CSE584: Software Engineering

Lecture 10 (December 8, 1998)

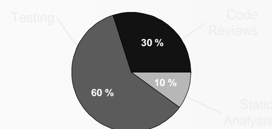
David Notkin
Dept. of Computer Science & Engineering
University of Washington
www.cs.washington.edu/education/courses/584/CurrentQtr/

Software quality assurance

- Broad topic
  - Testing
  - Static analysis
  - Reviews and inspections
  - Software process
  - Reliability
  - Safety
- High-level overview of some of these areas
  - Specific domains have added techniques, etc.

Key (unanswered) question?

- Many problems with software engineering research arise because of a weak understanding of time and money constraints
- This holds for much SQA research
  - *Given a fixed resource (dollars, time) how do you allocate SQA activities?*



Reviews, etc.

- Reviews, walkthroughs, and inspections are all in a family of activities where an artifact (specification, code, etc.) is studied by a peer group to improve the artifact's quality
- There is a large and increasing literature that demonstrates the effectiveness (although not always the cost-effectiveness) of these approaches
  - Some evidence of cost-effectiveness may be available, although it's hard to generalize

Reviews, etc.

- N-heads are better than one
- Intended to
  - identify defects
  - identify needed improvements
  - encourage uniformity and conformance to standards
  - enforce subjective rules

Purposes

- Increase quality through peer review
- Provide management visibility
- Encourage preparation

- Explicit non-purpose
  - Assessment of individual abilities for promotion, pay increases, ranking, etc.
  - Management often (usually) not permitted at reviews

Walkthrough

- A formal activity
- A programmer (designer) presents a program (design)
- Values of sample data are traced
- Peers evaluate technical aspects of the design

Inspections

- Formal approach to code review
- Intended explicitly for defect detection (not correction)
- Defects include logical errors, anomalies in the code (such as uninitialized variables), non-complicance with standards, etc.

Inspection requirements

- A precise specification must be available
- Peers must be knowledgeable about organizational standards
- Code should be syntactically correct and basic tests passed
- Error checklist must be provided

Inspection process

- Plan
- Overview
- Individual preparation
  - Code, documentation distributed in advance
- Meeting
- Rework
- Follow-up

Inspection teams

- Four or more members
- Author of code
- Reader of code (reads to team)
- Inspector of code
- Moderator chairs meeting, takes notes, etc.

Inspection checklists

- Checklist of common errors drives inspection
- Checklist dependent on programming language
  - Weaker type systems usually imply longer checklists
- Examples
  - Initialization, loop termination, array bounds, ...

Inspection rate

- 500 statements/hour during overview
- 125 statements/hour during individual prep
- 90-125 statements/hour during review
- Inspecting 500 statements can take 40 person-hours
  - For 1MLOC, this would be about 40 person-years of effort

Issues in inspections

- Can groupware technology significantly improve inspections?
- Can you have inspections without meetings?
  - Since meetings are expensive to hold and schedule
  - Since the preparation may catch more defects than the meetings

SQA Statistical approaches

- There are a number of approaches to quality assurance that are (in varying senses) based on statistics
  - Software reliability
  - N-version programming
  - Cleanroom

Software reliability [RST]

- The probability that software will provide failure-free operation in a fixed environment for a fixed interval of time
  - A system might have reliability 0.96 when used for a one week period by an expert user
- Mean-time-to-failure is the average interval of time between failures
- One common use of software reliability models is to decide when it's OK to ship a product

Operational profiles

- A sufficiently accurate operational profile is needed
  - Frequency of application of specific operations for the program being studied
- An operational profile is the probability density function (over the entire input space) that best represents how the inputs would be selected during the life-time of the software

Understood domains

- In industries such as telecommunications, operational profiles can be fairly easily gathered
- The phone company has records of virtually every call made in the last 20 years
  - Phones are used in pretty consistent ways

Less understood domains

- But for shrink-wrapped software products, operational profiles are harder to divine
- How will different users using different products with different features behave?
  - CPA's vs. college students using a spreadsheet?
- Will usage change over time?
  - More or less than the phone system?

Cost

- To assess reliabilities past the 3rd or 4th decimal place can require an enormous amount of testing
  - Is it necessary to do so?
- Should all failures be considered equally bad?
  - Showstoppers vs. "wrong color"
- Oracles of "correctness" aren't always easy
- Monitoring phone switches is relatively easy; monitoring shrinkwrap isn't

Applying reliability models

- There is extensive real use of models in this style
- There is also a ton of theoretical work that is never validated
  - Variants on models never compared to reality
- There are courses, books, etc. about how to apply reliability modeling in practice

N-version programming

- The idea of N-version (multi-version) programming comes from a common hardware reliability approach--replication
- The basic notion is simple
  - Have N independent teams write N versions of a program
  - Run them all simultaneously and have them vote at specified points

Objective

- Since the programs are built independently, the objective is to improve the quality by a multiplicative factor
  - A bug only hurts if it also appears in another (N/2)+1 versions
  - This idea indeed works pretty well in hardware
- The cost issue in software is different, though
  - Not a matter of producing and testing multiple chips, but of producing multiple implementations

Assumption

- But there is an underlying assumption at work
  - The implementations will fail independently
    - Like the chips in hardware that fail based on physical structures
  - Otherwise, a multiplicative factor will not be gained
- Do independently built implementations of the same specification fail independently?

Probably not

- Knight and Leveson did some experiments that showed that this assumption is probably false
  - In particular, they showed that similar errors often arise in independently implemented versions of the same specifications
- An additive benefit may arise from N-version programming, but not a multiplicative one

Why?

- Errors are often in the specification
- Errors are often made at boundary conditions
- The complexity of a program is often in a small piece or two, which each group has trouble with
- The background and training of people in an organization are often similar

And now...

- N-version advocates are still out there in an aggressive way
- There are some experiments showing independence
- There are attempts to introduce variety explicitly
  - Different specs, different languages, etc.
- I'm still completely opposed to this approach based on the Knight/Leveson experiments

Cleanroom [Harlan Mills]

- Cleanroom combines managerial and technical activities into a process intended to lead to very high quality software
  - Combines formal methods with statistical testing for reliability with incremental development
  - Does not allow unit execution or testing
- Effectiveness is a controversial issue

Basics: five points

- Formal specification
  - "Required system behavior and architecture"
  - Black box stimulus-response specification
- Incremental development
  - Partitioned into "user-function increments" that "accumulate into the final product"
- Structured programming
  - Limited use of control and data abstraction constructs; stepwise refinement of specification

Basics: five points (con't)

- Static verification
  - Components statically verified using mathematic correctness arguments
  - Individual components neither executed nor tested
    - No white box testing, no black box testing, no coverage analysis
- Statistical testing
  - Each increment is tested statistically based on operational profile

Three teams

- Specification team
- Development team
  - Codes
  - Statically verifies using inspections
- Certification team
  - Develops and applies statistical tests
  - Reliability models used to decide when to ship

Claims

- Very aggressive positive claims
  - About 20-30 systems (all under 500KLOC)
- 100KLOC systems with (well) under 10 errors in the field in the first year or two
- Finds 1-4 errors/KLOC during statistical testing
- Some projects claim 70% improvement in development productivity

Counterclaims [Beizer]

- Several (related) questions raised
  - Are comparisons to other methods fair?
  - Why eliminate unit testing?
  - Why trust software reliability modeling so much?
    - Especially hard to get good operational models
- Claim is that unless Cleanroom embraces modern testing approaches, it will fail to be used broadly

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

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?"

Kinds of testing

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

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

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

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 … ?

White- vs. black-box testing

- A common dichotomy for testing is white-box vs. black-box testing
- In white-box, the tester sees the code
  - A key question is, "What code is covered?"
  - Often done earlier
- In black-box, 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

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

Black-box 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

Black-box 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?

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

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

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

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

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
- This is a good example where entirely random tests are essentially useless

White-box testing

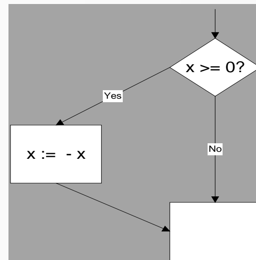
- A central objective of white-box testing is to increase coverage
  - That is, ensure that as much code in the program as possible is exercised
  - The theory is that any code that is exercised by no test case is likely to have defects
- The actual output may, at times, be of secondary interest
- For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

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`

Program and its CFG

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



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

Condition coverage I

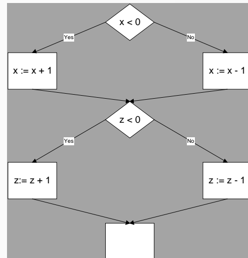
- A weakness in edge coverage arises with compound conditionals
  - ```
    if x >= 0 and x <= 1000 then
         S1
    else
         S2
    endif
    ```
- A test set of {x = 10,x = 1997} will satisfy edge coverage

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

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}`



A brief aside

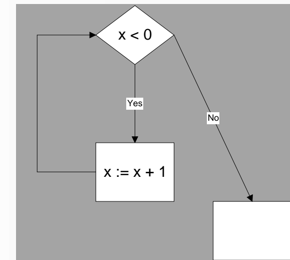
- A key problem in coverage testing of any sort arises when one learns that specific elements are not covered by your test set
- How do you create a new test case that covers some or all of those unexercised elements?
- I don't know of much research that addresses this, although there may be some
  -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

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}`
- A 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

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"



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

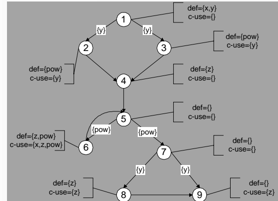
Example (xy)

- *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

```
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);
```

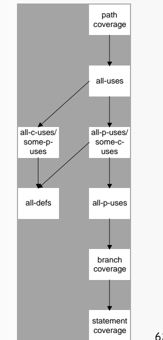
Flow graph

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



Relationships among approaches

- 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



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

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 `-`

Other kinds of mutations

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

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

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

Open testing questions

- 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 … ?

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

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

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

Why controversial?

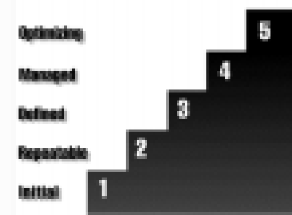
- Importance of technical issues and decisions w.r.t. managerial issues and decisions?
- Prescriptive processes vs. descriptive processes?
- Capturing processes as programs?

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"

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."



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.

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.

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.

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 far too much

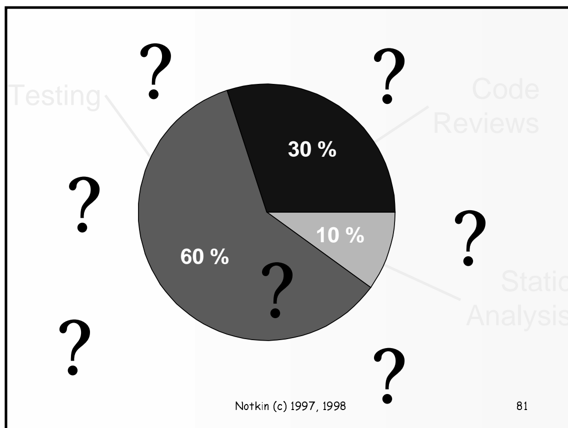
CMM mania

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

Notkin (c) 1997, 1998 79

ISO 9000

Notkin (c) 1997, 1998 80



Notkin (c) 1997, 1998 81

Last words

- Lots of software engineering research misses the mark w.r.t. industrial practice
- But lots contributes, too: both directly and by laying a foundation for the future
  - Research papers and prototypes are our "products", and they have many of the same pressures as your projects
    - Evaluate the research "alphas" & "betas" in that light
- Practitioners have some responsibility in helping to guide and refine research in software engineering

Notkin (c) 1997, 1998 82