Software Testing

Big Picture, Major Concepts and Techniques

Suppose you are asked:

- Would you trust a completely automated nuclear power plant?
- Would you trust a completely automated pilot?
  - What if the software was written by you?
  - What if it was written by a colleague?
- Would you dare to write an expert system to diagnose cancer?
  - What if you are personally held liable in a case where a patient dies because of a malfunction of the software?

State of the Art

- Currently the field cannot deliver fault-free software
  - Studies estimate 30-85 errors per 1000 LOC
    - Most found/fixed in testing
  - Extensively-tested software: 0.5-3 errors per 1000 LOC
- Testing is postponed, as a consequence: the later an error is discovered, the more it costs to fix it (Boehm: 10-90 times higher)
- More errors in design (60%) compared to implementation (40%).
  - 2/3 of design errors not discovered until after software operational

Testing

- Should not wait to start testing until after implementation phase
- Can test SRS, design, specs
  - Degree to which we can test depends upon how formally these documents have been expressed
- **Testing software shows only the presence of errors, not their absence**

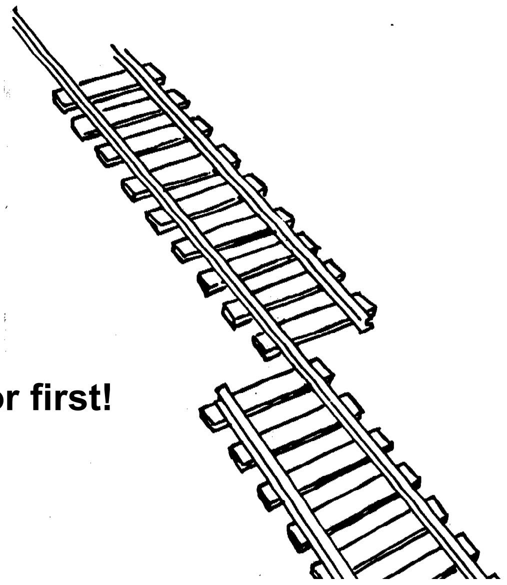
Testing

- Could show absence of errors with Exhaustive Testing
  - Test all possible outcomes for all possible inputs
  - Usually not feasible even for small programs
- Alternative
  - Formal methods
  - Can prove correctness of software
  - Can be very tedious
  - **Partial coverage testing**

Terminology

- **Reliability:** The measure of success with which the observed behavior of a system confirms to some specification of its behavior.
- **Failure:** Any deviation of the observed behavior from the specified behavior.
- **Error:** The system is in a state such that further processing by the system will lead to a failure.
- **Fault (Bug or Defect):** The mechanical or algorithmic cause of an error.
- **Test Case**: A set of inputs and expected results that exercises a component with the purpose of causing failures and detecting faults

What is this?



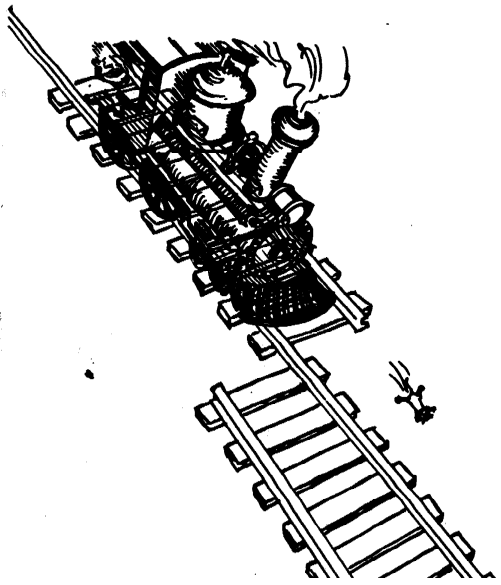
A failure?

An error?

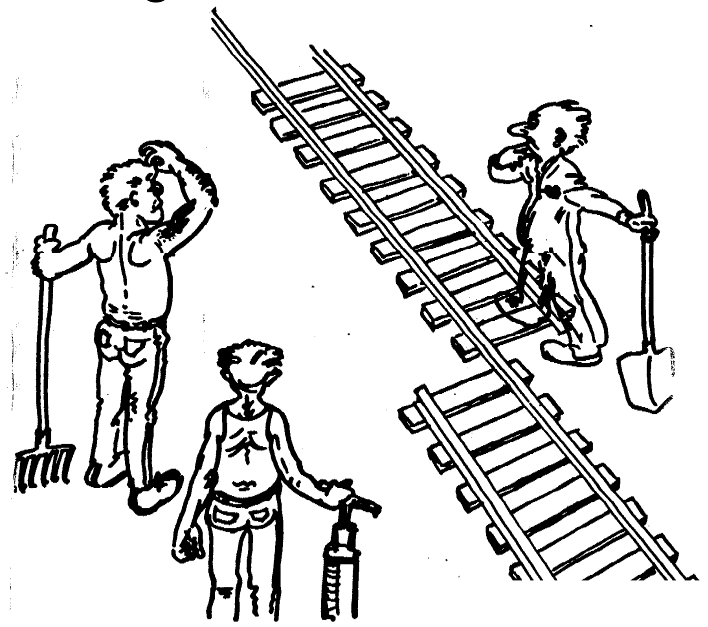
A fault?

Need to specify the desired behavior first!

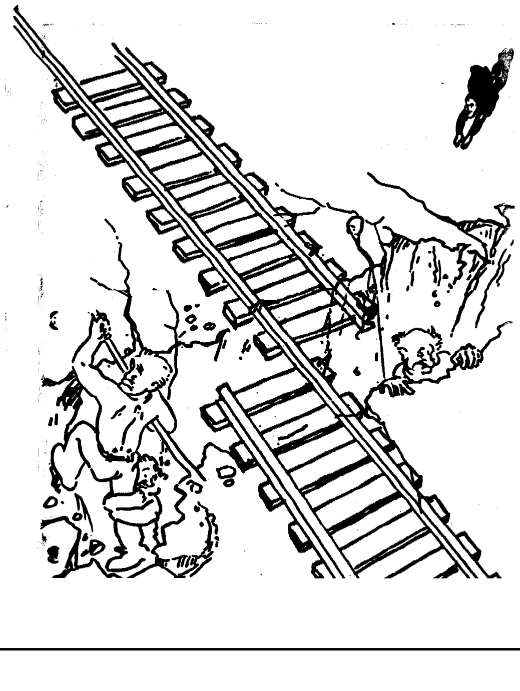
Erroneous State ("Error")



Algorithmic Fault

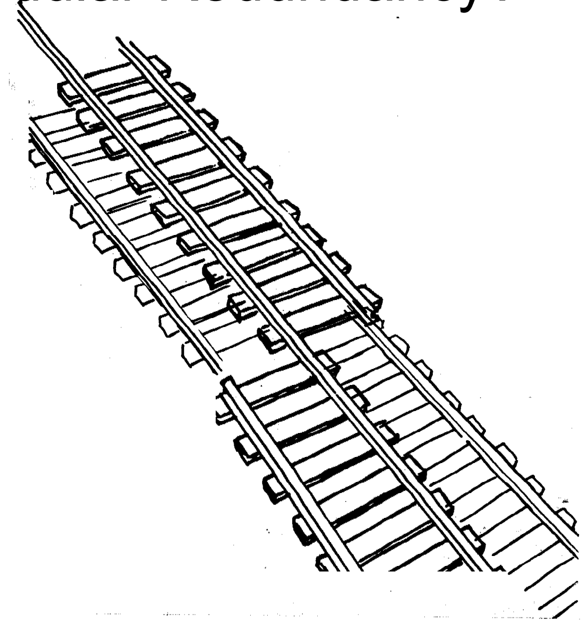


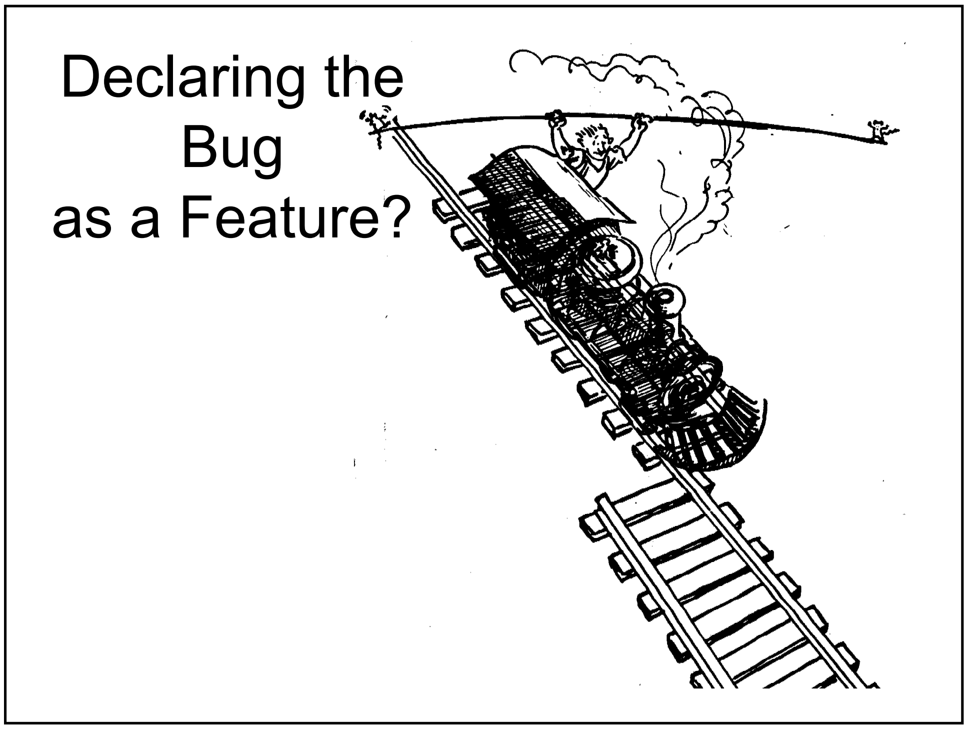
Mechanical Fault



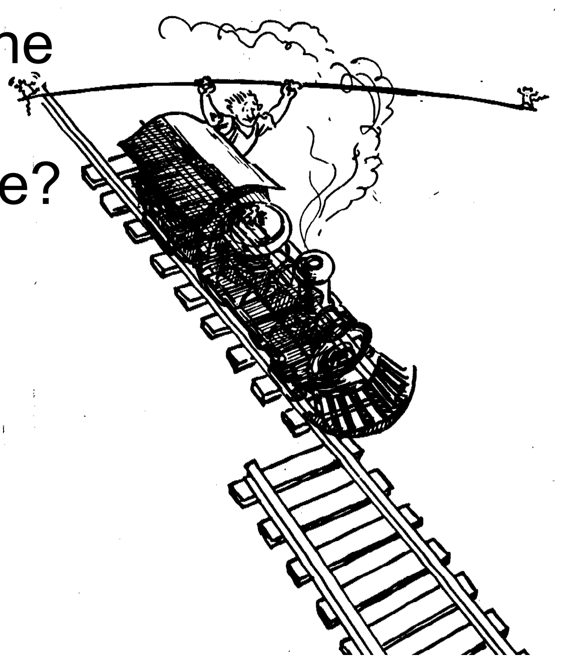
How do we deal with Errors and Faults?

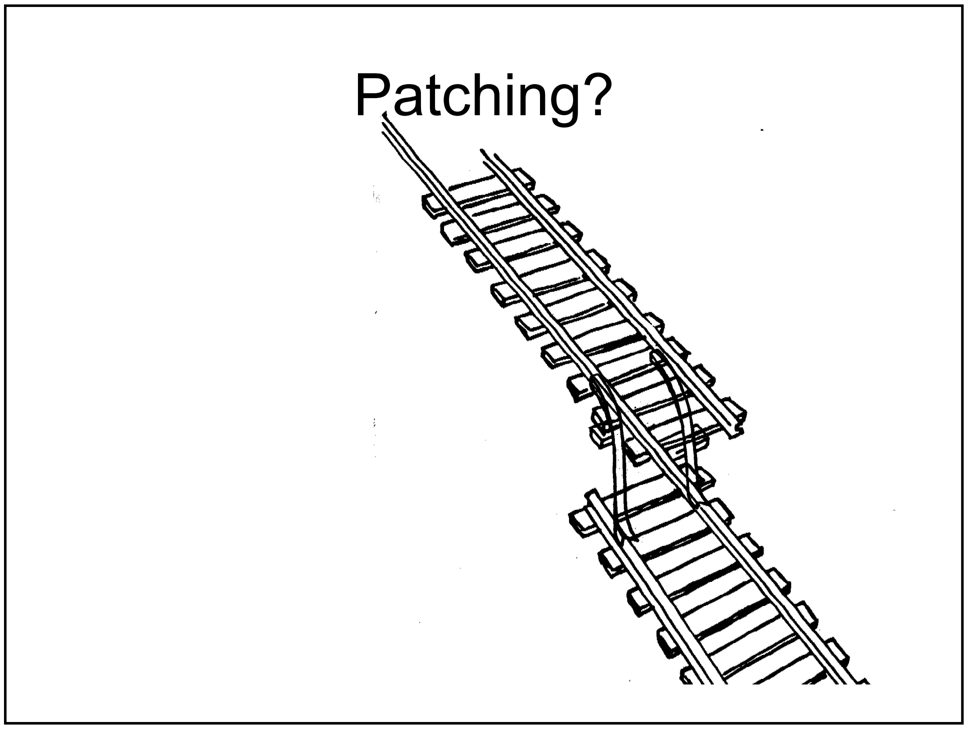
Modular Redundancy?



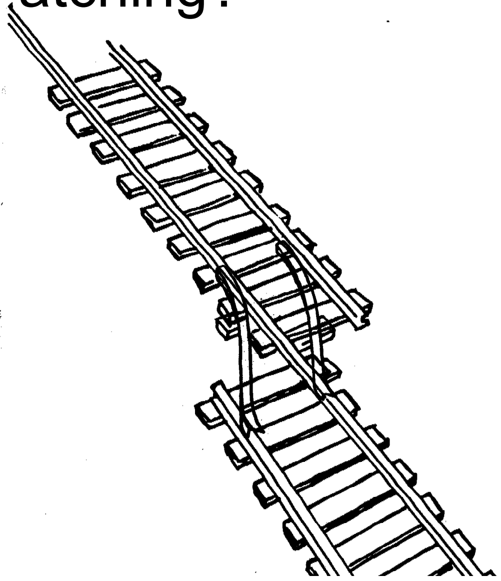


Declaring the Bug as a Feature?

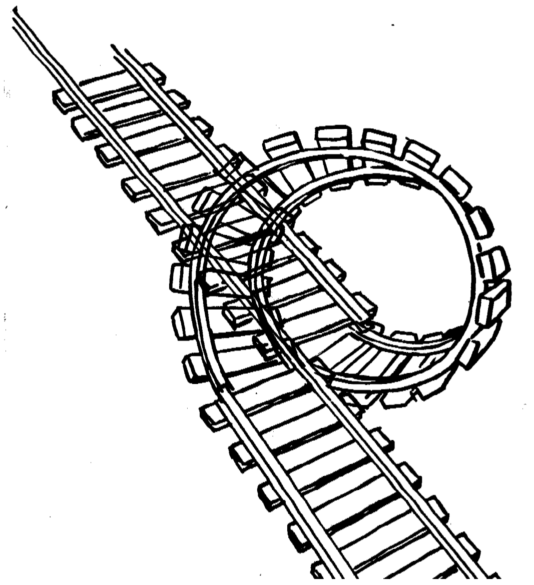




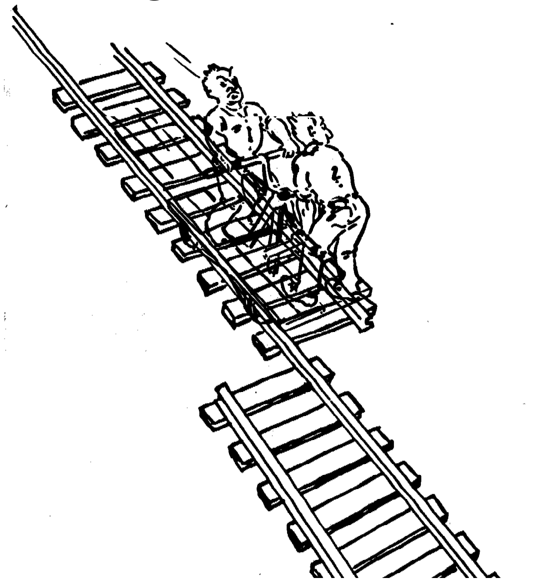
Patching?



Verification?



Testing?



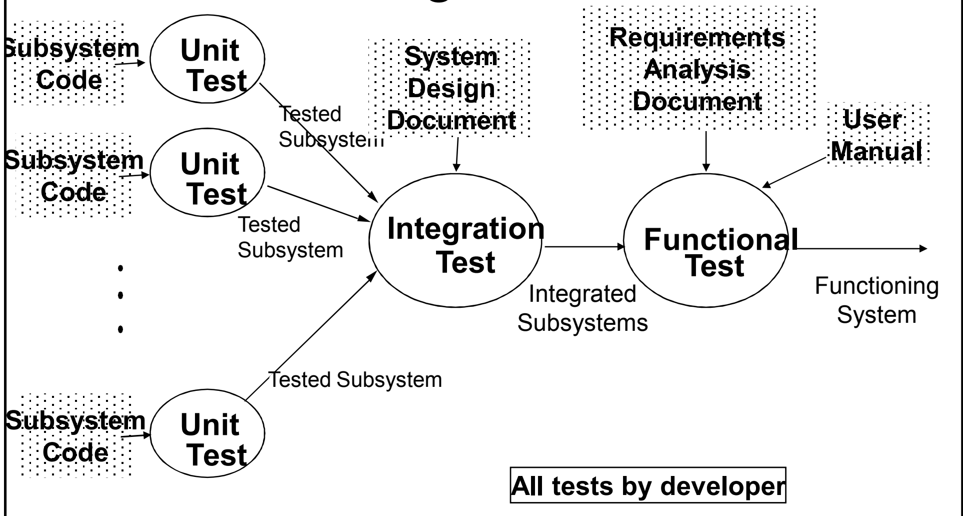
How do we deal with Errors and Faults?

- Verification:
  - Assumes hypothetical environment that does not match real environment
  - Proof might be buggy (omits important constraints; simply wrong)
- Modular redundancy:
  - Expensive
- Declaring a bug to be a “feature”
  - Bad practice
- Patching
  - Slows down performance
- Testing (this lecture)
  - Testing alone not enough, also need error prevention, detection, and recovery

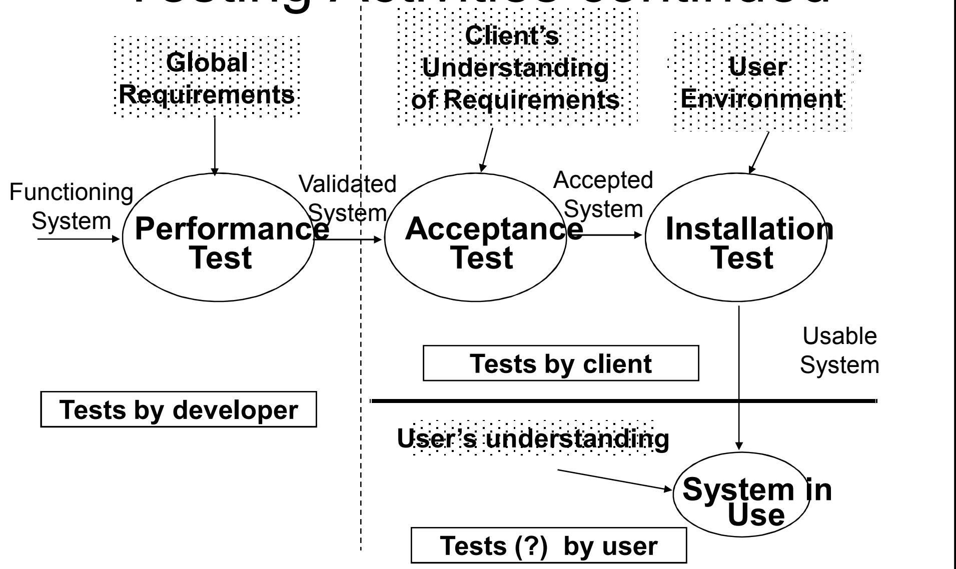
Testing takes creativity

- Testing often viewed as dirty work.
- To develop an effective test, one must have:
    - Detailed understanding of the system
    - Knowledge of the testing techniques
    - Skill to apply these techniques in an effective and efficient manner
- Testing is done best by independent testers
  - We often develop a certain mental attitude that the program should in a certain way when in fact it does not.
- Programmer often stick to the data set that makes the program work
- A program often does not work when tried by somebody else.
  - Don't let this be the end-user.

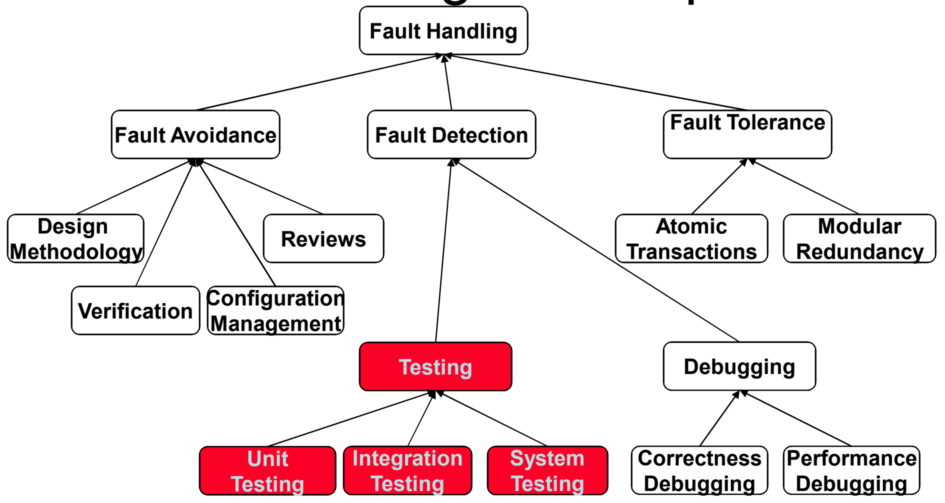
Testing Activities



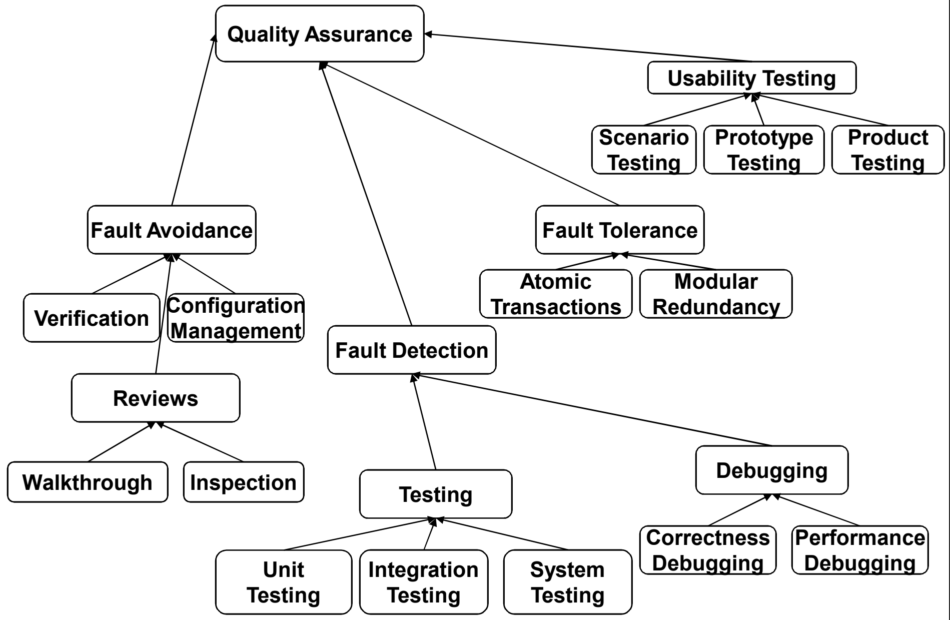
Testing Activities continued



Fault Handling Techniques



Quality Assurance Encompasses Testing



Types of Testing

- Unit Testing:
  - Individual subsystem
  - Carried out by developers
  - Goal: Confirm that subsystems is correctly coded and carries out the intended functionality
- Integration Testing:
  - Groups of subsystems (collection of classes) and eventually the entire system
  - Carried out by developers
  - Goal: Test the interface among the subsystem

System Testing

- System Testing:
  - The entire system
  - Carried out by developers
  - Goal: Determine if the system meets the requirements (functional and global)
- Acceptance Testing:
  - Evaluates the system delivered by developers
  - Carried out by the client. May involve executing typical transactions on site on a trial basis
  - Goal: Demonstrate that the system meets customer requirements and is ready to use
- Implementation (Coding) and Testing go hand in hand

Testing and the Lifecycle

- How can we do testing across the lifecycle?
  - Requirements
  - Design
  - Implementation
  - Maintenance

Requirements Testing

- Review or inspection to check whether all aspects of the system are described
- Look for
  - Completeness
  - Consistency
  - Feasibility
  - Testability
- Most likely errors
  - Missing information (functions, interfaces, performance, constraints, reliability, etc.)
  - Wrong information (not traceable, not testable, ambiguous, etc.)
  - Extra information (bells and whistles)

Design Testing

- Similar to testing requirements, also look for completeness, consistency, feasibility, testability
  - Precise documentation standard helpful in preventing these errors
- Assessment of architecture
- Assessment of design and complexity
- Test design itself
  - Simulation
  - Walkthrough
  - Design inspection

Implementation Testing

- “Real” testing
- One of the most effective techniques is to carefully read the code
- Inspections, Walkthroughs
- Static and Dynamic Analysis testing
  - Static: inspect program without executing it
    - Automated Tools checking for
      - syntactic and semantic errors
      - departure from coding standards
  - Dynamic: Execute program, track coverage, efficiency

Manual Test Techniques

- Static Techniques
  - Reading
  - Walkthroughs/Inspections
  - Correctness Proofs
  - Stepwise Abstraction

Reading

- You read, and reread, the code
- Even better: Someone else reads the code
  - Author knows code too well, easy to overlook things, suffering from implementation blindness
  - Difficult for author to take a destructive attitude toward own work
- Peer review
  - More institutionalized form of reading each other’s programs
  - Hard to avoid egoless programming; attempt to avoid personal, derogatory remarks

Walkthroughs

- Walkthrough
  - Semi to Informal technique
  - Author guides rest of the team through their code using test data; manual simulation of the program or portions of the program
  - Serves as a good place to start discussion as opposed to a rigorous discussion
  - Gets more eyes looking at critical code

Inspections

- Inspections
  - More formal review of code
  - Developed by Fagan at IBM, 1976
  - Members have well-defined roles
    - Moderator, Scribe, Inspectors, Code Author (largely silent)
    - Inspectors paraphrase code, find defects
    - Examples:
      - Vars not initialized, Array index out of bounds, dangling pointers, use of undeclared variables, computation faults or possibilities, infinite loops, off by one, etc.
  - Finds errors where they are in the code, have been lauded as a best practice

Correctness Proofs

- Most complete static analysis technique
- Try to prove a program meets its specifications
- {P} S {Q}
  - P = preconditions, S = program, Q = postconditions
  - If P holds before the execution of S, and S terminates, then Q holds after the execution of S
- Formal proofs often difficult for average programmer to construct

Stepwise Abstraction

- Opposite of top-down development
- Starting from code, build up to what the function is for the component
- Example:

```
Procedure Search(A: array[1..n] of integer, x:integer): integer;
Var low,high,mid: integer;  found:boolean;
Begin
    low:=1; high:=n; found:=false;
    while (low<=high) and not found do
        mid:=(low+high)/2
        if (x<A[mid]) then high:=mid-1;
        else if (x>A[mid]) then low:=mid+1;
        else found:=true;
        endif
    endwhile
    if found then return mid else return 0
End
```

Stepwise Abstraction

- If-statement on lines 7-10

```
7. if (x<A[mid]) then high:=mid-1;
8. else if (x>A[mid]) then low:=mid+1;
9. else found:=true;
10.     endif
```

- Summarize as:
  - Stop searching (found:=true) if x=A[mid] or shorten the interval [low..high] to a new interval [low'..high'] where high'-low' < high-low
  - (found = true and x=A[mid]) or
    (found = false and x∉A[1..low'-1] and
    x ∉ A[high'+1..n] and high'-low' < high-low)

Stepwise Abstraction

- Consider lines 4-5

```
4.  low:=1; high:=n; found:=false;
5.  while (low<=high) and not found do
```

- From this it follows that in the loop
  - low<=mid<=high
- The inner loop must eventually terminate since the interval [low..high] gets smaller until we find the target or low > high
- Complete routine:

  if Result > 0 then A[Result] = x
  else Result=0

Dynamic Testing

- Black Box Testing
- White Box Testing

Black-box Testing

- Focus: I/O behavior. If for any given input, we can predict the output, then the module passes the test.
  - Almost always impossible to generate all possible inputs ("test cases")
- Goal: Reduce number of test cases by equivalence partitioning:
  - Divide input conditions into equivalence classes
  - Choose test cases for each equivalence class. (Example: If an object is supposed to accept a negative number, testing one negative number is enough)

Black-box Testing (Continued)

- Selection of equivalence classes (No rules, only guidelines):
  - Input is valid across range of values. Select test cases from 3 equivalence classes:
    - Below the range
    - Within the range
    - Above the range
  - Input is valid if it is from a discrete set. Select test cases from 2 equivalence classes:
    - Valid discrete value
    - Invalid discrete value
- Another solution to select only a limited amount of test cases:
  - Get knowledge about the inner workings of the unit being tested => white-box testing

White-box Testing

- Focus: Thoroughness (Coverage). Every statement in the component is executed at least once.
- Four types of white-box testing
  - Statement Testing
  - Loop Testing
  - Path Testing
  - Branch Testing

White-box Testing (Continued)

- Statement Testing
  - Every statement is executed by some test case (C0 test)
- Loop Testing:
  - Cause execution of the loop to be skipped completely. (Exception: Repeat loops)
  - Loop to be executed exactly once
  - Loop to be executed more than once
- Path testing:
  - Make sure all paths in the program are executed
- Branch Testing (C1 test): Make sure that each possible outcome from a condition is tested at least once

```
if ( i =TRUE) printf("YES\n");elseprintf("NO\n");
Test cases: 1) i = TRUE; 2) i = FALSE
```

White-box Testing Example

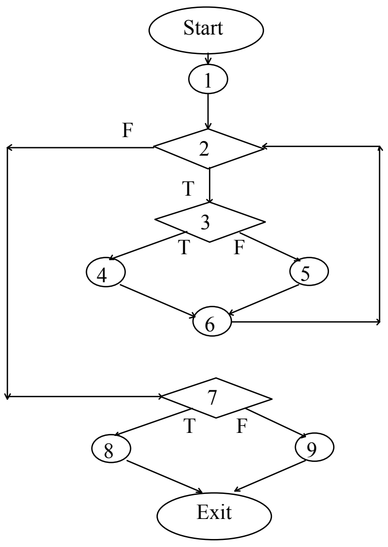
```
FindMean(float Mean, FILE ScoreFile)
{ SumOfScores = 0.0; NumberOfScores = 0; Mean = 0;
 Read(ScoreFile, Score);  /*Read in and sum the scores*/
 while (! EOF(ScoreFile) {
         if ( Score > 0.0 ) {
                SumOfScores = SumOfScores + Score;
                NumberOfScores++;
          }
          Read(ScoreFile, Score);
 }
 /* Compute the mean and print the result */
 if (NumberOfScores > 0 ) {
              Mean = SumOfScores/NumberOfScores;
              printf("The mean score is %f \n",  Mean);
 } else
              printf("No scores found in file\n");
}
```

White-box Testing Example: Determining the Paths

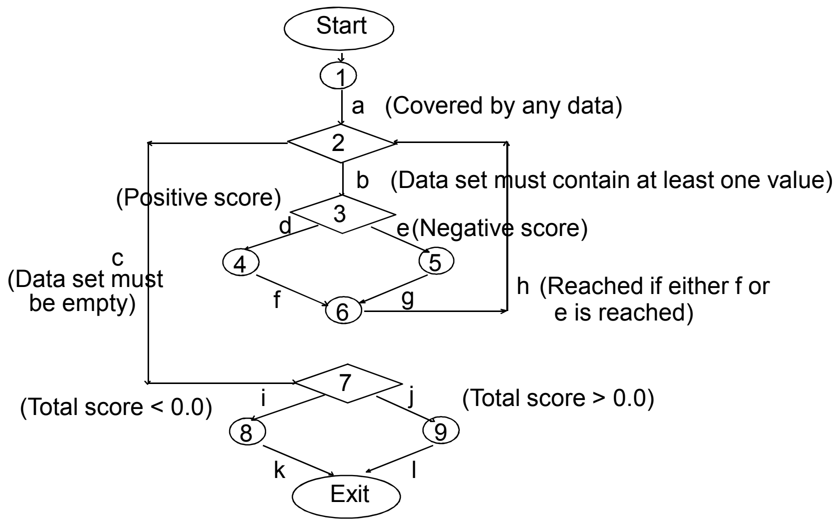
```
FindMean (FILE ScoreFile)
{   float SumOfScores = 0.0;
    int NumberOfScores = 0;
    float Mean=0.0; float Score;                       1
    Read(ScoreFile, Score);
2   while (! EOF(ScoreFile) {
    3 if (Score  > 0.0 ) {
            SumOfScores = SumOfScores + Score;          4
            NumberOfScores++;
          5 }

        Read(ScoreFile, Score);                         6
    }
    /* Compute the mean and print the result */
7   if (NumberOfScores > 0) {
          Mean = SumOfScores / NumberOfScores;
          printf(" The mean score is %f\n", Mean);      8
    } else
          printf ("No scores found in file\n");         9
}
```

Constructing the Logic Flow Diagram



Finding the Test Cases



Test Cases

- Test case 1 : ? (To execute loop exactly once)
- Test case 2 : ? (To skip loop body)
- Test case 3: ?,? (to execute loop more than once)

These 3 test cases cover all control flow paths

Comparison of White & Black-Box Testing

- White-box Testing:
  - Potentially infinite number of paths have to be tested
  - White-box testing often tests what is done, instead of what should be done
  - Cannot detect missing use cases
- Black-box Testing:
  - Potential combinatorical explosion of test cases (valid & invalid data)
  - Often not clear whether the selected test cases uncover a particular error
  - Does not discover extraneous use cases ("features")
- Both types of testing are needed
- White-box testing and black box testing are the extreme ends of a testing continuum.
- Any choice of test case lies in between and depends on the following:
  - Number of possible logical paths
  - Nature of input data
  - Amount of computation
  - Complexity of algorithms and data structures

Fault-Based Test Techniques

- Coverage-based techniques considered the structure of code and the assumption that a more comprehensive solution is better
- Fault-based testing does not directly consider the artifact being tested
  - Only considers the test set
  - Aimed at finding a test set with a high ability to detect faults

Fault-Seeding

- Estimating the number of salmon in a lake:
  - Catch N salmon from the lake
  - Mark them and throw them back in
  - Catch M salmon
  - If M’ of the M salmon are marked, the total number of salmon originally in the lake may be estimated at: (M-M’) * N/M’
- Can apply same idea to software
  - Assumes real and seeded faults have the same distribution

How to seed faults?

- Devised by testers or programmers
  - But may not be very realistic
- Have program independently tested by two groups
  - Faults found by the first group can be considered seeded faults for the second group
  - But good chance that both groups will detect the same faults
- Rule of thumb
  - If we find many seeded faults and relatively few others, the results can be trusted
  - Any other condition and the results generally cannot be trusted

Mutation Testing

- In mutation testing, a large number of variants of the program is generated
  - Variants generated by applying mutation operators
    - Replace constant by another constant
    - Replace variable by another variable
    - Replace arithmetic expression by another
    - Replace a logical operator by another
    - Delete a statement
    - Etc.
  - All of the mutants are executed using a test set
  - If a test set produces a different result for a mutant, the mutant is dead
  - Mutant adequacy score: D/M
    - D = dead mutants, M = total mutants
    - Would like this number to equal 1
- Points out inadequacies in the test set

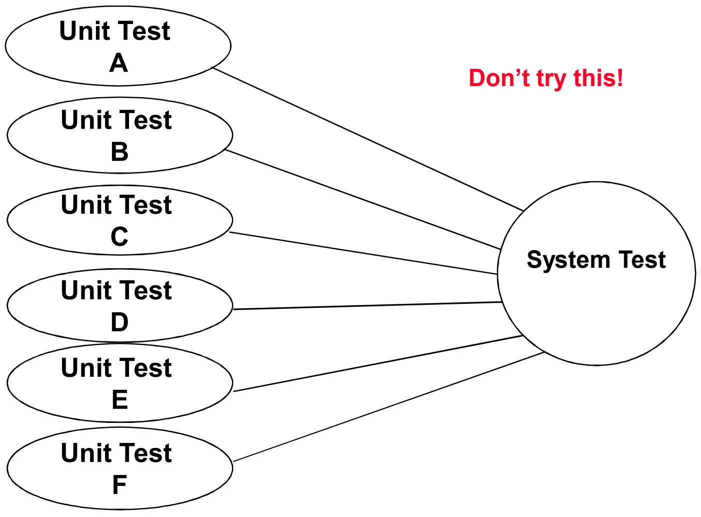
Error-Based Test Techniques

- Focuses on data values likely to cause errors
  - Boundary conditions, off by one errors, memory leak, etc.
- Example
  - Library system allows books to be removed from the list after six months, or if a book is more than four months old and borrowed less than five times, or ….
  - Devise test examples on the borders; at exactly six months, or borrowed five times and four months old, etc. As well as some examples beyond borders, e.g. 10 months
- Can derive tests from requirements (black box) or from code (white box) if code contains if (x>6) then .. Elseif (x >=4) && (y<5) …

Integration Testing Strategy

- The entire system is viewed as a collection of subsystems (sets of classes) determined during the system and object design.
- The order in which the subsystems are selected for testing and integration determines the testing strategy
  - Big bang integration (Nonincremental)
  - Bottom up integration
  - Top down integration
  - Sandwich testing
  - Variations of the above

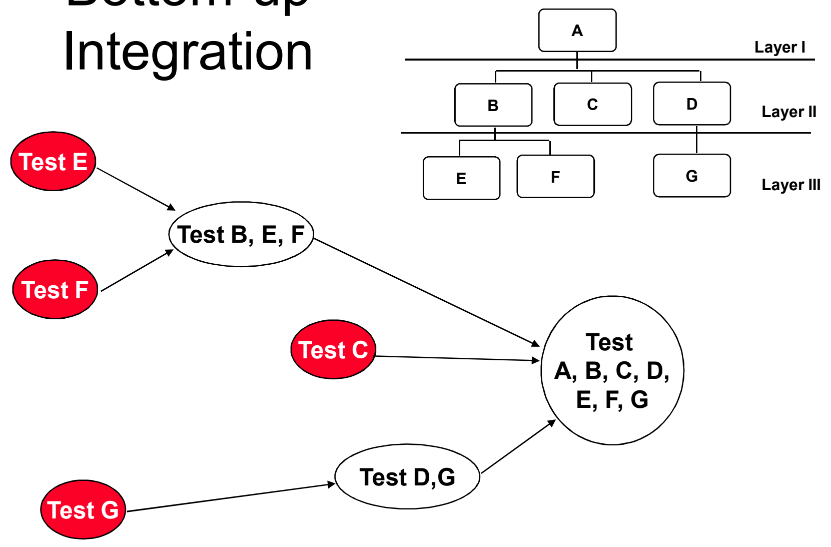
Integration Testing: Big-Bang Approach



Bottom-up Testing Strategy

- The subsystem in the lowest layer of the call hierarchy are tested individually
- Then the next subsystems are tested that call the previously tested subsystems
- This is done repeatedly until all subsystems are included in the testing
- Special program needed to do the testing, Test Driver:
  - A routine that calls a subsystem and passes a test case to it

Bottom-up Integration



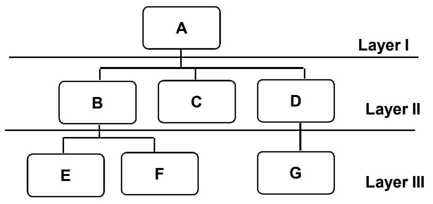
Pros and Cons of bottom up integration testing

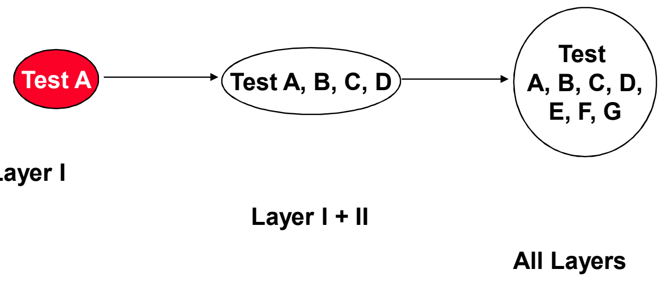
- Bad for functionally decomposed systems:
  - Tests the most important subsystem (UI) last
- Useful for integrating the following systems
  - Object-oriented systems
  - real-time systems
  - systems with strict performance requirements

Top-down Testing Strategy

- Test the top layer or the controlling subsystem first
- Then combine all the subsystems that are called by the tested subsystems and test the resulting collection of subsystems
- Do this until all subsystems are incorporated into the test
- Special program is needed to do the testing, *Test stub :*
  - A program or a method that simulates the activity of a missing subsystem by answering to the calling sequence of the calling subsystem and returning back fake data.

Top-down Integration Testing





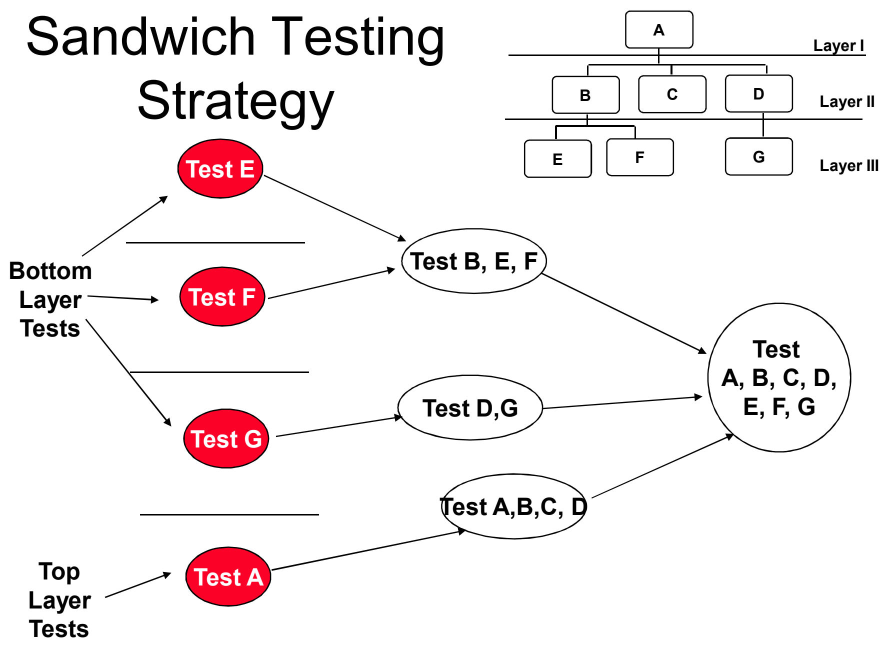
Pros and Cons of top-down integration testing

- Test cases can be defined in terms of the functionality of the system (functional requirements)
- Writing stubs can be difficult: Stubs must allow all possible conditions to be tested.
- Possibly a very large number of stubs may be required, especially if the lowest level of the system contains many methods.

Sandwich Testing Strategy

- Combines top-down strategy with bottom-up strategy
- *The system is view as having three layers*
  - A target layer in the middle
  - A layer above the target
  - A layer below the target
  - Testing converges at the target layer
- Need stubs/drivers if there are more than three layers; the stubs/drivers would approximate one “middle” layer

Sandwich Testing Strategy



Performance Testing

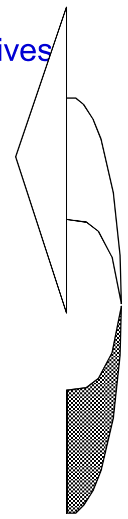
- Stress Testing
  - Stress limits of system (maximum # of users, peak demands, extended operation)
- Volume testing
  - Test what happens if large amounts of data are handled
- Configuration testing
  - Test the various software and hardware configurations
- Compatibility test
  - Test backward compatibility with existing systems
- Security testing
  - Try to violate security requirements
- Timing testing
  - Evaluate response times and time to perform a function
- Environmental test
  - Test tolerances for heat, humidity, motion, portability
- Quality testing
  - Test reliability, maintain- ability & availability of the system
- Recovery testing
  - Tests system's response to presence of errors or loss of data.
- Human factors testing
  - Tests user interface with user

Acceptance Testing

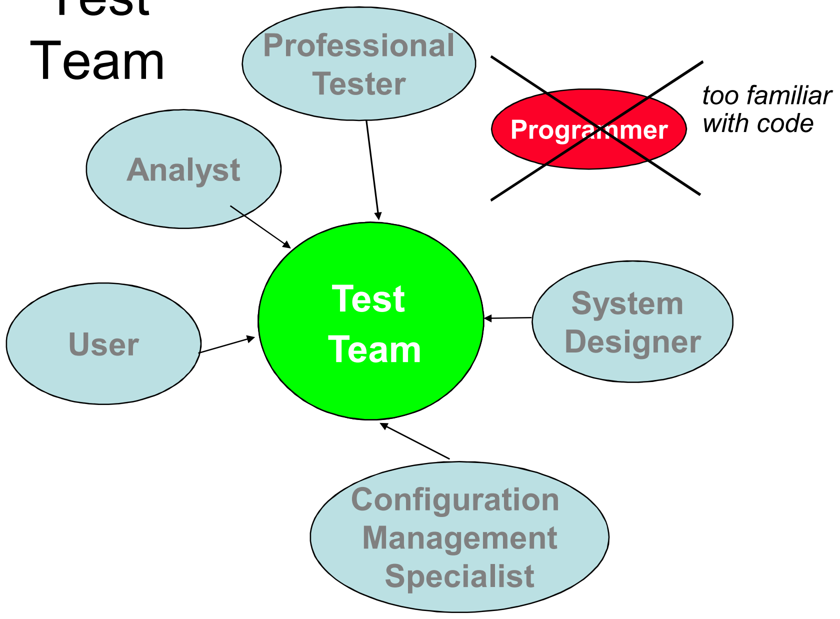
- Goal: Demonstrate system is ready for operational use
  - Choice of tests is made by client/sponsor
  - Many tests can be taken from integration testing
  - Acceptance test is performed by the client, not by the developer.
- Majority of all bugs in software is typically found by the client after the system is in use, not by the developers or testers. Therefore two kinds of additional tests:
- *Alpha test:*
  - Sponsor uses the software at the *developer's site.*
  - Software used in a controlled setting, with the developer always ready to fix bugs.
- *Beta test:*
  - Conducted at *sponsor's site* (developer is not present)
  - Software gets a realistic workout in target environment
  - Potential customer might get discouraged

Testing has its own Life Cycle

Establish the test objectives
Design the test cases
Write the test cases
Test the test cases
Execute the tests
Evaluate the test results
Change the system
Do regression testing



Test Team



Summary

- Testing is still a black art, but many rules and heuristics are available
- Test as early as possible
- Testing is a continuous process with its own lifecycle
- Design with testing in mind
- Test activities must be carefully planned, controlled, and documented
- We looked at:
  - Black and White Box testing
  - Coverage-based testing
  - Fault-based testing
  - Error-based testing
- Phases of testing (unit, integration, system)
- Wise to use multiple techniques

IEEE Standard 1012

- Template for Software Verification and Validation in a waterfall-like model

1. Purpose
2. References
3. Definitions
4. Verification & Validation Overview
   - 4.1 Organization
   - 4.2 Master Schedule
   - 4.3 Resources Summary
   - 4.4 Responsibilities
   - 4.5 Tools, techniques, methodologies

IEEE Standard 1012

5. Life-cycle Verification and Validation
   - 5.1 Management of V&V
   - 5.2 Requirements V&V
   - 5.3 Design V&V
   - 5.4 Implementation V&V
   - 5.5 Test V&V
   - 5.6 Installation & Checkout V&V
   - 5.7 Operation and Maintenance V&V
6. Software V&V reporting
7. V&V Administrative Procedures
   - 7.1 Anomaly reporting and resolution
   - 7.2 Task iteration policy
   - 7.3 Deviation policy
   - 7.4 Control procedures
   - 7.5 Standards, practices, conventions

Test Plan

- The bulk of a test plan can be structured as follows:

- Test Plan
  - Describes scope, approach, resources, scheduling of test activities. Refinement of V&V
- Test Design
  - Specifies for each software feature the details of the test approach and identify the associated tests for that feature
- Test Cases
  - Specifies inputs, expected outputs
  - Execution conditions
  - Test Procedures
    - Sequence of actions for execution of each test
  - Test Reporting
    - Results of tests

Sample Test Case 1

- Test Case 2.2 Usability 1 & 2
- Description: This test will test the speed of PathFinder.
- Design: This test will verify Performance Requirements 5.4 Usability-1 and Usability-2 in the Software Requirements Specification document.
- Inputs: The inputs will consist of a series of valid XML file containing Garmin ForeRunner data.
- Execution Conditions: All of the test cases in Batch 1 need to be complete before attempting this test case.
- Expected Outputs:
- The time to parse and time to retrieve images for various XML Garmin Route files will be tested.
- Procedure:
- 1. The PathFinder program will be modified to time its parsing and image retrieval times on at least 4 different sized inputs and on both high-speed and dial-up internet.
- 2. Results will be tabulated and options for optimization will be discussed if necessary.

Sample Test Case 1 (continued)

- Test Case 2.2 Usability 1 & 2
- **Completed 12/9/04.**
- **Results:**

File	File Size*	DataPoints	Dialup (56Kbps)	Broadband(128Kbps)
tinyrun.xml	2428	6	@12 seconds	<2 seconds
walk.xml	5840	16	@12 seconds	<2 seconds
exit.xml	366705	1152	@13 seconds	@2.5 seconds
run2.xml	654417	3000	@13 seconds	@2.5 seconds

- **According to this test data, the main delay in retrieving and displaying the data is entirely dependent upon the user's connection speed rather than on the parsing of the DataPoints (which seemed to introduce almost no delay, as evidenced by the minimal difference in times between the delay for tinyrun, which consists of 6 data points, and run2, which consists of 3000 data points). Optimization of the code was therefore deemed unnecessary.**

Sample Test Case 2

- Test Case 1.6 - GetImage
- Description: This test will test the ability of the GetImage module to retrieve an image from the TerraServer database given a set of latitude and longitude coordinate parameters.

- Design: This will continue verification of System Feature 3.1 (Open File) of the software requirements specification functions as expected. This test will verify the ability of the GetImage module to retrieve and put together a MapImage from a given set of latitude and longitude parameters.

- Inputs: The input for this test case will be a set of Data Points as created by the File modules in the above test case scenarios.

- Execution Conditions: All of the execution conditions of Test Case scenarios 1.0-1.5 must be met, and those test cases must be successful. Additionally, there must be a working copy of the GetImage class, and the TerraServer must be functioning properly, and this test case must run on a computer with a working internet connection.

Sample Test Case 2 (continued)

- Expected Outputs: The View will display the given MapImage retrieved from the TerraServer. This image will be compared to the image retrieved from the PhotoMap program to make sure that the latitude and longitude coordinates are correct.

- Procedure:

1. The User will open Pathfinder and will call the File class with the name of the XML file to be parsed by selecting “F)ile, O)pen” from the menu and finding the test file.
2. The File class will open the XML Parser.
3. The File class will call the XML Parser with the name of the XML file to be opened.
4. The XML Parser will open the file.
5. The XML Parser will create a new Data Point from the XML data returned and will insert each Data Point into a LinkedList.
6. The XML Parser will return the LinkedList to the File class when finished.
7. Using the Route’s Get method, the File class will update the LinkedList instance of a Route class.
8. The Route class, by way of its Notify method, will notify the GetImage class that its data has changed.
9. The GetImage class will retrieve the appropriate Image(s) from the TerraServer database.
10. The GetImage class will modify a MapImage’s image to be that of the Images satisfying the given parameters, using the MapImage’s Set methods.
11. The MapImage will notify its observers (View).
12. View will redraw its bottom Image to be that of the Map.
13. The User will close the program.