**Name:**

# Final Exam

COMP 150-SEN

Software Engineering Foundations

Spring 2019

May 3, 2019

## Instructions

**This exam contains 14 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 | Topic | Max |
|---|---|---|
| 1 | Short Answer | 20 |
| 2 | Design Patterns | 20 |
| 3 | Testing and Reflection | 20 |
| 4 | Program Verification | 10 |
| 5 | Refactoring | 10 |
| 6 | Delta Debugging | 20 |
| Total | | 100 |

**Question 1. Short Answer (20 points).**

**a. (5 points)** In his paper, Parnas listed three potential categories of benefits of software modularity: *managerial*, *product flexibility*, and *comprehensibility.* Explain briefly how **two** of these benefits could arise from modularity.

> **Answer:** Managerial—different groups can work on different modules with less communication. Product flexibility—developers can change one module without changing another. Comprehensibility—the system can be understood one module at a time.

**b. (5 points)** Briefly explain the difference between *confidentiality* and *integrity.*

> **Answer:** Confidentiality means a system protects private information from release to adversaries. Integrity means a system protects its internal information from being modified by adversaries. (Many people drew the pictures from the lecture slides, which was also acceptible.)

**c. (5 points)** In Brooks's essay *No Silver Bullet...*, what does "silver bullet" refer to?

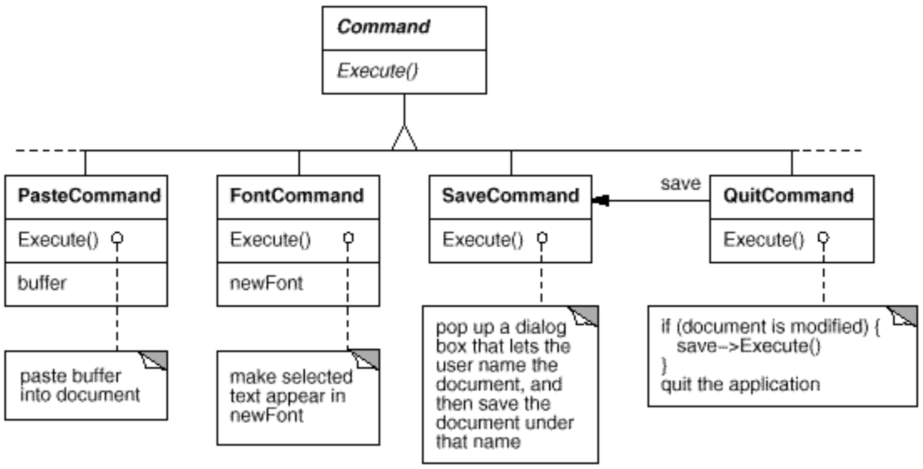
> **Answer:** From the paper: "But, as we look to the horizon of a decade hence, we see no silver bullet. There is no single development, in either technology or management technique, which by itself promises even one order of magnitude improvement in productivity, in reliability, in simplicity."

**d. (5 points)** Consider the following ethical situation. Blocker Plus is a content filter that blacklists web sites not suitable for children. Blocker Plus does not disclose how it decides which sites to block, but in fact it uses machine learning to automatically identify inappropriate content. Recently, the developers found their technology is blocking violent web sites (as its users would like), but users are complaining that it is blocking web sites about vaccinations. Weighing the tradeoffs, and without justification to its users, the company selling Blocker Plus decides not to change their product despite the complaints.

List one ethical principle Blocker Plus is following, and one ethical principle Blocker Plus is violating. Explain your answers very briefly (1-2 sentences each). Only refer to principles in Part 1 of the ACM Code of Ethics.

> **Answer:** Any plausible answer is okay. The system mostly adheres to 1.1, Contribute to Society, by helping solve a societal problem. The system violates 1.2, Avoid Harm, by blocking critical information, and 1.3, Be Honest and Trustworthy, by not being transparent about their technology.

**Question 2. Design Patterns (20 points).** In the *command pattern*, command objects represent actions. For example, here is a UML diagram (from the Design Patterns book) showing the command pattern for a word processing app. There is an interface, Command, with four implementations for the paste, font, save, and quit commands:


Command
Execute()
PasteCommand
Execute()
buffer
paste buffer into document
FontCommand
Execute()
newFont
make selected text appear in newFont
SaveCommand
Execute()
pop up a dialog box that lets the user name the document, and then save the document under that name
save
QuitCommand
Execute()
if (document is modified) {
save->Execute()
}
quit the application

We will use this idea to implement undo-able cut and paste commands for a text editing program. Here are the program's APIs:

```
public class State {
  public static StringBuffer doc; /* The text document */
  public static String buffer; /* The paste buffer */
}
```

```
interface Command {
  void execute();   // run the command
  void undo();      // undo a command
}
```

Below, you will develop Cut and Paste commands. For example:

```
State.doc = "abcdefgh"
c = new Cut(3, 5);
c.execute();   // doc = "abcfgh" and buffer = "de"
p = new Paste(0);
p.execute();   // doc = "deabcfgh" and buffer = "de"
p.undo();      // doc = "abcfgh" and buffer = "de"
c.undo();      // doc = "abcdefgh" and buffer = "de"
```

In the questions below, n and m are non-negative integers, and position 0 is the position of the first character in doc. Assume that your cut and paste commands will be executed exactly once, and will be undone at most one time and in the correct sequence (e.g., if undo is called on two different commands, the calls will be in the reverse order the commands were executed in). You'll probably need the following methods:

```
class StringBuffer {
  void delete(int start, int end);  // Removes substring from start to end-1, inclusive
  void insert(int offset, String s);  // Inserts s at offset
  String substring(int start, int end); // Returns substring from start to end-1, inclusive
}
```

**a. (6 points)** Implement a Paste command such that new Paste(n) creates a command whose execute method inserts the contents of buffer at position n in doc, and its undo method reverses the action.

```
class Paste implements Command {
```

**Answer:**

```
int n;
int pasted_size ;
Paste(int n)  { this.n = n; }
void execute() { State.doc.insert (n, State.buffer );  pasted_size  = State.buffer.length ();  }
void undo() { State.doc.remove(n, n+pasted_size );  }
}
```

Another approach is to save State.doc in the object. That technically doesn't work because StringBuffer is mutable, but since that wasn't explained in the problem such solutions were acceptable.

**b. (6 points)** Implement a Cut command such that new Cut(n, m) creates a command whose execute method removes the contents of doc from n to m-1, inclusive, and stores the removed text in buffer. Its undo method reverses the action. You can assume n and m are in-bounds.

```
class Cut implements Command {
```

**Answer:**

```
int n, m;
String removed;
Cut(int n, int m)  { this.n  = n; this.m = m; }
void execute() { removed = State.doc.substring(n, m); State.doc.remove(n, m); }
void undo() { State.doc.insert (n, removed);  }
}
```

**c. (8 points)** Suppose you wanted to make two changes to the Command interface:

1. Add a redo() operation to Command that reapplies a command that was undone.
2. Add checks (assertions) to ensure that execute, undo, and redo operations are called in a sensible order. For example, suppose a = new Cut(...), b = new Paste(...), c = new Cut(...). Then calling a.execute(); b.execute(); c.execute(); c.undo(); b.undo(); b.redo(); makes sense. But calling a.execute(); b.execute(); a.undo(); or a.execute(); a.redo(); or a.execute(); a.execute(); do not make sense.

Describe, in English, how you would need to change Cut and Paste above to support these changes.

**Answer:** Key points:

- To support redo(), we need to either restore the paste buffer when undoing commands or save what the paste buffer was. Otherwise the code is similar to execute().
- Add a flag to each operation to make sure it's only executed once.
- For the assertions, the easiest solution is to store the commands in a stack. Then when undoing an operation, we make sure that operation is on top of the stack.
- However, it can't quite be a stack, because we shouldn't pop at an undo. We should leave the commands on the stack and then allow redos to apply in the same order they were undone, until running an execute command resets the stack so previously undone commands can't be redone.

There are also alternative reasonable approaches such as effectively keeping a stack of operations by linking each operation that's carried out to the previous operation. But just remembering a single, global last operation is not sufficient, nor is counting operations.

(This page intentionally left blank)

**Question 3. Testing and Reflection (20 points).** Using Java's dynamic proxy facility, write a class Recorder that can be used to wrap an object in a proxy that *records* all calls to the object and subsequently emits (to stdout) Java source code that can be used to *replay* those calls.

For example, suppose we have the following code:

```
interface I {                           class A implements I {
  void m(int a, int b);                   void m(int a, int b) { System.out.println ("a=" + a + ",b=" + b; }
}                                       }
```

Here is an example use of the Recorder class:

```
A a = new A();
Recorder r = new Recorder(a, I.class);   // Create recorder object
I wrapped = (I) r.getProxy();            // Get the stand-in for a
wrapped.m(1, 2);                         // Records call, then prints "a=1,b=2"
wrapped.m(3, 4);                         // Records call, then prints "a=3,b=4"
r.printTest();                           // Prints the following four lines:
                                         // void test(I o) {
                                         //   o.m(1, 2);
                                         //   o.m(3, 4);
                                         // }
```

- Don't worry about exceptions from reflection, and you can assume all preconditions are satisfied.
- Calling new Recorder(o, c) creates a new recorder object, where o is the underlying object to record method calls to, and c is an interface it implements.
- if r is a Recorder, calling r.getProxy() returns an object proxy that implements c. Calling r.getProxy() multiple times always returns the same object.
- Invoking methods on proxy first records (internal to r) the called method name and arguments. Then it delegates to o and returns the result. You can assume that all arguments are integers.
- Calling r.printTest() prints, to standard output, source code of a method test that takes one argument that implements c and invokes the recorded method calls, in the same order of the calls and with the same arguments.
- You can use any part of the Java standard library. You'll probably want to use the methods below.
- Write your answer on the next page.

```
class Class { String getName(); }
class Method {
  Object invoke(Object obj, Object[] args); // invoke this method on obj with args
}
interface InvocationHandler {
  Object invoke(Object proxy, Method method, Object[] args); // Processes an invocation of method on proxy with args
}
class Proxy {
  // Return a proxy for iface that dispatches method invocations to h. (We've simplified this method a bit.)
  static Object newProxyInstance(Class iface, InvocationHandler h);
}
class LinkedList<E> { boolean add(E e); /* appends e to the end of the list */ }
```

```
import java.lang. reflect .*;
import java. util .*;
```

**Answer:**

```
public class Recorder implements InvocationHandler {
  List<String> log = new LinkedList<String>();

  private Object base;
  private Class interface;
  Recorder(Object base, Class interface) { this.base = base; this.interface = interface; }

  private Object theProxy;
  public Object getProxy() {
    if (theProxy == null) {
     theProxy = Proxy.newProxyInstance(interface, this);
    }
    return theProxy;
  }

  Object invoke(Object proxy, Method m, Object[] args) {
    String sargs = "";
    for (Object o: args) {
      if (sargs != "") { sargs += ",␣"; }
      sargs += ((Integer) o). toString ();
    }
    log.add("o." + m.getName() + "(" + sargs + ");");
    return m.invoke(base, args);
  }

  void printTest () {
    System.out. println ("void␣test(" + interface.getName() + "␣o)␣{");
    for (String line : log) {
      System.out. println ( line );
    }
    System.out. println ("}");
  }
}
```

**Question 4. Program Verification (10 points).**

**a. (3 points)** In the following code, B is a subclass of A. Are the preconditions of m appropriate for the subclass relationship, according to the Liskov substitution principle? Explain why or why not. Your example should explain briefly in English using a concrete example of calling m.

```
class A {                                class B extends A {
  // precondition : x < y                  // precondition : 0 < x < y
  void m(int x, int y) { ... }             void m(int x, int y) { ... }
}                                        }
```

**Answer:** No. Given an A a, we could invoke a.m(-4, 10) according to the precondition on A#m. But a could actually be a B based on subclassing, and that call would violate B#m's precondition.

**b. (3 points)** In the following code, B is a subclass of A. Are the postconditions of m appropriate for the subclass relationship, according to the Liskov substitution principle? Explain why or why not using a concrete example. Ignore the possibility of integer overflow.

```
class A {                                class B extends A {
  // postcondition : ret > 0             // postcondition : ret > x*x
  int m(int x) { ... }                       int m(int x) { ... }
}                                        }
```

**Answer:** Yes. Given an A a, we could invoke a.m(10), expecting only a positive result. If the a is actually a B, we will also have a positive result (assuming the square of an integer is always positive).

**c. (2 points)** In the following code, B is a subclass of A. For B to be a subclass of A according to the Liskov substitution principle, should we have Exn1 extends Exn2 or Exn2 extends Exn1? Explain briefly.

```
class A {                                class B extends A {
  int m(int x) throws Exn1 { ... }         int m(int x) throws Exn2 { ... }
}                                        }
```

**Answer:** We need Exn2 extends Exn1. Given an A a, a caller that invokes a.m(...) can handle an Exn1, and therefore they can handle any subclass of Exn1.

**d. (2 points)** Use the assignment rule to compute the weakest precondition of the following assignment statement and postcondition. (In other words, fill in the ?.)

{ ? }   x = y + 2;   { x > z }

**Answer:** y+2 > z or equivalent.

**Question 5. Refactoring (10 points).** Consider the following code:

```
public class Rental {
    private Movie _movie;
    private int _daysRented;

    public Rental(Movie movie, int daysRented) {
        _movie = movie;
        _daysRented = daysRented;
    }
    public int getDaysRented() { return _daysRented; }
    public Movie getMovie() { return _movie; }

    public double amountFor() {
        double thisAmount = 0;
        //determine amounts for each line
        switch (getMovie().getPriceCode()) {
            case Movie.REGULAR:
                thisAmount += 2;
                if (getDaysRented() > 2)
                    thisAmount += (getDaysRented() - 2) * 15;
                break;
            case Movie.NEW_RELEASE:
                thisAmount += getDaysRented() * 3;
                break;
            case Movie.CHILDRENS:
                thisAmount += 1.5;
                if (getDaysRented() > 3)
                    thisAmount += (getDaysRented() - 3) * 1.5;
                break;
        }
        return thisAmount;
}   }

public class Movie {
    public static final int CHILDRENS = 2;
    public static final int REGULAR = 0;
    public static final int NEW_RELEASE = 1;

    private String _title ;
    private int _priceCode;

    public Movie(String title , int priceCode) {
        _title = title ;
        _priceCode = priceCode;
    }

    public int getPriceCode() { return _priceCode; }
    public void setPriceCode(int arg) { _priceCode = arg; }
    public String getTitle () { return _title ; }
}
```

Describe at least **two different refactorings** we could perform on the sample code, and **explain why those refactorings are useful.** Your refactorings should be different not just in what source code they operate on but what they actually do (e.g., don't list three uses of Move Method). You may not use renaming of variables, methods, or fields by themselves as refactorings. You may use any reasonable refactoring, not just those we discussed in class. Don't worry about getting the names exactly right.

**Answer:**

1. Add parameter daysRented to replace call to getDaysRented() in amountFor(), in preparation for step 2.
2. Move method amountFor() from Rental into Movie, since this will differ from one kind of movie to another.
3. Replace conditional switch by polymorphism by replacing the single class Movie by three subclasses ChildrensMovie, RegularMovie, and NewRelease. Split the behavior of the amountFor() method accordingly. This refactoring has many advantages. For example, it means we have an enumerate of movies rather than using an integer type; it's easy to add new movies using subclassing.

There are many other possible refactorings for this program. Any reasonable refactorings, along with an explanation, were acceptable.

**Question 6. Delta Debugging (20 points).** Suppose we have a class with a method test that takes a string s and returns true if s passes the test and false otherwise:

```
class A {
  public static boolean test(String s){  ... } // Returns true if s passes, false if it fails
}
```

Implement a method String dd(String init) that takes a string init such that A.test(init) == false and minimizes it, using the delta debugging algorithm from class and returning the minimal string. For example, if the body of test were return !s.contains("abc"), then dd("123456abc78901") would return "abc", using a sequence of calls to test like the following. (Your code doesn't need to follow this exact sequence.)

| | | | |
|---|---|---|---|
| A.test("bc78901") | passes, try other half | A.test("c7") | passes |
| A.test("123456a") | passes, increase granularity | A.test("ab") | passes, increase granularity |
| A.test("456abc78901") | fails, now have a smaller test | A.test("bc7") | passes |
| A.test("c78901") | passes | A.test("ac7") | passes |
| A.test("456ab") | passes, increase granulairty | A.test("abc") | fails, now have a smaller test |
| A.test("abc78901") | fails, now have a smaller test | A.test("bc") | passes |
| A.test("8901") | passes | A.test("ac") | passes |
| A.test("abc7") | fails, now have a smaller test | A.test("ab") | passes, can't increase granularity |

*Hint.* You'll definitely want to use String methods int length() and String substring(int beginIndex, int endIndex). The latter returns the substring beginning at beginIndex and ending at endIndex - 1, inclusive.

*Hint.* Use integer arithmetic throughout and don't worry about rounding issues. (The algorithm is very robust and will work fine.)

```
class DeltaDebug {
  public static String dd(String init) {
```

**Answer:**

```
    String cur = init;
    boolean done = false;
    int sz = cur.length() / 2; // partition size
    while (!done) {
        boolean failed = false;
        for (int i = 0; i < cur.length()/sz; i++){
            String trial = cur.substring(0, i*sz) +
                cur.substring((i+1)*sz, cur.length());
            if (!A.test(trial)) {
                cur = trial;
                sz = cur.length() / 2;
                failed = true;
                break;
            }
        }
        if (! failed) {
            // no test case fails at this granularity
            sz = sz / 2;
            if (sz==0) break;
        }
    }
    return cur;
  }
```

Another reasonable approach is to call dd recursively on finding a shorter string.