Name:

Final

CS 121
Software Engineering
Fall 2022

December 15, 2022

Instructions

This exam contains 15 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		20
2		25
3		26
4		17
5		12
Total		100

Question 1. Short Answer (20 points).

a. (4 points) Briefly explain why the autograder for a project may fail to compile against your code if you add a **throws** clause to a method.

> **Answer:** A **throws** clause lists the *checked exceptions* thrown by a method. Any caller of a method with such a clause must either catch the exceptions or itself have a **throws** clause listing any uncaught checked exceptions. So, if the autograder calls one of your methods and it's not expecting a **throws** clause, it won't compile because it will do neither of those things.

b. (4 points) Briefly explain the role of the *model* in the model–view–controller architecture.

> **Answer:** The model is the data storage (often a databaes) that the view and controller both access. Separating out the model lets multiple views and controllers access the model concurrently while always seeing the latest version of the shared state.

c. (4 points) Briefly explain what *statement coverage* and *branch coverage* (also known as *condition coverage*) are in testing. Does full statement coverage imply full branch coverage? Explain briefly.

Answer: Statement coverage measures how many program statements are executed when a test suite is executed. Branch coverage measures how many branches—i.e., edges in the control-flow graph of a program—are executed when a test suite is executed. No, a test suite can have full statement coverage without full branch coverage. For an example, see the testing slides.

d. (4 points) In his *No Silver Bullet* paper, Brooks discusses the *accidental* difficulties of software engineering. Briefly explain what Brooks means by accidental difficulties and list two solutions that have been developed for accidental difficulties of software engineering.

Answer: According to Brooks, accidental difficulties are those that don't have to do with the core of the problem being solved, but rather from the tools available to solve it. Example solutions include high-level languages, time sharing, unified programming environments, high-level language advances, object-oriented programming, expert systems, machine learning, program synthesis, program verification, and more.

e. (4 points) Briefly explain what *delta debugging* is and why it may be useful.

Answer: Delta debugging is a method for simplifying a test input successively removing unnecessary circumstances from the input, at finer and finer granularities. It's useful because when trying to debug a failing test case, a smaller test has fewer things happening which makes it easier to find the source of the problem.

Question 2. Verifying Logs (25 points). In this question, you will implement methods to help verify that logs of events are correct. An *event* is an instance of the following class:

```
class Move {
    // Represents a move of train from station start to station end
    String train, start, end;
    Move(String train, String start, String end) {
        this.train = train; this.start = start; this.end = end;
}   }
```

and a *log* is a List<Move>. You will write several methods that implement the following interface:

```
interface Prop {
    // Returns true if the property holds for entry i of the log
    // Returns false otherwise or if i ≥ log.size()
    boolean check(List<Move> log, int i);
}
```

Below, we say "Write a Prop C" to mean "Write a class C that implements Prop," and we say "p holds at i" for some Prop p to mean "calling p.check(log, i) returns **true**."

a. (5 points) Write a Prop TuftsStart such that TuftsStart#check(log, i) is true if and only if the log entry at position i is a move that starts at the station "Tufts".

Answer:

```
class TuftsStart implements Prop {
    boolean check(List<Event> log, int i) {
        return (i < log.size() && (log.get(i).start.equals("Tufts")));
    }
}
```

b. (5 points) Write a Prop And with a constructor And(Prop p1, Prop p2) such that And#check(log, i) is true if and only if both p1 and p2 hold at position i.

Answer:

```
class And implements Prop {
    Prop p1, p2;
    And(Prop p1, Prop p2) { this.p1 = p1; this.p2 = p2; }
    boolean check(List<Event> log, int i) {
            return p1.check(log, i) && p2.check(log, i);
    }
}
```

c. (5 points) Write a Prop Next with a constructor Next(Prop p) such that Next#check(log, i) returns true if and only if p holds at position i+1.

Answer:

```
class Next implements Prop {
    Prop p;
    Next(Prop p) { this.p = p; }
    boolean check(List<Event> log, int i) {
            return p.check(log, i+1);
    }
}
```

d. (5 points) Write a Prop Eventually with a constructor Eventually(Prop p) such that Eventually#check(log, i) returns true if and only if p holds at some position $j \geq i$.

Answer:

```
class Eventually implements Prop {
    Prop p;
    Eventually(Prop p) { this.p = p; }
    boolean check(List<Event> log, int i) {
        for (j = i; j < log.size(); j++) {
            if (p.check(log, j)) { return true; }
        }
        return false;
    }
}
```

e. (5 points) Using **new** to create instances of the classes from parts a–d above, write code that creates a Prop object that checks that at position i, the log entry is a move that starts at Tufts, and there exists some position j > i (notice greater than, not greater than or equal to) at which the log entry is also a move that starts at Tufts.

Answer:

```
new And(new TuftsStart(), new Next(new Eventually(new TuftsStart())));
```

Question 3. Reflection and Testing (26 points). In this problem, you will use reflection to build a very simple testing framework. Your framework will use the following two annotations, which will be explained in the questions below.

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Test { }
```

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Expected {
    public Class<?> expected() default null;
}
```

Here are some useful methods from the Reflection API; ignore checked exceptions thrown by these methods.

- `Class<?> Class.forName(String s)` - return the class named `s`.
- `Constructor<?> Class#getConstructor(Class<?>... parameterTypes)` - return the constructor with the given parameter types.
- `Method[] Class#getMethods()` - return the methods of the class.
- **boolean** `Class#isInstance(Object o)` - return true if and only if `o` is an instance of the class.
- `Object Constructor#newInstance(Object... args)` - create a new instance using the constructor and the given constructor arguments `args`.
- `Annotation Method#getAnnotation(Class<?> annotationClass)` - return the annotation of `annotationClass` if present on the method, otherwise return **null**.
- `Object Method#invoke(Object obj, Object... args)` - call the method with `obj` as the receiver and arguments `args`.
- **boolean** `Method#isAnnotationPresent(Class<?> annotationClass)` - return true if and only if an annotation from `annotationClass` is present on the method.

a. (5 points) Write a method `List<Methods> tests(Class<?> c)` that returns all the methods of `c` that have the `@Test` annotation.

Answer:

```
List<Method> tests(Class<?> c) {
    List<Method> ms = new LinkedList<>();
    for (Method m : c.methods()) {
        if (m.isAnnotationPresent(Test.class)) { ms.add(m); }
    }
    return ms;
}
```

b. (5 points) Write a method Class<?> expected(Method m) that takes a method m and either returns **null** if it does not have an @Expected annotation; or, if it does have an @Expected annotation, returns the value of the expected field of that annotation.

Answer:

```
Class<?> expected(Method m) {
    Expected e = (Expected) m.getAnnotation(Expected.class);
    if (e == null) return e;
    return e.expected();
}
```

c. (16 points) Write a method List<Method> runTests(String clazz) that creates an instance of class clazz and invokes all its @Test methods, which you can assume take no arguments and return **void**. It returns a list containing any @Test methods that failed, i.e., that raised an exception when run. However, if any @Test method is also annotated @Expected(expected = exceptionClass), then it is considered to pass if it raises an exception that is an instance of exceptionClass, and to fail otherwise. The @Test methods can be called in any order. Your code may call the methods written in parts a and b above.

Answer:

```
List<Method> runTests(String clazz) {
    Class<?> c = Class.forName(clazz);
    Constructor<?> cons = c.getConstructor();
    Object o = c.newInstance();
    List<Method> failures = new LinkedList<>();
    for (Method m : tests(c)) {
        Class<?> exn = expected(m);
            try {
                m.invoke(o);
                if (exn != null) {
                    failures .add(m);
                }
            }
            catch (Exception e) {
                    if (exn == null || !exn. isInstance (e)) {
                        failures .add(m);
                }
        }
    }
    return m;
}
```

Question 4. Scheduling (17 points). In this question you will develop and use a class Scheduler that can force a multi-threaded Java program to follow a particular schedule. The class should work as follows:

- Scheduler has a constructor Scheduler(List <Thread> sched), which creates a scheduler that runs threads in the order given by sched.
- Scheduler has a method **void** yield (). Clients of a Scheduler must include code to call yield at the start of their run method, at the end of their run method, and any time they are willing to let another thread run.
- Each of the threads call yield , but initially the only thread that continues (i.e., for which yield returns instead of blocks) is the one listed as the first element of sched. The rest of the threads block. That first thread runs for a while and eventually calls yield , at which point it blocks and the second thread in sched runs. That runs until it calls yield , at which point it blocks and the third thread in sched runs, and so on.
- We won't specify what happens when the end of sched is reached; handle it however you like.
- You'll want to use locks and wait/ notifyAll or await/ signalAll for this problem. You can use Thread Thread.currentThread() to get the current thread.
- This class sounds complicated but doesn't need much code.

As an example, if Scheduler s = **new** Scheduler(List.of(t1, t1, t2, t1, t1, t2, t2, t2)) is shared between threads t1 and t2, the code below will print A B D C E F and both run methods will exit.

```
// Thread t1
void run() {
  s.yield();
  System.out.println("A");
  s.yield();
  System.out.println("B");
  s.yield();
  System.out.println("C");
  s.yield();
}
```

```
// Thread t2
void run() {
  s.yield();
  System.out.println("D");
  s.yield();
  System.out.println("E");
  s.yield();
  System.out.println("F");
  s.yield();
}
```

a. (8 points) Write the class Scheduler as described above.

Answer:

```
class Scheduler {
    List<Thread> sched;

    Scheduler( List<Thread> sched) { this.sched = sched; }

    synchronized void yield() {
        notifyAll ();
        while (sched.isEmpty() || sched.get(0) != Thread.currentThread()) {
            wait();
        }
        sched.remove(0);
    }
}

// Note most correct solutions instead tracked an index i into sched and incremented it
// rather than remove elements from sched.
```

b. (9 points) In class, we showed how the following program has a data race, and therefore may sometimes print 1 instead of 2. Modify the code below to use Scheduler to force the threads to execute with a certain interleaving so the code *always* prints 1. Mark up the code below directly with your changes.

```
class Racer extends Thread {

    static int cnt;

    void run() {

        int y = cnt;

        cnt = y + 1;

    }

}
class Main {

    static void main(String[] args) {

        Thread t1 = new Racer();

        Thread t2 = new Racer();

        t1.start(); t2.start();

        try { t1.join(); t2.join() } catch (InterruptedException e) { }

        System.out.printlnt(Racer.cnt);

    }
}
```

Answer:

```
class Racer extends Thread {
    static int cnt;
    Scheduler s;
    void sched(Scheduler s) { this.s = s; }
    void run() {
        s.yield();
        int y = cnt;
        s.yield();
        cnt = y + 1;
        s.yield();
    }
}
class Main {
    public static void main(String[] args) {
        Thread t1 = new Racer();
        Thread t2 = new Racer();
        List<Thread> l = List.of(t1, t2, t2, t2, t1, t1);
        Scheduler s = new Scheduler(l);
        t1.sched(s); t2.sched(s);
        t1.start(); t2.start();
        try { t1.join(); t2.join() } catch (InterruptedException e) { }
        System.out.println(Racer.cnt);
    }
}
```

Question 5. Concurrency (12 points). For each of the following code snippets and scenarios, briefly explain any potential concurrency issues, or briefly explain why there are no such issues.

a. (3 points) Suppose the following class is used with multiple threads that call produce and consume:

```
class Buffer {
    Object buf;
    synchronized void produce(Object o) {
        if (buf != null) { wait(); }
        buf = o;
        notifyAll();
    }
    synchronized Object consume() {
        if (buf == null) { wait(); }
        Object tmp = buf; buf = null;
        notifyAll();
        return tmp;
}   }
```

Answer: This code may not work because the wait calls are not in a loop and signaling is done with notifyAll. For example, a consumer may be woken up from a wait but the buffer may have become empty in the meantime due to another consumer running.

b. (3 points) Suppose the fileAccess method is called from multiple threads.

```
Lock l = new ReentrantLock();
void fileAccess() throws IOException {
    l.lock();
    // Files.readAllLines(Path path) throws IOException reads all the lines of a file
    List<String> lines = Files.readAllLines(Path.of("foo.txt"));
    l.unlock();
}
```

Answer: This code may fail to unlock if an exception is raised by readAllLines, which could lead to a deadlock.

c. (3 points) Suppose an instance of C is created and its inc and dec methods are called by multiple threads.

```
class C {
    static int cnt;
    void inc() { synchronized(this) { cnt++; } }
    static synchronized void dec() { cnt--; }
}
```

Answer: This code has a data race on cnt because it is thread shared and writable, but two different locks are used to protect it (one named by the instance of C, the other named by C.class).

d. (3 points) Suppose an instance of D is created and its inc method is called by multiple threads.

```
class D {
    int cnt;
    Lock l = new ReentrantLock();
    void inc() {
        int y;
        l.lock(); y = cnt; l.unlock();
        l.lock(); cnt = y+1; l.unlock();
}   }
```

Answer: This program does not have a data race—all accesses to the shared variable cnt are protected by the same lock—but it has an insufficiently larger critical/atomic section. The lock should not be released between the read of cnt and increment of cnt.