DbC4J: Design by Contract™ for Java™

User Manual

June 6, 2007

Sérgio Agostinho, Pedro Guerreiro

{sergioag, pg} @di.fct.unl.pt

CITI / DI – Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa

2829-516 Caparica, Portugal +351 21 294 83 00

Table of Contents

Introduction

Design by Contract (DbC) is an approach introduced in the Eiffel programming language by Bertrand Meyer. It is based in the idea of an agreed contract between a supplier and a client. As such, a class may have:

- an operation **precondition**: an obligation to the client (a property that must be valid before the invocation of the operation) and a benefit to the supplier (which this way does not have to verify it);
- an operation **postcondition**: an obligation to the supplier (a property that must be valid after the invocation of the operation) and a benefit to the client (which this way does not have to verify it);
- an **invariant**: a property that must be valid before and after the execution of any operation.

While Eiffel has native support for DbC through the use of these conditions, most programming languages have little or no support for it. The purpose of DbC4J is to extend the Java language to support these concepts. Unlike other implementations, which use a different constraint language (in the form of comments or annotations) for contract specification, our aim is to use Java executable code. This provides a more transparent and natural way for specifying the contract.

DbC4J is implemented in AspectJ, an Aspect-Oriented extension to Java. This technology allows a non-intrusive behaviour introduction (unlike other technologies, such as the use of preprocessors) and creates no tool dependency. The contract is written in Java or AspectJ code, but the produced binaries are standard Java. The contract conditions are identified through the use of naming conventions, as explained through this document.

This library was written in Java 1.5 and AspectJ 1.5, using AJDT 1.4 for Eclipse 3.2.

Distribution Content

DbC4J is distributed through several archives. Namely:

- binaries
  - `dbc4j.jar` – the library JAR file;
  - `dbc4j-placebo.jar` – the library "stub" JAR file;
  - `rules-example.properties` – a rules configuration example file;
  - `license.txt` – the (GPL) license;
- source
  - `src` – the source code directory;
    - `examples` – a directory with some examples on using the library;
      - `test` – a directory with some JUnit tests for testing the examples;
- documentation
  - `docs` – the javadocs directory;
- manual
  - `dbc4j-manual.pdf` – this file!

Installation

Requirements

The software requirements depend on the way that you intend to develop your software.

If you are using the command line:

- Java Development Kit (JDK) 1.5 [download]
- AspectJ 1.5 [download]

If you are using the Eclipse IDE, the following is required:

- Java Development Kit (JDK) 1.5 [download]
- Eclipse Integrated Development Environment (IDE) 3.2 [download]
- AspectJ Development Tools (AJDT) 1.5 for Eclipse [download]

If you are using NetBeans IDE, the following is required:

- Java Development Kit (JDK) 1.5 [download]
- NetBeans Integrated Development Environment (IDE) 5.5 [download]
- AspectJ Development Environment (AJDE) 1.5 for NetBeans [download]

Command Line

1. Copy the supplied JAR file, as well as the AspectJ runtime library JAR to a directory in your project (for the following steps we are assuming that would be “lib”).
2. Compile your source code with the 'ajc' tool, as show in the example above:

```
[/home/user/project]$ ajc src/Foo -aspectpath lib/dbc4j.jar -inpath lib/dbc4j.jar
```

Alternatively, you can also create a 'lst' file (for example, sources.lst), with the following content:

```
src/Foo.java
-aspectpath
lib/dbc4j.jar
-inpath
lib/dbc4j.jar
```

This way, compiling is a little simpler:

```
[/home/user/project]$ ajc -argfile sources.lst
```

In order to run your application, you just need to use 'java' tool, including the necessary JAR files, as show in the examples above:

```
[/home/user/project/src]$ java -classpath ".:../lib/aspectjrt.jar:../lib/dbc4j.jar" Foo
```

(Linux or Unix systems)

```
C:\Documents and Settings\user\project\src> java -classpath ".;../lib/aspectjrt.jar;../lib/dbc4j.jar" Foo
```

(Windows systems)

(Tested under JDK 1.5.0 and AspectJ 1.5.3)

Eclipse IDE

1. Create a new AspectJ project (“File -> New -> Project...”). If you already have a Java project you can convert it to an AspectJ one (right-click the project in the explorer and select “AspectJ Tools -> Convert to AspectJ Project”).

2. If you plan to transport the project to other computers/operating systems, you need to take an extra sub-steps. If not, you can skip this step.
    - Create a new package in your project. Use a familiar convention, such as “lib” (to represent that this project will only contain external libraries). Eclipse creates a directory with that name under your project.
    - Copy the supplied JAR file to that directory (you can do this by simply dragging the file to the package in the Eclipse explorer). Refresh your workspace. The JAR file should appear there.
3. Expose the supplied JAR to weaving. To do so, go to “Project -> Properties -> AspectJ Inpath-> Libraries and Folders -> Add JARs” (or “Add External JARs” if you skipped step 2).
4. Expose the supplied JAR to the class path. To do so, go to “ Project -> Properties -> Java Build Path -> Libraries and Folders -> Add JARs” (or “Add External JARs” if you skipped step 2). Note that if do not intent to use the public classes supplied by DbC4J (ContractMemory, ContractRules and ContractLogic), you can skip this step.

If all went well, you should see arrows pointing to your class methods indicating that they are under advice. Sometimes you need to clean and build your project to see the changes.

From now on, any time you run your application, it is under the DbC4J contract evaluation. Notice that this happens either by running the application as “Java Application” or “AspectJ/Java application”. This is not a bug, but a way so that you can use DbC4J independently on how you run your class (e.g. as a “JUnit Test”).

(Tested under JDK 1.5.0, Eclipse 3.2.1 and AJDT 1.4.0)

NetBeans IDE

1. Create a new project (“File -> New Project...”). You can choose any type of project.
2. Turn on AJDE (“Tools -> AspectJ -> Start AJDE”).
3. Copy the supplied JAR file to a directory (e.g. lib).
4. Now you need to add the supplied JAR file to the project as a library. To do so, right-click on “Libraries” and select “Add JAR/Folder”.
5. Create a Lst file at “New -> File/Folder... -> Source roots Lst file” (for example, sources.lst)
6. Add the following lines to the file (the last line may need to be adjusted to the path where you stored the JAR file):

```
-sourceroots
.
-aspectpath
../lib/dbc4j.jar
```

If all went well, you just need to build the project, by using “Tools -> AspectJ -> Select Build Configuration...” and selecting the Lst file created in step 5. Then run your application as you normally would, by right-clicking the main class and selecting “Run File”.

(Tested under JDK 1.5.0, NetBeans 5.5 and AJDE 1.5.0)

How to use

Preconditions

Preconditions are methods with the same name as the original method, but prefixed with the pre keyword, using "lower camel case". These methods return a boolean value (either the primitive type as well as the wrapper class).

Example:

```
/* original method */
```

```
public int foo() {
    // ...
}

/* precondition method */
public boolean preFoo() {
    //...
}

/* precondition method - alternative */
public Boolean preFoo() {
    //...
}
```

It is possible to use the arguments supplied to the method in the precondition. All the arguments can be used, or only an ordered subset of them.

Example:

```
/* original method */
public int foo(int x, int y, char c) {
    // ...
}

/* precondition method - all arguments */
private boolean preFoo(int x, int y, char c) {
    //...
}

/* precondition method - first two arguments */
public boolean preFoo(int x, int y) {
    //...
}

/* precondition method - first argument */
public boolean preFoo(int x) {
    //...
}

/* precondition method - no arguments */
public boolean preFoo() {
    //...
}

/* precondition method - invalid use! */
public boolean preFoo(char c) {
    //...
}
```

Preconditions methods can be public or private; the methods under contract can have any non-private visibility (public, private, or package).

Note that the argument names for the precondition/postcondition methods are not necessarily the same as in the original method. However, it is a good convention to do so, in order to avoid ambiguities.

Postconditions

Postconditions are methods with the same name as the original method, but prefixed with the `post` keyword, using "lower camel case". These methods return a boolean value (either the primitive as well as the wrapper class).

Example:

```
/* original method */
public int bar() {
    // ...
```

```
}

/* postcondition method */
public boolean postBar() {
    //...
}

/* postcondition method - alternative */
public Boolean postBar() {
    //...
}
```

It is possible to use the returned value of the method, as well as the arguments supplied to the method in the postcondition. All the arguments can be used, or only an ordered subset of them. If you are using the arguments, you must also use the result.

Example:

```
/* original method */
public int bar(char a, char b) {
    // ...
}

/* postcondition method - result and all arguments */
public boolean postBar(int result, char a, char b) {
    //...
}

/* postcondition method - result and first argument */
public boolean postBar(int result, char a) {
    //...
}

/* postcondition method - result */
public boolean postBar(int result) {
    //...
}

/* postcondition method - no arguments */
public boolean postBar() {
    //...
}

/* postcondition method - invalid use! */
public boolean postBar(char a) {
    //...
}
```

The exception to the previous rule is for postconditions of void returning methods or constructors.

Example:

```
/* original method */
public void setX(int x) {
    // ...
}

/* postcondition method - using argument */
public boolean postSetX(int x) {
    //...
}

/* postcondition method - no argument */
public boolean postSetX() {
    //...
}
```

Postconditions methods can be public or private; the methods under contract can have any non-private visibility (public, private, or package).

Invariants

Invariants are methods named invariant. These methods return a boolean value (either the primitive as well as the wrapper class).

Example:

```
/* invariant method */
public boolean invariant() {
    //...
}

/* invariant method - alternative */
public Boolean invariant() {
    //...
}
```

Invariants are public or private methods that take no arguments. Note that invariants are always checked before and after any non-private, non-static method execution. However, in constructors they are only checked afterwards, since invariants are meant to evaluate the internal state of the class.

Exceptions

Now that you know the basic contract specifications mechanisms, what happens when then fail? Just like in standard Java programming: through the exception mechanism. The exception hierarchy is explained in Figure 1.

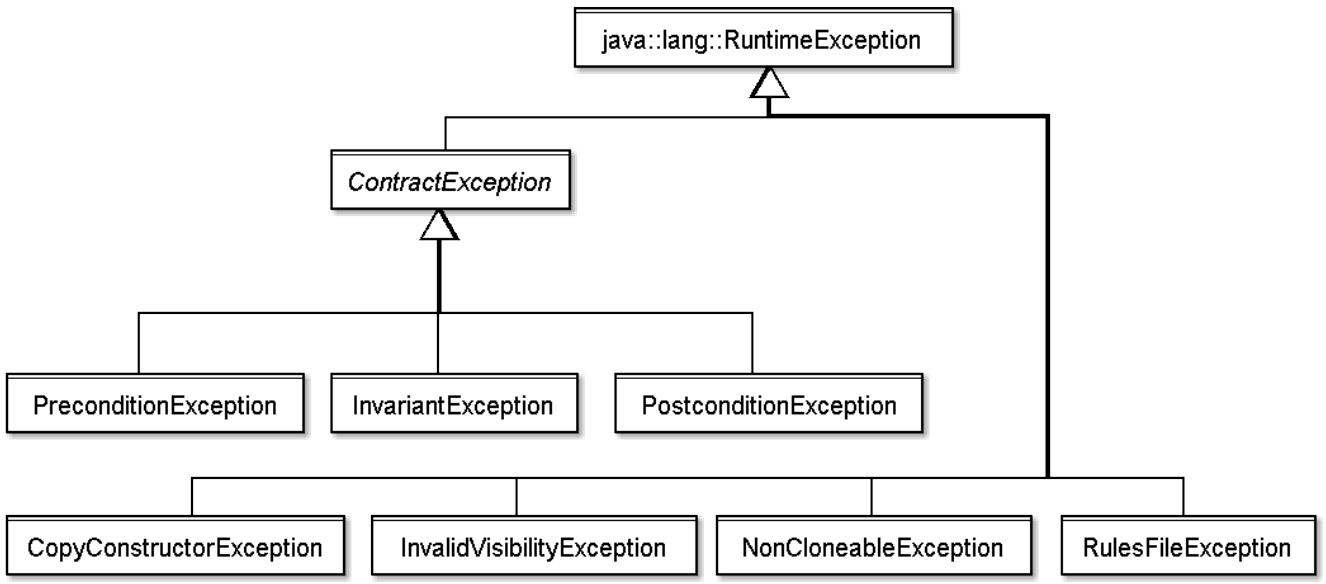

Figure 1: DbC4J main package class diagram (detail)

Exceptions inheriting from the abstract class ContractException are relative to the evaluation value of the assertions. The exceptions that inherit directly from RuntimeException, are relative to errors in the writing of the assertions.

Notice that these are runtime exceptions, which means you don't have to catch them (and you shouldn't). Other exceptions include: the CopyConstructorException, which occurs if you try to use write public contracts on an extended class with no copy constructor; the InvalidVisibilityException, which occurs if you try to write a package or protected

contract; the NonCloneableException, which occurs if you try to use the old mechanism in postconditions with a class that does not implement Cloneable; the RulesFileException, which occurs if you try to load a malformed rules file.

Static methods

When defining pre/postconditions for static methods, the methods must also be declared as static.

Example:

```
/* original static method */
public static String load() {
   // ...
}

/* static precondition */
public static String preLoad() {
   // ...
}
```

Constructors

Constructors are also evaluated by the contract. However, the class invariants will only be evaluated after the constructor invocation. This happens because class invariants evaluate the internal state of the class, as such it makes no sense in evaluating them before the actual creation of the class instance. The preconditions/postconditions naming rules for constructors are the same as for method execution. As in void returning methods, there is no result variable.

Public vs. Private assertions

So far, as shown in the previous examples, assertions are written as public methods. However, when coming to overloading (see the next section), some changes are required. The simplest solution is declare assertions as private methods. Alternatively, if you want to keep your contract public, then you must supply a public *copy constructor* in your overridden class.

Example:

```
/* private contracts version */

class A
{
    private int n;

    public A(int n) {
        //...
    }

    public int getN() {
        return n;
    }

    public setN(int n) {
        // ...
    }
    private preSetN(int n) {
        return n > 0;
    }

}

class B extends A
{
    public setN(int n) {
        // ...
    }
    private preSetN(int n) {
        return n == 0;
    }
}
```

```
/* public contracts version */

class A
{
    private int n;

    public A(int n) {
        //...
    }

    // copy constructor
    public A(A original) {
        this(original.getN());
    }

    public int getN() {
        return n;
    }

    public setN(int n) {
        // ...
```

```
    }
    public preSetN(int n) {
        return n > 0;
    }

}

class B extends A
{
    public setN(int n) {
        // ...
    }
    public preSetN(int n) {
        return n == 0;
    }
}
```

Overloading

It is possible to overload preconditions/postconditions when using method overloading.

Example:

```
class Point foo {

/* original method */
public void setX(int x) {
   // ...
}

/* overloaded method */
public void setX(String x) {
   // ...
}

/* precondition for original method */
private boolean preSetX(int x) {
   // ...
}

/* precondition for overloaded method */
private boolean preSetX(String x) {
   // ...
}
```

While using condition overloading, the conditions signatures must be non-ambiguous. Failing to do so may generate an unexpected behaviour.

Example:

```
Class foo {

/* original method */
public void setX(int x) {
   // ...
}

/* overloaded method */
public void setX(String x) {
   // ...
}

/* precondition for original method - ambiguous */
private boolean preSetX() {
   // ...
}

/* precondition for overloaded method - never executed */
private boolean preSetX(String x) {
   // ...
}
```

Note: the behaviour shown in the previous example is dependent on the current internal algorithm for method discovery. As this may change in the future, the situation should be avoided at all.

Overriding

It is possible to use contract inheritance. In programming language theory it is said that preconditions are *contravariant*; postconditions and invariant conditions are *covariant*. This means that preconditions can only be replaced by weaker conditions, while postconditions and invariant can only be replaced by stronger conditions.

In practical terms this means that inherited preconditions are (logical) ORs; inherited postconditions are (logical) ANDs. Missing preconditions/postconditions/invariants are assumed to have value `true`. Note that this means that if a class has a method with no precondition, then a class that overrides that method cannot add a precondition (well, it can, but it always evaluates `true`!).

Java vs. AspectJ specification

The contract can be specified either on Java or in AspectJ code. Both approaches have advantages and disadvantages, and can be used together. Writing the contract in Java is the simplest way, it is only necessary to write the contract methods in the class as you would write regular code. Writing the contract in AspectJ implies creating a new file (an aspect) for each class and writing the contract methods there. The first approach is the most "correct" one, since contracts are part of the class. However, while the latter implies learning (a small subset of) the AspectJ language, it provides a greater separation of concerns, since the contract is separated from the core code, and therefore it is easier to unplug (see the "Unplugging" section for more details).

Example:

```
public class Point foo {
    //...

    public void setX(int x) {
        this.x = x;
    }

    private boolean preSetX(int x) {
        return x > 0;
    }
}
```

```
public class Point foo {
    //...

    public void setX(int x) {
        this.x = x;
    }
}

public aspect fooContract {
    private boolean Point.preSetX(int x) {
        return x > 0;
    }
}
```

If the contract will not be used in deployment, then writing the contract in AspectJ is probably the best option. If AspectJ will not be used in deployment but the contract will, then there is no option but using Java. See the next sub-section for a special case where AspectJ is particularly useful.

"Old" state

In postconditions, it is possible to use the "old" class state for comparison with the class state after the execution. Next, an example of the use of the mechanism written in the Object Constraint Language (OCL) is presented:

```
Account::withdraw(Real: amount): Real
post: old.credit - amount <= 200
```

The equivalent with DbC4J is this:

```
public class Account {

    float credit;
    // ...

    // method under contract
    public float withdraw(float amount) {
    // ...
    }
```

```
    // artificial precondition
    private boolean preWithdraw() {
        ContractMemory.remember();
        return true;
    }

    // actual postcondition
    private boolean postWithdraw(float result, float amount) {
        Account old = (Account) ContractMemory.old();
        return old.credit - amount <= 200;
    }

}
```

As seen in the example, when using the old variable, it must stored previously through the ContractMemory static class. As such, there is a need to create an artificial precondition to store the old state. The solution used for this mechanism is not particularly elegant in execution, but the Java language does not allow to introduce the old variable in another way. Notice that the remembered object must be from a cloneable class. Attempting to remember a non-cloneable class will produce a NonCloneableException.

In the previous example, although the whole Account object was stored, we could instead only stored the credit (since the class can occupy several megabytes of memory and we may only need a couple of bytes for the integer value). To do so, we need to use different methods, since the credit is a primitive type (an int, in this case).

The same example this way would be:

```
import pt.unl.fct.di.dbc4j.ContractMemory;

public class Account {

    int credit;
    // ...

    // method under contract
    public float withdraw(float amount) {
    // ...
    }

    // artificial precondition
    private boolean preWithdraw() {
        ContractMemory.observe("Account.credit", credit);
        return true;
    }

    // actual postcondition
    private boolean postWithdraw(float result, float amount) {
        int oldCredit = ContractMemory.attribute("Account.credit");
        return oldCredit - amount <= 200;
    }

}
```

Note that the first argument of observe is an arbitrary tag, used to identify the primitive type. We recommend a convention though, such as the *class.method* name.

Furthermore, the reference to the ContractMemory or ContractRules classes in the contract, creates an unwanted coupling between the core code and its contract (and not only the way around). To avoid this, one solution is to use AspectJ for specifying contract conditions with the old mechanism.

Static rule configuration

Imagine that in your project you do not want the check all classes, are just want to check preconditions (e.g. for performance reasons). To do so, you need to restrict contract checking. In DbC4J this is possible to achieve through rules. The simplest way to do this is adding a "rules.properties" file to the root directory of your project.

This file looks something like this:

```
default.checks=pre
no.checks.classes=acme.Xpto
pre.checks.classes=
post.checks.classes=
all.checks.classes=acme.Foo;acme.Bar
```

The first line is mandatory and configures the default assertion checking level for classes:

- no: no checks are made;
- pre: only preconditions are checked;
- post: both preconditions as well as postconditions are checked;
- all: preconditions, postconditions and invariants are checked.

The following lines' field values are optional, and configure the assertion checking level to specific classes. Each of these values is a list with class names (complete, including package path), separated by a semi-colon (';').

If the file is not present in your project, then the default rules are checking all assertions, with no specific class rules.

Note that even if you set the configuration to no checks for all classes, DbC4J will introduce a performance overhead, and increase the *bytecode* size.

Run-time rule configuration

Besides static rules configuration, you might want to change this configuration during execution. As such, there is a simple API to do so, through the ContractRules class.

Imagine that you are using the default configuration, but at some point of execution you change it. From that point, you want to check only preconditions, except for two specific classes – Foo and Bar, with no and full checking, respectively.

This is how it would look:

```
import pt.unl.fct.di.dbc4j.ContractRules;

public class SomeClass {
    (...)
    public static void someMethod() {
        (...)
        Rules rules = ContractRules.getRules();
        rules.setDefaultChecks(CheckLevel.PRE);
        rules.addClass("Foo", CheckLevel.NO);
        rules.addClass("Bar", CheckLevel.ALL);
        (...)
    }
}
```

Note that it is possible to use both static and run-time configuration in the same application.

"Exposed" code

In some situations, one might want temporarily violate a contract with an operation, only to recover it in subsequent operations. Here is an example:

```
public class Person {
    String name;
    String address;
    (...)
}

public class Client extends Person {
    Integer idNumber;

    public boolean invariant() {
        return
            (getIdNumber() == null && getAddress() == null)
            || (getIdNumber() != null && getAddress() != null)
        ;
    (...)
}
```

In the example above, due to a *business rule*, in instances of the SpecialPerson class, the person id number and address must both be set, or not set at all. One might attempt to do the a method that sets both attributes at the same time:

```
public class Person {
    (...)
    public void setAddress(String address) { (...) }
}

public class Client extends Person {

    public void setIdNumber(Integer idNumber) { (...) }

    public void setAddressAndIdNumber(String address, Integer idNumber) {
        setAddress(address);
        setIdNumber(idNumber);
    }
}
```

This will not work, because both setAddress() and setIdNumber() are public, and the invariant will be checked before and after them. Although, we might want to rewrite the setters as private, setAddress() is inherited, and rewriting might affect other code. (and maybe we would like to continue using the separate setters!)

The issue here is that we would like to atomise both setters as one, without changing the Person class. The expose mechanism, supplied in the Rules class, offers this option:

```
public class Client extends Person {

    public void setAddressAndIdNumber(String address, Integer idNumber) {
        Rules rules = ContractRules.getRules();
        rules.expose(Client.class);
        setAddress(address);
        setIdNumber(idNumber);
        rules.unexpose(Client.class);
    }
}
```

Calls made between an expose() and an unexpose() to instances of the specified class will not be checked for contracts.

Limitations

There are some limitations when using DbC4J, namely:

- the contract only applies to non-private methods and constructors (this is an approach limitation, not a technical one);
- if you choose to write contract assertions as private, you will get unused method warnings; you can fix this by disabling these type of warnings through your IDE, or by adding an annotation (`@SuppressWarnings("unused")`);
- in the postconditions of `void` methods, the result is never an argument, since in the Java language `void` is not a type but a *pseudo-type* (by analogy we don't allow the `Void` wrapper class either, although we could);
- it is not possible to define contract for contract assertions (e.g. a precondition for a precondition is not evaluated);
- the execution performance is greatly degraded with DbC4J (while tolerable for development, it is an issue for deployment);
- contract is never checked for classes in the DbC4J and J2SE Platform packages (usually you don't want to do it, but you could, since it is open-source);
- introducing DbC4J in your code should not change the behaviour of the application, as long as contract assertions do not change the state of the objects – this is not checked automatically: it is up to the programmer to assure it;
- method calls inside your assertion will not be checked for contract, in order to avoid infinite recursivity;
- classes in which you use `ContractMemory.old()` must implement the `Cloneable` interface and implement the `clone()` method (this is due to a Java technical limitation);
- contracts for methods overridden from `Object` cannot be contracted, since their use is inherent with Java regular use.

Unplugging

After adding the DbC4J library, you might choose to unplug it from your project. For example, you might want to produce a production build, and don't want the overhead of the library. In order to do so, you should follow these steps:

1. Remove the "AspectJ InPath" reference to `dbc4j.jar`;
2. Edit the "Java Build Path" reference to `dbc4j-placebo.jar`.
3. If you are using Eclipse, you can also convert your project to a standard Java project. To do so, just right-click on you project and choose "AspectJ Tools->Remove AspectJ Capability".

That should be enough. You might need to rebuild your project/workspace, if you are using an IDE.

Note that the described procedure will only remove the execution overhead to your application. Your contract assertions present in classes will still be compiled. If you want to remove this overhead from the *bytecode*, you need to specify your contracts using AspectJ (see "Java vs. AspectJ specification" section for more information).

Unit Testing

Design by Contract has an interesting synergy with Unit Testing. JUnit is the original and most popular implementation for Java, which includes integration with most popular IDEs. Writing unit tests with DbC4J is simple, just write some tests that try to break the contracts and test them with `PreconditionException` or `InvariantException` (you don't need to test `PostconditionException` – if it is thrown, then there definitely a bug in your code).

Where is an example class:

```
public class Simple2DCoordinates {
    private int latitude;  // degrees
```

```
    private int longitude; // degrees

    public boolean invariant() {
        return
            (getLatitude() >= -90 && getLatitude() <= 90)
            &&
            (getLongitude() >= -180 && getLongitude() <= 180);
    }

    public Simple2DCoordinates(int latitude, int longitude) {
        this.latitude = latitude;
        this.longitude = longitude;
    }

    public Simple2DCoordinates() {
        this(0, 0);
    }

    // getters and setters...

    public static Simple2DCoordinates parse(String str) {
        String tokens[] = str.split(" ");
        int latitude =
            Integer.parseInt(tokens[0].substring(0, tokens[0].length()-1));
        int longitude =
            Integer.parseInt(tokens[1].substring(0, tokens[1].length()-1));
        Simple2DCoordinates result = new Simple2DCoordinates(latitude, longitude);
        return result;
    }
    public static boolean preParse(String str) {
        return str.matches("(-)?(\\d)?\\d\\dº (-)?(\\d)?\\d\\dº"); // example: 45º 90º
    }
}
```

And here is a test class for the previous example:

```
public class Simple2DCoordinatesTest {

    // Normal behaviour.
    @Test
    public void testConstructor() {
        @SuppressWarnings("unused")
        Simple2DCoordinates coords = new Simple2DCoordinates(14, 41);
    }

    // Invariant fails, latitude is not valid.
    @Test(expected=InvariantException.class)
    public void testConstructor1() {
        @SuppressWarnings("unused")
        Simple2DCoordinates coords = new Simple2DCoordinates(100, 100); // here
    }

    // Normal behaviour.
    @Test
    public void testDefaultConstructor() {
        Simple2DCoordinates coords = new Simple2DCoordinates();
        assertEquals(coords.getLatitude(), 0);
        assertEquals(coords.getLongitude(), 0);
    }

    // Normal behaviour.
    @Test
    public void testSetters() {
        Simple2DCoordinates coords = new Simple2DCoordinates();
        coords.setLatitude(14);
        coords.setLongitude(41);
```

```
        assertEquals(coords.getLatitude(), 14);
        assertEquals(coords.getLongitude(), 41);
    }

    // Invariant fails, longitude is not valid.
    @Test(expected=InvariantException.class)
    public void testSetters1() {
        Simple2DCoordinates coords = new Simple2DCoordinates();
        coords.setLatitude(90);
        coords.setLongitude(200); // here
    }

    // Normal behaviour.
    @Test
    public void testParse() {
        String input = "45º 90º";
        Simple2DCoordinates coords = Simple2DCoordinates.parse(input);
        assertEquals(coords.getLatitude(), 45);
        assertEquals(coords.getLongitude(), 90);
    }

    // Normal behaviour.
    @Test
    public void testParse1() {
        String input = "-45º 180º";
        Simple2DCoordinates coords = Simple2DCoordinates.parse(input);
        assertEquals(coords.getLatitude(), -45);
        assertEquals(coords.getLongitude(), 180);
    }

    // Normal behaviour.
    @Test
    public void testParse2() {
        String input = "-90º -180º";
        Simple2DCoordinates coords = Simple2DCoordinates.parse(input);
        assertEquals(coords.getLatitude(), -90);
        assertEquals(coords.getLongitude(), -180);
    }

    // Invariant fails, latitude is not valid.
    @Test(expected=InvariantException.class)
    public void testParse3() {
        String input = "91º -180º";
        Simple2DCoordinates coords = Simple2DCoordinates.parse(input);
        assertEquals(coords.getLatitude(), 91);
        assertEquals(coords.getLongitude(), -180);
    }

    // Precondition fails, input is malformed.
    @Test(expected=PreconditionException.class)
    public void testParse4() {
        String input = "91 -180";
        @SuppressWarnings("unused")
        Simple2DCoordinates coords = Simple2DCoordinates.parse(input); // here
    }
}
```

Further Reading

If you would like to know more about the subjects mentioned in this manual, try the following references.

Design by Contract:

- B. Meyer, *Object-Oriented Software Construction (second edition)*, Prentice-Hall, 1997
- R. Mitchell and J. McKim, *Design by Contract, by Example*, Addison-Wesley, 2002

Java:

- Sun Microsystems, *Java 2 Platform Standard Edition 5.0 API Reference*, 2004, available at http://java.sun.com/j2se/1.5.0/docs/api/index.html
- J. Gosling, B. Joy, G. Steele and G. Bracha, *The Java Language Specification (third edition)*, Prentice-Hall, 2005, available at http://java.sun.com/docs/books/jls/
- T. Lindholm and F. Yellin, *The Java Virtual Machine Specification (second edition)*, Prentice-Hall, 1999, available at http://java.sun.com/docs/books/vmspec/

AspectJ:

- R. Laddad, *AspectJ in Action*, Manning, 2003
- AspectJ Team, *The AspectJ Programming Guide*, 2003, available at http://www.eclipse.org/aspectj/doc/released/progguide/index.html
- AspectJ Team, *The AspectJ 5 Development Kit Developer's Notebook*, 2005, available at http://www.eclipse.org/aspectj/doc/released/adk15notebook/index.html
- AspectJ Team, *The AspectJ Development Environment Guide*, 2005, available at http://www.eclipse.org/aspectj/doc/released/devguide/index.html

JUnit:

- K. Beck, *Test-Driven Development: By Example*, Addison-Wesley, 2002
- J. B. Rainsberger, *JUnit Recipes: Practical Methods for Programmer Testing*, Manning, 2004
- G. Doshi, *JUnit 4.0 in 10 minutes*, 2006, available at http://www.instrumentalservices.com/media/articles/java/junit4/JUnit4.pdf.