CS 501: Software Engineering
Fall 2000

Lecture 22

Dependable Systems II

Validation and Verification

Administration

Final Presentation

- Completed code, demonstration of operational system
- Program documentation
- User documentation

Client should be present at final presentation

Defensive Programming

Murphy's Law: If anything can go wrong, it will.

Defensive Programming:

- Redundant code is incorporated to check system state after modifications
- Implicit assumptions are tested explicitly

Defensive Programming Examples

- Use *boolean* variable not *integer*
- Test *i* <= *n* not *i* = = *n*
- Assertion checking
- Build debugging code into program with a switch to display values at interfaces
- Error checking codes in data, e.g., checksum or hash

Terminology

Fault avoidance

Build systems with the objective of creating fault-free systems

Fault tolerance

Build systems that continue to operate when faults occur

Fault detection (testing and validation)

Detect faults before the system is put into operation.

Fault Tolerance

Basic Techniques:

- After error continue with next transaction
- Timers and timeout in networked systems
- Error correcting codes in data
- Bad block tables on disk drives
- Forward and backward pointers

Report all errors for quality control

Fault Tolerance

Backward Recovery:

- Record system state at specific events (checkpoints). After failure, recreate state at last checkpoint.
- Combine checkpoints with system log that allows transactions from last checkpoint to be repeated automatically.

Fault Tolerance

General Approach:

- Failure detection
- Damage assessment
- Fault recovery
- Fault repair

N-version programming -- Execute independent implementation in parallel, compare results, accept the most probable.

Validation and Verification

Validation: Are we building the right product?

Verification: Are we building the product right?

In practice, it is sometimes difficult to distinguish between the two.

That's not a bug. That's a feature!

Cleanroom Software Development

Software development process that aims to develop zero-defect software.

- Formal specification
- Incremental development with customer input
- Constrained programming options
- Static verification
- Statistical testing

It is always better to prevent defects than to remove them later.

Example: The four color problem.

Static and Dynamic Verification

Static verification: Techniques of verification that do not include execution of the software.

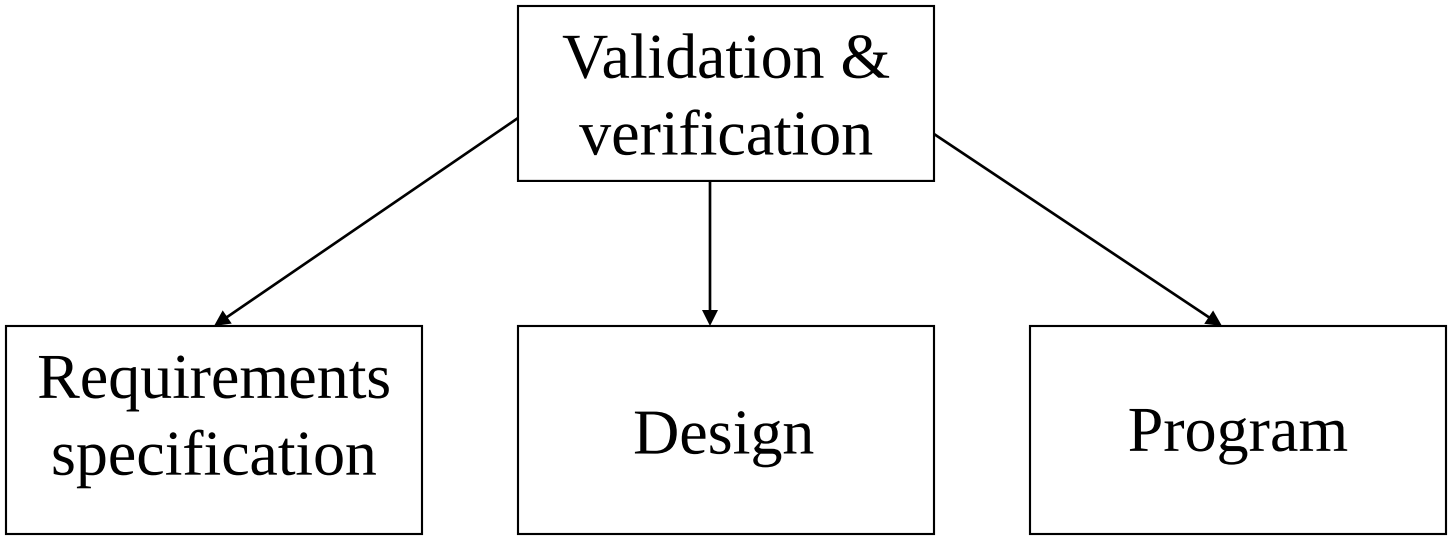
- May be manual or use computer tools.

Dynamic verification

- Testing the software with trial data.
- Debugging to remove errors.

Static Validation & Verification

Carried out throughout the software development process.



Static Verification: Program Inspections

Program reviews whose objective is to detect faults

- Code may be read or reviewed line by line.
- 150 to 250 lines of code in 2 hour meeting.
- Use checklist of common errors.
- Requires team commitment, e.g., trained leaders

So effective that it can replace unit testing

Inspection Checklist: Common Errors

Data faults: Initialization, constants, array bounds, character strings

Control faults: Conditions, loop termination, compound statements, case statements

Input/output faults: All inputs used; all outputs assigned a value

Interface faults: Parameter numbers, types, and order; structures and shared memory

Storage management faults: Modification of links, allocation and de-allocation of memory

Exceptions: Possible errors, error handlers

Static Analysis Tools

Program analyzers scan the source of a program for *possible* faults and anomalies (e.g., Lint for C programs).

- Control flow: loops with multiple exit or entry points
- Data use: Undeclared or uninitialized variables, unused variables, multiple assignments, array bounds
- Interface faults: Parameter mismatches, non-use of functions results, uncalled procedures
- Storage management: Unassigned pointers, pointer arithmetic

Static Analysis Tools (continued)

- Cross-reference table: Shows every use of a variable, procedure, object, etc.
- Information flow analysis: Identifies input variables on which an output depends.
- Path analysis: Identifies all possible paths through the program.

Test Design

Testing can never prove that a system is correct. It can only show that (a) a system is correct in a special case, or (b) that it has a fault.

- The objective of testing is to find faults.
- Testing is never comprehensive.
- Testing is expensive.

Testing and Debugging

Testing is most effective if divided into stages:

- Unit testing at various levels of granularity
  - tests by the developer
  - emphasis is on accuracy of actual code
- System and sub-system testing
  - uses trial data
  - emphasis is on integration and interfaces
- Acceptance testing
  - uses real data in realistic situations
  - emphasis is on meeting requirements

Acceptance Testing

Alpha Testing: Clients operate the system in a realistic but non-production environment

Beta Testing: Clients operate the system in a carefully monitored production environment

Parallel Testing: Clients operate new system alongside old production system with same data and compare results

The Testing Process

System and Acceptance Testing is a major part of a software project

- It requires time on the schedule
- It may require substantial investment in datasets, equipment, and test software.
- Good testing requires good people!
- Management and client reports are important parts of testing.

What is the definition of "done"?

Testing Strategies

- Bottom-up testing. Each unit is tested with its own test environment.
- Top-down testing. Large components are tested with dummy stubs.

  user interfaces
  work-flow
  client and management demonstrations

- Stress testing. Tests the system at and beyond its limits.

  real-time systems
  transaction processing

Test Cases

Test cases are specific tests that are chosen because they are likely to find faults.

Test cases are chosen to balance expense against chance of finding serious faults.

- Cases chosen by the development team are effective in testing known vulnerable areas.
- Cases chosen by experienced outsiders and clients will be effective in finding gaps left by the developers.
- Cases chosen by inexperienced users will find other faults.

Test Case Selection: Coverage of Inputs

Objective is to test all classes of input

- Classes of data -- major categories of transaction and data inputs.

Cornell example: (undergraduate, graduate, transfer, ...) by (college, school, program, ...) by (standing) by (...)

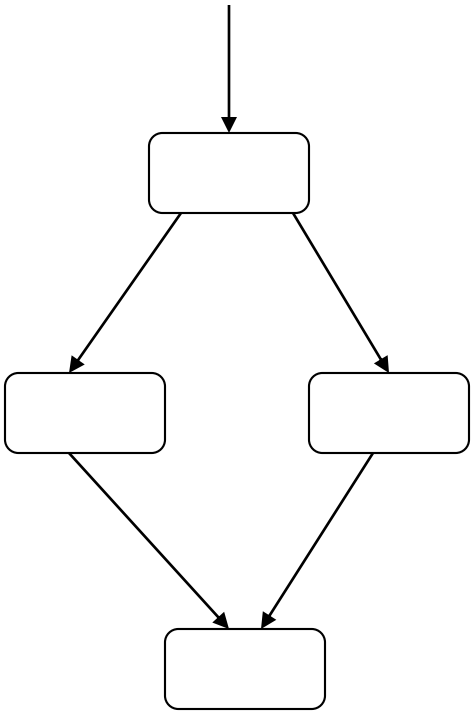
- Ranges of data -- typical values, extremes
- Invalid data, reversals, and special cases.

Test Case Selection: Program

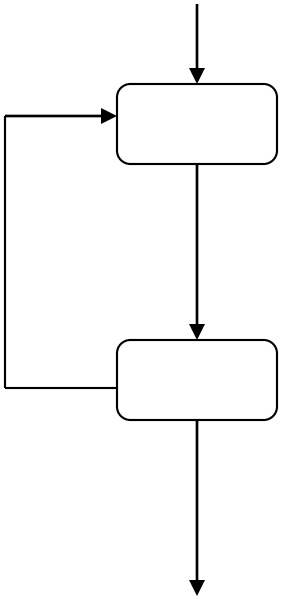
Objective is to test all functions of each computer program

- Paths through the computer programs
  - Program flow graph
  - Check that every path is executed at least once
- Dynamic program analyzers
  - Count number of times each path is executed
  - Highlight or color source code
  - Can not be used with time critical software

Program Flow Graph



if-then-else



loop-while

Fixing Bugs

- Isolate the bug
  - Intermittent --> repeatable
  - Complex example --> simple example
- Understand the bug
  - Root cause
  - Dependencies
  - Structural interactions
- Fix the bug
  - Design changes
  - Documentation changes
  - Code changes

Moving the Bugs Around

Fixing bugs is an error-prone process!

- When you fix a bug, fix its environment
- Bug fixes need static and dynamic testing
- Repeat all tests that have the slightest relevance (regression testing)

Bugs have a habit of returning!

- When a bug is fixed, add the failure case to the test suite for the future.

Regression Testing

Applied to modified software to provide confidence that modifications behave as intended and do not adversely affect the behavior of unmodified code.

- Basic technique is to repeat entire testing process after every change, however small.

Real Time Software Development

Testing and debugging need special tools and environments

- Debuggers, etc., can not be used to test real time performance
- Simulation of environment may be needed to test interfaces -- e.g., adjustable clock speed
- General purpose tools may not be available

Software Engineering for Real Time

The special characteristics of real time computing require extra attention to good software engineering principles:

- Requirements analysis and specification
- Development of tools
- Modular design
- Exhaustive testing

Heroic programming will fail!

Some Notable Bugs

- Built-in function in Fortran compiler ($e^0 = 0$)
- Japanese microcode for Honeywell DPS virtual memory
- The microfilm plotter with the missing byte (1:1023)
- The Sun 3 page fault that IBM paid to fix
- Left handed rotation in the graphics package

Good people work around problems.
The best people track them down and fix them!