Name:

Midterm

COMP 150-SEN

Software Engineering Foundations

Spring 2019

March 13, 2019

Instructions

This exam contains 9 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		30
2		25
3		25
4		20
Total		100

Question 1. Short Answer (30 points).

a. (5 points) Briefly explain the difference between *overriding* and *overloading* in Java.

b. (5 points) Briefly explain one potential advantage and one potential disadvantage of using *information hiding* to build software.

c. (5 points) Suppose we wanted to apply fuzz testing to a Java compiler. If our goal is to achieve high coverage, could we apply the fuzzing technique in the Miller et al paper directly? If not, what extension(s) in american fuzzy lop (AFL) or libfuzzer would we need to use?

d. (5 points) Briefly explain what a *regression* is and what *regression testing* is.

e. (5 points) List three software architectures that might be involved in a software system for storing and retrieving student records on the web. Explain your answer briefly, in just a few sentences. Be sure to state what the architectural components correspond to (e.g., if you propose using a peer-to-peer architecture, be sure to say what the peers are).

f. (5 points) What is *cohesion*? Give one reason that high cohesion within a module is good.

Question 2. Java (25 points). In project 1, you developed an implementation of the following interface:

```
public interface Graph {
    boolean addNode(String n); // return true if node was not in graph
    boolean addEdge(String n1, String n2); // return true if edge was not in graph
    boolean hasNode(String n);
    boolean hasEdge(String n1, String n2);
    List<String> succ(String n);
    // List<String> pred(String n); - skip this
    // boolean connected(String n1, String n2); - skip this
}
```

Below, write a class AdjGraph that implements the first five methods of Graph using an adjacency matrix. Your class should use the field given below to store the graph. The APIs for HashMap and HashSet are on the next page. (You may also use methods or classes not listed on the next page.) You can continue your code on the next page if you need more space.

```
import java.util.*;

public class AdjGraph implements Graph {

  // nodes.get(src).contains(dst) == true if and only if there is an edge from src to dst
  private HashMap<String, HashSet<String>> nodes;
```

```
class HashMap<K,V> {
  boolean containsKey(Object key); // returns true if this contains a mapping for key
  V get(Object key); // returns value mapped to key, or null if none
  V put(K key, V value); // associates value with key in this map
}

class HashSet<E> {
  boolean add(E e); // add e to set if not already present; returns false if e was in set
  boolean contains(Object o); // returns true if this set contains o
  Iterator<E> iterator(); // returns an iterator over the set
}

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

class LinkedList<E> {
  boolean add(E e); // add e to the end of this list
}
```

Question 3. Design Patterns (25 points). Consider the following abstract syntax tree (AST) for boolean expressions:

```
interface BExpr { Object accept( Visitor v); }
class BVal implements BExpr {
  public boolean val; BVal(boolean val) { this.val = val; }
}
class BNot implements BExpr {
  public BExpr child; BNot(BExpr child) { this.child = child; }
}
class BAnd implements BExpr {
  public BExpr left, right; BAnd(BExpr left, BExpr right) { this.left = left; this.right = right; }
}
```

For example, we can represent the expression $true \wedge \neg false$ as

```
BExpr example = new BAnd(new BVal(true), new BNot(new BVal(false)))}
```

In this problem, you will implement the visitor pattern for the following visitor interface, which is slightly extended from what we saw in class:

```
interface BVisitor {
  Object visit (BVal e);
  Object visit (BNot e, Object child );
  Object visit (BAnd e, Object left , Object right );
}
```

Here, the visit methods return an object, and they take as additional arguments the objects returned by visiting their children, if any. To make this work, the accept method has also been modified to return an object.

a. (13 points) Write the missing code for the accept methods of BVal, BNot, and BAnd to implement a postorder traversal (visit the children first, left to right, and then the node itself). You don't need to copy the field or constructor definitions.

```
class BVal {
  public Object accept( BVisitor v) {
```

b. (12 points) Next, write code for a visitor that traverses a BExpr and returns the result of evaluating it. For example, example.accept(new BEval()) == Boolean.TRUE. Do *not* modify the BExpr subclasses. You can use the booleanValue() method of class Boolean to convert a boxed boolean into an unboxed one.

```
class BEval implements BVisitor {
```

Question 4. Reflection (20 points). Use reflection to develop an alternative implementation of question 3a in which accept is defined just once, in BExpr. More specifically, change BExpr into a class rather than an interface, and write code for an accept method that, when inherited, will behave correctly for every subclass. Part of the reflection API is shown below (you might need some, none, or all of these methods). Ignore the potential exceptions reflective methods may raise, and also ignore generics (e.g., write Class instead of Class< ... >). Recall that boolean.class is a Class instance representing booleans, and similarly for Object.class. To keep things simpler, your code can be specialized to this particular set of BExpr subclasses.

```
class Object {
  Class getClass(); // returns the runtime class of this object
}
class Class {
  static Class forName(String name); // returns the class named name
  Field getField(String name); // return field with given name, throws NoSuchFieldException if field not found
  Field[] getFields(); // return all public fields of this class
  Method getMethod(String name, Class... paramTypes); // return method with given name and arg types
  Method[] getMethods(); // return all public methods of this class
  String getName(); // return the name of this class
}
class Field {
  Object get(Object obj); // return the value of obj's field
}
class Method {
  Object invoke(Object obj, Object... args); // invoke this method on obj with args
}
```

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

class BExpr {
  Object accept(BVisitor v) {
```