Question 1

***Marking scheme*:**
4 points for a correct explanation why unit testing of Diff is hard
2 points for suitable modifications of the code
4 points for a good explanation of the implications of your modifications

Why is the class Diff hard to unit test

(a) The class `Diff` internally instantiates its fields `in1` and `in2` (of type `java.io.BufferedReader`). A successful instantiation of Diff requires the existence of the two files to be compared on the system. Each test case execution would then require creation of the corresponding files. It would be easier when performing unit tests on this class to use `java.io.Reader` objects instead of files. One may then use `java.io.StringReader`. Strings are easier to create and manipulate. This can be done directly in the testing code.

(b) Considering that one may test methods individually, some additional methods implementation might be required on auxiliary classes. For instance one could implement a comparison method for the class `DiffConflictLine` (which does not currently exist) when testing the method `getDiffLines` because it returns a list of `DiffConflictLine` objects.

(c) When trying to reach some structural coverage goals, the complicated control structure of the methods `bestMatch` and `getDiffLines` may be difficult to deal with.

The class `MatchResult` does not make unit testing of `Diff` difficult because it has a very simple structure and fully accessible class members.

Modify the class to make it easier to unit test

(a) One solution may be to create a constructor for the class `Diff` accepting two parameters of type `java.io.Reader` instead of `java.io.File`. This allows one to use any class that implements the `java.io.Reader` interface with the class `Diff`.

```
public Diff(Reader reader1, Reader reader2) throws Exception {
      if (null == reader1 || null == reader2) {
         throw new Exception("null reader object passed");
      }
      in1 = new BufferedReader( reader1 );
      in2 = new BufferedReader( reader2 );

      lineCounter.put(in1, new Integer(1));
      lineCounter.put(in2, new Integer(1));
}
```

(b) Implement other required methods in subclasses of corresponding classes (e.g equals in `DiffConflictLine`), depending on the purpose of the test.

(c) It is not a good idea to change the internal structure of the algorithms implemented by the developers just for testing purposes since we want to test what it is actually happening in the production code. But suggestions to the developers on how to change the code can be made.

Recall that sometimes (e.g: in Test-Driven Development) test cases are written before creating the actual code. And this may be done by developers different from the one writing the production code.

Examine the rest of the code. Discuss the implications of your changes

The classes `JAGTemplateEngine` and `VelocityTemplateEngine` from the project use the class `Diff`. For unit testing purposes, as we should just add new code through subclasses, no modifications should be required into the production code.

Question 2

***Marking scheme*:**
2 points for the correct handling / explanation of each predicate (18 points in total)
2 points for explaining why you think `CACC` is achieved by your tests

Prepare a suite of Junit tests which ensure Correlated Active Clause Coverage (`CACC`) on your modified version of getDiffLines

The following predicates are present in the method `Diff.getDiffLines`:

Formula	Line	Code
a	101	`line1.isEof()`
b	105	`line2.isEof()`
$!(a \;\&\&\; b)$	100	`!(line1.isEof() && line2.isEof())`
c	110	`line1.lineEquals(line2)`
d	116	`containsLine(file2, line1.getLine(), line2.getLineNumber())`
e	120	`betterMatch != null`
f	122	`line1.getLineNumber() < betterMatch.endFile1Sequence`
g	144	`in1 != null`
h	145	`in2 != null`

To achieve `CACC`, two test requirements are needed for each predicate p ranging from c to h: p should evaluate to true and p should evaluate to false.

The predicate $!(a \;\&\&\; b)$ can be written as $\neg(a \wedge b)$ and is equivalent to:

$$P = \neg a \vee \neg b$$

Here is the truth table for $\neg a \vee \neg b$:

	a	b	$\neg\ a \vee \neg b$
1	F	T	T
2	F	F	T
3	T	T	F
4	T	F	T

When **a is major clause of P**:

$$\begin{aligned} P_a &= P_{a=true} \oplus P_{a=false} \\ &= (\neg true \vee \neg b) \oplus (\neg false \vee \neg b) \\ &= (false \vee \neg b) \oplus (true \vee \neg b) \\ &= \neg b \oplus true \end{aligned}$$

b has to be true for a to determine P. This corresponds to **rows 1 and 3** of the truth table.

When **b is major clause of P**:

$$\begin{aligned} P_b &= P_{b=true} \oplus P_{b=false} \\ &= (\neg a \vee \neg true) \oplus (\neg a \vee \neg false) \\ &= (\neg a \vee false) \oplus (\neg a \vee true) \\ &= \neg a \oplus true \end{aligned}$$

a has to be true for b to determine P. This corresponds to **rows 3 and 4** of the truth table.

To achieve `CACC` for the predicate !(a && b), we then have the following tests requirements:

$$\langle a : \mathsf{false}, b : \mathsf{true}\rangle \qquad \langle a : \mathsf{true}, b : \mathsf{true}\rangle \qquad \langle a : \mathsf{true}, b : \mathsf{false}\rangle$$

The test requirement $\langle a : \mathsf{true}, b : \mathsf{true}\rangle$ is required because `CACC` requires the values chosen for each clause must cause the predicate to be true for one of its value and false for the other one. Thus leaving this test requirement will make that no value for a will cause !(a && b) to be false.

The test requirements for the predicate !(a && b) subsumes the tests requirements for the predicates a and b.

Question 3

```
import java.beans.PropertyChangeEvent;
import java.beans.PropertyChangeListener;
import java.lang.reflect.Field;

import com.eteks.sweethome3d.model.Wall;

import org.junit.After;
import static org.junit.Assert.*;
import junit.framework.TestCase;

import org.junit.Before;
import org.junit.Test;
import static org.easymock.EasyMock.createMock;
import static org.easymock.EasyMock.replay;
import static org.easymock.EasyMock.verify;
import static org.easymock.EasyMock.notNull;
import static org.easymock.EasyMock.capture;
import org.easymock.Capture;
import org.easymock.EasyMock;

/*
 * Solutions prepared by Bob Zhang.
 *
 * Each m3 and m4 mutation suggestion is 2 marks
 * m7 suggestion is 1 mark
 * Each m1 - m7 test case is 5 marks
 */

public class WallTestSolution extends TestCase {
```

```
Wall w, w1;
PropertyChangeListener pcl;

@Before
public void setUp() {
  w = new Wall(0,0,0,0,0);
  w1 = new Wall(1,1,1,1,1);
  pcl = createMock(PropertyChangeListener.class);
  w.addPropertyChangeListener(pcl);
}

@Test
public void testM1andM6() {
  pcl.propertyChange((PropertyChangeEvent)notNull());
  replay(pcl);
  w.setWallAtEnd(w1);
  verify(pcl);
}

@Test
public void testM2() {
  Wall w2 = new Wall(2,2,2,2,2);
  w.setWallAtEnd(w2);
  Capture<PropertyChangeEvent> capturedEvent =
    new Capture<PropertyChangeEvent>();
  pcl.propertyChange(EasyMock.capture(capturedEvent));
  EasyMock.expectLastCall();
  EasyMock.replay();
  w.setWallAtEnd(w1);
  EasyMock.verify();
}

@Test
//m3: RHS becomes null
public void testM3() {
  w.setWallAtEnd(w1);
  assertTrue(w.getWallAtEnd() != null && w.getWallAtEnd().equals(w1));
}

@Test
//m4: RHS becomes oldWallAtEnd
public void testM4() {
  w.setWallAtEnd(w1);
```

```
    assertTrue(w.getWallAtEnd() != null && w.getWallAtEnd().equals(w1));
  }

  @Test
  public void testM5() {
    Class<?> wallClass = null;
    try {
      wallClass = Class.forName("com.eteks.sweethome3d.model.Wall");
    } catch (ClassNotFoundException e) {
      e.printStackTrace();
    }
    Field pointsCacheField = null;
    try {
      pointsCacheField = wallClass.getDeclaredField("pointsCache");
    } catch (Exception e) {
      e.printStackTrace();
    }
    pointsCacheField.setAccessible(true);
    try {
      pointsCacheField.set(w1, new float[1][1]);
    } catch (Exception e) {
      e.printStackTrace();
    }
    w.setWallAtEnd(w1);
    try {
      assertTrue(pointsCacheField.get(w1) == null);
    } catch (Exception e) {
      e.printStackTrace();
    }
  }

  @Test
  public void testM7() {
    Capture<PropertyChangeEvent> pce = new Capture<PropertyChangeEvent>();
    pcl.propertyChange(capture(pce));
    replay(pcl);
    w.setWallAtEnd(w1);
    verify(pcl);
    assertEquals(w1, pce.getValue().getNewValue());
    assertEquals(null, pce.getValue().getOldValue());
  }
}
```