Basic Embedded Software Engineering: The Software Lifecycle

These lecture notes created by Mr. James B. (Jim) Carlson, NCSU

Learning Objectives

- Based on requirements specifications, develop
  - high-level design specs with tests
  - detailed design specs with tests
  - other testing as needed
- Understand how to provide and verify traceability across the V throughout the process
- Learn how to perform design reviews and code walkthroughs

Lecture Topics

- Software lifecycle V-model overview
- Implementing the V
  - Requirements specification
  - High-level module and test design (state-based, control-flow-based)
  - Detailed module and test design
  - Code Design Reviews
  - Unit testing
  - Integration testing

*Portions derived from **Software Engineering: A Practitioner's Approach,** 6th Edition, Roger S. Pressman, McGraw-Hill*

Software Engineering Definitions

- "The establishment and use of sound engineering principles in order to ***obtain economically*** software that is ***reliable*** and ***works efficiently*** on ***real machines.***" [Fritz Bauer 1969]
  - *Goals*

- "The application of a ***systematic***, ***disciplined***, ***quantifiable*** approach to the ***development***, ***operation*** and ***maintenance*** of software; that is, the application of engineering to software." [IEEE 1993]
  - *Methods*

- How can we make software *correct enough* without making the company go bankrupt?
  - Need to be able to develop software efficiently – *this requires a set of sound processes* built into the company

Capability Maturity Model (CMM)

- CMM developed by CMU's Software Engineering Institute (SEI) to define maturity of software development processes
- **Level 1: Initial.**
  - Ad hoc and occasionally even chaotic.
  - Few processes defined.
  - Success often requires "heroes". Overtime, weekends, no vacation.
- **Level 2: Repeatable.**
  - Basic project management processes track cost, schedule and functionality.
- **Level 3: Defined.**
  - Processes for management and engineering are documented, standardized, and integrated into overall organization software process.
- **Level 4: Managed.**
  - Software process and product quality are measured quantitatively.
  - Metrics are used to adjust process.
- **Level 5: Optimizing.**
  - Continuous process improvement is performed.
  - Innovative ideas are applied and evaluated using metrics.

Real World: CIMM

- What could possibly be worse than Level 1 (*ad hoc* and *chaotic*)?
- Capability Im-Maturity Model, developed by Capt. Tom Schorsch, U.S. Air Force
- The CIMM adds maturity levels below Level 1
- **Level 0: Negligent**
  - "Indifference:
    - Failure to allow successful development process to succeed.
    - All problems are perceived to be technical problems.
    - Managerial and quality assurance activities are deemed to be overhead and superfluous to the task of software development process.
    - Reliance on silver pellets."
- **Level -1: Obstructive**
  - "Counterproductive:
    - Counterproductive processes are imposed.
    - Processes are rigidly defined and adherence to the form is stressed.
    - Ritualistic ceremonies abound.
    - Collective management precludes assigning responsibility.
    - Status quo forever."

Real World: CIMM

- **Level -2: Contemptuous**
  - “Arrogance:
    - Disregard for good software engineering institutionalized.
    - Complete schism between software development activities and software process improvement activities.
    - Complete lack of a training program.”
- **Level -3: Undermining**
  - “Sabotage:
    - Total neglect of own charter,
    - conscious discrediting of peer organizations software process improvement efforts.
    - Rewarding failure and poor performance.”
- http://www.stsc.hill.af.mil/crosstalk/1996/11/xt96d11h.asp
- Finkelstein, "A Software Process Immaturity Model," *ACM SIGSOFT, Software Engineering Notes*, Vol. 17, No. 4, October 1992, pp. 22-23.

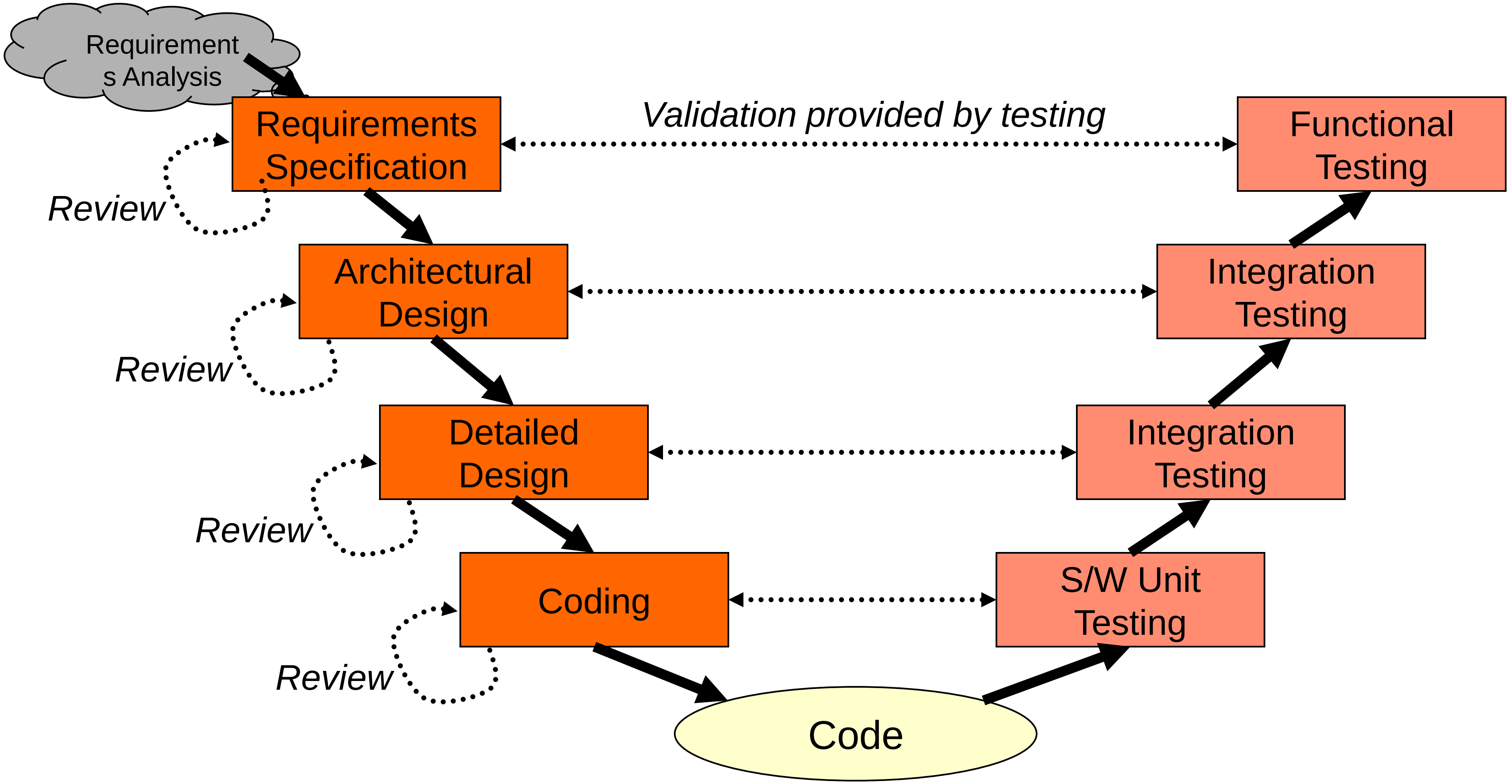
Good Enough Software

- How do we make software *correct enough* without going bankrupt?
  - Need to be able to develop (and test) software efficiently
- Follow a good plan
  - Start with customer requirements
  - Design architectures to define the building blocks of the systems (tasks, modules, etc.)
  - Add missing requirements
    - Fault detection, management and logging
    - Real-time issues
    - Compliance to a firmware standards manual
    - Fail-safes
  - Create detailed design
  - Implement the code, following a good development process
    - Perform frequent design and code reviews
    - Perform frequent testing (unit and system testing, preferably automated)
    - Use revision control to manage changes
  - Perform post-mortems to improve development process

Software Lifecycle Concepts

- Coding is the most visible part of a software development process but is not the only one
- Before we can code, we must know
  - What must the code do? *Requirements specification*
  - How will the code be structured? *Design specification*
    - *(only at this point can we start writing code)*
- How will we know if the code works? *Test plan*
  - Best performed when defining requirements
- The software will likely be enhanced over time
  - Corrections, adaptations, enhancements & preventive maintenance

V Model Overview



- Themes:
  - Link front and back ends of life-cycle for efficiency
  - Provide "traceability" to ensure nothing falls through the cracks

Implementing the V

1. Requirements specification
   - Define what the system must do
2. Architectural (high-level) module and test design (state-based, control-flow-based)
3. Detailed module and test design
4. Coding and Code Inspections

1. Requirements Specification and Validation Plan

- Result of Requirements Analysis
- Should contain:
  - *Introduction* with goals and objectives of system
  - *Description of problem* to solve
  - *Functional description*
    - provides a "processing narrative" per function
    - lists and justifies design constraints
    - explains performance requirements
  - *Behavioral description* shows how system reacts to internal or external events and situations
    - State-based behavior
    - General control flow
    - General data flow
  - *Validation criteria*
    - tell us how we can decide that a system is acceptable. (*Are we there yet?*)
    - is the foundation for a validation test plan
  - *Bibliography and Appendix* refer to all documents related to project and provide supplementary information

Specification Guidelines

- Use a layered format that provides increasing detail with Deeper levels
- Use consistent graphical notation and apply textual terms consistently (stay away from aliases)
- Be sure to define all acronyms
- Write in a simple, unambiguous style
- Avoid vague terms (sometimes, often, etc.)
- Beware of incomplete lists
- Avoid passive voice which hides responsibility
- Avoid dangling pronouns
- Beware of implied requirements (e.g. numerical base)

Specification Review

- High-level review to ensure specification is
  - complete
  - consistent
  - accurate
- Difficult to “test” specification, so need to consider and discuss requirements interactively

Implementing the V

1. Requirements specification
2. Architectural (high-level) module and test design (state-based, control-flow-based)
   - Define big-picture view of what pieces will exist, how they will fit together, and how they will be tested
3. Detailed module and test design
4. Coding and Code Inspections

2. Architectural (High-Level) Design

- Architecture defines the structure of the system
  - Components
  - Externally visible properties of components
  - Relationships among components
- Architecture is a representation which lets the designer…
  - Analyze the design's effectiveness in meeting requirements
  - Consider alternative architectures early
  - Reduce down-stream implementation risks
- Architecture matters because…
  - It's small and simple enough to fit into a single person's brain (as opposed to comprehending the entire program's source code)
  - It gives stakeholders a way to describe and therefore discuss the system

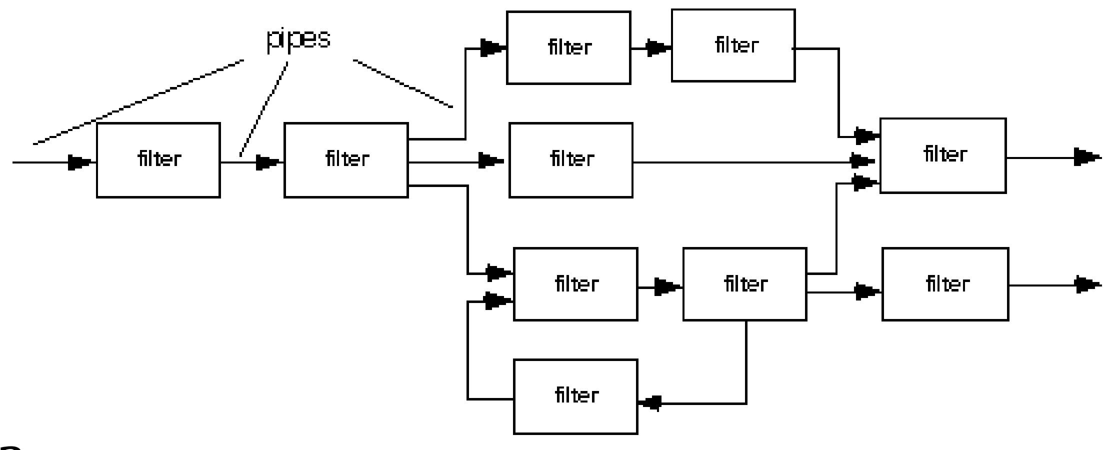
Architectural Styles for Software

- Several general styles or patterns exist
- Each describes
  - *Components* which perform some function
  - *Connectors* with which components communicate, coordinate and cooperate
  - *Constraints* which define how components fit together
  - *Semantic Models* which allow designer to determine system behavior by composing component and connector behavior

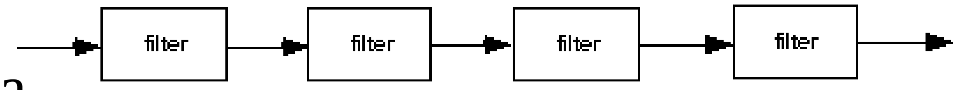
Architectural Patterns

- Concurrency—applications must handle multiple tasks in a manner that simulates parallelism
  - *operating system process management* pattern
  - *task scheduler* pattern
- Persistence—Data persists if it survives past the execution of the process that created it. Two patterns are common:
  - a *database management system* pattern that applies the storage and retrieval capability of a DBMS to the application architecture
  - an *application level persistence* pattern that builds persistence features into the application architecture
- Distribution— the manner in which systems or components within systems communicate with one another in a distributed environment
  - A *broker* acts as a 'middle-man' between the client component and a server component.

Data-Flow Architectures



(a) pipes and filters



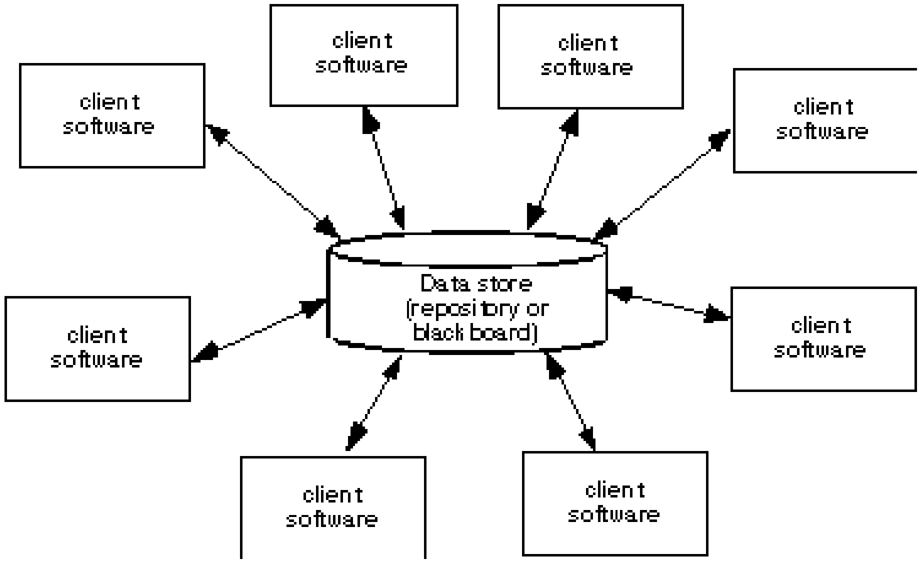
(b) batch sequential

- Transforms input data into output data
- Pipes and filters
  - Pipes communicate data
  - Filters transform data
- Filters operate independently of each other
- Commonly used for signal processing

Object-Oriented Architectures

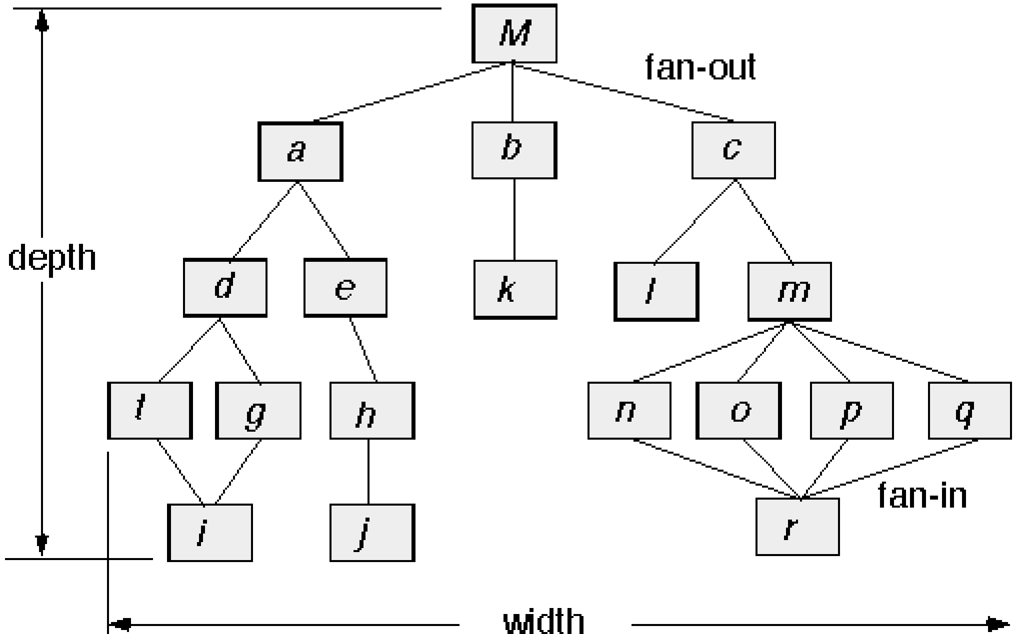
- Components contain data and operations
- Message passing used to communicate and coordinate among components
- Many approaches possible

Data-Centered Architectures



- Central data repository is core component
- Other components (clients) modify, add, or delete data.
  - Each client is independent process
- Repository style
  - Passive: client software gets data regardless of data changes or activities of other clients
  - Blackboard: repository sends change notifications to clients
- Easy to add client components without affecting other clients

Call and Return Architectures



- Represents hierarchical nesting of control in program
  - "main calls get_data which calls convert_input…"
- Scalability and modification straightforward
- Sub-types of Call and Return architectures
  - main program/subprogram: single processor
  - remote procedure call: subprograms may be distributed across remote processors

Data Design

- Important data should be clearly specified and described
- Principles
  - Use Data Dictionary to define data and program design
  - Defer low-level decisions until late in design process
  - Use abstraction and information hiding to separate logical and physical views of system
  - Develop a library of useful data structures and operations for reuse

Implementing the V

1. Requirements specification
2. Architectural (high-level) module and test design (state-based, control-flow-based)
3. Detailed module and test design
   - Now we design the software, using flow charts or finite state machines
   - Also define module tests (discussed at end of lecture)
4. Coding and Code Inspections

4. Coding and Code Inspections

- Coding driven directly by Detailed Design Specification
- Use a version control system while developing the code
- Follow a coding standard
  - Eliminate stylistic variations which make understanding code more difficult
  - Avoid known questionable practices
  - Spell out best practices to make them easier to follow
- Perform code reviews
- Test effectively
  - Automation
  - Regression testing

Version Control

- Use a version control system
  - "Time machine" for coding
  - Procedure
    - Check out code module
    - Do your programming and testing
    - Update the module (in case anyone else has changed it)
    - Check code module into the repository when the code works (commit)
  - Examples: CVS, RCS, SourceSafe
- Use regression tests
  - Verify the code works before checking it into the repository
    - Use test cases which exercise as much of the code as possible
    - Include
  - Automation extremely useful
    - Shell scripts, etc.
    - Can even automate finding the bugs sometimes

Peer Code Review

- Inspect the code before testing it
- Extensive positive industry results from code inspections
  - IBM removed 82% of bugs
  - 9 hours saved by finding each defect
  - For AT&T quality rose by 1000% and productivity by 14%
- Finds bugs which testing often misses
  - 80% of the errors detected by HP's inspections were unlikely to be caught by testing
  - HP, Shell Research, Bell Northern, AT&T: inspections 20-30x more efficient than testing

Types of Peer Code Review (1)

- Code inspection
  - IBM, Michael Fagan, 1976
  - Reader (not author) explains what code seems to do
  - Formal, heavy-weight process for reviewing code
    - Preparation stage involves participants reading code
    - Measure where all bugs are, classify them, how quickly bugs are found, etc.
  - Author fixes all problems after review
- Over-the-shoulder review
  - Author explains to reviewer what changed and why
    - Very easy to implement
  - Fix small small problems immediately, large problems afterwards
  - Very light-weight process has disadvantages
    - No way for manager to ensure all changes are reviewed
    - Easy for author to miss a change
    - No way to track nature of bugs, how efficient this process is
    - No way for reviewer to verify bugs were fixed

Types of Peer Code Review (2)

- E-mail pass-around
  - Author emails files/change logs to reviewers for discussion (by email)
  - Can use version control system to determine automatically what has changed
  - Advantages
    - Easy to implement, easy to add reviewers
    - Simple scheduling
  - Disadvantages
    - Scales badly – gets hard to track threads and changes for large projects
    - Email slows down communication
    - Doesn’t ensure all changes are reviewed
    - Doesn’t ensure all fixes are made and are correct
- Tool-assisted reviews
  - Use special review tool
    - Automatically gathers files, integrated with version control system
    - Provides combined display of file differences, comments and defects
    - Automatically collects metrics (e.g. lines inspected per hour, defects found per hour, defects found per thousand lines of code)
    - Allows monitoring of which changes have been reviewd, and which bugs have been fixed
    - Presents efficient specialized interfaces for coders, reviewers and managers

Types of Peer Code Review (3)

- Pair-programming
  - Two developers sit at one workstation
  - Only one types at a time
  - Advantages
    - Very effective at finding bugs and transferring knowledge
    - Reviewer is very familiar with code, given long-term involvement with development
  - Disadvantage
    - Reviewer may be too involved to see big-picture issues (may need other form of review)
    - Takes a lot of time (two whole programmers, full-time)
    - Doesn’t work well with remote developers

Testing: Integrate Test Definition with Designing

- We now revisit specification and design documents to insert testing plan.
- In practice, we would do them concurrently.
- Levels
  - Requirements Specification and Validation Plan
  - High-Level Design and Test Plan
  - Detailed Design and Test Plan

Software Testing Challenge

- Complete testing is impossible
  - There are too many possible inputs
    - Valid inputs
    - Invalid inputs
    - Different timing on inputs
  - There are too many possible control flow paths
    - Conditionals, loops, switches, interrupts…
    - And you would need to retest after every bug fix
    - Combinatorial explosion
  - Some design errors can’t be found through testing
    - Specifications may be wrong
  - You can’t prove programs correct using logic
    - Even if the program completely matches the specification, the spec may still be wrong
  - User interface (and design) issues are too complex

So Why Test Software?

- Testing IS NOT "the process of verifying the program works correctly"
  - You can't verify the program works correctly
  - The program probably won't work correctly in all possible cases
    - Professional programmers have 1-3 bugs per 100 lines of code after it is "done"
  - Testers shouldn't try to prove the program works
    - If you want and expect your program to work, you'll unconsciously miss failure because human beings are inherently biased
- The purpose of testing is to find problems quickly
  - Does the software violate the specifications?
  - Does the software violate unstated requirements?
- The purpose of finding problems is to fix them
  - Then fix the most important problems, as there isn't enough time to fix all of them
  - The *Pareto Principle* defines "the vital few, the trivial many"
    - Bugs are uneven in frequency – a vital few contribute the majority of the program failures. Fix these first.

Approaches to Testing

- Incremental Testing
  - Code a function and then test it (*module/unit/element testing*)
  - Then test a few working functions together (*integration testing)*
    - Continue enlarging the scope of tests as you write new functions
  - Incremental testing requires extra code for the *test harness*
    - A *driver* function calls the function to be tested
    - A *stub* function might be needed to simulate a function called by the function under test, and which returns or modifies data.
    - The test harness can *automate* the testing of individual functions to detect later bugs
- Big Bang Testing
  - Code up all of the functions to create the system
  - Test the complete system
    - Plug and pray

Why Test Incrementally?

- Finding out what failed is much easier
  - With Big Bang, since no function has been thoroughly tested, most probably have bugs
  - Now the question is “Which bug in which module causes the failure I see?”
  - Errors in one module can make it difficult to test another module
    - Errors in fundamental modules (e.g. kernel) can appear as bugs in other many other dependent modules
- Less finger pointing = happier SW development team
  - It’s clear who made the mistake, and it’s clear who needs to fix it
- Better automation
  - Drivers and stubs initially require time to develop, but save time for future testing

Clear Box (White Box) Testing

- How?
  - Exercise code based on *knowledge of how program is written*
  - Some tools enable automatic branch/code coverage analysis
    - If requirements state that all code be executed, this coverage information is essential
- Subcategories
  - Condition Testing
    - Test a variation of each condition in a function
      - True/False condition requires two tests
      - Comparison condition requires three tests
        - » A>B? A < B, A == B, A > B
    - Compound conditions
      - E.g. (n>3) && (n != 343)
  - Loop Testing
    - Ensure code works regardless of number of loop iterations
    - Design test cases so loop executes 0, 1 or maximum number of times
    - Loop nesting or dependence requires more cases

Black Box Testing

- Complement to white box testing
- Goal is to find
  - Incorrect or missing functions
  - Interface errors
  - Errors in data structures or external database access
  - Behavioral or performance errors
  - Initialization and termination errors
- Want each test to
  - Reduce the number of additional tests needed for reasonable testing
  - Tell us about presence or absence of a class of errors

Perform Project Postmortems

- Goals – improve your engineering processes
  - extract all useful information learned from the just-completed project – provide "virtual experience" to others
  - provide positive non-confrontational feedback
  - document problems and solutions clearly and concisely for future use
- Basic rule: problems need solutions
  - A postmortem is not…
    - a forum for assigning blame to engineers
    - a way to complain about management
    - a place to whine
- Often small changes improve performance, but are easy to forget

Example Postmortem Structure

- Product
  - Bugs
  - Software design
  - Hardware design
- Process
  - Code standards
  - Code interfacing
  - Change control
  - How we did it
  - Team coordination
- Support
  - Tools
  - Team burnout
  - Change orders
  - Personnel availability