

Evaluation of Assertion Support for the Java Programming Language

Reinhold Plösch, Johannes Kepler University Linz, Austria

Abstract

There is some evidence, that assertion techniques, i.e., preconditions, postconditions and invariants have a positive effect on the overall software quality. Unfortunately only a limited number of commercially relevant programming languages support assertion techniques (e.g., Eiffel). Even modern programming languages like Java have very limited built-in support for assertions. Nevertheless a number of systems exist for the the Java programming language, that support assertion techniques in different ways (language extensions, preprocessors, metaprogramming approaches). In order to make these different approaches comparable we developed a set of criteria and used these criteria to evaluate these systems.

1 INTRODUCTION

Understanding, using and customizing object-oriented class libraries are major tasks in object-oriented software development. In statically typed object-oriented programming languages like C++, Eiffel and Java, designers and programmers are both supported and restricted by the underlying type system; e.g., overriding methods must meet the type rules imposed by the base classes (static type of parameters and return values) or must meet covariant redefinition rules (e.g. Eiffel).

Additionally assertions enable describing the behavior of a class or method more precisely. Assertions are elements of formal specifications and express correctness conditions for classes and/or methods. Assertions are part of the implementation and can be checked at run time. The roots of assertions were defined by Hoare [Hoare72] and Meyer [Meyer97a]; the latter developed the idea of design by contract (DBC). Assertions may be specified at the class level (invariants) or on the method level (preconditions and postconditions).

An invariant is a correctness condition imposed on a class, i.e., a condition that must not be violated by any method of a class. A precondition is associated with a particular method and imposes a correctness condition on the client of the method; i.e., the client must ensure that the precondition is fulfilled; otherwise the method is not executed. A postcondition is also associated with a particular method, but it imposes a correctness

condition on the implementation of the method; a violation of a postcondition indicates an error in the implementation of the method. For more details on assertions and on design by contract see [Meyer97a], [Meyer97b], [Plösch97].

The Java programming language has very little support for assertions. Nevertheless a number of systems exist that add assertion support for this programming language, using different techniques. Section 2 describes principal strategies for adding assertion support for a programming language. Section 3 defines criteria for evaluating assertion support. Section 4 gives an overview of systems supporting assertions for the Java programming language. Section 5 evaluates these systems based on the developed criteria.

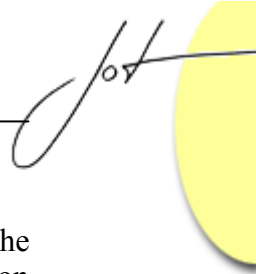
2 PRINCIPAL STRATEGIES FOR ASSERTION SUPPORT

The goal of this section is to describe different possible approaches for supporting contracts for the Java programming language. Generally three different approaches are possible:

Built in: This means that support for contracts is directly included in the programming language. The programming language contains language constructs to formulate assertions in one way or another. The syntactical correctness of assertions is checked directly by the compiler. In addition a runtime environment must be available to perform the runtime assertion checks. Ideally the runtime environment is flexible enough to allow a very fine-grained control of the assertion checking mechanism, i.e., it should be possible to selectively enable and disable assertion checking. The main advantage of this approach is the homogeneous integration of assertions into the programming language, i.e., compiler error messages are consistent, debugging tools can properly consider assertions (e.g., correct line numbers and stack traces).

Preprocessing: This is the most popular kind of support for assertions in a programming language. The general idea is to formulate assertions separate from the program or to include the assertions as comments. A preprocessor is used to weave the assertions into the program or to transform the comments containing assertion formulas into programming language code. The main advantage of this approach is the separation of programming logic and contracts. This is important in cases, where the programming language itself does not support assertions and the programming language must not be altered for various reasons (e.g., conformance to standards, not enough knowledge available for changing the compiler). The main disadvantage of this approach is that the original program code is changed by the preprocessor, i.e., line numbers of compiler errors do not actually fit the line numbers of the program. The same problem arises with debugging or runtime exceptions.

Metaprogramming: According to Templ metaprogramming refers to “programming at the level of program interpretation, or in other words, to extending the interpreter of a given programming language in an application-specific way. Traditionally, this concept is available only in dynamically typed and interpreted languages” [Templ94]. Programs that



have the possibility to reason about themselves have so called reflective capabilities. The Java programming language, e.g., has reflective capabilities and may access information about elements of a Java program by means of a reflection API. The main advantage of metaprogramming approaches is that no specialized preprocessor has been used but the native compiler. Nevertheless a specialized runtime environment has to be used to enable assertion checking.

CRITERIA FOR EVALUATING CONTRACT SUPPORT

The criteria to be used for evaluation may be split into four groups:

- Basic assertion support (BAS)
- Advanced assertion support (AAS)
- Support for Behavioral subtyping (SBS)
- Runtime monitoring of assertions (RMA)

The criteria for each group are explained in more detail below. The abbreviation of a group combined with a criteria identifier is unique and used later to compactly describe the evaluation results. The criteria themselves are explained by means of questions; this is a very straightforward and simple approach and should clearly show what the respective criteria is about. For simplicity reasons we will use the term system when describing the criteria, meaning programming languages, programming language extensions and programming systems.

Basic assertion support (BAS): This group of criteria form the very basis and should be supported by any system that claims to support a contract style of programming and modeling.

- *BAS-1 (Basic assertions):* Does the system support basic assertion annotations in the implementation of a method?
- *BAS-2 (Preconditions and Postconditions):* Does the system support preconditions? Does the system support postconditions? May assertion expressions access properties of a class? Are properties of a class guaranteed to remain unchanged during assertion checking? May assertion expressions also contain method calls? Is it guaranteed, that a method call does not produce any side effects (especially changes of the state of the object)?
- *BAS-3 (Invariants):* Is it possible to formulate invariants? Are there any restrictions in formulating invariants (compared to the formulation of preconditions or postconditions)?

Advanced assertion support (AAS): Programming languages and systems that support this group of criteria lead to more expressive contracts that allow to capture more complex contract situations.

- *AAS-1 (Enhanced assertion expressions)*: May assertions contain Boolean implications? May postconditions access the original values of parameters, i.e., the values at method entry? May postconditions access the original values of instance variables, i.e., the values at method entry? May an arbitrary expression be evaluated at method entry?
- *AAS-2 (Operations on collections)*: Does the system support assertion expressions on collections? Is it guaranteed that collections remain immutable in assertion expressions? May universal quantifications be expressed in the expression language? May existential quantifications be expressed in the expression language?
- *AAS-3 (Additional expressions)*: Does the assertion expression language have additional features? Are these additional features guaranteed to be side effect free?

Support for Behavioral subtyping (SBS): Most systems that provide assertion support for classes, will also take inheritance and interfaces into consideration.

- *SBS-1 (Interfaces)*: Is it possible to specify contracts for interfaces? May contracts be added for classes implementing assertion-enriched interfaces?
- *SBS-2 (Correctness I)*: Does the system impose any restrictions on subcontracts? Does the system ensure that preconditions may only be weakened? Does the system ensure that postconditions may only be strengthened?
- *SBS-3 (Correctness II)*: Does the system impose stronger requirements on subcontracts as specified in SBS-2? Does the system ensure, that the correctness rules for behavioral subtyping [Liskov94] are not violated?

Runtime monitoring of assertions (RMA): Runtime monitoring of assertions is very important.

- *RMA-1 (Contract violations)*: Is an exception handling mechanism available in case of violations of assertions? Are there additional features available for dealing with assertion violations (e.g., log files)?
- *RMA-2 (Configurability)*: Is it possible to enable and disable precondition checking, postcondition checking and invariant checking selectively? Is it possible to enable and disable assertion checking on a package, class or even method level?
- *RMA-3 (Efficiency)*: Are there any additional memory requirements even when assertion checking is disabled? Is there any additional processor usage even when assertion checking is disabled?



3 OVERVIEW OF SELECTED SYSTEMS

In this section we give an overview of systems that provide support for contract-based software development for Java. We selected approaches, that are at least available as prototypes, i.e., are not mere paper work. Approaches that use algebraic techniques or higher order logics are not considered. Therefore Systems like JML [Leavens99] are not considered in this section.

In this overview section we provide for each system its name, the kind of approach used (programming language, programming language extension, preprocessor, metaprogramming), an overview, special features and references.

Biscotti:

- *Type of support:* Java language extension; builtin compiler support
- *Overview:* Biscotti concentrates on enabling behavioral specification of Java RMI interfaces by introducing additional keywords.
- *Special features:* There are no changes necessary to the java virtual machine, as Biscotti uses reflection facilities to make preconditions, postconditions, and invariants visible at run time.
- *References:* [Cicalese99]

Java Assertion Facility (JAF):

- *Type of support:* Builtin since Java 1.4 release
- *Overview:* The Java 1.4 release adds an additional keyword `assert` as well as some configuration flags to the compiler respectively java virtual machine. Java only supports a very simple assertion mechanism that allows to formulate correctness conditions within methods.
- *Special features:* Assertion checking can be easily enabled and disabled and traces of assertions may be eliminated completely from class files.
- *References:* [Sun02], [Rogers01a], [Rogers01b]

ContractJava:

- *Type of support:* ContractJava is a typical preprocessor generating Java code.
- *Overview:* The preprocessor supports the usual kind of preconditions, postconditions and assertions.
- *Special features:* ContractJava supports behavioral subtyping in the sense, that the elaborator checks, whether a type really is a behavioral subtype of another type.
- *References:* [Findler01]

iContract:

- *Type of support:* Preprocessor that generates Java code that is passed to the standard Java compiler after preprocessing.
- *Overview:* Assertions are described as documentation for a method or class (for invariants) using special documentation tags. These documentation tags are extracted by the preprocessor and converted to Java code. An iContract program is a well-formed Java program and can therefore be translated directly by a Java compiler.
- *Special features:* iContract supports operations on collections and provides additional tools for documenting the assertions in a javadoc-like fashion (iDoclet) and for facilitating the redesign of Java classes (iDarvin).
- *References:* [Kramer98], [Enseling01]

Jass:

- *Type of support:* Preprocessor that generates Java code that is passed to the standard Java compiler after preprocessing.
- *Overview:* Assertions are specified by means of a specific comment for methods or classes. A Jass program is a well-formed Java program and can therefore be translated directly by a Java compiler.
- *Special features:* Jass supports universal and existential quantification that range over finite sets. The Jass preprocessor tries to detect violations of subtyping relationships, although there is not a sound theory underlying it (as is the case for ContractJava). Additionally Jass supports so called trace assertions to specify the dynamic behavior of a class.
- *References:* [Bartezko01]

Jcontract:

- *Type of support:* Preprocessor that generates Java code that is passed to the standard Java compiler after preprocessing.
- *Overview:* Assertions are specified by means of a specific comment for methods or classes. JContract makes use of the assert command available in Java 1.4.
- *Special features:* Jcontract supports universal and existential quantification of collection types. Additionally, Jcontract is tightly integrated with the test tool Jtest [Parasoft02a]. Jtest examines the specification information contained in the contract, then creates and executes test cases that test whether the class functions as specified. Jcontract includes a javadoc generation tool, that allows to generate API documentation containing the specified assertions.
- *References:* [Parasoft02b]



Handshake:

- *Type of support:* Provides assertion support by means of reflection
- *Overview:* Contracts are specified in separate contract files. The usual kind of preconditions, postconditions and assertions may be specified. At class load time the handshake system combines this contract file with the original class file, i.e., the handshake system dynamically changes the bytecode of the class.
- *Special features:* Due to the dynamic nature of Handshake, contracts can be added to classes where the source code is not available but only the bytecode.
- *References:* [Duncan98]

jContractor:

- *Type of support:* jContractor is a pure library-based approach which utilizes the metalevel information found in Java class files and uses dynamic class loading to perform reflective on-the-fly bytecode modification.
- *Overview:* Contract code (preconditions, postconditions and invariants) is added to a class by means of methods following specific naming conventions.
- *Special features:* Inside a method's postcondition it is possible to check the result associated with the method's execution with a specified result.
- *References:* [Karaorman96]

4 EVALUATION OF CONTRACT SUPPORT

In this section we will evaluate the systems, roughly described in the previous section. The emphasis of this evaluation is to provide a consistent evaluation of the criteria developed in section 3. A quantitative assessment of the criteria is omitted. Nevertheless for sake of simplicity we partly use an ordinal scale to express the evaluation result for a criteria. The possible values for a criteria are:

- 1: excellent support
- 3: reasonable support
- 5: poorly supported
- ns: not supported

The evaluation result is presented in a tabular way. In order to make the results easily readable, the evaluation result is split into three groups. Each group contains the evaluation results for one type of system. There is one table for systems that provide assertion support builtin in the programming language or by direct compiler support. Another table describes the evaluation results for systems that provide assertion support

by means of preprocessing. A third table shows the results for systems that provide assertion support by means of metaprogramming or reflection. For readability reason the evaluation table for each type of system is split into two tables, i.e., one table dealing with the basic assertion support criteria (BAS) and the advanced assertion support (AAS), and the second table listing the results for support for behavioral subtyping (SBS) and the runtime monitoring of assertions (RMA).

Each table is accompanied by a comment section that tries to explain the evaluation results in the table.

BAS and AAS criteria (builtin systems)

System	BAS-1	BAS-2	BAS-3	AAS-1	AAS-2	AAS-3
Biscotti	ns	3	1	3	ns	ns
JAF	1	ns	ns	ns	ns	ns

Biscotti does not guarantee, that methods used in assertion expression do not change the state of the object. Boolean implications are not supported by Biscotti.

JAF provides very basic assertion support only. Preconditions, postconditions and invariants are not directly supported.

SBS and RMA criteria (builtin systems)

System	SBS-1	SBS-2	SBS-3	RMA-1	RMA-2	RMA-3
Biscotti	1	1	ns	1	ns	ns
JAF	ns	ns	ns	1	3	3

Biscotti does not allow explicit enabling or disabling of assertions. Therefore assertions checking is always enabled and influences execution speed as well as memory usage.

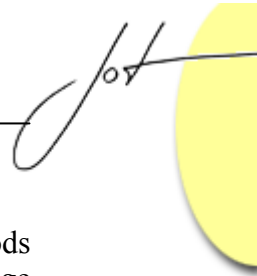
JAF allows to enable and disable assertions on a package or class level. As *JAF* does not support true preconditions or postconditions but only basic assertions, this granularity is reasonable. Using the conditional compilation idiom of Java (see Java language specification, chapter 14.19) [Gosling96] all traces of assertions can be removed from class files by an optimizing compiler. As this idiom leads to scattered code we consider the support to be reasonable, but not excellent.

BAS and AAS criteria (Preprocessors)

System	BAS-1	BAS-2	BAS-3	AAS-1	AAS-2	AAS-3
ContractJava	ns	3	1	ns	ns	ns
iContract	ns	3	1	1	1	ns
Jass	1	3	1	3	1	1
Jcontract	1	3	1	1	1	ns

ContractJava allows method calls in assertion expressions but the system does not ensure, that the methods called do not change the state of the object.

iContract allows method calls in assertion expressions but the precompiler does not ensure, that the methods called do not change the state of the object. *iContract* supports operations on Java collections and provides universal and existential quantification, too.



Jass allows method calls in assertion expressions without ensuring, that the methods called do not change the state of the object. Side effects may be reduced by using change lists in postconditions. A change list allows to specify which instance variables of this object may be changed. *Jass* does not allow Boolean implications but supports access to the original values of instance variables. The developer is responsible for proper cloning of the object. Access to the original values of parameters is not possible. As already mentioned, *Jass* provides change lists and loop invariants. In addition *Jass* allows to formulate so called trace assertions that allow to specify valid sequences of method calls.

Jcontract allows assertions in the implementation of a method but does not ensure that methods used in assertions do not change the state of the object. *Jcontract* allows to formulate boolean implications and allows the evaluation of arbitrary expressions at method entry that may be used in postconditions (old values). *Jcontract* supports universal and existential quantifications on Java collections.

SBS and RMA criteria (Preprocessors)

System	SBS-1	SBS-2	SBS-3	RMA-1	RMA-2	RMA-3
ContractJava	1	ns	1	ns	ns	ns
iContract	1	1	ns	1	1	1
Jass	ns	1	5	1	1	1
Jcontract	1	1	ns	1	1	1

ContractJava supports behavioral subtyping – it's the major contribution of this work to ensure the correctness of assertions in subtyping situations. The preprocessor is in a prototype stage and does not offer build options for assertion checking.

iContract supports fine-grained control of assertion checking by excluding packages, classes or single methods from assertion checking. Source code that contains assertions remains fully compatible with Java without any performance penalties.

Jass considers behavioral subtyping. An abstraction function has to be provided by the implementer, which must return a reference of the type of the superclass. The returned instance must be in an abstract state that is mapped from the concrete state through the abstraction function. This support is rather complicated to use and time consuming. Additionally not the entire hierarchy is considered but only the superclass of a class. The type of assertion checks enabled can be configured easily by means of flags, passed to the preprocessor. As *Jass* is a typical preprocessor, any traces may be excluded.

Jcontract supports contracts in Java interfaces and ensures that preconditions are weakened and postconditions are strengthened. *Jcontract*, by default, throws exceptions in case of assertion violations, but also allows to store such results in log files. Additionally, arbitrary handlers for dealing with assertion violations may be implemented and integrated. As *Jcontract* is a typical preprocessor, any traces may be excluded.

BAS and AAS criteria (Reflective systems)

System	BAS-1	BAS-2	BAS-3	AAS-1	AAS-2	AAS-3
Handshake	ns	3	1	3	5	ns
iContractor	ns	3	1	5	5	ns

Handshake does not ensure side effect free usage of methods in assertions. While Boolean implications are not supported, unlimited access to values of instance variables or parameters at method entry in postconditions is possible. There is no special support for dealing with collections.

jContractor also does not ensure side effect free usage of methods in assertion expressions. Boolean implications are not supported and references to values of parameters at method entry are not supported, only values of instance variables at method entry. Assertion expressions in *jContractor* may be arbitrary expressions, i.e., operations on Java collections are possible – a specific syntax for dealing with universal and existential quantification is not provided.

SBS and RMA criteria (Reflective systems)

System	SBS-1	SBS-2	SBS-3	RMA-1	RMA-2	RMA-3
Handshake	1	1	ns	1	1	3
<i>jContractor</i>	1	1	ns	1	5	5

Handshake forces to specify assertions separate from the implementation of the class in a specific syntax. The Handshake system transforms the assertion specification into Java code and weaves it into the original byte code. Assertion checking can therefore be selectively be controlled by commenting and uncommenting assertions or by entirely replacing assertion specifications. Unfortunately enabling and disabling is not supported by compiler flags.

jContractor provides full support for contracts in interfaces. *jContractor* offers two opportunities for runtime instrumentation: (1) by using a special class loader responsible for instrumentation or (2) a factory style instantiation of classes. In the latter case assertion checking can be enabled at the class level, in the first case, assertion checking can be enabled and disabled globally, i.e., for one virtual machine, only. Assertions, i.e., preconditions, postconditions, invariants, are ordinary Java methods following specific naming patterns. These specific methods can either be implemented in the class to be instrumented or in a separate class using – once again – a specific naming pattern. Assertions specified in separate classes do not have impact on the runtime performance.

5 CONCLUSION

Although Sun Microsystems added some support for assertions with the introduction of the Java Assertion Facility in Java 1.4, the support available is not sufficient to formulate preconditions, postconditions and invariants in a contract style. Most available third party products allow to formulate common preconditions, postconditions and invariants, some of them offer additional support for dealing with collections. Systems, implementing assertion support based on preprocessing have the major advantage, that the Java source code can be compiled with a standard Java compiler and that in every system any runtime penalties can be omitted when assertion checking is disabled. ContractJava and Jass are



interesting because of their special support for ensure behavioral subtyping (ContractJava) and for the additional support for trace assertions and change lists (Jass). For companies interested in applying assertion techniques in everyday projects, Jcontract seems to be a suitable choice at present, as it sufficiently supports preconditions, postconditions and invariants and additionally provides tool support for automatically generating test cases from assertion specifications.

REFERENCES

- [Bartezko01] Bartezko D., Fischer C., Möller M., Wehrheim H.: „Jass – Java with Assertions“, Electronic Notes in Theoretical Computer Science, Proceedings of RV 01, Paris, France, Volume 55, Issue 2, July 2001
- [Cicalese99] Cicalese C.T.T., Rotenstreich S.: “Behavioral Specification of Distributed Software Component Interfaces”, IEEE Computer, July 1999
- [Duncan98] Duncan A., Hölzle U.: „Adding contracts to Java with handshake“, Technical Report TRCS98-32, The University of California at Santa Barbara, December 1998
- [Enseling01] Enseling O.: “iContract: Design by Contract in Java”, JavaWorld, November 2001, see http://www.javaworld.com/javaworld/jw-02-2001/jw-0216-cooltools_p.html (last visited: March 2002)
- [Findler01] Findler R.B., Felleisen M.: „Contract Soundness for Object-Oriented Languages“, in Proceedings of OOPSLA 2001, ACM, 2001
- [Gosling96] Gosling J., Joy B., Steel G.: “The Java Language Specification”, Addison-Wesley, 1996
- [Hoare72] Hoare C. A. R.: "Proof of Correctness of Data Representations", Acta Informatica, Vol. 1, 1972, pp 271-281
- [Karaorman96] Karaorman M., Hölzle U, Bruno J.: „jContractor: A reflective Java library to support design by contract“, in Proceedings of Meta-Level Architectures and Reflection, Lecture Notes in Computer Science (LNCS), Volume 1616, Springer International, 1996
- [Kramer98] Kramer R.: "iContract - The Java Design by Contract Tool", Proceedings of TOOLS USA '98 conference, IEEE Computer Society Press, 1998
- [Leavens99] Leavens G.T, Baker A.L., Ruby C.: “JML: A Notation for Detailed Design”, In Haim Kilov, Bernhard Rumpe, and Ian Simmonds (editors), Behavioral Specifications of Businesses and Systems, chapter 12, pages 175-188. Copyright Kluwer, 1999.

- [Liskov94] Liskov B.H., Wing J.M.: “A Behavioral Notion of Subtyping”, ACM Transactions on Programming Languages and Systems, November 1994
- [Meyer97a] Meyer B.: "Object-Oriented Software Construction", 2nd Edition, Prentice Hall, 1997
- [Meyer97b] Meyer B., Jézéquel J.M. "Design by Contract – The Lessons of Ariane", IEEE Computer, Vol. 30, No. 2, January 1997
- [Parasoft02a] Parasoft: “Using Design by Contract to Automate Software and Component Testing, see http://www.parasoft.com/jsp/products/tech_papers.jsp?product=Jcontract (last visited: July 2002)
- [Parasoft02b] Parasoft: “Automatic Java Software and Component Testing: Using Jtest to Automate Unit Testing and Coding Standard Enforcement”, see <http://www.parasoft.com/jsp/products/article.jsp?articleId=839&product=Jtest> (last visited: July 2002)
- [Plösch97] Plösch R.: “Design by Contract for Python”, Proceedings of joint APSEC'97, ICSC'97 conference, December 2-5, 1997, Hong Kong, IEEE Computer Society Press, pp 213-219
- [Rogers01a] Rogers P.: “J2SE 1.4 premieres Java’s assertion capability” – Part 1, JavaWorld, November 2001, see http://www.javaworld.com/javaworld/jw-11-2001/jw-1109-assert_p.html (last visited: March 2002)
- [Rogers01b] Rogers P.: “J2SE 1.4 premieres Java’s assertion capability” – Part 2, JavaWorld, November 2001, see http://www.javaworld.com/javaworld/jw-12-2001/jw-1214-assert_p.html (last visited: March 2002)
- [Sun02] Sun Microsystems: Java Assertion Facility - <http://java.sun.com/j2se/1.4/docs/guide/lang/assert.html> (last visited: March 2002)
- [Templ94] Templ J.: “Metaprogramming in Oberon”, ETH Dissertation No. 10655, Zurich, 1994



About the author



Reinhold Plösch is Assistant Professor at the Johannes Kepler University of Linz. His research interests include reliable components and mobile agents. He can be reached at reinhold.ploesch@jku.at.