Software Testing, Quality Assurance & Maintenance (ECE453/CS447/SE465): Final

April 21, 2017

This open-book midterm has 7 questions, each worth 20 points. Answer the questions in your answer book. You may consult any printed material (books, notes, etc).

Question 1. Short Answer

Which tool could you use to find the following problems? **Name the tool** and **describe** which class of techniques it belongs to. Also **name a limitation of/challenge for** that tool/technique in that situation. **Write** one or two sentences for each situation.

[Example: "race conditions": Helgrind. This *dynamic analysis tool* looks for a write to memory that occurs concurrently with a read of the same location with no common locks between the read or the write. Dynamic analysis only detects problems on code paths that get executed.]

a. program does not conform to project style guides forbidding use of global variables.

b. program is accessing memory outside array bounds.

c. variable names don't describe their contents.

d. program is difficult to use (not user-friendly).

e. developers commit source to shared repository that fails to compile.

Question 2: Code Review

Consider the Java code in `SourceFileScope`, attached. **Review** 3 aspects of the code. **Explain** each aspect in a couple of sentences, supporting it with examples from the code that you're reviewing.

```java
/* Question 2 task: Review 3 aspects of the code. Explain your answer
 * in a couple of sentences, supporting it with examples from the code. */

/**
 * BSD-style license; for more info see http://pmd.sourceforge.net/license.html
 */
package net.sourceforge.pmd.lang.java.symboltable;


import java.util.Collections;
import java.util.HashMap;
import java.util.HashSet;
import java.util.List;
import java.util.Map;
import java.util.Set;

import net.sourceforge.pmd.lang.ast.Node;
import net.sourceforge.pmd.lang.java.ast.ASTImportDeclaration;
import net.sourceforge.pmd.lang.symboltable.NameDeclaration;
import net.sourceforge.pmd.lang.symboltable.NameOccurrence;
import net.sourceforge.pmd.lang.symboltable.Scope;

/**
 * This scope is the outer most scope of a Java file.
 * A Source File can contain one ore more classes.
 */
public class SourceFileScope extends AbstractJavaScope {

    private String packageImage;
    private TypeSet types;

    public SourceFileScope() {
        this("");
    }

    public SourceFileScope(String packageImage) {
        this.packageImage = packageImage;
    }

    /**
     * Configures the type resolution for the symbol table.
     * @param classLoader the class loader to use to find additional classes
     * @param imports the import declarations
     */
    public void configureImports(ClassLoader classLoader, List<ASTImportDeclaration> imports) {
        this.types = new TypeSet(classLoader);
        types.setASTCompilationUnitPackage(packageImage);
        for (ASTImportDeclaration i : imports) {
            if (i.isImportOnDemand()) {
                types.addImport(i.getImportedName() + ".*");
            } else {
                types.addImport(i.getImportedName());
            }
        }
    }

    public Set<String> getExplicitImports() {
        return types != null ? types.getExplicitImports() : Collections.<String>emptySet();
    }

    /**
     * Whether an auxclasspath has been configured or not.
     * This can be used to enable/disable more detailed symbol table analysis and type resolution
     * can be used - or to fall back to more simple implementation.
     * @return <code>true</code> if the auxclasspath is configured and types can be resolved reliably.
     * @see #resolveType(String)
     */
    public boolean hasAuxclasspath() {
        return types.hasAuxclasspath();
    }
```



```java
    }

    /**
     * Tries to resolve a class by name.
     * @param name the name of the class
     * @return the class or <code>null</code> if no class could be found
     */
    public Class<?> resolveType(String name) {
        try {
            return types.findClass(name);
        } catch (ClassNotFoundException e) {
            return null;
        }
    }

    public String getPackageName() {
        return packageImage;
    }

    /**
     * {@inheritDoc}
     * @throws IllegalArgumentException if declaration is not a {@link ClassNameDeclaration}
     */
    @Override
    public void addDeclaration(NameDeclaration declaration) {
        if (!(declaration instanceof ClassNameDeclaration)) {
            throw new IllegalArgumentException("A SourceFileScope can only contain classes.");
        }
        super.addDeclaration(declaration);
    }

    /**
     * Convenience method that casts the declarations to {@link ClassNameDeclaration}s.
     * @see #getDeclarations()
     * @return all class name declarations
     */
    public Map<ClassNameDeclaration, List<NameOccurrence>> getClassDeclarations() {
        return getDeclarations(ClassNameDeclaration.class);
    }

    public String toString() {
        return "SourceFileScope: " + glomNames(getClassDeclarations().keySet());
    }

    public ClassNameDeclaration findClassNameDeclaration(String name) {
        ImageFinderFunction finder = new ImageFinderFunction(name);
        Applier.apply(finder, getClassDeclarations().keySet().iterator());
        return (ClassNameDeclaration)finder.getDecl();
    }

    protected Set<NameDeclaration> findVariableHere(JavaNameOccurrence occ) {
        Set<NameDeclaration> result = new HashSet<>();
        ImageFinderFunction finder = new ImageFinderFunction(occ.getImage());
        Applier.apply(finder, getDeclarations().keySet().iterator());
        if (finder.getDecl() != null) {
            result.add(finder.getDecl());
        }
        return result;
    }

    // [2 functions omitted]
}
```

Question 3: Mock Objects

You are given the following code that represents a one dimensional world with transportation, with distances measured as integer kilometers and prices measured as integers.

```
1  interface Transport {
2      Vehicle hailVehicle();
3      int getPricePerKm();
4  }
5
6  interface Vehicle {
7      Driver getDriver();
8      void waitUntilArrivedAt(int location);
9  }
10
11 interface Driver {
12     void setDestination(int location);
13 }
14
15 interface Wallet {
16     Wallet(int initialAmount);
17
18     void takeOut( int amount ) throws YouAreBrokeException;
19     int getAmount();
20 }
21
22 class Person {
23     Person( int home, int currentLocation, Wallet wallet ) { /* init fields */ }
24
25     Wallet wallet; Wallet getWallet() { return wallet; }
26     int home; int getHome() { return home; }
27     int location; int getLocation() { return location; }
28
29     void goHomeUsing(Transport t) {
30         Vehicle v = t.hailVehicle();
31         v.getDriver().setDestination( getLocation() );
32         v.waitUntilArrivedAt( getHome() );
33
34         int payment = t.getPricePerKm() * Math.abs(getHome() - getLocation());
35         getWallet().takeOut( payment );
36         v.getDriver().pay( payment );
37     }
38 }
```

The following is what your colleague implements to test the Passenger.goHome() method using mocking. Assume that these mocks throw an exception—i.e. fail the test—if any

unexpected calls are made. Also assume that the order doesn't matter.

```
1  @Test
2  void testGoHome() {
3      Transport   mockTransport   = mock(Transport);
4      Vehicle     mockCar         = mock(Vehicle);
5      Driver      mockDriver      = mock(Driver);
6
7      // Tells our Transport interface to return our mocked Vehicle instead.
8      mockTransport.hailVehicle().andReturn(mockCar);
9
10     // Like above...except for Driver and we allow it to be called
11     // as many times as the dev desires.
12     mockCar.getDriver().anyTimes().andReturn(mockDriver);
13
14     mockDriver.setDestination( 5 );
15     mockTransport.getPricePerKm().anyTimes().andReturn( 1 );
16     mockDriver.pay( 10 );
17
18     // Done setting expectations.
19     replayAll();
20
21     // Testing...
22     Person  patrick = new Person( 5, 100, new Wallet(500) );
23     patrick.goHomeUsing( new UberTransport() );
24
25     // Verify our expectations were met.
26     verifyAll();
27 }
```

Part a. There are 3 mistakes in the above test. (A mistake causes the test case to not encode/verify the behaviour of the actual `Person` class.) **Identify** and **fix** them.
Part b. Your colleague then mentions that they forgot to add an assert to the test and, with haste, appends

```
assertTrue( patrick.getWallet().getAmount() == 405 );
```

to the end of the test. **Explain** why this is inconsistent and propose a change to the test code that utilizes mocking instead.

Question 4: Finite State Machines

Consider a candy dispensing machine that can either output Snarties or N&Ns. Possible input actions are:

- insert money (let's assume the input is $1, one coin only);
- select Snarties;

- select N&Ns;
- cancel operation.

Once the machine has received both the money and the choice of candy (in either order), it performs the output action, "dispense candy". The cancel button may trigger the "refund money" output action.

Draw a Finite State Machine which summarizes the potential behaviours of this machine; edges should be labelled with input actions and output actions (when appropriate). **Write down** a minimal test set (smallest total # of actions) that achieves Simple Round Trip Coverage. **Write down** a test set that achieves Complete Round Trip Coverage.

Question 5: Selenium

Imagine the following situation: Your task is to go through a list of 88 search results on a webpage. For each of the results, you need to visit the link, enter a value on the resulting page, and hit save. (You find yourself wondering which of your poor life choices resulted in this situation[1].)

You remember that you heard something about Selenium, which allows you to automate web interactions (or run tests). You attempt to record the required interaction, and you get the following XPath query as part of your result.

```
xpath=(//input[@id=dp1492644142915])
```

Regrettably, you find your macro does not generalize to other search results. **Explain why.**

You open the relevant input field in Firebug and find the following HTML:

```
<td class="label">
  <label for="custom_1_170">Board approval date</label>
</td>
<td class="html-adjust">
  <input id="custom_1_170" class="crm-form-text crm-hidden-date"
         data-crm-custom="OAC_Regular_Membership_fields:Board_approval_date"
         type="text">
  <input id="dp1492644142915" class="crm-form-text crm-form-date hasDatepicker valid"
         type="text" aria-invalid="false">
```

Write down an XPath query that will select the same input field as above, but that will also work across different search results. Include the assumptions that you are making to support your answer.

After using your macro a few times, you notice that it is flaky and sometimes fails to select the desired input. Assume that your macro is correct. **Explain** why your macro is flaky. Also **explain** how you can fix the macro to mitigate the failure.

[1] This scenario is not imaginary, but it is simplified.

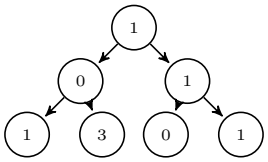
Question 6: Coverage

Consider the following code:

```
class TreeNode {
  int data;
  TreeNode left, right;
}

void traverse(TreeNode n, List<Direction> choices) {
  TreeNode cur = n;              // line A
  for (Direction d : choices) { // line B
    print (cur.data % 2);       // line C
    switch (d) {                // line D
      case Direction.LEFT:
        cur = cur.left; break;  // line E
      case Direction.RIGHT:
        cur = cur.right; break; // line F
    }
  }
  print (cur.data % 2);       // line G
}
```

Consider the following tree as the `TreeNode` input:



Part (a). Write down an input `choices` that ensures statement coverage when used together with the above tree.

Part (b). Assume that you have a single test case which takes the same tree as above. You can't look at the value of `choices`. When executed, this case produces output "1 0 1". For each labelled statement `A-G`, **write down** what you know about whether that statement was reached on this input (definitely reached; definitely not reached; don't know) and why you know it.

Question 7: Mutation

Propose two distinct non-stillborn and non-equivalent mutants for the following method. **Describe** the mutation operator that you used and where you apply it. Show that you can kill the mutants, demonstrating that they are non-equivalent mutants, by **writing down** test cases and relevant outputs for each of these mutants and the original method.

```
1  public static int odd(int[] x)
2  // Effects: if x == null, throw NullPointerException
3  // else: return the number of elements in x that are odd
4  {
5    int count = 0;
6    for (int i = 0; i < x.length; i++) {
7      if (x[i]%2==1 || x[i]%2==-1) {
8        count++;
9      }
10   }
11
12   return count;
13 }
```