Fundamental of Software Engineering
CSE 3205

Chapter Five
Coding and Testing

School of Electrical Engineering and Computing
Department of Computer Science and Engineering
ASTU

February 9, 2024

Introduction

In modern software engineering work, coding may be

- the direct creation of programming language source code (e.g., Java),
- the automatic generation of source code using an intermediate design-like representation of the component to be built, or
- the automatic generation of executable code using a "fourth-generation programming language" (e.g., Visual C++).

Cont..

- The objective of **Coding** is to transform the design of a system into code in a high level language and then to unit test this code .

Good Characteristics of a Programming Language for Implementation

- **Readability:** A good high-level language will allow programs to be written in some ways that resemble a quite-English description of the underlying algorithms. .
- **Portability:** High-level languages, being essentially machine independent, should be able to develop portable software.
- **Generality:** Most high-level languages allow the writing of a wide variety of programs, thus relieving the programmer of the need to become expert in many diverse languages

Cont..

- **Familiar notation:** A language should have familiar notation, so it can be understood by most of the programmers.
- **Quick translation:** It should admit quick translation.
- **Efficiency:** It should permit the generation of efficient object code.
- **Modularity:** It is desirable that programs can be developed in the language as a collection of separately compiled modules, with appropriate mechanisms for ensuring self-consistency between these modules.
- **Widely available:** Language should be widely available and it should be possible to provide translators for all the major machines and for all the major operating systems.

Coding Standards

- Rules for limiting the use of global variable
- Contents of the headers preceding codes for different modules
- Naming conventions for global variables, local variables, and constant identifiers
- Error return conventions and exception handling mechanisms

Coding Guidelines

- **Code should be easy to understand.**
- **Do not use an identifier for multiple purposes**
- **The code should be well-documented**
- **The length of any function should not exceed 10 source lines**

Coding Principles

- The principles that guide the coding task are closely aligned with programming style, programming languages, and programming methods.
- However, there are a number of fundamental principles that can be stated

Preparation principles:

Before you write one line of code, be sure you

- Understand of the problem before you trying to solve.
- Understand basic design principles and concepts.
- Pick a programming language that meets the needs of the software to be built and the environment in which it will operate.
- Select a programming environment that provides tools that will make your work easier.
- Create a set of unit tests that will be applied once the component you code is completed.

Programming principles: As you begin writing code, be sure you

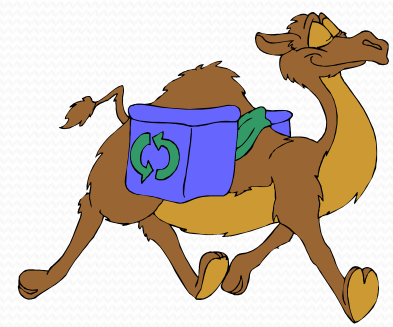
- Consider the use of pair programming.
- Select data structures that will meet the needs of the design.
- Understand the software architecture and create interfaces that are consistent with it.
- Keep conditional logic as simple as possible.

Goal: Self-Documenting Code

- **Self-documenting** explains *itself* without need for external documentation, like flowcharts, UML diagrams, process-flow diagrams, etc.
  - *Doesn't imply we don't like/use those documents!*
- **Coding conventions** target:
  - How you write **statements** in the language, **organize** them into "modules," **format** them in the source files
    - *Module:* generic term meaning C function, Java/C++ class, etc.
  - How you create **names**
  - How you write **comments**

Standard Coding convention

- Teams strive to use the same **coding conventions** in every regard:
  - **Name** your classes similarly, your variables, your functions.
  - **Comment** the same way, **format** your code the same way.
    - By doing this, you ensure rapid understanding of whatever module needs changing, and as they evolve, your modules will not degenerate into a *HorseByCommittee* appearance.



Benefits

- Projects **benefit** from having strong Coding Conventions/Standards because…
  - People can stop **reformatting** code and **renaming** variables and methods whenever working on code written by **other** people.
  - It's slightly **easier to understand** code that is consistently formatted and uses a consistent naming standard.
  - It's **easier to integrate** modules that use a common consistent naming standard -- less need to look up and cross-reference the different names that refer to the same thing.

Coding Conventions Apply To…

- Comments, 3 types:
  - File headers
  - Function headers
  - Explanations of variables and statements
- Names (chosen by programmer)
- Statements
  - Organization: files, “modules,” nesting
  - Format: spacing and alignment

Organization of Program

- *Analogy:* Organize programs for readability, as you would organize a book:
  - Title page & Table of contents → File header
  - Chapter → Module (function or logical group of functions)
  - Paragraph → Block of code
  - Index & Glossary → can be generated automatically if comments are used wisely (Javadoc, doxygen)
  - Cross references → ctags.sourceforge.net free tool

Organization of Modules

- Apply comp. sci. principle of information hiding
  - Hide details of of implementation that users don't need to know
- Divide each module into a **public** part and a **private** part.
  - public part goes into an *include* (.h) file
  - private part goes into a *source* (.c) file

How Many Source Files?

- Matter of policy and/or taste
  - One extreme: just one module per file
    - OO programming: 1 class per file is common
    - Large project → explosion of files! .h .c .o
  - Other extreme: all modules in one file
    - Reasonable for quite small project
    - Large project → lose benefit of separate compilation
- Middle way: group *related* modules in file
  - marcutil.h/c: all MARC utility functions

  >2-3000 lines is getting too large

File Headers

- *Creation date*
  - Provides a creation timestamp for copyright purposes, but it does more than that. It provides a quick clue to the *context* that existed at the time the module was created. Not as accurate as source control, but *quick* and *maintenance free*.
- *Author's Name or Initials*
- *Copyright banner*
  - This identifies the uses to which this code can be put.

Variable Names

- Use simple, descriptive variable names.
- Good names can be created by using one word or putting multiple words together joined by underscores or caps
  - prefer usual English word order

```
#define MAX_FIELD 127
int numStudents, studentID;
char *homeAddr;
```

Use Vertical Alignment (Type A)

- Makes lines at same level of nesting stand out.

```
if ( flag == 0 ) {
    var1 = 0;
    if ( var2 > level1 ) {
        var2 = level1;
        level1 = 0;
    }
    printf ( "%d/n", var2 );
}
```

Good General Coding Principle

- **KISS**
  - Keep it simple and small - always easier to read and maintain (and debug!)
- **Be Explicit**
  - SWYM - Say What You Mean

if (WordCount) vs. if (WordCount != 0)

n+3*x-5/y vs. n + ((3*x)-5)/y

Tips of Fixing Errors

- Understand the problem before you fix it
- Understand the program, not just the problem
- Confirm the error diagnosis
- Relax
- Save the original source code
- Fix the problem, not the symptom
- Make one change at a time
- Check your fix
- Look for similar errors

Testing

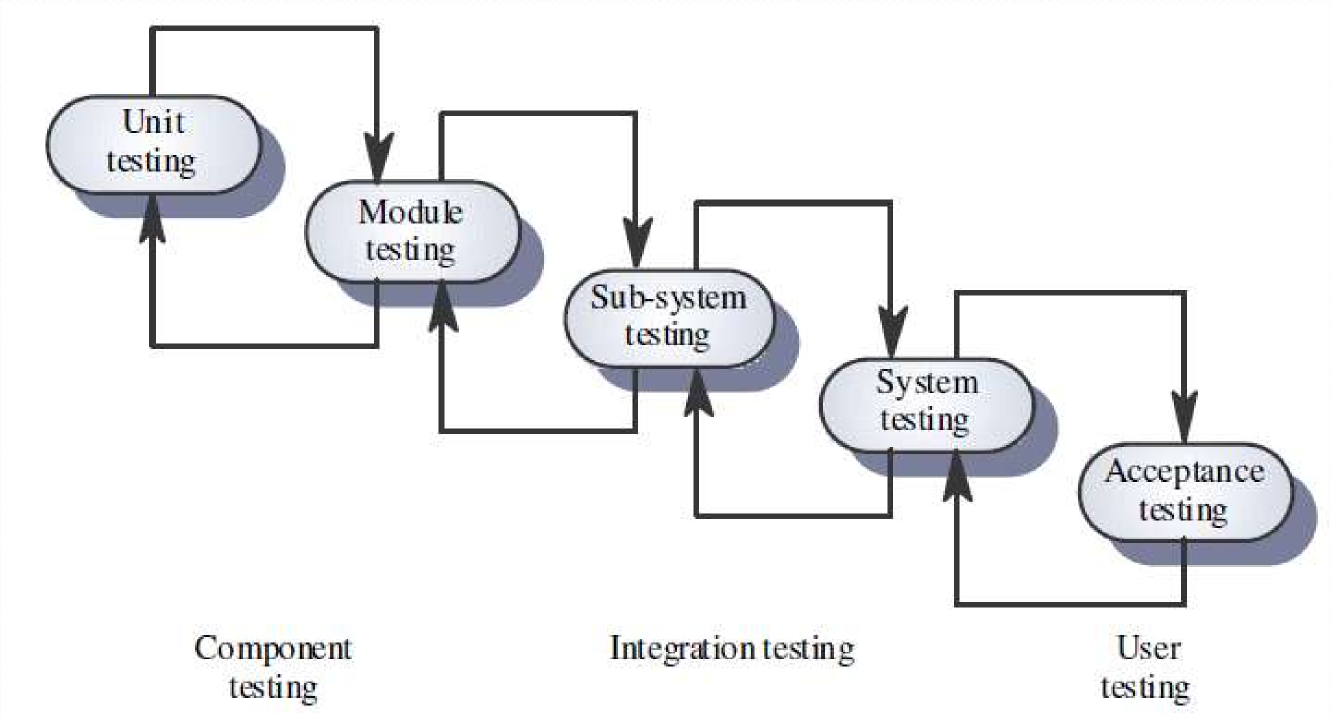
Observations about Testing

- “Testing is the process of executing a program with the intention of finding errors.” – Myers
- “Testing can show the presence of bugs but never their absence.” - Dijkstra

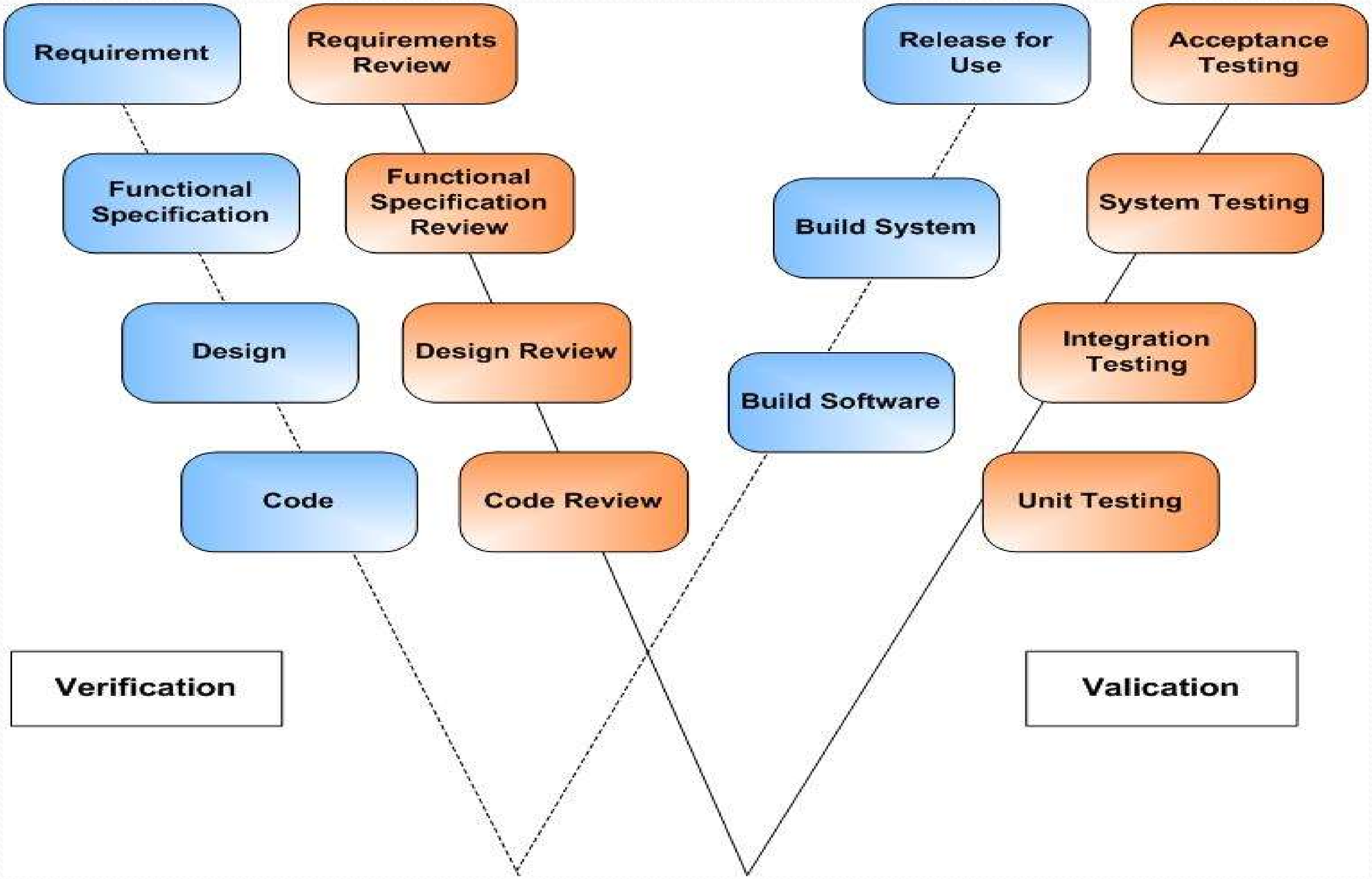
What is our goal during testing?

- Objective 1: find as many faults as possible
- Objective 2: make you feel confident that the software works OK

The Testing Process



W-Model for Testing



Testing and the life cycle

- requirements engineering
  - criteria: completeness, consistency, feasibility, and testability.
  - typical errors: missing, wrong, and extra information
  - determine testing strategy
  - generate functional test cases
  - test specification, through reviews and the like
- design
  - functional and structural tests can be devised on the basis of the decomposition
  - the design itself can be tested (against the requirements)
  - formal verification techniques
  - the architecture can be evaluated

Testing and the life cycle (cnt'd)

- Implementation
  - check consistency implementation and previous documents
  - code-inspection and code-walkthrough
  - all kinds of functional and structural test techniques
  - extensive tool support
  - formal verification techniques
- maintenance
  - regression testing: either retest all, or a more selective retest

Levels of Testing

- Unit /component Testing
- Integration Testing
- Validation Testing
  - Regression Testing
  - Alpha Testing
  - Beta Testing
- Acceptance Testing

Unit/Component Testing

- Algorithms and logic
- Data structures (global and local)
- Interfaces
- Independent paths
- Boundary conditions
- Error handling
  - Usually the responsibility of the component developer (except sometimes for critical systems);
  - Tests are derived from the developer's experience.
- White-Box testing

Integration Testing

Why Integration Testing Is Necessary

- One module can have an adverse effect on another sub-functions , when combined, may not produce the desired major function
- Individually acceptable imprecision in calculations may be magnified to unacceptable levels

Why Integration Testing Is Necessary (cont'd)

- Interfacing errors not detected in unit testing may appear
- Timing problems (in real-time systems) are not detectable by unit testing
- Resource contention problems are not detectable by unit testing

Driver and Stub

- **A test driver is a routine that calls a particular component and passes** test cases to it. (Also it should report the results of the test cases).
- **A test stub is a special-purpose program used to answer the calling** sequence and passes back output data that lets the testing process continue.

```
public class RandIntTest {
 public static void main(String[] args) {
   RandInt myRand = new RandInt();
   System.out.println("My first rand int
is :" + myRand.generateRandInt());
 }
}
```

Driver

```
Public class RandInt{
...
public int generateRandInt() {
 return (int)(Math.random()*100 + 1);
}
...
}
```

Module

```
Public class RandInt{
...
public int generateRandInt() {
       return 1;
}
...
}
```

Stub

Top-Down Integration

1. Develop the skeleton of the system and populate it with components
2. The main control module is used as a driver, and stubs are substituted for all modules directly subordinate to the main module.
3. Depending on the integration approach selected (depth or breadth first), subordinate stubs are replaced by modules one at a time.

Top-Down Integration (cont'd)

3. Tests are run as each individual module is integrated.
4. On the successful completion of a set of tests, another stub is replaced with a real module
5. Regression testing is performed to ensure that errors have not developed as result of integrating new modules

Problems with Top-Down Integration

- Many times, calculations are performed in the modules at the bottom of the hierarchy Stubs typically do not pass data up to the higher modules
- Delaying testing until lower-level modules are ready usually results in integrating many modules at the same time rather than one at a time
- Developing stubs that can pass data up is almost as much work as developing the actual module

Bottom-Up Integration

- Integration begins with the lowest-level modules, which are combined into clusters, or builds, that perform a specific software subfunction
- Integrate infrastructure components then add functional components
- Drivers (control programs developed as stubs) are written to coordinate test case input and output
- The cluster is tested
- Drivers are removed and clusters are combined moving upward in the program structure

Problems with Bottom-Up Integration

- The whole program does not exist until the last module is integrated
- Timing and resource contention problems are not found until late in the process

Validation and Regression Testing

- Determine if the software meets all of the requirements defined in the SRS
- Having written requirements is essential
- Regression testing is performed to determine if the software still meets all of its requirements in light of changes and modifications to the software
- Regression testing involves selectively repeating existing validation tests, not developing new tests

Alpha and Beta Testing

- It's best to provide customers with an outline of the things that you would like them to focus on and specific test scenarios for them to execute.
- Provide with customers who are actively involved with a commitment to fix defects that they discover.

Acceptance Testing

- Similar to validation testing except that customers are present or directly involved.
- Usually the tests are developed by the customer

System Testing

- Recovery testing
  - checks system's ability to recover from failures
- Security testing
  - verifies that system protection mechanism prevents improper penetration or data alteration
- Stress testing
  - program is checked to see how well it deals with abnormal resource demands
- Performance testing
  - tests the run-time performance of software
- **Volume Testing**
  - Heavy volumes of data
  - If a program is supposed to handle files spanning multiple volumes, enough data is created to cause the program to switch from one volume to another

Characteristics of Testable Software

- Operable
  - The better it works (i.e., better quality), the easier it is to test
- Observable
  - Incorrect output is easily identified; internal errors are automatically detected
- Controllable
  - The states and variables of the software can be controlled directly by the tester
- Decomposable
  - The software is built from independent modules that can be tested independently

Good Testing Practices

- A good test case is one that has a high probability of detecting an undiscovered defect, not one that shows that the program works correctly
- It is impossible to test your own program
- A necessary part of every test case is a description of the expected result

Good Testing Practices (cont'd)

- Avoid nonreproducible or on-the-fly testing
- Write test cases for valid as well as invalid input conditions.
- Thoroughly inspect the results of each test
- As the number of detected defects in a piece of software increases, the probability of the existence of more undetected defects also increases

Good Testing Practices (cont'd)

- Assign your best people to testing
- Ensure that testability is a key objective in your software design
- Never alter the program to make testing easier
- Testing, like almost every other activity, must start with objectives

Test Characteristics

- A good test has a high probability of finding an error
  - The tester must understand the software and how it might fail
- A good test is not redundant
  - Testing time is limited; one test should not serve the same purpose as another test
- A good test should be “best of breed”
  - Tests that have the highest likelihood of uncovering a whole class of errors should be used
- A good test should be neither too simple nor too complex
  - Each test should be executed separately; combining a series of tests could cause side effects and mask certain errors

Two Unit Testing Techniques

- Black-box testing
  - Knowing the specified function that a product has been designed to perform, test to see if that function is fully operational and error free
  - Includes tests that are conducted at the software interface
  - Not concerned with internal logical structure of the software
- White-box testing
  - Knowing the internal workings of a product, test that all internal operations are performed according to specifications and all internal components have been exercised
  - Involves tests that concentrate on close examination of procedural detail
  - Logical paths through the software are tested
  - Test cases exercise specific sets of conditions and loops

Classification of testing techniques

- Classification based on the criterion to measure the adequacy of a set of test cases:
  - coverage-based testing
  - fault-based testing
  - error-based testing
- Classification based on the source of information to derive test cases:
  - black-box testing (functional, specification-based)
  - white-box testing (structural, program-based)

Cont.

- Coverage-based: e.g. how many statements or requirements have been tested so far
- Fault-based: e.g., how many seeded faults are found
- Error-based: focus on error-prone points, e.g. off-by-one points

- Black-box: you do not look inside, but only base yourself on the specification/functional description
- White-box: you do look inside, to the structure, the actual program/specification.

Some preliminary questions

- What exactly is an error?
- How does the testing process look like?
- When is test technique A superior to test technique B?
- What do we want to achieve during testing?
- When to stop testing?

Error, fault, failure

- an *error* is a human activity resulting in software containing a fault
- a *fault* is the manifestation of an error
- a *fault* may result in a failure

When exactly is a failure?

- Failure is a relative notion: e.g. a failure w.r.t. the specification document
- *Verification*: evaluate a product to see whether it satisfies the conditions specified at the start:

  Have we built the system right?
- *Validation*: evaluate a product to see whether it does what we think it should do:

  Have we built the right system?

Test adequacy criteria

- Specifies requirements for testing
- Can be used as *stopping rule*: stop testing if 100% of the statements have been tested
- Can be used as *measurement*: a test set that covers 80% of the test cases is better than one which covers 70%
- Can be used as *test case generator*: look for a test which exercises some statements not covered by the tests so far
- A given test adequacy criterion and the associated test technique are opposite sides of the same coin

Test Planning

- The Test Plan – defines the scope of the work to be performed
- The Test Procedure – a container document that holds all of the individual tests (test scripts) that are to be executed
- The Test Report – documents what occurred when the test scripts were run

Test Plan

- Questions to be answered:
  - How many tests are needed?
  - How long will it take to develop those tests?
  - How long will it take to execute those tests?
- Topics to be addressed:
  - Test estimation
  - Test development and informal validation
  - Validation readiness review and formal validation
  - Test completion criteria

Test Estimation

- Number of test cases required is based on:
  - Testing all functions and features in the SRS
  - Including an appropriate number of ALAC (Act Like A Customer) tests including:
    - Do it wrong
    - Use wrong or illegal combination of inputs
    - Don’t do enough
    - Do nothing
    - Do too much
  - Achieving some test coverage goal
  - Achieving a software reliability goal

Considerations in Test Estimation

- Test Complexity – It is better to have many small tests that a few large ones.
- Different Platforms – Does testing need to be modified for different platforms, operating systems, etc.
- Automated or Manual Tests – Will automated tests be developed? Automated tests take more time to create but do not require human intervention to run.

Test Team Members

- Professional testers.
- Analysts.
- System designers.
- Configuration management specialists.
- Users.

Debugging

- Debugging (removal of a defect) occurs as a consequence of successful testing.
- Some people better at debugging than others.
- Is the cause of the bug reproduced in another part of the program?
- What "next bug" might be introduced by the fix that is being proposed?
- What could have been done to prevent this bug in the first place?

Test Report

- Completed copy of each test script with evidence that it was executed (i.e., dated with the signature of the person who ran the test)
- Software Problem Reports (SPRs) are submitted for each test that fails.
- Copy of each SPR(Software Problem Reports) showing resolution
- List of open or unresolved SPRs
- Identification of SPRs found in each baseline along with total number of SPRs in each baseline
- Regression tests executed for each software baseline

Thank You!

Q?