Midterm Exam
CS169, Fall 2004

November 10, 2004

- Please read all instructions (including these) carefully.
- **Write your name, login, and SID.**
- There are 8 pages in this exam and 6 questions, each with multiple parts. Some questions span multiple pages. All questions have some easy parts and some hard parts. If you get stuck on a question move on and come back to it later.
- You have 1 hour and 20 minutes to work on the exam.
- The exam is closed book, but you may refer to your two pages of handwritten notes.
- Please write your answers in the space provided on the exam, and clearly mark your solutions. You may use the backs of the exam pages as scratch paper. Please do not use any additional scratch paper.
- Solutions will be graded on correctness and clarity. Each problem has a relatively simple and straightforward solution. Partial solutions will be graded for partial credit.

LOGIN: _____

NAME: _____

SID: _____

Problem	Max points	Points
1	20	
2	10	
3	15	
4	15	
5	15	
6	25	
TOTAL	100	

1 Software Processes

(a) List the stages of the waterfall software process.

requirements gathering / analysis, specification, design, implementation / coding, testing, integration, usage / product testing / maintenance

(b) Describe briefly the difference between specifications and design.

specifications are a concise and complete description of user requirements ("what"); design is a technical plan for implementation of code which fulfills those specifications ("how").

(c) What is a risk of using the waterfall software process?

risk of not catching errors (in any stage) till too late; risk of requirements changing during development; risk of long waits before anything works.

(d) In extreme programming, what code do you write before you write a module? Explain why.

unit tests and testing infrastructure are written before the declarative code for each module. This focuses the programmer on meeting the specified functionality exactly, protecting him from unbounded abstraction. Unit tests are also clearly helpful in detecting failures later.

(e) What is the connection between extreme programming and refactoring?

Refactoring is central to XP, which emphasizes working, small-scope code and frequent iteration. such practices demand constant structural redesign, which is refactoring: reorganizing without changing functionality. Through this process the codebase is both flexible and robust.

2 Requirements and Specifications

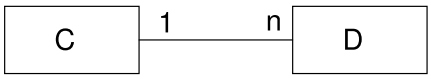
(a) What is one problem with informal specifications?

Informality usually implies ambiguity, which defeats their purpose as specifications. This problem is inherent in specifications written in natural language prose, so we frequently use more precise means of communication, such as graphs, in the specification process.

(b) What does the acronym UML stand for?

unified modeling language.

(c) Write to the right of each UML diagram below what relationship it describes between classes C and D.



each C is related to n Ds; each C contains two Ds; C is a superclass of D.

3 Debugging

(a) When debugging programs, do you turn compiler optimizations on or off? Why?

You first turn them off, because otherwise the debugger will not be able to correlate the executable with the source code. Then you turn them on, to find if the bug only appears in the optimized version (e.g., due to timing issues)

(b) In the delta debugging algorithm, why can you have unresolved tests?

There may be combinations of tests for which the program does not compile, or does not run.

For the rest of this problem, consider the use of the delta debugging algorithm for a set of 4 changes $\{A, B, C, D\}$, such that the test fails when all are present and the test succeeds when none is present. For a combination of changes, we write X when the test fails and $\surd$ when it succeeds.

(c) Given the following tests using the delta debugging algorithm, write all the possible minimal subsets of changes that reproduce the failure.

A	B			X
		C	D	$\surd$

The minimal subsets may be {A}, {B}, or {A, B}.

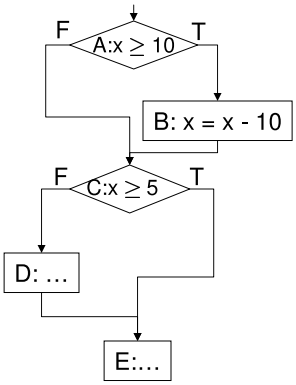
(d) Given the following tests using the delta debugging algorithm, write all the possible minimal subsets of changes that reproduce the failure.

A				$\surd$
	B	C	D	$\surd$
A	B		D	X

The minimal subsets may be {A, B}, {A, D}, or {A, B, B}. Note that we know that A is part of any minimal subset.

4 Testing

Consider the following program fragment, where A, B, C, D, E are names of the code blocks. For each of the following code coverage criteria, find the *minimum* number of runs required to obtain the required coverage. Give an example of the initial value of x for each run, and write also the sequence of blocks being executed.



a) Statement coverage

b) Edge coverage

c) Path coverage

- **Statement coverage:** One run is sufficient. A value of $x \in \{10, 11, 12, 13, 14\}$ will ensure that the path $ABCDE$ is followed.
- **Edge coverage:** Two runs are necessary and sufficient. One is the $ABCDE$ from above, and ACE (for $x \in \{5, 6, 7, 8, 9\}$). Another solution with two runs is $ABCE$ (for $x \geq 15$) and $ACDE$ (for $x \leq 4$).
- **Path coverage:** Four runs are necessary and sufficient (the four that have been mentioned above)

(d) Explain why 100% path coverage is not always sufficient to ensure absence of bugs.

There are many good answers here. One that is shorter than what we expected, is that testing cannot prove the absence of bugs. Also good answers, is that just because you have tried all the paths at least once, it does not mean that you have tried them with all input values. Many people also pointed out that in code with loops the bug may surface only in a late iteration of the loop.

5 Data Races

(a) Why are data races hard to debug?

Data races may manifest themselves only in rare interleaving of instructions, and the scheduler does not interleave the threads in exactly the same way every time (depending on the system, the system load, the needs and priorities of other processes that are running, etc). Because races are often rare, it is difficult to find in the first place, and difficult to track down (and to tell when you've fixed it) because you cannot reproduce it predictably.

(b) In a multi-threaded program do you have to worry about data races on local variables? Explain your answer.

No, you do not have to worry about data races on local variables. Local variables are not shared between threads, only global variables are. A local variable can be a pointer to (shared) global data, but in this case it is the global data that needs to be locked to protect from data races not the local pointer (all access to that shared data need to use the same global lock).

(c) Consider the following sequence of actions taken by one thread. Fill in the two columns corresponding to the locksets of `x` and `y` inferred by Eraser. For each action, write the lockset inferred after seeing that action. Consider only the basic Eraser algorithm (the one that does not handle initialization and does not distinguish between read and write locks).

	x	y
initial	{ a, b}	{ a, b}
1. lock(a)		
2. y = 0		{ a }
3. lock(b)		
4. x = y	{a, b}	{ a }
5. unlock(a)		
6. x = y + 1	{ b }	{}
7. unlock(b)		
8. lock(a)		
9. x = 1	{}	{}
10. unlock(a)		

(d) Where in the above program does Eraser warn about data races, and for which variables?

In line 6 about variable y, and in line 9 about variable x

6 Short Answers

(a) Name and describe briefly one design pattern that we learned about.

example: Bridge Pattern: Make an abstract hierarchy that uses other interfaces internally (such as a specialized window management hierarchy that can be used as a layer on top of other window managers). Construct one class (hierarchy) for each external interface that you want to support (such as one for each external window manager) where all of them show same interface to the user. The bridge often does work internally so that its functionality is greater than the intersection of the functionalities of the supported external interfaces.

(b) The usage of the following C macro looks like a function call. Show an actual invocation where the behavior is different than a function call.

```
#define square(x) ((x) * (x))
```

This macro does not work if the argument has side-effects: square(x++), square(foo()), ...

(c) Explain what does it mean for a static analysis to be conservative? Give an example on which type checking is conservative.

Static analysis is conservative in that it produces false positives. For example, static type checking will complain about the following:

```
char x; if(!y) x = "string";
```

even when it is guaranteed in your program that y is never equal to 0 (or false).

(d) Write below some advantages of run-time monitoring and static analyses.

Advantages of run-time monitoring	Advantages of static analyses
does not require source code (and is language-agnostic), can find errors that are caused by the environment (and thus can only be found by running the code or by extremely conservative static analysis), can be used by programmer to easily check for an invariant or suspected bug (e.g. asserts), does not give false positives about paths that are not possible in the program or about assignments that are not explicitly casted (but probably should be) but that the programmer knows are correct in the range of values used by the program	checks all possible execution paths even if they are not covered by a testcase, does not degrade performance at runtime, does not require running the program (halting problem, program might run for a long time, etc), enables higher-level understanding of program (possible to have knowledge of past, present, and future instead of just past and present)

(e) What is regression testing and how do you use it effectively?

Regression testing is a testing strategy where every time you find a bug you write a test case to exhibit the bug, fix the bug, and add the test case to your test suite. Ideally you run this entire test suite regularly on the program as it changes (at CVS checkin, at every build, etc). This way you ensure that old bugs do not reappear without you noticing (which happens frequently)