1 Configuration Management

One way in which the “soft” nature of software manifests itself is that software can be modified easily, even after production. That is, the act of bringing about changes in software is easy[1], certainly compared to changing hardware. Just open a file in an editor, modify it, and save the changes. However, this ease of change also brings some dangers.

- It may be difficult to revert a change, when the change turns out to be bad.
- Changes may get lost, when multiple developers concurrently change the same item. (Do you see how that can happen?)
- There can be confusion about which version you refer to, when you have saved multiple versions of the same item. Or which combination of versions.
- After a while, it may be difficult to find out exactly what changes were made, and when and why they were made.

That is why change must be controlled in software development projects. *Revision Control* [5] is part of the broader discipline of *Software Configuration Management* [6], which deals with all aspects of managing the various items that together play a role in a software project, in particular as they evolve over time.

2 Mercurial for Revision Control

There are many software tools available to support configuration management, and especially revision control. In this course, we will use *Mercurial* [2]. For a quick introduction to using Mercurial on the command line see [1]. Note, however, that Mercurial support is integrated into NetBeans [3], and that command-line usage of Mercurial is not needed for this course.

With Mercurial, you can put any directory tree (a folder and everything that it contains, including other folders and their contents) under revision control. Mercurial uses an optimized format to store all **revisions** of files in a so-called **repository**. This repository is hidden, so that it is not in your way. You start with an empty repository, by letting Mercurial **initialize** it for a given directory tree. Then you **add**[2] new files to the repository. At any point in time[3], you can **commit** the current state of the directory tree, by saving the latest **change set** to the repository. You can **update** the actual state of the directory tree to reflect any committed state in the past, by having it reconstructed from the repository. Thus, the repository makes it possible to trace the development process.

[1]Knowing what to change, how to change it, and whether it is the right change, can still be hard.

[2]NetBeans usually knows which files to add, and which not to add.

[3]Usually, upon completing a step in the development process. See Section 3.

3 Using Mercurial with NetBeans for TDD

Use the following workflow to do *Test-Driven Development* (TDD) [4] of software with NetBeans, using Mercurial to track the change history. You submit development work to Momotor by zipping[4] the NetBeans project folder, which includes the entire repository.

1. Create New Project:
   (a) Category: Java; Project: Java Application
   (b) Provide a Project Name, say `MercurialDemo`, and choose a Project Location.
   Do not Use Dedicated Folder for Storing Libraries.
   Do not Create Main Class.
2. Initialize Mercurial repository:
   (a) Either: Right-click[5] on the project and, from the pop-up menu, select Versioning > Initialize Mercurial Repository. . . .
   Or: Select the project and, in the menu bar, select Team > Mercurial > Initialize Repository. . . .
   (b) Confirm to use the default Root path, viz. the top-level project directory itself.
3. Add a new class:
   (a) Expand the project, in the left-hand column, by clicking on the triangle (when not expanded this triangle points to the right; when expanded, it points down).
   (b) Right-click on Source Packages, and select New > Java Class. . . .
   (c) Provide a Class Name, and, when appropriate, choose a Package[6].
   For example: `Counter`, in the default package (leave Package empty).
   (d) Enter the top-level javadoc for the class, and possibly some method signatures with javadoc, but empty bodies. For example:

```
/**
 * A Counter object keeps track of an integer count,
 * that can be inspected, incremented by one, and
 * reset to zero.  The initial value is zero.
 *
```

[4]Currently, you are also requested to submit Java source files of the last version separately.
[5]Mac: control-click
[6]In simple projects, we put classes in the default package. More complex projects will have a prescribed package structure. For `MercurialDemo` use the default package.

```java
 * @author Tom Verhoeff (TU/e)
 */
public class Counter {

    /**
     * Get the current count.
     */
    public long getCount() {
        return 0;
    }

    /**
     * Increment the count by one.
     */
    public void increment() {
    }

    /**
     * Reset the count to zero.
     */
    public void reset() {
    }

}
```

4. Review all changes:

   (a) Either: Right-click on the project and, from the pop-up menu, select Mercurial > Diff > Diff To Base.

   Or: Select the project and, in the menu bar, select Team > Diff > Diff To Base.

   This presents an overview of all changes. The first time after creating the project, this will show a couple of files that are all Locally New.

5. Commit the changes:

   (a) Either: Right-click on the project and, from the pop-up menu, select Mercurial > Commit. . . .

   Or: Select the project and, in the menu bar, select Team > Commit. . . .

   Or: In the [Diff] overview obtained in Step 4, click on the Commit All button.

   (b) Provide a sensible Commit Message.

   For example: Created specification of class Counter.

Sensible: Indicate the *purpose* of the changes, answering the *why*. Note that answers to *what* changed *where* and *how* can already be found in the repository through the Mercurial tool.

Also provide an Author, and optionally deselect files with changes that you do not want to commit yet.

Note that NetBeans is aware of which files should never be committed to the repository (either because they can be generated from the source files, such as documentation and class files, or because they contain private data, such as personal settings).

(c) After the commit completes, the [Diff] overview will be updated. When all files were committed, that overview will be empty. By default, it shows the changes with respect to the latest committed version (the so called *parent*).

6. Create test cases:

(a) Right-click on the class to be tested (click the expand triangles, until you see the class), and select Tools > Create/Update Tests.

(b) Confirm to use the default Class Name.

For instance, `CounterTest`[7].

You can ask to have Default Method Bodies, Javadoc Comments, and Source Code Hints generated.

(c) Enter the test methods.

For example:

```java
import org.junit.Test;
import static org.junit.Assert.*;

/**
 * Test cases for class Counter.
 *
 * @author Tom Verhoeff (TU/e)
 */
public class CounterTest {

    /**
     * Test of getCount method, of class Counter.
     */
    @Test
    public void testGetCount() {
        System.out.println("getCount");
        Counter instance = new Counter();
```

[7]The default name of a test class equals the name of the class being tested extended with `Test`. It will be located in the same package as the class being tested, but NetBeans stores it in a separate subdirectory within Test Pacakages.

```
        long expResult = 0L;
        long result = instance.getCount();
        assertEquals("result", expResult, result);
    }

    /**
     * Test of increment method, of class Counter.
     */
    @Test
    public void testIncrement() {
        System.out.println("increment");
        Counter instance = new Counter();
        instance.increment();
        long expResult = 1L;
        long result = instance.getCount();
        assertEquals("getCount", expResult, result);
    }

    /**
     * Test of reset method, of class Counter.
     */
    @Test
    public void testReset() {
        System.out.println("reset");
        Counter instance = new Counter();
        instance.increment();
        instance.reset();
        long expResult = 0L;
        long result = instance.getCount();
        assertEquals("getCount", expResult, result);
    }

}
```

7. Run the test cases, fix any defects in test cases, and commit your work.
   (a) Right-click on the class to be tested, and select Test File (there is also a keyboard shortcut for this).
   (b) Some of these test cases will fail (because the class under test has not yet been implemented).
   (c) Review and commit changes (see Steps 4 and 5).
   For example, with commit message Added test cases for class Counter.

8. Implement the class:
   (a) Provide a data representation through instance variables, and provide method bodies.
   For example:

```
/** The count. */
private long count;

// Representation invariant: count >= 0
// Abstraction function: count

 /**
 * Get the current count.
 */
public long getCount() {
    return count;
}

/**
 * Increment the count by one.
 */
public void increment() {
    count = count + 1;
}

/**
 * Reset the count to zero.
 */
public void reset() {
    count = 0;
}
```

(b) Test it again (see Step 7). Where necessary, improve the implementation (and/or the test cases).

(c) When happy, review all changes, via the [Diff] overview (see Step 4). You can traverse changes one by one, using the two arrow buttons.

(d) Commit the changes (see Step 5). Do not forget the commit message.

9. For submission to Momotor, create a zip archive of the NetBeans project directory, including the entire repository.

   (a) Zip the entire NetBeans project directory (including the hidden repository). Use the zip format; do not use other compression formats. For example, put it in `MercurialDemo.zip`.

   (b) Usually, you will be asked to submit the latest Java source files in your project together with the zip archive. In case of this example, you submit:

   - `CounterTest.java`

   - `Counter.java`
   - `MercurialDemo.zip`, containing `MercurialDemo` as top-level directory

10. If Momotor reports any problems or you discover defects yourself, then:

    - If your implementation is defective, then first add test cases to detect this problem yourself: see Steps 6 and 7)
    - Fix the problem(s): see Step 8.
    - Review and commit changes: see Steps 4 and 5.
    - Resubmit your work to Momotor: see Step 9.

References

[1] *Hg Init: a Mercurial tutorial.* `http://hginit.com/`

[2] *Mercurial.* `http://mercurial.selenic.com/`

[3] *NetBeans.* `http://www.netbeans.org/`

[4] Tom Verhoeff. *Test-Driven Development*. A separate note for Programming Methods (2IPC0, 2015).

[5] *Revision Control*, Wikipedia, `https://en.wikipedia.org/wiki/Revision_control`

[6] *Software Configuration Management*, Wikipedia, `https://en.wikipedia.org/wiki/Software_configuration_management`

Main author: Tom Verhoeff, Department of Mathematics & Computer Science, Eindhoven University of Technology

Minor updates: Loek Cleophas, Department of Mathematics & Computer Science, Eindhoven University of Technology