Name:

Final

COMP 121
Software Engineering
Spring 2021

May 7, 2021

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		20
3		20
4		25
5		15
Total		100

Question 1. Short Answer (20 points).

a. (4 points) Briefly, explain what *delegation* is, in the context of design patterns.

Answer: Delegation is when one object `o` holds an instance of another object `p` and invokes `p`'s methods to carry out the corresponding operations of `o`. Delegation is a form of code reuse.

b. (4 points) In at most a few sentences, explain what *refactoring* is and why it might be useful.

Answer: Refactoring is the process of changing code without changing its behavior. The main reason to refactor is to improve the design of the code, which can make future changes easier.

c. (4 points) In class, we discussed using `ReentrantLock` for mutual exclusion. What does it mean that such a lock is *reentrant*? Explain briefly.

Answer: A reentrant lock is one that can be reacquired (without blocking) by a thread that already holds the lock. A reentrant lock is released when it is unlocked the same number of times it was locked by the thread that holds it.

d. (4 points) In *No Silver Bullet*, Brooks claims that high-level programming languages help ease some of the *accidental* difficulty of writing software. Describe two ways Java eases the accidental difficulty of using lower-level languages such as assembly or C.

Answer: There are lots of possible answers!

e. (4 points) At a high level, explain what *delta debugging* is for and why it might be useful.

Answer: Delta debugging reduces the size of test inputs, which can make it easier to diagnose a bug.

Question 2. Java Programming Grab Bag, with a Tiny Bit of Reflection (20 points).

a. (10 points) Recall the Graph interface from Project 1:

```
public interface Graph {
    boolean addNode(String n); // returns true if node was not previously in graph and false otherwise
    boolean addEdge(String n1, String n2); // throws exception if n1 or n2 was not previously added to graph
    boolean hasNode(String n);
     ...
}
class ListGraph implements Graph { ... }
```

Write a new class, GraphBuilder, that uses a fluent interface to construct a graph as in the following example:

```
Graph g = GraphBuilder.edge().from("a").to("b").edge().from("a").to("c");
// g has nodes a, b, and c, and edges from a to b from a to c
```

- The exact design of GraphBuilder is up to you, but the code above must work.
- You can write helper classes and methods if you like.
- Your code should enforce (either via run-time or, if you want to get fancy, compile-time checks) that edges are always added as a sequence of calls to edge, then from, then to. Raise any exception (or fail to compile) if a user tries to call those methods in a different order.

Answer:

```
public class GraphBuilder {
    public static GraphBuilderHelper edge() { return new GraphBuilderHelper(); }
}
public class GraphBuilderHelper extends ListGraph implements Graph {
  int state; // 0 = edge next, 1 = from next, 2 = to next
  String f;

  public GraphBuilderHelper edge() {
      if (state != 0) { throw new UnsupportedOperationException(); }
      state = 1;
      return this;
  }
  public GraphBuilderHelper from(String f) {
    if (state != 1) { throw new UnsupportedOperationException(); }
    this.f = f; state = 2; return this;
  }
  public GraphBuilderHelper to(String t) {
    if (state != 2) { throw new UnsupportedOperationException(); }
    addNode(f); addNode(t); addEdge(f, t); state = 0; return this;
} }
```

b. (8 points) In this question, you will use reflection to write a method Object[][] readBoard(List<String> lines) that like your solution to Project 2, converts a textual chess board layout to an 8×8 array representing the board. The input to your method lines is a List<String>, each String representing a line of the input file (you don't need to write the code to read the file). The format of each string is xn Class c where x is a column a–h, n is a row 1–8, Class is the name of a class, and c is a color, either b for black or w for white. For example, here is part of the standard layout of a chess board:

```
List<String> layout = new List.of("a1 Rook w", "a2 Pawn w", "a7 Pawn b", "a8 Rook b");
```

Implement your code below, following these rules:

- Your code should throw an exception (any exception) if one of the input strings in malformed.
- Your code can assume a constructor that takes a Color exists, and can throw any exception if not.
- Your method should return an 8×8 array, where a1 is at position [0][0], h1 is at position [7][0], and a8 is at position [0][7].

Some useful library methods are below, plus the definition of the Color class.

```
public class Class<T> {
  static Class<?> forName(String className);
  Constructor<T> getConstructor(Class<?>... parameterTypes);
}
public class Constructor<T> {
  T newInstance(Object ... initargs );
}
public class String {
  char charAt(int index);
  String [] split (String regex);
  int length ();
}

public enum Color { BLACK, WHITE };
```

Answer:

```
Object [][] readBoard(List<String> lines) {
  Object [][] b = new Object[8][8];

  for (String line : lines ) {
    String [] sp = line. split (" ");
    if (sp.length != 3) { throw new UnsupportedOperationException(); }
    if (sp [0]. length () != 2) { throw new UnsupportedOperationException(); }
    int row = sp[0].charAt(0) - 'a';
    int col = sp[0].charAt(1) - '1';
    if (row < 0 || row > 7 || col < 0 || col > 7) { throw new UnsupportedOperationException(); }
    Constructor<?> cons = Class.forName(sp[1]).getConstructor(Color. class );
    Color c;
    if (sp [2]. equals("b")) { c = Color.BLACK; }
    else if (sp [2]. equals("w")) { c = Color.WHITE; }
    else { throw new UnsupportedOperationException(); }
    b[row][col ] = cons.newInstance(c);
  }
  return b;
}
```

c. (2 points) List one advantage and one disadvantage of using reflection, as in part b above, to construct a board compared to the approach used in Project 2, where we registered factory objects in a hash map.

> **Answer:** Advantage: Increased flexibility since we can add a new chess piece type without needing to modify the registration code. Disadvantage: There is less checking, as the layout might refer to a class that's not intended to be a piece but has the correct constructor, or that is a piece but wasn't supposed to be available yet.

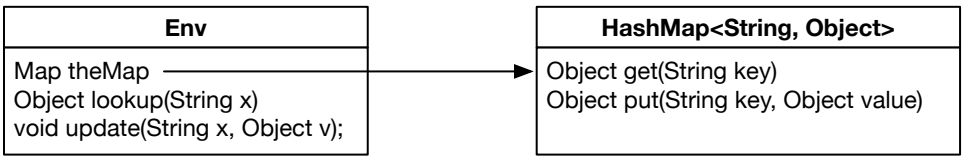
Question 3. Java and Reflection (20 points). In this problem, you will use reflection to build a simple Java interpreter—in Java! Your interpreter will handle a very small language: *expressions*, which can be integers, strings, or variables; and *statements*, which are either static field assignments x = cls.fld or calls x = r.m(args). A program is a list of statements. Here are the classes to represent programs in this language:

```
public interface JExpr { }                                                  /* expressions */
public class JInt implements JExpr { int i; }                               /* literal integer i */
public class JStr implements JExpr { String s; }                            /* literal string s */
public class JVar implements JExpr { String x; }                            /* variable x */

public interface JStmt { }                                                  /* statements */
public class JSFld implements JStmt { String x; String cls; String fld; }   /* x = (static) cls.fld */
public class JCall implements JStmt { String x; JExpr r; String m; List<JExpr> args; }  /* x = r.m(args) */

/* A program is a List<JStmt> */
```

In the questions below, you will implement a class Env to store the values of variables; JExpr#eval to evaluate expressions; JStmt#eval to evaluate statements (and which calls JExpr#eval); and interpret(List<JStmt>) to run a program (and which calls JStmt#eval). For example, here is how we would evaluate temp = java.lang.System.out; x = temp.println("Hello, world!"):

```
JStmt stmt1 = new JSFld("temp", "java.lang.System", "out");
JStmt stmt2 = new JCall("x", new JVar("temp"), "println", List.of(new JStr("Hello, world!")));
List<JStmt> prog = List.of(stmt1, stmt2);

interpret(prog); // run the program!
// 1. interpret begins evaluating stmt1, initially env, the variable environment, is empty
// 1a. JSFld#eval looks up the object stored in field "out" in class "java.lang.System"; call this o1
// 1b. The JSFld "writes" to "temp", so env is updated so "temp" maps to o1
// 2. interpret begins evaluating stmt2, in the current env (with the mapping for temp)
// 2a. JCall#eval begins by evaluating the receiver and the arguments by calling their eval methods
// 2aa. The receiver is a JVar("temp"), so JVar#eval looks up "temp" in env, returning o1
// 2ab. The argument is a JStr("Hello, world!"), so JStr#eval just returns the string "Hello, world!"
// 2b. Next, JCall#eval looks up the method "println(java.lang.String)" in the class of o1
//     The argument type is java.lang.String because that is the class of "Hello, world!"
// 2c. Next, JCall#eval calls the method on the receiver o1 with the argument "Hello, world!"; the returned object is o2
// 2d. The JCall "writes" to "x", so env is updated so "x" maps to o2
```

a. (3 points) Your interpreter will track the values of variables using an *environment*. Implement the class Env as shown in the diagram below. Env is an example of what design pattern? Explain briefly.



Answer:

```
public class Env {
  Map<String, Object> theMap = new HashMap<>();
  Object lookup(String x) { return theMap.get(x); }
  void update(String x, Object v) { theMap.put(x, v); }
}
```

This is an example of the *adapter pattern*, because Env uses the functionality of HashMap but with a slightly different interface. It would also be reasonable to say this is a *wrapper*. It's slightly wrong to say this is the *proxy pattern*, because the interfaces of Env and Map are different, but that answer was accepted.

b. (3 points) Next, let's extend the JExpr interface to include a method eval for returning the object corresponding to an expression:

```
public interface JExpr { Object eval(Env env); }
```

Write the code for eval for all classes that implement JExpr. Your code for JVar will simply look up the value of the variable in the environment. This code may raise any exception if the given variable is not in the environment.

Answer:

```
public class JInt implements JExpr { int i; Object eval(Env env) { return i; } }
public class JStr implements JExpr { String s; Object eval(Env env) { return s; } }
public class JVar implements JExpr { String x; Object eval(Env env) { return env.lookup(x); } }
```

c. (12 points) The next step is to extend JStmt to include a method eval for running a statement. This eval method does not return any result, though it should update its env argument with the effect of the statement. For example, after running stmt1, the env should be updated so that env.lookup("temp") == System.out. When looking up the Method to call using reflection, use the run-time types of the arguments. Below is the revised JStmt interface plus some useful methods from the Java standard library.

```
public interface JStmt { void eval(Env env); }

public class Object {
  Object getClass();
}
public class Class<T> {
  static Class<?> forName(String className);
  Field getField(String name);
  Method getMethod(String name, Class<?>... parameterTypes);
}
public class Field {
  Object get(Object obj);
}
public class Method {
  Object invoke(Object org, Object... args);
}
public interface List<E> {
  void add(int i, E e); // add e to position i of list
  void add(E e); // add e to end of list
  Object[] toArray();
}
```

Write the code for eval for all classes that implement JStmt.

Answer:

```
public class JSFld implements JStmt {
  String x; String cls; String fld;
  void eval(Env env) {
    Object val = Class.forName(cls).getField(fld).get(null);
    env.put(x, val);
  }
}
public class JCall implements JStmt {
  String x; JExpr r; String m; List<JExpr> args;
  void eval(Env env) {
    Object o_r = r.eval();
    List<Object> o_args = new LinkedList<>();
    List<Class<?>> o_arg_types = new LinkedList<>();
    for (JExpr arg : args) { Object v = arg.eval(); o_args.add(v); o_arg_types.add(v.getClass()); }
    Object val = o_r.getClass().getMethod(m, o_arg_types.toArray()).invoke(o_r, o_args.toArray());
    env.put(x, val)
  }
}
```

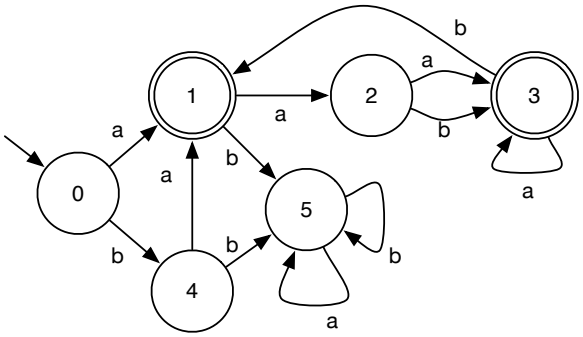
d. (2 points) Finally, write a function void interpret(List<JStmt> stmts) that, starting with an empty Env, executes a sequence of statements.

Answer:

```
void interpret ( List <JStmt> stmts) {
    Env env = new Env();
    for (JStmt stmt : stmts) { stmt.eval(env); }
}
```

Question 4. Testing (25 points). In this problem, you will consider testing deterministic finite automata (DFAs), which are just like the NFAs from Project 5, except (1) they do not have ε-transitions and (2) each state can have at most one outgoing transition per input character.

a. (4 points) A *test case* for a DFA is an input string along with its expected result (*accept*, *reject*). The *state coverage* of a test suite is the set of states cumulatively visited when running a DFA on those tests. For the following DFA over strings of `a`'s and `b`'s, give a test suite that provides full state coverage. You may or may not need to use all lines in the table.



Input	Accept or reject?	States covered (in order visited)
bb	reject	0,4,5
aaa	accept	0,1,2,3

Answer: (There are also solutions with just one test case; can you find one?)

b. (6 points) The *transition coverage* of a test suite is the set of transitions cumulatively traversed when running a DFA on those tests. For the DFA above, give a test suite that provides full transition coverage. Write a transition as nx, where n is the state the transition starts from and x is the character of the transition. For example, the transition from state 0 to state 1 on `a` would be written as `0a`. You may or may not need to use all lines in the table.

Input	Accept or reject?	Transitions covered (in order traversed)
bbab	reject	0b,4b,5a,5b
bab	reject	0b,4a,1b
aaababa	accept	0a,1a,2a,3b,1a,2b,3a

c. (12 points) Implement the following method test (some useful class definitions appear above it), which runs a DFA test suite and returns its coverage and which inputs failed. You may write helper method(s) if needed. *You may want to use method String#charAt(int n), which returns the character at position n in the String.*

```
public class DFA { /* DFA representation */
  int start; /* start state */
  Map<Pair<Integer, Character>, Integer> ts; /* transitions */
  /* A mapping from (s1, a) to s2 indicates that starting in state s1, there is a transition on a to state s2 */
  Set<Integer> fs; /* set of final states */
}
public class Pair<A, B> { A a; B b; Pair(A a, B b) {this.a=a; this.b=b;} A fst() {return a;} B snd() {return b;}
  boolean equals (...)... int hashCode (...)... /* etc */
}
public interface Set<E> {
  boolean add(E e); // inserts e in the set, returns true if e was not already in set
  boolean addAll( Collection <E> c); // add all elements of c to the set, returns true if new elements added
  boolean contains(Object o); // return true iff o in set
}
public class HashSet<E> implements Set<E> { ... }

public class Result { /* Result of running a test suite */
  Set<String> failures; /* failing inputs */
  Set<Integer> sCov; /* states covered */
  Set<Pair<Integer,Character>> tCov; /* transitions covered */
}

// dfa is a DFA, represented as described above
// suite is a list of pairs (input, expected result), where the result is true for accept and false for reject
// your code can crash if an input tries to take a non-existent transition
Result test(DFA dfa, List<Pair<String, Boolean>> suite) {
```

Answer:

```
Result test(DFA dfa, List<Pair<String, Boolean>> suite) {
  Result r = new Result();
  for (Pair<String, Boolean> test : suite) {
    match(r, dfa, test.fst(), test.snd());
  }
  return r;
}

void match(Result r, DFA dfa, String input, boolean expected) {
  int s = dfa.start;
  r.sCov.add(s);
  for (int i = 0; i < input.length(); i++) {
    Pair<Integer, Character> t = new Pair(s, input.charAt(i));
    r.tCov.add(t);
    s = dfa.ts.get(t);
  }
  if (dfa.fs.contains(s) != expected) { r.failures.add(input); }
}
```

d. (3 points) Suppose we have a DFA d and a test suite List<Pair<String, Boolean>> suite, and that d passes the full test suite. Now suppose we change d slightly to yield a new DFA d'. For example, we might add a new state or change a few transitions. To save time, rather than running all of suite, we would like to only run tests that might be affected by the changes. In English, briefly discuss how you might compute the smallest subset of suite to run on d' based on the changes.

Answer: When running suite on d, we can compute the states and transitions covered. Then on d', we only need to run the subset of tests whose coverage intersects with the differences (states and transitions) between d' and d.

Question 5. Multi-Threading (15 points).

a. (5 points) Consider the following interface for a *barrier*, which is a synchronization construct in which n parties wait for each other:

```
public class Barrier {
  Barrier(int n); // create a barrier for n parties
  void barrier(); // wait at the barrier until all n parties arrive
}
```

Constructing a b = new Barrier(n) creates a barrier for n parties. When the first n-1 threads call b.barrier(), they should block (no busy waiting allowed). When the nth thread calls b.barrier(), the other, n-1 waiting threads should wake up and continue; the barrier should be reset to its initial state so it can be used again; and the nth thread should also continue (without blocking).

Implement the Barrier class. *Hint: You will probably want to use await/signalAll or wait/notifyAll, but you probably won't need them in a loop. Here are some methods you might need from the standard library.*

```
public class Object {
  void wait() throws InterruptedException;
  void notifyAll();
}
public class ReentrantLock implements Lock {
  void lock();
  void unlock();
  Condition newCondition();
}
public interface Condition {
  void await() throws InterruptedException;
  void signalAll();
}
```

Answer:

```
public class Barrier {
  int n, cur;

  Barrier(int n) { this.n = n; }

  synchronized void barrier() {
    cur++;
    if (cur == n) { notifyAll(); cur = 0; }
    else {
      try { wait(); } catch (InterruptedException e) { }
} } }
```

b. (7 points) Consider the following class that simulates a bank account (with strings for account names and integers for balances).

```
public class Bank {
  Map<String, Integer> accts = new HashMap<>(); /* map from account name to balance */

  private synchronized int getBalance(String name) {
    return accts.getOrDefault(name, 0); /* a */ /* return balance of account or 0 if not in map */
  }
  private synchronized void setBalance(String name, int amt) {
    accts.put(name, amt);                /* b */
  }
  public void deposit(String name, int amt) {
    int balance = getBalance(name);      /* c */
    setBalance(name, balance + amt);     /* d */
} }
```

Unfortunately, if the above code is used in a multi-threaded context, it might result in an incorrect balance. Demonstrate the problem by writing a complete, multi-threaded Java class with a main method that uses the Bank class; runs at least two threads; and may exhibit the problem under certain thread schedules.

Answer:

```
public class Main implements Runnable {
  static Bank b = new Bank();
  public void run() { b.deposit("a", 100); }

  public static void main(String[] args) {
    Thread t1 = new Thread(new Main()); Thread t2 = new Thread(new Main());
    t1.start(); t2.start();
} }
```

Briefly describe a bad thread schedule that will result in an incorrect final balance for your code above and say what that final balance is. You may wish to refer to the statement labels a–d above in your description.

Answer: The scheduler may run t1 first until it reaches c, so its balance==0. Then it may switch to t2 until it reaches c, so its balance==0 also. Now if t2 continues it sets balance=100, and when the scheduler switches back to t1 it will also set balance=100. So the final balance is 100 even though 200 was deposited.

c. (3 points) Does the Bank class have any *data races*? Explain your answer in terms of the definition of a data race we gave in class.

Answer: No. There is shared memory (accts), and it is accessed by multiple threads with at least one access a write (on line b), but all accesses are protected by the same lock, so there is no data race.