

Class 8 CIS 573

(lecture 7)

Gregg Vesonder
University of Pennsylvania
Penn Engineering - Computer & Information Science
©2009 Gregg Vesonder

Roadmap

- Mid-Term
- Quality
- Coding
- Readings this class: Brooks chapter 12, Sommerville chapter 29, Andersson, et.al., chapter 7.
- Readings next class: Sommerville chapter 23
- Readings next week - posted on wiki Sunday, If not earlier

Critical Dates

- Every class project review
- ~~July 23rd Mid Term~~
- August 6th log books due
- August 11th project presentations
- August 13th Final

Teams

- Team 1 - Klein Keane, Beck, Buchman, Richardson, Nunez
- Team 2- Wilmarth, Caputo, Xiang, Francis, Nanda
- Team 3- Noronha, Fang, Huang
- Team 4-Whitehead, Liu, Ratnakar

Project Reports

- Presentation each class
 - Green, yellow, red -simplified model + gaps
 - Current pressing issues
 - What was done since last class
 - What will be done before next class
 - Gaps

Log Book

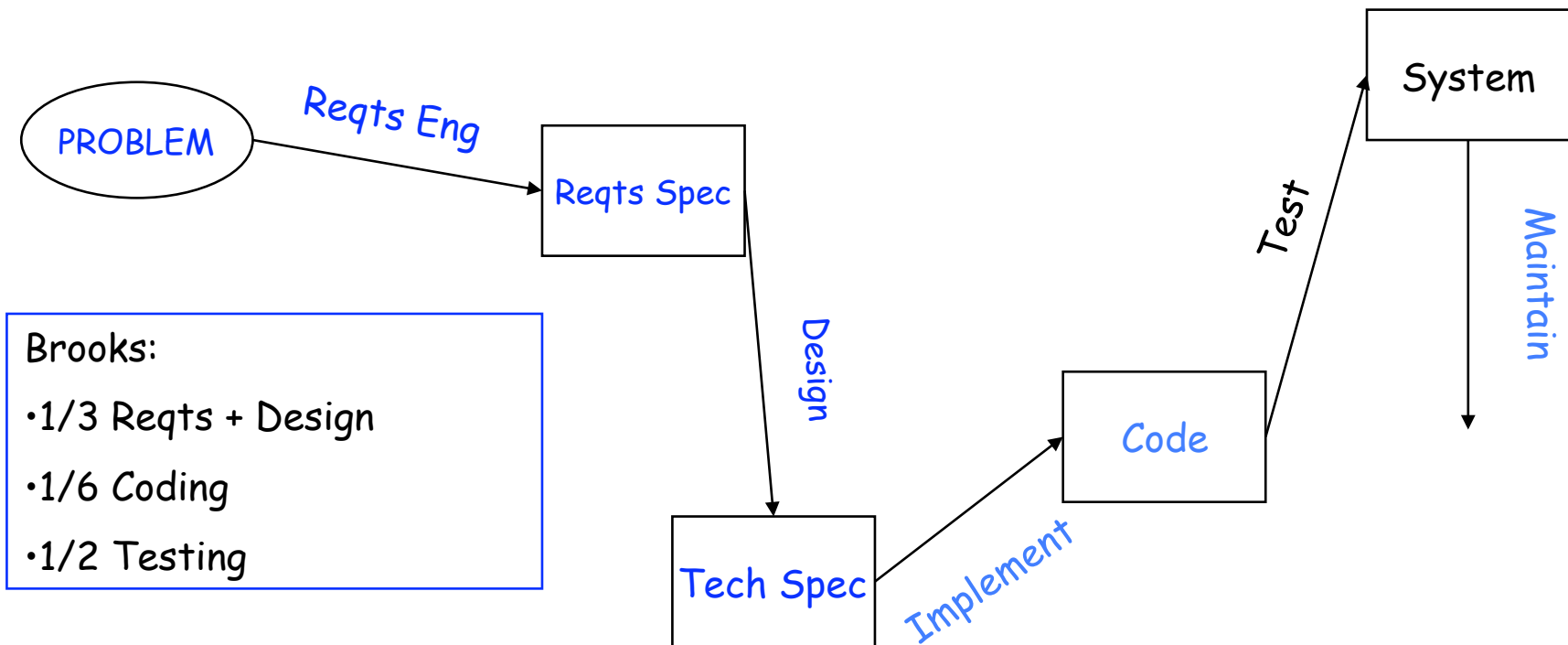


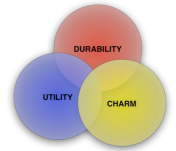
From <http://www.culture.gouv.fr/culture/arcnat/lascaux/en/>

Software Engineering Knowledge

- SWEBOK, SoftWare Engineering Body Of Knowledge:
 - Software requirements analysis
 - Software design
 - Software construction
 - Software testing
 - Software maintenance
 - Software configuration management
 - Software quality analysis
 - Software engineering management
 - Software engineering infrastructure
 - Software engineering process

Simplified Model





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
- 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)

Testing Overview

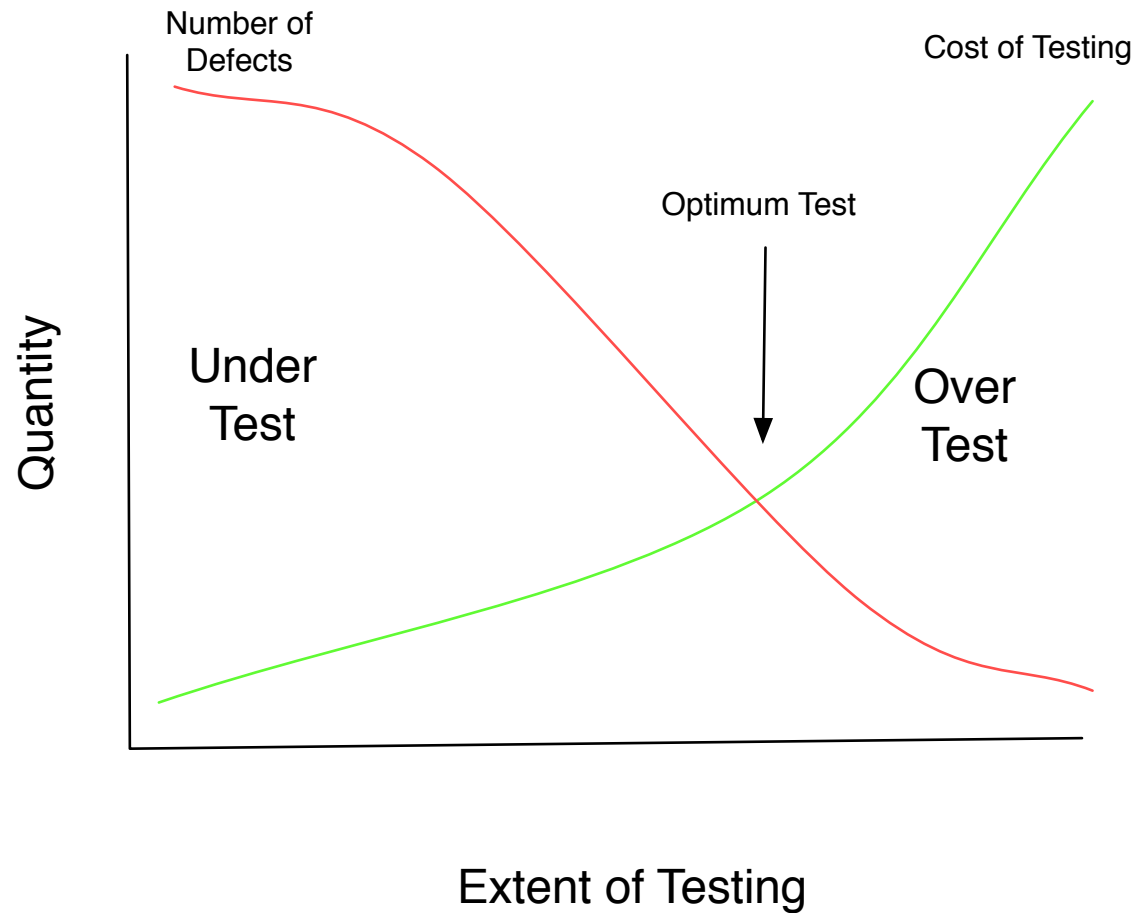
- History of testing: from user confidence to finding errors, from phase to life cycle
- Never enough - start early
- Many types of testing
- Test not only the software but the test set itself
- Traceability is important
- Coverage based techniques - only mention
- Extreme Testing
- Really important- more coverage in class

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?

View of Test Effort

Perry(2000)



Types of Testing

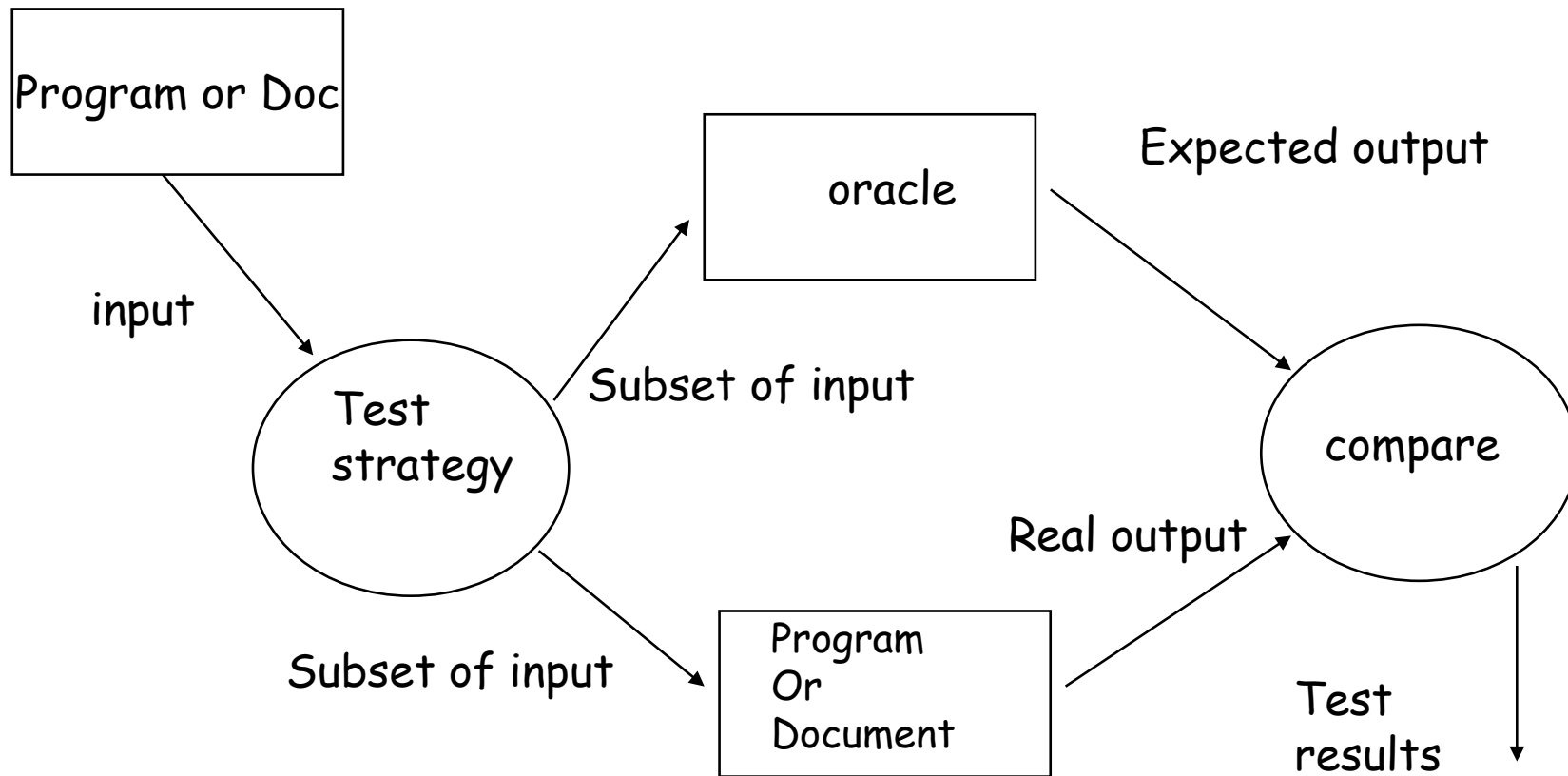
- Coverage based - coverage of product, e.g., all statements must be executed at least once
- Fault based- detect faults, artificially seed and determine whether tests get at least X% of the faults
- Error based - focus on typical errors such as boundary values (off by 1) or max elements in list
- Black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

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."

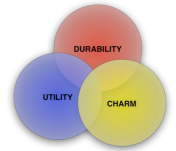
Global View of Test Process

Fig 13.2



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

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
- Roots 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 thoroughly!)

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

Test Plan

Perry(2000)

- **General Information**
 - Functions of application and tests to be done
 - Environment and Pretest background -history, place of testing, any prior testing
 - References: authorization, documentation best practices of organization, ...
- **Plan:**
 - Software description at flow chart level including inputs, outputs and functions
 - Milestones - includes responsibility
 - Testing - location, participating organizations
 - Schedule
 - Requirements: Equipment, Software, Personnel available during test
 - Testing materials: Documentation, software to be tested and its medium, test inputs and sample outputs, test control software (harness)

Test Plan - 2

- Specifications and Evaluation
 - Specifications:
 - Requirements - functional requirements
 - Software functions - detailed application functions to be exercised
 - List tests and relate to software functions
 - Test progression - ordering of tests
 - Methods and constraints
 - General strategy
 - Conditions - type of input used, live or test data, volume and frequency
 - Test results recorded and format
 - Constraints- hardware, load, ...

Test Plan - 3

- Evaluation - process to evaluate test results
 - Criteria: range of values used, ...
 - Data reduction: test results data necessary fo reports
- Test Descriptions - real detail
 - Control - how tests are done, manual, automated
 - ... detailed plan of each test

Test Report

- General Information - summary
- Test results and findings, includes dynamic and static
- Software function findings - for each performance and limits
- Analysis summary: capabilities, deficiencies, recommendations and estimates, opinion

Manual Test Techniques

- Reading - peer review, insight via best and worst technique
- 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 - active reviews

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

(unit testing)

- 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

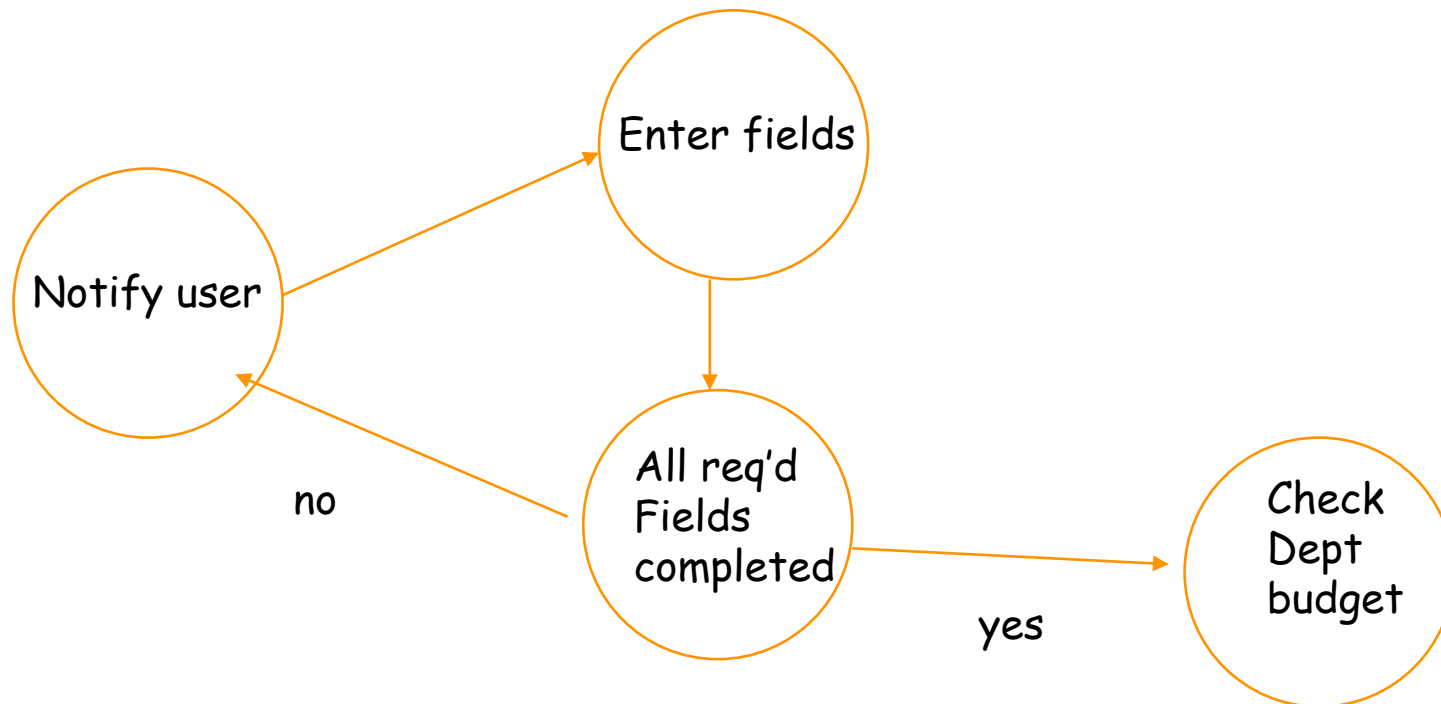
Coverage Based Techniques -2

- **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
 - 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 = $((\text{tot err found} - \text{tot seed err found}) * \text{tot seed err}) / \text{tot seed err found}$
- 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 $\text{dead mutants} / \text{total mutants}$
- **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, each of which could have 3 values, exhaustive testing would be 81 tests ($3 \times 3 \times 3 \times 3$) whereas OATS technique would only require 9 tests yet would test all pair-wise interactions

OATS Method

1. How many independent variables (factors)
2. How many values will each variable have (levels)
3. Find a suitable orthogonal array -- pre-calculated 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

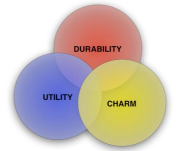
- HOW LONG TO TEST
- Weyuker's properties:
 - Applicability -adequate test set for every program of reasonable size
 - Non-exhaustive applicability-do not require exhaustive testing
 - Monotonicity - for adequately tested software, more tests cause no harm
 - Inadequate empty set property- no tests =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



Top-down and Bottom-up

	Bottom-up	Top-down
Major Features	<ul style="list-style-type: none"> • 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. 	<ul style="list-style-type: none"> • The control program is tested first • Modules are integrated one at a time • Major emphasis is on interface testing
Advantages	<ul style="list-style-type: none"> • No test stubs are needed • It is easier to adjust manpower needs • Errors in critical modules are found early 	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • Test drivers are needed • Many modules must be integrated before a working program is available • Interface errors are discovered late 	<ul style="list-style-type: none"> • 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

Some Other Heuristics

- Test incrementally
- Test under no load (but very long), light 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)
 - ...

Cleanrooms

- Cleanroom techniques - build correctness into software as it is being developed
- Instead of analysis, design, code, test and debug
- Write code increments correctly the first time and verify correctness before testing - Do it right the first time
- Verification is through mathematical techniques
- Testing certifies software reliability

Steps in Cleanroom Design

- Analysis and design uses a box structure notation
 - Box encapsulates some aspect of the system (or entire system) at a level of abstraction
 - Correctness verification is done once the box structure design is completed
 - This is done in place of unit testing
- Testing is done by defining set of usage scenarios and probability of use for each scenario, then defining random tests that conform to the probabilities
- Error records from testing are analyzed to compute reliability
- Heavily incremental approach

UE - Undesired Events

- Basis for exception handling
- Always aspects of a program's execution environment that do not behave as we wish - if you will, defensive programming
 - Arises from the "normal" behavior of the real world
- Goal is to anticipate what can go wrong and make (the possibility for) accommodations in advance that do not mess with the structure
- Basis for variants of throw and catch

More on UEs

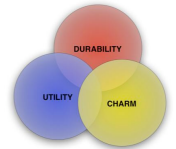
- Differentiate from errors that should be corrected
- UEs demand an evolutionary approach - they evolve as experience with system increases
- Responses to UEs include:
 - Self diagnosis
 - Saving of partial results
 - Printing of diagnostic information
 - Retrying
 - Use of alternate resources
 - Job cessation only occurs as a last resort
- Traps keep code for UE separate from code for modules, lexical separation of normal use, from detection and correction procedures

Classes of UEs to detect

- Limitations on values of parameters
- Capacity limits
- Requests for undefined information
- Restrictions on order of operations
- Detections of action which are likely to be unintentional
 - "open" when opened, but this could be overloaded by convention
- Errors of mechanism - failure of modules is more difficult than failure by applicability condition

More on UEs

- Assign priorities to these traps usually more than one is active
- Try to consolidate UE routines, not one for one
- Eliminate some of the redundant checks as data is passed around for efficiency.
- Incidents, UEs that were expected and where recovery attempts were successful, versus crashes

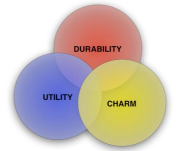


Open Source Software Cathedral and the Bazaar

- **Cathedral** - commercial software world, **bazaar** - linux and open source
- Key names:
 - Richard Stallman - emacs, gnu, Free Software Foundation
 - Linus Torvalds, linux, open source process, open source license (General Public License (GPL), BSD, Perl's Artistic License)
 - Larry Wall - PERL

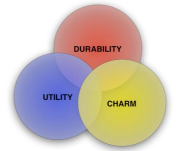
Flavor of Open Source

- Torvalds style - release early and often, delegate be very open - VERY developer centric!
- Axiom 1 - Every (?) good work of software begins by scratching a developer's itch (does not conform to numbers in the paper)
- 2- Good programmers know what to write, great ones know what to rewrite and reuse
 - **Constructive laziness**
- 3-Plan to throw one away, you will anyhow
 - You do not understand problem until after first time you implement
- 4- If you have the right attitude interesting problems will find you
- 5- When you lose interest in a program, your last duty is to hand it off to a competent successor



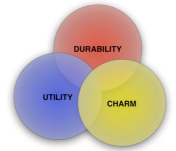
Flavor of OS - 2

- 6- treating your users as codevelopers is your least hassle way to rapid code improvement and effective debugging
 - If you have that luxury
 - Torvalds "too lazy to fail"
- 7- release early, often and listen to customers
 - Released a new linux kernel in the early days more than once a day!
- 8- given a large enough beta tester and code developer base, almost every problem will be characterized quickly and the fix obvious to someone
 - Linus' Law "Given enough eyeballs, all bugs are shallow"
 - Delphi effect
 - Debugging is parallelizable
 - Brooks: more users find more bugs
 - Non source aware users do not provide great bug reports
 - "eat your own dog food"



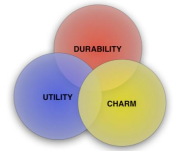
Flavor of OS 3

- Team: Usually 1-3 core developers, beta testers and contributors in the 100s
- 9- If you treat your beta testers as if they're your most valuable resource, they become it
- 10- the next best thing to having good ideas is recognizing good ideas from users. Sometimes the latter is better
- 11-often the most striking and innovative solutions come from realizing your concept of the problem was wrong
- 12 - Perfection (in design) is achieved not when there is nothing to add, but rather when there is nothing to take away, Antoine de Saint-Exupery
 - Debugging is not only parallelizable, so is development and exploration of the design space!



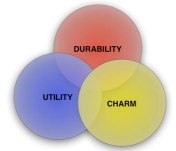
Flavor of OS 4

- 13 - any tool should be useful in the expected way but a truly great tool lends itself to uses you never expected
- 14 -International flavor of participants is a plus in **globalization** (extract)
- 15-To solve an interesting problem, start by finding a problem that is interesting to you



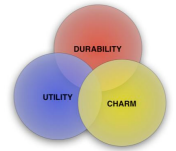
Preconditions for the Bazaar style

- Hard to originate code in this style, test, debug, improve yes
 - You need to have something running - **attractor**
 - It has to run and convince others that it can evolve into something neat in reasonable time
- Leader/coordinator of Open Source project does not need to be a great designer but **needs to recognize good ideas from other folks**:
 - Robust and simple rather than cute and complicated
 - Community's internal market based on reputation exerts subtle pressure in self-selecting competent leaders
 - A bazaar leader must have good people and communication skills



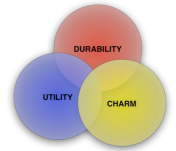
Social context for Open Source

- Evolution of software in the presence of a large, active community of users
- Programmer time is not fungible - small number of codevelopers obeys Brooks communications links
- Programmers cannot be territorial about code, encourage others to look for bugs and improvement XXP!
- "while coding remains an essentially solitary activity, the really great hacks come from harnessing the material and brain power of entire communities"
- Internet helped
- Development of leadership style and set of cooperative customs
- Utility function of linux hackers is maximizing in their own eyes satisfaction and reputation among peers and users
- Boring is essential - software and documentation
- Open source is fun - joy as an asset
- Not so easy as to be boring, not too hard to achieve!



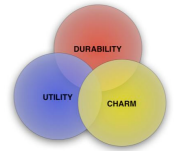
Homesteading the Noosphere

- Open Source culture: zealotry varies, hostility to commercial software varies
- Open Source must protect an unconditional right of any party to modify (and redistribute modified versions of) open source software
- **Taboos of Open Source:**
 - Strong social pressure against forking projects
 - Distributing changes to a project without cooperation of moderators is frowned upon
 - Removing a person's name from the project history is not done without the person's explicit consent



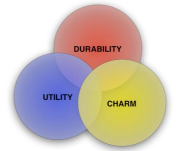
Ownership in Open Source

- Owner of the software project is the person who has the exclusive right to distribute modified versions
- How to own:
 - Start the project - [homesteading](#)
 - Have ownership handed to you - [deed transfer](#)
 - Observe that a project needs work and owner has lost interest- try to find owner to get to have ownership handed to you - "[adverse possession](#), moves on and improves"



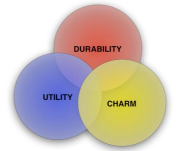
Hacker culture as a gift culture

- Excludes crackers and warez d00dz - different culture
- Not what you control but what you give away - reputation among peers
- Along with sheer joy of making something work
- Craftsmanship model as a corollary - still linked to reputation
- In hacker culture status is based on critical judgment of peers



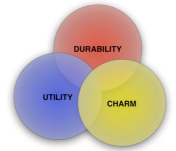
On Reputation

- Ego is despised, yet the whole system runs on it!
- One's work is one's statement, no one attacks one's technical competence, emacs bugs not Stallman's bugs
- More prestige in founding a project than working on an existing one
- More prestige for innovative rather than me too
- Being carried in a major distribution (Red Hat, SuSE) is prestigious
- Continued devotion to hard, boring work (debug, write doc) is more praiseworthy than fun and easy hacks.



Governance

- Project leader - codevelopers - contributors
- Apache has a voting committee
- PERL has rotating dictatorship among codevelopers



Open Source Resources

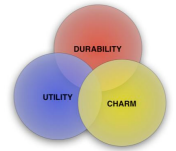
- www.opensource.org - jump off point, Halloween papers
- SourceForge.net - project pages
- Freshmeat.net - products and product announcements
- Slashdot.org - fun

Microsoft's response

- [The Halloween documents](#) - response to Open Source, quotes Raymond
- Considers Open Source a threat especially in server market (H II - also in desktop)
- Commercial quality can be achieved!
- [To target Open Source, you target a process not a product](#)
- Open source is long term credible - you cannot FUD it!
- Implementation provides a high visibility showcase for OS
- Linux has done well in mission critical commercial environments
- Linux can win as long as services and protocols are commodities

Microsoft Development - Lucovsky

- NT: 6 staff from DEC, 1 staff from Microsoft
- Design goals for NT family:
 - Portability - abstract away machine dependencies
 - Reliability - nothing should be able to crash the OS
 - Extensibility
 - Compatibility
 - Performance but all of the above are more important
- Design Workbook - written by engineers for engineers
 - Every functional interface was defined and reviewed - small teams essential
 - Spread review duties and everyone shares culture



Time to Big

- To scale a team you need to establish a culture
 - Common way of evaluating designs, tradeoffs
 - Common way of developing code
 - Common way to establish ownership of problems
- Goal setting as foundation for the culture - hard as it grows
- Every decision made in the context of design goals
- Everyone owns all the code, so whenever something is busted anyone has a right and duty to fix
 - Works in small groups (< 150!)
- Sloppiness is not tolerated
- Accept that mistakes will happen

NT source code control system

- Internally developed, by a non-NT tools team - no branch capability
- Small hard drive could hold whole tree (6M LOC), 10-12 source projects
- Easy to stay in synch
- Onto Win 2000 needed branching, (29M LOC), 180 source projects - full source, 50 gig, up to date machine, 2 hours to sync

NT build

- 4 hours sync period, could check in code the other 20 hours
- Build lab syncs during 4 hour period (in morning) and begins a complete build - 5 hours on 486/50
- Preliminary test is done then off to 4pm stress test on ~100 machines

Win 2000 build

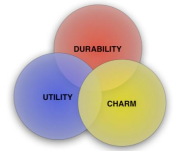
- No source tree changes w/o explicit permission
- Build lab approves ~100 changes each day and manually syncs and builds
 - A developer mistyping a build instruction can stop build which stops 5000 people
- Build 8 hours on 4 way PIII Xeon with 50 gig disk and 512K
- Build boot and baseline tested then 4pm stress testing on 1000 machines

Team Sizes

PRODUCT	Dev Team	Test Team
NT 3.1	200	140
NT 3.5	300	230
NT 3.51	450	325
NT 4.0	800	700
Win2000	1400	1700

Defect Rates Data

Product team size	Defects/yr /dev	Time to fix /defect	Defect/day	Total fix time
NT 3.1, 200	2	20 min	1	20 min
NT 3.5, 300	2	25 min	1.6	41 min
NT 3.51, 450	2	30 min	2.5	1.2 hours
NT 4.0, 800	3	35 min	6.6	3.8 hours
Win2000, 1400	4	40 min	15.3	10.2 hours



"Sync and Stabilize"

Cusmano and Selby (1997)

- Scale up loosely structured teams (3-8 developers)
- The "Process"
 - Vision statement - defining goals of new product
 - Program managers create functional specification
 - During development team members revise feature set and details
- Sync involves daily builds, Stabilize involves milestones
- "Always" a deliverable project

10 OO Development Anti-Patterns

- **THE BLOB** - Procedural style design leads to one object with Lion's share of responsibilities, while most other objects hold data or execute simple processes - redistribute and refactor
- **LAVA FLOW** - dead code and forgotten design information is frozen in a dynamic design - configuration management process to eliminate dead code
- **FUNCTIONAL DECOMPOSITION** - FORTRAN in a class structure! - hard to determine at times, back to design and look for keys, e.g., if a class has a single method try to incorporate in other classes

More AntiPatterns

- POLTERGEISTS classes with very limited roles and effective life cycles, start processes for other objects - place responsibility in more long-lived objects
- BOAT ANCHORS a piece of software or hardware that serves no useful purpose on project, usually very costly
- GOLDEN HAMMER - familiar technology or concept applied obsessively - education to expand technical horizons
- SPAGHETTI CODE - ad hoc software structure with very little clarity caused by excessive patching - rewrite it
- DEAD END modifying a reusable component no longer maintained by the supplier, lost the original reason for it - replace

End of AntiPatterns

- CUT AND PASTE PROGRAMMING code reused by copying source statements - black box reuse and alternate forms of reuse
- MUSHROOM MANAGEMENT - system developers isolated from system's end users - connect them!
- BONUS patterns - DEATH BY PLANNING and ANALYSIS PARALYSIS

Lecture Resources

- R.S. Pressman, Software Engineering a Practitioner's Approach, McGraw-Hill, 5th edition, 2001, ISBN:0-07-365578-3.
- 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
- Parnas, D.L. and Wurges, H. " Response to undesired events in software systems." In Hoffman and Weiss
- E.S. Raymond:
 - The cathedral and the bazaar
 - Homesteading the Noosphere
 - Halloween papers
- Mark Lucovsky, "Windows a software engineering odyssey" - build centric view
- W.J. Brown, et.al., AntiPatterns: Refactoring software, architectures and projects in crisis, Wiley 1998
- Cusumano, M.A & Selby, R.W., CACM, 1997.
- Perry, W.E. Effective methods for software testing, Wiley, 2000