University of Illinois at Urbana-Champaign
Department of Computer Science

Final Exam

CS 427: Software Engineering I
Fall 2013

December 18, 2013

TIME LIMIT = 3 Hours
COVER PAGE + 14 PAGES

Write your name and netid neatly in the space provided below; **write your netid** in the upper right corner of **every page**.

Name: _____
Netid: _____

This is a closed book, closed notes examination. You may not use calculators or any other electronic devices. Any sort of cheating on the examination will result in a zero grade.

We cannot give any clarifications about the exam questions during the test. If you are unsure of the meaning of a specific question, write down your assumptions and proceed to answer the question on that basis.

Do all the problems in this booklet. Do your work inside this booklet, using the backs of pages if needed. The problems are of varying degrees of difficulty so please pace yourself carefully, and answer the questions in the order which best suits you. **Answers to essay-type questions should be as brief as possible.** If the grader cannot understand your handwriting, you will get 0 points.

There are 17 questions on this exam and the maximum grade on this exam is 85 points.

Page	Points	Score
1	10	
2	8	
3	5	
4	6	
5	10	
6	4	
7	10	
Total:	53	

Page	Points	Score
8	5	
9	6	
10	8	
11	5	
12	3	
13	5	
Total:	32	

1. XP Project Management

(a) During Iteration 1 of the course project, your team could submit a "spike" as a deliverable.

[2] i. Which of these correctly describe an XP spike? (Circle **ALL** that apply.)

1. A spike may be used to try out a tough design.
2. A spike is another name for a major milestone.
3. A spike is a way to quickly try out an unfamiliar technology.
4. A spike must be done at the beginning of every XP project.
5. A first attempt at a solution which may be thrown away.

[2] ii. In terms of testing, how is working on a spike different from regular XP iterations?

(b) Circle **one** of **Planning Game** or **Refactoring** and answer the following.

[2] i. Why is that practice advocated in XP?

[2] ii. How did your team follow the practice during the course project?

[2] (c) In MP4, you learned that some development groups write their user stories on physical 3inch X 5inch index cards (called "bits of plain card" on the web site given for MP4). Why are user stories written on such small cards?

2. Software Quality Assurance and XP

[2] (a) In the lecture on Software Quality Assurance (SQA), we mentioned that one of the goals of measuring metrics is to be able to answer the question: *"How accurate are our estimates?"* In this context, **what does the term "velocity" mean** in an XP process?

(b) A crucial issue that arises in software development is how to know whether the code that the developers are writing will result in software that the customer actually needs.

[2] i. In this context, briefly describe the **two** terms, **Verification** and **Validation**.

[2] ii. **Describe one** XP practice that can be used to improve the chances that the software being developed will meet customer needs.

3. Bug Advocacy

[2] (a) The lecture and slides on Bug Advocacy talked about a process called **Bug Triage**. Why is this process needed during software development?

4. Testing Mistakes

(a) In "Classic Testing Mistakes", Brian Marick writes about some common pitfalls in the way many organizations approach software testing. For each of the following, **describe one reason** why it is considered a "Testing Mistake", according to the reading and the lecture on this topic.

[1] i. "Sticking stubbornly to the test plan". (Hint: We saw a related video during a lecture.)

[1] ii. "The testing team is the only team responsible for assuring software quality".

[1] iii. "Over-reliance on beta testing".

[2] (b) One of the "Classic Testing Mistakes" is **Attempting to automate every test**. Describe **two** cases when manual testing is better than automated testing. Similarly, describe **two** cases when automated testing is better than manual testing.

5. Metrics

[1] (a) Does a large number of tests directly translate to a better quality software? Why or why not? **Give one reason** to support your answer.

[1] (b) When people talk about testing code, sometimes they refer to the static code size, e.g., "I tested a million-line software". What would be one better metric to use in this context?

(c) Circle **one** of **Cyclomatic Complexity** or **Code Coverage** and answer the following.

[1] i. What does this code metric measure?

[1] ii. How can this metric be used to improve software testing?

[2] (d) Recall the reading and lecture on Chidamber & Kemerer object-oriented metrics. Circle **one** of **Coupling Between Object Classes (CBO)** or **Weighted Method per Class (WMC)** and answer the following. Why does a higher value of that metric lower the quality of a software system?

3 6. Components and Reuse

For your final projects, we highly recommended that you extend an existing Jenkins plug-in instead of creating one from scratch. The lecture on "Components and Reuse" mentioned that such reuse should only be done when the benefits of reuse outweigh the costs of reuse. Describe **two benefits** and **one drawback** of reusing an existing Jenkins plug-in for your course project instead of building one from scratch.

7. Testing & Debugging Techniques

1 (a) What is the difference between black-box and white-box testing?

2 (b) According to Chapter 29 of Code Complete 2, which statements are true about **smoke testing**. (Circle **ALL** that apply.)

1. Smoke tests should only test GUI
2. Smoke tests should be automated
3. Smoke tests are a kind of regression tests
4. Smoke tests should run fast

2 (c) What is the class invariant for the example class below? (Hint: Think of boundary values.)

```
class ValueRange {
  int lo = Math.minInt();
  int hi = Math.maxInt();
  // Write the class invariant relating lo to hi.
   // [TODO] WRITE YOUR CODE BELOW


  void setLo(int i) { if (i <= hi) lo = i; }
  void setHi(int i) { if (i >= lo) hi = i; }
}
```

2 (d) Why is simplifying test cases important in debugging? (Circle **ALL** that apply.)

1. A simplified test case is easier to communicate.
2. A simplified test case *usually* means smaller fixes.
3. A simplified test case *usually* means smaller program states.
4. A simplified test case *usually* means fewer program steps.

4 8. JUnit and Branch Coverage

Consider the simple class `Student` with a method that gets a `result` based on the `score`.

```
public class Student {
  private String name; private int score;
  public Student(String name, int score) {
    this.name = name; this.score = score;
  }
  public String result() {
    if (score > 100 || score < 0) { return "Not a valid input!"; }
    if (score >= 80) { return "Good"; }
    if (score < 80 && score >= 60) { return "Pass"; } else { return "Fail"; }
  }
}
```

Write a JUnit test class with several tests for `result`. All tests together should cover **all** branches for the `result` method, but each test should call `result` only once. Use boundary values as appropriate. Do **not** duplicate code among tests but use helper methods or `@Before`.

```
public class ResultTest {
  // [TODO] WRITE YOUR CODE BELOW









}
```

9. Reverse Engineering

4 (a) During Iteration 1, you had to make sure you can build and install Jenkins code for your project. This is an instance of the reverse engineering pattern called Do a Mock Installation from Chapter 2 of the book *Object-Oriented Reengineering Patterns (OORP)*. If you used another pattern from the book in your project, **describe that pattern and how you used it** in your project. If you did not use any other pattern in your project, then **name and describe any two patterns** that are **NOT** Do a Mock Installation.

2 (b) In the lecture on "Reverse Engineering", we discussed potential tools for reverse engineering. (Chapter 30 of Code Complete 2 also discusses tools in general.) Sometimes tools can help, e.g., a tool that visualizes relations between classes. However, sometimes you would rather do things manually. **Describe one scenario** when you would **NOT** use tools for reverse engineering.

10. Code Smells and Refactorings

2 (a) Refactorings change the structure of a program without changing its behavior. This goes against the maxim "if it ain't broken, don't fix it." When you change the code, you risk introducing new faults into the program. What is the main goal of refactoring?

2 (b) Which of these are code smells **NOT** based directly on code but rather on software development activities done with the code, such as changing code. (Circle **ALL** that apply.)

1. Non-localized plan
2. Empty commit messages
3. Refused bequest
4. Hard to understand code

11. Introduce Parameter Object

Consider the following class `GradeCenter`.

```
class GradeCenter {
  void printAllGrades(String netid) {
    printGrades(netid, true, true, true, true);
  }
  void printGrades(String netid,
      boolean mp0, boolean mp1, boolean mp2, boolean mp3) {
    saveGrades(System.out, netid, mp0, mp1, mp2, mp3);
  }
  void saveGrades(PrintStream stream, String netid,
      boolean mp0, boolean mp1, boolean mp2, boolean mp3) {
    stream.println("Printing grades for " + netid);
    if (mp0) {
      // print mp0 grades
    }
    ... // similarly for mp1, mp2, and mp3; code omitted for brevity
  }
  ...
}
```

The method `printGrades` selectively prints MP grades for a student based on the parameters. In the lectures, we talked about a refactoring called Introduce Parameter Object which is useful in situations like this.

1 (a) **Name the code smell** that the Introduce Parameter Object refactoring fixes.

4 (b) Write a new Java class `GradeSelection` that you could use as the parameter object. You should write **all fields and at least one constructor** for this class. You do not need to write getters, setters, or other methods. (Basically, write a simple Data Class.)

```
class GradeSelection {
  // [TODO] WRITE YOUR CODE BELOW









}
```

6 (c) Refactor all `GradeCenter` methods to use the new class `GradeSelection`. Make sure to **update the method signatures** and **modify the call sites** that use the parameter object. For `saveGrades`, rewrite **all the code** that was originally given. If you use some non-constructor methods from `GradeSelection`, comment on what those methods do.

```
class GradeCenter {
  // [TODO] WRITE YOUR CODE BELOW
  void printAllGrades









  // [TODO] WRITE YOUR CODE BELOW
  void printGrades









  // [TODO] WRITE YOUR CODE BELOW
  void saveGrades









}
```

[1] 12. Jenkins

When you graduate and join a software development organization, will you champion using (not necessarily extending) Jenkins or a similar continuous integration system? Describe why or why not.

13. Version Control

[2] (a) Briefly describe **two reasons** why one should not commit compiled code (such as `.class` files) to a version-control repository.

[2] (b) While it is generally not recommended to commit automatically generated files into version control, there are situations when it is useful. If you encountered such a situation in your project, **describe the situation and give one reason** why it was useful to commit such files. If you did not encounter such a situation in your project, **describe two situations** when it could be useful to commit such files.

[3] (c) In the MPs and course projects, we used SVN for version control. Typically, an SVN repository has three directories: `trunk`, `tags`, and `branches`. Answer the following questions according to the Software Configuration Management Patterns reference cards (from the assigned reading).

i. Which directory should have the *active development line*? (Circle **One** that applies.)
1. trunk 2. branches 3. tags

ii. Which directory should have the *task branch*? (Circle **One** that applies.)
1. trunk 2. branches 3. tags

iii. Which directory should have the *release line*? (Circle **One** that applies.)
1. trunk 2. branches 3. tags

14. Software Documentation

1 (a) One of the requirements for the Final Presentation was that your team document your project. Many other software systems also have some kind of documentation. Discuss when it is preferable to document an entire software system versus just the change from the previous system version.

2 (b) There are many types of system documentation. **Identify two types of documentation** and **describe both of them**.

2 15. Design Patterns

In the lectures we discussed the following **eight design patterns**: Observer, Composite, Interpreter, Visitor, Iterator, Template Method, Command, and Strategy. The book *Design Patterns* has 15 more patterns: Abstract Factory, Adaptor, Bridge, Builder, Chain of Responsibility, Decorator, Façade, Factory Method, Flyweight, Mediator, Memento, Prototype, Proxy, Singleton, and State.

Choose **one** of the 15 design patterns that we **did NOT discuss** in the lecture. Briefly describe this pattern, including its "Intent" (i.e., the reason for using this pattern) and one real example.

3 16. Iterator Design Pattern

In the code shown below, the class `SongCollection` stores a list of `SongInfo` objects.

Your aim is to execute some operation, e.g., `findOnYouTube(SongInfo s)`, on each song info in the song collection. One way to do that is using an **internal** iterator on `SongCollection`. We are providing you some classes that implement the operation and call the internal iterator method `SongCollection.apply(Command cmd)`. Write the body of that method.

```
public interface Command {
  public void execute(SongInfo s);
}

public class MyCommand implements Command {
  public void execute(SongInfo s) {
    findOnYouTube(s);
  }
}

public class Client {
  public static void main(String[] args) {
    SongCollection sc = new SongCollection();
    sc.addSong("Imagine", "John Lennon", 1971);
    sc.addSong("American Pie", "Don McLean", 1971);
    sc.addSong("I Will Survive", "Gloria Gaynor", 1979);
    Command cmd = new MyCommand();
    sc.apply(cmd);
  }
}
```

```
import java.util.LinkedList;
import java.util.List;
public class SongCollection {
  List<SongInfo> bestSongs;
  public SongCollection() {
    bestSongs = new LinkedList<SongInfo>();
  }
  public void addSong(String songName, String bandName, int yearReleased) {
    SongInfo songInfo = new SongInfo(songName, bandName, yearReleased);
    bestSongs.add(songInfo);
  }
  private Iterator<SongInfo> createIterator() {
    return bestSongs.iterator();
  }
  public void apply(Command cmd) {
    // [TODO] WRITE YOUR CODE BELOW









  }
}
```

5 17. Visitor Design Pattern

Suppose you start from this hierarchy for `Car` parts:

```
interface ICarElement {};
class Wheel implements ICarElement {};
class Body implements ICarElement {};
class Car implements ICarElement {};
```

If you want to add operations to this hierarchy, e.g., to print car parts, you would need to change all the classes for each new operation if you use straight polymorphism.

Using the Visitor design pattern, we need to change the hierarchy once and can then add new operations without modifying the car classes. Our goal is to add methods that print all parts of a given car using a visitor. We are providing you some changes to the original classes and a working visitor class `CarElementPrintVisitor` that prints the parts.

```
// this is changed to accept the visitor
interface ICarElement {
  void accept(ICarElementVisitor visitor);
}

interface ICarElementVisitor {
  void visit(Wheel wheel);
  void visit(Body body);
  void visit(Car car);
}

class CarElementPrintVisitor implements ICarElementVisitor {
  public void visit(Wheel wheel) {
    System.out.println("Printing " + wheel.getName() + " wheel");
  }

  public void visit(Body body) {
    System.out.println("Printing body");
  }

  public void visit(Car car) {
    System.out.println("Printing car");
  }
}

public class VisitorDemo {
  public static void main(String[] args) {
    ICarElement car = new Car();
    car.accept(new CarElementPrintVisitor());
  }
}
```

You should write the bodies of the `accept` methods.

```
class Wheel implements ICarElement {
  private String name;

  public Wheel(String name) { this.name = name; }
  public String getName() { return this.name; }

  @Override
  void accept(ICarElementVisitor visitor) {
    // [TODO] WRITE YOUR CODE BELOW









  }
}

class Body implements ICarElement {
  @Override
  void accept(ICarElementVisitor visitor) {
    // [TODO] WRITE YOUR CODE BELOW









  }
}

class Car implements ICarElement {
  ICarElement[] elmnts;

  public Car() {
    this.elmnts = new ICarElement[] {new Body(),
      new Wheel("front left"), new Wheel("front right"),
      new Wheel("back left") , new Wheel("back right")};
  }

  @Override
  void accept(ICarElementVisitor visitor) {
    // [TODO] WRITE YOUR CODE BELOW
    // VISIT "elmnts" BEFORE "this"
```

```
  }
}
```