*On the final exam you will be provided UML diagrams for design patterns (as in the midterm) and also JavaDoc description of methods in the Reflection API.*

1. True/False (if false, explain why)

   a. The following method would allow an object of the static type `List<String>` to be passed to it as an argument.

      ```
      void method(List<? extends Object> arg);
      ```

      **True**

   b. Complex software systems are less likely to be attacked by hackers because there are more paths of execution for attackers to consider.

      **False, the principle of Economy of Mechanism states that secure software should be as simple as possible.**

   c. A software tester has an ethical duty to prove that all bugs are removed from a program prior to delivery.

      **False, software testing does not prove the absence of bugs (only the presence of bugs).**

   d. Cross-site scripting attacks are common because users will often open multiple web sites in different tabs of their web browser at the same time.

**False, “code downloaded from site A can’t access data in browser tab of site B (and vice versa)”.**

2. Short Answer

a. "The term **'need to know'**, when used by military organizations, describes a restriction on data which is considered very sensitive. Under need-to-know restrictions, even if one has the necessary military rank to access certain information, one would not be given access to such information, unless one has a specific *need to know*."

Which Security Design Principle is most related to this concept as it might be applied to computer software design?

**Principle of Least Privilege**

b. Why would equivalence class partitioning be better way of selecting test cases than just choosing a set of random test input values?

**Equivalence class partitioning helps to limit the number of test cases while still covering equivalent behaviors in the software specification.**

c. What is the motivation behind boundary value analysis (i.e. why is it useful)?

**Errors tend to occur at the boundaries between equivalence classes**

d. What is the difference between black box testing and white box testing?

**Black box testing: Use the documented specification to choose test cases.**

**White box testing: Use the program structure (control-flow graph) to choose test cases.**

e. In the StockWatcher tutorial (or any GWT project), you need to write a "Service" interface in order to access server classes from client classes. Which Design Pattern that we discussed in class is being used in this situation?

**Remote Proxy**

f. Describe one similarity and one difference between Scrum and Extreme Programming?

**Similarity: both try to include customers in the creation of test cases or acceptance criteria.**

**Difference: Scrum focuses only on general project management (e.g. in any engineering discipline) while Extreme Programming also includes some additional software engineering specific practices.**

3. Explain why a test set which achieves branch coverage for a particular program also achieves statement coverage for that program. You can assume there is no "dead code" in the program (i.e. statements which could never be executed).

   Hint: you could (but don't have to) use a short proof-by-contradiction e.g. you could start with "*Assume for the sake of contradiction, there is a test set which achieves branch coverage but not statement coverage for some program ….*"

   **Assume for the sake of contradiction, there is a test set which achieves branch coverage but not statement coverage for some program.**

   **This implies there is some statement node in the CFG which is not covered.**

   **Without loss of generality, choose any statement node which is not covered.**

   **This statement node has some number of directed edges that point to this node, otherwise this statement could never be executed.**

   **Based on the original assumption, all of these edges are covered (because there is branch coverage).**

   **This implies that the statement node is covered, which is a contradiction.**

4. Consider the following method:

```
String triangleType (int x, y, z) {
        String r="scalene";
        if (x==y || y==z)
                r="equilateral";
        if (x==z)
                r="isosceles";
        if (x <= 0 || (x+y) <= z)
                r="";
        return r;
}
```

4.a On the following page, draw a control flow graph which captures the order that expressions in the code might be evaluated. In other words, each distinct path through the CFG will represent the order that the expressions are evaluated for one possible execution of the method. Model the evaluation of the || operator as follows:

each operand to an || should be placed in a separate node and conditionals ("ifs") containing an || operator should be modeled in the diagram based on the following logical template, where A, B, C, and D are placeholders for other Java code.

```
if(A || B) {
        C;
}
D;
```

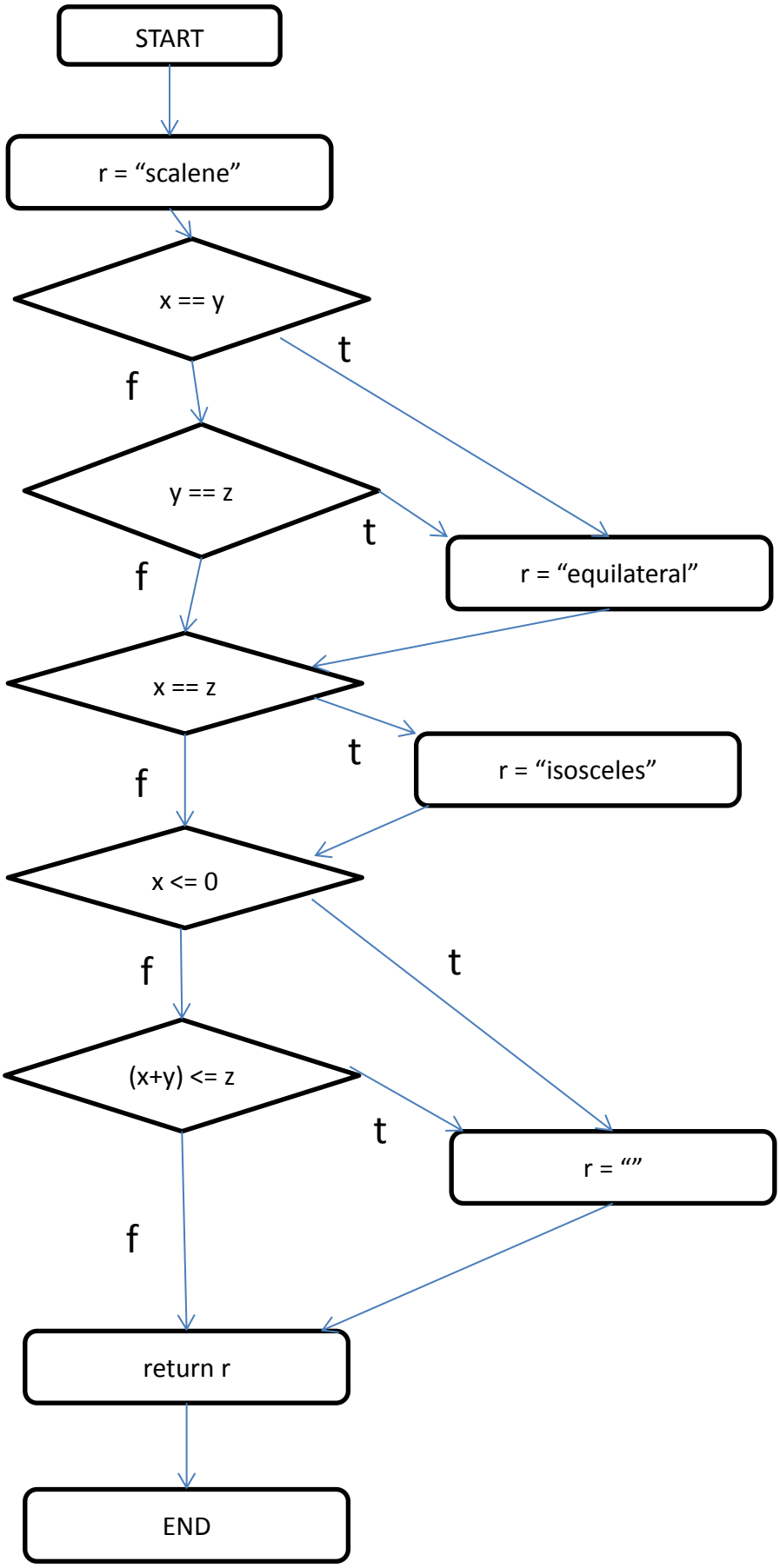
If A is true, then C is executed immediately following A (B is skipped).

If A is false, then B is executed immediately following A.

If A is false and B is true, then C is executed immediately following B.

If A is false and B is false, then D is executed immediately following B (C is skipped).

4.a. Draw the CFG for the method according to the previous described logic here.



4.b. Give a test set of minimal size for `triangleType` which provides branch coverage.

Use three test cases.

1. (x==y) && (x == z) && (x <= 0)

{ x = -1, y = -1, z = -1}

2. (x != y) && (y == z) && (x != z) && (x > 0) && ((x+y) > z)

{ x = 1, y = 5, z = 5}

3. (x != y) && (y != z) && (x != z) && (x > 0) && ((x + y) <=z)

{ x = 1, y = 2, z = 10}

5. You are testing a method that computes the cost for a golf game reservation. The regular cost is $35. However, a senior (age 55 and over) and a junior (age 18 and below) pay a reduced rate at $30. Also, if the time is on a weekday, an additional 10% discount is applied. Suppose that the method takes the age of a player and day of the week as the input. Define a test set for the method using equivalence classes and boundary value analysis. For each test case input, also provide the expected output.

Equivalence Classes:

1. ReducedRate && Weekday
2. !ReducedRate && Weekday
3. ReducedRate && !Weekday
4. !ReducedRate && !Weekday

Test cases:

Could have included more edges cases around each boundary (e.g. age = 54, age = 55, age = 56) but only one is required for exam.

a. Eq. Class 1
   **{ age = 10, day= "Wednesday"} = $27**
b. Eq. Class 2
   **{ age = 25, day= "Wednesday"} = $31.50**
c. Eq. Class 3
   **{ age = 60, day= "Saturday"} = $30**
d. Eq. Class 4
   **{ age = 25, day= "Saturday"} = $35**
**e.** Boundary between 1 and 2
   **{ age = 18, day= "Wednesday"} = $27**
   **{ age = 55, day= "Wednesday"} = $27**
f. Boundary between 1 and 3
   **{ age = 10, day= "Saturday"} = $30**
   **{ age = 60, day= "Friday"} = $27**
g. Boundary between 1 and 4
   **{ age = 18, day= "Saturday"} = $30**
   **{ age = 55, day= "Friday"} = $27**
h. Boundary between 2 and 3

**Same as boundary between 1 and 3**

i. Boundary between 2 and 4
   **{ age = 25, day= "Saturday"} = $35**
   **{ age = 25, day= "Friday"} = $31.50**
j. Boundaries between 3 and 4
   **{ age = 18, day= "Saturday"} = $30**
   **{ age = 55, day= "Sunday"} = $30**

6.a. Draw a state diagram for a bank account class which has the following requirements. Use transition guards where appropriate. You should use at least three state nodes in your diagram (i.e. you may use more but you are not required to do so).

i. New account has 0 balance and is open

ii. Money can be credited (added) to the account at any time

iii. Money can be sometimes be debited (removed) from the account

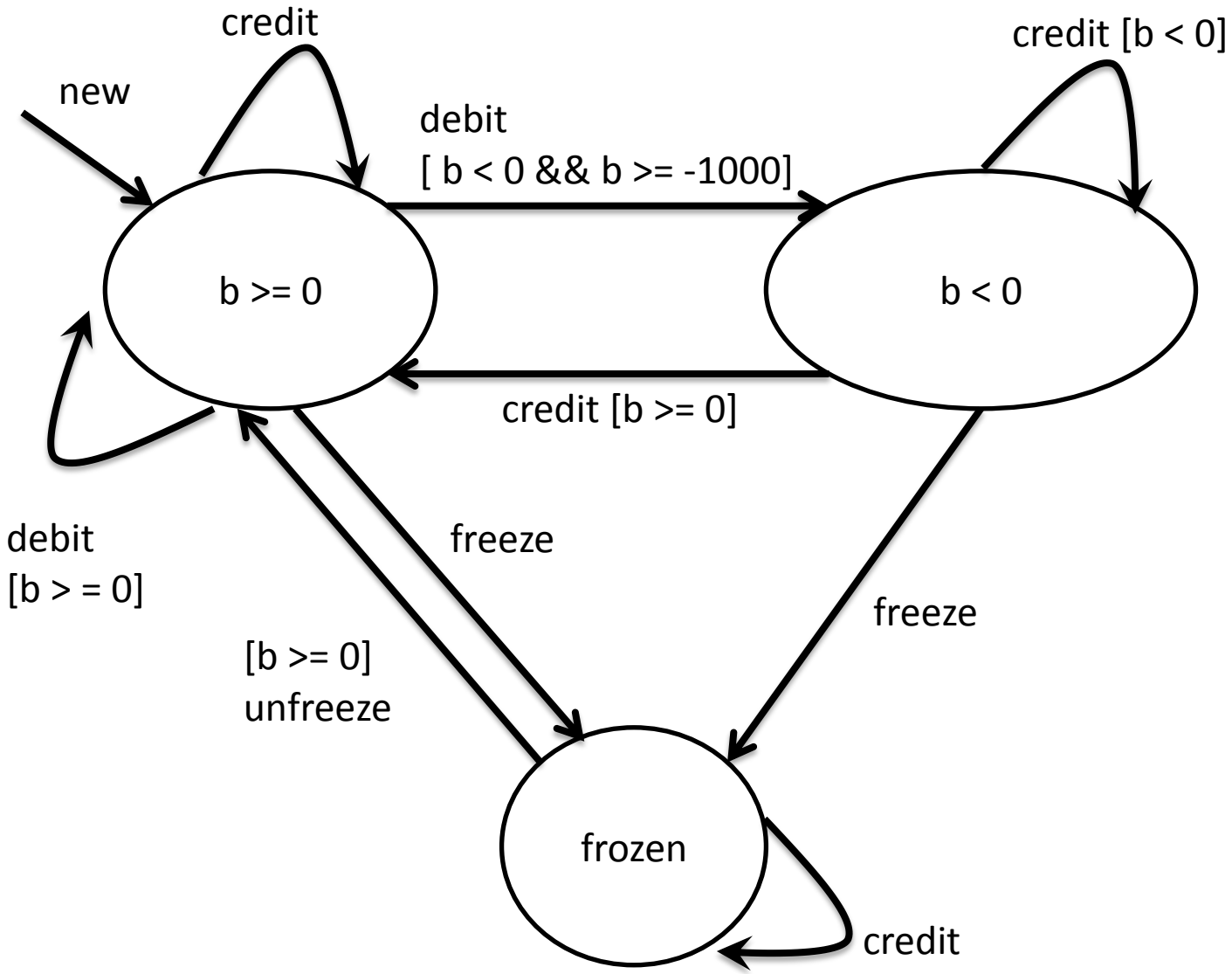
iv. Account can't be debited if the balance is currently less than zero or would be less than -1,000 after debit

v. Account can be "frozen" at any time, unless it is already frozen.

vi. Can't debit a frozen account

vii. A frozen account can only be unfrozen if the balance is greater than or equal to zero.

Balance is variable “b”



6.b. Write a set of test cases for which execution of all the test cases would ensure that every edge (i.e. transition) of the diagram is covered at least once. In other words, the covered edges of the test set is the union of the set of edges covered by each test case. The test cases only need to show the sequence of method calls and their arguments, it does not need to assert the output of the test case. There is no restriction on the number of test cases you need to use (other than that they cover the edges).

e.g. use a format for a test case following this example syntax:

**{** method1(), method2(5), method3(false), …**}**

{ credit(10), debit(5), freeze(), credit(5), unfreeze(), debit(20), credit(5), freeze() }

{ debit(20), credit(40) }

7. Implement a simple multi-player turn-based game framework using the Template Method pattern. You only need to implement one base class, not any sub-classes or other data-structures. Additional space is provided on the next page if needed.

The logical flow for the template (implemented in the base-class):

1. Setup. This step will determine the number of players.
2. Repeat, for each player in some fixed order:
    - Player takes their turn, if they win, immediately goto Finish.
3. Finish.

• Subclasses would implement the logic of Setup, Player's turns, and Finish, not the base-class.

• Subclasses would only implement individual steps, not any of the overall control-flow

```
public abstract class TemplateMethodQuestion {

     public void run() {
          int numPlayers = setup();
          int turn = 1;
          while(true) {
               boolean won = playerTurn(turn);
               if(won) {
                    break;
               }
               turn++;
               if(turn > numPlayers) {
                    turn = 1;
               }
          }
          finish();
     }

     public abstract int setup();

     /*currentPlayer argument is optional here, could
     potentially keep track of this in the sub-class*/
     public abstract boolean playerTurn(int currentPlayer);

     public abstract void finish();
}
```

8. *Scenario: your team has noticed that your project implementation has become cluttered by methods which contain too many arguments (i.e. parameters), decreasing the cohesion of your design. Your team has decided to search for all of them so that you can apply refactoring.*

On the final exam, the sections called "**Method Summary**" of these two web pages will be available to you:

http://docs.oracle.com/javase/7/docs/api/java/lang/Class.html

http://docs.oracle.com/javase/7/docs/api/java/lang/reflect/Method.html

Question: Write a method (you may include helper methods if you would like), to help identify these problematic methods. Given a single class name string as input, print the method name for any method with more than 5 parameters in its signature and the class name where that method is declared. This search should include any method declared in the class that was directly identified by the input string or in any of its super-classes transitively (you can ignore interfaces).

```
public static void printLongMethods(String className) throws Exception {
		Class<?> clazz = Class.forName(className);
		printLongMethodsHelper(clazz);
}

public static void printLongMethodsHelper(Class<?> clazz) {
		Method[] methods = clazz.getDeclaredMethods();
		for(Method method : methods) {
			Class<?>[] types = method.getParameterTypes();
			if(types.length > 5) {
				System.out.println("Method named " + method.getName() +
				"in class " + clazz.getName());
			}
		}
		Class<?> superClass = clazz.getSuperclass();
		if(superClass != null) {
			printLongMethodsHelper(superClass);
		}
}
```