Software Testing Fundamentals

CSCE 740 - Lecture 20- 11/09/2015

We Will Cover

- Revisiting Verification & Validation
- Testing definitions
  - Let’s get the language right.
- What is a test?
- Testing stages.
  - Unit, Subsystem, System, and Acceptance Testing

V&V: Definition

- Verification
  - "Are we building the product right?"
  - The software should conform to its specifications.
- Validation
  - "Are we building the right product?"
  - The software should so what the user really requires.

Both are required to determine whether your software is ready for release.

Verification: Alternate Definitions

- Verification
    - "The process of evaluating a system or component to determine whether the products... **satisfy the conditions imposed**."
    - "**Checking** a program against the most closely related **design or specification documents**."
    - "An attempt to find errors by **executing a program in a test or simulated environment**."
    - "The software should **conform to its specifications**."

Under the conditions we set, does the software work?

Validation: Alternate Definitions

- Validation
  - "The process of evaluating a system or component to determine whether the products… **satisfy user requirements**."
  - "**Checking** a program against the published **user or system requirements**."
  - "An attempt to find errors by executing a program in a **real environment**."
  - "The software should so what the **user really requires**."

Does the software work in the real world?

Verification and Validation

- Verification
  - Does the software work under the conditions we impose?
  - **Quantitative** - we can perform experiments, evaluate the software and provide evidence.
- Validation
  - Does the software work in the real world?
    - Does it meet the needs of your users?
  - **Qualitative** - we cannot control real-world use. Often boils down to the feelings of your users.

Verification and Validation: Motivation

Which is more important?

- Both are important.
  - A well-verified system might not meet the user's needs.
  - A system can't meet the user's needs unless it is well-constructed.

When do you perform verification and validation?

- Constantly, throughout development.
  - Verification requires specifications, but can begin then and be executed throughout development.
  - Validation can start at any time by seeking feedback.

Types of Verification

Static Verification

- Analysis of static system artifacts to discover problems.
  - Proofs: Posing hypotheses and making a logical argument for their validity using specifications, system models, etc.
  - Inspections: Manual “sanity check” on artifacts (such as source code) by other people or tools, searching for issues.

Advantages of Static Verification

- During execution, errors can hide other errors. It can be hard to find all problems or trace back to a single source. Static inspections are not impacted by program interactions.
- Incomplete systems can be inspected without additional costs. If a program is incomplete, special code is needed to run the part that is to be tested.
- Inspection can also assess quality attributes such as maintainability, portability, poor programming, inefficiencies, etc.

Dynamic Verification

- Exercising and observing the system to argue that it meets the requirements.
  - Testing: Formulating controlled sets of input to demonstrate requirement satisfaction.
- Static verification is not good at discovering problems that arise from runtime interaction, timing problems, or performance issues.
- Dynamic verification is often cheaper than static - easier to automate.

Software Testing

- Investigation conducted to provide information about system quality.
- The main purpose of testing is to find errors.
  - “Testing is the process of trying to discover every conceivable fault or weakness in a work product” - Glenford Myers

Axiom of Testing

"Program testing can be used to show the presence of bugs, but never their absence."

- Dijkstra

What Does Testing Accomplish?

Your current goal shapes what scenarios the tests cover:

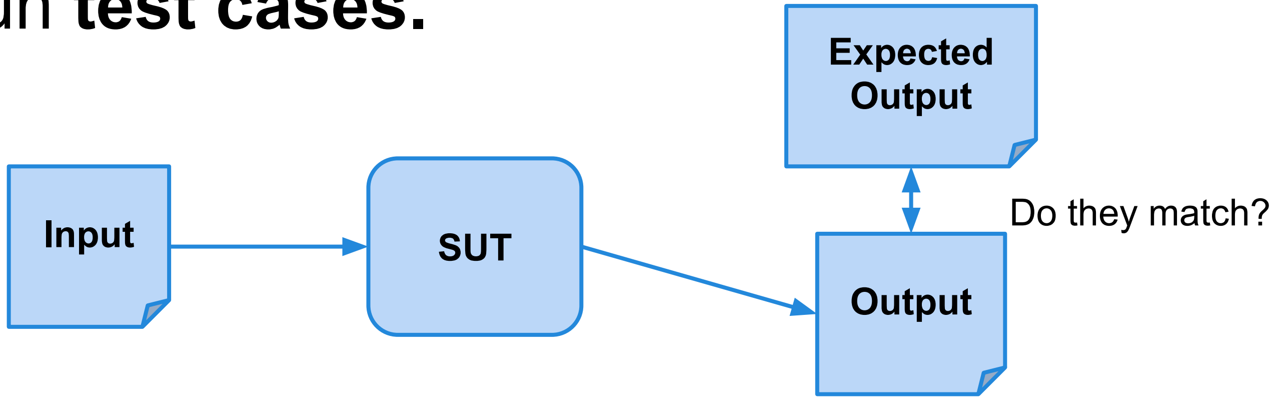
- **Verification:** Demonstrate to the customer that the software meets the specifications.
  - Tests tend to reflect “normal” usage.
- **Fault Detection:** Discover situations where the behavior of the software is incorrect.
  - Tests tend to reflect extreme usage.

Bugs? What are Those?

- Bug is an overloaded term - does it refer to the bad behavior observed, the source code problem that led to that behavior, or both?
- **Failure**
  - An execution that yields an incorrect result.
- **Fault**
  - The problem that is the source of that failure.
  - For instance, a typo in a line of the source code.
- When we observe a failure, we try to find the fault that caused it.

What is a Test?

During testing, we instrument the **system under test** and run **test cases.**



A test case is made up primarily of:

- **Test Input** - Stimuli fed to the system.
- **Test Oracle** - The expected output, and a way to check whether the actual output matches the expected output.

Anatomy of a Test Case

- Input
  - Any required input data.
- Expected Output (Oracle)
  - What *should* happen, i.e., values or exceptions.
- Initialization
  - Any steps that must be taken before test execution.
- Test Steps
  - Interactions with the system, and comparisons between expected and actual values.
- Tear Down
  - Any steps that must be taken after test execution.

Black and White Box Testing

- Black Box Testing
  - Designed without knowledge of the program's internal structure and design.
  - Based on functional requirements.
- White Box Testing
  - Examines the internal design of the program.
  - Requires detailed knowledge of its structure.
  - Tests typically based on coverage of the source code (all statements/conditions/branches have been executed)

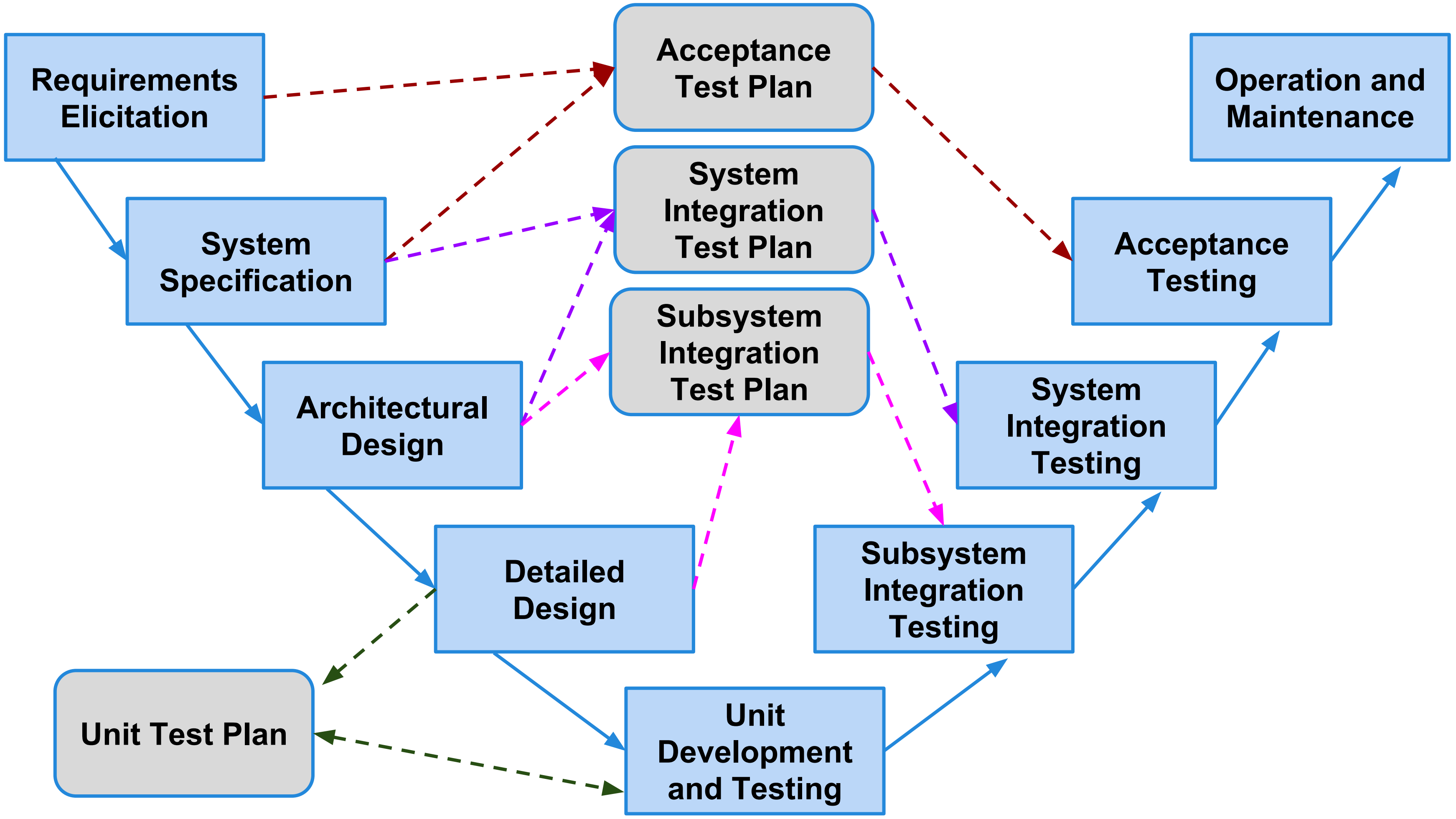
Testing Stages

- Unit Testing
  - Testing of individual methods of a class.
  - Requires design to be final, so usually written and executed simultaneously with coding of the units.
- Module Testing
  - Testing of collections of dependent units.
  - Takes place at same time as unit testing, as soon as all dependent units complete.
- Subsystem Integration Testing
  - Testing modules integrated into subsystems.
  - Tests can be written once design is finalized, using SRS document.

Testing Stages

- System Integration Testing
  - Integrate subsystems into a complete system, then test the entire product.
  - Tests can be written as soon as specification is finalized, executed after subsystem testing.
- Acceptance Testing
  - Give product to a set of users to check whether it meets their needs. Can also expose more faults.
  - Also called alpha/beta testing.
  - Acceptance planning can take place during requirements elicitation.

The V-Model of Development

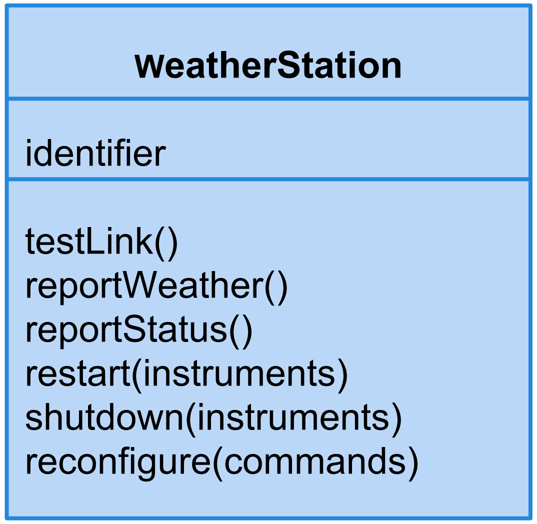

Requirements Elicitation
System Specification
Architectural Design
Detailed Design
Unit Test Plan
Unit Development and Testing
Acceptance Test Plan
System Integration Test Plan
Subsystem Integration Test Plan
Subsystem Integration Testing
System Integration Testing
Acceptance Testing
Operation and Maintenance

Unit Testing

Unit testing is the process of testing individual method of a class.

- Test input should be calls to methods with different input parameters.
- For a class, tests should:
  - Test all operations associated with the class.
  - Set and check the value of all attributes associated with the class.
  - Put the class into all possible states.

Unit Testing - WeatherStation



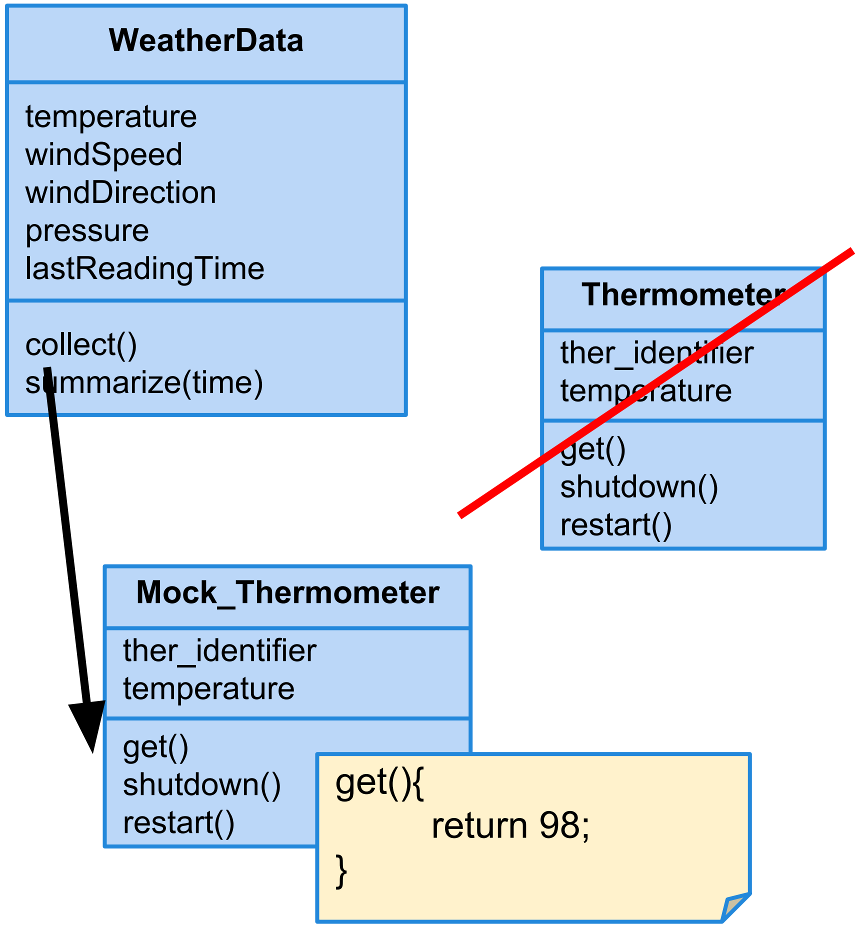
When writing unit tests for WeatherStation, we need:

- Set and check identifier.
- Tests for each method.
- Tests that hit each outcome of each method (error handling, return conditions).

Unit Testing - Object Mocking

Components may depend on other, unfinished (or untested) components. You can **mock** those components.

- Mock objects have the same interface as the real component, but are hand-created to simulate the real component.
- Can also be used to simulate abnormal operation or rare events.



Writing a Unit Test

JUnit is a Java-based toolkit for writing

- You wr
  containing a series of unit
  tests ce
  either c
  functionality of a class
  being te
- The cla
  initializa
  tear down method, and
  tests.

Setup steps common to all tests. Denoted by the keyword **@before**

Tear down steps common to all tests. Denoted by the keyword **@after**

Each test is denoted with keyword **@test**.

Convention - name the test class after the class it is testing or the functionality being tested.

```
public class GetStudentsTest {

    @Befo
    publi
    Exception { .. }

    @After
    public void tearDown()
    throws Exception { .. }

    @Test
    public void
    testGetStudentList_Valid()
    { .. }
    ..
}
```

Setup Method

@Before annotation defines a common test initialization:

```
@Before
public void setUp() throws Exception
{
   this.grads = new GRADS();
   this.grads.setUser(this.csUser);
}
```

Teardown Method

@After annotation defines a common test tear down:

```
@After
public void tearDown() throws Exception
{
   this.grads.logout();
   this.grads = null;
}
```

Test Skeleton

@Test annotation defines a single test:

```
@Test
public void test<MethodName>_valid() {
   //Define Inputs
   try{ //Try to get output.
   }catch(Exception error){
      fail("Why did it fail?");
   }
   //Compare expected and actual values through
assertions or through if statements/fails
}
```

Assertions

Assertions are a "language" of testing - constraints that you place on the output.

Some JUnit assertions:

- assertEquals, assertArrayEquals
- assertFalse, assertTrue
- assertNull, assertNotNull
- assertSame,assertNotSame

Assertion Example

```
@Test
public void testGetStudentList_valid() {
    actualOut = grads.getStudentList();

    //The list we got back should not be null.
    assertNotNull(actualOut);

    //Check that the expected number of students are present.
    assertEquals(this.expectedOut.size(),actualOut.size());

    //Now, check each value.
    for each output from expectedOut and actualOut{
        assertNotNull(actualValue);
        assertEquals(expectedValue,actualValue);
    }
}
```

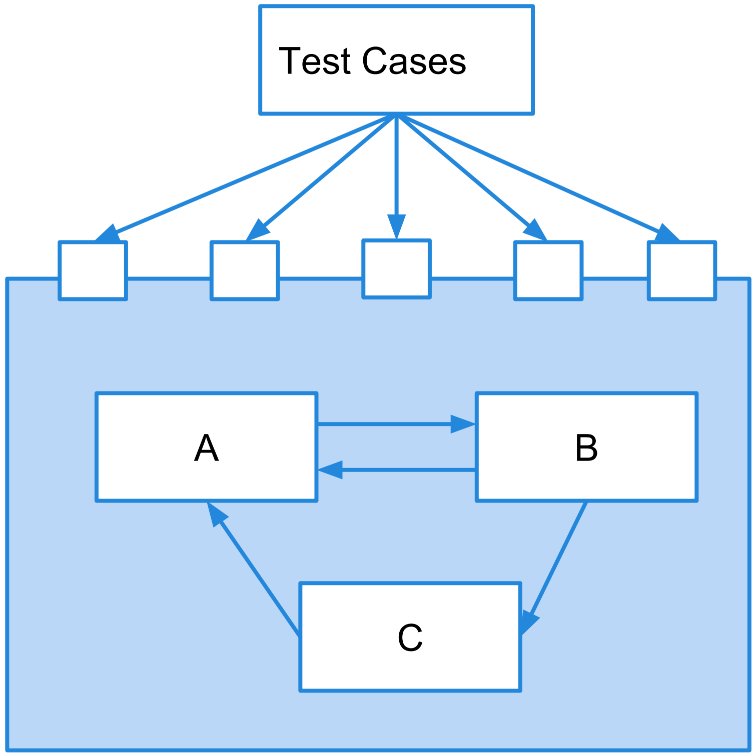
Subsystem Testing

- Most software works by combining multiple, interacting components.
  - In addition to testing components independently, we must test their *integration*.
- Functionality performed across components is accessed through a defined interface.
  - Therefore, integration testing focuses on showing that functionality accessed through this interface behaves according to the specifications.

Subsystem Testing

We have a subsystem made up of A, B, and C. We have performed unit testing...

- However, they work together to perform functions.
- Therefore, we apply test cases not to the classes, but to the interface of the subsystem they form.
- Errors in their combined behavior result are not caught by unit testing.



Interface Types

- Parameter Interfaces
  - When data is passed from one component to another.
  - All methods that accept arguments have a parameter interface.
  - If functionality is triggered by a method call, test different parameter combinations to that call.
- Procedural Interfaces
  - When one component encapsulates a set of functions that can be called by other components.
  - Such as the GRADS interface.
  - Controls access to subsystem functionality. Thus, is important to test rigorously.

Interface Types

- Shared Memory Interfaces
  - A block of memory is shared between components.
  - Data is placed in this memory by one subsystem and retrieved by another.
  - Common if system is architected around a central data repository.
- Message-Passing Interfaces
  - Interfaces where one component requests a service by passing a message to another component. A return message indicates the results of executing the service.
  - Common in parallel systems, client-server systems.

Interface Errors

- Interface Misuse
  - A calling component calls another component and makes an error in the use of its interface.
  - Wrong type or malformed data passed to a parameter, parameters passed in the wrong order, wrong number of parameters.
- Interface Misunderstanding
  - Incorrect assumptions made about the called component.
  - A binary search called with an unordered array.
- Timing Errors
  - In shared memory or message passing - producer of data and consumer of data may operate at different speeds, and may access out of data information as a result.

Interface Testing Guidelines

- Pass extreme parameters to any call to another subsystem.
- When pointers are passed across an interface, always test the interface with null pointers.
- Where a component is called through a procedural interface, design tests that will cause that component to fail.
- Use stress testing in message-passing systems.
- When shared memory is used, vary the order of component activation.

System Testing

Systems are developed as interacting subsystems. Once units and subsystems are tested, the combined system must be tested.

- Advice about interface testing still important here (GRADS interface is the primary means of entry to the system).
- Two important differences:
  - Reusable components (off-the-shelf systems) need to be integrated with the newly-developed components.
  - Components developed by different team members or groups need to be integrated.

Acceptance Testing

Once the system is internally tested, it should be placed in the hands of users for feedback.

- Users must ultimately approve the system.
- Many faults do not emerge until the system is used in the wild.
  - Alternative operating environments.
  - More eyes on the system.
  - Wide variety of usage types.
- Acceptance testing allows users to try the system under controlled conditions.

Acceptance Testing Types

Three types of user-based testing:

- Alpha Testing
  - A small group of users work closely with development team to test the software.
- Beta Testing
  - A release of the software is made available to a larger group of interested users.
- Acceptance Testing
  - Customers decide whether or not the system is ready to be released.

Acceptance Testing Stages

- Define acceptance criteria
  - Work with customers to define how validation will be conducted, and the conditions that will determine acceptance.
- Plan acceptance testing
  - Decide resources, time, and budget for acceptance testing. Establish a schedule. Define order that features should be tested. Define risks to testing process.
- Derive acceptance tests.
  - Design tests to check whether or not the system is acceptable. Test both functional and non-functional characteristics of the system.

Acceptance Testing Stages

- Run acceptance tests
  - Have users complete the set of tests. This should take place in the same environment that they will use the software. Some training of end-users may be required.
- Negotiate test results
  - It is unlikely that all of the tests will pass the first time. The developer and customer must negotiate to decide if the system is good enough or if it needs more work.
- Reject or accept the system
  - Developers and customer must meet to decide whether the system is ready to be released.

We Have Learned

- More about verification and validation than you ever wanted to know.
- Testing terminology and definitions.
- Testing stages include unit testing, subsystem testing, system testing, and acceptance testing.
    - and what a unit test looks like.

Next Time

- Structural (White-Box) Testing
  - Using the source code to derive test cases.
- Homework 4
  - Any questions?
  - Feedback soon!