Design by Contract™ for Java™ Using JMSAssert™

Design by Contract (DBC) pioneered by Bertrand Meyer [1] is widely acknowledged to be a powerful technique for writing reliable software. The three key elements of DBC are preconditions, postconditions and class invariant. An example of a language that has direct support for DBC is Eiffel. Unfortunately, Java™ does not directly support DBC. However, *jmsassert* brings the benefits of DBC to Java. In this document, you will find information that will assist you in incorporating "contract" into a Java program. *jmsassert* is currently available on Windows™ platforms only.

Here is how, in a nutshell, you will use *jmsassert*. The java program that requires rigorous specification will be annotated at the source level by using certain markers within Javadoc™ comments. The next step is to run *jmsassert* on the Java source code; this process results in the automatic creation of certain contract files that contain code in JMScript™, a Java-based scripting language developed by Man Machine Systems. The generated JMScript code actually represents triggers that are called by the assertion runtime to enforce the explicitly stated contractual obligations on the part of suppliers and consumers..

The three essential elements of DBC are precondition, postcondition and class invariant. To use jmsassert, you need to first specify these elements (any or all of them) within Javadoc comments using special tags (or markers). Table 1 lists these tags.

Tag	***Description***
@inv	Class invariant
@pre	Method precondtion
@post	Method postcondition

Table 1. Tags to Specify DBC Elements

Class Invariant

Class invariant is represented by the marker "@inv". The marker is followed by a boolean expression that may reference any element of the class or its direct/indirect bases, including private elements. This tag may appear within any Javadoc comment as part of a class. However, it is preferable (in the interest of readability) to define it just before the class. In the case of anonymous classes, where it is not possible to specify the invariant before the class, it may appear in any Javadoc comment as part of the class. Multiple @inv tags may appear for the same class within one or more Javadoc comments. The effective class invariant will then be the conjunction of all following boolean expressions. Here is an example of correct invariant specification.

```
/** x
    @inv a > 0
*/
class A {
    int a = 1;
    int b;
    int c;
    /**
        @inv a < 45
        @inv (b >= 100 && b < 900)
    */
    void ff() { /* ... */ }
    /**
        @inv c != b
    */
    public void gg() { /* ... */ }
}
```

The effective invariant for the above class is

```
    a > 0 && a < 45 && (b >= 100 && b < 900) && c != b
```

Precondition

The corresponding marker is "@pre". Preconditions must be defined within Javadoc comments preceding the respective methods. The marker is followed by a boolean expression as in the class invariant. The condition may reference arguments passed to the method, in addition to accessing elements of the class. If multiple @pre markers appear within the same Javadoc comment, the effective precondition for the method will be the conjunction of all such preconditions. The following is an example.

```
/**
@inv a > 0
*/

class A {
    int a = 1;
    /**
        @pre val > 100
        @pre val <= 200
    */
    void ff(int val) {
        a = val;
    }
}
```

The effective precondition for method *ff()* is:

```
   (val > 100 && val <= 200)
```

Postcondition

The marker used to denote a postcondition is "@post". Postconditions are specified within Javadoc comments preceding the respective methods. The condition may be any boolean expression as in the precondition case. In addition, the boolean expression may use the qualifier "$prev" on an expression to denote the expression's value at the method's entry point. For example, "$prev (top)" denotes the value of variable "top" at method entry (without the qualifier, it indicates its current value). The keyword "$ret" may appear in the condition to denote the method's return value.

The following is an example.

```
class A {
    int a;
    /**
        @pre val > 10
        @post a == $prev(a) + val
    */
    public void setVal(int val) {
        a += val;
    }
    /**
        @post $ret == a
    */
    public int getVal() {
        return a;
    }
}
```

The order of tags within a Javadoc comment is not important. The following is a legal specification.

```
class A {
    /**
        @post a > 100
        @pre val > 100
        @post a <= 300
        @inv a > 0
        @post val <= 300
    */

    void ff(int val) {
        a = val;
    }
}
```

The assertion expressions that appear after tags must conform to the syntax of JMScript boolean expressions. To capture more complex conditions, one may use the special escape marker "@macro" to directly embed a JMScript code fragment. As an example of where escaping to JMScript code via @macro directive helps, consider the following class.

```
class MyVector {
    private int[] arr = new int[10];
    /**
        @post @macro foreach(elm in arr) assertPost(elm == 0);
    */
    public void clear() {
        for(int i = 0; i < 10; ++i)
            arr[i] = 0;
    }
    // Other elements
}
```

When are the contract elements checked?

It is important to know at what points in the program execution different assertions are checked. Table 2 shows how pre-, postcondition and invariant are used in the context of a class.

Event		Invariant	Pre-	Post-

			condition	condition
Method Entry	***Instance Method***	Y	Y	-
	Constructor	N	Y	-
	Private Method	N	Y	-
	Static Method	N	Y	-
Method Exit	***Instance Method***	Y	-	Y
	Constructor	Y	-	Y
	Private Method	N	-	Y
	Static Method	N	-	Y

Table 2. JMScript Triggers Context

It can be seen from the above table that invariant is not checked in the context of private methods and postcondition is not checked when a method terminates abnormally by throwing an exception.

Recursive Calls

In case of recursive calls, JMScript executes triggers as one would expect. The triggers are executed each time the method is called irrespective of whether the call is recursive or not.

It is important to note that the trigger for a Java method is executed even if the call originates from another trigger. In particular, calling a Java method in the method's trigger can lead to a potentially recursive situation.

Contracts for Interfaces and Base Classes

Contracts may be specified for an interface, which gets propagated to its implementing classes. The same rule is applied to class hierarchies. For both interfaces and class hierarchies, preconditions are disjuncted, whereas postconditions and class invariants are conjuncted. Take the example of an interface shown below.

```
interface Employee {
    /**
        @pre age > 25
    */
    public void setAge(int age);

    /**
        @post $ret > 25
    */
    public int getAge();
}
```

Pre- and postconditions have been established for the two declared methods. Now consider a class that implements this interface:

```
class ImpEmployee implements Employee {
    private int eage;
    /**
        @pre age < 65
    */
    public void setAge(int age) {
        eage = age;
    }

    /**
        @post $ret < 65
    */
    public int getAge() {
        return eage;
    }
}
```

The effective precondition for ImpEmployee.setAge() is (age > 25 || age < 65) and the effective postcondition for ImpEmployee.getAge() is ($ret > 25 && $ret < 65). This can be extended to a class implementing multiple interfaces.

Similarly, whenever an inner class method is executed it can potentially change the outer class fields thus affecting its invariant. The test environment automatically checks the invariant of the outer class(es) when an inner class method is executed.

Table 3 summarizes how pre-, postcondition and invariant of related methods and classes are called in the context of a class.

Trigger Type	**Executed Triggers**	**Conditions to be satisfied**
Invariant	• Invariant of the class. • Invariant of all the base classes and base interfaces, if any. • Invariant of all its outer classes, if any.	• The invariant of the class, its bases classes and interfaces must all be satisfied. • The invariant of all its outer classes must be satisfied.
PreCondition	• Precondition of the method. • Precondition of all the methods that it overrides	• At least one of the preconditions must be true (disjuncted).
PostCondition	• Postcondition of the method. • Postcondition of all the methods that it overrides	• All the postconditions must be true (conjuncted).

Table 3. JMScript Trigger Execution Pattern.

Illustrative Example

The following discussion serves to illustrate our approach. Consider the canonical stack implementation in Java (to avoid confusion with JDK Stack class, the following class has been named *MyStack*):

```
/**
    @inv (top >= 0 && top < max)
*/
class MyStack {
    private Object[] elems;
    private int top, max;

    /**
        @pre (sz > 0)
        @post (max == sz && elems != null)
    */
    public MyStack(int sz) {
        max = sz;
        elems = new Object[sz];
    }

    /**
        @pre !isFull()
        @post (top == $prev (top) + 1) && elems[top-1] == obj
    */
    public void push(Object obj) {
        elems[top++] = obj;
    }

    /**
        @pre !isEmpty()
        @post (top == $prev (top) - 1) && $ret == elems[top]
    */
    public Object pop() {
        return elems[--top];
    }

    /**
        @post ($ret == (top == max))
    */
    public boolean isFull() {
        return top == max;
    }

    /**
        @post ($ret == (top == 0))
    */
    public boolean isEmpty() {
        return top == 0;
    }
} // End MyStack
```

Once a Java source file has been annotated, as above, the next step is to use the preprocessor "jmsassert" utility to generate JMScript triggers for the various assertions. In this case, we do the following:

> jmsassert –s MyStack.java

The preprocessor creates two output files:

1. default_MyStack.jms
2. Startup.jms

The first file (see Figure 1) contains JMScript triggers corresponding to the embedded assertions in Java source. The second file (Figure 2) makes it convenient to register the automatically generated triggers (in particular, if there are more than one class) with JMScript runtime.

```
/*
 * Trigger file for class #default.MyStack.
 * Generated by JMSAssert on Monday, April 12, 1999.
 * Any changes you make to this file will be overwritten if
 * you regenerate this file.
*/
```

```
import macro;

// Postcondition for method - MyStack(int)
MyStackPost(meth, $obj, sz) {
    assertPost(($obj.max == sz && $obj.elems != null));
}

// Precondition for method - MyStack(int)
MyStackPre(meth, $obj, sz) {
    assertPre((sz > 0));
}

// Postcondition for method - void push(Object)
pushPost(meth, $obj, obj, $ret) {
    assertPost(($obj.top == this.top$prev + 1) && $obj.elems[this.top$prev] == obj);
}

// Precondition for method - void push(Object)
pushPre(meth, $obj, obj) {
    this.top$prev = $obj.top;
    assertPre(!$obj.isFull());
}

// Postcondition for method - Object pop()
popPost(meth, $obj, $ret) {
    assertPost(($obj.top == this.top$prev - 1) && $ret == $obj.elems[$obj.top]);
}

// Precondition for method - Object pop()
popPre(meth, $obj) {
    this.top$prev = $obj.top;
    assertPre(!$obj.isEmpty());
}

// Postcondition for method - boolean isFull()
isFullPost(meth, $obj, $ret) {
    assertPost(($ret == ($obj.top == $obj.max)));
}

// Postcondition for method - boolean isEmpty()
isEmptyPost(meth, $obj, $ret) {
    assertPost(($ret == ($obj.top == 0)));
}

MyStackinv(meth, $obj) {
    assertInv(($obj.top >= 0 && $obj.top <= $obj.max));
}

static {
    assertStrMyStack = {
        { "<init>(I)V", "POSTCONDITION", "MyStackPost" },
        { "<init>(I)V", "PRECONDITION", "MyStackPre" },
        { "push(Ljava/lang/Object;)V", "POSTCONDITION", "pushPost" },
        { "push(Ljava/lang/Object;)V", "PRECONDITION", "pushPre" },
        { "pop()Ljava/lang/Object;", "POSTCONDITION", "popPost" },
        { "pop()Ljava/lang/Object;", "PRECONDITION", "popPre" },
        { "isFull()Z", "POSTCONDITION", "isFullPost" },
        { "isEmpty()Z", "POSTCONDITION", "isEmptyPost" },
        { "", "INVARIANT", "MyStackinv" }
    };
    setClassTrigger("MyStack", assertStrMyStack);
}

load() {}
```

Figure 1. Triggers in JMScript for MyStack class

```
//Startup trigger file - c:\ranga\jverify\Startup.jms
import macro;
load() {}
static {
    default_MyStack.load();
}
```

Figure 2. The Startup JMScript File

The third step is to compile the relevant Java source files. Assume that in addition to the MyStack definition as above, we also have the following test driver class:

```
class StackTest {
    public static void main(String[] args) {
        MyStack s = new MyStack(2); // Can push at most two elements
        s.push(new Integer(1));
        s.push(new Integer(23));
        s.push(new Integer(0)); // Precondition violation here!
    }
}
```

This driver, along with the MyStack class, is compiled using a regular Java compiler such as "javac". The final step is, of course, to run the Java code with assertions enabled. To do this, invoke the Java interpreter as follows:

>java -Xdebug -Xnoagent -Djava.compiler=NONE –Xrunjmsdll:Startup StackTest

The CLASSPATH environment variable must be appropriately set to JDK1.2 runtime files, and additionally, must include mmsclasses.jar, which is supplied as part of the assertion environment.

The DLL jmsdll includes the JMScript interpreter. This DLL registers itself with JVM and assigns assertion triggers to the respective Java methods. When the JVM invokes a Java method, the call is intercepted by the DLL, and if a trigger (pre-, postcondition or invariant) is associated with it, the corresponding JMScript trigger method is invoked. Notice how the trigger code in JMScript is able to access private elements of a class. We consider this to be a significant advantage in terms of testing. Though, as brought out in this article, JMScript is primarily useful for bringing DBC to Java, the language has been designed for general purpose scripting, and has some very attractive features such as partial function specialization, multimethod, dynamic inheritance, and so on. Its strength derives from the underlying JVM.

One of the benefits of this approach is that the original Java source code is unmodified; what is tested with assertions is the same as the one executed normally. To run the test driver without enabling assertion code, simply run as follows:

>java StackTest

The limitation that Java source code must be available in order to specify contracts is overcome when you use the JVerify environment. That environment allows assignment of triggers to Java by inspecting compiled class files.

Conclusion

Though the Java programming language does not directly support design by contract as yet, there are several ways to add such a support from outside. The approach described in this article uses a preprocessor to map contracts embedded in Java source code to triggers in

JMScript, a Java-based scripting language. The triggers are then automatically executed by an extension DLL that includes the JMScript interpreter. To run without assertions, all that is required is to run the Java program without the extension DLL. This approach ensures that the released program is identical to the one tested. In order to test a class, no special modifications are necessary (other than source annotation). Both JVerify and JMScript are currently available on Windows™ platforms. The assertion package described in this article is available for unrestricted free public use.

References

1. Bertrand Meyer, Object-Oriented Software Construction,
2. Reto Kramer, "iContract – The Java ™ Design by Contract ™ Tool", http://www.reliable-systems.com/tools
3. Andrew Duncan and Urs Holzle, "Adding Contracts to Java with Handshake", Technical Report TRCS98-32, Department of Computer Science, University of California, Santa Barbara.
4. Murat Karaorman, Urs Holzle, and John Bruno, "jContractor: A Reflective Java Library to Support Design by Contract", Technical Report TRCS98-31, Department of Computer Science, University of California, Santa Barbara.
5. Rangaraajan, K., "How Can I Test Java Classes?", Dr.Dobb's Journal , July 1999, pp 107-110.