



# Code Quality, Unit Testing and Version Control workshop

# Contents

Contents.....	2
Unit Testing.....	4
1. Initial Reading .....	4
2. About the code used in the exercise .....	5
3. Setting up the environment for the exercise.....	6
4. Running tests .....	7
5. Exercises.....	11
Code Quality.....	13
6. Initial Reading .....	13
7. About the code .....	13
8. Setting up the environment for the exercise.....	13
9. Running ECL Emma .....	14
10. Add a test to improve the coverage of the “processAddition” method.....	16
11. Running FindBugs.....	17
Version Control.....	20
12. Initial Reading .....	20
13. About the code.....	20
14. Setting up the environment for the exercise.....	20
15. Connecting to the Repository .....	21
16. Sharing the code .....	22
17. Making updates.....	23
18. Conflicts.....	24
19. Branches / Tagging.....	25
20. Merging.....	26



# Unit Testing

## 1. INITIAL READING

Please study this material and the section in the Citi handbook relating to unit testing, prior to commencing on the exercises. The exercises are in java using the JUnit framework.

### (1) "Test infected"

<http://junit.sourceforge.net/doc/testinfected/testing.htm>

This article outlines what unit testing is about, and how to use the JUnit framework to write unit tests. It's based on JUnit 3.x framework. We will be using JUnit 4.0, where the framework was re-designed to use Java annotations for processing, but the same principles apply. The next article illustrates how some of those tests can be written in JUnit 4.0.

### (2) "JUnit Cookbook"

<http://junit.sourceforge.net/doc/cookbook/cookbook.htm>

This article describes how to write unit tests in JUnit 4.0.

## 2. ABOUT THE CODE USED IN THE EXERCISE

This exercise involves testing a calculator class, which models a desk calculator. Its interface looks like this:

```
public interface Calculator {  
  
    public void switchOn();  
  
    public void switchOff();  
  
    public boolean isOn();  
  
    public String getCurrentlyDisplayed();  
  
    public void keyPressed(char key) throws InvalidCalcKeyException;  
  
}
```

The class *CalculatorImpl* supplies the implementation we are testing. The calculator will update its display according to the sequence of key inputs from the user. You ‘press’ a key by calling the *keyPressed* method, and get the current display by calling the *getCurrentlyDisplayed* method.

Here’s one of the supplied test examples to illustrate how the calculator class works, and how it can be tested with JUnit.

```
@Test  
public void testSimpleAddition() throws InvalidCalcKeyException {  
    calc.switchOn();  
    calc.keyPressed('4');  
    calc.keyPressed('2');  
    calc.keyPressed('+');  
    calc.keyPressed('5');  
    calc.keyPressed('8');  
    calc.keyPressed('=');  
    Assert.assertEquals("100", calc.getCurrentlyDisplayed());  
}
```

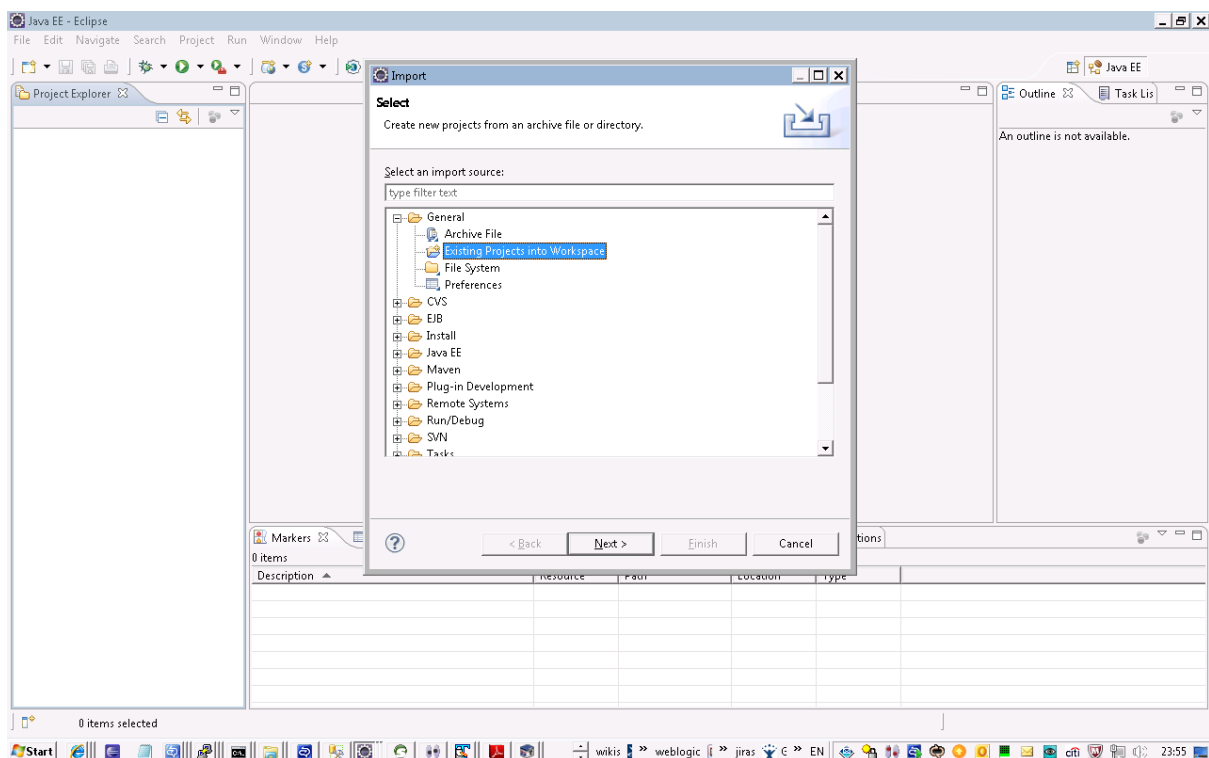
### 3. SETTING UP THE ENVIRONMENT FOR THE EXERCISE

For this exercise we will be writing and running JUnit tests within the eclipse IDE.

Pre-requisites

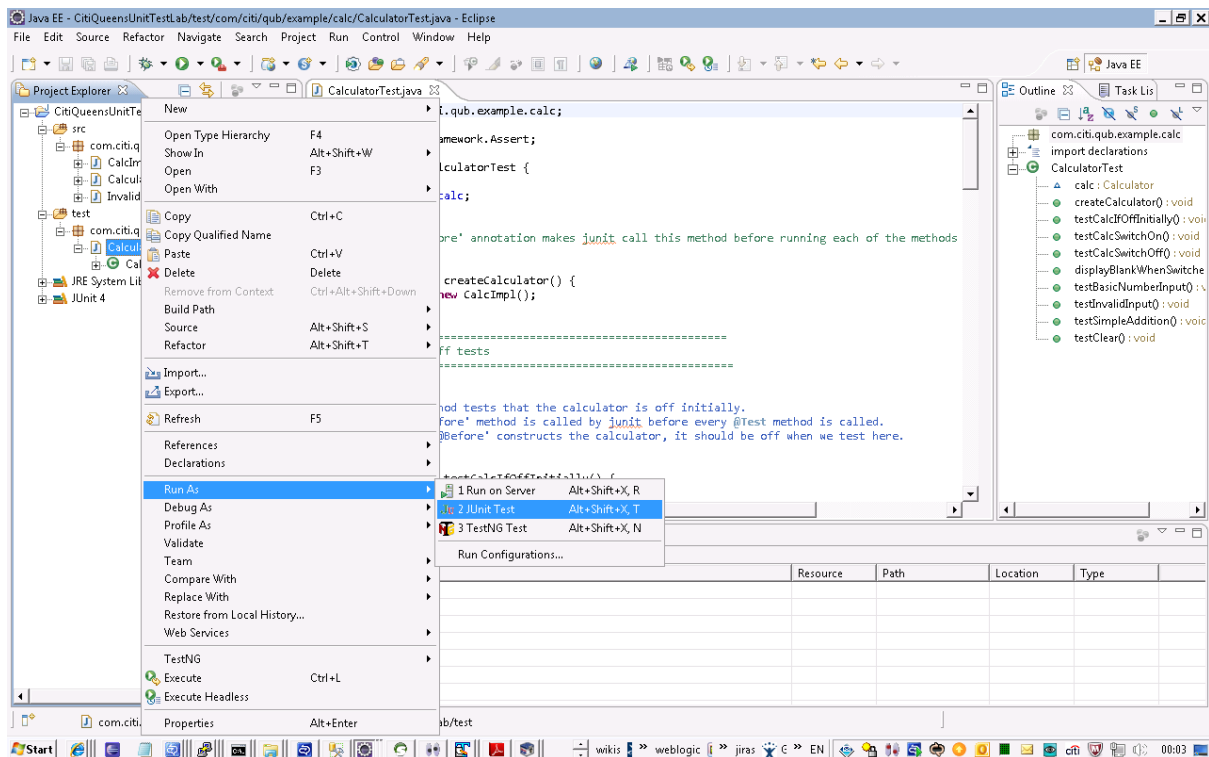
- Eclipse IDE [ version 3.3 (Helios) or higher ]
- J2SE JDK 1.6 or higher

(1). Extract the supplied code and import into eclipse as below:

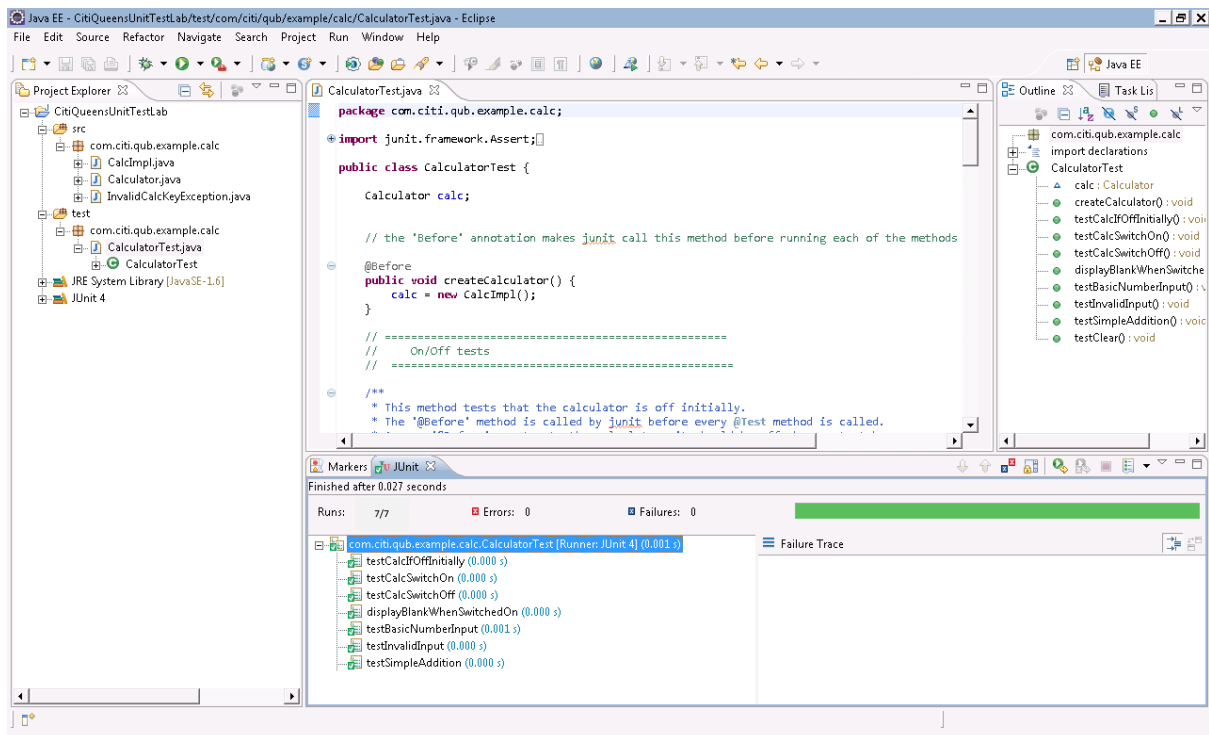


## 4. RUNNING TESTS

Try running the tests in Eclipse as follows:



The JUnit window shows that JUnit ran 7 tests and all passed. Note the green bar – this is an important visual indicator that ‘all is well’.





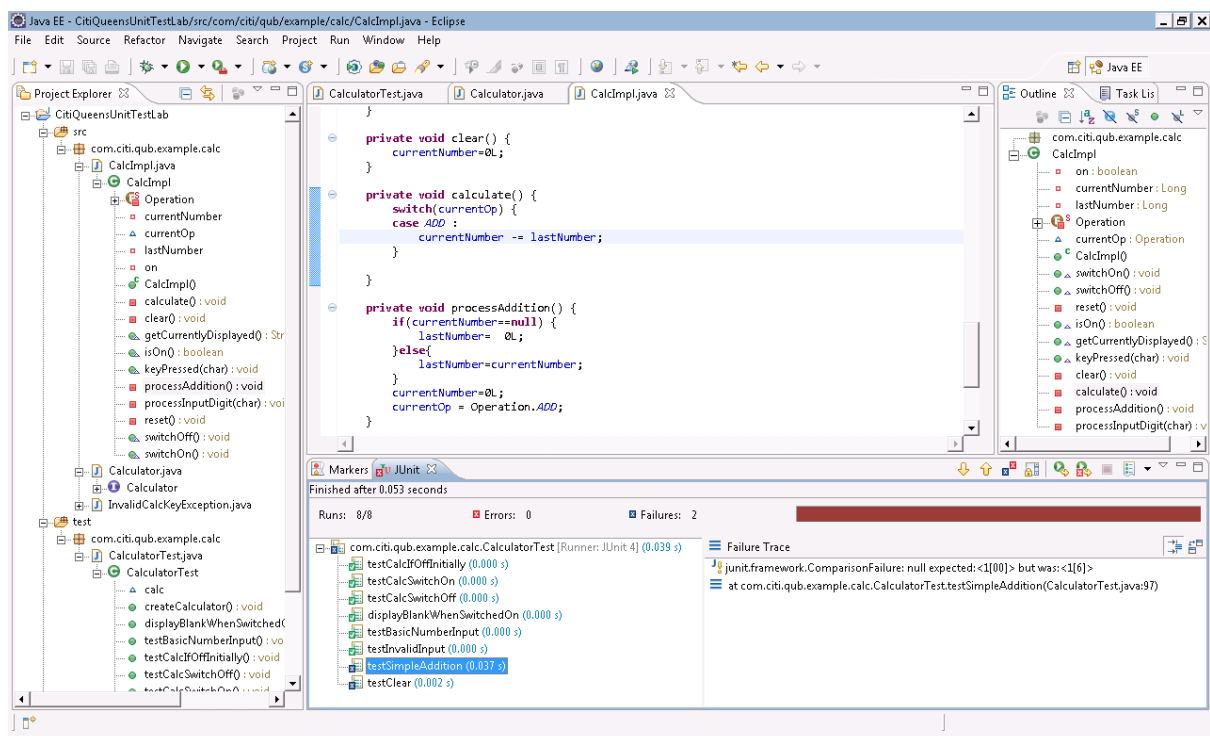
Now let's see what JUnit tell us when something goes wrong.  
Open the CalculatorImpl class and change this method as follows:

```
private void calculate() {  
    switch(currentOp) {  
        case ADD :  
            currentNumber += lastNumber;  
    }  
}
```

Change the '+' to '-' :

```
private void calculate() {  
    switch(currentOp) {  
        case ADD :  
            currentNumber -= lastNumber;  
    }  
}
```

Clearly something will break when we do this. Run the tests in Eclipse as before. You should see a red bar in the JUnit window now – indicating something is wrong.



JUnit has detected the change, as we had it covered in a test for addition, and has also told us what got broken – the *'testSimpleAddition'* method failed with this error:

```
junit.framework.ComparisonFailure: null expected:<1[00]> but was:<1[6]>
```

This is powerful tool for detecting bugs in code as you are developing.

Finally, undo the change just made and run the tests again to make sure the bar is green and therefore that all tests are passing before starting the exercises.

---

## 5. EXERCISES

---

### A ADD A TEST FOR THE 'CLEAR' OPERATION

The calculator has a 'clear' operation to remove the currently displayed number. Add a test for the 'clear' operation - when the user enters 'C', make sure 'last number' has not been cleared.

For example:

User enters '1','1','+','2','C','3','=' --> should give 14.

You should be able to run the tests and see the green bar in eclipse.

---

### B ADD A TEST FOR SUBTRACTION

Add a test for subtraction, and run the test. E.g.

User enters '8','8','-','7','7','=' --> should give 11.

You should see the test fail with an *InvalidCalcKeyException*. A red bar shows in eclipse. This is failing as the code to implement subtraction has been commented out. Now in the file `CalcImpl.java`, uncomment the code for subtraction (look for 'Uncomment for lab exercise 2'). Re-run the test. The test should now pass. (Note – there are 3 separate sections to uncomment).

---

## C ADD TESTS FOR THE MEMORY FUNCTION.

The calculator supports a memory function – the ‘S’ key stores the current number to memory, and the ‘R’ key retrieves the current value stored in memory. Add some tests to check the memory function. At least try and implement tests for:

- (i) Storing and fetching
- (ii) Using the stored memory value in an addition operation

No examples provided this time.

---

## D IMPLEMENT MULTIPLICATION

The practice of Test Driven Development (TDD) advocates creating test code before writing the real code that implements a new feature. This last exercise uses this approach.

Write a test for a multiplication operation. Run the test – it should fail with an *InvalidCalcKeyException* for the ‘\*’ key.

Once you have written the test, modify the *CalcImpl* class to implement the multiplication operation. Re-run your test until it passes – and make sure all the other tests still pass (i.e. that nothing has been broken by adding the multiplication feature).

No examples are provided this time, but you can base your work on the tests and code that exist for addition / subtraction.

# Code Quality

---

## 6. INITIAL READING

Please study the sections in the Citi handbook relating to Code Quality, specifically the sections on ECL Emma & FindBugs, prior to commencing on the exercises.

---

## 7. ABOUT THE CODE

This exercise builds upon the code created during the “Unit Testing” workshop. We will be analysing the product of this workshop for common Code Quality metrics including

- Unit Test Coverage (using ECLEmma)  
How much of the “executable” code do the unit tests actually run? This gives us an indication of how complete the unit test suite actually is
- Static Code Analysis (using FindBugs)  
Checking for common coding issues that can produce unexpected behaviour, are unnecessary or deviate from the norm.

---

## 8. SETTING UP THE ENVIRONMENT FOR THE EXERCISE

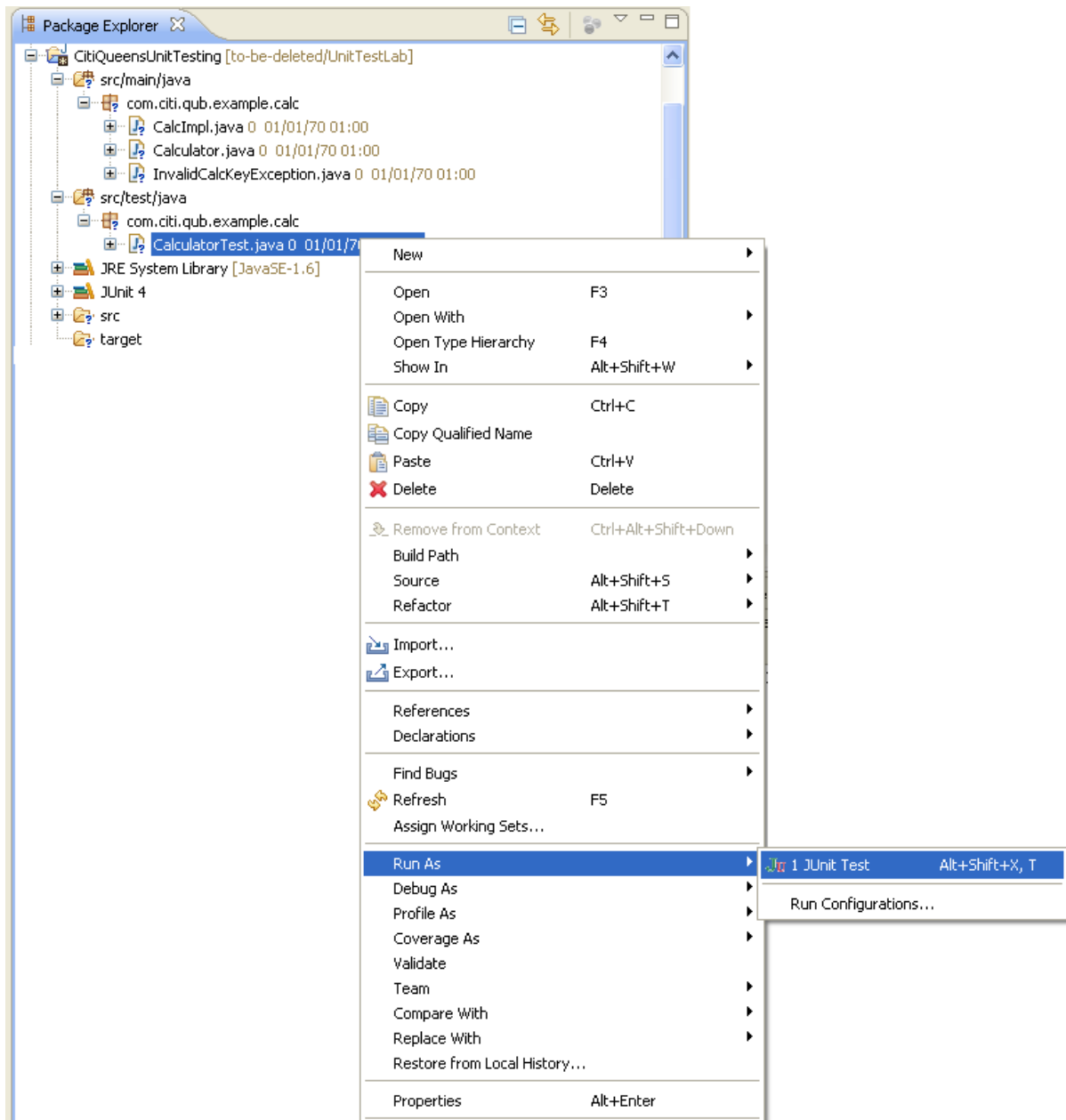
For this exercise we will be running plugins within the eclipse IDE.

Pre-requisites

- Eclipse IDE [ version 3.6 (Helios) or higher ]
- J2SE JDK 1.6 or higher
- ECLEmma (Eclipse Plugin)
- FindBugs (Eclipse Plugin)

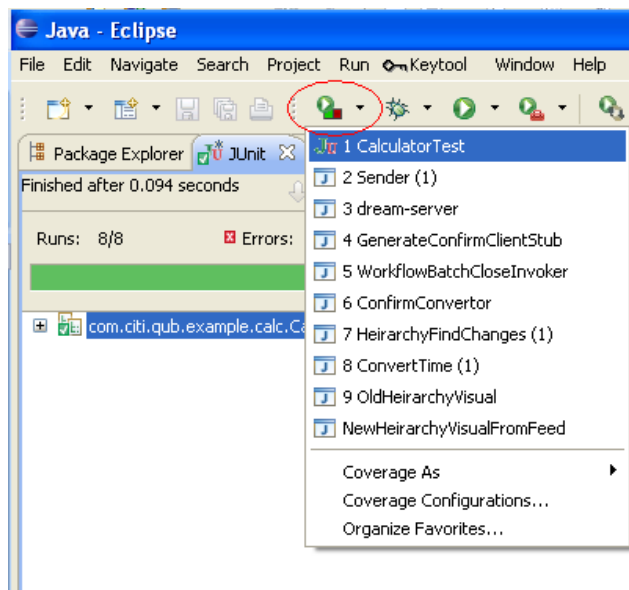
It is assumed that the user has already completed the “Unit Testing” workshop and has the resulting workspace from that workshop available to build upon.

Try running the tests in eclipse as follows:



The JUnit window shows that JUnit ran the tests and all passed. Note the green bar – this is an important visual indicator that ‘all is well’.

Now let's run the same sets of tests within ECLEmma. Note that this executes exactly the same tests in exactly the same way; ECLEmma simply monitors the code that is executed

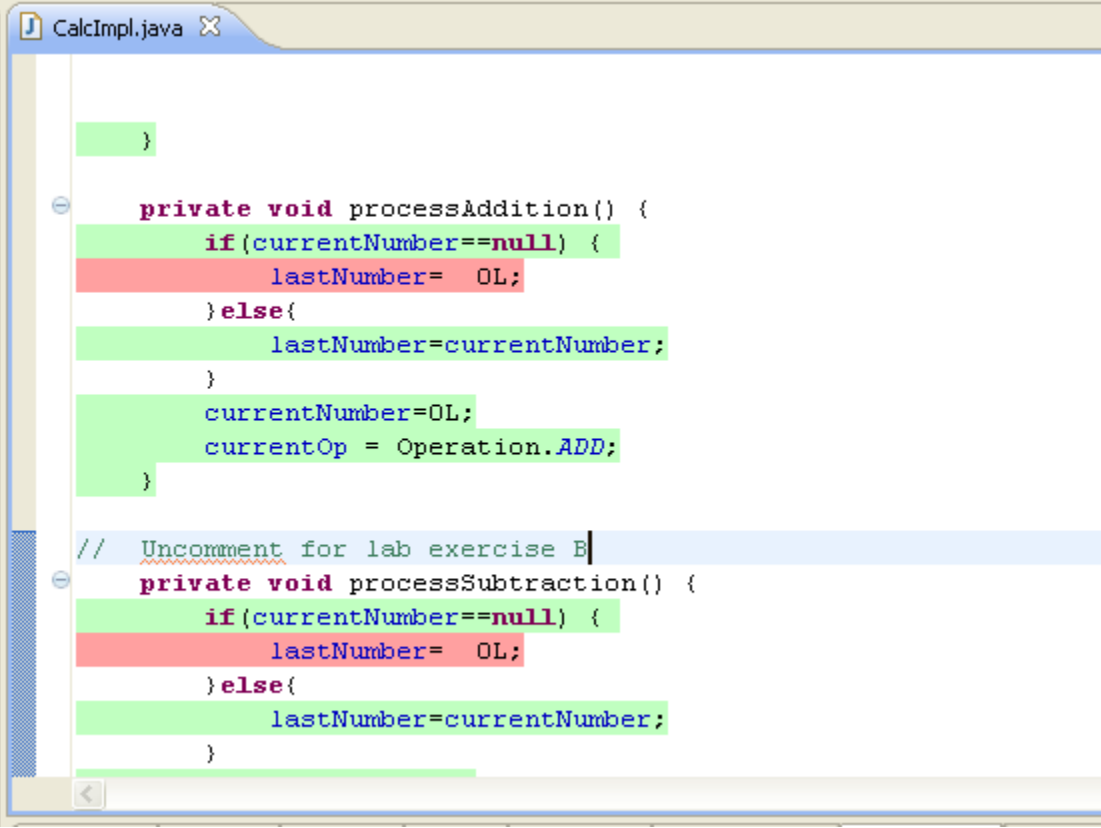


You will notice that the JUnit results window is refreshed, still showing all green. You should also notice that a new "Coverage" window has opened. Expand the project, packages & classes

Element	Coverage	Covered Instructions	Missed Instru...	Total Instructions
CitiQueensUnitTesting	95.0 %	593	31	624
src/main/java	92.7 %	294	23	317
com.citi.qub.example.calc	92.7 %	294	23	317
CalcImpl.java	92.5 %	282	23	305
CalcImpl	92.5 %	282	23	305
Operation	91.7 %	55	5	60
processAddition()	75.0 %	15	5	20
processMultiply()	75.0 %	15	5	20
processSubtraction()	75.0 %	15	5	20
CalcImpl()	100.0 %	5	0	5
calculate()	100.0 %	39	0	39
clear()	100.0 %	5	0	5
getCurrentlyDisplayed()	100.0 %	9	0	9
isOn()	100.0 %	3	0	3
keyPressed(char)	100.0 %	40	0	40
memRetrieve()	100.0 %	5	0	5
memStore()	100.0 %	5	0	5
processInputDigit(char)	100.0 %	23	0	23
reset()	100.0 %	10	0	10
switchOff()	100.0 %	3	0	3
switchOn()	100.0 %	4	0	4
InvalidCalcKeyException.java	100.0 %	12	0	12
src/test/java	97.4 %	299	8	307

This window allows us to see what percentage of the code was executed while the tests ran, we can also drill down and view the coverage percentage per package, class and even method.

Now, double click on one of the methods showing less than 80% coverage, for example, in this case, “processAddition”.



```
CalcImpl.java X
}
private void processAddition() {
    if(currentNumber==null) {
        lastNumber= 0L;
    }else{
        lastNumber=currentNumber;
    }
    currentNumber=0L;
    currentOp = Operation.ADD;
}

// Uncomment for lab exercise B
private void processSubtraction() {
    if(currentNumber==null) {
        lastNumber= 0L;
    }else{
        lastNumber=currentNumber;
    }
}
```

The screenshot shows a code editor window titled 'CalcImpl.java'. The code contains two private methods: 'processAddition()' and 'processSubtraction()'. In 'processAddition()', the 'if' condition and the 'else' block are highlighted in green, while the 'if' block is highlighted in red. In 'processSubtraction()', the 'if' condition and the 'else' block are highlighted in green, while the 'if' block is highlighted in red. A comment '// Uncomment for lab exercise B' is present above the 'processSubtraction()' method.

You will notice that's the code editor window opens displaying the selected method with highlighting for each line. Green indicates that the line **was** executed, Red indicates that the line **was not**.

#### 10. ADD A TEST TO IMPROVE THE COVERAGE OF THE “PROCESSADDITION” METHOD

- Review the method to determine the path that is not being executed, and the conditions that must be satisfied to take that path
- Add a new unit test that will execute the path
- Note that in our example this is a private method so it cannot be called directly from your test. You will need to determine a suitable set of inputs to the public interface (i.e. “keyPressed”) that will execute the desired code



## 11. RUNNING FINDBUGS

Find Bugs can be run against as broad or narrow a section of code as you desire (e.g. execute against a package and only that package and its classes will be analysed), but for the purposes of our workshop lets look at it all.

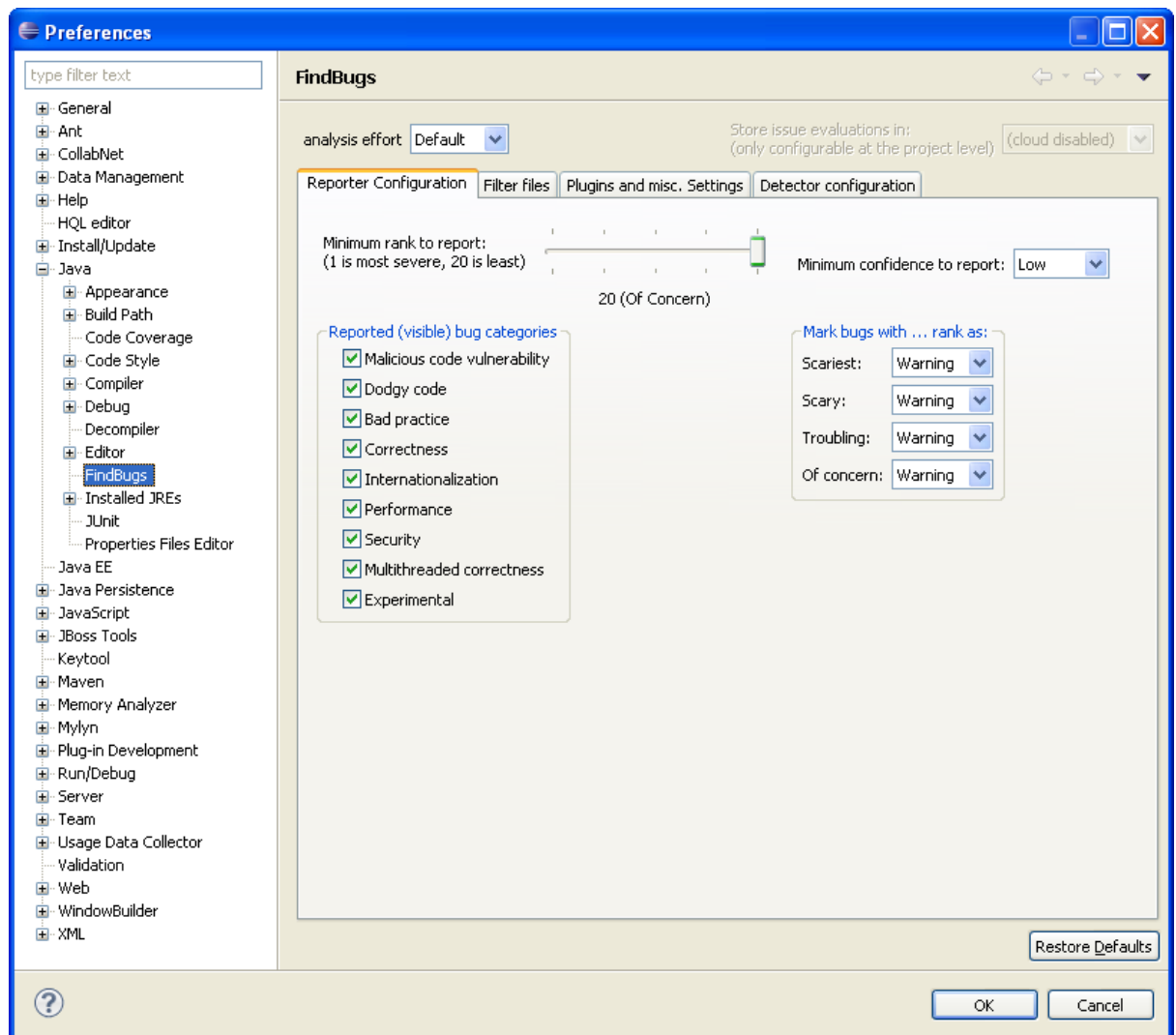
First, open the “Bug Explorer”:

Window -> Show View -> Other -> FindBugs -> Bug Explorer

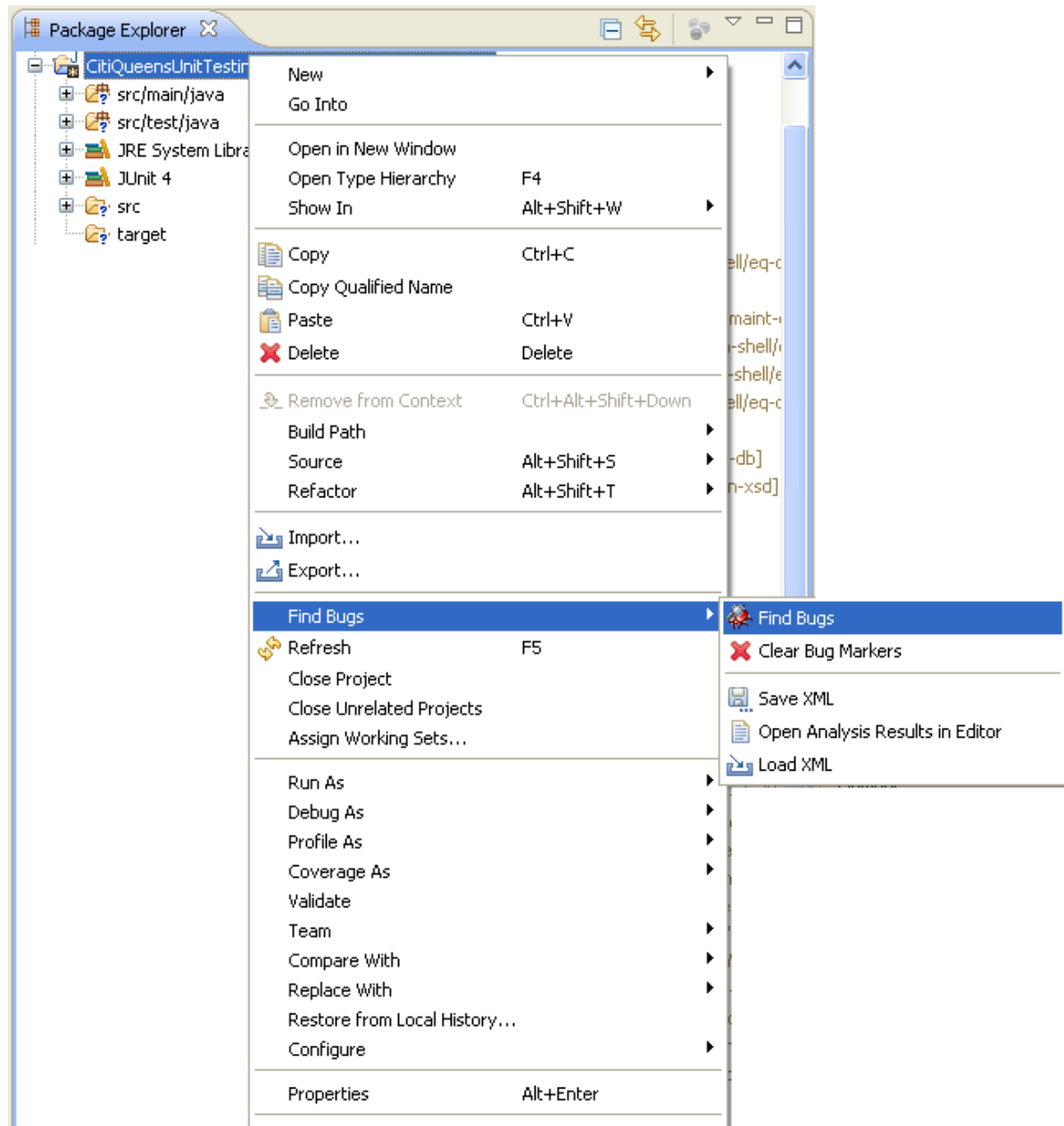
Secondly, configure FindBugs to show us everything :

Window -> Preferences -> Java -> FindBugs

Setup the configuration as below



Now execute FindBugs against the entire Project:



Once complete the “Bug Explorer” window will be populated with any issues discovered. These same issues will display in the “Problems” window as well.

If we’ve been doing things correctly so far there shouldn’t have been any issues list so let’s create one and see how it’s reported.

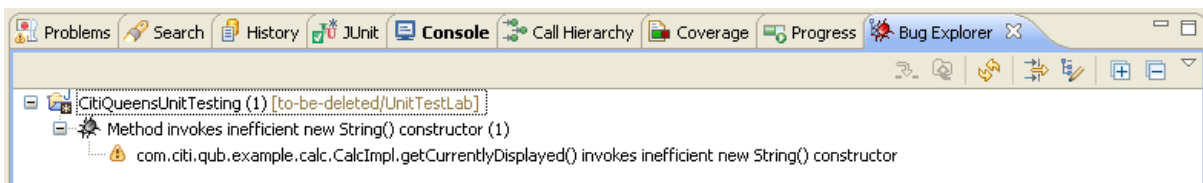
---

## 11.1 ADDING A “BUG”

- Update the **CalcImpl.getCurrentlyDisplayed** method to read as below (highlighted portion is the change)

```
@Override
public String getCurrentlyDisplayed() {
    return currentNumber==null ? new String() : currentNumber.toString();
}
```

- Rerun FindBugs on the project
- A ‘bug’ symbol appears next to the problematic line
- Note that the detected issue is displayed in the “Problems” and “Bug Explorer” windows
- In the “Bug Explorer” window expand the tree until the warning is visible



- By double-clicking on the warning you can jump to the problem code in an editor window
- Alternatively you can jump to a fuller description of the bug by right-clicking the warning and choosing the ‘Show Bug Info’ option. This will open the “Bug Info” panel which provides a description of the bug, why it is of concern and a suggested course of action.

---

## 11.2 REVIEW THE STANDARD RULES

- Open the FindBugs configuration  
Window -> Preferences -> Java -> FindBugs
- Switch to the “Detector Configuration” Tab
- Review a selection of rules in the list. Note that when you choose an item in the list a fuller description is displayed below.

# Version Control

---

## 12. INITIAL READING

Please study this material and the section in the Citi handbook relating to Subclipse, prior to commencing on the exercises.

### 1. “Version Control – Fundamental Concepts”

<http://svnbook.red-bean.com/en/1.7/svn.basic.html>

This section of the Subversion book covers the fundamentals of Version Control and Subversion’s approach to it.

---

## 13. ABOUT THE CODE

This exercise builds upon the code created during the “Unit Testing” & “Code Quality” workshops. We will demonstrate the use of version control and Subclipse using this code base

---

## 14. SETTING UP THE ENVIRONMENT FOR THE EXERCISE

For this exercise we will be running plugins within the eclipse IDE.

Pre-requisites

- Eclipse IDE [ version 3.6 (Helios) or higher ]
- J2SE JDK 1.6 or higher
- Subclipse (Eclipse Plugin)
- Subversion Repository and user details

It is assumed that the user has already completed the “Unit Testing” workshop and has the resulting workspace from that workshop available to build upon.

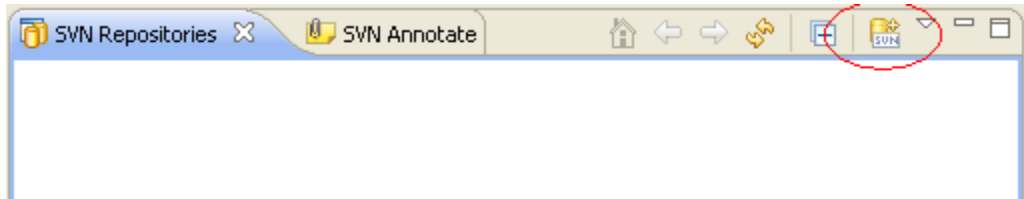
*It is also recommended that this workshop is completed in pairs, with each partner at a separate machine.*

- *Person 1 should have the workspace from the “Unit Testing” workshop open*
- *Person 2 should have a new/clean workspace open*

**Person 1 AND Person 2:**

Our first step is pointing eclipse towards our repository, the central store where the code will be managed and shared.

- Open the “SVN Repositories” perspective  
Window -> Open Perspective -> Other -> SVN Repository Exploring
- Add the repository using the provided details



- Configure the Repository URL as prompted
- Configure the user/password as prompted

The first stage in Version Control is sharing the project, i.e. adding it to a central repository and having all developers connected to that location

**Person 1 : Share the Code**

- Open the “Java” perspective : Window -> Open Perspective -> Java
- Right-click on the Project -> Team -> Share Project
- Choose “SVN” Type
- Choose to use the existing Repository Location
- Choose to use a specified folder name
  - Use “trunk/UnitTestingLab”
  - *note the full destination URL is displayed below*
- Choose “Next”
- Enter a comment to reflect what you are doing, then Choose “Finish”
- Wait for the operation to complete
- Choose “No” when promoted to switch perspective

*At this stage Person 1 has connected their Project to the Repository location, but has not actually added the code to the repository, to do so*

- Open the “Team Synchronizing” perspective :  
Window -> Open Perspective -> Other -> Team Synchronizing
- Right-click the Project -> Commit
- Enter a commit comment, then Choose “OK”
- Wait for the operation to complete
- Open the “Java” perspective : Window -> Open Perspective -> Java
- Note that the Project now displays with the location to its right in the Package Explorer

**Person 2 : “Checkout” (Retrieve) the Code**

- Open the “SVN Repositories” perspective  
Window -> Open Perspective -> Other -> SVN Repository Exploring
- Browse to the folder name specified by Person 1 previously
- Right-click the Folder -> Checkout
- Accept the default options, choose “Finish”
- Wait for the operation to complete
- Note that the Project now displays with the location to its right in the Package Explorer

You now have a single project shared from the repository that both Persons are connected to.

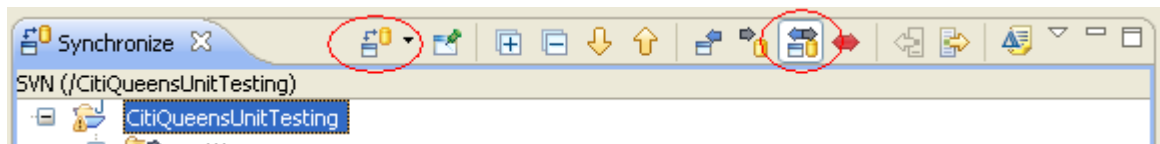
As developers work on fulfilling requirements/making changes they will periodically commit changes to the repository to

- Serve as a backup location
- Make those changes available to others.

Note that it is critical when committing changes that you leave the code in a working state!

### Person 2 : Making & Committing a change

- Open the “Java” perspective : Window -> Open Perspective -> Java
- Make a change to the code (e.g. add some additional comments)
- Open the “Team Synchronizing” perspective :  
Window -> Open Perspective -> Other -> Team Synchronizing
- Synchronize with the repository, this will provide a list of incoming and outgoing changes. (If prompted, choose the “SVN” type and all projects in the workspace)



- Wait for the operation to complete, you will now see the outgoing change. You can double click on the changed files to get a side-by-side comparison between your copy and that in the repository.
- Right-click the changed file -> Commit
- Enter a commit comment, Choose “OK”
- Wait for the operation to complete

### Person 1 : Receiving the change

- Open the “Team Synchronizing” perspective :  
Window -> Open Perspective -> Other -> Team Synchronizing
- Synchronize with the repository, this will provide a list of incoming and outgoing changes. (If prompted, choose the “SVN” type and all projects in the workspace)
- Wait for the operation to complete, you will now see the incoming change. You can double click on the change to get a side-by-side comparison between your copy and that in the repository.
- Right-click the changed file -> Update
- Wait for the operation to complete

Both users are now “in-sync” with the repository.

## 18. CONFLICTS

It is common place for multiple developers to make changes to the same file around the same time, in this case we operate on a “first-come, first served” basis – the first person to commit is allowed, the second person has to resolve the conflict locally before committing.

### Person 1 : Making & Committing a change

- Open the “Java” perspective : Window -> Open Perspective -> Java
- Make a change to the code (e.g. add some additional comments)
- Open the “Team Synchronizing” perspective :  
Window -> Open Perspective -> Other -> Team Synchronizing
- Synchronize with the repository as before
- Right-click the changed file -> Commit
- Enter a commit comment, Choose “OK”
- Wait for the operation to complete

### Person 2 : Making a change, Resolving the conflict & Committing

- Open the “Java” perspective : Window -> Open Perspective -> Java
- Make a change to the code (e.g. add some additional comments) in **the same file** that Person 1 made their updates.
- Open the “Team Synchronizing” perspective :  
Window -> Open Perspective -> Other -> Team Synchronizing
- Synchronize with the repository as before
- Notice the changed file has a red icon indicating the conflict.
- Double click the file for a side-by-side comparison. It is now **your responsibility** to update the local copy of the file ensuring that it includes both sets of changes. (Note: you can use the “change copy” links in the middle for simple changes).
- Once complete you need to mark your copy as “merged” before you can commit your changes, to do so Right-click the file -> Mark as Merged.
- Right-click the changed file -> Commit
- Enter a commit comment, Choose “OK”

### Person 1 : Re-syncing

- Open the “Team Synchronizing” perspective :  
Window -> Open Perspective -> Other -> Team Synchronizing
- Synchronize with the repository as before
- Right-click the changed file -> Update
- Wait for the operation to complete

Both users are now “in-sync” with the repository.



Both concepts involve “copying” the code from one area of the repository to another. The code can then be checked out and edited as normal. Note that although tags are generally perceived to be read-only, this is not enforced.

- *Tagging* is generally used to *checkpoint* the code-base, i.e. record the code-base at a given point in case recovery is needed. We are keeping a Record of the code as it was built, etc.
- *Branching* is generally used to *fork* the code-base, i.e. creating parallel streams for development with the same origin point.

#### Person 1 OR Person 2

- Open the “SVN Repositories” perspective  
Window -> Open Perspective -> Other -> SVN Repository Exploring
- Navigate through the repository to  
trunk/UnitTestLab
- Right-click the folder -> Branch/Tag
- Choose the “Copy To” URL, i.e. the destination folder in the repository  
**<repository root>/branches/testBranch1/UnitTestLab**
- Ensure that “create any intermediate folders ...” is checked, Choose “Next”
- Ensure the “HEAD” revision is chosen, Choose “Next”
- Enter a comment, Choose “Finish”

Your new branch is now available to be checked out and worked on

---

### 19.1 MAKE CHANGES TO THE ORIGINAL

#### Person 1 AND Person 2

- Make a number of changes to the current project (from trunk) **and commit them**
- Ensure that at least one change involves the creation of a new directory/package containing some files

---

### 19.2 MAKE CHANGES TO THE BRANCH

#### Person 1 AND Person 2

- Delete the current project from your workspace
- Checkout the “branched” copy of the project
- Make a number of changes to the project (from the branch) **and commit them**
- Ensure that at least one change involves the creation of a new directory/package (containing some files) that has the same name and location as the

directory/package you created in Section 19.1. (The intention here is to provoke a ‘tree conflict’ – something you would normally seek to avoid in a live project!)

---

## 20. MERGING

*Merging* is the process of re-integrating into the original codebase the changes that have been made on a branch. This process is generally followed when a release has been confirmed for deployment and you want to integrate new features back into the trunk for release.


Merging a branch with the trunk after changes have been made to both the branch and the trunk is likely to be problematic – and the scenario discussed here is deliberately designed to provoke file conflicts and tree conflicts. Ideally the trunk should be kept stable and working while branches are changing. This section should, therefore, be regarded as an exploratory exercise and a guide to how file conflicts and tree conflicts might be handled – rather than a simple stepwise solution. Each combination of conflicts is likely to differ from the next, and may cause the development environment to offer you new information or options to help you resolve the problem: examine the resources that Eclipse and Subclipse make available to you and adapt the steps suggested to your own amended code.

### Person 1 OR Person 2

Stage 1 involves retrieving a “clean” checkout of the **destination**

- Disable Auto-Building : Project -> Build Automatically
- Delete the current project (from the branch) from your workspace (making sure to select “Delete project contents from disk”).
- Checkout the original copy of the project from “/trunk/UnitTestLab”

Stage 2 involves performing the merge

- Open the “Java” perspective : Window -> Open Perspective -> Java
- Right-click the project and select Team -> Merge
- Ensure “Merge a range of revisions” is selected
- Ensure “Perform pre-merge best practice checks” is checked. This will give you a clean checkout for the merge.
- Click “Next”.
- Choose the “Merge From” source as “/branches/testBranch1/UnitTestLab”.
- Choose “Finish” and let the merge operation complete.
- A Merge Results Summary window will appear, giving an overview of changes that the system has made to your working copy of the project, and indicating any conflicts that have occurred. Dismiss the summary window by clicking OK. The “Merge Results” window should now be open (if the Merge Results window has not opened by itself, look for and click the “Merge Results” icon: ). The Merge Results

window is similar to the “Team Synchronizing” view in that it shows new project content that has been imported and any conflicts that have occurred.

- Items marked with a small blue arrow are changes (new files or amendments to existing files) that have been automatically performed.
- Orange/Green Double Arrows are tree conflicts that need to be examined more closely (see below).
- Red Double Arrows are merge conflicts, which also require special action.
- For each tree conflict (Orange/Green double arrow)
  - Right-click the conflict and choose Team’ -> Show Tree Conflicts.  
(Occasionally, if a Tree Conflict occurs at file, rather than folder level – because a file has been renamed, say, but an existing folder was used – there will be no ‘Show Tree Conflicts’ option: in such a case, renaming the file in the Package Explorer will generally resolve the conflict and allow you to re-commit).
  - Determine if items in conflict should be retained/removed. This may mean that you will have to examine changes that you previously made to your local files and folders, and changes that are being imported as a result of the merge. Right-click the folder in the SVN Tree Conflicts Window, and select Resolve...
  - Select ‘Compare’ from the Resolution steps and click Finish. The contents of the folder in conflict may be reviewed in the Structure Compare editor window.
  - In the Structure Compare editor window, you will see files (marked with a black box) that have been changed, files (marked +) that are contained in the trunk but not in the branch, and files (marked -) that are contained in the branch but not in the trunk. In the Java perspective’s Package Explorer you may now need manually to add – by copying from the branch’s SVN repository – or delete files in the trunk, so that it matches the structure of the branch.
  - If you are satisfied that a conflict has been resolved, right-click the item in question (in the Package Explorer) and select ‘Team’ -> ‘Mark Resolved’.
- For each conflict (Red double arrow)
  - Double click the conflict to open the editor.
  - Resolve all conflicts in the file (you may use the editor to move content from the incoming file to the local file and perform additional edits).
  - Save the file.
  - Close the file editor. On closing you will be prompted to indicate whether you have resolved all conflicts in that file (choose ‘Yes, ...’ if all are resolved)

Stage 3 involves committing the merge changes to the destination

- Open the “Team Synchronizing” perspective :  
Window -> Open Perspective -> Other -> Team Synchronizing
- Synchronize with the repository as before.
- Right-click the changed file (or the project as a whole) -> Commit.
- Enter a commit comment, Choose “OK”
- Wait for the operation to complete.

The changes that you have chosen to incorporate from the branch (“/branches/testBranch1/UnitTestLab”) are now integrated into the original (“/trunk/UnitTestLab”).