Name:

Midterm

COMP 180
Software Engineering
Spring 2020

March 11, 2020

Instructions

This exam contains 10 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		35
3		15
4		30
Total		100

Question 1. Short Answer (20 points).

a. (5 points) Briefly list two potential benefits and one potential drawback of *information hiding*.

Answer: Benefits: Can change implementation without changing client; can partition work more effectively across teams; can understand code more easily.

Drawbacks: Might decrease performance; might decide to hide the wrong information, making it harder to use the component; abstractions are often leaky despite our attempts to hide information.

b. (5 points) Give two potential drawbacks to using *inheritance*.

Answer:

- Tight coupling between superclass and subclass.
- Superclass methods pollute namespace of subclass.
- Class hierarchies are brittle.

c. (5 points) List one similarity and one difference between the *adapter pattern* and the *proxy pattern*.

Answer: Similarities: Both are wrappers; both can interpose operations before/after a component's methods

Differences: Proxy maintains the same interface as the subject, while adapter changes it.

d. (5 points) List two potential disadvantages of using reflection.

Answer:

- Extremely verbose
- Potential security concerns
- No compile-time type checking
- Performance overhead

Question 2. Programming in Java (35 points). In this problem, you will implement several methods for class ArraySet, which stores a set of integers using an array, and also stores next, the index of the first free space in the array:

```
public class ArraySet {
  int [] elts ;
   int next = 0;
  ArraySet() { elts = new int[17]; next = 0;}
```

For example, the set {1,2,3} might be represented with an ArraySet with elts = [1,2,3,0,0,0,...,0] and next=3.

a. (5 points) Write a method bool member(int x) that returns true if and only if x is in the set.

```
public bool member(int x) {
```

Answer:

```
    for (int i = 0; i < next; i++) {
      if ( elts [i] == x) { return true; }
    }
    return false ;
  }
```

b. (7 points) Write a method boolean remove(int x) that removes x from the set and returns true if x was previously in the set, and otherwise returns false.

```
public boolean remove(int x) {
```

Answer:

```
    int pos = next;
    for (int i = 0; i < next; i++) {
      if ( elts [i] == x) { pos = i; break; }
    }
    if (pos == next) { return false ; }
    for (int i = pos; i < next; i++) {
      elts [i] = elts [i+1];
    }
    next--;
  }
```

c. (9 points) Write a method boolean insert(int x) that either adds x to the set and returns true if x was not already in the set, or returns false if x was in the set. Be sure to support the case when the array is full and needs to be reallocated.

```
public boolean insert(int x) {
```

Answer:

```
  if (member(x)) { return false; }
  if (next == elts.length) {
    int [] new_elts = new int[elts.length*2];
    for (int i = 0; i < next; i++) { new_elts[i] = elts[i]; }
    elts = new_elts;
  }
  elts[next++] = x;
  return true;
}
```

d. (7 points) Write a method Iterator<Integer> iterator() that returns a standard, Java-style iterator for the set. For your reference, the part of the Iterator interface you need to implement is below:

```
public interface Iterator<E> {
  boolean hasNext();
  E next();
}

public Iterator<Integer> iterator() {
```

Answer:

```
    return new Iterator<Integer>() {
      int n = next;
      public boolean hasNext() { return n < next; }
      public Integer next() { int tmp = elts[n]; n++; return tmp; }
    }
  }
```

e. (7 points) In English, briefly, but precisely, discuss what changes you would need to make to ArraySet and any or all of its methods so your iterator from part d above would throw ConcurrentModificationException (let's abbreviate that CME) if the set is modified after the iterator is created. There are a few design choices in solving this problem—just explain one design that you believe is reasonable.

Answer:

- Add an int nonce field to the class, and increment it each time insert or remove is called.
- When creating the iterator, store the current nonce at the creation time in the iterator object.
- At a call to hasNext or next, raise an exception if the nonce of the underlying set has changed.
- It is not strictly necessary to do this at insert calls because insertion is always done at the end. It is not strictly necessary to do this on insert and remove calls that don't change the set.

Question 3. Design Patterns and Reflection (15 points).

a. (10 points) Below, we've started writing a class MemProxy, which uses Java's *dynamic proxies* to create a *memoizing* wrapper for an object. More specifically, calling MemProxy.create(o) returns a new object that implements all the interfaces of o. Finish the code by implementing the invoke method so that calls handled by a MemProxy first look up the arguments in the cache and return the corresponding result if it's there, and otherwise compute the result using the underlying object, store the result in the cache, and then return it. For example, the sequence p = MemProxy.create(o); p.expensive_method(42); p.expensive_method(42); will only call expensive_method once—the second time, the cached result from the first call will be returned.

If you forget the exact interface for maps or lists, just guess something reasonable and make a note about your guess.

```
import java.lang.reflect.*;
public class MemProxy implements InvocationHandler {
  private Object wrapped;
  private MemProxy(Object wrapped) { this.wrapped = wrapped; }
  public static Object create(Object o) throws Exception {
    return Proxy.newProxyInstance(o.getClass().getClassLoader(), o.getInterfaces(), new MemProxy(o));
  }

  /* Be sure to add a field to store prior calls, i.e., fill in the ...'s in the following:

     private Map<...> cache = new HashMap<>();

  */

  /* proxy is the receiver whose method was invoked; method is the method; args is the actual arguments */
  Object invoke(Object proxy, Method method, Object[] args) {
```

Answer:

```
    Pair k = new Pair(method, args);
    if (cache.containsKey(k)) { return cache.get(k); }
    Object r = method.invoke(wrapped, args);
    cache.put(k, r);
    return r;
  }

  private Map<Pair<Method, Object[]>, Object> cache = new Map<>();
```

Note: The solution above doesn't quite work because Array#equals and #hashCode don't do deep equality/hashing. But the above is good enough for an exam answer. We'll leave it as an exercise to the reader to implement a wrapper (a proxy?!) for Array to fix this.

b. (5 points) Memoization makes a key assumption about the underlying methods being memoized. Write a class C such that if we set C p = (C) MemProxy.create(new C()), then calling p.m() twice will return a different result than setting C o = new C() and calling o.m() twice.

Answer: Memoization is only guaranteed to work with pure methods, i.e., those without state.

```
class C { int x; int m() { return x++; } }
```

Question 4. Testing (30 points). Consider the following method:

```
   void m(int a, int b, int c) {
1.   int x = 0;
2.   if (a == 0) {                    /* (a) */
3.     x++;
     }
4.   if (b < 5) {                     /* (b) */
5.     if (a >= c && c != 0) {        /* (c) */
6.       x--;
       }
7.     else {
8.       x++;
       }
9.     x++;
     }
10.  if (x == 3) {                    /* (d) */
11.    x = 42;
     }
   }
```

a. (8 points) Write down two calls to m that cumulatively achieve full *statement coverage* of m. For example, one call might be m(0,0,0). For each call, list the numbered statements that it covers.

Call	Statements covered
m(0,0,0)	1,2,3,4,5,7,8,9,10,11
m(1,0,1)	1,4,5,6,9

b. (12 points) Write down three calls to m that collectively achieve full *branch coverage* of m, meaning they exercise the true and false cases for the lettered branch conditions. For each call, list the lettered branch conditions that it covers, with a + after them for true branches or a − after them for false branches. For example, $a+$ would indicate branch (a) is true. For this problem, *do not* split the condition (c) even though we did that in class.

Call	Branches covered
m(0,0,0)	a+, b+, c-, d+
m(1,0,1)	a-, b+, c+, d-
m(0,5,0)	a+, b-, d-

c. (5 points) Write a method `void m(int a)` for which, in practice, no test suite can provide full *path coverage*. Explain your answer in at most two sentences. For full credit, your method should be much harder to test than its code size would suggest, i.e., if the method body has n expressions, it should take much worse than $O(n)$ time to test all paths.

There are very short answers to this question, but if you can only think of a long answer, you can write things like "repeat this statement n times" rather than writing out the code fully.

Answer:

```
void m(int a) { for (int i=0; i<a; i++); }
```

There are as many paths through this program as there are nonnegative values of `a`, which is far too many to test.

d. (5 points) Briefly explain the difference between *unit tests* and *integration tests*. Also say whether the tests you wrote in parts a and b of this question are unit tests or integration tests.

Answer: Unit tests focus on an individual component at a time, typically a method, but possibly a class or package. Thus, the tests in parts a and b are unit tests. In contrast, integration tests look at the whole system, to make sure it works correctly together.