Name:

Final Exam

COMP 180

Software Engineering

Spring 2020

May 1, 2020

Instructions

This exam contains 13 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		25
2		20
3		15
4		25
5		15
Total		100

Question 1. Short Answer (20 points).

a. (5 points) In at most a few sentences, explain the difference between an *instance method* and a *class method.*

> **Answer:** In an instance method, `this` is bound to the receiver object of a method called, e.g., in the execution of `m` in the call `o.m(x)`, `this` is bound to `o`. A class method is one that can be invoked via the class name, and `this` is not available in the method body.

b. (5 points) In at most a few sentences, explain the difference between *black box testing* and *glass box (a.k.a. clear box a.k.a. white box) testing.*

> **Answer:** Black box testing test the program based on its specification, independently (in theory) of its implementation. Glass box testing looks at the implementation to determine test cases.

c. (5 points) In at most a few sentences, explain what *refactoring* software means. Also briefly explain why refactoring is useful.

> **Answer:** Refactoring means changing a program's code without changing the program's behavior. Refactoring is useful to help evolve the design of software over time so that future changes are easier to make. A key feature of refactoring is that, before and after a refactoring, the same set of test cases pass.

d. (5 points) In the KLEE symbolic executor, when execution reaches a branch, KLEE may potentially fork execution to explore both branches. Suppose we were to apply KLEE to Java. In addition to `if` and `switch` statements, list three more kinds of Java expressions and/or statements at which KLEE may possibly fork execution (i.e., list places in Java that conditionally branch at runtime).

> **Answer:** Loops (`for`, `while`); dynamic dispatch (`o.m(...)`); field access because an exception may be raised for a null object (`o.f`); `catch` because the exception might or might not match; operations that do comparisons and return booleans `x<y`, `x>y`, etc.; checked downcasts that may or may not succeed (`(T) e`); the `instanceof` operation.

e. (5 points) In at most a few sentences, define *confidentiality* and *integrity*.

Answer: Confidentiality means sensitive data does not leak from the program to an adversary. Integrity means that an adversary cannot modify the sensitive data of a program.

f. (0 points) What is the literal English translation of *theobroma cacao*?

Answer: Food of the Gods

Question 2. Design Patterns (20 points). Consider the Regex interface and implementations from Project 5:

```
interface Regex { }
class RChar implements Regex { public final char c; RChar(char c) { ... } }
class RSeq implements Regex { public final Regex left, right; RSeq(Regex left, Regex right) { ... } }
class ROr implements Regex { public final Regex left, right; ROr(Regex left, Regex right) { ... } }
class RStar implements Regex { public final Regex re; RStar(Regex re) { ... } }
```

a. (5 points) Below is the standard visitor interface. Implement the accept method for classes RChar, RSeq, ROr, and RStar. Your code should do a postorder traversal, in which the children are visited, left-to-right, before the parent.

```
interface Visitor { void visit(RChar re); void visit(RSeq re); void visit(ROr re); void visit(RStar re); }
interface Regex { void accept(Visitor v); }

// Here's a start to the code you need to write
// Be sure to write all four classes!
class RChar implements Regex { void accept(Visitor v) {
```

Answer:

```
class RChar implements Regex { void accept(Visitor v) { v.visit(this); } }
class RSeq implements Regex { void accept(Visitor v) { left.accept(v); right.accept(v); v.visit(this); }
class ROr implements Regex { void accept(Visitor v) { left.accept(v); right.accept(v); v.visit(this); }
class RStar implements Regex { void accept(Visitor v) { re.accept(v); v.visit(this); }
```

b. (5 points) Write a visitor StarCount such that the sequence sc = new StarCount(); re.accept(sc); int x = sc.count; sets x to the number of RStar's in re. For example, if re were new ROr(new RStar(new RChar('a')), new RStar(new RChar('b'))), then x would be 2.

```
class StarCount implements Visitor {
```

Answer:

```
class StarCount implements Visitor {
  int count;
  void visit(RChar re) { } void visit(RSeq re) { } void visit(ROr re) { }
  void visit(RStar re) { count++; }
}
```

c. (5 points) Write a visitor Example such that the sequence ex = new Example(); re.accept(ex); String x = ex.str; returns one example string matched by re. For example, if re were new ROr(new RChar('a'), new RChar('b')), then x could be *either* a or b. *Hint: If you need, you can add field(s) to the classes that implement Regex. You can't just add a field to the interface because such fields are public, static, and final. You might find the following API methods useful:*

```
class Character { public static String toString(char c); }
class String { public String concat(String str); }
```

```
class Example implements Visitor {
```

Answer:

```
class RChar { String str; } class RSeq { String str; }
class ROr { String str; } class RStar { String str; }
class Example implements Visitor {
  String str = "";
  void visit (RChar re) { re.str = Character.toString(re.c); }
  void visit (RSeq re) { re.str = re.left.str.concat(re.right.str); }
  void visit (ROr re) { re.str = re.left.str; }
  void visit (RStar re) { re.str = ""; }
}
```

d. (5 points) The Visitor interface for the first three parts of this problem always performs a postorder traversal. Propose an alternative design and implementation for the Visitor interface and the accept methods so that Visitors can specify whether to do a pre- or postorder traversal (preorder means visiting the node before the children). Describe your design concisely and precisely.

Answer: There are many possible answers. One idea is to add a method or a field to the visitor specifying the order

```
interface Visitor { ... boolean preorder(); /* true for preorder, false for postorder */ }
```

Then the preorder() flag can be tested inside the accept methods, e.g.,

```
class RSeq implements Regex {
  void accept(Visitor v) {
    if (v.preorder()) { v.visit(this); }
    left.accept(v);
    right.accept(v);
    if (!v.preorder()) { v.visit(this); }
  }
}
```

Question 3. Testing (15 points). In this question, you will implement an alternative design for assertions in which assertions are objects that implement the following interface:

```
interface Checker<T> {
  boolean check(T x); // returns true if x passes the check, false otherwise
}
```

a. (10 points) Implement five classes, Null, Equals, Not, All, and Some that implement Checker<T>, with the following constructors:

Constructor	check(x) Behavior
Null()	Returns true if and only if x is null
Equals(Object y)	Returns true if and only if y.equals(x)
Not(Checker<T> c)	Returns true if c.check(x) returns false, and vice-versa
Some(Checker<T>[] c)	Returns true if at least one c[i].check(x)'s returns true, and false otherwise

For example, if c = new Some(new Checker<String>[] { new Null(), new Equals("COMP"); }), then c.check("COMP") == c.check(null) == true and c.check("MATH") == false.

Answer:

```
class Null<T> implements Checker<T> {
  boolean check(T x) { return x == null; }
}
class Equals<T> implements Checker<T> {
  T y;
  Equals(Object y) { this.y = y; }
  boolean check(T x) { return y.equals(x); }
}
class Not<T> implements Checker<T> {
  Checker<T> c;
  Not(Checker<T> c) { this.c = c; }
  boolean check(T x) { return !c.check(x); }
}
 class Some<T> implements Checker<T> {
  Checker<T>[] cs;
  Some(Checker<T>[] cs) { this.cs = cs; }
  boolean check(T x) {
    for (Checker<T> c : cs) { if (c.check(x)) { return true; } }
    return false;
  }
}
```

Note: I saw a lot of solutions that had code like if (something) { return true; } else { return false; }. This can be replaced simply by return(something).

b. (5 points) In Project 3, you implemented a *fluent* interface for assertions that looked like the following:

```
String s = ...;
Assertion.assertThat(s).isNotNull().startsWith("COMP");
```

From the perspective of a developer writing test cases, compare and contrast the assertion style from Project 3 with the assertion style from part a of this problem. List some advantages and disadvantages of each style.

> **Answer:** Many answers are possible! The most obvious difference is that in project 3, an assertion is a chain of calls whose assertions are conjoined. In fact, it is rather difficult in the syntax of p3 to implement `not` and `or` in a simple way, and it's essentially impossible to introduce parentheses for grouping without extending the notation. On the other hand, the notation from project 3 is much more compact and doesn't scatter `news` all over the place, though that could be fixed by introducing some methods to construct the checkers.

Question 4. Software Architecture (25 points). Recall the *pipe and filter* software architecture, in which *filters* transform input streams to output streams, and *pipes* connect up filters. In this problem, you will implement a number of methods for working with pipes and filters.

In this problem, a filter is an object that implements the following interface:

```
interface Filter<T> {
  T next(); // returns null if no next element
}
```

In words: A Filter is an object f such that calling f.next(); returns the next element from the filter. Notice that a filter is parameterized by its output type.

For example, here is a filter that consumes integers from its input and returns their squares:

```
class Square {
   Filter<Integer> g;
  Square( Filter<Integer> g) { this.g = g; }
  Integer next() {
    Integer i = g.next();
    if (i == null) { return null; }
    return i*i;
}
```

For example, if g is a filter such that calling g.next() successively returns 0, 1, 2, 3, null, null, . . ., then if we set f = new Square(g), then calling f.next() will successively return 0, 1, 4, 9, null, null, ...

For some of the problems below, you'll need to use the following utility class:

```
class Pair<K,V>
  Pair(K k, V v) - construct a pair of key k and value v
  getKey() - return the key
  getValue() - return the value
```

a. (4 points) Write a filter Ints such that f = new Ints(n) produces a filter f such that successively calling f.next() returns 0, 1, . . ., n, null, null, ...

```
class Ints implements Filter<Integer> {
```

Answer:

```
  int next, last;
  Ints(int last) { this.last = last; }
  Integer next() {
    if (i > last) { return null; }
    int tmp = next; next++; return tmp;
} }
```

b. (6 points) Suppose f and g are filters such that f.next() and g.next() return f0, f1, ..., and g0, g1, ..., respectively. Write a filter Mix such that m = new Mix(f, g) is a filter such that m.next() returns f0, g0, f1, g1, ..., alternating between f and g and starting with f. As soon as one of f.next() or g.next() returns null, then m.next() should return null from then on. *Hint: Don't worry about getting the constructor type signature exactly right.*

```
class Mix implements Filter<Object> {
```

Answer:

```
Filter <? extends Object> left, right ;
boolean which; // false = left , true = right
Object last = new Object();
Mix( Filter <? extends Object> left, <? extends Object> right) {
   this . left = left ; this . right = right;
}
Object next() {
   if ( last == null) { return null ; }
   if (which) { last = right . next (); } else { last = left . next (); }
   which = !which;
   return last ;
}
```

c. (5 points) Write a filter Zip such that, if f is a filter such that f.next() returns f0, f1, ..., and g is a filter such that g.next() returns g0, g1, ..., then if z = new Zip(f, g), then z.next() returns new Pair(f0, g0), new Pair(f1, g1), ... If either f.next() or g.next() returns null, then z.next() should return null (not new Pair(null, null)!).

```
class Zip implements Filter<Pair<T, U>> {
```

Answer:

```
Filter <T> f; Filter<U> g;
Zip( Filter <T> f, Filter<U> g) { this.f = f; this .g = g; }
Pair<T, U> next() {
   T left = f.next (); U right = g.next ();
   if ( left == null || right == null) { return null ; }
   return new Pair( left , right );
} }
```

d. (10 points) Suppose f is a Filter<Pair<T, U>> such that f.next() returns (f0, g0), (f1, g1), ..., where (x, y) is shorthand for new Pair(x, y). Write a method split such that if p = split(f), then p is a Pair<Filter<T>, Filter<U>> that behaves as follows. Let k = p.getKey() and v = p.getValue(). Then k.next() returns f0, f1, ..., and v.next() returns g0, g1, ... The calls to k.next() and v.next() may be interleaved, and it should not affect the result. For example, k.next(); v.next(); k.next(); v.next(); would return f0, g0, f1, g1, and k.next(); k.next(); v.next(); v.next(); would return f0, f1, g0, g1. *Hint: You will also need to create at least one new class.*

```
<T, U> Pair<Filter<T>, Filter<U>> split(Filter<Pair<T, U>> f) {
```

Answer:

```
  return new Buffer<T,U>(f).split();
}

class Buffer<T, U> {
  List<> lefts = new LinkedList<T>();
  List<> rights = new LinkedList<U>();
  Filter <Pair<T, U>> f;
  Buffer( Filter <Pair<T,U>> f) { this.f = f; }
  void getNext() {
    Pair<T, U> p = f.next();
    if (p != null) { lefts.addLast(p.getKey()); rights.addLast(p.getValue()); }
  }
  T nextLeft() {
    if (lefts.size() == 0) { getNext(); }
    if (lefts.size() == 0) { return null; }
    return lefts.removeFirst();
  }
  T nextRight() {
    if (rights.size() == 0) { getNext(); }
    if (rights.size() == 0) { return null; }
    return rights.removeFirst();
  }
  Pair<Filter<T>, Filter<U>> split() {
    return new Pair(new Filter<T>() { T next() { return nextLeft(); } },
                    new Filter<U>() { U next() { return nextRight(); } });
  }
}
```

Question 5. Concurrency (15 points). A *future* is a computation that is run in a separate thread while the main thread continues its own work. At some time in the future, the main thread *gets* the result of the future, which either returns the future's result immediately, if the future was already finished, or it blocks until the future is finished and then returns. Implement the following generalization of futures. **You can use threads, locks, and condition variables (wait/notifyAll or await/signalAll). You may not use other parts of java.util.concurrent. Feel free to add comments to your code. We will give partial credit if the comments are right, even if the code is not**

```
// A callable is an object with a call method that returns a result.
// The callables are the computations that are run in separate threads.
interface Callable<V> { V call(); }

class Future<V> { // A Future is paramterized by the type it returns
  Future(Callable<V>[] cs);   // The constructor takes an array of n callables to run.

  // Calling start launches n threads, one for each callable passed to the constructor. Each
  // thread invokes the call methods of the callables.
  void start();

  // Some time after start() has been called, the code may call getFirst(), which has the following behavior:
  // * If none of the threads has finished, it blocks
  // * As soon as one thread has finished, it returns the value computed by that thread's callable.
  // * If more than one thread has finished or finishes at once, either thread's result may be returned.
  // * You don't need to worry about stopping the threads that haven't yet finished.
  // * Multiple calls to getFirst should always return the same value.
  V getFirst();
}
```

Answer:

```
class Future<V> {
  boolean[] done;
  V[] results;
  Task<V>[] tasks;
  class Task<V> extends Thread {
    int i;
    Callable<V> c;
    Task(int i, Callable<V> c) { this.i = i; this.c = c; }
    void run() {
      V v = c.call();
      synchronized(Future.this) { done[i] = true; results[i] = v; Future.this.notifyAll(); }
      // could also use a ReentrantLock!
  }
  Future(Callable<V>[] cs) {
    done = new boolean[cs.length];
    results = new V[cs.length];
    tasks = new Task<V>[cs.length];
    for (int i = 0; i < cs.length; i++) { tasks[i] = new Task(i, cs[i]); }
  }
  void start() { for (Task<V> t : tasks) { t.start(); } }
  V synchronized getFirst() {
    int j = -1;
    while (true) {
      for (int i = 0; i < done.length; i++} { if (done[i]) { j = i; break; } }
      if (j != -1) { break; }
      wait();
    }
    return results[j];
} }
```

Answer:

The above solution saves all the results from the different threads, which would be useful if there were other ways besides getFirst to look at the results. But since this particular interface only has getFirst in it, here's an alternative, simpler solution:

```
class Future<V> {
    boolean done;
    Callable <V>[] cs;
    V result ;

    Future( Callable <V>[] cs) {
        this.cs = cs;
    }

    class Caller extends Thread {

        Callable<V> callable;

        public Caller ( Callable<V> callable) {
            this. callable = callable ;
        }

        public void run() {
            V res = callable . call ();
            synchronized (Future.this) {
                if (!done) {
                    done = true;
                    result = res;
                    Future.this. notifyAll ();
    } } } }
    void start () {
        for ( Callable<V> callable : cs) {
            new Caller( callable ). start ();
    } }
    synchronized V getFirst() throws InterruptedException {
        if (done) {
            return result ;
        }
        this.wait ();
        return result ;
} }
```