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.

Answer: When invoking a method `o.m(...)`, Java looks at the run-time type of `o`, finds it method `m` (searching up the inheritance hierarchy until it is found), and then calls that method.

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

Answer: Refactoring means changing code without changing its behavior. Refactoring is useful because it can make future code changes, such as adding features, much easier. Using refactoring also means a software system's design need not be fixed up front, but it can evolve over time. Having a good test suite is important because it helps ensure that the refactoring truly does not change behavior.

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

Answer: The model is a database that stores the state of the system. The controller receives requests from the user and interacts with the model to carry them out. The view displays the result of a controller action to the user.

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.

Answer: Implementing support for future needs that do not occur increases complexity, which makes it harder to debug, maintain, and evolve code. It also incurs unnecessary cost.

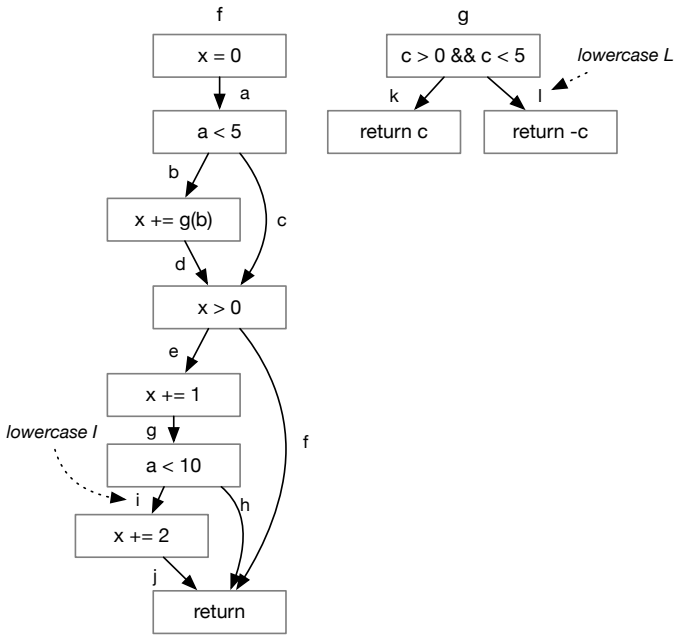
e. (4 points) List two benefits of simplifying test cases.

Answer:

- Helps restrict the hypothesis space for root causes
- Makes tests faster to run, because they're shorter
- Makes tests more orthogonal to each other

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
f(1,0)	f, g

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

Call	Statements Covered (Numbers)
f(1,1)	1, 2, 3, 8, 9, 4, 5, 6, 7
f(1,0)	1, 2, 3, 8, 10, 11, 4

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

Call	Branches Covered (Letters)
f(1,1)	a,b,k,d,e,g,i,j
f(1,0)	a,b,l,d,f
f(6,0)	a,c,f

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

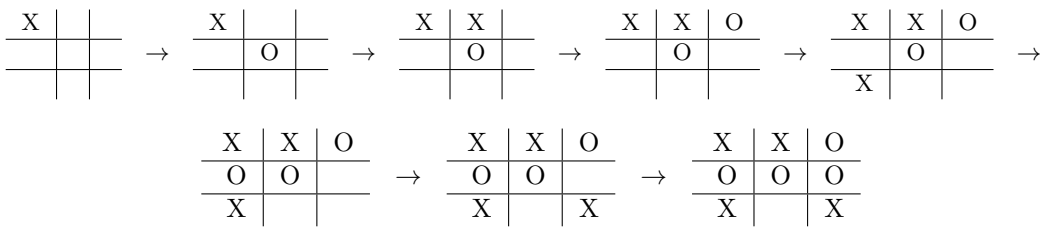
Answer: It is not possible to cover h, because it would require $\mathsf{a} \geq 10$, in which case x would always be zero, contradicting the condition $\mathsf{x} > 0$, which would also need to be true.

e. (4 points) Write a method 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.

Answer: In the following program, the starred statement can never be covered because the conditional guard is always false.

```
void f(void) {
  if (false) {
    int x = 3; /* * */
} }
```

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:



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).

Answer:

```
public class Player {
  private String name;
  private Player(String name) { this.name = name; }
  static Player X = new Player("X");
  static Player O = new Player("O");
  public String name() { return name; }
}
```

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.

Answer:

```
public static void verify(List<MoveEvent> moves) {
  Board b = new Board();
  Player last = null;
  for (Move m : moves) {
    if (b[row][col] != null) { throw new RuntimeException("Move in filled cell"); }
    if (last == null && p == Player.O) { throw new RuntimeException("O shouldn't go first"); }
    if (last == p) { throw new RuntimeException("Moves don't alternate"); }
    if (b.winner() != null) { throw new RuntimeException("Move after game ended"); }
    b[row][col] = p;
    b.last = p;
  }
  if (b.winner() != null && !b.full()) { throw new RuntimeException("Game ended too soon"); }
}
```

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 %.*

Answer:

```
public class X {
  public static Seq takes(int n) { return new Seq(n); }
}
public class Seq {
  private List<MoveEvent> l = new LinkedList<>();
  private Player p;
  public Seq(int n) { addTakes(Player.X, n); }
  public Seq X() { p = Player.X; return this; }
  public Seq O() { p = Player.O; return this; }
  public Seq takes(int n) { addTakes(p, n); return this; }
  public List<MoveEvent> toMoves() { return l; }
  private void addTakes(Player p, int n) {
      l.add(new MoveEvent(p, n/3, n%3));
  }
}
```

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.*

Answer:

```
public class ThreadPool {
  private List<Runnable> tasks = new LinkedList<>();
  private List<Thread> workers = new LinkedList<>();
  private int active;
  public ThreadPool(int n) {
    for (int i=0; i<n; i++) { workers.add(new Thread(new Worker())) }
    for (Thread t : workers) { t.start(); }
  }
  public int size() { return workers.size(); }
  public synchronized void execute(Runnable r) {
    tasks.add(r); notifyAll();
  }
  syhchronized public int getActiveCount() { return active; }
  syhchronized public int getPendingCount() { return tasks.size(); }
  public void shutdown() { for (Thread t : workers) { t.interrupt(); } }

  public class Worker implements Runnable {
    public void run() throws InterruptedException {
      while (!Thread.interrupted()) {
        Runnable r;
        synchronized(ThreadPool.this) {
          // synchronizes on outer class this, which is a trick you haven't seen, but
          // there are many alternative approaches that also work
          while (tasks.isEmpty()) { ThreadPool.this.wait(); }
          r = tasks.remove(0);
          active++;
        }
        r.run(); // don't do this in the synchronized block!
        synchronized(ThreadPool.this) { active--; }
        }
      }
    }
  }
}
```

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.

Answer:

```
public class TThread {
  private static List<TThread> tts = new LinkedList<>();
  private Thread t;
  public TThread(Runnable r) {
    tts.add(this);
    t = new Thread(r);
  }
  public void start() {
    t.start();
  }
  public void join() throws InterruptedException {
    t.join(); // Okay not to deal with InterruptedException ...
  }
  public void stopAll() {
    for (TThread tt : tts) {
      tt.t.stop();
    }
    tts = new LinkedList<>();
  }
}
```

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.

> **Answer:** TThread is an example of the *decorator* pattern, which wraps another class and adds some functionality. However, note that in class, we described decorators as implementing the same interface—which TThread does except there's no implements clause and it is not a subclass of Thread y design. It would also be reasonable to say that it's an example of an adapter, proxy, or wrapper.

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.

> **Answer:** If a thread is in the middle of accessing a shared resource and it is killed, then that resource might be left in an inconsistent state. For example, if the thread is in the middle of updating a complex data structure, it might leave the data structure such that its invariants are violated.