approach to specify contracts. However, this approach does not work for non-static methods in Java interfaces, which can only have declarations but not implementation code. Thus, as illustrated in Fig. 2, to define inside contracts for interfaces, a special inner class `$Contract` (line 22), which is static and final, is declared in the same file of the interface. This inner class must implement the interface to be annotated, in this case, the `IStack` interface. Since a concrete class is defined, we can use the same approach described in Section 2.1 to annotate contracts for the interface.

2.3 Outside contracts

To improve program readability, contracts should be collocated with their annotated targets as much as possible. However, if you cannot access the source code, you cannot define inside contracts, either. In this situation, you can use outside contracts, which are defined in a separate file. Fig. 3 gives an example.

Outside contracts are defined in a class which can specify contracts for interfaces or classes. To annotate itself an outside contract, the class must implement the tag interface `IOutsideContract` so that it can be bound to its annotated target.

2.4 Old expressions and results
Figure 3. An outside contract file example

Eiffel’s old expression provides a helpful mechanism in specifying postconditions. Implementation of the old expression in Java is a difficult problem because it is required to automatically generate a copy of any object whose value ought to be preserved. Common approaches taken by existing Java DBC implementations to support the old expression include: (1) making a deep copy with object cloning \[1\]|4| and (2) making a shallow copy \[6|11|12|13].

At first glance, the first approach seems to be the best choice. However, although Java supports object cloning with the clone() method inherited from the root class Object, to correctly clone objects, classes need to implement the Cloneable interface and provide their own implementation of the clone() method, which indeed involves nontrivial implementation efforts \[5]. To access old values of an object regardless of the whole object or the instance variables of the object that might be primitive types, the implementations of the old expression in both \[1|4] require a class to implement the Cloneable interface, which is arguable overkill and inflexible (Note that a shallow copy is sufficient for preserving old values of primitive objects or immutable objects).

Thus, we take the second approach (i.e., making a shallow copy) to implement the old expression. As shown in Fig. 1, lines 23 and 25, we use the marker method Old.value() to indicate the need to preserve the value passing as an argument of the method. The argument can be an expression and its value is evaluated and preserved when entering the pop() method. Note that it is still possible to make a deep copy by cloning an object and passing it into the Old.value() method.

In postconditions, the return value of a method can be accessed with the marker Result.value, as shown in Fig. 1, line 23. The Result class defines a static instance variable named value whose type is Object so that it can be assigned to a return value of any type. Even if the return value is of primitive types, it can be assigned to the Result.value due to the auto boxing and unboxing mechanism provided in Java 5.0 \[2].

2.5 Marker library

The implementation of markers is quite easy. Fig. 4 shows the Ensure marker class. Because markers are used only to identify a specific block or location in the bytecode, they do not have any particular programming logic. Marker classes should not and cannot be instantiated. Thus, the default constructor is private. Marker methods are static so that they can be directly invoked without an object reference.

Figure 4. Code snippets of Ensure class

3. Bytecode instrumentor

During runtime, a contract-enabled method is executed as follows. First, the class invariant and the precondition are checked. Second, the method body is executed. Third, the postcondition and the class invariant are checked. Lastly, the subcontracting rule \[16] must be checked. That is, preconditions are or-ed, and postconditions and class invariants are and-ed with those of the parent class and outside contract classes, respectively.

Any contract violation throws a runtime exception. Note that nested contract evaluation is not allowed because it causes endless recursive evaluation of contracts \[16]. The bytecode instrumentor (instrumentor for short) must correctly implement the above rules.

The instrumentor performs two types of operations on the bytecode: injection and extraction, which generate new contract checking methods by extracting the annotated contracts from the bytecode and rewrite the original method to make it contract-enabled.

3.1 Injection operations

3.1.1 Injection of original methods

Fig. 5 depicts the mapping between the original annotated source and the instrumented bytecode (for readability, it is represented in source code format).
Note that for simplicity the detailed implementation of the instrumented method, for example, nested contract evaluation checking, is not presented in the figure.

The first block of the instrumented method (the right-hand side of Fig. 5) declares a list for storing and retrieving the values of all old expressions. Then, the list is passed to a method which is used to put each old expression value on the list. This old-value-preserving method is automatically generated by the instrumentor and has the naming pattern `methodName$old(arg1, …, argn, list)`. It contains all old expressions extracted from the original method. Note that all arguments must be passed to it so that the extracted old expressions can access them.

Class invariant and precondition checking is conducted in the second block. Preconditions in the original method have been extracted into a precondition-checking method, which has the naming pattern `methodName$require(arg1, …, argn)`. The implementation of this method is discussed later.

The third block represents the code in the original method body, which has been extracted into `methodName$body(arg1, …, argn)`. It also declares a local variable `$result` to keep the return value of the method body. This value is used in the postconditions and explicitly returned at the end of the instrumented method.

Postcondition and class invariant checking are performed in the fourth block. Similar to the precondition-checking method, postconditions are extracted into `methodName$ensure(arg1, …, argn, $old, $result)`, a postcondition-checking method. Finally, the `$result` is returned in the fifth block.

### 3.1.2 Injection of precondition and postcondition checking methods

The template of the precondition checking method is presented in Fig. 6. Since preconditions are or-ed, before evaluating a class’s preconditions, an `if` expression is used to check preconditions of its parent class, implemented interfaces, and outside contracts. Evaluation of several preconditions defined in different interfaces is possible because a Java class can implement multiple interfaces.

Once a precondition violation is detected, i.e., an `AssertionError` exception is caught, a refined `PreconditionError` exception is thrown to indicate the violation of preconditions.

The implementation of postcondition checking is similar to that of precondition checking and is not discussed further.

```java
public AssertionError methodName$require (arg1, …, argn) {
    AssertionError error = null;
    try {
        if (super.methodName$require(arg1, …, argn) != null ||
            interface1..j.methodName$require(arg1, …, argn, this) != null ||
            outside1..j.methodName$require(arg1, …, argn, this) != null)
            // Inject extracted preconditions here
    } catch (AssertionError e) {
        error = new PreconditionError(e);  }
    return error;  }
```

Figure 6. The precondition checking template

### 3.1.3 Injection of old expression preserving methods
Fig. 7 illustrates the template of the old expression preserving method. Because old expressions can only be used in postconditions, they are and-ed just as postconditions are. The actual statements that stores values of old expressions are first extracted from the bytecode of the original method and then modified to have the ability to preserve the values. Finally, the modified bytecode is injected into the template. The extraction and modification of old expressions is discussed later.

```java
public void methodName$old (arg1, ..., argn, oldList) {
    try {
        super.methodName$old(arg1, ..., argn, oldList);
        interface1..j.methodName$require(arg1, ..., argn, this, oldList);
        outside1..j.methodName$require(arg1, ..., argn, this, oldList);
        // Statements that store values of old expressions is injected
        // into here
    } catch (Exception e) {
        error = new Error("Internal error when storing old values", e);
    }
}
```

Thus, we have to fetch another instruction (the object reference), which is theaload_0 (line 41).

A more sophisticated algorithm can be derived from a complete study of the Java instruction set. Currently, we support the invocation of virtual and static functions, and the access to instance variables in the old expressions. Our approach is simple yet sufficient to cover the primary use of old expressions, that is, accesses of object states from query methods (e.g., getter methods in Java) and instance variables.

3.2 Extraction operations

3.2.1 Extraction of preconditions, postconditions, and bodies

Fig. 8 shows the bytecode snippets of the pop() method represented in Fig. 1. Because the blocks of pre- and post-conditions have been identified by the marker methods, extraction of the bytecode into the corresponding pre- and post-condition-checking methods is relatively straightforward. To find these code ranges from the bytecode, we search for an invokevirtual instruction (a Java bytecode instruction used to invoke a static method) with operands referencing to the marker methods of pre- and post-conditions (e.g., lines 0, 26, 29, and 93).

Finding the bytecode of the method body is relatively easy. The bytecode of the method body is between the line following Ensure.end() and the end of the method (lines 96-111).

3.2.2 Extraction of old expressions

The bytecode snippets regarding the two old expressions of Fig. 1 are illustrated in Fig. 9. To find an old expression, we first search for an invokevirtual instruction with the operand referencing to Old.value (lines 45 and 75). Then, we look forward from this location to identify a range of bytecode representing an expression. For example, expression Old.value(top()) is compiled into three bytecode instructions: aload_0, invokevirtual, invokestatic (see Fig. 9).

Thus, when we look for the preceding instruction of line 45 to find an expression, we first read an invokevirtual instruction. According to the Java Virtual Machine (JVM) specification [14], when this instruction is fetched, the state of the JVM stack is:

```
..., object reference, [arg1, [arg2 ...]]
```

Figure 7. Code snippets of the old expression preserving template

Figure 8. Bytecode snippets of the pop() method

Figure 9. Bytecode snippets of old expressions in the top() method
3.2.3 Modification of old expressions

The extracted old expressions are merely markers and have no ability to store the values. Thus, when they are extracted, the markers are replaced with invocations of methods of a particular object OldList which can perform the actual preservation task. Fig. 10 shows the code snippets of OldList. When an old expression, for example, Old.value(top()), is extracted, it is replaced by an invocation of add(top()) of an OldList object. As a result, the OldList object eventually preserves values of all old expressions.

Since old expressions are used in the postconditions, the Old.value(top()) and Old.value(size()) expressions must be replaced by removeAsObject() and removeAsInt(), respectively. That is, the access of the old value is type specific and the instrumentor is responsible for choosing the suitable method returning a correct type.

Figure 10. Code snippets of the OldList class

The extraction, modification, and injection of old expressions and other contract elements are all performed at the bytecode level by the instrumentor. The instrumentor is built on top of Javassist [10], a bytecode engineering tool. Javassist has two levels of API: source level and bytecode level. Code templates such as precondition checking and old expression preservation (see Fig. 6 and Fig. 7) are manipulated using the source level API while bytecode extraction and modification are performed by the bytecode level API. Note that the detailed architecture and design of the instrumentor is non-trivial and thus is not discussed in this paper.

4. Related work

A survey of ten different DBC tools regarding contract annotation and contract processing is listed in Table 1. In designing ezContract, we strived to encompass the advantages of existing DBC tools while avoiding their shortcomings. Compared to existing tools, the proposed ezContract tool has all the following major benefits.

- **Maintaining source compatibility.** Because the markers are implemented by regular Java classes, the annotated programs can be directly compiled by standard Java compilers.
- **No symbolic barrier.** Since contracts are standard Java code and are bound to their targets at compile time, there is no symbolic barrier problem. For example, if the top() method defined in the IStack interface (see Fig. 2, line 15) is renamed to head() by a refactoring tool, this change can be propagated to all places where the method is called (e.g., line 31). In contrast, if contracts are annotated in comments such as iContract and Jass do, the refactoring tool does not have the syntactical information to reflect this change.
- **Improving readability.** Contracts are collocated with their targets as much as possible as a means to improve readability. The use of the Java language as the contract specification language contributes to readability as well.
- **Improving writability and enabling streamlined integration with IDEs.** The specification of contracts is the same as writing regular Java programs. Thus, with the help of modern IDEs which provide programming assistance such as auto-completion, incremental compilation, and quick fix, writing contracts is much easier than that in a syntax-unaware environment (e.g., in comments or in a general text editor).
- **Supporting contract specification without source code.** By providing the ability to define external contracts, specifying contracts for binary Java classes is supported.

5. Conclusion

We have surveyed existing contract specification mechanisms and proposed a new approach to annotate contracts in Java. Our approach includes specifying contracts with dummy marker objects and implementing runtime contract checking with a bytecode instrumentor. Major issues in generating runtime contract checking are presented. Our approach
avoids the source compatibility and the symbolic barrier problems while improving program readability and writability. Thus, the proposed tool should be easier to integrate with modern IDEs.

The proposed tool is still under construction regarding its robustness, completeness, and performance. We are currently improving the quality of the tool to make it publicly available. In addition, an Eclipse plug-in is concurrently developed to provide a sophisticated development environment for adopting Design by Contract in Java.

6. Acknowledgement

This research is supported by the National Science Council of Taiwan under the grant NSC95-2221-E-027-046-MY3.

References


<table>
<thead>
<tr>
<th>DBC Tool</th>
<th>Contract Specification</th>
<th>Contract Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>iContract</td>
<td>comments (internal contracts)</td>
<td>preprocessor</td>
</tr>
<tr>
<td>JML</td>
<td>comments (internal contracts)</td>
<td>preprocessor</td>
</tr>
<tr>
<td>Jass</td>
<td>comments (internal contracts)</td>
<td>preprocessor</td>
</tr>
<tr>
<td>Kopi</td>
<td>keyword extension for Java (internal contracts)</td>
<td>full language compiler</td>
</tr>
<tr>
<td>HandShake</td>
<td>separate contract files (external contracts)</td>
<td>contract compiler + postprocessor</td>
</tr>
<tr>
<td>jContractor</td>
<td>standard source with naming pattern (internal and external contracts)</td>
<td>postprocessor</td>
</tr>
<tr>
<td>Contract4J</td>
<td>standard source with naming pattern (internal contracts)</td>
<td>contract compiler + AOP-based postprocessor</td>
</tr>
<tr>
<td>Contract4JBeans</td>
<td>standard source with naming pattern (internal contracts)</td>
<td>AOP-based postprocessor</td>
</tr>
<tr>
<td>ezContract</td>
<td>standard source with marker library (internal and external contracts)</td>
<td>postprocessor</td>
</tr>
<tr>
<td>Spec#</td>
<td>keyword extension for C# (internal contracts)</td>
<td>full language compiler</td>
</tr>
</tbody>
</table>
[17] Sun Microsystems, Programming with Assertions,
http://java.sun.com/j2se/1.4.2/docs/guide/lang/assert.html