University of Illinois at Urbana-Champaign
Department of Computer Science

Final Exam

CS 427 Software Engineering I
Fall 2008

December 16, 2008

TIME LIMIT = 3 hours
COVER PAGE + 18 PAGES

Upon receiving your exam, print your name and netid neatly in the space provided below; print 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 back 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 may get 0 points.

There are 23 questions on this exam and the maximum grade on this exam is **110 points**.

Page	Points	Score
1	12	
2	8	
3	8	
4	4	
5	9	
6	12	
7	10	
Total:	63	

Page	Points	Score
8	7	
9	5	
10	9	
11	9	
12	4	
15	10	
18	3	
Total:	47	

1. Answer true or false.

[1] (a) Automated testing is ideal for regression testing.

(a) _____

[1] (b) A program transformation is a refactoring only if you can automate the transformation.

(b) _____

[1] (c) Developers are always the best testers of their own code.

(c) _____

[1] (d) The only goal of testing is to maximize the number of bugs found.

(d) _____

[1] (e) Configuration management is another name for version control.

(e) _____

[1] (f) All classes that implement an application GUI should be abstract classes.

(f) _____

[1] (g) Object-oriented frameworks should have only concrete classes.

(g) _____

[1] (h) Number of Children is a metric used for object-oriented systems.

(h) _____

[1] (i) Testing does not need to consider usability problems.

(i) _____

[1] (j) *Shotgun Surgery* is when making a change in one class requires cascading changes in several other classes.

(j) _____

[2] 2. Which of the following correctly describe an XP spike? (circle all that apply)

I. A spike is another name for a milestone.

II. A spike helps to reduce uncertainty in an estimate.

III. A spike results in a complete analysis of a new technology to present to the client.

IV. A spike is a way to quickly try out an unfamiliar technology.

V. A spike is a short period of heightened activity following a change of requirements when the team must redo the estimates.

6 3. In MP5 you have seen and used many different automated refactorings. *Pull Up Field* is a refactoring that moves a field (instance variable) from a subclass to some superclass. Write a test input to test the *Pull Up Field* refactoring that involves **three** classes. Your test input should show a case where the refactoring **cannot** be performed i.e. the refactoring system will tell that the user that it cannot perform the refactoring because it will result in an invalid program.

The three classes must have a valid relationship with at least one other class, that is, you cannot have stand-alone classes. Follow what we did in MP5: provide your test input, indicate clearly which field you are performing the *Pull Up Field* refactoring on and explain why the refactoring cannot be performed.

2 4. Define the term *oracle* in the context of software testing.

4 5. In *Classic Testing Mistakes*, Brian Marick talks about the dangers of an over-reliance on beta testing. According to him, what are two reasons why, as a software developer, you shouldn't rely too much on your beta testers?

4 6. Is code coverage a good performance and quality measure of testing? Give one reason each for both sides of the argument.

7. Sometimes you need to write a test for an important feature but it appears to be too difficult or tricky to do. How would you design automated tests (not manual) for the following scenarios? In both cases you are not allowed to change the specifications but you can refactor your implementation so that it is more suited for automated tests.

2 (a) You need to implement a method with the signature `void printDetails()` that will print a String message to the console. How would you test that your printed message meets the requirements? ***Hint****: The issue here is that the return type of the printDetails() method is void; so you cannot easily get the String.*

2 (b) You need to implement a search dialog. On the dialog, there is a Find button that will initiate the search operation when it is clicked. How would you test that your search operation actually meets the requirements?

8. The book *Design Patterns* has 23 patterns: Abstract Factory, Adaptor, Bridge, Builder, Command, Chain of Responsibility, Composite, Decorator, Facade, Factory Method, Flyweight, Interpreter, Iterator, Mediator, Memento, Observer, Prototype, Proxy, Singleton, State, Strategy, Template Method, Visitor.

   Choose the appropriate pattern from the list to answer these questions.

[2] (a) Suppose you were making a spreadsheet. It supports very large spreadsheets, up to 10,000 by 10,000 cells. You decide that if one cell has a rule that uses another cell, you want a change to the used cell to automatically cause the first cell to recalculate. Which design pattern should you use?

[2] (b) Each cell can have a rule. Rules support all arithmetic operations plus lots of functions, such as trig functions. You could implement rules by translating them to machine language, but that seems like hard work and they don't really have to be fast. What design pattern could you use?

[2] (c) Suppose you wanted the spreadsheet to support undo. What design pattern could you use for this?

9. Design Patterns in Photran

[3] (a) One of the key parts of Photran is the ASTs (Abstract Syntax Trees). This data structure is a good example of the Composite Pattern. Explain.

4 (b) In what way is the Photran AST like the Interpreter pattern? In what way is it different?

4 (c) Visitor is one of the least frequently used patterns from Design Patterns, but it is used in Photran. What does Photran use it for? Why is Visitor a good fit for this purpose?

4 10. Describe how the documentation you write at the beginning of a project is **similar** to the documentation you write at the end. Also, describe how the documentation you write at the beginning of a project is **different** from the documentation you write at the end.

11. In the paper *A Rational Design Process: How and Why to Fake It*, David Parnas describes his ideas of a rational design process.

4 (a) According to Parnas, what are **two** reasons why it is not possible for software projects to achieve a fully rational process?

6 (b) Parnas claims: "Most programmers regard documentation as a necessary evil, written as an afterthought only because some bureaucrat requires it. They don't expect it to be useful". According to Parnas, what are **two** problems that plague documentation? For each problem that you list, explain whether using a wiki-approach for writing documentation would solve it. If you think that using a wiki would solve the problem, explain how. If you think that using a wiki would not solve the problem, explain why not.

[4] 12. Describe four different types of costs associated with reusing commercial off-the-shelf components.

[3] 13. Mention three differences between *frameworks* and *components*.

[3] 14. The leaders of commercial projects usually steer the direction of the project by providing and managing the requirements of the project. On the other hand, the leaders of open source projects steer the projects differently: as keepers of the version control system. So who actually provides the requirements in an open source project? And as keepers of the version control system, how do open source leaders steer the project?

[2] 15. In *The Cathedral and the Bazaar*, Eric S. Raymond says:

> In The Mythical Man-Month, Fred Brooks observed that programmer time is not fungible; adding developers to a late software project makes it later. As we've seen previously, he argued that the complexity and communication costs of a project rise with the square of the number of developers, while work done only rises linearly. Brooks's Law has been widely regarded as a truism. But we've examined in this essay a number of ways in which the process of open-source development falsifies the assumptions behind it–and, empirically, if Brooks's Law were the whole picture Linux would be impossible.

Do you agree or disagree with Eric Raymond that open-source development falsifies the assumptions behind Brooks's Law? Why?

[1] 16. Give one reason why Creative Commons is not recommended for software.

[1] 17. What does the MIT license have in common with the BSD, Apache and Eclipse licenses?

18. Coupling and cohesion are software quality attributes.

[4] (a) First define what we mean by "coupling". Then differentiate low coupling and high coupling.

[3] (b) Define cohesion and state the relationship between the degree of coupling and the degree of cohesion.

[4] 19. *A Pattern Language for Reverse Engineering*, presents reverse engineering patterns. Identify and briefly describe two patterns.

[3] 20. Should every software test be automated? Provide a cost-based justification of why some tests might be manual.

[2] 21. Which of the following is true about software testing? (Circle one)

A. Software testing should be exhaustive.

B. Software testing cannot prove the absence of bugs.

C. Dijkstra's *Humble Programmer* paper proved that software testing can overcome the halting problem.

D. Software tests must necessarily be more complex than the code being tested.

4 22. The following narrative describes one day of activities for a recently hired software developer named Victor. Identify two activities that are in line with XP and two that go against it, and state the value or principle demonstrated or violated by each.

> Victor had recently met with the client to gather requirements about some enhancements to an existing financial software system. He had tried to clarify as much as possible during the meeting, but when it was over, he still was not quite sure how everything fit together. So, he decided to start with something small that made sense, and use this as a building block to get clarification from the customer at the next regularly scheduled meeting. As he proceeded with development, he realized that the existing code was in need of some refactoring, but he was afraid to change the other team members' code, so he left it alone. Later, he had a few questions about the preferred approach to the data layer, so he called a quick, informal meeting to understand the team's approach so that he could continue. Toward the end of the day, Victor had lost track of time and realized that it was 3 p.m. and he had to leave to make an appointment. He had been working on the data layer and didn't have time for a final build of his code. He knows it is a good idea to check in his code to the repository, since that drive is backed up regularly, so he committed his changes and left for the day.

23. Object-Oriented Design and Refactoring

Read the entire question first before you begin so you have a better idea of what you need to do.

```
1  public class Movie {
2    private String title;
3    private int state;
4
5    private static final int AVAILABLE_STATE = 0;
6    private static final int RENTED_STATE = 1;
7    private static final int RETURNED_STATE = 2;
8
9    public Movie(String _title) {
10     title = _title;
11   }
12
13   public void rentMovie() {
14     if (state == AVAILABLE_STATE) {
15       state = RENTED_STATE;
16       rentMethodAction();
17     } else if (state == RENTED_STATE) {
18       // print some error message
19     } else if (state == RETURNED_STATE) {
20       // print some error message
21     }
22
23   }
24
25   public void returnMovie() {
26     if (state == AVAILABLE_STATE) {
27       // print some error message;
28     } else if (state == RENTED_STATE) {
29       state = RETURNED_STATE;
30       returnMovieAction();
31     } else if (state == RETURNED_STATE) {
32       // print some error message;
33     }
34
35   }
36
37   public void processMovie() {
38     if (state == AVAILABLE_STATE) {
39       // print some error message;
40     } else if (state == RENTED_STATE) {
41       // print some error message;
42     } else if (state == RETURNED_STATE) {
43       state = AVAILABLE_STATE;
44       processMovieAction();
45     }
46
47   }
48   ...
49 }
```

Listing 1: Movie class that implements our current requirements.

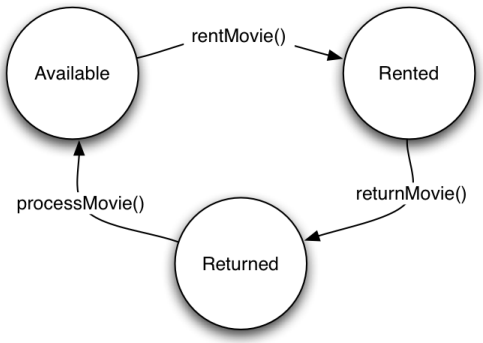

Figure 1: The states (and the transitions between them) for each `Movie` object.

Imagine that you are implementing a system to take care of movie rentals. You decide to create a `Movie` class to represent each item. Each `Movie` object can be in three different states as shown in Figure 1. And in each state, the `Movie` object can only respond correctly to certain messages (shown on the edges). Calling any other method on the `Movie` object will result in an appropriate error message for that method.

The "Available" state means that the movie is available for rent. The "Rented" state means that movie has been rented. The "Returned" state means that the movie has been returned and is awaiting processing before the next customer can rent it.

The relevant code snippets from your first implementation is shown in Listing 1. It is short and simple; but more importantly it works.

Then the owner of the movie rental company comes to you with the new requirements for your system. He has discovered that most customers don't return their movies on time. And he needs a way to keep track of overdue movies so he can charge those customers. So the two of you discuss the latest requirements and come up with the implementation shown in Figure 2.

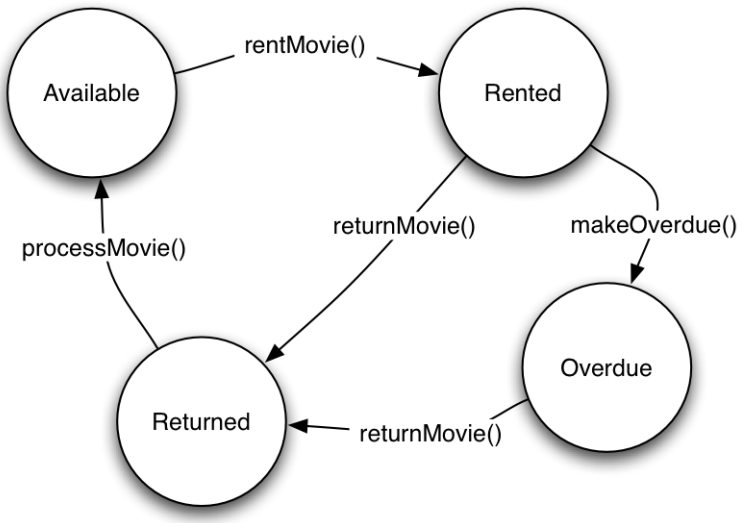

Figure 2: The **new** states (and the transitions between them) for each `Movie` object.

3 (a) You decide to augment your **existing** implementation to support the new requirements. In the space below, rewrite the `rentMovie()` method from **Listing 1** to support the new requirements.

2 (b) Having made that first change you realize that you are going to have to make those little changes throughout the entire `Movie` class. Suddenly you recall a lesson you learned while taking CS427 at UIUC: your design is suffering from a code smell! **Describe** the code smell that your current implementation suffers from.

5 (c) So you decide to encapsulate each state for the *Movie* object into its own class. Listing 2 shows the new `Movie` class, the `State` class and the `AvailableState` class.

```
1  public class Movie {
2    private String title;
3
4    float overdueFee;
5    State currentState;
6
7    State available;
8    State rented;
9    State overdue;
10   State returned;
11
12   public Movie(String _title) {
13     title = _title;
14     available = new AvailableState(this);
15     rented = new RentedState(this);
16     overdue = new OverdueState(this);
17     returned = new ReturnedState(this);
18
19     currentState = available; // starts of the movie in available state
```

```
20      overdueFee = 0.0f;
21    }
22
23    public void setCurrentState(State nextState) {currentState = nextState;}
24
25    public void rentMovie() {currentState.rentMovie();}
26
27    public void returnMovie() {currentState.returnMovie();}
28
29    public void makeOverdueMovie() {currentState.makeOverdueMovie();}
30
31    public void processMovie() {currentState.processMovie();}
32  }
33
34  public abstract class State {
35    protected Movie movie;
36
37    public void rentMovie() {/*print some default error message*/}
38
39    public void returnMovie() {/*print some default error message*/}
40
41    public void processMovie() {/*print some default error message*/}
42
43    public void makeOverdueMovie() {/*print some default error message*/}
44  }
45
46  public class AvailableState extends State {
47    public AvailableState(Movie _movie){ movie = _movie;}
48
49    public void rentMovie() {
50      rentMovieAction();
51      movie.setCurrentState(movie.rented);
52    }
53
54    public void returnMovie() {/*print errror message for AvailableState*/}
55
56    public void processMovie() {/*print errror message for AvailableState*/}
57
58    public void makeOverdueMovie() {/*print errror message for AvailableState*/}
59
60    ...
61  }
```

Listing 2: Movie class that implements our current requirements.

Study the implementation of refactored `Movie` class and the new `State` and `AvailableState` classes above. On the next page, write your implementation for the `RentedState` class following the **new** requirements (see Figure 2).

Extra space for question 23...

3 (d) Look at the new design. Imagine that the owner of the rental store comes to you with new states e.g. **JustOrdered** or **Lost** for the `Movie` object. **Describe** — you do **not** need to show the implementation — how you would add those new states using your new design.

By the way, congratulations, you have just implemented the State Pattern, one of the design patterns that we did not cover in class. Give yourself a pat on the back!