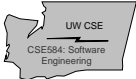


## CSE584: Software Engineering

### Lecture 9 (June 2, 1997)

---

David Notkin  
 Dept. of Computer Science & Engineering  
 University of Washington  
[www.cs.washington.edu/homes/notkin](http://www.cs.washington.edu/homes/notkin)




Notkin (c) 1997 1

## Lecture 9, Outline

---

- ◆ Testing
- ◆ Break
- ◆ PSP, CMM, ISO 9000
- ◆ Hot topics in software engineering research
- ◆ Course evaluations




Notkin (c) 1997 2

## Testing

---

- ◆ Testing is the process of executing programs to improve their quality
  - This clearly contrasts with proofs of correctness and static analysis (like LCLint, type checking, etc.) in which the analysis is performed on the program text
- ◆ There are other forms of testing, such as usability testing, that are quite different




Notkin (c) 1997 3

## Confidence

---

- ◆ Dijkstra observed a long time ago that testing cannot show that a program is correct, testing can only show that a program is incorrect
- ◆ This is accurate, but largely immaterial
  - The objective is to build confidence, even in safety-critical applications
  - A more important question is, “Can testing be made more rigorous?”




Notkin (c) 1997 4

## A question

---

- ◆ Is model checking more like proofs of program correctness or more like testing?




Notkin (c) 1997 5

## Kinds of testing

---

◆ Symbolic testing	◆ Black-box testing
◆ Mutation testing	◆ Boundary testing
◆ Functional testing	◆ Cause-effect testing
◆ Algebraic testing	◆ Regression testing
◆ Random testing	◆ ... more ...?
◆ Data-flow testing	
◆ Integration testing	
◆ White-box testing	



Notkin (c) 1997 6

## Other definitions

- ◆ A *failure* occurs when the program acts in a way inconsistent with its specification
- ◆ A *fault* occurs when an internal system state is incorrect (even if it doesn't lead to a failure)
- ◆ A *defect* is the piece of code that led to a fault (and then usually a failure)
- ◆ An *error* is the human mistake that led to the defect



Notkin (c) 1997

7

## Test cases

- ◆ A test case succeeds if it reveals a defect in a program
  - Test cases used to succeed if they executed as expected
- ◆ But test cases that “fail” help improve confidence
  - Many test cases are chosen because they are characteristic of a collection of real executions of the program



Notkin (c) 1997

8

## Challenges of testing

- ◆ Producing effective test sets
- ◆ Producing reasonably small test sets
- ◆ Testing both normal and off-normal cases
- ◆ Testing for different classes of users
- ◆ Testing for different SW environments
- ◆ Testing for different HW environments
- ◆ Tracking results over time
- ◆ ... more ... ?



Notkin (c) 1997

9

## White- vs. black-box testing

- ◆ A common dichotomy for testing is white-box vs. black-box testing
- ◆ In white-box testing, the tester also sees the code
  - A key question is, “What code is covered?”
  - Often done earlier
- ◆ In black-box testing, the tester sees the specification but not the code
  - The primary question is, “Does the code satisfy the specification for specific test cases?”
  - Often done later



Notkin (c) 1997

10

## Black-box testing

- ◆ Incorrect functions
- ◆ Missing functions
- ◆ Interface errors
- ◆ Performance problems
- ◆ Initialization and shutdown errors
- ◆ Can be done at the system level and/or at the module level



Notkin (c) 1997

11

## Black-box testing challenges I

- ◆ What classes of input provide representative coverage?
- ◆ Is the system particularly sensitive to certain input values?
- ◆ How are boundaries of the system tested?
- ◆ How do we produce the appropriate oracle?
  - That is, how do we know the “right” answer



Notkin (c) 1997

12

## Black-box testing challenges II

- ◆ How do we efficiently compare true output to the expected output?
- ◆ What data rates and data volume can the system handle?
- ◆ What effect will specific combinations of operations and data have on system operation?



Notkin (c) 1997

13

## Coverage criteria [Ghezzi, Jazayeri, Mandrioli]

- ◆ In black-box testing one (consciously or sub-consciously) partitions the inputs into a set of classes
  - Again, the expectation is that a single test case will identify common defects for that class
- ◆ There are formal definitions of test selection criteria and of consistent and complete criteria



Notkin (c) 1997

14

## Test selection criterion

- ◆ A test selection criterion specifies a condition that must be satisfied by a test set
  - Ex: Over integers, there must be positive, negative, and zero test values
- ◆ There are (almost) always multiple test sets that satisfy a given criterion



Notkin (c) 1997

15

## Consistent and complete

- ◆ A consistent criterion is one for which any two test sets that satisfy the criterion, either both sets succeed or both sets fail on the program
- ◆ A complete criterion is one for which, if the program is incorrect, there exists a satisfying test set that demonstrates this
- ◆ Having a consistent and complete criterion would guarantee finding errors in a program



Notkin (c) 1997

16

## But

- ◆ There is no way to guarantee consistency and completeness
  - In general, there is no way to compute whether a criterion is consistent or complete
- ◆ So we tend to use informal or heuristic approaches to approach consistency and completeness



Notkin (c) 1997

17

## Syntax-driven testing

- ◆ In some situations, the possible inputs to a program are characterized using a formal grammar
  - Ex: Compilers, simple user interfaces, etc.
- ◆ In these cases, one can generate test sets such that each grammar rule is used at least once



Notkin (c) 1997

18

### Decision table testing

- Describe tabularly how different inputs, in various combinations, can produce different outputs
- Coverage of individual rows is needed
- But tables may be huge

P	*								
B		*							*
I			*						*
E				*	*				
SE					*	*	*	*	
E=B			*						
E=I				*					
SE=B					*				
SE=I							*		
SE=B+I								*	
action	p	b	i	b	i	b	i	b,i	b,i

UW CSE CSE584 Software Engineering Notkin (c) 1997 19

### Cause effect graphing

- Build boolean functions that capture specific transformations captured by the program
- Constraints can be defined to pare the input space
  - Ex: B and P are mutually exclusive

UW CSE CSE584 Software Engineering Notkin (c) 1997 20

### White-box testing

- A central objective of white-box testing is to increase *coverage*
  - That is, ensure that as much code in the program as possible is exercised
  - The theory is that any code that is exercised by no test case is likely to have defects
- The actual output may, at times, be of secondary interest
- For large systems, effective white-box testing requires tool support

UW CSE CSE584 Software Engineering Notkin (c) 1997 21

### Statement coverage

- The simplest notion of coverage is to ensure that all (as many as possible) statements are exercised
- Problem: what's a statement?
  - Solution: represent program as a control flow graph and ensure all statement nodes are executed
- Problem:
  - if  $x > y$  then  $max := x$  else  $max := y$

UW CSE CSE584 Software Engineering Notkin (c) 1997 22

### Program and its CFG

- if  $x > 0$  then  $x := -x$ ; endif
- The test set  $\{x = 1\}$  exercises all nodes (statements)

UW CSE CSE584 Software Engineering Notkin (c) 1997 23


### Edge coverage

- To eliminate the obvious problems with statement coverage, one can require that all edges of the program's CFG be exercised
  - Now a test set like  $\{x = 1, x = -1\}$  is needed
- Edge coverage is always at least as good as statement coverage
  - That is, any test set that satisfies edge coverage for a program will also satisfy statement coverage

UW CSE CSE584 Software Engineering Notkin (c) 1997 24

### Condition coverage I


- ◆ A weakness in edge coverage arises with compound conditionals
  - if  $x \geq 0$  and  $x \leq 1000$  then
    - S1
  - else
    - S2
- endif
- ◆ A test set of  $\{x = 10, x = 1997\}$  will satisfy edge coverage



Notkin (c) 1997 25

### Condition coverage II

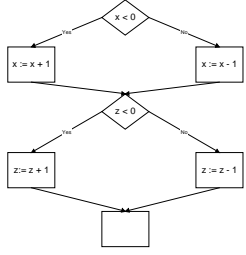

- ◆ Instead, one can require that all combinations of the compounds be tested
  - However, this may not be feasible in all situations; some combinations may never arise
- ◆ An alternative is to require edge coverage plus a requirement that each clause in the condition independently take on all values



Notkin (c) 1997 26

### Condition coverage III


- ◆ A weakness with edge and condition coverage is that combinations of control flow aren't checked
- ◆ This example is covered with  $\{x=-1, z=-1; x=1, z=1\}$

Notkin (c) 1997 27

### A brief aside


- ◆ A key problem in coverage testing of any sort arises when one learns that specific elements are not covered by your test set
- ◆ How do you create a new test case that covers some or all of those unexercised elements?
- ◆ I don't know of any research that addresses this, although there may be some
  - Some work in program slicing might help



Notkin (c) 1997 28

### Path coverage

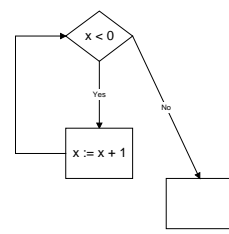

- ◆ Path coverage requires that all paths through the control flow graph be exercised
- ◆ For the last example, we'd need four cases
  - $\{x=-1, z=-1; x=1, z=1; x=-1, z=1; x=1, z=-1\}$
- ◆ The problem with path coverage is that loops are intractable
  - It's generally impossible to ensure that all possible paths are taken in a program with a loop
- ◆ Also, not all paths are feasible



Notkin (c) 1997 29

### Loops with path coverage


- ◆ The path taken by  $x = -10$  is different from the path taken by  $x = -20$
- ◆ All paths cannot be tested, so representative ones must be used
  - Boundaries
  - "Average"

Notkin (c) 1997 30

### Data flow approaches

- ◆ The coverage approaches shown so far rely on control flow information
- ◆ Rapps and Weyuker (1985) suggested using data flow information for coverage instead
  - Basic idea uses def-use graphs
  - Coverage of variable definitions (essentially, assignments) and uses considered




Notkin (c) 1997 31

### Example (x<sup>y</sup>)

```

1. scanf(x,y); if (y < 0)
2.   pow = -y;
3.   else pow = y;
4.   z = 1.0;
5.   while (pow != 0)
6.     {z=z*x;pow--;}
7.   if (y < 0)
8.     z = 1.0/z;
9.   printf(z);
    
```

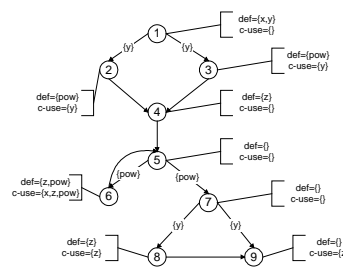

- ◆ *def* is an assignment to a variable
- ◆ *c-use* is a computational use of a variable
- ◆ *p-use* is a predicate use of a variable



Notkin (c) 1997 32

### Flow graph

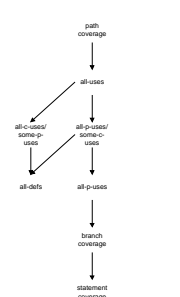

- ◆ There are many alternative criteria
  - all-defs
  - all-p-uses
  - all-uses
- ◆ Could require O(N<sup>2</sup>) in 2-way branches, but empirically it's linear
- ◆ Need to find test cases

Notkin (c) 1997 33

### Relationships among approaches [RW]


- ◆ Frankl & Weyuker have shown empirically that none of these do significantly better than the others
- ◆ They also showed that branch coverage and all-uses perform better than random test case selection

Notkin (c) 1997 34

### Mutation testing [Demillo, Lipton, et al.]


- ◆ The idea here is to take a program, bring a variant (mutant) of it, and compare how the program and the variant execute
- ◆ The objective is to find test cases that distinguish between the program and its mutants
  - Otherwise, the test cases (or the mutation approach) are weak



Notkin (c) 1997 35

### Example [Jalote]

- ◆ Consider the “program”
  - a := b\*(c-d)
  - written in a language with five arithmetic operators {+, -, \*, /, \*\*}
- ◆ There are eight “first order” mutants of this program
  - Four operators can replace \* and four can replace -



Notkin (c) 1997 36

## Other kinds of mutations

- ◆ Replacing constants
- ◆ Replacing names of variables
- ◆ Replacing conditions
- ◆ ...



Notkin (c) 1997

37

## Process

- ◆ Define a set of test cases for a program
  - Test until no more defects are found
- ◆ Produce mutants
  - Test these mutants with the same test set as the base program
  - Score “dead” vs. “live” mutants (ones that are and are not distinguished from the original program)
  - Add test cases until there are no dead mutants



Notkin (c) 1997

38

## Utility?

- ◆ There are some questions about whether mutation testing is sensible
  - Does it really help improve test sets?
  - The evidence is murky
- ◆ There are also performance questions
  - If not automated, it’s a lot of management
  - Computation of the mutants and applying the tests to the mutants can be very costly



Notkin (c) 1997

39

## Open questions include

- ◆ Minimizing test sets
- ◆ Testing OO programs
- ◆ Incremental re-testing
  - “Cheap” regression testing
- ◆ Balancing static analysis with testing
  - Can some properties be “proven” using this combination?
- ◆ ... more ... ?



Notkin (c) 1997

40

## Software process

- ◆ Capability maturity model (CMM), ISO 9000, Personal Software Process (PSP), ...
- ◆ These are all examples of approaches to improving software quality through a focus on software process and software process improvement
  - Relatively little focus on the technical issues of software



Notkin (c) 1997

41

## A little history

- ◆ The waterfall model, etc., have been considered since the late 1950’s/early 1960’s
- ◆ Incremental development models, the spiral model (Boehm), and others arose as refinements




Notkin (c) 1997

42

### 1987


- ◆ At the 1987 International Conference on Software Engineering (ICSE), Lee Osterweil presented a paper titled, "Software processes are programs, too"
  - Essentially, this said that one could and should represent software processes explicitly
  - Allowing one to "enact" processes as part of an environment
- ◆ Highly controversial, including a response by Manny Lehman



Notkin (c) 1997 43

### Why controversial?


- ◆ Importance of technical issues and decisions w.r.t. managerial issues and decisions?
- ◆ Prescriptive processes vs. descriptive processes?
- ◆ Capturing processes as programs?



Notkin (c) 1997 44

### In any case...

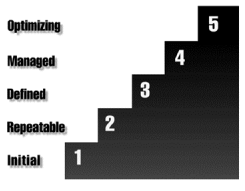

- ◆ This led to an enormous increase in
  - industrial interest, and
  - research in software process
- ◆ Software process workshops
- ◆ More recently
  - A journal or two
  - A number of conferences
  - Lots of papers in general software engineering conferences
- ◆ "Most influential paper of ICSE 9"



Notkin (c) 1997 45

### CMM (SEI's web page)


- ◆ "The Software CMM has become a de facto standard for assessing and improving software processes.
- ◆ Through the SW-CMM, the SEI and community have put in place an effective means for modeling, defining, and measuring the maturity of the processes used by software professionals."

Notkin (c) 1997 46

### CMM (Levels 1 and 2)


- ◆ Initial. The software process is characterized as ad hoc, and occasionally even chaotic. Few processes are defined, and success depends on individual effort and heroics.
- ◆ Repeatable. Basic project management processes are established to track cost, schedule, and functionality. The necessary process discipline is in place to repeat earlier successes on projects with similar applications.



Notkin (c) 1997 47

### CMM (Level 3)

- ◆ Defined. The software process for both management and engineering activities is documented, standardized, and integrated into a standard software process for the organization. All projects use an approved, tailored version of the organization's standard software process for developing and maintaining software.



Notkin (c) 1997 48



## CMM (Levels 4 and 5)

- ◆ Managed. Detailed measures of the software process and product quality are collected. Both the software process and products are quantitatively understood and controlled.
- ◆ Optimizing. Continuous process improvement is enabled by quantitative feedback from the process and from piloting innovative ideas and technologies.



Notkin (c) 1997

49

## My opinion(s)

- ◆ For some organizations, moving upwards at the very low levels is sensible
- ◆ The focus on process improvement is inherently good
- ◆ The details of the actual levels are not especially material for most organizations
- ◆ Technical issues are still downplayed too much



Notkin (c) 1997

50

## CMM mania

- ◆ SW CMM
- ◆ People CMM
- ◆ Systems engineering CMM
- ◆ Integrated product development CMM
- ◆ Software acquisition CMM
- ◆ CMM integration



Notkin (c) 1997

51

## ISO 9000



Notkin (c) 1997

52

## Hot software engineering topics

- ◆ Recent ICSE session titles
  - Exploiting the Internet, formal specifications, reliability, inspections & reviews, user interface & specifications, legacy systems & testing, static analysis, metrics, process, hardware/software issues, reverse engineering, process improvement, economic & legal issues, OOT, testing



Notkin (c) 1997

53

## Recent NSF grants I

- ◆ Reasoning about open systems
- ◆ Formalisms for requirements analysis and design
- ◆ Theoretical underpinnings of formal analysis of concurrent systems
- ◆ Formal reasoning about reactive systems
- ◆ Unifying real-time design and implementation
- ◆ A framework for specifying and verifying generic system components
- ◆ Empirical investigations of software inspections




Notkin (c) 1997

54

### Recent NSF grants II

---

- ◆ Testing OOP
- ◆ Exploratory programming techniques for program execution monitoring
- ◆ Reflexion models
- ◆ Scalable static techniques for exhaustive and incremental analyses of C systems
- ◆ And more: inspections & reviews, process-based environments, VPL, software architecture, program restructuring, evolution of persistent ADTs, and many more




Notkin (c) 1997 55

### My top few

---


- ◆ Lightweight tools and analyses
- ◆ Relationship between compiler-based and software engineering analyses
- ◆ Software archaeology



Notkin (c) 1997 56

### Your top few?

---



Notkin (c) 1997 57