University of Illinois at Urbana-Champaign
Department of Computer Science

Final Exam

CS 427: Software Engineering I
Fall 2015

December 16, 2015

TIME LIMIT = 3 Hours
COVER PAGE + 17 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 18 questions on this exam and the maximum grade on this exam is 95 points.

Page	Points	Score
1	11	
2	7	
3	5	
4	3	
5	6	
6	6	
7	8	
8	4	
Total:	50	

Page	Points	Score
9	5	
10	8	
12	6	
13	7	
14	7	
15	2	
16	6	
17	4	
Total:	45	

5 1. Warmup: Circle **T** if the statement is (mostly) true or **F** if the statement is (mostly) false.

1. **(T)** / [✓ **(F)**] The purpose of testing is only to find bugs.
2. **(T)** / [✓ **(F)**] Throughout your entire testing, you should stick stubbornly to the original test plan you made.
3. [✓ **(T)**] / **(F)** A good test plan should include configuration testing and documentation testing besides functional testing.
4. **(T)** / [✓ **(F)**] Usability problems are not considered bugs.
5. [✓ **(T)**] / **(F)** Smoke tests are a kind of fast regression tests.
6. [✓ **(T)**] / **(F)** Refactoring is a risk: it incurs a cost now, in return for potential payoff later.
7. [✓ **(T)**] / **(F)** Refactoring can be harmful to performance.
8. [✓ **(T)**] / **(F)** Refactorings can be combined into larger composite refactorings.
9. **(T)** / [✓ **(F)**] Refactoring is an important phase for test-driven development.
10. [✓ **(T)**] / **(F)** Cohesion refers to how closely all the operations in a module are related.

2. XP Project Management

2 (a) In CS427, you are encouraged to do pair programming for several MPs and the final project. According to the lecture slides, (1) what is pair programming and (2) what are the two roles in pair programming?

2 (b) Working in pairs can lead to some benefits over working alone. List **two** benefits of pair programming and briefly explain why you think working in pairs leads to each benefit.

3. Planning Game

2 (a) Velocity is used in XP to measure a team's rate of progress. Briefly explain (1) how velocity is calculated and (2) why calculating velocity is important in the planning game.

$\boxed{2}$ (b) The customer has a user story to create a feature for a mobile phone app and passes the user story to the developer to estimate the time cost. The developer evaluates the time cost based on her previous experience and estimates the user story to be too big for one iteration. The developer decides to divide the user story into three independent user stories that can each fit into an iteration, and prioritizes these three stories. Is this a correct way to do the planning game? If it is correct, list two key factors for dividing the user story as a developer; if not, explain why you think it is not correct.

4. Software Configuration Management (SCM)

$\boxed{1}$ (a) Our lectures on SCM discussed four aspects of SCM. List **two** of them.

$\boxed{2}$ (b) While working on your project, you were asked to follow the XP process and perform refactoring. We required you to create some separate commits for refactorings. Eventually, the code will look the same (whether you commit the refactorings separately, or you mix them with functionality changes). Why can it be in general beneficial to commit refactorings separately?

$\boxed{2}$ (c) In the second lecture about SCM, we defined a software product to be a set of components/documents. List **four** of these components.

2 (d) Alice and Bob are two CS427 students that CATME put in the same team for the final project. They paired up to work together during the final iteration but rather than actually working together, decided to work separately. Here is how they carry out the work:

Dec 4 (Fri): Alice and Bob run `svn up` each to update their local copies of the shared SVN repository that already contains some file `C.java`.
Dec 4 (Fri): Alice starts editing her local copy of the file `C.java` and finishes her task.
Dec 6 (Sun): Bob starts editing his local copy of the same file `C.java`. He works on it for a few hours but does not finish his task.
Dec 7 (Mon): Bob works more and finishes his task.
Dec 8 (Tue): An hour before the iteration meeting with the TA, both Alice and Bob are committing their changes, and they get a merge conflict.

Effectively, Alice and Bob were not working in parallel, so there was no need to get a merge conflict. What exact SVN commands could they have run to avoid the conflict? List the day/time when they should run each command, who should run it, and the exact command(s) to run.

5. Jenkins and Project

1 (a) Developers for Jenkins plugins can use `JenkinsRule` to write tests that check how their features interact with the Jenkins core. However, using `JenkinsRule` to create a Jenkins instance for each test is quite costly. Describe one way to test plugin interaction with the Jenkins core without this high cost.

2 (b) Jenkins is used as a continuous integration system for developers. List **two** benefits from using a continuous integration system.

6. Fault, Error, Failure

   In his guest lecture, Professor Tao Xie discussed how the word “bug” is ambiguous, and there are three more precise terms to use in its place depending on what needs to be described: fault, error, and failure. Consider the following Java method for binary searching an array of integers. (Hint: Pay close attention to the Javadoc for this method.)

```java
/**
 * This method searches for a value in a sorted array.
 *
 * @param a a non-null array of integers with no duplicates,
 *          sorted in ascending order
 * @param val the integer value to search for in the array
 * @return index of the search value, if it is contained in the array;
 *         otherwise, -1.
 */
public int binarySearch(int[] a, int val) {
  int lower = 0;
  // Hint: Java arrays are 0-indexed, so last element is at index (length - 1)
  int upper = a.length - 1;

  while (lower < upper) {
    int i = lower + (upper - lower) / 2;
    if (val == a[i]) {
      return i;
    } else if (val < a[i]) {
      upper = i - 1;
    } else {
      lower = i + 1;
    }
  }
  return -1;
}
```

2 (a) There is a **fault** in the code snippet. Describe the fault.

1 (b) If possible, write values for (1) the inputs `a` and `val` and (2) the expected output such that executing `binarySearch` leads to an **error** after executing this fault but does not lead to a **failure**. If it is not possible to write such values, explain why.

$\boxed{1}$ (c) If possible, write values for (1) the inputs `a` and `val` and (2) the expected output such that executing `binarySearch` leads to a **failure** after executing this fault but does not lead to an **error**. If it is not possible to write such values, explain why.

$\boxed{1}$ (d) If possible, write values for (1) the inputs `a` and `val` and (2) the expected output such that executing `binarySearch` leads to both an **error** and a **failure** after executing this fault. If it is not possible to write such values, explain why.

7. Testing And Coverage

$\boxed{2}$ (a) Alice is trying to write tests for her code. Part of her code does **streaming** (processing a sequence of elements), and she wants to test this functionality. Fortunately, the Testing Catalog (from the assigned reading) describes different kinds of streams she should write tests for. List **two** kinds of such streams. (Hint: "empty stream" is **not** an answer.)

$\boxed{2}$ (b) If you were the manager of the testing team at a company, would you allow testers to do some manual testing? If yes, why would you allow it? If no, why would you require automated testing all the time?

(c) The slides about testing described different types of coverage.

$\boxed{1}$ i. What is branch coverage?

$\boxed{1}$ ii. What is condition coverage?

$\boxed{2}$ iii. If some tests achieve 100% branch coverage of the code under test, do these tests always achieve 100% condition coverage? If yes, explain why. If no, construct a simple Java method and tests for the method that show this situation.

$\boxed{2}$ (d) One way to write black-box tests is to start from the specification, and then perform **equivalence class partitioning** over the input domain. What is equivalence class partitioning?

(e) Quiz 5 asked for a post-condition for the absolute value; many answers looked like this:

```
int abs(int n) {
  int out = n;
  if (n < 0) { out = -n; }
  assert out >= 0; // post-condition
  return out;
}
```

Suppose that you want to test this `abs` method using equivalence class partitioning.

2 i. What equivalence classes would you build for the input `n` based on its type (integer)?

2 ii. What are **all** the boundary values that should be selected for testing from those equivalence classes?

2 iii. The assertion (post-condition) is actually wrong and can fail for some input.

1. What input is it?

2. What is the name of the type of assertions that inform developers not to pass a wrong value to the method?

2 iv. The post-condition `out >= 0` is rather weak, e.g., if instead of returning `out`, the method just returned 1 regardless of the input, the given post-condition would still pass. Write a stronger post-condition that checks the correctness of the method.

```
int abs(int n) {
  int out = n;
  if (n < 0) { out = -n; }

  assert ________________________________________
  return out;
}
```

8. Regression Testing

$\boxed{2}$ (a) In the lecture on regression testing, we described the problems caused by the presence of **flaky tests** in a regression test suite.

i. Define flaky tests in terms of **faults** and **failures**.

ii. Given that regression testing is focused on testing the changes made to the code, how can flaky tests be harmful for regression testing?

$\boxed{2}$ (b) Regression test selection is a technique that selects to run a subset of tests in the regression test suite based on the changes between two versions of software. A lecture showed regression test selection by selecting tests that depend on the changed methods between two versions. Bob starts thinking about how regression test selection can speed up testing.

"To select what tests to run, we need to know what methods they depend on. To collect those dependencies, we need to run the tests. However, it seems we need to run *all* the tests every time to get these dependencies for each test! How can we save testing time with regression test selection if we have to rerun all the tests after every change?"

What is wrong with Bob's reasoning here (i.e., why does regression test selection need not rerun all the tests)?

9. Code Smells and Refactorings

$\boxed{1}$ (a) i. Select **one** of the following three code smells and describe how to identify that smell: *Non-localized plan*, *Refused Bequest*, *Data class*

$\boxed{2}$ ii. Describe what refactoring steps could eliminate the code smell selected from part (i)

$\boxed{1}$ (b) 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." Whenever you change the code, you risk introducing new faults into the program. What is the main goal of refactoring?

$\boxed{1}$ (c) Even if your code is ugly and smelly, sometimes you would not refactor it. Describe **one** reason for which, in general, a developer would make such a decision of **not** performing refactoring.

10. REMOVE DUPLICATED CODE
Consider these two classes that represent undergraduate and PhD students with a lot of duplicated code.

```java
abstract class Student {
}
class UndergradStudent extends Student {
  private int year;
  public UndergradStudent(int year) {
    assert year < 5;
    this.year = year;
  }
  public double calculateSleepHours() {
    return 8 - (year * 0.5);
  }
  public void printKnowledge() {
    System.out.println("I know everything.");
  }
}
class PhDStudent extends Student {
  private int year;
  public PhDStudent(int year) {
    assert year < 7;
    this.year = year;
  }
  public double calculateSleepHours() {
    return 8 - (year * 1);
  }
  public void printKnowledge() {
    System.out.println("I know nothing.");
  }
}
```

$\boxed{2}$ (a) Describe **one future scenario** where a change in the given code could be problematic if DUPLICATED CODE is not removed.

$\boxed{6}$ (b) On the next page, rewrite all **three** classes to remove **ALL** DUPLICATED CODE. Make sure to **write the entire classes, including the constructors, fields, and methods**. (Hint 1: In terms of refactorings, you are considering a combination of PULL UP METHOD and PULL UP FIELD.) (Hint 2: any duplication of method calls such as `println` should be eliminated as well, e.g., using the TEMPLATE METHOD pattern.)

```
abstract class Student {
 // TODO: Write your code below















}
class UndergradStudent extends Student {
 // TODO: Write your code below











}
class PhDStudent extends Student {
 // TODO: Write your code below











}
```

11. Metrics

$\boxed{2}$ (a) Some developers like to use Lines of Code as a metric, and some don't like it. Describe **one** way in which "Lines of Code" metric can be **useful** and **one** way in which it can be **useless** in a software development process.

$\boxed{2}$ (b) The software quality metrics of coupling and cohesion are based on characteristics of *good* programming practices that can reduce maintenance and modification costs.

i. Should a *well-designed* software system have low or high cohesion? Explain why.

ii. Should a *well-designed* software system have low or high coupling? Explain why.

12. Software Reuse

$\boxed{2}$ (a) List **one benefit** and **one drawback** of reusing software compared to building it from scratch.

13. Requirements Engineering and Risk Management

$\boxed{2}$ (a) The lectures mentioned a few questions that could appear on the final exam. Here is a question from the lecture on Requirements and Risks. How does XP handle incomplete requirements?

$\boxed{2}$ (b) In the lecture and readings about risk management, Risk Exposure (RE) is applied in Risk Prioritization. How do we calculate Risk Exposure (RE)? Write down the equation and briefly describe the meaning of each component in the equation.

14. Frameworks, components, libraries

$\boxed{2}$ (a) Does your plugin use the Jenkins core as a library or as a framework? Explain your answer.

$\boxed{1}$ (b) List **one** problem with frameworks according to the reading "Frameworks = (Components + Patterns)" (or "How frameworks compare to other object-oriented reuse techniques") by Ralph Johnson.

$\boxed{3}$ 15. Bug Advocacy

The purpose of bug advocacy is not just to report bugs but to get the bugs fixed. The slides "Black Box Software Testing: Bug Advocacy" by Cem Kaner and James Bach discuss more efficient bug advocacy and list five key challenges to consider when writing bug reports. List **three** of the five key challenges.

16. Debugging and Reverse Engineering

$\boxed{1}$ (a) Based on the debugging lecture, before encountering some failure in the future, which of these are good practices that developers should do now to make debugging easier in future? (Circle **ALL** that apply.)

1. Writing automated unit tests.
2. Writing highly coupled code.
3. Writing assertions.
4. Implementing methods with high cyclomatic complexity.
5. Using static analysis tools to detect/fix anti-patterns.
6. Pair programming.

$\boxed{1}$ (b) What is the difference between reverse engineering and reengineering?

$\boxed{2}$ (c) In the class project for CS427, you had the opportunity to apply some of the reverse engineering patterns described in the book "Object-Oriented Reengineering Patterns". Choose one of the book's reverse engineering patterns that your team used and describe how your team used the pattern.

17. Composite Design Pattern and Visitor Design Pattern

In MP3, you implemented some code that represents a library with collections and books, where a collection can hold both other collections and books. Suppose that Grainger Library wants to use your code but with an improvement for printing out the elements (collections and books). They want your code to print out all elements in the library such that each element name is printed on its own line, and the nesting is apparent using indentation (with size of **two** spaces). For example, if collection "computer science collection" contains book "cs1", collection "operating system collection", and book "cs2", and collection "operating system collection" contains book "os1" and book "os2", the expected console output is the following:

```
computer science collection
  cs1
  operating system collection
    os1
    os2
  cs2
```

(Hint: You can use `StringUtils.fill(' ', n)` that returns a string with `n` spaces.)

$\boxed{2}$ (a) Implement the required print functionality in classes following the **Composite Pattern**.

```
// assume that you have import java.util.*; in all classes
public abstract class Element {
  String name;
  public abstract void print(int indentation);
}

public class Book extends Element {
  public Book(String name) { this.name = name; }
  public void print(int indentation) {
  // TODO: Fill in this method body




  }
}

public class Collection extends Element {
  List<Element> elements;
  public Collection(String name, List<Element> elements) {
    this.name = name;
    this.elements = elements;
  }
  public void print(int indentation) {
  // TODO: Fill in this method body








  }
}
```

6 (b) Implement the required print functionality by following the **Visitor Pattern**. (Hint: As discussed in the lectures/slides on the Iterator Design Pattern, iteration of the collection elements can be done in either Collection or Visitor.)

```
// assume that you have import java.util.*; in all classes
public abstract class Element {
  String name;
  // TODO: Add the necessary method for the Visitor pattern




}
public class Book extends Element {
  public Book(String name) { this.name = name; }
  // TODO: Add the necessary method for the Visitor pattern




}
public class Collection extends Element {
  List<Element> elements;
  public Collection(String name, List<Element> elements) {
    this.name = name;
    this.elements = elements;
  }
  public List<Element> getElements() { return elements; }
  // TODO: Add the necessary method for the Visitor pattern




}

public abstract class ElementVisitor {
  public abstract void visit(Book book);
  public abstract void visit(Collection collection);
}
public class IndentedPrintVisitor extends ElementVisitor {
  private int indentation = 0;
  public void visit(Book book) { // TODO: Fill in this method body




  }
  public void visit(Collection collection) { // TODO: Fill in this method body











  }
}
```

$\boxed{1}$ (c) Assume Grainger Library extended this code with many kinds of printing (e.g., save to file or send via network). Which pattern (Composite or Visitor) would make it easier to add a new subclass of `Element` (e.g., magazine or newspaper) that supports all kinds of printing in this scenario? Explain why.

$\boxed{1}$ (d) Assume Grainger Library had many subclasses of `Element` (magazine, newspaper, etc.). Which pattern (Composite or Visitor) would make it easier to add a new kind of printing in this scenario? Explain why.

$\boxed{2}$ 18. Factory Pattern

In MP1 you had to write a class that implements a sequence. Suppose each sequence has two parameters: (1) the starting size and (2) the maximum size. Write two methods that each returns a `Sequence` object such that one method takes **only** the starting size (and the default maximum size should be `MAX_INT`) and the other method takes **only** the maximum size (and the default starting size should be `0`).

```
class Sequence {
  int starting;
  int maximum;
  private Sequence(int starting, int maximum) {
    this.starting = starting;
    this.maximum = maximum;
  }
  // TODO: Add the necessary method for instantiating objects of this class
  // with only the given starting size (and maximum is MAX_INT)







  // TODO: Add the necessary method for instantiating objects of this class
  // with only the given maximum size (and starting is 0)







}
```