Name:

Final Exam

CS 121

Software Engineering

Fall 2021

December 16, 2021

Instructions

This exam contains 12 pages, including this one. Make sure you have all the pages. Write your name on the top of this page before starting the exam.

Write your answers on the exam sheets. If you finish at least 15 minutes early, bring your exam to the front when you are finished; otherwise, wait until the end of the exam to turn it in. Please be as quiet as possible.

If you have a question, raise your hand. If you feel an exam question assumes something that is not written, write it down on your exam sheet. Barring some unforeseen error on the exam, however, you shouldn't need to do this at all, so be careful when making assumptions.

Question	Score	Max
1. Short Answer		20
2. Testing		20
3. Design Patterns and Testing		25
4. Concurrency		20
5. Design Patterns and Concurrency		15
Total		100

Question 1. Short Answer (20 points).

a. (4 points) Briefly explain how *dynamic dispatch* works in Java.

b. (4 points) Briefly explain what *refactoring* is and why it may be useful. Why is having a good test suite important for refactoring?

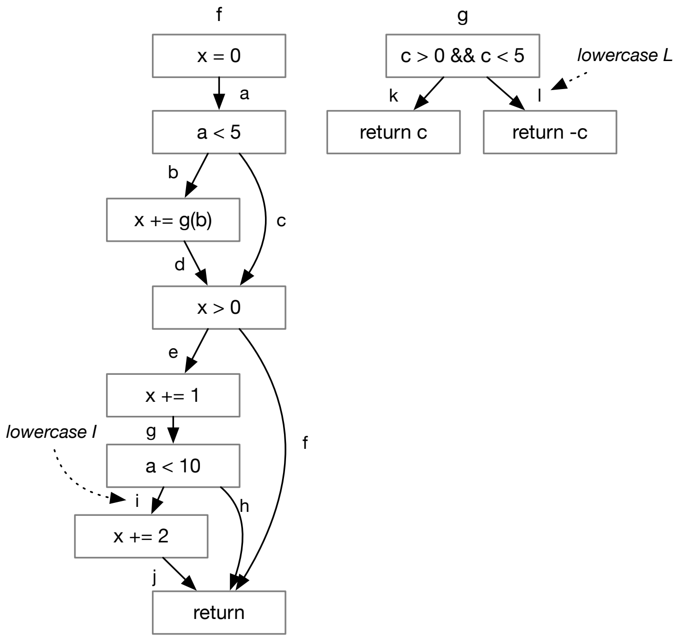
c. (4 points) In one sentence each, explain the role of the *model*, *view*, and *controller* in the model-view-controller architecture.

d. (4 points) The authors of *What Makes a Great Software Engineer* reported that *"informants discussed the propensity to overly anticipate needs in the face of uncertainty."* Briefly explain why overly anticipating needs when developing software may be detrimental.

e. (4 points) List two benefits of simplifying test cases.

Question 2. Testing (20 points). Consider the following program and its control-flow graph, where we've numbered statements and labeled control-flow edges with letters.

```
           void f(int a, int b) {
/*  1 */     int x = 0;
/*  2 */     if (a < 5) {
/*  3 */       x += g(b);
             }
/*  4 */     if (x > 0) {
/*  5 */       x += 1;
/*  6 */       if (a < 10) {
/*  7 */         x += 2;
           } } }
           int g(int c) {
/*  8 */     if (c > 0 && c < 5) {
/*  9 */       return c;
/* 10 */     } else {
/* 11 */       return -c;
           } }
```



a. (3 points) Write a single call that covers all *methods* in the program:

Call	Methods Covered

b. (4 points) Write at most two calls that cover all *statements* in the program:

Call	Statements Covered (Numbers)

c. (6 points) Write at most three calls that cover all possible reachable *branches* in the program.

Call	Branches Covered (Letters)

d. (3 points) Which branch in the program cannot be covered? Explain briefly.

e. (4 points) Write a function `f` for which full *statement* coverage is impossible, no matter how the function is called. Please keep your answer short, and briefly explain which statement cannot be covered and why.

Question 3. Design Patterns and Testing (25 points). In this problem, you will write code to help verify simulated moves in a game of tic-tac-toe. This game is played by two players, X and O, on a 3×3 grid. Starting with X, the players take turns entering their letter into an unoccupied cell. The game ends when one player has established a row, column, or diagonal all of their letter—in which case that player wins—or the grid is full yet no player has won—in which case it is a draw. For example, here is a complete game (moving from left to right) in which O wins:

$$\begin{array}{c|c|c} \mathrm{X} & & \\ \hline & & \\ \hline & & \end{array} \rightarrow \begin{array}{c|c|c} \mathrm{X} & & \\ \hline & \mathrm{O} & \\ \hline & & \end{array} \rightarrow \begin{array}{c|c|c} \mathrm{X} & \mathrm{X} & \\ \hline & \mathrm{O} & \\ \hline & & \end{array} \rightarrow \begin{array}{c|c|c} \mathrm{X} & \mathrm{X} & \mathrm{O} \\ \hline & \mathrm{O} & \\ \hline & & \end{array} \rightarrow \begin{array}{c|c|c} \mathrm{X} & \mathrm{X} & \mathrm{O} \\ \hline & \mathrm{O} & \\ \hline \mathrm{X} & & \end{array} \rightarrow$$

$$\begin{array}{c|c|c} \mathrm{X} & \mathrm{X} & \mathrm{O} \\ \hline \mathrm{O} & \mathrm{O} & \\ \hline \mathrm{X} & & \end{array} \rightarrow \begin{array}{c|c|c} \mathrm{X} & \mathrm{X} & \mathrm{O} \\ \hline \mathrm{O} & \mathrm{O} & \\ \hline \mathrm{X} & & \mathrm{X} \end{array} \rightarrow \begin{array}{c|c|c} \mathrm{X} & \mathrm{X} & \mathrm{O} \\ \hline \mathrm{O} & \mathrm{O} & \mathrm{O} \\ \hline \mathrm{X} & & \mathrm{X} \end{array}$$

a. (6 points) Without using the enum keyword, implement class Player as a (typesafe) enumeration with exactly two instances, Player.X and Player.O. Make sure it is impossible to create any other instances of Player. Also include an instance method String name() that returns the name of the player (X or O).

b. (9 points) Below is code for Board, to store the tic-tac-toe grid, and MoveEvent(p, r, c), which indicates that player p marked board[r][c] with their letter. We've also given you two convenience methods, Board#full and Board#winner, for checking whether the board is full or has a winner.

```
public class Board {
  public Player[][] board = new Player[3][3]; // null for empty cell
  public boolean full() { ... } // Returns true if and only if all cells in the board are non-null
  public Player winner() { ... } // Returns the winner of the board, or null if there is no winner
}

public class MoveEvent {
  public Player p; public int row, col;
  public MoveEvent(Player p, int row, int col) { this.p = p; this.row = row; this.col = col; }
}
```

Write a method void verify(List<MoveEvent> moves) that runs, in order, the moves in moves, starting from an empty board. The method should return normally if the following rules are satisfied, and otherwise it should raise an exception:

- X goes first.
- The players alternate moves.
- Players may only mark an empty cell.
- Game play ends when either the board is full, or when there is a winner.

c. (10 points) For testing purposes, we would like to make it easy to create a List<MoveEvent>. Implement a fluent API that provides an interface for creating lists using code like the following:

```
List<MoveEvent> ml = X.takes(0).O().takes(4).X().takes(7).toMoves();
```

which is equivalent to the following code

```
ml = new LinkedList<MoveEvent>();
ml.add(new MoveEvent(Player.X, 0, 0));
ml.add(new MoveEvent(Player.O, 1, 1));
ml.add(new MoveEvent(Player.X, 2, 1));
```

Here the argument to takes is a cell position using the following numbering scheme:

0	1	2
3	4	5
6	7	8

You interface must work for any valid sequence of moves. You can assume all sequences start with a move by X. **You should not check that thse API is used correctly or that the rules are followed.** For example, you don't need to reject calls like X.takes(3).X().takes(2).O().O().takes(1).takes(3)—your API may behave arbitrarily for such invalid API usage. *Hint: Java integer division / always rounds down (it does a "floor" of the division result), and the Java modulo operator is %.*

Question 4. Concurrency (20 points). In this question, you will implement a *thread pool*, which is a fixed-size collection of worker threads that execute tasks. Thread pools let you create many tasks to be done concurrently without overwhelming the system with too many threads.

Implement a class ThreadPool with the following constructor and methods:

- ThreadPool(int n) - Create a thread pool with n worker threads and start those threads running.
- int size() - Return the number of worker threads in the thread pool.
- execute(Runnable r) - Submit a task to the thread pool to be done by the next available worker thread. This method just adds the task to a list (without waiting for an available worker). Meanwhile, the started worker threads will wait for tasks to be added to the list. The first worker to consume the task off the list removes it from the list and runs it. After a worker thread is finished with a task, the worker goes back to waiting for another task to arrive. *Hint: Make sure your worker threads do not hold a lock while they are running their tasks! Otherwise there's not much point to the thread pool.*
- int getActiveCount() - Return the number of worker threads that are actively doing work (the remaining threads are idle, waiting for work to be submitted). *Hint: Increment the count of active threads before calling run and decrement it after.*
- int getPendingCount() - Return the number of tasks that are in the list, waiting for a worker thread to execute the task.
- void shutdown() - Interrupt all worker threads. *Hint: Don't worry about whether this is truly sufficient to stop the threads.*

Question 5. Design Patterns and Concurrency (15 points). In this problem we will explore the idea of being easily able to kill all the threads created in a program.

a. (7 points) Implement a class TThread (short for *tracked thread*) with the following constructor and methods. You must use delegation in your solution rather than inheritance.

- TThread(Runnable r) creates a new tracked thread and keeps track of the thread in some global list.
- void TThread#start() starts a separate Thread, called the *underlying thread*, that executes the run method from the constructor argument.
- void TThread#join() blocks until the underlying thread has terminated.
- static void TThread.stopAll() calls Thread#stop on all underlying threads of all TThreads that have been previously constructed (using the global list of TThreads) and resets the global list of tracked threads back to the empty list.

b. (4 points) Assume we extend TThread to include the other methods of Thread. What design pattern that we discussed in class is TThread an example of? Explain briefly. It's okay if TThread doesn't match the pattern exactly as long as it's close.

c. (4 points) Unfortunately, the Thread#stop method is deprecated in Java because, as the documentation says, *"This method is inherently unsafe."* Briefly explain why it is unsafe.