Student ID: **Student Name:**

Signature: **Course Number:**

Department of Electrical and Computer Engineering
SE465/ECE453/CS447/ECE653/CS647—Software Testing, Quality Assurance & Maintenance
Instructor: Lin Tan
Examination Date and Time: Monday April 11, 2016, 12:30 PM–3:00 PM
Rooms: PAC 1,2,3,9,10

Instructions:

- You have **2 hours and 30 minutes** to complete the exam.
- You can bring **printed or handwritten** material, e.g., books, slides, notes, etc.
- If you separate the pages, make sure your names and student IDs are on the top of every page.
- Unauthorized duplication or archival of these questions is not permitted. To be returned after the completion of the exam.
- If information appears to be missing from a question, make a reasonable assumption, state your assumption, and proceed. Do not simplify the question.
- You must answer questions in the space provided.
- **Illegible answers receive NO points.**

Question	Mark	Points
Q1		14
Q2		11
Q3		16
Q4		9
Q5		10
Q6		14
Q7		10
Q8		8
Q9		8
Total		100

1 Short Answers: Integration Testing, Coverage Criteria, Tools, Concurrency, and Subsumption (14 points)

Briefly answer the following questions. Illegible answers receive no points.

(a) Describe one *integration* testing techniques, and explain the pros and cons. [3 points]

(b) Why aren't the coverage criteria All-du-Paths Coverage (ADUPC), Correlated Active Clause Coverage (CACC), and Multiple Base-Choice Coverage (MBCC) more widely used in industry? List three reasons. Each reason should apply to at least one of the criteria above. [3 points]

(c) List two symbolic-execution-based testing tools. [2 points]

(d) Briefly describe and explain two potential types of bugs that can arise from multi-threaded programming. Pick one type of bugs from the two types of bugs you give, and describe at least one testing technique that can detect this type of bugs. [3 points]

(e) General Active Clause Coverage (GACC) does not subsume Predicate Coverage (PC). Provide a counterexample. [3 points]

2 Better Bug Reports and Coverity [11 points]

Consider the following function that was extracted from Apache Commons Lang 3.4's file `StrBuilder.java`:

```
1   /**
2    * Check the contents of this builder against another to see if they contain
3    * the same character content ignoring case.
4    *
5    * @param other    the object to check, null returns false
6    * @return true if the builders contain the same characters in the same order
7    */
8    public boolean equalsIgnoreCase(StrBuilder other){
9     if (this == other) {
10      return true;
11    }
12    if (this.size != other.size) {
13      return false;
14    }
15    char thisBuf[] = this.buffer;
16    char otherBuf[] = other.buffer;
17    for (int i = size -1; i>=0; i--){
18      char c1 = thisBuf[i];
19      char c2 = otherBuf[i];
20      if (c1 != c2 && Character.toUpperCase(c1) != Character.toUpperCase(c2)){
21        return false;
22      }
23    }
24    return true;
25  }
```

We ran Coverity on this project and obtained the following warning:

```
--------------------------------------------------------------------------------
Defect 10001
Checker NULL_RETURNS
File: /src/main/java/org/apache/commons/lang3/text/StrBuilder.java   line: 12
Dereference Null Return Value "other".
--------------------------------------------------------------------------------
```

(a) Explain the Coverity warning. Is it a true bug or a false positive? Justify. If it is a true bug, propose a fix for this bug. [4.5 points]

(b) List at least 3 elements that should be included in a good bug report. [1.5 points]

(c) Write a good bug report for this warning. In your bug report, use only the provided information. For example, there is no need to write about hardware since such information is not provided. Be sure to include a test case in JUnit format. [5 points]

This page is left blank intentionally. Continue writing your answer for Q2 in this page.

3 Graph Coverage and Specifications [16 points]

Robot Luna is a coffee delivery android that is responsible for brewing coffee and delivering the brewed coffee to the boss. Luna checks the status of the boss's coffee cup periodically. If the status shows the cup is empty, it brews a new cup of fresh coffee for the boss. If the status shows the cup is not empty, it waits an hour before checking again. If the status shows the boss had drunk too much coffee, it shuts down to protect the boss's health and safety. After Luna brews a cup of coffee, it obtains the status of the coffee machine, and if the status is "0" (no error), it cleans the paper filter. The status of the coffee machine is obtained by calling `brewCoffee()` in line 11, where status "0" indicates the coffee machine has no error, and a non-zero status value indicates that there is an error. The following code snippet implements robot Luna's behaviour.

```
1  public static void main (String[] args) {
2      Boolean paperFilterIsClean = true;
3      String returnedStatus;
4      while (true) {
5          returnedStatus = checkCoffeeStatus();
6          switch (returnedStatus) {
7              case ``drinked too much coffee'':
8                  System.exit(); // final node
9              case ``coffee cup is empty'':
10                 //brewCoffee crashes if paperFilterIsClean is false
11                 int status = brewCoffee(paperFilterIsClean);
12                 if (status == 0) { // no error in coffee machine
13                     changePaperFilter();
14                     paperFilterIsClean = true;
15                 } else { // error in coffee machine
16                     paperFilterIsClean = false;
17                 }
18                 deliverCoffee();
19                 break;
20             case ``coffee cup is not empty'':
21                 wait();
22                 break;
23         }
24     }
25 }
```

(a) Draw the minimal node (hint: 8) basic block control flow graph (CFG) for the function `main()`. You must use only the line numbers provided above as the content of each node (e.g., 20–25, if lines 20–25 belong to a basic block), and label the nodes using only circled lowercase letters (i.e., nodes (a), (b), (c), (d), (e), (f), (g), and (h)). Exclude line 1, 10, 17, 23–25 from your graph (include all other lines). Ensure the graph is clear and legible. Unclear graphs will not receive full points. [5 points]

(b) Create the Def-Use CFG for variable `paperFilterIsClean` by annotating the relevant nodes with "def" and "use". Ensure the graph is clear and legible. Unclear graphs will not receive full points. [2 points]

(c) The hardware designers of the robot stated that robot Luna should never attempt to brew a coffee without a clean paper filter. Show a **minimal** test path that violates this constraint. [2 point]

(d) Suppose that `brewCoffee()` (line 11) crashes if Luna attempts to brew coffee without a clean paper filter. A coverage criterion guarantees to uncover this failure if any test set that satisfies the coverage criterion will uncover the failure. For example, Complete Path Coverage (CPC) guarantees to uncover this failure. List all structural and data flow coverage criteria (choose from PPC, EPC, EC, NC, ADUPC, AUC, and ADC) that guarantee to uncover this failure (along with the test requirement that uncovers the failure). [5 points]

(e) What is reachability condition of line 11? [2 points]

This page is left blank intentionally. Continue writing your answer for Q3 in this page.

4 Inter-procedural Data Flow Coverage [9 points]

The following functions, together, implement the pancake sort algorithm for an array of integers. The algorithm works on the principle that a spatula can be inserted at any point in a stack of pancakes, and then used to flip all pancakes above it. The algorithm considers each integer value to represent the diameter of a pancake in the stack, and the order of the integers denote the order of the pancakes in that stack. Index position 0 represents the top of the stack. The largest index position represents the bottom of the stack. A sort is a sequence of flips which arranges the pancakes from smallest to largest, with the largest pancakes at the bottom. The programmer invokes the `pancakeSort()` function on an array of integers to sort the integers.

```
1  public static void pancakeSort(int [] pancakes) {
2      if (pancakes.length <= 1)   // nothing to flip
3          return;
4      int i, the_spot;
5      // analyze the pancake stack from bottom to top
6      for (i = pancakes.length;
7           i > 1;
8           i--) {
9          // find the index of the largest pancake not yet sorted
10         the_spot = findFlipSpot(pancakes, i);
11         // if this pancake is already in place, continue
12         if(the_spot == i - 1)
13             continue;
14         // else, flip this pancake to index 0
15         if(the_spot > 0)
16             flip(pancakes, the_spot + 1);
17         // then flip this pancake to its place
18         flip(pancakes, i);
19     }
20     return;
21 }
22
23 // finds the largest pancake in the stack from index 0 to i
24 private static int findFlipSpot(int [] pancakes, int i) {
25     int the_spot = 0;
26     for(int eh = 0; eh < i; eh++) {
27         if(pancakes[eh] > pancakes[the_spot])
28             the_spot = eh;
29     }
30     return the_spot;
31 }
```

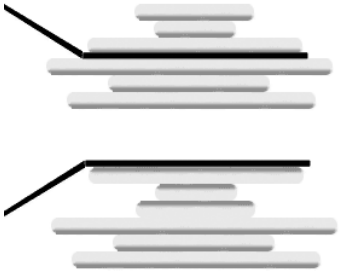

Figure 1: Pancake Flip

List three inter-procedural DU pairs related to `pancakeSort()`. Use a tuple (function name, variable name, line number) to denote a location. For example,

```
(pancakeSort, i, 6) -> (findFlipSpot, i, 26)
```

denotes an inter-procedural DU pair: the last def of `i` in line 6 of method `pancakeSort` reaches the first use of `i` in line 26 of method `findFlipSpot`. Don't use this example in your answer. [9 points]

5 Applications of Statement Coverage [10 points]

Test case prioritization schedules test cases for regression testing in an order that attempts to maximize specific objectives, e.g., achieving coverage as fast as possible. *Total statement coverage* and *additional statement coverage* are two kinds of coverage-based test case prioritization strategies.

Total statement coverage: sorts test cases in descending order of total number of statements covered by each test case. If two test cases cover the same number of statements, randomly select one.

Additional statement coverage: selects, in turn, the next test case that covers the maximum number of statements not yet covered. If two test cases cover the same number of additional statements, randomly select one.

In this question, you will apply these two test case prioritization strategies on a sample program with 10 statements and 6 test cases. Each test case covers at least one statement. We've computed test case coverage for you in the following table. TC stands for "test case". An × means that a test case covers a statement, e.g, test case TC1 covers statements s1, s3, s5, s7, s8, and s9.

Statements	**TC1**	**TC2**	**TC3**	**TC4**	**TC5**	**TC6**
s1	×				×	×
s2		×			×	
s3	×			×	×	
s4			×			
s5	×					×
s6		×				
s7	×					×
s8	×			×		
s9	×					
s10		×			×	

(a) Write down prioritization orders with respect to *total statement coverage* and *additional statement coverage*. A prioritization order is a sequence of test cases. [6 points]

(b) For the sample program, which test case prioritization strategy do you think is better? Explain why. [2 points]

(c) In practice, one often collects the code coverage of test cases from the last execution of the regression test suite. In other words, the code coverage is from executing the test cases on the version without the code change. However, the collected code coverage may not be accurate for the purpose of regression testing, e.g., achieving *total statement coverage* and *additional statement coverage*. Why? [2 points]

6 Logic Coverage [14 points]

Consider a learning management system where students can inquiry about their exam results (like Waterloo Learn).

(a) Suppose that a student can make an inquiry only if he or she has taken the exam and the grade has been uploaded by the instructor or a TA. The corresponding clauses are listed below:
a = *"the student has taken the exam"*
b = *"the grade has been uploaded by the instructor"*,
c = *"the grade has been uploaded by a TA"*,
Please write down the corresponding predicate p. Use only the three clauses a, b, and c, and logic operators $\wedge$, $\neg$, and $\vee$. [2 points]

(b) According to the clauses and predicate you wrote in question (a), compute and simplify the conditions under which each of the clauses determines predicate p. Show all steps that simplify the conditions. The conditions must use $\wedge$, $\neg$, and $\vee$ only, and contain the minimal number of logic operators. Complete the table. [6 points]

	a	b	c	p	p_a	p_b	p_c
1	T	T	T				
2	T	T	F				
3	T	F	T				
4	T	F	F				
5	F	T	T				
6	F	T	F				
7	F	F	T				
8	F	F	F				

(c) Identify all pairs of rows from your table that satisfy CACC with respect to each clause. [2 points]

(d) Write a minimal (the number of test requirements is minimal) set of test requirements (TR) for CACC. Write the minimal set of test requirements as a set of rows. [2 points]

(e) Identify all pairs of rows from your table that satisfy RACC with respect to each clause. [2 points]

7 Mutation Testing [10 points]

Consider the following original method, where `aLength` is the length of array `a`:

```
1  static int findMax(int a[], int aLength) {
2      // if a is null, throw NullPointerException
3      // if aLength <= 0, undefined behaviour
4      int max = a[0];
5
6      for (int i=0; i < aLength; i++) {
7          if (a[i] > max) {
8              max = a[i];
9          }
10     }
11     return max;
12 }
```

Here is a mutant of the above method:

```
1  static int findMax(int a[], int aLength) {
2      // if a is null, throw NullPointerException
3      // if aLength <= 0, undefined behaviour
4      int max = 0; // <-- mutation
5
6      for (int i=0; i < aLength; i++) {
7        if (a[i] > max) {
8          max = a[i];
9        }
10     }
11     return max;
12 }
```

(a) Identify the necessary and sufficient condition for weakly killing the mutant. Justify your answer. [6 points]

(b) The test case ($a = [0, 1, 2]$, $aLength = 3$, and *expected output* $= 2$) does not kill this mutant. Propose a new mutant for the original method that will be killed by this test case, and explain why this test case kills the mutant. [4 points]

8 Applying Input Space Partitioning for Test Case Design [8 points]

Consider the method `String TriangleType(int a, int b, int c){...}`, which given three `int` parameters, determines the types of triangles these three integers form. The three `int` values represent the lengths of the three sides of a triangle. This method outputs a message that states whether the triangle is scalene (i.e., no two sides have equal length), isosceles (two sides have equal length), or equilateral (all three sides have equal length). This method assumes that the three integers given form a valid triangle, meaning that the sum of any two integers must be greater than the third. All other inputs are considered invalid.

(a) This method can take infinite number of inputs. To design test cases, we can divide all possible inputs into equivalence classes. Examples are listed below. Please list two new equivalence classes for valid inputs, and two new equivalence classes for invalid inputs. Note that these equivalence classes do not have to be disjoint from each other. [4 points]

Equivalence classes for valid inputs:

(1) The three input integers are identical ($a = b = c$).

(2) Two of the three input integers are equal ($a = b \neq c$);

(3)

(4)

Equivalence classes for invalid inputs:

(1) Some integer input values are 0, e.g., $a = 0$.

(2) Only two input values are provided.

(3)

(4)

(b) Based on the listed equivalence classes, design at least four test cases to test the function. Follow the example and fill in the table. [4 points]

Test Case No.	Equivalence Class	Input Values (a,b,c)	Expected Result Messages
1	Valid (1)	2,2,2	Equilateral Triangle
2	Valid (2)	3,3,2	Isosceles Triangle
3	Invalid (1)	1,1,0	Invalid Inputs
4	Invalid (2)	1,1	Invalid Inputs
5	Valid (3)		
6	Valid (4)		
9	Invalid (3)		
10	Invalid (4)		

9 Input Space Partitioning [8 points]

Consider the following characteristics and blocks.

	Characteristics	Block 1	Block 2	Block 3	Block 4
Partition 1	**Size**	2-door	4-door	hatch-back	
Partition 2	**Energy Source**	Gas	Electric	Hybrid	
Partition 3	**Where Made**	North America	Europe	Asia	Other

(a) Give test cases to satisfy the Base Choice Coverage (BCC) criterion. Assume the base choices are $Size = hatch - back$, $EnergySource = Hybrid$, and $WhereMade = NorthAmerica$. Complete the table. [3.5 points]

Test Case No.	Size	Energy Source	Where Made
1	hatch-back	Hybrid	North America
2			
3			
4			
5			
6			
7			
8			

(b) How many *test cases* are needed to satisfy the All Combinations Coverage (ACoC) criterion? Show your calculation. (Do not list all the test cases!) [2 point]

(c) How many *test requirements* are there for the Pair-Wise Coverage (PWC) criterion? Show your calculation. (Do not list all the test requirements!) [2.5 points]