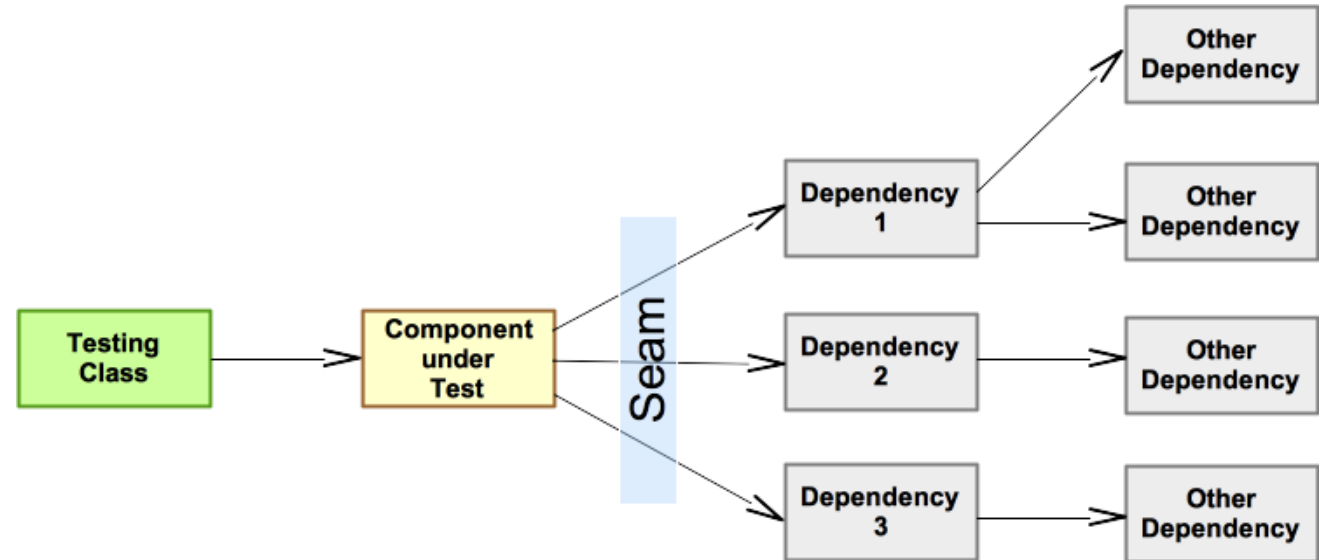


Unit Testing



SWEN-261

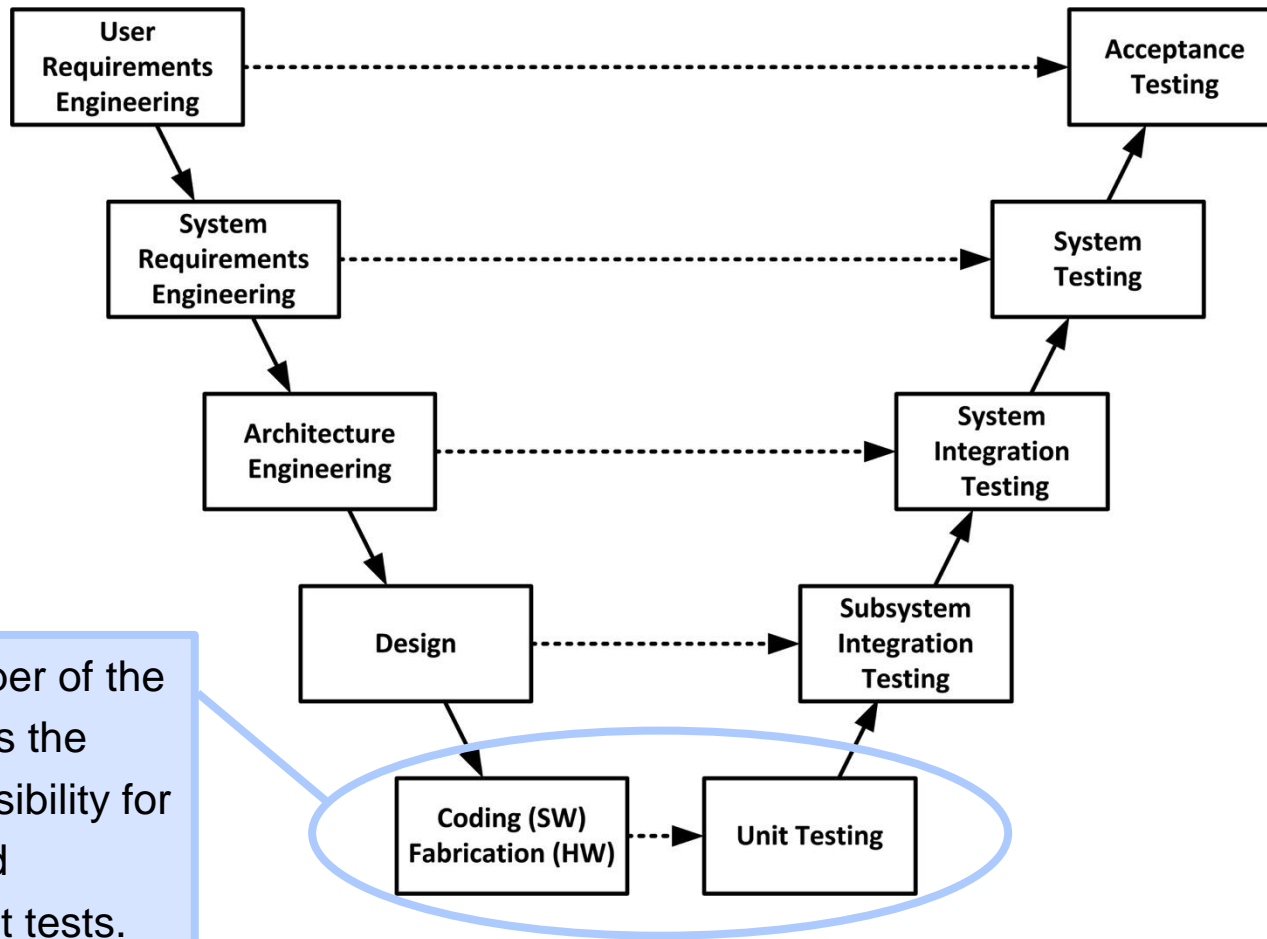
Introduction to Software Engineering

Department of Software Engineering
Rochester Institute of Technology



Software Engineering
Rochester Institute
of Technology

There are many levels of software testing.



https://insights.sei.cmu.edu/sei_blog/2013/11/using-v-models-for-testing.html

The scientific method that you studied in your physics courses is not just for science!

- Software testing and debugging is experimentation using the scientific method
 - ***Make a hypothesis***
 - ◆ Your software executes per the requirements
 - ◆ A certain place in your code is responsible for a bug
 - ***Create an experiment to test the hypothesis***
 - ◆ Write unit tests
 - ◆ Determine the scenario that triggers the bug and set breakpoints where you think the bug arises
 - ***Run the experiment***
 - ◆ Run the unit tests
 - ◆ Run the system to the breakpoint
 - ***Analyze the results to decide if the hypothesis is true***
 - ◆ If unit tests pass, the hypothesis is true
 - ◆ If the problem is seen, you have found the bug location



These are the essential characteristics that should be your goals when you write unit tests.

- Automatic
- Thorough
- Repeatable
- Independent
 - *Test only one thing at a time*
 - *Tests should not depend on each other*
- Professional
 - *Readable*
 - *Use standard OO principles*
- Fast
 - *Using Maven, unit tests are run on each build*

This is the checklist of items that you should ensure get tested by a class' unit tests.

- Business logic
 - *Tests for each path through a method*
 - *Happy path as well as failure paths*
- Constructors and accessors
- Defensive programming checks
 - *Validation of method arguments*
 - ◆ `NullPointerException`
 - ◆ `IllegalArgumentException`
 - *Validation of component state*
 - ◆ `IllegalStateException`
- The `equals` and `hashCode` methods, as needed
- Exception handling

Here's the typical formula for a test.

- Create a test class for each component under test (CuT) that has privileged access to the component.
 - *In Java, the test code typically has package access.*
- Create a test method that covers each happy and failure scenario.
 - *Setup test scenario*
 - *Invoke the test*
 - *Analyze the test results*
- This simple formula can be adjusted as needed.
- Note: see the [after-class exercise](#) for more detailed instructions for unit testing with JUnit, Mockito, and Maven along with examples from the **GuessGame** unit tests.



Unit testing frameworks like JUnit have many built-in assertions for analysis in your tests.

- Test truth-hood
 - *assertTrue(condition[, message])*
 - *assertFalse(condition[, message])*
- Test values or objects for equality
 - *assertEquals(expected, actual[, message])*
 - *assertNotEquals(expected, actual[, message])*
- Test objects for identity (`obj1 == obj2`)
 - *assertSame(expected, actual[, message])*
 - *assertNotSame(expected, actual[, message])*
- Test null-hood
 - *assertNull(object[, message])*
 - *assertNotNull(object[, message])*
- Automatic failure: *fail(message)*

One tip for coding for testability is to make strings and constants available to the unit test.

■ Idioms for testable code:

- *Make message strings (and other values) constants*
- *Make these members package-private (or public)*

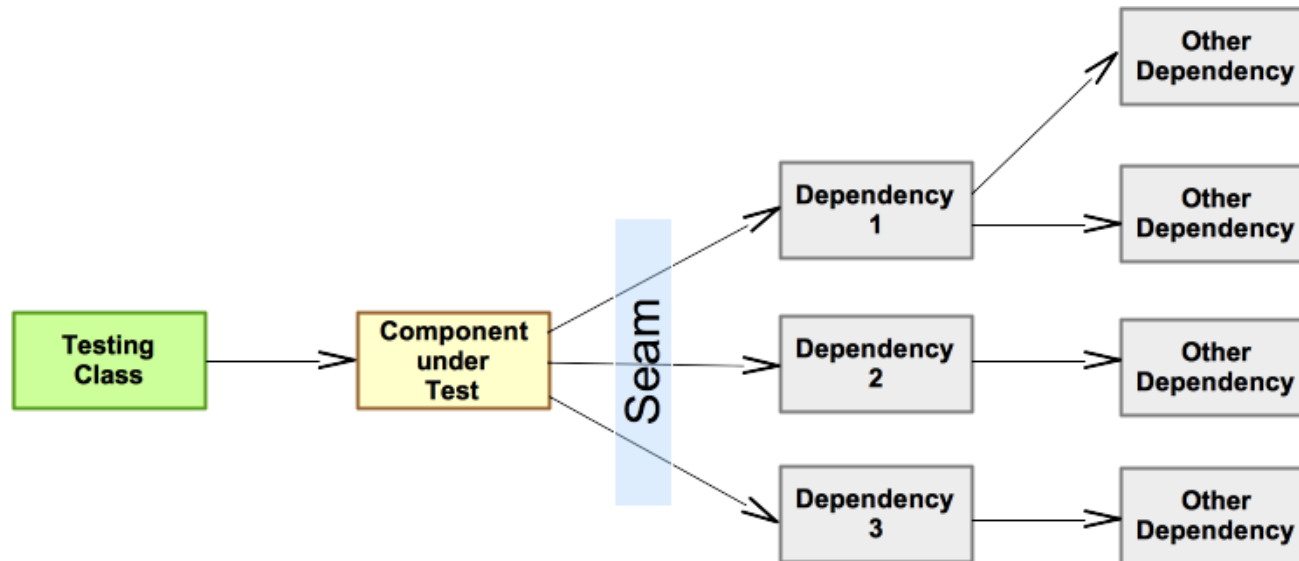
■ Example:

```
public class GameCenter {  
  
    final static String NO_GAMES_MESSAGE = "No games have been played so far.";  
    final static String ONE_GAME_MESSAGE = "One game has been played so far.";  
    final static String GAMES_PLAYED_FORMAT = "There have been %d games played.";  
  
    public synchronized String getGameStatsMessage() {  
        if (totalGames > 1) {  
            return String.format(GAMES_PLAYED_FORMAT, totalGames);  
        } else if (totalGames == 1) {  
            return ONE_GAME_MESSAGE;  
        } else {  
            return NO_GAMES_MESSAGE;  
        }  
    }  
}
```



To write an "independent" unit test of a class, you will need to control the dependencies.

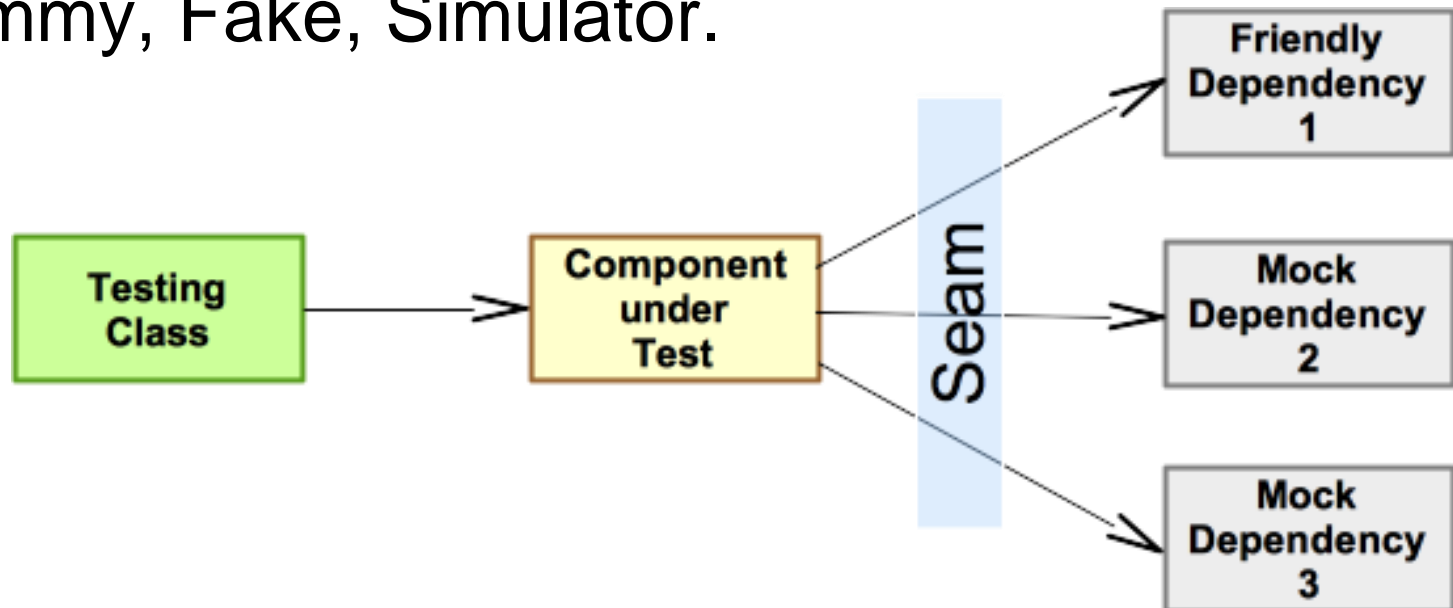
- Most components will have dependent classes
 - *Stored as attributes*
 - *Passed in as arguments*
- This coupling forms a "seam" between the component under test and its dependencies:



- Use dependency injection rather than direct instantiation.

Isolate the CuT by injecting *friendly* and mock objects.

- Fully tested internal components (usually Model objects like Entities and Value Objects)
- Mock objects that stand in for internal or external components
- There are other variations on test instruments: Spy, Dummy, Fake, Simulator.



Mock object frameworks like Mockito have APIs for setting up test scenarios and inspecting results.

- Setting up test scenarios
 - *Specify the response, e.g. return value or exception thrown, when a method is called on a mock object*
 - *`when(mock.method(args)).thenReturn(value)`*
- Verify that interactions occurred during a unit test
 - *Check if methods were called the expected number of times in the manner expected*
 - *`verify(mock, times(1)).method(args)`*
- Answer objects that allow you to capture and later test intermediate results of unit test



Make unit testing part of your project's *Definition of Done* and your quality will rise!

- Unit testing is a subtle art.
- Keep your test code isolated from production code.
- Use the testing formula: arrange, invoke, analyze.
- Use an Assertion library like JUnit.
- Use a Mock object framework like Mockito.
- See [after-class exercise](#) for more detailed examples of doing unit tests with JUnit, Mockito, and maven.

Make unit testing part of your project's *Definition of Done* and your quality will rise!

☒ Definition of Done Checklist [Delete...](#)

0%

- ☐ acceptance criteria are defined
- ☐ solution tasks are specified
- ☐ feature branch created
- ☐ unit tests written
- ☐ solution is *code complete*, i.e. passes full suite of unit tests
- ☐ design documentation updated
- ☐ pull request created
- ☐ user story passes all acceptance criteria
- ☐ code review performed
- ☐ feature branch merged into master
- ☐ feature branch deleted

See [after-class exercise](#) for more detailed examples of doing unit tests with JUnit, Mockito, and maven.