Florida Institute of Technology

Department of Computer Sciences

Comprehensive Exam (Spring 2010)

SOFTWARE ENGINEERING

Friday, March 19, 2010; 10:00am – 11:30am

Instructions – Read Carefully Before You Start:

- Write your student identification number in the space below.
- This exam consists of 17 pages (including this cover).
- **The exam consists of two parts:**
  - **You must answer Questions 1, 2, 3, and 4.**
  - **Answer any one (1) of Questions 5, 6, or 7**

    ➔ **Circle the one question that you want graded:**

    5 6 7

    (If you leave this blank, Question 5 will be graded.)
- Use a pen to write your answers in the space provided.
- When a question asks you to "describe," "discuss," or "explain" something, it means you must provide a convincing, clear, and reasonable answer; simply stating a fact without any supporting argument is insufficient.
- There are multiple parts to some of the questions. Read the questions carefully and answer all parts to the best of your ability.
- No study aids (notes, books, etc.) are permitted during the exam.

Good luck!

ID Number:

For Grading Use Only

	Question	Worth	Grade
1.	Process	20	
2.	Requirements	20	
3.	Design	20	
4.	Construction	20	
5.	Testing	20	
6.	Maintenance & Evolution	20	
7.	Management	20	
	Total	**100**	

1. Process (20%)

Process models are useful instruments to help software engineers manage large-scale projects. For example, the models can provide guidance in the context of improving software quality. Three of the software engineering process models commonly discussed are the waterfall model, the evolutionary model, and the spiral model.

Problem:

(a) Clearly explain each of these process models. Include diagrams of their phases.

(b) Describe 2 relative advantages and 2 relative disadvantages of each model.

(c) Under which circumstances would one choose the waterfall model over the other models? Clearly explain why. Provide a realistic example to support your answer.

Grading: (a) 9%: 6% explanation and 3% diagrams; (b) 6% (c) 5%

Note: Use the blank sheet of paper on the next page as needed.

Answer:

2. Requirements (20%)

Getting software requirements right is notoriously difficult. One of the main problems is getting everyone to agree to the same thing. In other words, developing a common understanding of the problem, from both a user (requirements definition) and an engineering (requirements specification) perspective. The goal of a requirement engineering process is to create and maintain a system requirements document. The overall process included three high-level requirements engineering sub-processes.

Problem:

(a) Describe the three main activities of the requirements engineering process and their corresponding products.

(b) Describe two techniques for representing requirements *specifications*. Include examples of each technique for a hypothetical system.

Grading: (a) 12%: 4% for each clear explanation; (b) 8%: 4% for each technique.

Note: Use the blank sheet of paper on the next page as needed.

Answer:

3. Design (20%)

Design is a key component of the overall software lifecycle. Good design contributes to the engineering of elegant and bug-free software.

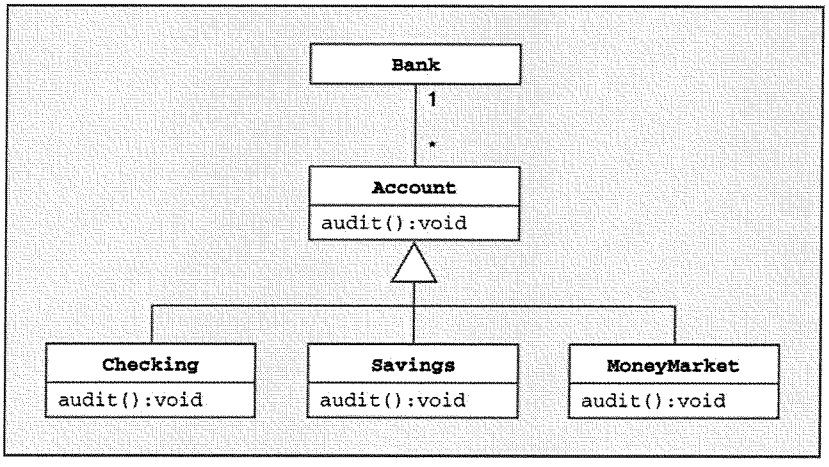
Problem:

(a) During high-level design, once the overall system organization has been chosen, one needs to make a decision on the approach to be used in decomposing sub-systems into modules. Sub-systems are composed of modules with defined interfaces that are used for communication with other subs-systems. A module is a lower-level artifact than a sub-system that is composed from a number of other simpler system components. Two common design approaches are object-oriented decomposition and function-oriented pipelining.

Describe *function-oriented pipelining*. Clearly identify the advantages and disadvantages of this approach to sub-system decomposition. Given an example of function-oriented pipelining for a hypothetical system's design. Explain why many legacy systems use this type of design.

(b) Several different notations for describing object-oriented designs were proposed in the 1980s and 1990s. The Unified Modeling Language (UML) is an integration of these notations. The UML is a graphical notation technique, not a design method or a process, that is used to communicate design intent.

Clearly explain this UML diagram. What type of UML diagram is it? Is it correct? Why or why not?



Grading: (a) 10%; (b) 10%;

Note: Use the blank sheet of paper on the next page as needed.

Answer:

4. Construction (20%)

The **uniq** program on UNIX removes duplicate lines from sorted data. Suppose, however, you need to remove duplicate lines from a data file (which might not be sorted), but that you wish to preserve the line ordering. A good example of this might be a shell history file. The history file keeps a copy of all the commands you have entered, and it is not unusual to repeat a command several times in a row, or several times per session. Occasionally you might wish to compact the history file by removing duplicate entries. Yet it is desirable to maintain the order of the original commands.

Problem: Construct an elegant and efficient program that implements this enhanced functionality. Document how your solution works, and the rationale behind your selection of algorithm(s) and data structure(s). Your program can be in C, C++, or Java.

Input: A text file containing the shell commands. Each line is terminated by the newline character (`'\n'`). Assume that the number of characters per line is usually less than 256. Assume that the number of lines in the input is usually less than 1,000. Assume that there are no blank lines in the input.

Output: A printout of the compressed history file in this format:

command # count

where *command* is the shell command, *count* is the number of times the command appeared in history file, and the '#' character separates the two (with a single space on either side of the '#'). Line ordering is important – it must be the same as the input.

Example: Given the following history file data as input:

```
ls
cat foo
date
ls
ls
cd
cd
ls
cd /tmp; ls
cd cs10/as1
g++ -Wall a2q2.cc -o a2q2
echo darn compiler
cd cs10/as1
cd cs10/as1
cd ~/cs10/a1
g++ -Wall a2q2.cc -o a2q2
a2q2
cd
cd
cd
```

Your program will produce exactly the following output:

```
ls # 4
cat foo # 1
date # 1
cd # 5
cd /tmp; ls # 1
cd cs10/as1 # 3
g++ -Wall a2q2.cc -o a2q2 # 2
echo darn compiler # 1
cd ~/cs10/a1 # 1
a2q2 # 1
```

Notes:

- Make sure your solution is constructed clearly and idiomatically, so that it adheres to the commonly accepted definition of good coding style.
- Be sure to properly comment your program; explain how the solution works and why you selected particular algorithm(s) and data structure(s).
- Provide citations to references you may have used in constructing your solution.
- Input to the program will come from standard input. Output must be to standard output. Do not prompt for input, nor produce spurious output.

Grading: Correctness: 15%; Documentation: 5%; Style: 5%

Note: Use the blank sheet of paper on the next page as needed.

Answer:

5. Testing (20%)

Black-box testing treats the system as a "black box" whose behavior is primarily determined by studying the inputs and the related outputs. A key problem for a tester is to select inputs that have a high probability of revealing the most amount of defects. Testers use their experience and domain knowledge of the system, along with a systematic approach to test data selection, to choose inputs to test the system.

Problem:

(a) Explain why it is practically impossible to exhaustively perform 100% testing of a program. Give an example.

(b) Explain equivalence partitioning to select test input data. Use your example from (a) to illustrate the use of this technique.

(c) What values are chosen for each partition in equivalence classes? Why?

(d) Compare and contrast verification and validation.

Grading: (a) 5%; (b) 8%: 5% explanation, 3% example; (c) 3%; (d) 4%.

Note: Use the blank sheet of paper on the next page as needed.

Answer:

6. Maintenance & Evolution (20%)

Maintenance is the act of modifying a program after system deployment. Following good software practice doesn't negate the need for maintenance—just its severity. In fact, maintenance is the most common form of software evolution.

Problem:

(a) Describe the three most common types of software maintenance. Be sure to provide examples of each type of maintenance using realistic scenarios.

(b) Discuss how software maintenance differs from software construction.

(c) Lehman's third law is the most interesting and perhaps most contentious law of evolution. It basically states that "Program evolution is a self-regulating process." Explain what this law means. Provide an example for a hypothetical system.

Grading: (a) 12%: 4% for each description; (b) 4%; (c) 4%.

Note: Use the blank sheet of paper on the next page as needed.

Answer:

7. Management (20%)

Estimating the cost and effort required for a particular task in a software project is an important management activity.

Problem:

(a) Describe two *classes* of productivity measures that are commonly used to aid cost and effort estimation. Note that in this context a *class* refers to a generic category of productivity measure, not to a specific type.

(b) Give specific examples of each class of productivity measure that are commonly used in practice. For each example, describe one reason why it is a good measure of estimating productivity and one reason why it is **not** a good measure.

Grading: (a) 10%: 5% for each class description; (b) 10%: 3% for each example (6%) and 1% for each reason (4%).

Note: Use the blank sheet of paper on the next page as needed.

Answer:

Software Engineering

Scott White

Christian Stocker