

# Design By Contract In Java

Raymond Shanley

September 2003

## Abstract

Design by contract<sup>1</sup> is an effective means of improving the quality of software, normally object-oriented, by annotating code with pre-conditions, post-conditions and invariants. Despite its benefits, most mainstream languages do not yet have built-in support for it - Java<sup>2</sup> is one such example. There are tools available, however, which make it possible to use design by contract in Java. This article examines what DBC<sup>3</sup> is and shows how it can be used in Java with the available tools. It then presents a set of criteria for evaluating DBC tools and uses it to evaluate those Java specific. Finally, it identifies potential types of DBC users and evaluates which system is most suitable for each user type.

## 1 Introduction to Software Quality

Design by contract is one technique which is claimed can improve the quality of software produced. This article will take a look at the meaning of quality. We need some sort of definition because “quality” is just a word that can be interpreted differently by different people. A good start would be to break it down into sub-components. Several definitions have already been created by international standards bodies (ISO, IEEE) for the term.

At the most basic level the measurement of software quality can be split into external and internal factors (see [Meyer 1997] chapter 1), whereby internal factors are those visible only to the developers of the software with access to the source code. An example of an internal factor in an object-oriented context is how loosely coupled the objects in the system are [Chidamber 1991]. External factors are those visible to end users of the software, which at the end of the day are by far the most important. The end user is only concerned with external factors and if the product is of low quality from this point of view it is of little consolation to the user if you tell him that an internal factor such as modularity is excellent. An example of an external factor is correctness - whether the software does what the specification says it should do. Correctness will be mentioned in more detail later in the article. Although the external factors are the ones that matter most, there is normally some relationship between how well the software measures up to internal quality factors and how it scores on measures of external quality.

In 1968 a NATO conference took place on the topic of “software engineering” [Naur 1969] - this was the first time the term was used - where a call was made for an improvement in the quality of software available. It was suggested that the rigour of other engineering disciplines should also be present in the production of software. Since that date there has been much work done on how to improve software quality. Edsger W. Dijkstra’s work on structured programming [Dahl 1972], in the 1970’s, was a very significant step forward for example, as was Niklaus Wirth’s work on stepwise refinement [Wirth 1971]. C.A.R. Hoare’s work on program correctness [Hoare 1972] was also very important and relates directly to the subject of this article.

The ISO 9126 standard quality model [ISO 2001] breaks quality into the categories: functionality, reliability, efficiency, usability, maintainability and portability. The categories that can be

---

<sup>1</sup>Design By Contract is a registered trademark of ISE Inc.

<sup>2</sup>When the word Java is used in this paper it normally covers both the Java Programming Language and the Java Platform. Java is a registered trademark of Sun Microsystems Inc.

<sup>3</sup>In this paper the term design by contract is often shortened to DBC for convenience.

directly improved by implementing a design by contract strategy are: functionality, reliability and maintainability. The other categories - efficiency, usability and portability - only benefit indirectly from the use of DBC. How it achieves this is looked at in section 3. The component of quality that can potentially be improved most by making use of DBC is reliability, so this will be described in more detail now.

## 1.1 Reliability

The ISO 9126 standard defines reliability as “A set of attributes that bear on the capability of software to maintain its level of performance under stated conditions for a stated period of time”. This definition implies that reliability is a measure of the number of faults produced per unit of time in a running system. But reliability can also be seen in another way as a combination of correctness and robustness [Meyer 2001].

## 1.2 Correctness

Correctness cannot be talked about in relation to a program alone - you cannot say a program is correct per se. Correctness is a mapping from a specification to an implementation.

Often the specification is vague or not even present in a persistent form (e.g. it is only in someone’s head). This is not good enough for the development of reliable software. The idea of DBC is to link the specification directly to the implementation and allow you to find out at run time or through static analysis if the two do not match. It achieves this by including part of the specification directly along with the code.

## 1.3 Robustness

Robustness is the ability of software systems to react appropriately to abnormal conditions [Meyer 1997]. Abnormal conditions are those cases which are not covered by the specification. A program which is reliable should be able to recover from unexpected conditions and not just crash straight away. The Java exception mechanism, for example, makes it easier to increase a program’s robustness.

## 1.4 Formal Methods

“The required techniques of effective reasoning are pretty formal, but as long as programming is done by people that don’t master them, the software crisis will remain with us and will be considered an incurable disease.”

Edsger W. Dijkstra [Dijkstra 2000]

The field of formal methods has been developing steadily over the years [FM 2003, Gannon 1995], and is really the field from which the ideas behind DBC came. Formal methods are operationally based approaches to specifying and developing computer software. As they are operationally based, the notion of change is not present. This is a key difference to imperative approaches (normal programming languages). When extremely high reliability is required of software, formal methods are the technique of choice. There are many formal approaches available. For more details see [Gibson 1993, Spivey 1992].

But if formal methods are effective at producing very reliable software, why aren’t they used more widely? There are a number of reasons. First of all the cost in terms of time and effort is very high, even for relatively small systems. Secondly, the expertise required to practice formal methods is very high. Practitioners must be mathematically adept and this actually rules out a big percentage of practitioners who currently develop software. It does not make economic sense in most cases to aim for producing high quality software such that would necessitate a formal approach. The market forces simply do not demand it. It is interesting to note that the means to produce high quality software are in existence today but are just not used very widely mainly

for this reason of economics. In the earlier days of the industry the situation was different in that these techniques were not available.

For these reasons formal methods are really only used in critical software which could cause for example loss of life if failure occurred. Examples of places where extremely high quality is called for, and thus formal development, is in control systems for aircraft and driverless trains [Behm 1999]. So while software quality is commonly lamented, it is perhaps not so difficult to see why it is so. One research area which is aiming to solve the problem with the quality of rapidly produced software is that of so-called trusted components [Meyer 2003].

While it often does not make economic sense to use a fully formal approach, many of the benefits of going in this direction can be gained by using DBC.

## 2 What DBC Is

Design by contract is a style of programming where the behaviour of interfaces between classes are specified with assertions. There are three main types of assertions:

- Method pre-conditions
- Method post-conditions
- Class invariants

There are also three further kinds of assertions, for use with loops and internally in method bodies:

- Loop variants
- Loop invariants
- Basic assertions / “Check” expressions

All these assertions can be turned either on or off at runtime, usually to different levels depending on the implementation. For example, it can be set so that only pre-conditions in application code are checked, and no assertions in library code are checked. Typically, at the testing stage they are turned on full (all assertions checked) and at production stage they are turned off completely or to a low level (only pre-conditions checked).

### 2.1 Small Example

Here is a short example of the use of design by contract given in Java/Jass syntax where a pre-condition for a `setAge()` method specifies that the argument must be greater than zero:

```
public void setAge(int age) {
    /** require age > 0; */
    this.age = age;
}
```

In the above example, the extra contract information can be seen as extending the type system. It has been specified that, as a pre-condition (`require`) to the method working properly, the parameter should not only be of type integer but it should also be a positive integer. The difference between pre-condition and type is normally that the type is checked at compile time and the pre-condition only at runtime [Joyner 1999]. Note that Jass and other tools for adding DBC to Java will be studied later in this article.

## 2.2 Division of Responsibilities

Use of design by contract leads to a clear separation of responsibilities between callers of methods and the methods themselves. Following is a table [Meyer 2001] showing the mutual benefits that *clients* and *suppliers* get from each other due to the use of contracts. In the terminology a client is a class that calls a method. A supplier is the class which owns the method being called.

	Obligations	Benefits
Client	Satisfy precondition	From postcondition
Supplier	Satisfy postcondition	From precondition

Responsibilities are clearly divided between the client (user of a class) and the supplier (the class itself, providing the functionality to the client). The client's responsibility is to ensure it satisfies the pre-conditions layed out. The supplier's benefit here is that it must not perform any checks, for example, on the parameters - it can just assume the pre-conditions have been satisfied. The supplier must ensure that the post-conditions are met (but only if the pre-conditions were met). The client can then assume the post conditions have been satisfied and must not check whether this is in fact the case.

Additionally, class invariants are used to specify unchanging characteristics of a class that must always remain true at all *stable times* (see later). It can be said that they capture the essential properties of the class<sup>4</sup>.

## 2.3 What It Isn't

Design by contract should not be confused with the so-called defensive programming approach which encourages attempting to make methods as robust against wrong inputs as possible by adding checks to the actual program code itself. The defensive approach increases complexity by increasing the number of execution paths through the code and creates the need for complex error/return code schemes. When looked at from the point of view of a single method, the defensive approach looks good. When looked at from the system-wide point of view, defensive programming can be seen to increase complexity. DBC helps avoid these problems and helps to simplify system architecture.

Another misunderstanding to be avoided is that assertions are a user input checking mechanism - they are not. As Meyer says, "make sure to note that each of the contracts discussed holds between a routine (the supplier) and another routine (its caller): we are concerned about software-to-software communication, not software-to-human or software-to-outside-world" [Meyer 1997]. This implies that user input should still be checked in a manner similar to defensive programming mentioned above.

## 2.4 Further Discussion

As mentioned above, class invariants must be true at all stable times. What constitutes a stable time? The answer is at public method entry and public method exit. During execution of the method body the invariant is allowed to be temporarily broken once it is restored before method exit. Likewise, when a private method is called from a public method, the invariant does not necessarily have to be satisfied. Pre-conditions must be satisfied at the time of method entry, and post-conditions must be satisfied at the time of method exit.

Referring back to the concept of correctness, in the view of design by contract a class is correct when its implementation is consistent with all pre-conditions, post-conditions and the invariant [Meyer 1997].

---

<sup>4</sup>Note that the discussion in this section is a simplification of the topic but a more detailed treatment is beyond the scope of this paper.

## 2.5 Mathematical Background

The idea of using preconditions, postconditions and invariants in object-oriented programming languages has a very solid foundation in the theory of Abstract Data Types (ADTs) (see [Meyer 1997] chapter 6).

An ADT is a notation for mathematically specifying a type. As an example consider the following ADT which specifies an integer type:

```
TYPE integer SORTS integer, boolean
OPNS
  0 :-> integer
  succ: integer -> integer
  eq: integer, integer -> boolean
  +: integer, integer -> integer
EQNS forall x,y: integer
  0 eq 0 = true; succ(x) eq succ(y) = x eq y;
  0 eq succ(x) = false; succ(x) eq 0 = false;
  0 + x = x; succ(x) + y = x + (succ(y));
ENDTYPE
```

The section `OPNS` is the operations section - it specifies what functions are available in the type, what their arguments are and what types they return. For example, the `eq` function takes two integer type arguments and returns a boolean type. This could be given the semantics that it compares two integers and returns true or false. The `EQNS` (equations) section specifies the axioms of the type. These can be interpreted as rewrite rules that can be mechanically followed such that valid strings can be derived from more basic strings. These axioms can be seen as supplying the rules for the semantics of the type. There is a striking resemblance between ADTs and classes, and that is because the idea of classes is based on the same theory as ADTs. The `OPNS` section can be seen as the method signatures in Java classes or interfaces for example - specifying the parameter types and return types. However, the `EQNS` section is missing in the traditional Java view of classes. This is where assertions come in - they correspond to the axioms in the `EQNS` section. They specify how the methods should behave and relate to each other in an operational (as opposed to imperative) way.

A useful notation when dealing with the issue of program correctness is that of Hoare triples. They take the form  $\{P\}A\{Q\}$  where  $A$  is an operation,  $P$  is a pre-condition and  $Q$  is a post-condition. It can be interpreted as any execution of  $A$  starting in a state where  $P$  holds, will terminate in a state where  $Q$  holds [Meyer 1997]. So  $P$  can be seen as the pre-condition and  $Q$  as the post-condition.

## 2.6 DBC Style

In this section the style of programming that is recommended when practicing design by contract is discussed. This style is not enforced in any way, but it has been found by experience to be the best practice of programming by contract.

First of all there is a separation made between methods which are said to be “queries” and methods which are said to be “commands”. A query is a method which returns a value. A query must not cause any side-effects, i.e. cause any change in the state of an object. A command is a method may which change the state of an object but does not return a result. An important principle of DBC is to distinguish carefully between queries and commands [McKim 2002]<sup>5</sup>. This simplifies the writing of assertions and also allows the use of queries as part of assertions because they do not cause side-effects. It also enables making up for some of the shortcomings of many implementations of DBC - they don’t support full predicate logic. For example, there is no way to assert that the top item should have been removed from a stack by a `remove()` command so an `item_at()` query is created and this is used in the post-condition of `remove()` to achieve the desired

---

<sup>5</sup>In Java, however, combination of query and command is allowed and is used quite often.

assertion. This sort of usage is given as a guideline that says that command post-conditions should say how the command affects queries.

There are varying degrees to which a programmer can specify a program using DBC. How rigorously he wants to carry this out depends on how critical the piece of code is and other factors such as time constraints. For critical parts of systems it is recommended to go as far as including assertions on what does not change - these types of assertions are called frame rules. They are in contrast to the more common type of assertions, which describe what changes. For example when adding an item to a queue the programmer normally asserts that the count increases by one, but doesn't assert what stays the same (for example the order of the items).

## 3 How DBC Helps Software Quality

### 3.1 Less Debugging Effort

According to a Caper Jones survey [CF 1998], on average 85% of the time taken to develop software is spent in defect removal. When the DBC approach is used, the programmer will spend more time writing contracts but less time debugging [Meyer 1997]. There will be less time spent searching for the bugs because in runtime tests, contract failures pinpoint the exact point and cause of the problem. As the most difficult part of debugging is finding where the fault lies and what caused it, using the DBC approach will save time [McKim 2002].

### 3.2 Increased Reliability

The main benefit put forward by design by contract is that of making it easier to produce more reliable software. As an example of this consider the following case. On the 4th of June 1996 the maiden flight of the Ariane 5 rocket crashed around 40 seconds after take-off. The cause of the \$500 million disaster was found to be a software bug. An article appearing in IEEE [Jézéquel 1997], written by Meyer and Jézéquel analyzes the problem and shows how design by contract could have prevented this particular disaster. In a letter to the editor of IEEE responding to the article Tom DeMarco made the following comment:

“I believe that the use of Eiffel-like module contracts is the most important non-practice in software today. By that I mean there is no other candidate practice presently being urged upon us that has greater capacity to improve the quality of software produced. As Jezequel and Meyer point out, this same sort of contract mechanism is the sine-qua-non of sensible software reuse.”

Tom DeMarco

According to the article the essence of the problem was that a 64-bit floating point value was automatically converted to a 16-bit value as an argument to a method calculating the flight's "horizontal bias". The problem could have been easily caught if there had been a pre-condition on the method stating that the value had to be able to fit in 16-bits. It was a piece of code that had been used in Ariane 4, and the developers had made the assumption that because it worked then it had to work now. When design by contract is used the interface states what assumptions can be made.

### 3.3 Better Documentation

One benefit of using a design by contract approach is that of trustworthy documentation (see pp.13 of [McKim 2002]). An ever-present problem of traditional software development has been how to keep documentation up-to-date and corresponding with the actual software text. Java has an excellent Javadoc tool which encourages developers to insert comments for methods, classes and interfaces directly in the code. This documentation can then be extracted and formatted by

the tool suitable for easy viewing. However, the problem still exists whereby if the comment is out-of-date (i.e. the code has changed since the comment was written and the comment itself has not been updated) the Javadoc tool can't help. This problem is overcome with the use of design by contract because the comment is a formal specification which is checked against the code at runtime. The two must correspond because the programmer has no choice but to keep the two synchronised - otherwise a runtime error will result during testing showing the discrepancy.

### **3.4 Better Maintainability**

As a consequence of better documentation, a product also becomes more maintainable with the use of DBC.

### **3.5 Reduced Testing Effort**

Kramer names one of the advantages of DBC as reduced test effort [Kramer 1998]. When writing tests, the engineers must figure out beforehand what the expected results of the tests are. But when the contract is explicitly specified the expected results are already apparent in most cases. So this aspect of testing requires less time. In relation to unit testing it is possible to use contract information to automatically generate test oracles. This topic is returned to later in this article.

### **3.6 Complexity Control**

No matter what technique is being used to improve the quality of code, it often may seem somewhat excessive for use on very small projects. Often the advantages of assertions only become apparent during development of larger projects. It can be seen as controlling the complexity of the application being built. The interactions between different parts of the program are specified precisely and problems can be caught straight away. The programmer will find that the code will simplify in comparison to a traditional approach. Many conditional checks will no longer be necessary and error return codes will also no longer be required.

As an aside, the argument also exists that the contract approach has benefits for small programs. It is argued that a successful small program will be re-used and therefore it should have a precise contract associated with it.

### **3.7 Methodological Benefits**

The design by contract approach forces the programmer to think through exactly what is required before starting to write program code. This is a practical advantage from the point of view of the methodology of programming. The programmer is more likely to understand the problem more clearly which immediately reduces the probability of bugs. It can be seen as bringing about a change in the process of programming. Good programmers that have never heard of DBC often think implicitly in this way - they think about what can be assumed to be true at points in a program, what conditions must be satisfied before calling a method, and what services the method must provide. DBC can be seen as making this process explicit, more formal and more reliable.

## **4 DBC in Java**

It is possible to simulate preconditions, postconditions and invariants using the standard Java language [Kramer 1998] but this is rather unwieldy and loses some of the benefits. To give an example of why its unwieldy, imagine you want to add an invariant to a class. To ensure that the invariant is checked you need to write a method for checking the invariant which throws an exception, and then add a method call to the start and end of each method in the class. Examples of the benefits you lose here are loss of a clear separation of the contract specification from the implementation code, being unable to switch the runtime checks on and off, and problems with preserving contracts when subtyping classes.

Inclusion of design by contract in Java was considered while the language was still under development [Gosling 1994] but due to time constraints it was not included in the final specification [Gosling 1996]. Since the release of Java 1.4 there has been an `assert` keyword in the Java language which allows the programmer to add basic assertions to code [JavaDoc 2003]. The JVM also supports turning the assertions on and off at different levels of granularity. This is only a small step in the right direction however, as full design by contract is not supported. If it is not possible to write pre-conditions, post-conditions and invariants then a system cannot be regarded as supporting design by contract for the reasons mentioned in [Kramer 1999].

There are many add-on systems available for adding full DBC facilities to Java. The systems looked at in this article are iContract, Jass, JContractor, JMSAssert, Kopi, ESC/Java and JML. There are other systems available that are not looked at in this article including DBC by Handshake [Duncan 1998] and the commercial Parasoft JContract [Parasoft 2003]. They were not looked at due to time constraints.

The primary disadvantage of most of these approaches is that you must run a preprocessor on the source files before compiling. This extra indirection and complexity in the process of building a program may discourage its use in many real-world projects (for legitimate reasons it must be said), unless the advantages to be gained are well known by those involved. Other approaches seen here are to use a library based solution, or to write a custom, extended compiler.

Now an introduction will be given to each of the contract systems mentioned above. In a later section their respective merits will be evaluated.

## 4.1 iContract

Appearing in 1998, iContract [Kramer 1998] was one of the first tools for adding DBC facilities to Java and it is also freely available. Contract information appears inside Java comments. iContract has keywords in the style of UML's OCL [Kleppe 1999] i.e. `pre`, `post` and `invariant`. When the iContract pre-processor is run it parses the Java code and comments and then generates "instrumented" Java code. This means that the contract information appearing inside Java comments are turned into Java assertion-checking code by the iContract pre-processor. The instrumented Java files are then compiled into class files.

As an example of the syntax consider a class designed to represent a board for the game of X's and O's. The constructor could look as follows where constraints are set on the values of the parameters and the board array is specified as not being null after the constructor has finished:

```
/**
 * @pre xdim > 2
 * @pre ydim > 2
 * @post board != null
 */
public X0Board(int xdim, int ydim) {
    this.xdim = xdim;
    this.ydim = ydim;
    board = new char[xdim][ydim];
}
```

Note that because contract information appears inside comments the Java code can still be compiled with a regular Java compiler. One of the advantages of iContract is that there is support for it built into Apache Ant [Ant 2003] - the most popular make tool for Java. What this means is that a developer can install iContract and Ant and then add a so-called task to Ant which will automatically call iContract for creating instrumented code, which has the effect that the extra indirection in the process is transparent once set up.

A side benefit of iContract is that there are other supporting tools available such as iControl, and iDoclet. iControl provides a graphical user interface for modifying set-up options such as specifying which files have contract information and which don't. This is useful in a team environment where not all developers working on the same project use iContract. iDoclet is a tool which adds the capability of displaying the assertion information as part of the JavaDoc documentation generated.

iContract has support for universal quantification (`forall`), existential quantification (`exists`) and implication (`implies`). Although `implies` is not strictly essential as `A implies B` can be rewritten as `!A || B`, it is useful to have for better readability. Originally quantification only worked on `Enumeration` types but this was since updated to work on all collection types and arrays.

Disadvantages of iContract include that it cannot detect violations of subtyping rules in contracts [Plösch 2002], but this is also the case for most of the other tools. Another more serious disadvantage is that the precompiler cannot ensure that methods called in assertions do not have side-effects. This point actually raises serious questions about whether this tools should be used at all.

Furthermore, in relation to subtyping the programmer has to establish the rule of making sure that variables referred to in contracts are not private so that supertype contracts propagated into subtypes [Enseling 2001] can be evaluated - this could have negative implications on encapsulation properties of programs. So while the programmer gains the advantages of being able to add contracts he loses some of the advantages of encapsulation. This is a trade-off that should not have to be made in the author's opinion.

Platform independence is compromised slightly by iContract. The documentation supplies instructions for running iContract that only work on a UNIX Korn shell - the script supplied is for the mentioned shell. By using Ant this problem can be solved but this information does not appear in the supplied documentation.

In the areas of setup and use iContract could be improved. When using iContract, a number (just over four times the number of source files) of intermediate files are generated which can be seen as cluttering up the project directory. There is a directory generated for the instrumented java files, a directory for "rep" (which are a side effect of instrumenting the files), in addition to the expected directory for the end result instrumented class files. Uninstrumented class files are also placed in the source directory for some reason. Moreover, four additional files need to be placed into the project root directory: "config", "files", "targets" and "icontrol.properties". This all seems to be more complex than it needs to be. iContract also has problems dealing with inner classes and certain methods with more than one point of return.

## 4.2 Jass

Jass [Bartetzko 2001] is another freely available system for adding contracts to Java which was developed more recently than iContract as a collaborative university project. It, like iContract, is a pre-processor based system, but its syntax is more similar to the style of Eiffel with the keywords `require` and `ensure` being used and the assertions appearing inside the method bodies. Basic assertions are supported via the `check` keyword.

Jass is relatively easy to install and set up. Unfortunately, getting it to compile code can be considerably more difficult due to a number of shortcomings.

The following is a slightly expanded `XOBoard` example, which shows pre-conditions, post-conditions, class invariants and loop invariants. Loop invariants, which have not been mentioned yet, ensure that global properties of the loop are satisfied and also that termination is guaranteed.

```
public class XOBoard {
    ...
    public XOBoard(int xdim, int ydim) {
        /** require [dimesions_ok] xdim > 2 && xdim < 10
            && ydim > 2 && ydim < 10; **/

        this.xdim = xdim;
        this.ydim = ydim;
        board = new char[xdim][ydim];
        clearBoard();
        /** ensure board != null; **/
    }
    ...
    public String toString() {
        StringBuffer sb = new StringBuffer();
        for (int i = 0; i < ydim; i++)
            /** invariant 0 <= i && i <= ydim; **/
            /** variant ydim - i **/
            {
                sb.append(getLine());
                sb.append("|");
                for (int j = 0; j < xdim; j++)
                    /** invariant 0 <= j && j <= xdim; **/
                    /** variant xdim - i **/
                    {
                        sb.append(board[j][i]+"|");
                    }
                sb.append("\n");
            }
        sb.append(getLine());
        return sb.toString();
    }
    ...
    /** invariant xdim > 2 && xdim < 10 && ydim > 2 && ydim < 10; **/
}

```

There has been much work done on Jass and it has a number of nice features. For instance, it is one of the only contract checkers for Java that can offer the ability of discovering errors in the type hierarchy [Felleisen 2001]. That said however, enabling Jass to do these “refinement checks” is not ideal as it requires manual programmer intervention as follows. The programmer must make sure that the subclass implements the interface `jass.runtime.Refinement` and therefore the method `jassGetSuperState()` which should return an object of type supertype. This also has the disadvantage that full class hierarchies are not checked in one go. Only the relationship between the superclass and subclass are checked. Compliance with classes further up the hierarchy is not checked.

Jass is a system that supports universal and existential quantification but with different syntax than that seen for iContract above. Jass also guarantees that contracts are side-effect free.

The keyword `changeonly` can be used to specify what variables a method may change. An empty `changeonly` expression means the method is pure.

One concept not available in any other implementation is that of trace assertions, which allow the specification of the temporal order of events. Jass events can be, for example, the beginning and end of method invocations, written as the method name followed by `.b` or `.e`. The syntax is of trace assertions is similar to that of CSP [Bartetzko 2001].

On the down side Jass has problems dealing with inner classes. There is also no support for implication (`implies`) in assertions. Furthermore, when the author used the following `changeonly` expression and other similar expressions Jass generated Java code that could not be compiled.

```
public int getX() { return x; /** ensure changeonly {}; **/}
```

As an aside, the expression should say that the method is not permitted to change any variable value in the object. Also when the author used the following trace assertion in the `XOGame` class, Jass strangely terminated compilation with a `FileNotFoundException` exception:

```
/** invariant trace(playGame(XOPlayer, XOPlayer) -> getWinner() -> STOP); **/
```

As an aside, the assertion should specify that `getWinner()` can only be called after `playGame()` has been called. Another problem worth mentioning is that Jass is unable to add specification information to abstract methods - and this is a serious shortcoming. Another shortcoming in the author's opinion is that in order to be able to access "old" values the programmer must write a `clone()` method implementation for that class. Other tools looked at here do not require this extra work. These examples would seem to suggest that there are unfortunately still a number of bugs and shortcomings to be ironed out in the Jass implementation and it is questionable whether it is suitable for commercial use.

### 4.3 jContractor

In contrast to the previous two approaches, `jContractor` [Bruno 1998] takes a library based approach to the problem of adding contracts to Java. Features that `jContractor` has include:

- Support for predicate logic quantifiers such as `forall` and `exists` using `JaQual` (Java Quantification Library). Unfortunately these are only usable on `Collection` types and not on arrays for example.
- Library files for adding `jContractor` as a task to the Apache Ant build tool.

To use `jContractor` the programmer writes methods with pre-determined naming conventions that specify the pre-conditions, post-conditions and invariants. When specifying the pre-condition for example the programmer inserts a new method that has a signature that looks as follows:

```
protected boolean methodName_Precondition(<arg-list>)
```

whereby `methodName` corresponds to the name of the method for which the pre-condition has been written. The body of the method then contains the pre-condition checking code itself. Post-conditions are specified in the same way. Specification of class invariants also follows a similar pattern:

```
protected boolean _Invariant()
```

`jContractor` also provides expressions for referring to `old` values and `result` values as follows:

```
OLD.attr for attributes  
OLD.foo() for methods  
RESULT
```

Note that a certain amount of programmer intervention is necessary to be able to use `OLD` and `RESULT` in `jContractor`. A private instance variable of the same type as the containing class must be set up in a class wishing to use `OLD`, `Cloneable` must be implemented and a proper `clone()` method must be provided.

To use `RESULT` in a post-condition, an argument with the same type as the original method return type must be added to the post-condition method signature, as the following example shows (where `getWinner()` has an `int` return type):

```
protected boolean getWinner_Postcondition(int RESULT) {
    return RESULT == DRAW || RESULT == P1 || RESULT == P2;
}
```

For contracts to be checked at runtime (which is desirable) the jContractor creators recommend that their custom written class loader is used. This custom class loader causes the appropriate contract method to be invoked at method entry and method exit. A second option is also provided, which is a factory style instantiation using the jContractor library, but this requires extra changes to client code whereby everytime `new` is used a custom `new` method should be used:

```
Standard Java: XOBoard board = new XOBoard();
jContractor: XOBoard board = (XOBoard) jContractor.New("XOBoard");
```

Both methods (using a custom class loader and using a custom `new` call) have negative effects on the compatibility of the approach, but of the two options using the custom class loader is probably the better. However, an additional tool called jInstrument is part of the jContract distribution whose function is to add contract information directly to methods in class files so there is no need for a custom class loader or a custom `new` method.

jContract also gives the option of specifying contract information in a separate source file of the form `ClassName_CONTRACT.java`. This may be a good option to have, as otherwise a source file could become cluttered up with up to twice the number of contract related methods as original methods.

Weaknesses of jContractor include the following. Assertions are not guaranteed to be side-effect free. There is also no support for specifying methods as pure. There is no facility for specifying basic assertions within methods, due to the approach taken, and likewise there is no support for loop invariants and variants.

jContractor provides support for integrating with the Ant build tool. jContractor also proved itself easy to set up and use.

## 4.4 JMSAssert

JMSAssert [MMS 2000] takes the approach of including assertion information as part of comments as with iContract and Jass above. It is a system that only works in a Microsoft Windows environment as it depends on the use of a DLL (dynamically linked library). This DLL registers itself with the JVM and assigns assertion triggers to methods so that when the method is invoked a corresponding "JMScript" trigger method is invoked [MMS 2000]. The corresponding JMScript code is generated beforehand by running a pre-processor over the annotated Java source files. One benefit this particular approach has over the others above is that private variables and methods can be accessed - so encapsulation principles do not have to be sacrificed when coding, as mentioned above for the iContract approach.

The assertions themselves are similar to that of iContract and OCL - `@pre`, `@post` and `@inv`. Access to old values is provided by the keyword `$prev` and access to return values is provided by the keyword `$ret`. Functionality similar to quantifiers is provided by the ability to insert JMScript macros. It is not possible to specify pure methods in JMSAssert, nor is it possible to specify loop variants, loop invariants or basic assertions. Other limitations of JMSAssert include the fact that it requires a Java Classic VM - it will not run on a Hotspot VM. Also the runtime execution speed with assertions enabled is very poor in comparison to the other systems looked at in this article.

Although it is possible to specify which assertions are checked at runtime, the method of configuration is less than ideal. The user must manually comment out entries in the generated `*.jms` files. For example, if the user wishes to disable all postcondition checks, they are obliged to comment out all postcondition entries in the `jms` files corresponding to all classes in the system. This

is a tedious and error prone process. Furthermore, if the user runs the pre-processor afterwards, their changes to the configuration will be overwritten.

## 4.5 Kopi

Kopi [Lackner 2002] takes the route of writing a custom Java compiler with support directly built-in for design by contract. It is a Java compiler itself written in Java and released under the GNU public license. It essentially extends the Java language with the new keywords `@invariant`, `@require`, `@ensure` and `@assert` and the associated syntax. This means there is no need for pre-processors, libraries or special class loaders. If the Java platform itself were to be extended in future Kopi would be a good (although not perfect) prototype implementation to take a look at. It proves that it is indeed possible to effectively add DBC to Java in contrast to what Sun say in the Java documentation about the assertion facility added in release 1.4 [JavaDoc 2003].

Following is a code example to illustrate the syntax of Kopi assertions. It is a `backtrack()` method in class `XOBoard` for going one step backwards in a game:

```
public void backtrack()
@require {
    @assert lastX != INVALID && lastY != INVALID;
    @assert !isEmpty();
}
{
    decrementCharCount(new Character(board[lastX][lastY]));
    board[lastX][lastY] = SPACECHAR.charValue();
}
@ensure {
    @assert board[lastX][lastY] == SPACECHAR.charValue();
    @assert @(getXCount()-1 == getXCount() || @(getOCount()-1 == getOCount());
}
```

Kopi has support for referring to the old object state using the syntax `@@(expr)`. The return value can be referred to as `@@()`. Note that Kopi does not allow pre-conditions to be specified for constructors.

Unfortunately, there are a number of features missing from Kopi that would be desirable in an effective implementation. Assertions are not guaranteed to be side effect free and there is no way to specify a method as being side effect free. There is no support for loop variants or invariants. Furthermore there is no support for quantification and there is no `implies` operator present. On another point, although Kopi allows contract information to be added to Java interfaces it was found that it does not enable pre and post conditions to be added to abstract methods in abstract classes. One area where Kopi lacks is the readability of runtime messages - for example, it is not always clear where or why a contract has been broken. Kopi also seems to have problems dealing with contracts on inner classes. Finally, an area that could be improved is that of runtime configurability as it is not possible to compile with assertions turned off, nor specify at runtime that assertions should not be checked.

## 4.6 ESC/Java

ESC/Java (Extended Static Checker for Java) [Detlefs 1998] is the only system described so far that includes a static checker. A static checker reads a program, analyzes it, and attempts to discover potential bugs without running the program [Aho 1986]. It performs the analysis by building a simplified representation of the program and checks it against given properties, usually by model checking or theorem proving [Artho 2001]. Contract information can be used to improve the level of checking at compile time - ESC/Java takes advantage of contract information to increase the number of things that can be checked and improve the messages provided. For example if a pre-condition says that an argument is not allowed to be null then ESC won't complain that the variable could be null at some point in the method body.

The problem of finding whether the code fulfills the contract is an undecidable problem in the same way the Halting problem is, but although this means a checker can never be fully successful a limit of success can be approached. ESC may also sometimes emit warning messages that are wrong. Although ESC is inherently incomplete and unsound, it can be very useful for finding many bugs as soon as possible.

One disadvantage of ESC/Java in comparison to some others systems here is that you can't use method calls in assertions. This has the effect that primitive types must be used in the application code where it might be better to use object types, simply so that they can be referred to in the contract specifications. Another weak point of ESC is that you can't specify methods as being side-effect free. It has a `modifies` keyword which is supposed to specify what variables are allowed to be changed in a method but it is not enforced in any implementation to date. The latest release of ESC/Java for download is dated 2001 and it seems that no further development has taken place on it since then.

ESC/Java does not include any runtime checking facilities and therefore it also does not perform any compilation. Source files are compiled as normal with a Java compiler.

## 4.7 JML

JML (Java Modelling Language) [Leavens 1999] is a so-called behavioral interface specification language (BISL) for Java that includes a runtime checker, documentation generator and unit test oracle generation functionality. The idea of BISLs is essentially the same as that of DBC except they normally offer more expressiveness of syntax.

Through a collaborative effort the syntax of JML and ESC/Java have been made similar and it is at the point where JML is nearly a superset of ESC/Java. So it should be easily possible to modify a program's JML contract specification and then run the ESC/Java static checker over it, although in the author's opinion it is not very practical to do this is a real world project situation. It would be better if there were a static analyser available for JML syntax as it stands.

JML is much more expressive for specification than Eiffel, containing quantifiers, specification-only variables and other enhancements. It takes the approach of including annotations as part of Java comments so the Java code can still be compiled with a standard compiler, as with most of the approaches for adding DBC to Java considered above. JML includes packages which contain "pure" classes for use in the specification. Pure here means that they are programmed so as to be side-effect free, so they can be used in assertions without affecting the state of the computation. This means that they have a well defined mathematical meaning (in contrast to Eiffel's assertions and some of the other Java solutions discussed here which allow user defined methods to be used which are not guaranteed side-effect free) [Leavens 1999]. There is a `pure` keyword available for use that guarantees a method to be side-effect free.

JML allows specifying different pre and post conditions for normal and exceptional behaviour. It also allows specifying two or more different sets of assertions for normal behaviour. These are interesting features not present in standard DBC implementations.

Here is an example of a JML specification using the X's and O's board example:

```
public class XOBoard {
    ...
    /*@ invariant board.length == xdim;
       @ invariant (\forall int i; i >= 0 && i < xdim; board[i].length == ydim);
       @ public invariant Math.abs(getXCount() - getOCount()) < 2;
    @*/
    ...

    /*@
       @ public normal_behavior
       @ requires xdim > 0 && xdim < 10;
       @ requires ydim > 0 && ydim < 10;
       @ ensures isEmpty();
    @*/
    public XOBoard(int xdim, int ydim) {
        this.xdim = xdim;
        this.ydim = ydim;
        board = new char[xdim][ydim];
        clearBoard();
    }

    /*@ pure
       @*/
    public boolean xyInRange(int x, int y) {
        return (x >= 0 && x < getWidth()) &&
            (y >= 0 && y < getHeight());
    }
}
```

JML allows considerably more complete specifications than the standard DBC solutions discussed above (however programmers can still limit themselves to using only the basic features which are roughly the same as with the other tools). Most practicing professionals will probably not invest so much time to learn the more advanced features of a specification language. Here the principle of diminishing returns applies. The more simple DBC features allow programmers to get immediate benefits from the approach without investing huge effort.

Cheon and Leavens [Cheon 2001] developed a system for automatically generating a JUnit [Beck 1998] test class from a JML-annotated Java class. Pre-conditions become the criteria for selecting test inputs, and post-conditions provide the properties for checking test results. In this system if a test causes a post-condition assertion violation (the method throws a post-condition exception) then it is taken to have failed. A pre-condition violation means the test was meaningless because the test input was outside the domain of allowed inputs. However, for pre-condition exceptions thrown within a method (i.e. it called another method itself) this signifies a failure.

## 5 Evaluating DBC Tools

In this section a set of criteria is identified by which any design by contract tool (not just in Java) can be judged. Firstly each criterion will be discussed separately, and then afterwards the orthogonality of the criteria and some other issues such as the type of scale which should be used when applying them will be discussed.

### 5.1 The Criteria

The following list of criteria is not an exhaustive list of what could be present in a DBC tool. There are features available in some tools that are not on the list. However, the criteria in this list are considered by the author to be the most important. It was also decided not to arrange the

criteria into a taxonomy as a flat list was seen to be just as effective. The criteria were created with object-oriented software in mind but with a little modification they could also be applied to non object-oriented approaches.

#### 5.1.1 Support for Pre-Conditions, Post-Conditions and Invariants

The ability to write pre-conditions, post-conditions and invariants is a necessary condition for a system to be considered as supporting design by contract. With the support of this category a tool will have reached the minimum level of support for DBC for object-oriented languages. Note that for convenience this criterion will be referred to later as **PPI**.

How good the support is will be judged by taking the following into account:

- Ease of use of syntax
- Whether guaranteed to be side-effect free
- Can include pure method calls

Note that the term “pure” means the same as “side-effect free”.

#### 5.1.2 Support for Basic Assertions

This criterion refers to the ability to specify the basic type of assertion. This criterion will be referred to later as **BASS**. Support for basic assertions is not sufficient on its own for supporting the design by contract method. This category should be judged in a similar way to PPI above.

Note that is possible that a tool provides no support at all for basic assertions but does provide support for PPI. Basic assertions are not exclusive to object-oriented languages.

#### 5.1.3 Support for Loop Invariants and Variants

Loop invariants are not exclusive to object-oriented languages but are a very useful part of the DBC method. It should be looked at whether or not the tool supports loop invariants and variants and how good the support is. This will be referred to as **LOOP**.

#### 5.1.4 Contracts Guaranteed Side-Effect Free

Assertions should not be allowed to have side effects as the program could behave differently when assertions are turned on/off. This criterion evaluates whether this is the case for the tool under evaluation. This will be referred to as **SAFE**.

#### 5.1.5 Support for Extending Contracts over Subtypes

This criterion is certainly an important feature to have in an object-oriented language setting as inheritance is an integral part of OO. It should be looked at whether the tool in question has this support and how it is implemented. This criterion will be referred to in later sections as **EXCO**.

#### 5.1.6 Contract Soundness Checking Support

In relation to extending contracts over subtypes, it would be desirable that a tool could spot problems in the contracts - for example, contradictions over the hierarchy and breach of subtyping rules. This would be very important so as to keep to the rules of object orientation - for example, type substitutability. This will be referred to as **CSO**.

#### 5.1.7 Support for Quantifiers

To enable specification of properties of collections, quantifiers such as “for all” and “exists” are necessary. Under this criterion, it should be looked at whether this facility is present and if so the level of support should be evaluated. This criterion will be referred to as **QUAN**.

### 5.1.8 Support for “Old” Values

This refers to whether assertions in post-conditions may refer to the state of variables at method entry. This is a very useful feature to have and we will refer to it in later sections as **OLD**.

### 5.1.9 Support for Specifying Side-Effect Free Methods

This useful feature is related to **OLD**. It would be possible to produce this semantic behaviour by specifying that every variable is equal to its old value. However, it is convenient to have a single keyword that explicitly says that a method may not change any variable value. When a method is specified as side-effect free, also known as pure, there is also the advantage that it can safely be used in an assertion expression. This will be referred to as **PURE** in later sections.

### 5.1.10 Support for Specifying the Result Value

It should be possible to specify contract information for the result or return value of a method in a post-condition. This will be referred to as **RES**. The absence of this feature can however be compensated for in some cases by following a style of programming where the value returned is stored in an instance variable - this then allows it to be referenced in a post-condition.

### 5.1.11 Support for Specifying Ordering of Method Calls

A useful feature to have would be the ability to specify the order in which methods may be called. For example, a pre-condition to calling method B is that method A has already been called. This feature can also be referred to as support for trace assertions and will be referred to later as **TRC**.

### 5.1.12 Support for Generating Documentation

As one of the benefits of the design by contract approach is improved documentation, any good tool should have a facility for extracting the contract information from source files and presenting it in an easily readable form. This means that programmers would need only refer to the documentation (i.e. not to the source file) to find out how to use a particular class. The ability to combine standard documentation (i.e. natural language comments) with formal documentation should also be evaluated. This criterion will be referred to as **DOC**.

### 5.1.13 Runtime Configurability

It should be possible to set what types of assertions will be checked at runtime. For example, a user may want to set to only check pre-conditions at runtime. The recommended configurable levels of assertion checking at runtime are [Meyer 1997] (ordered from least checking to most checking):

- No checking
- Check pre-conditions
- Check pre-conditions and post-conditions
- Check pre-conditions, post-conditions and invariants
- Check the above and additionally loop invariants and variants
- Check the above and additionally all assertion statements

The runtime configurability criterion will be referred to later as **RUN**.

#### 5.1.14 Support for Static Checking

It would be desirable that a tool include an extended static checking component that makes use of the contract information to discover potential bugs before the program is even run. This will be referred to as **STAT**.

#### 5.1.15 Integration with Testing Tools

It should be investigated whether a contract tool provides any functionality for interaction with testing tools. For example, maybe it can automatically generate a test oracle from the contract specification for use with a unit testing tool. This is seen as important because of the complementary nature of the unit testing approach and the design by contract approach. This criterion will be referred to as **TEST**.

#### 5.1.16 Additional Tool Support

This criterion gives the chance to rate the support provided by various other tools related to the contract tool being evaluated. For example, graphical user interface tools, inheritance analysis tools, integrated development environments or other tools not mentioned in other criteria (e.g. testing tools is mentioned as a separate criterion) can be taken into account here. This criterion will be called **ADD**.

#### 5.1.17 Ease of Use

Factors that directly affect the ease of use of the product include:

- Clarity of compile time messages
- Clarity of runtime messages
- Ability to give a name to an assertion

Other factors that influence ease of use include the level of what can be specified in the specification language (e.g. a basic design by contract tool will probably be easier to use than one with a richer syntax), ease of installation and configuration, documentation and support available. Another point taken into account here is whether there are many bugs or shortcomings in the implementation itself. Furthermore, this is a category that actually depends on many of the other criteria presented in this section. Finally, the quality of the supplied documentation is also a factor to taken into account here. This criterion will be referred to as **EASE**.

#### 5.1.18 Learning Curve

This criterion refers to how quickly a programmer can learn to use the system at basic, intermediate and advanced levels. It will be called **LC** in future sections. There is a subtle difference between ease of use and learning curve as follows. Learning curve is taken to be how easy or difficult it is to learn how parts of a system should work. Ease of use deals with the actual use of the system. Thus, a system that has a good learning curve (for example, because the concepts are kept simple or the syntax is intuitive) may not actually be easy to use in practice (for example, because of bugs in the implementation).

#### 5.1.19 Cost (Monetary, Time, Effort)

Here it should be investigated how much the product costs in terms of money at the most basic level and whether it offers a good price/ performance ratio. Additional factors to take into account here are how much use of the product costs in terms of time used up or effort invested to get return. This can also be seen to relate to learning curve and ease of use above. This will be called **COST**.

### 5.1.20 Platform Independence

What platforms the tool run on should be taken into account here. Whether the contract tool is available on all operating systems that the target language runs on is an important question. Or it may be that it is, for example, tied to features specific to one operating system. This will be referred to as **PLAT**.

### 5.1.21 Compatibility

The first aspect of compatibility to be looked at is whether existing code needs to be modified to be able to use the system. For example, whether privates have to be changed to publics to write useful contracts or whether object types have to be changed to primitive types. A tool can avoid this need by providing for example a keyword that declares that a private variable is to be treated as public for the sake of the specification only. Another factor of compatibility to be looked at is whether the code can be compiled by a regular Java compiler after contract information has been added. Then on the runtime side it needs to be investigated if any special software or library files are needed for example. The final aspect is whether the system in question has good support for different versions, including the newest version, of the target language and libraries. This criterion will be referred to as **COMP** later.

### 5.1.22 Speed of Compilation

Here it should be investigated how long it takes to compile a benchmark project (for example, in comparison to other tools). This will be referred to as **SC**.

### 5.1.23 Speed of Runtime Execution

Here it should be investigated how the speed of execution of a benchmark annotated program compares to those generated with other tools. This will be referred to as **SR**.

### 5.1.24 Scalability

This looks at what way the speed of compilation and execution is affected by increases in the size of a project. Also to be taken into account is whether the syntax and functionality of the specification language scale up for use on large projects. This will be referred to as **SCL**.

### 5.1.25 Maintainability

It should be investigated whether each of the factors evaluated interact to give an overall maintainable product. This will be referred to as **MAIN**.

## 5.2 Discussion of the Criteria

An attempt was made to keep the criteria as orthogonal as possible but clearly there are inter-relationships as discussed in the descriptions above.

An ordinal scale should be used to rate each of the criteria because of the inherent inexactness of the measurement. Use of a ratio scale (e.g. 0 to 10) was thought about but as it would not be possible to give an exact measurement in most cases this was decided against. For example, whether one tool receives a score of 7.0 or 7.2 for a certain criterion is not realistically objectively judgeable. So the ordinal scale that was decided upon was:

1. Unsatisfactory (*referred to as **UNSAT** below*)
2. Satisfactory (*referred to as **SAT** below*)
3. Excellent (*referred to as **EXCEL** below*)

With an ordinal scale there exists only the notion of ordering, there is no notion of the difference or gap between categories [Fenton 1997]. So for example, it is not specified whether there is a bigger gap between *Unsatisfactory* and *Satisfactory* or *Satisfactory* and *Excellent*. This means that values on an ordinal scale cannot be added or multiplied, which would have been the advantage of a ratio scale.

## 6 Evaluating DBC in Java

How to evaluate the different systems available was thought about at length and it was seen that it is not possible in a situation such as this to conduct a strictly scientific *experiment* to compare the merits of each system. It was then decided that the fairest method possible would be to use a common *case study* as a means of comparison. Below the case study problem decided upon will be described and thereafter it will be analysed for suitability of application. The case study will be applied in a later section to each of the specification languages. Each system will be evaluated both subjectively and objectively (where measurements are possible).

### 6.1 The Case Study Problem

The case study decided upon models the game of Xs and Os whereby different types of players can be played against each other in single games and tournaments.

The program itself consists of the following nine classes:

- XOController** This class gets the user input and sets up the game type and player types.
- XOGame** This class contains the logic for a single game of X's and O's. A game can be started by supplying two player classes to the `playGame()` method.
- XOBoard** This class represents the X's and O's playing board itself. It is the most heavily specified part of the program as the board is probably the most critical component. The logic of valid board states is well specified and this is easily done with a contract specification language.
- XOPlayer** This is an abstract class which contains a fully implemented `play()` method and a so-called template method called `getInput()` which is the only method a subclass must implement. This class also contains an inner class called `Point` which is useful for storing x,y coordinates.
- XOPlayerClever** This is a subclass of `XOPlayer` which represents a computer player that has "intelligence" built in. It has a few rules to follow when playing the game.
- XOPlayerHuman** This is also a subclass of `XOPlayer` which allows a person to enter moves via the keyboard.
- XOPlayerRandom** This subclass of `XOPlayer` represents a computer player that makes random moves.
- XOTournament** This class allows two players to be played against each other for a specified number of games. The overall number of wins for each player and draws are printed at the end, as well as the time taken.
- XOWinChecker** This class encapsulates the logic for checking if the `XOBoard` is in a winning state.

## 6.2 Suitability of the Case Study

It was decided to develop a program a little different from the usual sort of computer science problems used. For example, stacks and queues are often used as examples as they are small and well understood. However, these examples are sometimes not taken seriously by readers as evidence of the advantages or disadvantages of a technique being investigated and because of their limited size certain properties cannot be investigated at all. In an article such as this, which intends to investigate the merits of the various systems for real world application, the mentioned factors are important.

The total number of lines of code in the program developed is 940. So it is a relatively small program but it is still big enough to allow us to investigate a considerable range of the properties of each of the tools. It was considered whether to add on a GUI component, database component and network component to make it more similar in form to a business application but unfortunately time constraints did not allow that. Nevertheless, the vast majority of the criteria mentioned above can be judged well. Note also that the program has to be small enough to allow around seven different contract systems to be evaluated within reasonable time constraints.

The criteria which are more difficult to judge on the evidence of this case study are:

- Scalability
- Maintainability

It would be possible to make up for the weakness of the measurement of scalability by taking a much larger pre-existing project and compiling and running for each of the systems. The syntax of the specifications would still have to be changed for each system however and this would be a lot of work. For maintainability an educated guess was made judging by other criteria such as “ease of use”, “compatibility”, “platform independence”, and “additional tools”. What could also be seen as a weakness in the choice of case study is that there is not a lot of opportunity to use object-oriented features - it would be more interesting if there were more inheritance relationships present for example (although there are inheritance relationships present). These areas are open to future investigation.

As mentioned, there is a single player option and a tournament option. The tournament option can be used to automatically play for example 100 games of a “random” computer player against a “clever” computer player and time how long it takes. This is a useful test for giving an objective measure of the effect of runtime assertion monitoring on the speed of execution.

To measure the speed of compilation of each system a small Perl script was written. It records the start time, runs the compile command, and then records the finish time and finally calculates the elapsed time. This likewise gives an objective measure of the speed of compilation.

## 7 Applying the Case Study to Each DBC System

The following table summarizes the results of the evaluation for each of the tools. The leftmost column of the table contains the abbreviated names of the criteria. Please refer back to section five for the detailed meaning of the criteria; a summary of the abbreviations appears underneath the main table below:

	iContract	Jass	jContractor	JMSAssert	Kopi	ESC/Java	JML
PPI	SAT	SAT	SAT	SAT	SAT	SAT	EXCEL
BASS	UNSAT	EXCEL	UNSAT	UNSAT	SAT	SAT	EXCEL
LOOP	UNSAT	EXCEL	UNSAT	UNSAT	UNSAT	SAT	EXCEL
SAFE	UNSAT	SAT	UNSAT	UNSAT	UNSAT	SAT	EXCEL
EXCO	SAT	SAT	SAT	SAT	SAT	SAT	EXCEL
CSO	UNSAT	SAT	UNSAT	UNSAT	UNSAT	UNSAT	UNSAT
QUAN	SAT	SAT	SAT	SAT	UNSAT	EXCEL	EXCEL
OLD	EXCEL	SAT	SAT	SAT	EXCEL	EXCEL	EXCEL
RES	EXCEL	EXCEL	SAT	SAT	EXCEL	SAT	EXCEL
TRC	UNSAT	EXCEL	UNSAT	UNSAT	UNSAT	UNSAT	UNSAT
PURE	UNSAT	SAT	UNSAT	UNSAT	UNSAT	UNSAT	EXCEL
DOC	SAT	UNSAT	UNSAT	UNSAT	UNSAT	UNSAT	EXCEL
RUN	EXCEL	EXCEL	EXCEL	SAT	UNSAT	UNSAT	SAT
STAT	UNSAT	UNSAT	UNSAT	UNSAT	UNSAT	SAT	UNSAT
TEST	UNSAT	UNSAT	UNSAT	UNSAT	UNSAT	UNSAT	SAT
ADD	SAT	UNSAT	UNSAT	UNSAT	UNSAT	UNSAT	SAT
EASE	UNSAT	UNSAT	EXCEL	SAT	UNSAT	SAT	EXCEL
LC	SAT	SAT	EXCEL	SAT	SAT	SAT	SAT
COST	EXCEL	EXCEL	EXCEL	EXCEL	EXCEL	SAT	EXCEL
PLAT	SAT	EXCEL	EXCEL	UNSAT	EXCEL	EXCEL	EXCEL
COMP	SAT	EXCEL	SAT	UNSAT	UNSAT	SAT	EXCEL
SC	SAT	EXCEL	EXCEL	EXCEL	EXCEL	n/a	SAT
SR	EXCEL	SAT	EXCEL	UNSAT	EXCEL	n/a	SAT
SCL	SAT	SAT	EXCEL	UNSAT	SAT	SAT	SAT
MAIN	SAT	SAT	SAT	UNSAT	SAT	SAT	EXCEL

Summary of abbreviation meanings with corresponding section numbers in brackets:

<b>PPI</b>	Pre-,post-conditions, invariants (5.1.1)	<b>STAT</b>	Static checking (5.1.14)
<b>BASS</b>	Basic assertions (5.1.2)	<b>TEST</b>	Testing tool integration (5.1.15)
<b>LOOP</b>	Loop invariants and variants (5.1.3)	<b>ADD</b>	Additional tool support (5.1.16)
<b>SAFE</b>	Contracts guaranteed pure (5.1.4)	<b>EASE</b>	Ease of use (5.1.17)
<b>EXCO</b>	Extend contracts over subtypes (5.1.5)	<b>LC</b>	Learning curve (5.1.18)
<b>CSO</b>	Contract soundness (5.1.6)	<b>COST</b>	Cost (5.1.19)
<b>QUAN</b>	Quantifiers (5.1.7)	<b>PLAT</b>	Platform independence (5.1.20)
<b>OLD</b>	Refer to “old” values (5.1.8)	<b>COMP</b>	Compatibility (5.1.21)
<b>PURE</b>	Specify methods as pure (5.1.9)	<b>SC</b>	Speed of compilation (5.1.22)
<b>RES</b>	Specify return values (5.1.10)	<b>SR</b>	Speed of runtime (5.1.23)
<b>TRC</b>	Trace assertions (5.1.11)	<b>SCL</b>	Scalability (5.1.24)
<b>DOC</b>	Documentation generation (5.1.12)	<b>MAIN</b>	Maintainability (5.1.25)
<b>RUN</b>	Runtime configurability (5.1.13)		

Notes:

- An ordinal scale is used for rating as discussed above. **UNSAT** means unsatisfactory, **SAT** means satisfactory and **EXCEL** means excellent.
- n/a = Not applicable

The following table shows compile times for the case study and runtimes for the runtime test scenario of the case study. The runtime test scenario measured a tournament between a “random” computer player and a “clever” computer player over 100 games. The average time was taken from an equal number of tournaments where the random player went first and where the clever player went first:

	iContract	Jass	jContractor	JMSAssert	Kopi	ESC/Java	JML	Java
CMPL	40	23	5	6	10	n/a	49	5
RNT_ON	27	194	24	1050	25	n/a	36	n/a
RNT_PRE	25	192	23	11	n/a	n/a	n/a	n/a
RNT_OFF	24	n/a	23	11	n/a	n/a	35	9

Notes:

- CMPL = Compile time for benchmark project.
- RNT\_ON = Average runtime for benchmark test with all assertion checks ON.
- RNT\_PRE = Average runtime for benchmark test with only pre-condition checks ON.
- RNT\_OFF = Average runtime for benchmark test with all assertion checks OFF.
- n/a = Not applicable.
- All measurements above are in seconds.
- The Java column is a reference column which shows how long it took to compile with a standard Java compiler (Sun javac) or run the code with the same virtual machine as used elsewhere. It is given for means of comparison.
- Some systems may have less contract information than others depending on what it is possible to specify therein.
- Compile time is the time taken to convert the original source files to class files. In cases where a system only generates instrumented source files, a script was created to run the standard Sun java compiler straight after the pre-processor. The time given is the total time taken.
- As a rule all runtime measurements were done on a Sun Java 1.4.1 Hotspot VM. The exception to this rule was JMSAssert which only works in connection with a classic VM so the program was run on a Sun Java 1.2.2 Classic VM.

## 8 Conclusions

In this section several types of potential DBC users are identified and a set of desired minimum requirements is assigned to each to find out what systems satisfy their needs. Furthermore, conclusions are drawn as to the general suitability for deployment of the systems examined.

**Student / lecturer / academic:** An appropriate system should satisfy the following criteria: LC, PPI, BASS, LOOP, EXCO and SAFE. The reasoning is that the learning curve should be good, and all the basic concepts of DBC should be usable. The chosen system should also support either static checking (STAT) or be configurable for runtime checking (RUN) so that the concepts can be tested practically. The system should not cause any side-effects (which is a pre-requisite for any system being used by any user). Unfortunately, only three systems out of the seven evaluated satisfy the SAFE criterion. The three systems that qualify for this category are JML, Jass and ESC/Java.

**Professional software developer working on a small business project:** An appropriate system should satisfy all of the above criteria and additionally: QUAN, OLD, RES, PURE, DOC, EASE and COST. The only system that qualifies for this category is JML.

**Professional software developer working on a large business project:** Here an appropriate system should satisfy all of the above criteria plus: PLAT, COMP, SC, SR, SCL and MAIN. Again, the only system that qualifies for this category is JML.

**Safety critical/embedded systems developer:** To qualify for this category a system should additionally provide CSO, TEST and both STAT and RUN. No system in this article qualifies for this category.

In the author's opinion it is obvious from the evaluation results that the DBC tools examined are generally not yet suitable for widespread commercial use. In addition, judging by the results the tool that performs best overall and comes closest to being ready for commercial deployment is JML. However, design by contract in Java will probably not be widely adopted until good support is directly added to the language and platform.

## References

- [Aho 1986] Aho, Alfred V., Ravi Sethi, Jeffrey D. Ullman "Compilers Principles, Techniques and Tools", Addison Wesley, 1986
- [Ant 2003] Apache Ant, "<http://ant.apache.org/>", 2003
- [Artho 2001] Artho, Cyrille "Finding Faults in Multi-threaded Programs", Masters Thesis, 2001
- [Bartetzko 2001] Bartetzko, Detlef, Clemens Fischer, Michael Möller, Heike Wehrheim "Jass - Java with Assertions", Electronic Notes in Theoretical Computer Science 55 No. 2 2001
- [Behm 1999] Behm, Patrick, Paul Benoit, Alain Faivre, Jean-Marc Meynadier "Météor: A Successful Application of B in a Large Project", FM'99, Vol. I, LNCS 1708, pp. 369-387, 1999
- [Beck 1998] Beck, Kent, Erich Gamma "Test Infected: Programmers Love Writing Tests", Java Report, 3(7), July 1998
- [Bruno 1998] Bruno, John, Urs Hölzle, Murat Karaorman "jContractor: A Reflective Java Library to Support Design By Contract", 1998
- [Cheon 2001] Cheon, Yoonsik, Gary T. Leavens, "A Simple and Practical Approach to Unit Testing: The JML and JUnit Way", TR #01-12a, 2001
- [Chidamber 1991] Chidamber, Shyam R., Chris F. Kemerer "Towards a Metrics Suite for Object Oriented Design", OOPSLA, 1991
- [CF 1998] Computer Finance, "The cost of poor quality software", Computer Finance, Issue Number 9.06, 1998
- [Dahl 1972] Dahl, Ole-Johan, Edsger W. Dijkstra, C.A.R. Hoare "Structured Programming", Academic Press, 1972
- [Detlefs 1998] Detlefs, David L., K. Rustan M. Leino, Greg Nelson, James B. Saxe "Extended Static Checking", Compaq SRC Research Report, 1998

- [Duncan 1998] Duncan, Andrew, Urs Hölze "Adding Contracts to Java with Handshake", Technical Report TRCS98-32, University of California, Santa Barbara, 1998
- [Dijkstra 2000] Dijkstra, Edsger W. "Answers to questions from students of Software Engineering", EWD 1305, 2000
- [Enseling 2001] Enseling, Oliver "iContract: Design by Contract in Java", Javaworld, February 2001
- [Felleisen 2001] Felleisen, Matthias, Robert Bruce Findler "Contract Soundness for Object-Oriented Languages", OOPSLA 2001
- [Fenton 1997] Fenton, Norman E., Shari Lawrence Pfleeger "Software Metrics: A Rigorous & Practical Approach" 2nd edition, PWS Publishing Company, 1997
- [FM 2003] Formal Methods Europe, "<http://www.fmeurope.org>", 2003
- [Gannon 1995] Gannon, John D., Jim Purtilo, Marvin V. Zelkowitz "Software Specification: a Comparison of Formal Methods", Intellect Books, 1995
- [Gibson 1993] Gibson, J. Paul "Formal Object Oriented Development of Software Systems using LOTOS", Ph.D. Thesis, University of Stirling, 1993
- [Gosling 1994] Gosling, James "Oak Language Specification", 1994
- [Gosling 1996] Gosling, James, Bill Joy, Guy Steele, Gilad Bradcha "The Java Language Specification Second Edition", Addison Wesley, 1996
- [Hoare 1972] Hoare, C.A.R. "Proof of Correctness of Data Representations", Acta Informatica, Vol. 1, 1972
- [ISO 2001] ISO, "ISO/IEC 9126-1:2001 Software engineering – Product quality – Part 1: Quality model", 2001
- [JavaDoc 2003] JavaDoc, "Programming with Assertions", <http://java.sun.com/j2se/1.4.2/docs/guide/lang/assert.html>, JavaDocs, 2003
- [Jézéquel 1997] Jézéquel, Jean-Marc, Bertrand Meyer "Design by Contract: The Lessons of Ariane", Computer(IEEE) vol. 30, no. 2, pages 129-130, January 1997
- [Joyner 1999] Joyner, Ian "Objects Unencapsulated", Prentice Hall, 1999
- [Kleppe 1999] Kleppe, Anneke, Jos Warmer "The Object Constraint Language", Addison Wesley, 1999
- [Kramer 1998] Kramer, Reto "iContract - The Java Design by Contract Tool", TOOLS USA 1998
- [Kramer 1999] Kramer, Reto "Examples of Design by Contract in Java", Object World Berlin May 1999
- [Lackner 2002] Lackner, Martin, Andreas Krall "Supporting Design by Contract in Java", TOOLS USA 2002
- [Leavens 1999] Leavens, Gary T., Albert L. Baker, Clyde Ruby "JML: A Notation for Detailed Design", Kluwer Academic Publishers, 1999
- [MMS 2000] Man Machine Systems "Design by Contract for Java Using JMSAssert", <http://www.mmsindia.com/DBCForJava.html>, 2000
- [Meyer 1997] Meyer, Bertrand "Object-Oriented Software Construction, Second Edition", Prentice Hall, 1997

- [Meyer 2001] Meyer, Bertrand "An Eiffel Tutorial, ISE Technical Report TR-EI-66/TU", July 2001
- [Meyer 2003] Meyer, Bertrand "The Grand Challenge of Trusted Components", ICSE 25, IEEE Computer Press, May 2003
- [McKim 2002] McKim, Jim, Richard Mitchell "Design by Contract, by Example", Richard Mitchell and Jim McKim, Addison Wesley, 2002
- [Naur 1969] Naur, Peter, Brian Randell "Software Engineering: Report on a conference sponsored by the NATO science committee", January 1969
- [Parasoft 2003] Parasoft, "Using Design by Contract to Automate Java Software and Component Testing", [www.parasoft.com](http://www.parasoft.com), 2003
- [Plösch 2002] Plösch, Reinhold "Evaluation of Assertion Support for the Java Programming Language", Reinhold Plösch, Johannes Kepler University Linz, Austria, in Journal of Object Technology, Vol. 1, No. 3, Special issue: TOOLS USA 2002 proceedings, pages 5-17, 2002
- [Spivey 1992] Spivey, J. M. "The Z Notation: A Reference Manual", Prentice Hall, 1992
- [Wirth 1971] Wirth, Niklaus "Program Development by Stepwise Refinement", Communications of the ACM, Volume 14, Issue 4, April 1971