University of Washington
CSE 403 Software Engineering
Winter 2016

Final Exam

Monday, March 14, 2016

Name: _____

CSE Net ID (username): _____

UW Net ID (username): _____

This exam is closed book, closed notes, closed neighbor. You have **110 minutes** to complete it. The exam contains 8 pages (including this cover page) and 10 problems.

Before you start, please check your copy to make sure it is complete. Turn in all pages, together, when you are finished. **Write your initials on the top of *all* pages**, in case a page gets separated during test-taking or grading.

When you are asked for multiple answers, give answers that are as different as possible, and give the most important answers.

Please write neatly; we cannot give credit for what we cannot read.

Good luck!

Problem	Points	Score
1	8	
2	4	
3	16	
4	4	
5	22	
6	12	
7	10	
8	8	
9	6	
10	10	
Total:	100	

1 Software Architecture

1. (8 points) List and briefly explain the criteria for evaluating a software architecture.

   (a) Coupling: the kind and quantity of interconnections among modules

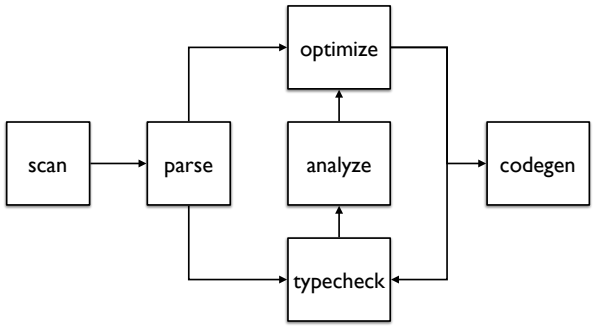
   (b) Cohesion: how closely the operations in a module are related

   (c) Style conformity: whether the architecture belongs to a family of systems defined by a particular architectural style (e.g., a pipe and filter architecture)

   (d) Component matching: whether the components in an architecture make compatible assumptions about their operating environment

> Many answers to this question involved describing general properties of software or code, rather than the properties of architectures (as covered in Lecture 8, slides 18-27). We gave partial credit (1 point) for each answer that was related to software architectures, even if it was not one of the four criteria discussed in class.

2. (4 points) Does the following diagram describe a pipe-and-filter architecture? Why or why not?



No, because a pipe-and-filter architecture cannot have cycles.

> We gave partial credit (2 points) for answering 'no' but giving an incomplete explanation.

2 Design Patterns

Ben Bittwiddle wrote the obfuscated Java code in Figure 1 to prevent others from understanding his design. But despite his efforts, it is clear that he used **three design patterns** we discussed in class.

```
public interface I1 {
  public I2 method1();
}

public interface I2 {
  public void method2(I3 i3);
}

public interface I3 {
  public void method3();
}

public final class C1 implements I1 {
  private static C1 c1 = new C1();
  private C1() {}

  public static C1 method4() { return c1; }

  public I2 method1() { return new C2(); }

  private static class C2 implements I2 {

    C2() {}

    public void method2(I3 i3) {
      i3.method3();
    }
  }
}

public class C3 implements I3 {
  private final String s;

  public C3(String s) {
    this.s = s;
  }

  public void method3() {
    System.out.print(s);
  }
}
```

```
public class C4 implements I3 {
  private final I3 i3;

  public C4(I3 i3) {
    this.i3 = i3;
  }

  public void method3() {
    System.out.print("***");
    i3.method3();
  }
}

public class Main {
  public static final void main(String[] args) {
    final I3 x1 = new C4(new C3("*"));
    final I2 x2 = C1.method4().method1();
    for(int i = 0; i < 100; i++) {
      x2.method2(x1);
    }
    x2.method2(new C4(new C3("")));
  }
}
```

Figure 1: Ben's code. Assume that all of the shown interfaces and classes are in the same package.

3. (16 points) Which three patterns did Ben use and how? Provide your answer below by filling the circle in the row r and column c **if and only if** Ben used the pattern in the row r, and the class or interface in the column c participates in that pattern. The same class or interface may participate in multiple patterns. You get to fill in **eight circles**, with each correct choice earning 2 points. No credit will be given if more than eight circles are filled.

	I1	I2	I3	C1	C2	C3	C4
Template Method	○	○	○	○	○	○	○
Factory Method	○	○	○	○	○	○	○
Singleton	○	○	○	●	○	○	○
Null Object	○	○	○	○	○	○	○
Abstract Factory	●	●	○	●	●	○	○
Decorator	○	○	●	○	○	●	●
Composite	○	○	○	○	○	○	○

Class `C1` uses the Singleton pattern. Classes `C1` and `C2`, together with the interfaces `I1` and `I2`, implement the Abstract Factory pattern. In particular, `I1` and `I2` define the abstract factory and product, respectively, and `C1` and `C2` define the concrete factory and product, respectively. Classes `C3` and `C4` and interface I3 implement the Decorator pattern, where `C4` decorates implementations of `I3` (such as `C3`) by adding extra functionality.

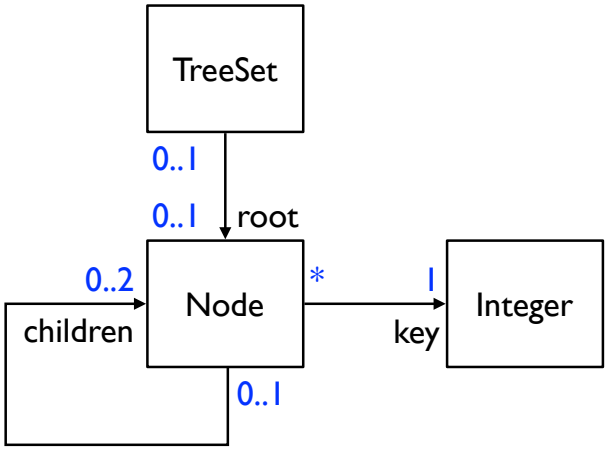
Many answers confused the Factory Method with the Abstract Factory pattern. If no circles were filled in the Abstract Factory row, we gave partial credit (1 point) for each circle in the I1, I2, C1, and C2 columns of the Factory Method row. A few answers identified the right patterns (row) but not the right participants (columns). We gave 1 point for each correctly identified pattern (row).

4. (4 points) What is the output of the `main` method of the `Main` class?

It prints a string of 403 stars (‘*‘) with no whitespace symbols.

3 UML Diagrams

Consider implementing a set of integers using a binary search tree. Below is an incomplete UML diagram that describes such an implementation.



5. (22 points) Which of the following constraints can be expressed using multiplicities? Mark your answers by circling T (true) for those that can and F (false) for those that cannot.
   (a) **T** / **F** The tree of Nodes is acyclic.
   (b) **T** / **F** A TreeSet can be empty.
   (c) **T** / **F** A Node can have no children.
   (d) **T** / **F** A Node can have one child.
   (e) **T** / **F** A Node can have at most two children.
   (f) **T** / **F** The tree of Nodes is balanced, so that the path from the root Node to the farthest leaf is no more than twice as long as the path from the root Node to the nearest leaf.
   (g) **T** / **F** Nodes are not shared among TreeSets.
   (h) **T** / **F** An Integer may appear in any number of TreeSets.
   (i) **T** / **F** Each Node has exactly one key.
   (j) **T** / **F** Subtrees are not shared within the Node tree.
   (k) **T** / **F** A TreeSet has no more than one root Node.

6. (12 points) Add multiplicity annotations to the TreeSet diagram so that it expresses *all* of the constraints you marked as true (T) in the previous question.

> We gave partial credit (1 point) for each annotation provided in Question 6 that was consistent with an incorrect answer to Question 5. The most common mistake on Question 5 involved marking 5g, 5h, and 5j as 'false'. We also gave partial credit (1 point) for writing down one correct multiplicity bound for the source and target multiplicities of the `root` and `children` edges.

4 Testing

Ben Bittwiddle just joined a startup that builds electronic voting machines for the upcoming presidential elections. His first programming task is to implement an efficient algorithm for determining which candidate received the majority of votes. Luckily, Ben remembers hearing once about the Boyer-Moore majority-vote algorithm, which runs in linear time and constant space. He looks it up on Wikipedia and finds the Java implementation shown in Figure 2.

Being a good engineer, Ben wants to test the implementation before deploying it. In particular, he wants to make sure that the implementation satisfies the specification shown in Figure 2. But Ben forgot his laptop charger at home, and his battery is down to 2%! The following two questions ask you to save the day (and the democratic process) by helping Ben test his code with minimal computing resources.

7. (10 points) To make the most of his resources, Ben devises a way to compare the sizes of two test suites for the `majorityElement` method. First, he defines a helper procedure `packArray`(a) that takes as input an array of positive integers and concatenates all of its elements into a single integer. For example, `packArray`($[5, 3, 4]$) returns 534. Next, he defines a helper procedure `packArrays`(T) that takes as input a list T of integer arrays, packs each element of T into an integer using `packArray`, sorts the resulting integers, and then packs those into a single integer using `packArray`. For example, `packArrays`($\{[5, 3], [4], [1, 6]\}$) returns 41653. Using `packArrays`, Ben can now say that a test suite T_1, expressed as a list of integer arrays, is *smaller* than a test suite T_2 if and only if `packArrays`(T_1) $<$ `packArrays`(T_2).

   Using Ben's definition of test suite size, write down a *minimal test suite* for the `majorityElement` method that achieves full statement coverage and that exercises all specification outcomes (i.e., a positive and a negative output). In particular, your test suite T should be satisfy both of these criteria, and there should be no test suite T' that also satisfies these criteria while being strictly smaller than T. For each test input in T, write down the expected output.

   The minimal test suite is $\{[1, 1], [1, 2]\}$, with expected outcomes $[1, 1] \to 1, [1, 2] \to -1$.

   The minimal test suite achieves full statement coverage, while testing the case where a majority element exists and also the case where it does not exist.

   > Some answers included the empty array [] as well as $\{[1, 1], [1, 2]\}$, because it is not prohibited by the spec. Those answers received full credit. Note that the spec prohibits the input array(s) from containing zeros. We deducted 1 point for using zeros. Partial credit (8 points) was given for non-minimal test suites that satisfied the coverage criteria.

8. (8 points) Running your test suite, Ben discovers an error! The implementation returns a wrong result.

   (a) Briefly explain which test case fails and why. That is, state which line of code in Figure 2 is faulty.
   The code fails to produces 1 instead of -1 on the input $[1, 2]$, because the condition on line 24 uses integer division. With integer division, $3 \backslash 2 = 1$, which causes the method to erroneously return 1.

   > Partial credit (3 points) was given for identifying the correct line but providing an incorrect explanation. If the test suite included the empty array, full credit (4 points) was given for identifying line 9 and explaining that it causes an out-of-bounds exception.

   (b) Write a fix for the faulty line of code.
   Many answers are possible. For example, `if (counter < (n + 1.0)/ 2.0) return -1;`

   > We also gave full credit (4 points) for answers that proposed a correct fix to line 9.

```java
public class MajorityVote {
    /**
     * Given an array of n positive integers, this method returns the element k,
     * if one exists, that is stored at more than ⌊n/2⌋ indices of the array.
     * If no such integer exists, the method returns −1.
     **/
    public int majorityElement(int[] num) {
        int n = num.length;
        int candidate = num[0], counter = 0;
        for (int i : num) {
            if (counter == 0) {
                candidate = i;
                counter = 1;
            } else if (candidate == i) {
                counter++;
            } else {
                counter--;
            }
        }
        counter = 0;
        for (int i : num) {
            if (i == candidate) counter++;
        }
        if (counter < (n + 1) / 2) return -1;
        return candidate;
    }
}
```

Figure 2: A Java implementation of the Boyer-Moore majority voting algorithm, as published on Wikipedia (March 03, 2016).

5 Static Analysis and Symbolic Execution

9. (6 points) Finish the implementations of static analysis tools sketched in Figure 3 so that
   (a) `ToolA` is sound for all Java programs p and all properties s.
   (b) `ToolB` is complete for all Java programs p and all properties s.
   (c) `ToolC` is neither sound nor complete for all Java programs p and all properties s.

> We gave partial credit (3 points) for answers that consistently flipped soundness and completeness.

```
public class ToolA {

    public static boolean analyze(Program p, Property s) {

      return false;

    }
}

public class ToolB {

    public static boolean analyze(Program p, Property s) {

      return true;

    }
}

public class ToolC {

    public static boolean analyze(Program p, Property s) {

      // Many answers are possible, for example:
      return java.util.Random.nextBoolean();

    }
}
```

Figure 3: A sketch of three static analysis tools, to be filled in so that `ToolA` is sound, `ToolB` is complete, and `ToolC` is neither. A tool should return `true` if the input program satisfies the property, and `false` otherwise. The types `Program` and `Property` represent the program and the property being analyzed.

10. (10 points) Assume that Ben has applied your patch to the `majorityElement` method in Figure 2. What feasible path conditions are generated by symbolically executing the resulting code on the array $[X, Y]$, where X and Y are symbolic integer values? For each such path condition, write down a simplified formula that omits trivial constraints (e.g., 'true'), as well as the return value of the patched `majorityElement` method.

   There are two feasible path conditions:

   $X = Y$ leads to the return value X

   $X \neq Y$ leads to the return value -1