# How version control software can be used to aid software maintainability in a working environment comprising multiple teams across multiple locations?

Danish Hussain - P14181072

March 2016

**Supervisor 1:** Dr Richard Smith

# 1 Report

One of the key elements of software configuration management is version control which is also known as revision or source control. Version control is a system that allows individuals and organisations to manage and change multiple revisions of the same files without losing the original files for example source codes and documents, in other words it can be seen as a process that differentiates the drafts from the final piece. This report will highlight how version control supports software maintainability in a working environment.

Software maintainability is simple terms is a measure of the difficulty surrounding the alterations made to software which can relate to adding new features, increasing performance etc., from a strict engineering perspective it can mean bug fixing (Gilb, 2008). There are two main types of version control software's that organisations have to consider when selecting the most applicable to them, the first one being centralised which includes software such as CVS and Subversion(SVN) and the second one being distributed which includes software such as Git. These are the most common types of software which can be supported through recent statistics which portray that the highest popularity rate rests with Subversion at a staggering 48% followed by Git with 37% and CVS being less common at just 9% (Duck, 2014).

Version control aids software maintainability in a working environment through keeping the code and all related documentation up to date and synchronised to match the user specification, through this multiple teams can contribute across various locations and merge their work without losing the original versions. Version control also makes it easier to track the changes made for each individual file as the teams work under the same rules which makes it easy to monitor updates. Another benefit of using version control to aid maintainability is that it becomes easier to list the changes that have been made which can then be used to portray the multiple versions of the software overtime, furthermore if an error was made during the change stage it is easy to roll back to an older version and fix any issues and finally all the changes that are made can be merged into a common, updated version. Branches in version control allow multiple copies

of the code within a single repository this is advantageous for companies dealing with a variety of customers as it allows flexibility to maintain and adjust the software accordingly to match each customer's requirements without having to write the whole thing from scratch, another advantage of this is that locating and fixing bugs becomes easier as a number of versions of the software can be ran and compared to determine where the problems occur.

It is apparent that without version control software maintenance across teams would become difficult because multiple people would be working on the same set of files which is very problematic, error-prone and time consuming and could eventually lead to the overwriting of important work or software that was developed by a team member as there is no control of who produced what. Furthermore, people would use a new folder for each updated version which has various drawbacks such as its hard to identify how each file differs and what changes have been made. Version control has countered this problem by allowing developers to merge their changes with others and multiple developers can work on the same file simultaneously which shows how each file differs and also illustrates any conflicts that may commence.

In conclusion version control is an essential component of software maintenance primarily because it allows the collaboration of teams across various locations to work on projects, which ensures efficiency and reliability when making changes to software. Another key aspect of the necessity of version control is that it saves a lot of time which is an important factor when maintaining time critical software. As software development progresses the demand for version control will possibly increase because without version control software would be hard to maintain which would lead to various failures during the software development cycle.

# 2 Test Cases

```java
package counter;

import org.junit.Test;

/**
 * Tests that have been conducted for the Counter class
 * Three tests should fail and three should pass
 * @author Danish Hussain
 */

public class CounterTest {

//Setting the timeout constant to 2000 milliseconds
public static final int TIMEOUT = 2000;

 /**
  * Lab 7 Item 5:
  * Testing counter to make it fail deliberately (Should Fail)
  */

//This method will fail if it doesn't finish running within 2000 milliseconds
@Test(timeout = TIMEOUT)
public void test_DeliberatefailedTest()
{
Counter c1 = new Counter();        //new instance of the counter object created
      try{                              //try is where the exceptions occur
          c1.increment();               //Increment counter(add one)
      } catch(InvariantException e){    //Catch used to handle the exceptions

          e.printStackTrace();          //Prints the stack trace of the exception
      }
      //AssertEquals - Compares two objects for equality
      assertEquals("Deliberate Fail Test Returned: ", 2, c1.getValue());
      assertEquals("Deliberate Fail Test Returned: ", 1, c1.getLowerLimit());
}
```

```java
 /**
  * Lab 7 Item 6:
  * Testing The Reset Method (Should Pass)
  */

//This method will fail if it doesn't finish running within 2000 milliseconds
@Test(timeout = TIMEOUT)
public void test_ResetCounter_ToValueSpecified()
{
  Counter c1 = new Counter(10);   //new instance of counter object (value of 10)
       try {                      //try is where the exceptions occur
        c1.set(20);              //Setting the counter value to 20
        }
        catch (InvariantException e) {  //Catch used to handle the exceptions
            e.printStackTrace();        //Prints the stack trace of the exception
        }
        c1.reset();                     //counter is reset(should set value to 10)

//AssertEquals - Compares two objects for equality
assertEquals("Counter value after resetting = ", 10, c1.getValue());
assertEquals("Counter lower limit after resetting = ", 10, c1.getLowerLimit());

}


 /**
  * Lab 8 Item 7
  * Test case for NegativeIncreaseException (Should Fail – No Throw Exception)
  * @throws counter.NegativeIncreaseException
  * Test case for NegativeIncreaseExceptionB (Should also Fail)
  */

//This method will fail if it doesn't finish running within 2000 milliseconds
@Test(expected = NegativeIncreaseException.class, timeout = TIMEOUT)
public void test_Negative_IncreaseExceptionTest() throws NegativeIncreaseException
{
  Counter c1 = new Counter();//Creating a new instance of the counter object
  c1.increase(-2);           //Setting the counter to increase by -2
}

//This method will fail if it doesn't finish running within 2000 milliseconds
@Test(timeout = TIMEOUT)
public void test_Negative_IncreaseExceptionTestB()
{
   Counter c1 = new Counter();//Creating a new instance of the counter object
        try{                  //try is where the exceptions occur
        c1.increase(-2);       //Setting the counter to increase by -2
        fail("Exception should have been thrown");  //Making sure exception thrown
        }
         catch (NegativeIncreaseException e) {
       //Message to print if exception found
      System.err.println("Caught NegativeIncreaseException: " + e.getMessage());
      //AssertEquals - Compares two objects for equality
      assertEquals("value after negative number increase = ", 0, c1.getValue());
      assertEquals("lower limit value after increase = ", 0, c1.getLowerLimit());
   }
} **(NOTE ON LAST PAGE)
```

```java
/**
 * Lab 8 Item 8
 * Test case for InvariantExceptionA Increment(Pass but throw InvariantException)
 * @throws counter.InvariantException
 * Test case for InvariantExceptionB Increment (Should Pass)
 */

//Method should throw an InvariantException to pass the test but will fail if it
doesn't finish running within 2000 milliseconds
@Test(expected = InvariantException.class, timeout = TIMEOUT)
public void test_Increment_TestingIncrementA() throws InvariantException
{
Counter c1 = new Counter(Integer.MAX_VALUE); //new instance of the counter(maxval)
c1.increment();                              //Incrementing the counter (add one)
}

//This method will fail if it doesn't finish running within 2000 milliseconds
@Test(timeout = TIMEOUT)
public void test_Increment_TestingIncrementB() {
  Counter c1 = new Counter();    //Creating a new instance of the counter object
        try{
        c1.set(Integer.MAX_VALUE); //Setting the counter to have a MAX VALUE
        c1.increment();            //Incrementing the counter (adding one)
       fail("Exception should have been thrown"); //Making sure exception thrown
        }
        catch (InvariantException e) {   //Catch used to handle the exceptions
//Message to print if exception found
System.err.println("Caught InvariantException: " + e.getMessage());
//AssertEquals - Compares two objects for equality
assertEquals("Counter value after increment", Integer.MAX_VALUE, c1.getValue());
assertEquals("Counter lower limit after incrementing ", 0, c1.getLowerLimit());
       }

}

}
```

**Note for 8.7 – If you add the following code to the Counter.java class Increase method above the for loop then this test would also throw an exception and pass:

```
if (n < 0) {

Throw new NegativeIncreaseException();

}
```