Class 6 CS540

Gregg Vesonder
Stevens Institute of Technology
© 2005 Gregg Vesonder

Roadmap- Class 6

- Clarifications from last class
- Log Book volunteer
- Review of test
- Configuration Management
- Testing
- Reading: BY Chapter 11
- Reading next class: Brooks Chapters 7, 8, 9, 10 and BY 5, pp 223-247

Clarifications

- Thought Problem

Thought Problem

- Your company does not have a Quality program, lately there have been some issues with software Quality and your Director has asked you to institute a program for your company. What is your plan?

Calendar -Key Dates

- November 7th - second test
- November 21st - log books due
- December 12th - final exam

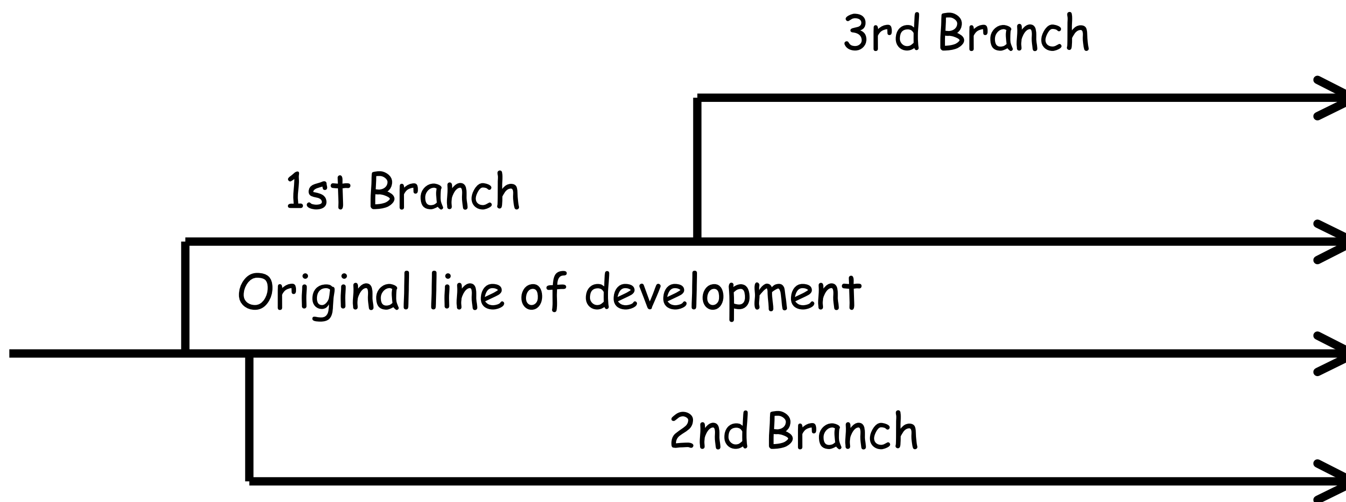
Logbook

- Your Entry
- Succession Planning

Configuration Management the Problem

- Not a simple task!
  - Different versions of software usually is in the field during the life cycle
  - Different parts of the team are on different versions of the software and documents (see branching)
  - The same release of a software product may have multiple versions consisting of different combinations of software components
- Configuration management is both a development and production issue, a life time issue

Branches of Development



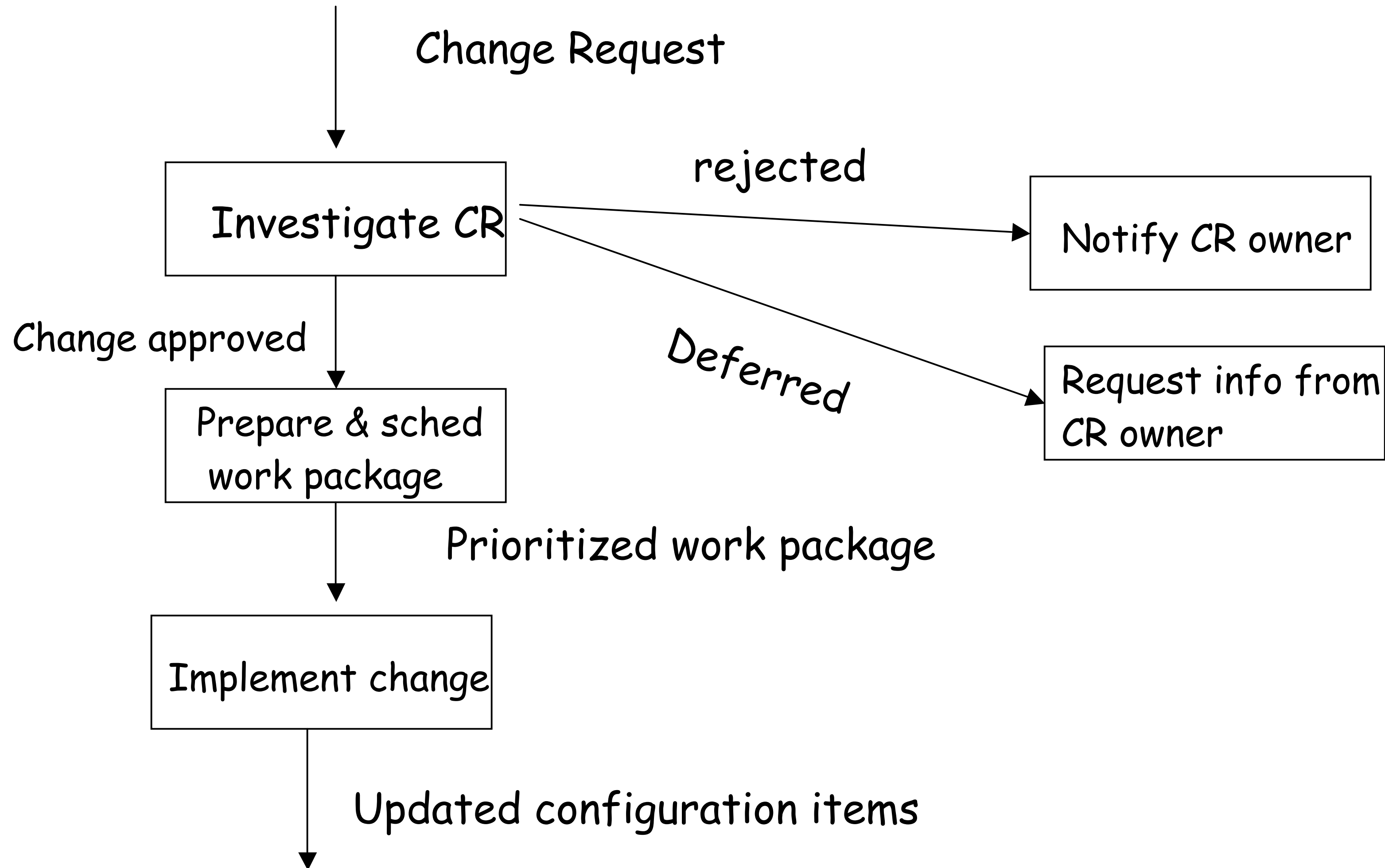
From Collins-Sussman, et.al., 2005

Time

The Baseline

- IEEE - “reviewed and agreed upon basis for further development which can be changed only through formal control procedures”
- Contained in the baseline are configuration items: source, objects, requirements
- Configuration management maintains integrity of these artifacts
- Major error- retrace steps through code, design documents and requirements specification -TRACEABILITY

Workflow of CR (MR)



Configuration Management Tools

- Manage the workflow of CRs
- If item is to be changed, developer checks it out and item is locked to other users
- When item check back in revision history is stored
- All versions are recoverable
- Should be able to accommodate branching - necessary more times than you think!
- Configuration management tools are very sophisticated, keeps only the changes, the deltas and the remarks, timestamps and who did what - essential for Buildmeister and testers
- New tools are change oriented release configuration is identified by a baseline plus a set of changes.

Configuration Management Plan

- Main parts:
  - Management:
    - How project is organized
  - Activities:
    - Who is on CCB, what are their responsibilities
    - What reports are required
    - What data is collected and archived - IMPORTANT
  - Schedules
  - Resources
  - Plan Management

Software Testing

- Testing is the last bastion of Quality - you can not “test in” Quality however testing is a necessary but not sufficient condition for Quality
- Dijkstra “Testing can show the preesence of bugs but not their absence!”
- The Quality of the systems we deliver increasingly determine the Quality of our existence
- Good testing is at least as difficult as good design (with asymmetric rewards)

When to Test

- NOW
- Postponing testing *for too long* is a severe mistake.
- Boehm- errors discovered in the operational phase incur cost 10 to 90 times higher than design phase
  - Over 60% of the errors were introduced during design
  - 2/3's of these not discovered until operations
- Given care you can test requirements specification, design and design specification
  - Also prototypes, story boards and even macromedia demos test aspects of the spec, arch and design
- Which testing strategy? Finding errors or confidence in functioning of software?

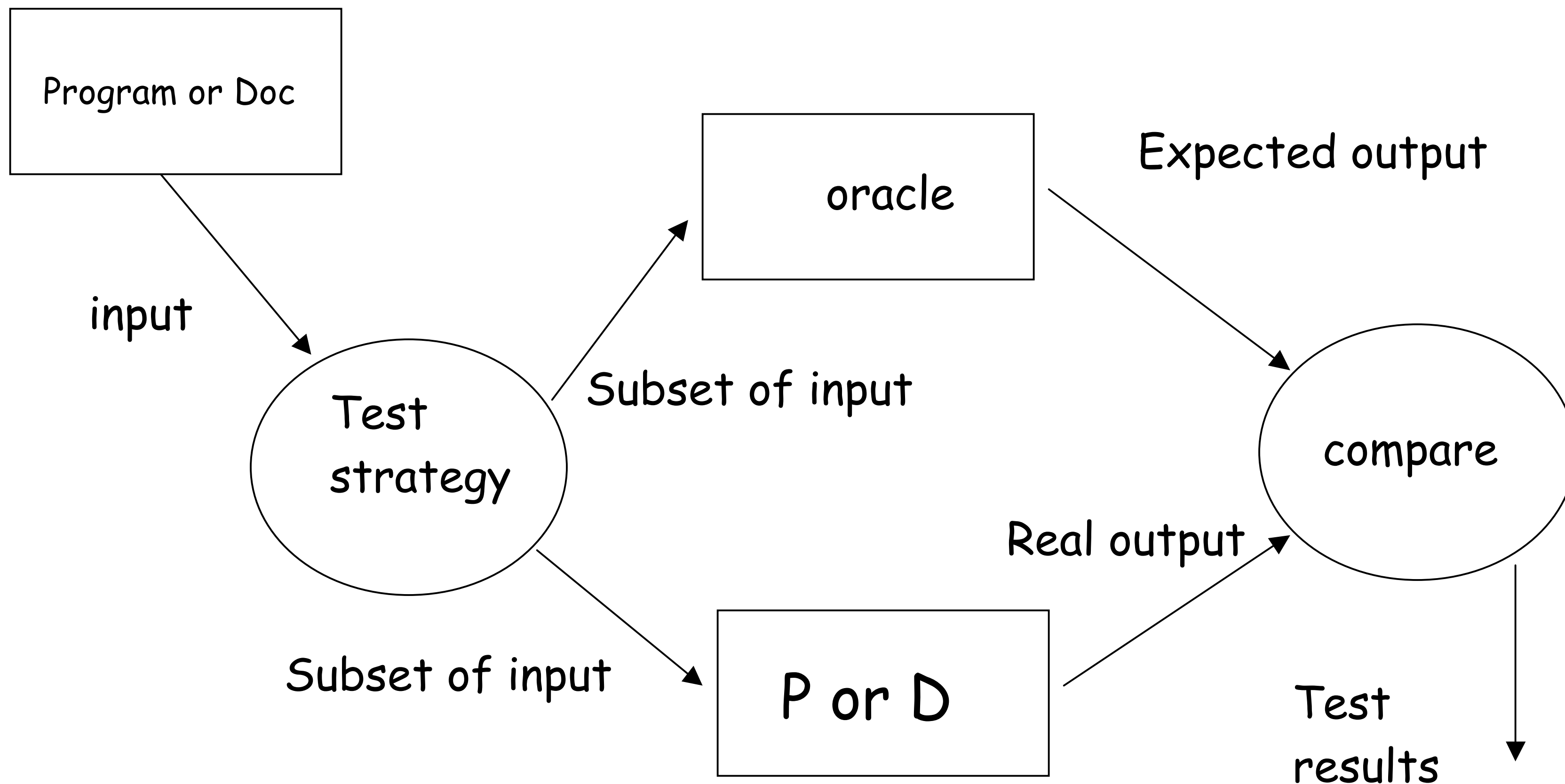
Types of Testing

- Coverage based - coverage of product, e.g., all statements must be executed at least once
- Fault based- detect faults, artificially seed and determine whether tests get at least X% of the faults
- Error based - focus on typical errors such as boundary values (off by 1) or max elements in list
- Black box - function, specification based,test cases derived from specification - system testing
- White box - structure, program based, testing considering internal logical structure of the software - unit testing

Testing Vocabulary

- Error - human action producing incorrect result
- Fault is a manifestation of an error
- Failure - sometime encountering a fault causes a failure, hard to define failure,it is relative and we must be aware of the standard
- “Due to an error by Gregg a fault was introduced in the software and when the fault was encountered, it caused the current failure.”
- Verification “The process of evaluating a system or component to determine whether the products of a given development phase satisfy conditions imposed at the start of the phase” e.g., ensure software correctly implements a certain function- have we built the system right
- Validation “The process of evaluating a system or component during or at the end of development process to determine whether it satisfies specified requirements”, e.g., software built traceable to customer requirements - have we built the right system

Global View of Test Process from van Vliet



Test Adequacy Criteria

- Critical to select subset of input domain that will be the test set
- Test techniques generally use a systematic way to generate test cases - each fault is not equally hazardous.
- Test adequacy criteria specify requirements for testing, e.g., stopping rule, measurement, test case generator - all closely linked to techniques

Fault Detection vs. Confidence Building

- Tension: intention is to provoke failure behavior - a good strategy for fault detection but does not inspire confidence
- User wants failure free behavior - high reliability
  - Frequently manifesting faults cause more damage (or workarounds)
  - Mimic the situation through random testing of scenarios

Cleanroom Techniques

- Developer cannot execute code - convinced of correctness through manual techniques
- These modules are integrated and tested by someone else using input that was generated to follow the distribution of actual use - goal is to achieve a given reliability level

Fault Detection to Fault Prevention

- Historical progression, in early days testing and debugging

MODEL	GOAL
Phase: -Demonstration -Destruction	 Software satisfies spec Detect implementation faults
Life Cycle: -Evaluation -Prevention	 Detect R, D & I faults Prevent R, D & I faults

Phase Models

- Demonstration - if it runs test set, it is good, purpose to convince someone there are no errors - dangerous
- Proper testing is destructive, you want to find errors.
  - Find as many faults as possible, look for test cases that reveal faults
  - Difficult to decide when to stop testing
    - When budget is exhausted or time runs out?
    - When all test cases pass
  - Usually has a systematic way to develop test cases

Lifecycle Models

- Evaluation oriented - emphasis on detecting faults in evaluation and design
- Prevention oriented - early design of test cases, careful planning and design of test activities
- Over years we are moving from demonstration to prevention
- BUT testing is still concentrated late in the development cycle (move to the left, NOT!)
- Testing is not only about errors but also about knowledge

Requirements Engineering

- Review or inspection to check that all aspects of the system have been described
  - Scenarios with prospective users resulting in functional tests
- Boehm's criteria for functional specification: consistency, completeness, feasibility, testability -- testing a requirements specification test these criteria
- Common errors in a specification:
  - Missing information
  - Wrong information
  - Extra information
- During requirements testing phase, testing strategy for other phases is generated: test techniques, plan, scheme and documentation

Boehm's Criteria

- Completeness- all components present and described completely - nothing pending
- Consistent- components do not conflict and specification does not conflict with external specifications --internal and external consistency. Each component must be traceable
- Feasibility- benefits must outweigh cost, risk analysis (safety-robotics)
- Testable - the system does what's described
- Beginnings of ICED-T

Traceability Tables

- Features - requirements relate to observable system/product features
- Source - source for each requirement
- Dependency - relation of requirements to each other
- Subsystem - requirements by subsystem
- Interface requirements relation to internal and external interfaces
- Part of a requirements database, how a change in a requirement affects aspects of the system

Traceability Table: Pressman

SUBSYSTEM

REQUIREMENTS

	S01	S02	S03…
R01	X		
R02	X		X
R03…		X	

Testing and Design

- Similar criteria to requirements
- Documentation standards help in this process (see previous tables)
- With refinement tests should become more detailed
- Test for the future in architecture/high level design (remember SARB goals)
  - scenarios for anticipated change
- Test design
  - Tracing back to requirements
  - Simulation
  - Design walk throughs and inspections

Testing and Implementation

- "real" testing, some techniques:
  - Read the code to find errors
  - Walk throughs -- Inspections
  - Stepwise abstraction - what does the code do
  - Static tools - inspect code without execution
  - Dynamic - run the code
  - Test for correctness through formal verification

Testing and Maintenance

- More than 50 % of the time spent in maintenance
- Modification causes another round of tests - regression tests
  - Library of previous test plus adding more (especially if the fix was for a fault not uncovered by previous tests)
  - Issue is whether to retest all vs selective retest, expense related decision (and state of the architecture design related decision -- if entropy is setting in you better test)

V&V Planning and Documentation

- Where test activities are planned
- IEEE 1012 specifies what should be in Test Plan
- Test Design Document specifies for each software feature the details of the test approach and lists the associated tests
- Test Case Document lists inputs, expected outputs and execution conditions
- Test Procedure Document lists the sequence of action in the testing process
- Test Report states what happened
- In smaller projects many of these can be combined

IEEE 1012

1. Purpose
2. Referenced Documents
3. Definitions
4. V&V overview
   1. Organization
   2. Master schedule
   3. Resources summary
   4. Responsibilities
   5. Tools, techniques and methodologies
5. Life cycle V&V
   1. Management of V&V
   2. Requirements phase V&V
   3. Design phase V&V
   4. Implementation V&V
   5. Test phase V&V
   6. Installation and checkout phase V&V
   7. O&M V&V
6. Software V&V Reporting
7. V&V admin procedures
   1. Anomaly reporting and resolution
   2. Task iteration policy
   3. Deviation policy
   4. Control procedures
   5. Standard practices and conventions

Manual Test Techniques

- Reading - peer review, insight via best and worst technique (2 good, 2 marginal, can developers detect marginal code)
- Walkthroughs and Inspections *
- Scenario Based Evaluation (SAAM)*
- Correctness Proofs
- Stepwise Abstraction from code to spec

Inspections

- Sometimes referred to as Fagan inspections
- Basically a team of about 4 folks examines code statement by statement
  - Code is read before meeting
  - Meeting is run by a moderator
  - 2 inspectors or readers paraphrase code
  - Author is silent observer
  - Code analyzed using checklist of faults: wrongful use of data, declaration, computation, relational expressions, control flow, interfaces
- Results in problems identified that author corrects and moderator reinspects
- Maintain constructive attitude - not used in programmer's assessment

Walk Throughs

- Guided reading of code using test data to run a "simulation"
- Generally less formal
- Learning situation for new developers
- Parnas advocates a review with specialized roles where the roles define questions asked - proven to be very effective

The Value of Inspections/Walk-Thrus

(Humphrey 1989)

- Inspections are up to 20 times more efficient than testing
- Code reading detects twice as many defects/hour as testing
- 80% of development errors were found by inspections
- Inspections resulted in a 10x reduction in cost of finding errors

SAAM

- Software Architecture Analysis Method
- Scenarios that describe both current and future behavior
- Classify the scenarios by whether current architecture directly (full support) or indirectly supports it
- Develop a list of changes to architecture/high level design - if semantically different scenarios require a change in the same component, this may indicate flaws in the architecture
  - Cohesion glue that keeps modules together - low=bad
    - Functional cohesion all components contribute to the single function of that module
    - Data cohesion - encapsulate abstract data types
  - Coupling strength of inter module connections, loosely coupled modules are easier to comprehend and adapt, low=good
- Overall evaluation is produced

Coverage Based Techniques

- Adequacy of testing based on coverage, percent statements executed, percent functional requirements tested
- All paths coverage is an exhaustive testing of code
- Control flow coverage:
  - All nodes coverage, all statements coverage recall Cyclomatic complexity graphs
  - All edge coverage or branch coverage, all branches chosen at least once
  - Multiple condition coverage or extended branch coverage covers all combinations of elementary predicates
  - Cyclomatic number criterion tests all linearly independent paths

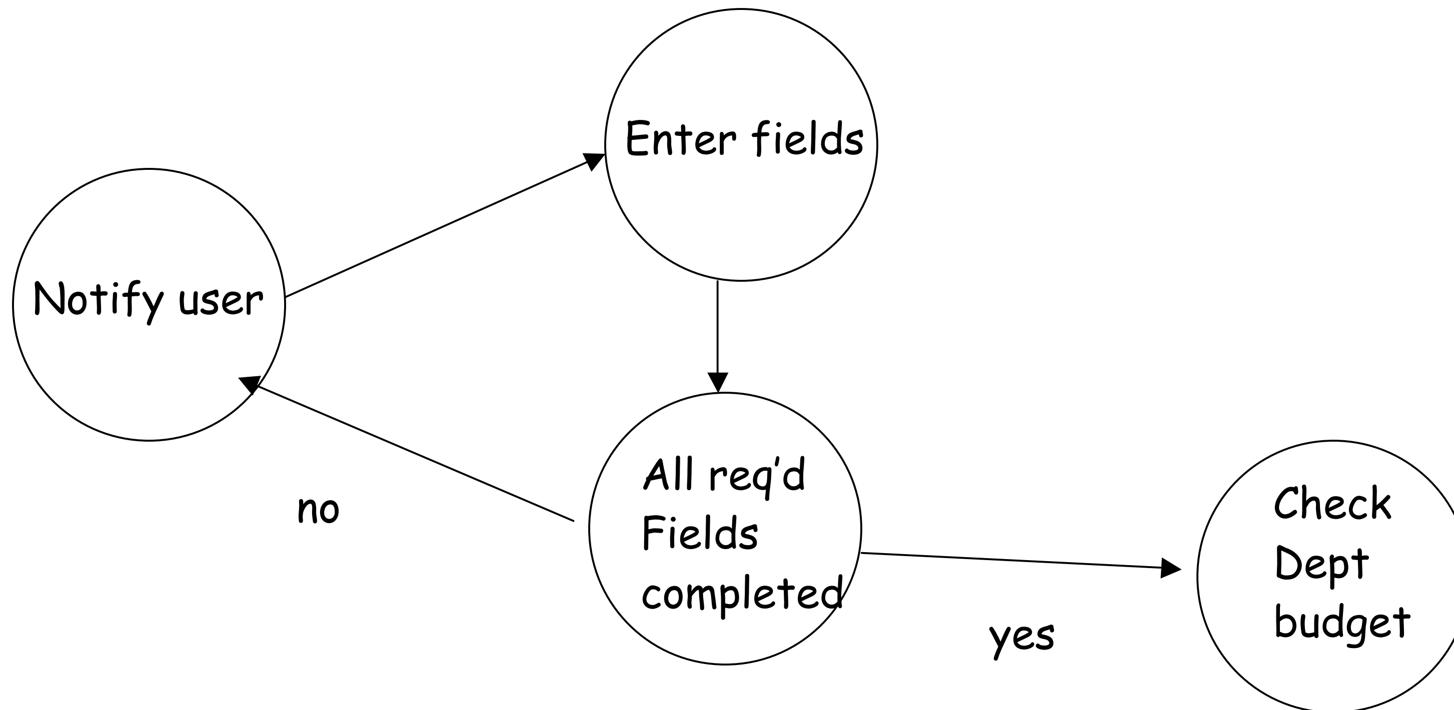
Coverage Based Techniques -2

- Data Flow Coverage - considers definitions and use of variables
  - A variable is defined if it is assigned a value in a statement
  - A definition is alive if the variable is not reassigned at an intermediate statement and it is a definition clear path
  - Variable use P-use (as a predicate) C-use (as anything else)
  - Testing each possible use of a definition is all-uses coverage
  - Requiring each definition path to be a simple cycle at most is all DU (Definition Uses) paths coverage
  - Many variants of these

Coverage Based Testing of Requirements

- Requirements can be transformed to a graph model with nodes denoting elementary requirements and edges denoting relations between elementary requirements
- Use this model to derive test cases and apply control flow coverage

Model of Requirements Specification



Fault Based Techniques

- Do not directly consider artifact tested, it is all about the test set
- Find a test set that is great at finding faults:
  - Fault seeding
  - Mutation testing

Fault Seeding

- Effort to estimate faults in a program
- Artificially seed faults, test to discover both seeded and new faults
- Total # of errors = ((tot err found - tot seed err found)* tot seed err)/tot seed err found
- Assumes real and seeded errors have same distribution
- Manually generating faults may not be effective
- Alternative is 2 groups, real faults found by X used as seeded faults by Y
- If we find many seeded faults and few others - results trusted, converse not true
- Many real faults found is not a positive sign - Poor Q
- Myers- probability of more errors in a section is proportional to the # of errors already found!

Mutation Testing

- Large # of variants of a program are generated by a set of transformation rules, e.g, replace constant by another, insert unary operator, delete statement
- “mutants” are executed using test set
- “mutants” producing same results as test expects are alive- if a test set leaves many alive it is poor, mutant adequacy score dead mutants/total mutants
- Based on 2 assumptions:
  - Competent programmer hypothesis - programs are close to correct - test small variants
  - Coupling effects hypothesis - tests that reveal simple faults can also reveal complex faults

Orthogonal Array Testing

- Intelligent selection of test cases
- Fault model being tested is that simple interactions are a major source of defects
  - Independent variables - factors and number of values they can take -- if you have four variables,e ach of which could have 3 values, exhaustive testing would be 81 tests (3x3x3x3) whereas OATS technique would only require 9 and would test all pairwise interactions

http://www.seilevel.com/OATS.html

OATS Table

	A	B	C	D
Run 1	0	0	0	0
Run 2	0	1	1	2
Run 3	0	2	2	1
Run 4	1	0	1	1
Run 5	1	1	2	0
Run 6	1	2	0	2
Run 7	2	0	2	2
Run 8	2	1	0	1
Run 9	2	2	1	0

OATS Method

1. How many independent variables (factors)
2. How many values will each variable have (levels)
3. Find a suitable orthogonal array -- premade tables
4. Map 1 & 2 onto array
5. Transcribe runs into test cases adding “obvious omissions” due to your knowledge
6. (simplified)

Test Adequacy Criteria

- Weyuker's properties:
  - Applicability -adequate test set for every program of reasonable size
  - Non-exhaustive applicability-do not require exhaustive testing
  - Monotonicity - for adequately tested software, more tests cause no harm
  - Inadequate empty set property- no tests = not adequately tested!

Weyuker -2

- Antiextensionality - semantic equivalence does not always permit using same test,e.g., sort
- General multiple change - same syntax/ data flow does not equal same test (arithmetic ops)
- Antidecomposition - component test is environment specific
- Anticomposition - unit testing still requires composition testing (interfaces and interactions)
- Renaming - if 2 programs differ in nonessential ways (variable names) same test sets are okay
- Complexity - more complex, more tests
- Statement coverage - every executable statement should be executed in testing

More on Testing

- Testing begins at component level and works outward (other direction is okay too)
- Different techniques are used at different points
- Testing involves the developer and an independent team and user advocates
- Testing and debugging are different but debugging must be accommodated
- Testing is the last bastion of Quality
- Quality cannot be “tested in”

Humphrey, 1989

Top-down and Bottom-up

	Bottom-up	Top-down
Major Features	•Allows early testing aimed at proving feasibility and practicality of particular modules. • Modules can be integrated in various clusters as desired. • Major emphasis is on module functionality and performance.	•The control program is tested first • Modules are integrated one at a time • Major emphasis is on interface testing
Advantages	•No test stubs are needed • It is easier to adjust manpower needs • Errors in critical modules are found early	•No test drivers are needed • The control program plus a few modules forms a basic early prototype • Interface errors are discovered early • Modular features aid debugging
Disadvantages	•Test drivers are needed • Many modules must be integrated before a working program is available • Interface errors are discovered late	•Test stubs are needed • The extended early phases dictate a slow manpower buildup • Errors in critical modules at low levels are found late

Types of Testing

- Unit testing - adjunct to coding, uses drivers and stubs, test cases source controlled
- Integration testing -test to uncover errors in interfacing
- Regression testing - subset of all tests to a given point to use when changes are made (part of build - smoke testing)
- Validation testing - succeeds when software functions in a manner that can be reasonably expected by the customer.. Alpha and beta testing are part of this
- System testing fully exercise the entire system:
  - Recovery testing - OA&M
  - Security testing
  - Stress testing
  - Performance testing
  - Reliability testing

Some Specialized Tests

- Testing GUIs
- Testing of Client server architectures
- Testing documentation and help facilities
- Testing real time systems
- Acceptance test
- Conformance test
- Your favorite here

BY Heuristics

- Test incrementally
- Test under no load (but very long), litle load, medium load, heavy load, over load - break it!
- Test error recovery code
- Spend more time testing stability and recovery than features
- Diabolic Testing - use data you do not expect the program to see
- Reliability testing to gauge rejuvenation level
- Regression testing - testing 50% of development time and 20% of costs, regression testing cuts this in half

Extreme Testing (sort of)

- J.A.Whittaker, How to break software. A different viewpoint
- How good testers do testing - flexible testing, not about rigid test plans - not an exact science
- "smart people doing exploratory testing have found all the best bugs I have ever seen"
- The difference between users and testers is that testers have clear goals
- Relies on a general software fault model
  - Familiar with the environment in which software operates
  - Understand capabilities of the application

Break Software - 2

- The Human User
  - Inputs delivered via GUI control
  - Inputs delivered by programs - through the API (developer as user), e.g., tools
- The File System User
  - E.g. file permissions
- The Operating System User
  - E.g., application works in low memory situations
- The Software User
  - E.g., external relational database - can it handle the data coming back?

Break Software - 3

- Software performs 4 basic tasks:
  - Accepts input from environment - test input
  - Produces output and transmits it to its environment - test output
  - Stores data internally in one or more data structures - test data
  - Performs computation using input and stored data - test computation

Break Software -4

- Examples for user interface:
  - Apply inputs that force all error messages to occur
  - Apply inputs that force the software to establish default values
  - Explore allowable character sets and data types
  - Overflow input buffers
  - Find inputs that may interact and test combinations of their values
  - Repeat the same input or series of inputs numerous times (consume resources)
  - …

Thought Problem

- A new manager heads testing and development and she believes in the SEI goal of "moving to the left" especially for testing. How would you get testing moved to the left?

- Do you think mutation testing would have any value in your current testing environment? Which phase of testing would it be most appropriate, unit, integration, system?

So Far

- Software Process Models, Software Project Planning (woosh!), Requirements, Estimation, Risk Analysis, Multics case study, Architecture Reviews, Questionnaire Design
- Software Quality Assurance
- This Time: Configuration Management and Testing
- Next Time: Architecture and Design

Lecture Resources

- R.S. Pressman, Software Engineering a Practitioner's Approach, McGraw-Hill, 5th edition, 2001, ISBN:0-07-365578-3.
- http://wiki.org/
- J.A.Whittaker, How to break software, Addison-Wesley, 2003. ISBN: 0-201-79619-8.
- D. Spinellis, Code reading, Addison-Wesley, 2003, ISBN: 0-201-79940-5
- B. Collins-Sussman, B.W. Fitzpatrick, C.M. Pilato, Version Control with Subversion For Subversion 1.1, http://svnbook.red-bean.com/
- Humphrey, Watts S , Managing the Software Process, Addison-Wesley Publishing Company, Inc., 1989