1 Introduction

This tutorial offers a practical and applied introduction to Design by Contract (DBC) for Java using the Java Modeling Language (JML) [4]. JML is a behavioral interface specification language that extends Java with support for DBC and non-null annotations, among other features. It has a Java-based syntax and semantics and is therefore easy to learn for Java programmers. Participants will be shown, and given an opportunity to work with, the most popular JML tools, namely: the JML compiler, ESC/Java2, and JmlUnit. The JML compiler offers support for the run-time checking of JML specifications. ESC/Java2, is an extended static checker that provides a compiler-like interface to fully automated checking of JML specifications [2, 3]. Like similar tools, ESC/Java2 compromises soundness and completeness for efficacy and utility. JmlUnit is a tool for generating JUnit test suites using JML specifications as test oracles. This tutorial will allow participants to get an appreciation for JML, and how these three main tools can help developers write accurate specifications and correct code.

1.1 Technical Objectives

Attendees will come out of this tutorial with first hand experience in using JML to write contracts for Java code while using run-time and compile-time checkers to "debug" specifications and formally verify the correctness of code.

1.2 Attendee Background and Intended Audience

Attendees are expected to be experienced with **Java** program development, and familiar with the basics of **Design by Contract** (assertions, method pre- and postconditions, and class invariants). The intended audience is Java developers wishing to make use of DBC, or educators interested in teaching applied DBC for Java.

1.3 Real Objectives --- READ THIS

The main objective of this tutorial is for you to have fun exploring the world of Java interface specifications via JML and its tools. When exploring new territory, you can usually have the most fun by striking a balance between the freedom to explore ("What happens if I try this?") vs. following the guided tour ("Now do this, then try this, …"). Getting the right balance is essential to "enjoying the trip without getting lost".

These tutorial notes have been written in a way that will allow you to work semi-independently. We believe that this is a more stimulating way to learn. Yet at the same time, we will require that all of us meet up at certain specified "synchronization points" (usually at the end of a section). The tutor will be moving through the set material at a set pace, supplementing the notes, offering guidance and answering questions along the way. Ah, one last thing. If you come across one of these "answer" boxes:

.

it is not because of a misprint. It is because you are expected to provide an answer. How can you find an answer? Some answers will be obvious when you actually perform the tutorial and look at the tool output. In other cases you might have to reflect about the situation while consulting the

- JML Reference Manual (http://www.jmlspecs.org/jmlrefman) [6]

Of course, we will be going over answers during the tutorial. I hope you enjoy the learning experience.

Patrice Chalin
DSRG, Concordia University

2 Preparing your Eclipse tutorial working environment

During this tutorial you will be working from within Eclipse. The following setup should have been performed for you, in case it has not, or if you want to setup the tutorial on your own machine which is running Eclipse, here are the details.

2.1 Installing the JML, ESC/Java2 and supporting tools package

- Obtain a copy of the tools from here: http://www.cs.concordia.ca/~chalin/Tutorials/FM06/jml-fm06.tar.gz.
- Unzip and untar under, say, `/pkg` (or C:\pkg). We will call this path **JML_FM06_TOOLS_DIR**.
- Ensure that you have a Java 1.4 JRE or SDK installed and that your Eclipse has been configured to support that version of Java.

2.2 Tutorial setup in Eclipse

- Obtain a copy of the tutorial project from here: http://www.cs.concordia.ca/~chalin/Tutorials/FM06/jml-proj.tar.gz.
- Launch Eclipse.
- Under the Window menu, select Preferences, then `Java` >> `Build Path` >> `Classpath Variables`. Select `New` and define `JML_FM06_TOOLS_DIR` to be the path where you installed the tools (`/pkg` if you followed the recommendation given above).
- Choose "Import …" from the File menu, then "Existing Projects into Workspace" and select `jml-proj.tar.gz`.
- If the project is set to be built automatically, then disable auto build.
- If your **JML_FM06_TOOLS_DIR** is *not* `/pkg`, then update the location of the `tools.dir` property in the `build.xml` file (it is at the top of the file).

2.3 Tutorial setup sanity check

- Ensure that Eclipse is in the *Java perspective*.
- Locate the **JML_FM06** project in the Project Navigator.
- Examine the content of the `org.dsrg.fm06.dbc` package (under `src`). You should find:
  - `Counter.java`, a simple counter class.
  - `TestCounter.java`, a JUnit test case exercising some of the functionality of Counter.java.
  - `Main.java`, the test runner for the JUnit test case (only needed if you are working outside Eclipse).
- While the project is selected in the Project Navigator, select "Build Project". No errors should be reported.
- From the project navigator, select Run As >> JUnit Test on the file `org.dsrg.fm06.dbc.TestCase`. You should get a green JUnit bar.

3 Very Brief Review of DBC and JML

3.1 What is Design by Contract? (Optional reading)

In this tutorial you will about to learn *how* to write contracts for your Java programs using JML [4-6]. While it is assumed that you know about DBC, here is a small reminder.

Design by contract (DBC) refers to a *method* of developing object-oriented software defined by Bertrand Meyer [7, 8]. The main concept that underlies DBC is the notion of a precise and formally specified agreement between a class and its clients. Such an agreement, named a *contract* in DBC, is called a *behavioral interface specification* (BIS) in its most general form [9]. Contracts and BISs are built from class invariants, method pre- and post-conditions, (and other constructs) which are expressed by means of *program assertions*. DBC as a programming language feature refers to a limited form of support for BISs where assertions are restricted to be expressions that are *executable*. (We stress that it is the *individual* assertion expressions that are restricted to being executable, not entire method or class contracts.)

A **key point** of BIS (and hence DBC) is that an interface specification binds two parties (a client and a supplier) and that both parties can be held accountable (e.g. for buggy behavior) relative to what is expressed in the interface specification. If an aspect of the behavior is not expressed in an interface specification, then it cannot be counted on. Hence the importance of having complete specifications!

3.2 How do I write contracts in JML?

JML annotations are contained inside Java comments that start with an @ character as the very first character. For example, an inline assertion could be written inside a method body like this:

```
//@ assert 0 == testee.get();
```

Method preconditions and postconditions are introduced by the `requires`, and `ensures` keywords respectively. E.g.

```
/*@ requires j != 0;
  @ ensures \result == i/j;     // the @ at the start of this line is optional
  @*/                           // the @ at the start of this line is optional
int myDiv(int i, int j) {
  return i/j;
}
```

4 Applied DBC

4.1 Check point

- Ensure that the steps in Section 2, "Preparing your Eclipse tutorial working environment" have been completed.
- Launch Eclipse and open the Java Perspective.
- Locate the **JML_FM06** project in the Project Navigator.
- Ensure that 'Build Automatically' is not enabled for this project.
- Examine the content of the `org.dsrg.fm06.dbc` package (under `src`). You should find:
  - `Counter.java`, a simple counter class.
  - `TestCounter.java`, a JUnit test case exercising some of the functionality of `Counter.java`.
  - `Main.java`, the test runner for the JUnit test case (only needed if you are working outside Eclipse).
- While the project is selected in the Project Navigator, select "Build Project".
  No errors should be reported.
- From the project navigator, select Run As >> JUnit Test on the file `org.dsrg.fm06.dbc.TestCase`. You should get a green JUnit bar.

4.2 Objectives

Let us jump right in and start using JML and ESC/Jav2! The main objectives of this first part of the tutorial are to have you:

- Use ESC/Java2 to statically (i.e. at compile-time) *prove* that the JUnit tests should pass.
- Write your first JML assertions and class contracts (in order to achieve the first objective).
- Use both JMLc and ESC/Java2 to help debug your specifications and code along the way.

4.3 Getting ESC/Java2 to prove assertion (1) of TestCounter

- Open the TestCounter class.
- Convert the last line of `testGet()` to an inlined JML assertion by ensuring that the line starts with '//@' rather than '//'. Notes:
  - The line of interest is marked with (1).
  - Do not change any other lines into inlined assertions yet.
- Save and build. What error is reported?

- Fix the error in the Counter class by declaring `get()` to be [ ]. I.e. add
  `/*@ [ ] */`
  in front of the declaration of get(). This essentially declares `get()` to be side-effect free.
- Save and build. What error is reported now?

- Why do you think that ESC/Java2 cannot prove the stated property? (The answer to this question is fundamental. For a hint, read the last paragraph of Section 3.1 on page 3.)

- Add a postcondition (i.e. `ensures` clauses) to the Counter constructor so that ESC/Java2 will know about the essential behavior of the constructor. I.e. add the following line just above the constructor:
  `//@ ensures` [ ]
- Save and build. Are there still errors? What is the error?

- Add a postcondition to the Counter `get()` method so that ESC/Java2 will know about the essential behavior of this method. I.e. add:
  `//@ ensures` [ ]
- Save and build. If there are still some errors, carefully review your postconditions.

4.4 Getting ESC/Java2 to prove assertions (2) and (3) of TestCounter

- In the TestCounter class, convert the last line of `testInc()` to an inlined JML assertion by ensuring that the line starts with '`//@`' rather than '`//`'.
- Save and build. What error is reported?

- Add a postcondition to Counter's `inc()` method in order to help out ESC/Java2. You will need to make use of JML's [ ] operator which allows you to refer to the [ ] of a given expression.
  `//@ ensures` [ ]
- Save and build.

- ESC/Java2 still cannot prove assertion (2)? It might be a bit tricky to get the contract of `get()` right. We will use JML's run-time assertion checker (RAC) to help give us a hint as to what might be wrong with your contract for `get()`. Do the following.
- Uncomment the `testManyInc()` in TestCounter.
- Save and build.
- Run TestCounter as a JUnit test.
- What error is reported? This should give you a hint as to why your contract for `get()` is not correct.

- Correct the contract of `inc()`.
- Save and build.
- Modify the for loop range so that running the `testManyInc()` method passes.

4.5 Sync Point

By the end of this section your two classes should look like this (you still have a few blanks to fill-in; answers will be given during the tutorial):

```
package org.dsrg.fm06.dbc;

public class TestCounter extends TestCase {

   // Note that we are purposefully not using setUp().

   public void testGet() throws Exception {
      Counter testee = new Counter();
      assertEquals(0, testee.get());
      //@ assert 0 == testee.get();        // (1)
   }

   public void testInc() throws Exception {
      Counter testee = new Counter();
      testee.inc();
      assertEquals(1, testee.get());
      //@ assert 1 == testee.get();        // (2)
   }

   public void testManyInc() throws Exception {
      Counter testee = new Counter();
      for(int i = 0; i < [      ]; i++) {
         testee.inc();
         assertEquals(i+1, testee.get());
         //@ assert i+1 == testee.get();   // (3)
      }
   }
}
```

```
package org.dsrg.fm06.dbc;

public class Counter {

   public byte _count; // temporarily declared public

   //@ ensures [                    ]
   public Counter() { _count = 0; }

   //@ ensures [                    ]
   /*@ [      ] */ public int get() { return _count; }

   //@ [                                ]
   //@ [                                ]
   public void inc() { _count++; }

}
```

5 Beyond DBC: Model variables, heavyweights and more

In this part of the tutorial we will explore BIS features that are beyond DBC, and that we believe are essential to enable developers to write complete interface specifications. For a more complete discussion of the features please refer to [1].

5.1 Setup

- Make a copy of the `org.dsrg.fm06.dbc` package, renaming the copy to `org.dsrg.fm06.jml`.
- Edit the project's build.xml file: rename `proj.pkg.unqualified` to `jml`.
- Save, build and run tests.
- No errors should be reported.
- From this point on we will be working on files in the `jml` package.

5.2 Using model variables

- Change the visibility of the Counter's count field to `private`.
- Save and build. You should get an error like this:
```
File "src\org\dsrg\fm06\jml\Counter.java", line 7, character 30 error:
Field "_count" (                    ) can not be referenced in a
specification context of                    [JML] …
```
- JML supports the notion of specification visibility. Notes:
  - By default, the visibility of, say, a method contract is the same as that of the method it is specifying.
  - A public specification cannot mention protected or private class attributes.

  In a typical design of the Counter class the _count field private; how then can we refer to this field in public specifications? We do so by introducing a **specification only field** that will ***model*** the _count field without exposing it.
- Add the following lines to the Counter class:
```
//@ public model byte count;
//@ private represents count = _count;
```
- Replace occurrences of _count in assertions by count.
- Save, build and run.
  [Extra & optional] Comment out the represents clause. Save and build. Which errors are reported? (Restore the represents clause before proceeding.)
  In the usual style of specification in JML, we tend to use model fields in contracts rather than method calls. Replace all occurrences of "`get()`" that occur in JML assertions with "count".
- Save, build and run. Any errors?

5.3 Full behavioral specifications

The current contract for `inc()` simply states that it must not be called when `_count` is greater than or equal to 127. We can complete the behavioral specification of `inc()` by mandating that `inc()` throw an `IllegalStateException` when `_count >= 127`. To achieve this we will make use of the "heavyweight" format of method specifications.

5.3.1 Heavyweight method specifications (`normal_behavior`)

- As a first step towards extending the specification of inc() we will need to change its contract to make use of the "heavy weight" format; i.e., change it to

```
/*@ public normal_behavior
  @   requires count < 127;
  @   ensures  count == \old(count) + 1;
  @*/
  public void inc() { _count++; }
```

- Save and build. The JML compiler should report a caution. What is it?

5.3.2 Frame property

- We need to specify which fields `inc()` is permitted to modify or assign to. We specify this by inserting an `assignable` clause in between the `requires` and `ensures` clauses, i.e.

```
/*@ public normal_behavior
  @   requires count < 127;
  @   assignable comma-separated-list-of-fields-here;
  @   ensures  count == \old(count) + 1;
  @*/
public void inc() { _count++; }
```

  - Note: an assignable clause defines the method's "frame property" or "frame axiom".

- Try using `_count` as a field name for the assignable clause.
- Save and build. What error is reported?

- How can you fix it—i.e. which other (real) or specification-only field can you use?
- Make the necessary changes, save and build.
- You will note that jmlc reports the following error:

```
File "src\org\dsrg\fm06\jml\Counter.java", line 23, character 25 error:
Field "_count" is not assignable by method "org.dsrg.fm06.jml.Counter.inc(
)"; only fields and fields of data groups in set "{count}" are assignable
[JML]
```

5.3.3 Datagroups

A *datagroup* is a collection of memory locations. If one member of a datagroup is listed in an assignable clause then the method can modify (assign to) any member of the datagroup. Note that a model field is considered to be a datagroup as well.

- To fix the current problem, the following annotation immediately after the declaration of `_count`:

```
private byte _count; //@ [                ];
```

- This declared _count to be in the same datagroup as count.
- Save and build.

5.3.4 Specification cases & Exceptional behavior

We want to enhance the specification of `inc()` by stating that it will throw an `IllegalStateException` when `_count >= 127`. To do so we need to add a new *specification case*. In its general form, a method contract consists of one or more specification cases separated by the `also` keyword. Since we want to describe exceptional behavior we will add an `exceptional_behavior` specification case like so:

```
/*@ public normal_behavior
  @   requires count < 127;
  @   assignable count;
  @   ensures  count == \old(count) + 1;
  @ also public exceptional_behavior
  @   requires count >= 127;
  @   signals (IllegalStateException) [          ];
  @   signals_only IllegalStateException;
  @*/
public void inc() { _count++; }
```

- The predicate after the signals clause is required to hold in the post-state following the raising of an `IllegalStateException`. Which predicate should be required to hold? (There are not that many choices actually.)
- Save and build. The JML compiler reports a caution.
- Add a frame property (assignable clause) to the `exceptional_behavior` specification case. What should `inc()` have been permitted to change in this case: [ ]. Hence add the following line between the requires and signals clauses:

  ```
  @   assignable [          ];
  ```

- Save and build. If there are errors, warnings or cautions. Fix them, save and build.

5.4 Behavioral subtyping (HOMEWORK, i.e. for after the tutorial)

A very important property of JML is that it enforces behavioral subtyping. While a full treatment of this topic is beyond the scope of this short tutorial, we invite you to read about the topic in the advance tutorial on JML [1].

In case you would like to see behavioral subtyping at work here is an *outline* of what you might want to try.

- Make a copy of the org.dsrg.fm06.jml package renaming the copy to org.dsrg.fm06.subtyping.
- Select the Counter class, and choose Eclipse's "Extra Superclass" feature. Name the superclass, `CCounter`.
- Manually move the interface specification statements of the Counter class to `CCounter`.
- Save, build and run tests.
- Create a subclass of `CCounter` named `WrappingCounter` and modify its behavior so that inc() will wrap to zero if _count = 127.
- Duplicate the tests in TestCounter so that is exercises the functionality of this new `WrappingCounter` class.
- Save and build. Errors will be reported.
- You will be required to modify the contract of `CCounter` (especially the specification of inc()) so as to accommodate the behavior of both of its subclasses. Make the required changes.
- Save, build and run tests.
- Add a JUnit test case to TestCounter that exercises the wrapping behavior of `WrappingCounter`'s inc() method.
- Try to get ESC/Java2 to prove that this test case should pass. To do so you will need to add a subclass specific contract for `WrappingCounter`'s inc(). Note that `WrappingCoutner`'s inc() specification will need to start with "also" because it will be extending the contract given in the superclass' inc().

6 Leverage your testing using JmlUnit

Writing contracts can start to be rewarding, but if you still have to write tests by hand, that can seem like double the work (i.e. specifying and writing tests). Wouldn't it be great if you could just ask a class to test itself? Have a class tested against its own interface specification. Enter JmlUnit.

In this portion of the tutorial you will make use of JmlUnit, a tool which automatically creates JUnit test cases for you based on a class' specification. All you need to supply are test value generators (and if you get lazy, you can even setup a common generator package for reuse in most of your test suites0.

6.1 Setup

- Make a copy of the `org.dsrg.fm06.jml` package, renaming the copy to `org.dsrg.fm06.utg` (where `utg` stands for Unit Test Generation).
- Edit the project's build.xml file: rename `proj.pkg.unqualified` to `utg`.
- Copy `extra/TestDataGenerator.java` into `org.dsrg.fm06.utg`.
- Save, build and run tests (via TestCounter).
- No errors should be reported.
- From this point on we will be working on files in the `utg` package.

6.2 Generating tests using JmlUnit

- From the Package Explorer, select "Run As >> Ant Build …" on build.xml.
- In the External Tools dialog, choose the "Targets" tab and then select the `jmlunit` target.
- Click "Run". This will generate the JUnit tests.
- Ensure that two files (containing the substring "`_JML_Test`") can be found in the `utg` package.

6.3 Running the generated tests

- Select `Run As` >> `JUnit Test` on `Counter_JML_Test` (from within the Package Explorer).
- You should get a green bar.
- Got to the JUnit tab. Look as the organization of the tests. How many tests were run? Not many yet. This is partly because there are not many ways to create a Counter.

6.4 Adding a constructor and generating more tests

- To add variability, add a constructor to Counter that accepts an initial `int` count value as argument.
  - Note: ensure that you declare the argument as `int` and not `byte`.
- Ensure that you write an appropriate contract for this new constructor. (You will have to wait for the project to finish rebuilding before the new files appear in the Package Explorer.)
- Save and build. Fix any reported errors.
- Run JmlUnit again (so that tests are generated for the new constructor).
- Run the unit tests. How many test cases now? How are the tests organized?
- Make the following changes to `TestDataGenerator`: change the value of `useGetCounter2` to true.
- Save, build and re-run tests. How many test cases have been generated now?

6.5 Is this really working?

- If all went well above, you should be getting a green bar with lots of successful tests. Is this really working or are the tests merely passing? Introduce a bug (i.e. a behavior that fails to respect the contract) in the implementation of the new constructor.
- Save, build (ignore any errors reported by ESC/Java2) and re-run the unit tests.

6.6 Class invariants

- We have not seen any examples of class invariants, hence let us introduce an invariant in the Counter class. Can you think of a property which should "always" be true of a Counter. Think of properties of real and specification-only fields of Counter. Once you have found such a property, add it to the Counter class:

```
//@ public invariant                                    ;
```

- Save and build.
- Modify one of the implementation and/or the contracts of a constructor or a method so that the invariant is violated.
- Save and then re-run JmlUnit.
- Build the project (ignore any errors reported by ESC/Java2) and then run the JmlUnit tests.
- See how the RAC now reports violation of invariants?

7 Finishing up

The JML compiler and ESC/Java2 are complimentary tools that can help you debug your specifications and code during development. JmlUnit can help the ease of creating unit tests, thought it will not replace manual creation of unit tests entirely.

Time permitting, I will discuss other JML features such as:

- JML API specifications for standard libraries. This can be very helpful in debugging your code which makes use of these libraries.
- Writing specifications for libraries (for those that are not shipped with JML).
- JmlDoc (e.g. doc for the JML tools is here: http://opuntia.cs.utep.edu/utjml/jml-javadocs).

I may also give a sneak peek at some new JML features (e.g. non-null types). I encourage you to try JML out on your favorite project and hope that you enjoyed the tutorial!

8 References

[1] P. Chalin, J. Kiniry, G. T. Leavens, and E. Poll, "Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2 (Tutorial Paper)". *Fourth International Symposium on Formal Methods for Components and Objects (FMCO'05)*, 2005, 2005.
[2] D. R. Cok and J. R. Kiniry, "ESC/Java2: Uniting ESC/Java and JML". In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean editors, *Proceedings of the International Workshop on the Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS'04)*, Marseille, France, March 10-14, 2004, vol. 3362 of *LNCS*, pp. 108-128. Springer, 2004.
[3] J. R. Kiniry, "ESC/Java2", http://secure.ucd.ie/products/opensource/ESCJava2, 2005.
[4] G. T. Leavens, "The Java Modeling Language (JML)": http://www.jmlspecs.org, 2006.
[5] G. T. Leavens and Y. Cheon, "Design by Contract with JML", Draft paper, 2005.
[6] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, and P. Chalin, "JML Reference Manual", http://www.jmlspecs.org, 2006.
[7] B. Meyer, "Applying Design by Contract", *Computer*, 25(10):40-51, 1992.
[8] B. Meyer, *Object-Oriented Software Construction*, 2nd ed. Prentice-Hall, 1997.
[9] J. M. Wing, "Writing Larch Interface Language Specifications", *ACM Trans. Program. Lang. Syst.* , 9(1):1-24, 1987.