ELEC 372

The Software Development Process (2)
Two Industrial Examples

References

- MicroSoft Secrets; Cusumano, Selby, Simon &Schuster Inc., 1995
- Ontario Hydro's Experience with New Methods for Engineering Safety Critical Software; M.Viola, Proceedings Safecomp 1995, Italy

The Microsoft Triangle

- Decision making focuses on three of the dimensions of quality in the end product:
  - Functions supported
  - Timing to Market
  - Implementation defects remaining
- rapid development, daily builds, testing buddies
- evolving requirements, work from vision statement
- heavy reliance on testing, multiple testing strategies
- well established plans and schedules

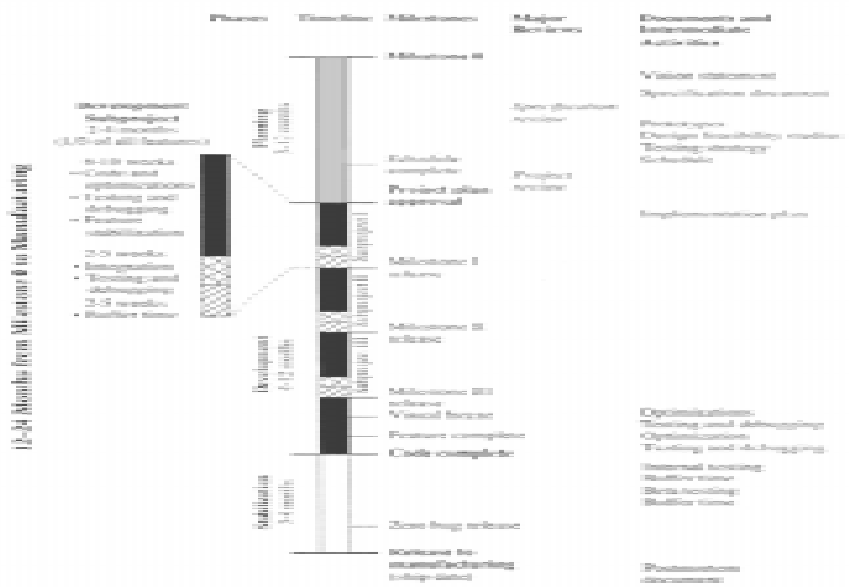
Common Principles

- manage by milestones
  - start with most difficult things first
- assume change will happen
  - don't write complete specification up front
- focus on customer problems
  - not on technologies or processes
- build in buffer time for unforeseen delays
- let people do own schedules
  - ownership of schedules
- hire the smartest people
  - best software programmers may write ten to twenty times more code in same time as the least productive member of team
- establish functional specialties
  - work in small teams

Approach

- "fix" project resources
  - number of people
  - amount of time
- set intended ship date
- define milestones backwards from target ship date
- no separate maintenance group
  - fixes included in next product release
- three phases
  - iterative approach
  - more like a "risk-driven" incremental spiral life cycle
  - planning phase, development phase, stabilization phase
  - uses concept of buffering throughout

MicroSoft Incremental Build



Planning Phase (3-12 months)

- marketing, vision statement, design goals
- initial specification
  - only preliminary outline
  - 20 to 30% changes from start to end of project
  - project managers keep it up-to-date
  - answers questions such as
    - What is the point of this feature?
    - What does the user want to use this feature for?
- project managers do extensive prototyping
  - with Visual Basic
  - from users viewpoint

Development Phase (6-12 months)

- 3 or 4 major milestone releases
  - pretend these will be shipped
- developers write code
- testers write test suites
- user education staff write draft documentation
- developers fix errors continuously as testers find them
- 1/3 of total development phase is "buffer" time
- "visual freeze"
  - major user interfaces frozen
  - for benefit of user education group
- "feature complete"
  - fully functional
  - test scenarios can run
  - still has bugs and performance issues
- "code complete"
  - all features work as described in specs and users doc
  - point where bug count should be steadily declining

Stabilization Phase (3-8 months)

- 1/3 of total time in project
- extensive testing & debugging
- no new features
- "zero bug release"
  - no more high-severity bugs have been found
  - all remaining bugs will be fixed in next release

Milestone 0

- after one release and before official start of next development cycle
- developers are responsible for making their own code work
- go through and clean up code that isn't right
- spend about 10% of time redoing existing features

Daily Build Process (1)

- Check Out
  - private copies of source code
  - from centralized master version
  - multiple developers can check out same code at same time
- Implement Feature
  - developer implements his/her own changes
  - could take a few hours to several days
- Build Private Release
  - developer builds own private copy of the product with newly implemented features
- Test Private Release
  - developer tests own copy of product

Daily Build Process (2)

- Synch Code Changes
  - compare private copies of source files back to master version
  - other developers may have made changes in the mean time
- Merge Code Changes
  - update private copies of source files to include any changes from other developers
- Build Private Release
  - developer builds own copy of product with both own changes and changes of other developers
- Test Private Release
  - test to make sure newly implemented code still works with others' changes

Daily Build Process (3)

- Execute Quick Test
  - done on private release
  - highly automated
  - makes sure new changes do not interfere with basic functionality of product
  - does not test new feature directly
  - e.g., quick test on EXCEL takes 30 minutes
  - also called smoke test or regression test
- Check in
  - developer officially checks his private copies into the master version
  - has to synch and merge again in case of further changes
  - if conflicts, may have to withdraw changes
  - deadline for check in
    - for EXCEL, 2PM

Daily Build Process (4)

- Generate Daily Build
  - after check in deadline
  - build master generates complete build of product from master version
    - called "daily build"
    - after successful build
      - executes series of automated tests
      - assures basic functionality works correctly
      - makes daily build available to all project staff

Testing (1)

- quick tests on daily build
- developers create debug versions
  - usually to view internal data structures
  - improve memory usage and execution speed
- use assertions "if then" tests
  - when making assumptions about data stored elsewhere in program
- instrumentation for user testing
  - captures commands users select
  - captures corresponding source code statements that get executed
- usability testing
  - lab with "people off the street"
  - user interfaces tested

Testing (2)

- weekly test releases
  - to testing group for thorough testing
  - use highly structured scripts
  - "guerrilla" (gorilla?) testing
    - try to break the product
- testing buddies
  - work side by side with developers
  - developer shares private release with his testing buddy
  - do private release testing before code goes to daily build
- others
  - performance testing
  - internal usage
  - beta testing

Final Notes

- little architectural documentation
  - source code is one main document
  - try to keep code as simple as possible
  - compartimentalize code
- very little code comments
  - uses "Hungarian" rules for variable naming
    - E.g., *pszName* means pointer to zero terminated string
  - half life of source code is as little as 18 months
- Cusumano & Selby recommend
  - more use of metrics
    - do use bug counts
  - design and code reviews
    - intensive code inspections
    - detect and fix problems before incorporated into product
    - refine product architecture
    - educate junior developers

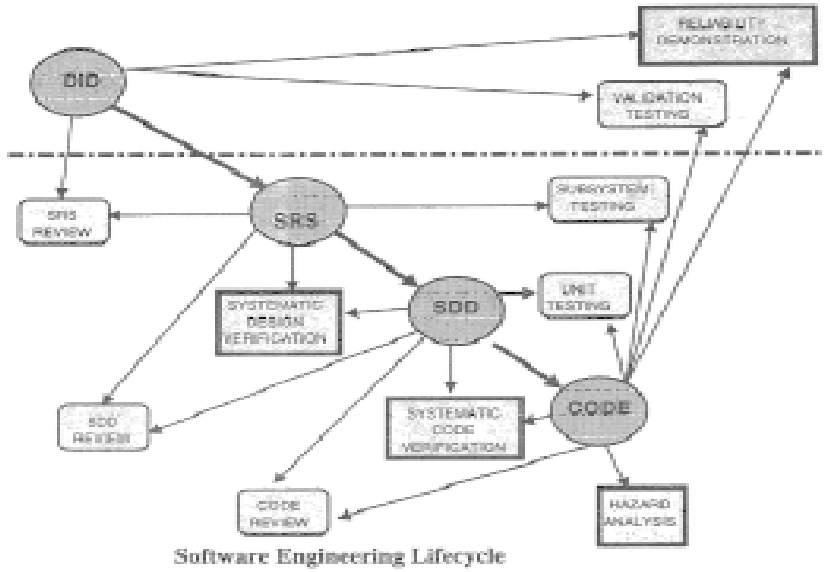
Safety Critical Software - Background

- 1990 Darlington Nuclear Generating Station brought on-line
  - first time trip-decision logic in reactor safety shutdown system (SDS) entirely software
  - regulator required that software be verified before put into use
  - software already written
    - two independent systems (physical and logical) to shut down the reactor
      - shutoff rods
      - poison injection
    - designed separately to reduce common mode errors
    - 7000 lines of FORTRAN
    - 13000 lines of PASCAL

Background (2)

- David Parnas hired by regulator to advise on process
- procedure based on Parnas Tables
  - formal verification
  - rendered code into tabular format
  - rendered requirements into tabular format
  - proofs to show code and requirements the same
  - took about 1 year to complete
    - about 60 people involved on FORTRAN side
- allowed software to go into production
  - agreed to redesign and rewrite software
  - had to establish standards and rigorous process

Safety Critical Software Process



Software Engineering Lifecycle

Features of this Process (1)

- uses step-wise refinement
  - at each level, additional and more specific requirements included to satisfy more abstract requirements identified in preceding level
- Design Input Document (DID)
  - collection of specifications fed into SE process
  - eg. digital design description, hardware reference manuals, human-machine interface requirements, performance requirements, maintenance requirements, applicability of standards and codes
- formal specification (SRS, SDD)
  - behaviour of software described using mathematical functions in a notation with well defined syntax and semantics

Features of this Process (2)

- review and verification
  - outputs from each process must comply with inputs to that process
  - outputs written using mathematical functions must be systematically verified against inputs using mathematical verification techniques
  - manual reviews against standards, checklists
- reliability testing
  - uses statistically valid random testing
- software hazard analysis
  - analyses performed to identify and evaluate potential unsafe failures in the computer system and in the software component of the system
  - eliminate or assist in reduction of risks to an acceptable level

Experience with the Process (1)

- trial use of process
  - microprocessor based digital trip meter for Pickering B NGS
    - temperature detectors send current back to trip meter
    - trip action initiated if signal exceeds setpoint
  - 1500 lines of C code
  - 25% of entire development cost was in two formal verifications
  - 27% of entire development cost was in four levels of testing
  - 5% of entire development cost was writing the code

Experience with the Process (2)

- SDS rewrite
  - new software installed in the field
    - started February 1999; ended December 1999
  - 5-year project
  - kept existing hardware
  - redeveloped software according to step-wise refinement and information hiding
  - about 40 to 50 people overall on project
    - largest group was formal verification group at 20 people
  - have some tool support
    - prototype verification system (PVS) (theorem prover) from Stanford Research
    - other in-house tools for formatting and consistency checking
    - input to one tool had to be produced by hand
      - 600 pages with word processor

Strengths

- formal tabular notation
  - provides for more complete specifications
  - errors eliminated earlier in lifecycle
  - facilitates formal verification
  - precise specifications leads to code with few errors
- information hiding design technique
  - lessens impact of changes
- hazard analysis provides confidence in safety objectives of software
- SRS and SDD used to generate test cases
  - achieved known test coverage
- validation tests using DID
  - done by individual outside development process
  - identified problems both during test case creation and during testing

Weaknesses

- formal tabular notation
  - expressing timing requirements
  - formalized requirements too early leading to costly rework
  - difficult to render code into this notation
    - done manually and subject to error
- lack of tool support
  - in particular, code review manually intensive
    - checklist with 154 questions
  - for creating and verifying test cases

Weaknesses

- information hiding design technique
  - subjective in nature & biased by individual's experience, background, knowledge
  - had to back off on information hiding design to get necessary performance
- hazard analysis done too late in design cycle
  - changes may have to be done back at SDD or SRS
- testing did not find any functionality errors not already identified through other verification processes

ELEC 372

Overview of Testing

adapted from lectures by Dr. Terry Shepard
Royal Military College of Canada
Electrical and Computer Engineering

Testing is Hard!

- hits budget and schedule limits
- may include heavy debugging
  - may even include product (re-)development
- bugs slip through to customers
  - user executes untested code
  - order in which statements are executed in actual use differ from that in testing
  - user applies combination of untested input values
  - user's operating environment is never tested

Skills of a Tester

- Not all testers are developers
  - e.g. testers at Corel who are photographers (Corel Draw testing)
  - e.g. testing buddies at Microsoft
- testers who are developers must have
  - ability to select good test cases
  - ability to withstand the most severe deadline and resource pressures
  - ability to organize and analyze large volumes of data
  - ability to sort out complex version control issues
  - plus the abilities of a developer
- roadmap for software test engineering: http://www.sdmagazine.com/articles/2001/0102/0102b/0102b.htm

Roadmap for Software Test Engineering

- Software engineering skills
  - understanding the rules of software engineering
  - computer programming
  - operating system level knowledge
- communication skills
- organizational skills
- hands-on experience
- attitude
  - destructive creativity
  - detective skills
  - understanding the product as the sum of its parts
  - appreciating the customer's perspective
  - requirements change
  - skeptical but not hostile attitude
  - ability to be the bearer of bad news and remain objective
  - an eagerness to embrace new technologies

Testing Overview Topics

- Definitions of Testing
- Purposes of Testing
- Testing Apparatus
- Stages of Testing
- Kinds of Testing
- Testing Process
- strategies for selecting test cases
  - controlling combinatorial explosion
  - test adequacy
- test automation
  - automation of test execution
  - automatic generation of tests
  - automatic analysis of test results
    - comparisons with expected results
    - determination of test coverage

Definitions of Testing (1)

- Myers, Glenford J., The Art of Software Testing, Wiley, 1979
  - "The Process of executing a program with the intent of finding errors"
- Hetzel, Bill, A Complete Guide to Software Testing, QED Information Sciences Inc., 1988
  - "Testing is any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results."

Definitions of Testing (2)

- IEEE Standard 610.12-1990
  - "The process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component"
- IEEE Standard 829-1983
  - "The process of analyzing a software item to detect the differences between existing and required conditions (that is , bugs) and to evaluate the features of the software items"

Secondary Definitions

- mistake, defect, error, fault, failure, bug
  - defect
    - issue found in the product by inspection
  - failure, fault
    - discrepancy detected by testing or use
- Static Analysis - e.g. proof of correctness, style checking
- Dynamic Analysis - generally requires execution of the code. but may not - eg. reachability analysis: dead code; checks for memory leaks in C++

Purposes of Testing

- To find bugs - including bugs in the tests!
- To show correctness: in the context of formal techniques supported by testing
- To estimate reliability in use by tracking the execution time interval between failures
- To estimate defects remaining by tracking the defects found per person-hour of testing and debugging time
- To decide on when to release: e.g. by deciding that the remaining known defects are acceptable in the target market
- To learn where process problems are by classifying and counting defects found

Testing Apparatus

- Test Plans, Test Cases, Test Suites
- Stubs
- Harnesses
- Drivers
- Oracles
- Generators
- Operational Profiles

Stages of Testing (1)

- Requirements and Design testing
  - Basis of operational approach: executable requirements and design
- Unit/Module Testing
- User Interface testing
  - Often done at prototyping stage
- Component/subsystem Testing

Stages of Testing (2)

- Integration Testing
  - Incremental Testing -the usual version of integration testing.
- Daily builds and Sanity testing; smoke testing
- Functional Testing
- Alpha and Beta Testing

Stages of Testing (3)

- System Testing
  - Performance and Load/Stress Testing
  - Usability Testing, but this is a late stage to be doing it!
- Acceptance Testing
- Installation/Compatibility Testing
- Platform/Configuration/Port testing

Stages of Testing (4)

- Integrity and Release testing
  - documentation consistent with product
  - media not infected with viruses
  - intended version is what is actually released,...
- Reliability/Certification Testing
  - estimates MTTF (mean time to failure) with reference to operational profiles
  - sometimes for certification of conformance to a standard - e.g. ACVC (Ada), ISO protocol test suites
- Security/Fault Tolerance/Recovery testing
- Regression Testing

Kinds of Testing: some details - just a sample!

- Black Box/Clear Box
- Approaches to Integration Testing
- Aspects of system testing
- Regression Testing Issues
- Acceptance Testing Issues

Black Box: Functional Testing

- Ideal: Controlled automated generation of test cases and expected results from specifications, and automated execution
- In practice: specifications are poor
  - must therefore select test cases manually
    - in the worst case, from the code (white box bias)
  - must calculate expected results by hand
  - can still use automated execution

Clear Box: Structural Testing

- Not a good basis for selecting test cases in general
  - coder may not see needed tests
  - may lead to focus on less important parts
  - intrinsically leads to incomplete testing
- may be useful to guide some testing
- Useful for assessing test coverage
  - e.g. Statement, Branch, and Path Coverage

Clear Box: Structural Coverage

```
void tst(int x)
{
if (x>0)
    pos = pos+1;
if (x%2==0)
    even = even+1;
return;
}
```

- Statement Coverage: tst(2)
- Branch Coverage: tst(-1),tst(2)
- Path Coverage: tst(-2),tst(-1),tst(1),tst(2)

Clear Box: Data Flow Testing

- exercise (in order of weakest to strongest testing)
  - all input and output variables
  - both truth values of all predicates
  - all definitions of variables (eg variables on LHS of assignments)
  - all uses of variables that are defined
  - all loops
- real time software generally runs inside an infinite loop
  - need to carefully define what you mean by path or loop coverage

Usually need mixed clear/black box testing

- Unit testing:
  - dual black box/clear box approach
  - level of smallest granularity: want to look at code
- Above unit testing level: gray box
  - awareness of structure, but not all details,
- System testing: primarily black box

Integration Testing (1)

- Tests uses between modules - source of 40% of all errors in code.
- Incremental: Top Down vs. Bottom Up
  - start with tested modules in either case
  - planning of increments starts early
    - should revolve around uses hierarchy
  - top down requires stubs
  - bottom up requires drivers
  - 'sandwich' variants exist

Integration Testing (2)

- Big Bang: System testing only!
  - generally least effective
  - avoid this
- Integration testing is followed by functional (component/subsystem/system) testing
  - verify that all required functionality is present and correct
- Real System I/O can be a problem
  - need to follow integration and functional testing by system testing

Aspects of System Testing

- Timing, Load, Performance and Stress Testing
- Usability Testing
- Security Testing
- Storage testing
- Conversion testing
- Serviceability testing
- Procedure and document testing
  - e.g. user manuals
- Recovery testing
- Hardware configuration testing

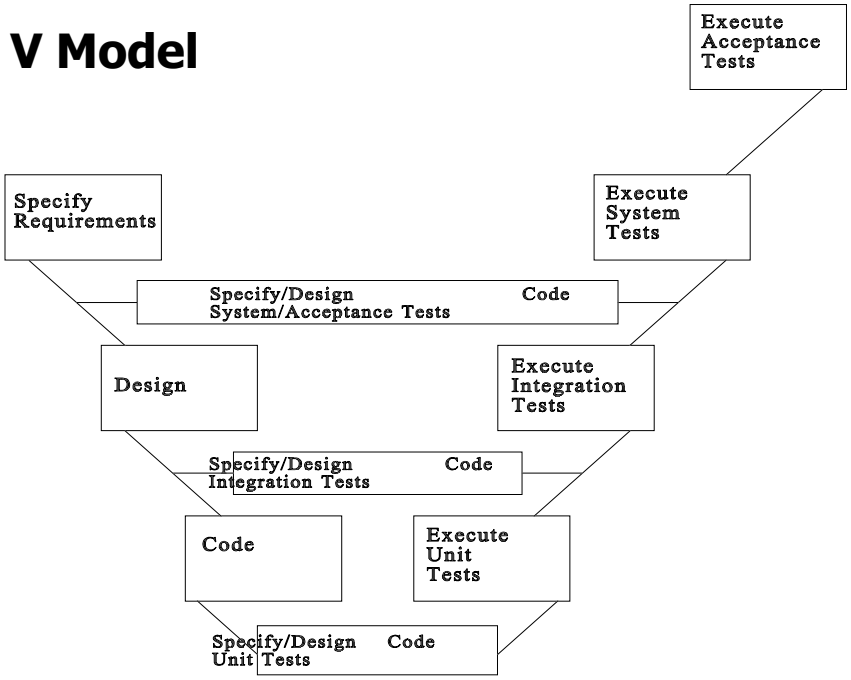
Regression Testing Issues

- Need to update test plan, test cases.
  - Existing test cases may be reusable, retestable, or obsolete
  - There is a trade-off between possible redundant testing costs and the costs of deciding which test results can be reused and which test cases can be rerun.
  - Some test cases may need to be dropped
  - new test cases may be needed
- 2 types of regression testing
  - progressive (specification has changed)
  - corrective (specification is unchanged).
- corrective modifications may require progressive regression testing if the original specifications were in error
- Version control is a major issue
  - needed on everything: code, test suites, data, documentation, level of operating system, compiler, linker, possibly hardware if changes are being made, ...

Acceptance Testing Issues

- Can be dangerous! If tests are pre-specified, software may pass tests but not do much else.
- Good acceptance testing does not involve detailed prior specification of the tests, but depends on precise, complete and correct system specifications.

V Model



Limiting Resources during testing (mostly at system testing)

- One or more of:
  - Budget
  - Schedule
  - Hardware
  - Testers
  - Debuggers
  - Fixers
  - Users

ELEC 372

Object-Oriented Testing

adapted from lectures by Dr. Terry Shepard
Royal Military College of Canada
Electrical and Computer Engineering

OO Testing: Myth and Reality (1)

- Robert Binder, Object Magazine 1995
- http://www.rbsc.com/pages/myths.html

- Myth:
  - OO testing is unnecessary. OO promotes iteration and reuse, so we have a more effective way to develop trustworthy classes.
- Reality:
  - Human error is as likely as ever. We have to check interactions, class behaviours, classes supplying functionality

OO Testing: Myth and Reality (2)

- Myth:
  - testing is a structured waterfall idea and isn't consistent with incremental OO development
- Reality:
  - tests can be designed and exercised at many points in the process
  - paradigm of "design a little, code a little" becomes "design a little, code a little, test a little"
  - Extreme Programming (XP) promotes early testing (more later)

OO Testing: Myth and Reality (3)

- Myth:
  - testing is too expensive
- Reality:
  - pay me now or pay me MUCH more later
  - failures in operational systems can cause severe secondary problems
  - proper testing is very cheap by comparison, even when done manually
  - efficient testing of large or complex systems needs some automated support

OO Testing: Myth and Reality (4)

- Myth:
  - testing is the same as conventional software
  - all you need is black-box system testing
    - define externally observable behaviour to be produced from a given input
    - don't need information about implementation
- Reality:
  - OO code structure matters
  - polymorphism, inheritance, encapsulation present opportunities for error (more later)

OO Testing: Myth and Reality (5)

- Myth:
  - conventional testing is useless for objects
- Reality:
  - there is a large body of knowledge about testing
  - basic testing techniques continue to apply with necessary changes
  - if scope of testing is small, ie classes and collaborations
    - testing techniques are very OO specific – more later
  - as scope of testing gets larger
    - testing techniques become more traditional

OO Testing: Myth and Reality (6)

- Myth:
  - inheritance means never having to say you are sorry
  - specializing from a trusted superclass means the inherited part of subclasses will also be correct
  - we don't need to retest inherited features
- Reality:
  - subclasses create new ways to misuse inherited features
  - different test cases are needed for each subclass (more later)

OO Testing: Myth and Reality (7)

- Myth:
  - reuse means never having to say you are sorry
  - reusing a trusted class means behaviour of server object is trustworthy and doesn't need to be tested
- Reality:
  - nothing prevents a new client class from using the server object incorrectly
  - all client class use of a server needs to be exercised

OO Testing: Myth and Reality (8)

- Myth:
  - black box testing is sufficient
  - designing test cases using class interface and specification assures the class is fully exercised
  - white box testing violates encapsulation
- Reality:
  - studies show black-box test suites typically only exercise 1/3 to 1/2 of the statements (let alone paths or states)
  - typically miss abnormal paths, exception and error-handling

XP's Approach to Testing

- programmer should
  - focus on testing in order to give chance to think about what code will do, independently of how it is implemented
  - write every test he/she can think of that won't run immediately on existing code
  - consider tests as giving confidence that things are working
    - reduces time spent debugging
  - consider testing as a type of metric:
    - software features that can't be demonstrated by automated tests, don't exist

XP's Design through Refactoring

- Based on test cases
  - pick up first test case
  - implement code to get test case running
  - pick up next test case
  - refactor code to get simplest implementation to get both test cases running
  - repeat for rest of test cases
- probably won't work for large systems
  - focus on testing up front is interesting

OO Testing in Four Phases

- modeling the software environment
- selecting test scenarios
  - we will look at unit testing only
- running and evaluating test scenarios
- measuring testing progress

Modeling the Software Environment (System Level Testing) (1)

- must identify and simulate interfaces software system uses
  - human interfaces
    - methods people use to communicate with software
    - must consider how to test input from different devices
  - software interfaces
    - how software uses operating system, database, runtime library
    - must test expected and unexpected activities of service classes

Modeling the Software Environment (System Level Testing) (2)

- file system interfaces
  - reads or writes to external files
  - must check to see if data from the file is appropriate in content and formatting
- communication interfaces
  - direct access to physical devices(eg. Drivers, controllers, other embedded systems)
  - must be able to generate valid and invalid protocol streams, eg packets

OO Testing in Four Phases

- modeling the software environment
- selecting test scenarios
  - we will look at unit testing (small scope testing) only
- running and evaluating test scenarios
- measuring testing progress

Selecting Test Scenarios

- consider application domain and variables
  - represents infinite number of test scenarios
  - how does tester choose a "reasonable" subset
    - generally based on coverage
      - code coverage - execution paths? (white box testing)
        - can be infinite
      - input domain coverage (black box testing)
        - not individual inputs, but input sequences
        - scenarios to which program must respond
        - also can be infinite
  - try to arrive at best possible test data adequacy criteria
    - very subjective

Example Code with two Branches (part of Myers Triangle)

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

main(argc, argv)
int argc;
char* argv[];
{
int             sideA;
int             sideB;
int             sideC;
double          s;
double          Area;

sideA = atoi(argv[1]);
sideB = atoi(argv[2]);
sideC = atoi(argv[2]);

if  (  (sideA == sideB) && (sideA == sideC )  )
{
s = 0.5 * (sideA + sideB + sideC);
Area = sqrt (s / (s - sideA) * (s - sideB) * (s - sideC)   );
printf ( "area = %g\n",  Area)  ;
}
else
puts ("not an equilateral triangle") ;
return 0;
}
```

Branch Coverage: 2 test cases needed

- To exercise the TRUE branch
  - use inputs
    - Side A = 2
    - Side B = 2
    - Side C = 2
  - output
    - area = 1.73205
- To exercise the FALSE branch
  - use inputs
    - Side A = 3
    - Side B = 4
    - Side C = 5
  - output
    - not an equilateral triangle

Two Errors not Caught by Test Cases

- wrong operand to read in sideC
  - should be argv[3]
- Heron's formula for area of a triangle
  - Area = [s (s - sideA) (s - sideB) (s - sideC)]
  - code has division operator instead of multiplication
  - 2 is the only value for which the wrong equation gives the right answer

Myers example in OO form

- Create class, eg. with methods:
  - setTriangle(A, B, C)
  - isEquilateral
  - getArea
- main() needs to be rewritten so it uses the new class
- test class, and then test main()

Other Considerations for Selecting Test Scenarios for Testing OO Code (1)

- Inheritance
  - need to retest inherited methods, even if not changed in subclass
    - methods A and B in superclass
    - subclass inherits A and B and adds method C
    - need to see that C uses A and B correctly
    - not sufficient to test A and B in superclass and C in subclass

Testing a Subclass

1. If any method in a subclass is different from the superclass (ie signature, semantics different in any way)
   - then must test each method and all its interactions
2. If methods have the same signature but different implementations from the superclass
   - then the methods and all their interactions have to be tested
3. If methods are inherited from the superclass (ie definition and implementation are same as in the superclass)
   - then test the methods when you test the superclass
   - if methods interact with methods of the subclass
     - then must test the interactions

Other Considerations for Selecting Test Scenarios for Testing OO Code (2)

- testing objects
  - exercise methods provided by object with goals of uncovering errors in implementation, state of object, or both
  - must test interaction between methods in object
  - must search for all patterns of method invocation of object
    - methods can be invoked in any random order leading to a large number of combinations

Other Considerations for Selecting Test Scenarios for Testing OO Code (3)

- objects have state
  - test all methods one-by-one against set of all states object can take
  - impractical!
- Approaches (not OO specific):
  - Use equivalence classes
    - identify those groups of values such that if we test one member of the group, fairly sure that outcome will be the same for all other members of the group
    - group formation done through examining input data
      - may want to check the code
  - Use interval testing
    - eg test data structures for empty, full, one member, etc.
    - eg test a parameter whose normal values are {1, …, 100}

OO Testing in Four Phases

- modeling the software environment
- selecting test scenarios
  - we will look at unit testing only
- running and evaluating test scenarios
- measuring testing progress

Running and Evaluating Test Scenarios (1)

- manually executing test scenarios is labour-intensive and error prone
  - automate as appropriate
    - James Bach says automated tests only find 15% of errors
      - manual tests find other 85% !!
  - often include extra code to provide information about variables, object properties, etc during test runs
  - write code for testability: controllability and observability

Running and Evaluating Test Scenarios (2)

- evaluation of test results
  - comparison of software's actual output to its expected output
  - often difficult to do
    - eg. Human oracle visually monitoring screen output and painstakingly analyzing data
      - however, human can spot glitches in test data, test results, suite of test cases
    - eg. Expected output difficult to accurately predict or generate (such as expected output for simulation programs for processes at a nuclear generating station)

Measuring Testing Progress

- determining
  - when to stop testing
    - usually when you run out of time and/or resources
  - when the product is ready to release
    - based on business judgement
- need quantitative measures
  - estimation of number of bugs left in software
    - based on
      - rate at which bugs are still being found
      - where in software bugs are being found
      - rate of bug fixing
    - probability any of these bugs will be discovered in the field