PAPER

Contract Specification in Java: Classification, Characterization, and a New Marker Method*

Chien-Tsun CHEN[†], *Nonmember*, Yu Chin CHENG^{†a)}, *Member*, and Chin-Yun HSIEH[†], *Nonmember*

SUMMARY Design by Contract (DBC), originated in the Eiffel programming language, is generally accepted as a practical method for building reliable software. Currently, however, few languages have built-in support for it. In recent years, several methods have been proposed to support DBC in Java. We compare eleven DBC tools for Java by analyzing their impact on the developer's programming activities, which are characterized by seven quality attributes identified in this paper. It is shown that each of the existing tools fails to achieve some of the quality attributes. This motivates us to develop ezContract, an open source DBC tool for Java that achieves all of the seven quality attributes. ezContract achieves streamlined integration with the working environment. Notably, standard Java language is used and advanced IDE features that work for standard Java programs can also work for the contract-enabled programs. Such features include incremental compilation, automatic refactoring, and code assist.

key words: Design by Contract, contract specification, classification, characterization, quality attribute, Java

1. Introduction

Introduced by Bertrand Meyer over two decades ago, *Design by Contract* (DBC) [20], [21] is now commonly acknowledged as a pragmatic method for building reliable software. Contracts, including preconditions, postconditions, and class invariants, are assertion statements that specify obligations and benefits of a component supplier and its client [21]. Preconditions specify obligations of clients while postconditions state the promises of the supplier. Class invariants specify conditions on publicly visible object states that must be satisfied during the object's lifetime. Contracts are also regarded as an effective debugging mechanism because they can pinpoint the error when a program deviates from its normal state.

Although Java does not have full built-in support for DBC [25], the benefits have been noticed by the Java community and many attempts have been made to bring DBC support to Java. With numerous tools available, there is the question of which one to choose. Although functionality-based evaluation criteria have been proposed in the past (e.g., see [8], [23]), in this paper, we take the position that the *impact of adopting a DBC tool on the developer's programming activities* - a non-functional aspect - constitutes

the most important evaluation criteria for tool selection. Specifically, is the host language extended? Are integrated development environments (IDEs) affected? Are features of IDEs such as incremental compilation, automatic refactoring, code assist, and syntax high-lighting applicable to contract code? And so on. Precisely because DBC is not part of most mainstream languages like Java, C++ and C#, extensions to languages and IDEs are necessary to bring about DBC support. Only tools that exert minimal impact on the developer's programming activities stand a chance of getting accepted by developers.

Key design issues of supporting DBC in Java and other languages include *contract specification* (how contract code gets written), *contract translation* (how contract code is translated into executable code), and *contract evaluation* (how contract checking is performed during runtime). Of the three, contract specification is most relevant to impact analysis as it affects both the developer and the working environment. The first contribution of this paper is the contract-specification based scheme for classifying existing DBC tools into categories. Tools belonging to a category are further characterized with seven quality attributes associated with the working environment where programming activities take place. The classification and impact characterization are applied to eleven DBC tools for Java and five tools for C#. Notably, our investigation shows that existing DBC tools lack some of the quality attributes.

The second contribution of this paper is *ezContract*, a DBC tool for Java that achieves all of the seven quality attributes. All essential DBC functionalities are supported, including preconditions, postconditions, class invariants, sub-contracting, old expressions, and accessing return values. Contract specifications for Java classes as well as interfaces are supported. The quality attributes are achieved by ezContract primarily by using a novel approach to delimit contracts in Java with markers. Specifically, markers are predefined *dummy methods* and *attributes* which are used to emulate contract defining keywords such as *require*, *ensure*, *invariant*, *old*, and *result* in Eiffel [20], [21]. The Java *assert* keyword is used to write assertions in pre-, postconditions, and class invariants delimited by the markers. Since contracts are written in standard Java language constructs, the program can be compiled by standard Java compilers. To produce contract checking code, the produced bytecode is further processed by a *bytecode instrumentor*, which injects contract evaluation statements into the bytecode.

Manuscript received April 3, 2008.

Manuscript revised June 26, 2008.

[†]The authors are with the Department of Computer Science and Information Engineering, National Taipei University of Technology, Taipei 106, Taiwan.

*A preliminary version of this paper was published in the 14th Asia-Pacific Software Engineering Conference (APSEC'07).

a) E-mail: yccheng@csie.ntut.edu.tw

DOI: 10.1093/ietisy/e91-d.11.2685

With ezContract, software developers can write contract without giving up their favorite IDEs; all that is necessary is to import the marker library and to invoke the bytecode instrumentor after compilation. Advanced IDE features such as incremental compilation, automatic refactoring, and code assist that work for standard Java programs can also work for the contract-enabled programs. We have made ezContract available as open source software, which can be accessed via <https://sourceforge.net/projects/ezcontract/>.

The rest of this paper is organized as follows. Section 2 surveys existing DBC tools for Java according to their contract specification methods. Section 3 establishes a quality attribute based framework for characterizing and evaluating DBC tools. In Sect. 4, ezContract is presented with concrete examples. Section 5 briefly illustrates the contract translation process of ezContract. Section 6 describes related work. Section 7 concludes this paper.

2. A Survey of Existing Tools for Contract Specification

Eleven DBC tools for Java are reviewed. Since some of these tools share certain common features, to better characterize the tools, they are first classified into a number of categories by the way contracts are specified. Specifically, the following questions about contract specification are asked in succession. First, does the tool make use of Java language constructs for contract specification or a purpose built contract specification language? In the former, we further ask if new keywords are introduced to Java for contract specification, or existing Java language constructs are used to emulate contract specification. In the case that a contract language is used, we further distinguish whether contracts are specified in comments, Java annotations, or aspect-like constructs. The eleven tools are classified into five categories as shown in Fig. 1.

1. *Keyword extension*: Contracts are specified with constructs of a Java language extended with DBC keywords. Kopi [18] is the only instance of this category with the keywords `@require`, `@ensure`, and `@invariant` for preconditions, postconditions, and class invariants, respectively. Since contracts are properly part of the language, a full language compiler is required for its compilation.
2. *Naming convention*: Contracts are specified in methods of the standard Java language to emulate DBC keywords. The methods are named following a special naming pattern. Two examples are jContractor [1] and Contract4JBeans [11]. For instance, in jContractor, pre- and post-conditions for a method `m` are written in two protected methods `m_PreCondition` and `m_PostCondition`, respectively. A post-compilation contract processor is employed to enable contract checking by bytecode manipulation.
3. *Contracts in comment*: Contract descriptions must con-

Fig. 1 A classification scheme of contract specification methods in Java.

form to the syntax of a contract language, which provides predefined tags for DBC keywords. For example, iContract [17] uses three tags `@pre`, `@post`, and `@inv` to mark the beginnings of contract sections in comments. Other instances of this method include Jass [6], JMSAssert [15], JML [19], and Jcontract [22]. Since contracts written in comments are ignored by standard Java compilers, a preprocessor is employed to convert contracts into standard Java programs before compilation. Contract specifications in comments are popular before Java annotations became available.

4. *Contracts in Java annotation*: Contracts are specified as string literals in tool-defined Java annotations. The string literals must conform to the syntax rules of the contract language. Two instances in this category are Contract4J5 [10] and Cona [9], which define the annotations `@Pre`, `@Post`, and `@Invar` (`@invariant` in Cona) for preconditions, postconditions, and class invariants. Specifying contracts in Java annotations is similar to specifying contracts in comments, except annotations are part of the standard Java language since Java 5.0 and can be preserved in bytecode after compilation. Annotations provide better implementation flexibility by allowing contracts to be processed both prior to and after program compilation. Annotation-based specification of contracts is gaining popularity; [16] and [24] are two instances outside of Java.
5. *Contracts in aspect-like construct*: Contracts are treated like aspects (i.e., cross-cutting concerns) in an aspect-oriented programming (AOP) language, which are extracted from the target Java program and described separately. However, unlike aspects in AOP, which are usually weaved to multiple targets, in the existing DBC tool of this category, contracts are weaved to one specific target such as a class or an interface. Thus, tools in this category use a limited form of AOP. Additionally, a mechanism (e.g., a configuration file) is required to direct the weaving of contracts to their tar-

gets. This method enables specifying contracts even if the source code is not available. HandShake [12] is the only instance of this category. Aspect-like contracts are first processed by a contract compiler. The generated code can then be injected to the Java program either prior to or after compilation. AOP-based implementation of DBC is gaining popularity but not without controversy [4].

3. An Evaluation Framework

In the classification of Fig. 1, DBC tools in the same category are more similar; tools in different categories are more different. Granted this, Fig. 1 does not answer questions like “Should I choose contracts in annotations over contracts by naming conventions?” and “Should I prefer contracts in comments to keyword extensions?” In order to answer these and other similar questions, we take a *developer-centric* view for evaluating the existing categories of DBC tools for contract specification. We propose a framework for qualitatively analyzing the existing tools by category. The analysis is done by assessing the impact of adopting a DBC tool on developer’s programming activities, by way of a number of *quality attributes* associated with the programming environment (Fig. 2).

In Fig. 2, a developer writes programs in a *host language* such as Java in a chosen *IDE*. The IDE has a number of *features*, including incremental compilation, automatic refactoring, code assist, and so on. In modern IDEs such as Eclipse, NetBeans, and Visual Studio.NET, these features are flexibly plugged in and unplugged from an IDE. The host language, the IDE, and the pluggable features constitute the sources of impact on developer’s programming activities should any of them become affected by the adoption of a DBC tool.

3.1 Quality Attributes for Evaluation

We identify seven DBC-related quality attributes that can affect the developer’s programming activities in adopting DBC. The quality attributes are stated in ways that, by achieving as many of them as possible, the developer can practice DBC with as little change on the host language, the IDE, and the pluggable features as possible.

Source compatibility (SC). The source language is not changed in order to support DBC. Source compatibility promises mainstream support in both languages and IDEs; breaking source compatibility denies it. Introducing new keywords to support DBC breaks source compatibility. In the case of Kopi, Java is extended to become an Eiffel-like language where contracts are concerned. However, until the extension becomes a standard part of Java, a proprietary compiler and/or an IDE must be made available. A proprietary compiler/IDE can be costly to maintain since it must be compatible with Java, a rapidly evolving language. Failing to keep up with evolution can mean that the developers

Fig. 2 Working environment for evaluating DBC tools.

are involuntarily locked in to an out-of-date version of Java.

No symbolic barrier (NSB). Contracts are correlated to the specified targets from the compiler’s perspective. The correlation implies that changes in the targets are automatically reflected in the contracts, and vice versa. This allows contracts to be covered by existing IDE features such as automatic refactoring and incremental compilation. Symbolic barrier [13] is introduced by DBC methods that use comments, naming conventions, Java annotations, and aspect-like constructs for contract specification. For example, contracts written in comments (e.g., in iContract) and in Java annotations (e.g., in Contract4J5) become invalid after applying automatic refactoring to rename a method parameter because the refactoring tool does not have the syntactical information to reflect such a change in the contracts. Also, renaming a method and a class invalidate contracts documented with naming conventions (e.g., in jContractor) and contracts documented in aspect-like constructs (e.g., in HandShake), respectively.

Line number information for debugging (LNIFD). Contract violation at runtime will raise exceptions with the line number of the violating statement made available. The line number information is essential for debugging because it pinpoints the location where the program first deviates from its correct state. When DBC tools that document contracts with comments, annotations, and aspect-like constructs are adopted, the IDE cannot readily provide this valuable information.

Syntax-aware specification (SAS). Syntax errors in contracts are checked at compile time by a standard compiler. This helps making contract specification an integrated activity of programming. In addition, a syntax-aware IDE usually provides a code assist feature that can save some coding time and avoid typographical errors in writing contracts. Contracts specified with new keywords, comments, Java annotations, and aspect-like constructs are not supported by the standard compiler and IDE.

Collocation of contract and target (COCAT). Contracts should be placed as closely as desirable to the targets [4], [21] so that they are easy to read and write. In the existing DBC methods, contracts specified with keywords, com-

Table 1 Characterization of six contract specification methods against the proposed quality attributes.

	SC	NSB	LNIFD	SAS	COCAT	CSWSC	DI
Keyword	No	Yes	Yes	No	Yes	No	High
Naming convention	Yes	No	Yes	Yes	No	No*	Medium
Comment	Yes	No	No	No	Yes	No	Low
Java annotation	Yes	No	No	No	Yes	No	Medium
Aspect-like construct	Yes	No	No	No	No	Yes	Low
Marker	Yes	Yes	Yes	Yes	Yes	No*	Medium

SC: Source compatibility

NSB: No symbolic barrier

LNIFD: Line number information for debugging

SAS: Syntax-aware specification

COCAT: Collocation of contract and target

CSWSC: Contract specification without source code

DI: Decommissioning impact

* jContractor (using naming convention) and ezContract (using marker) also support CSWSC via contract files.

ments, and Java annotations can be placed more closely to the targets than those with naming conventions and with aspect-like constructs.

Contract specification without source code (CSWSC). Existing components, which are often provided without source code, are written without contracts. In developing a DBC-enabled application, it becomes necessary to extend DBC support to any existing components used. This is possible only with contracts written separately from their target programs.

Decommissioning impact (DI). Some impact, including language compatibility, program readability and understandability, and so on, can be felt by the developers when DBC is decommissioned after some duration of use for any reason. Normally, developers will want to revert to standard language/compiler/IDE after decommissioning DBC. Depending on the methods used, DBC tools can have different decommissioning impact. Methods with low impact can encourage developers to try DBC. For example, in a method using new keywords, contracts in programs become syntax errors in the standard language/compiler/IDE and must be removed. In contrast, if contracts are written in comments, nothing needs to be done although the comments containing contracts become meaningless. Further, it can be argued that since programmers can more easily filter DBC-related comments after decommissioning. This is in contrast with methods using naming conventions, annotations, and markers since the programmers must make a cognitive effort to filter the DBC-related code after reading it. Finally, a method employing aspect-like constructs incurs the minimum decommissioning impact by simply excluding the contract files.

Table 1 tabulates the five categories of contract specification methods against the seven quality attributes. For example, jContractor, being an instance of the naming convention method, is characterized as follows. It is source compatible since standard Java is used; it induces symbolic barrier since a contract is written in a separate method with a name obtained by suffixing the target's method name; it can readily provide the line number of a contract in case of a contract violation; it achieves syntax-aware specification since contracts are written in standard Java; it fails collocation since contracts and targets are separate methods; it allows contract to be specified without source code; and it

achieves a medium level of decommissioning impact since the contract related methods bloat the class unless removed.

With the characterization, the developer can more judiciously decide whether a given tool is a good choice. For example, while jContractor achieves most of the quality attributes, it induces symbolic barrier that breaks refactoring tool support. Thus, if the developer constantly practices refactoring, jContractor may not be a good choice.

An alternative way to use Table 1 is to begin with the quality attributes one wishes to acquire and select the best tool available accordingly. Where a quality attribute is lacking in a selected DBC method, the developer can estimate the cost of its acquisition. In this way, the most cost-effective DBC method can be selected and enhancements to the developing environment can be planned. For example, in adopting jContractor, it can be important to enhance the refactoring tool to cover contracts.

4. Specifying Contracts with ezContract

In this section, we present a DBC tool for Java called ezContract that achieves all of the seven quality attributes. ezContract makes use of marker libraries, which are predefined dummy methods and attributes, to delimit the code segment for contract specification. Specifically, the marker method achieves all quality attributes except CSWSC, which is provided via separate contract files and is discussed in Sect. 4.3. ezContract supports contract specification for Java classes and interfaces. In this section we illustrate how to write contracts with ezContract by an example on stack.

4.1 Contracts for Classes

Figure 3 shows how contracts are specified for a `Stack` class with ezContract. Contracts, including preconditions, postconditions, and class invariants (lines 6–8, 10–15, and 20–22, respectively), are specified by the `assert` statements. Each group of assertions pertaining to the same contract is delimited by markers so that these assertions can be bound to particular types of contracts.

There are three types of markers. In the first type, markers are used in pairs to form a block for preconditions or postconditions. In Fig. 3, `Require.begin()` and `Require.end()` form a precondition block; `Ensure.`

`begin()` and `Ensure.end()` form a postcondition block.

Markers of the second type include `Old.value()` and `Result.value` (line 11). The former is used to preserve the value of an expression upon entering a method and the latter represents the return value of a method. The third type of markers has only one instance, namely, the `classInvariant()` method (lines 20–22), which is defined in the interface `IContract`. Classes with invariants must implement the `IContract` interface.

With the Java assertions and markers, writing contracts is just like writing regular Java programs. This achieves SC and NSB. LNIFD is provided because contracts are regular Java statements that occupy physical locations in source code. SAS is satisfied since contracts can be directly written in a syntax-aware environment provided by any regular Java IDE. ezContract accomplishes COCAT because contracts are collocated with their targets as in Eiffel. ezContract attains medium level of DI because contracts written with the `assert` statements are disabled by default [2], [25]. However, completely removing ezContract's contracts from source code requires manual operations.

4.2 Contracts for Interfaces

Contracts for interfaces are written similarly as in classes. However, since non-static methods in a Java interface can only have declarations but not implementations, a static and final inner class is used for defining contracts, as illustrated in `$Contract` (line 8) of Fig. 4. This inner class implements the interface to be contracted (the `IStack` interface in Fig. 4).

4.3 Separate Contract Files

Preferably, contracts written with ezContract are collocated with the targets in the source program. But sometimes source code is not available, e.g., when thirty-party components without source code are used. To achieve CSWSC, ez-

Contract supports writing contracts in separate contract files. To define a contract file for a class/interface, a new Java class is created that extends/implements the class/interface for which contracts are to be specified. For example, Figure 5 shows a contract file for the `Stack` class. The contract file extends (i.e., inherits) `Stack` class and overrides methods to write contracts, as shown in line 4–6 in Fig. 5.

4.4 Old Expressions and Results

The `old` expression in Eiffel is used to preserve the state of an expression (usually the value of an instance variable) upon entering a method. The preserved value is used in postconditions to ensure that the state of an object has been changed as desired. As shown in Fig. 3, lines 11 and 13, the marker method `Old.value()` emulates Eiffel's `old` expression.

In postconditions, the return value of a method can be accessed with the marker `Result.value`, as shown in Fig. 3, line 11. The `Result` class defines a static variable named `value` whose type is `Object` so that it can be as-

```
01 public interface IStack<T> {
02
03     public int size();
04     public T top();
05     public void push(T item);
06     public T pop();
07
08     static final class $Contract<T>
09         implements IStack<T>, IContract {
10
11         public T pop() {
12             Require.begin();
13             assert size() > 0 : "stack size should be > 0";
14             Require.end();
15
16             // ... postcondition here
17             return null;
18         }
19         // ... code for class invariant
20     }
21 }
```

Fig. 4 Contracts for interface `IStack`.

```
01 public class Stack<T> implements IContract{
02
03     protected List<T> fTable = new ArrayList<T>();
04
05     public T pop() {
06         Require.begin();
07         assert size() > 0 : "stack size should be > 0";
08         Require.end();
09
10         Ensure.begin();
11         assert Result.value == Old.value(top()) :
12             "item has been saved on top location";
13         assert size() == Old.value(size()) - 1 :
14             "stack size should be decreased by 1";
15         Ensure.end();
16
17         return fTable.remove(size() - 1);
18     }
19
20     public void classInvariant() {
21         assert size() >= 0 : "stack size should be >= 0";
22     }
23     // code for size, top, and push methods
24 }
```

Fig. 3 Contracts for class `Stack`.

```
01 public class SeparateContractForStack<T>
02     extends Stack<T> {
03     public T pop() {
04         Require.begin();
05         assert size() > 0 : "stack size should be > 0";
06         Require.end();
07
08         // ... postcondition here
09         return null;
10     }
11     // code for size, top, and push methods
12     // code for class invariant
13 }
```

Fig. 5 Contracts defined in a separate contract file.

```
01 public final class Ensure {
02     private Ensure() {}; // private constructor
03     public static void begin() {};
04     public static void end() {};
05 }
```

Fig. 6 `Ensure` class source code.

signed to a return value not just of an object type but also of a primitive type due to the auto boxing and unboxing mechanism provided since Java 5.0 [2].

4.5 Marker Libraries

Since markers are dummy class methods and attributes, their implementations are not difficult. Figure 6 shows the `Ensure` marker class. Because markers are used only to delimit code blocks in the bytecode, they contain no programming logic. Also, a marker class should not be instantiated. Thus, the default constructor is private. Marker methods are static so that they can be directly invoked without an object reference.

5. Translation

The implementation of `ezContract` has been presented in [7]. This paper gives a brief overview of the translation process of `ezContract`. Readers interested in the detail implementation of `ezContract` can also explore the publicly available source code.

Figure 7 shows the translation workflow of `ezContract`. Java programs with contracts and separate contract files are first compiled by a standard Java compiler. The generated Java bytecode is then processed by a bytecode instrumentor provided by `ezContract` to produce contract checking code, which is executable by standard JVM. A contract configuration file is provided to the bytecode instrumentor so that it can determine which separate contract files to use.

A contract-enabled method follows the following steps during its execution. First, the class invariant and the precondition are checked. Second, the method body is ex-

ecuted. Third, the postcondition and the class invariant are checked. Fourth, the class invariant is checked before leaving the method even if the evaluation of postconditions failed. Finally, the subcontracting rule [21] is checked. That is, preconditions are or-ed, postconditions and class invariants are and-ed with those of the base class and outside contract classes, respectively. Any contract violation causes a runtime exception to be thrown. Note that nested contract evaluation is not allowed because it causes endless recursive evaluation of contracts [21]. The bytecode instrumentor must correctly implement the above rules.

The work of contract translation is performed by the bytecode instrumentor. In a nutshell, the bytecode instrumentor performs three operations on the bytecode: extracting contracts from the bytecode, generating new contract checking methods by marker type, and injecting the generated contract checking methods into the original method.

6. Related Work

Although DBC has been proposed for more than two decades, only a handful of criteria for evaluating contract-supporting tools have been reported. Further, existing evaluations tend to focus on functional aspects of DBC support. Plösch [23] established four groups of evaluation criteria for contract tools for Java, including basic assertion support, advanced assertion support, support for behavioral subtyping, and runtime monitoring of assertions. The criteria cover basic DBC support such as preconditions, postconditions, and class invariants as well as advanced DBC support such as old expressions, assertion expressions on collections, and subcontracting. Clarke and Rosenblum [8] proposed a subset of similar evaluation criteria and evaluated DBC tools not only for Java, but also for C/C++, C#, and Eiffel. In this work, the DBC tools are evaluated from a quality attribute (i.e., non-functional) point of view. The quality attribute view complements the functional view because the latter does not reveal the impact of adopting DBC tools on the developer's programming activities. The evaluation framework also acts as a baseline for DBC tool designers and users against which to assess a DBC tool under development and use.

The proposed evaluation framework can be applied to assess DBC tool support for other languages as well. For example, we reviewed five DBC tools for C#, including `Spec#` [5], the tool by Köhne et al. [16], `XC#` [24], `Contract Wizard` [3], and `Contract Sharp` [14] and found that they can be put into the five categories of Fig. 1. To summarize, the `Spec#` employs the keyword extension method; Köhne et al. and `XC#` write contracts in .NET attributes (the .NET equivalent of Java annotations); and `Contract Wizard` and `Contract Sharp` document contracts in aspect-like constructs. More specifically, `Spec#` extends C# with three keywords: `requires`, `ensures`, and `invariant`. In the tool by Köhne et al., three .NET attributes are used to write contracts: `PreconditionAttribute`, `PostconditionAttribute`, and `InvariantAttribute`.

Fig. 7 The translation workflow of `ezContract`.

XC# supports the specification of preconditions and postconditions with `RequiresAttribute` and `EnsuresAttribute`. In Contract Wizard, contracts are written via a GUI-based form and saved in a generated Eiffel class file. The Eiffel class is compiled by an Eiffel compiler and generates contract checking code for the target class. Contract Sharp also applies a GUI-based method to write contracts. Contracts are saved in a XML file, which is translated to a C# program and compiled by a standard C# compiler.

It should be noticed that although programmers using Contract Wizard and Contract Sharp do not actually write contracts in aspects, the generated Eiffel classes and XML-based files are regarded as aspect-like constructs in our classification since they extract contracts from the target program and describe contracts separately. Also notice that although Contract Wizard and Contract Sharp provide GUIs to simplify contract specification, they fail to achieve COCAT and SAS because contracts are written in an environment detached from the source code. In addition, the use of aspect-like constructs causes the symbolic barrier problem and makes providing line number information impossible.

Another research topic of interests related to DBC is aspect-oriented programming. Although the literature of AOP frequently cites DBC as a candidate target to be aspectized, Balzer et al. argued that “contracts are not crosscutting by nature and aspectizing them risks creating interactions with other aspects” [4]. Our investigation shown in Table 1 supports this argument because the method of aspect-like constructs fails to achieve four of the seven quality attributes: NSB, LNIFD, SAS, and COCAT. Nevertheless, many attempts have been made to implement DBC with AOP. It is important to distinguish two closely related but different usages of AOP in light of supporting DBC: AOP for contract specification and AOP for contract translation. An example of writing contracts in aspects was given by Balzer et al. in [4]. In fact, many AOP-based DBC tools do not write contracts in aspects; they use AOP features such as inserting a code block before, after, or around a method to implement DBC. For example, in Cona and Contract4J5, contracts are written in Java annotations rather than in aspects. AOP languages are used to generate runtime checking statements for contracts by inserting preconditions and postconditions before and after a method invocation, respectively. It should be an interesting research issue to evaluate that whether AOP features are suitable for implementing a contract translator.

7. Conclusion

In this paper, we presented a DBC tool for Java called ezContract, which introduced a novel marker method for contract specification. To investigate the benefits of ezContract, a classification scheme which includes five categories of contract specification methods regarding existing DBC tools for Java was proposed. We further established seven quality attributes to characterize and evaluate these DBC tools

against the classification. The results showed that existing DBC tools for Java fails to achieve some quality attributes. The proposed marker method added a new contract specification category to the classification scheme. With markers and the support of separate contract files, we showed that ezContract achieves all seven quality attributes.

ezContract is an open source software. A preliminary version of ezContract has been released on SourceForge. So far, contracts for classes are fully supported; we are still working on the bytecode instrumentor to support contracts for interfaces. In addition, an Eclipse plug-in is concurrently developed to provide a sophisticated development environment for adopting Design by Contract in Java.

Acknowledgements

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

References

- [1] P. Abercrombie and M. Karaorman, “jContractor: Bytecode instrumentation techniques for implementing design by contract in Java,” *ENTCS*, vol.70, no.4, pp.55–79, Dec. 2002.
- [2] K. Arnold, J. Gosling, and D. Holmes, *Java Programming Language*, 4th ed., Addison-Wesley, Boston, 2005.
- [3] K. Arnout and R. Simon, “The .NET contract wizard: Adding design by contract to languages other than Eiffel,” *Proc. TOOLS’39*, pp.14–23, July 2001.
- [4] S. Balzer, P. Th. Eugster, and B. Meyer, “Can aspects implement contracts,” *LNCS*, vol.3943, pp.145–157, 2006.
- [5] M. Barnett, K. Rustan, M. Leino, and W. Schulte, “The Spec# programming system: An overview,” *LNCS*, vol.3362, pp.49–69, 2004.
- [6] D. Bartecko, C. Fischer, M. Möller, and H. Wehrheim, “Jass-Java with assertions,” *ENTCS*, vol.55, no.2, pp.103–117, Oct. 2001.
- [7] Y.C. Cheng, C.-T. Chen, and C.-Y. Hsieh, “ezContract: Using marker library and bytecode instrumentation to support design by contract in Java,” *Proc. APSEC’07*, pp.502–509, Dec. 2007.
- [8] L.A. Clarke and D.S. Rosenblum, “A historical perspective on runtime assertion checking in software development,” *ACM SIGSOFT Software Engineering Notes*, vol.32, no.3, pp.25–37, 2006.
- [9] Cona, <http://www.ccs.neu.edu/home/skotthe/cona/>
- [10] Contract4J5, <http://www.contract4j.org/contract4j/c4j5>
- [11] Contract4JBeans, <http://www.contract4j.org/contract4j/c4jbeans>
- [12] A. Duncan and U. Holzle, “Adding contracts to Java with hand-shake,” Technical Report TRCS98-32, University of California, Santa Barbara. Computer Science, Dec. 1998.
- [13] M. Fowler, “Language workbenches: The killer-app for domain specific languages,” <http://www.martinfowler.com/articles/languageWorkbench.html>
- [14] R. Henne-Wu, W. Mitchell, and C. Zhang, “Support for design by contract in the C# programming language,” *JOT*, vol.4, no.7, pp.65–82, Sept.-Oct. 2004.
- [15] JMSAssert, <http://www.mmsindia.com/DBCForJava.html>
- [16] K. Köhne, W. Schult, and A. Polze, “Design by contract in .NET using aspect oriented programming,” <http://www.dcl.hpi.uni-potsdam.de/research/loom/Slides/DBC.pdf>
- [17] R. Kramer, “iContract-the Java designs by contract tool,” *Proc. TOOLS’26*, pp.295–307, Aug. 1998.
- [18] M. Lackner, A. Krall, and F. Puntigam, “Supporting design by contract in Java,” *JOT*, vol.1, no.3, pp.57–76, Aug. 2002.
- [19] G.T. Leavens, A.L. Baker, and C. Ruby, “Preliminary design of JML: A behavioral interface specification language for Java,” *ACM SIGSOFT Software Engineering Notes*, vol.31, no.3, pp.1–38,

March 2006.

- [20] B. Meyer, *Eiffel: The Language*, Prentice-Hall, New York, 1991.
- [21] B. Meyer, *Object-Oriented Software Construction*, 2nd ed., Prentice Hall PTR, Upper Saddle River, NJ, 1997.
- [22] Parasoft, "Using design by contract to automate software and component testing," <http://www.parasoft.com/jsp/products/article.jsp?articleId=579&product=Jtest>
- [23] R. Plösch, "Evaluation of assertion support for the Java programming language," *JOT*, vol.1, no.3, pp.5–17, Aug. 2002.
- [24] ResolveCorp, "eXtensible C#," <http://www.resolvecorp.com/XCSharp.ppt>
- [25] Sun Microsystems, "Programming with assertions," <http://java.sun.com/j2se/1.4.2/docs/guide/lang/assert.html>

Chien-Tsun Chen is a PhD candidate at the Department of Computer Science and Information Engineering, the National Taipei University of Technology, Taiwan. Prior to commencing his graduate study, he served as the CTO of CanThink Inc, which provides e-learning software solutions and services. His research interests include Design by Contract, pattern languages, and software architecture. He has actively contributed to several successful commercial and open source software projects. Mr. Chen is a

member of the Software Engineering Association of Taiwan (SEAT).

Yu Chin Cheng received the MSE degree from the Johns Hopkins University in 1990 and the PhD degree from the University of Oklahoma, in 1993, both in computer science. He is currently a professor at the Department of Computer Science and Information Engineering of the National Taipei University of Technology, Taiwan, where he served as the department chairperson from 2002 to 2004. Dr. Cheng teaches object-oriented programming and software architecture. His research interests include

pattern languages for software design, software architecture and image analysis. He is a member of the IEEE Computer Society, the ACM, and the SEAT.

Chin-Yun Hsieh received his MS and PhD degrees from the University of Mississippi and the University of Oklahoma, respectively, both in computer science. Dr. Hsieh held several positions at the National Taipei University of Technology, including the director of the computing center, the chairperson of the Electronic Engineering department, and the director of the Library. His research interests include programming language theory, object-oriented software engineering, and distributed systems. Dr. Hsieh

is a member of the SEAT.