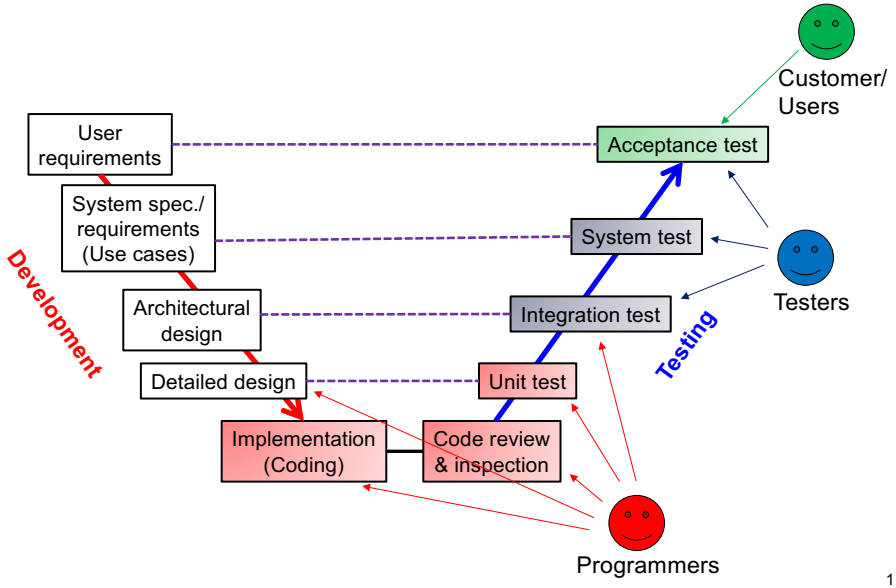
Division of Responsibilities



Test Levels and Test Types

- Test level
  - Corresponds to a "development level."
    - e.g., unit test, integration test, system test and acceptance test.
  - A group of test activities that are organized and managed together.
- Test type
  - Focuses on a particular test objective.
    - e.g. functional test, non-functional test, structural test, confirmation test, etc.
  - Takes place at one test level or at multiple levels.

- Different projects have difference policies on which test types involve in which levels.
- For example…

	Functional test	Non-functional test	Structural test	Confirmation test
Acceptance test	X	X		
System test	X	X		X
Integration test	X	?	X	X
Unit test	X	?	X	X
Code rev&insp.	X	?	X	X

Test Types: Functional Test

- Focuses on the functional (external) behaviors of tested code
  - Driven by the descriptions and use cases specified in the specification.
  - Black-box testing
    - Treat the tested code as a black-box
      - Testing *without* knowing the internals of tested code
    - Give an input to tested code and compare its output with the expected result.
      - Coarse-grained testing: Testing the external behaviors of tested code

Client code

```
Robot r = new Robot();
r.control(Robot.CMD_MOVE_FORWARD);
```

Robot
+ CMD_MOVE_FORWARD: int=0 {final} + CMD_STOP: int=1 {final} + CMD_MOVE_BACKWARD: int=2 {final}
+ control(command: int): void

Client code

```
Student s = new Student(...);
s.getTuition();
```

Student
- name: String
Student(status: StudentStatus, Name: String) +getTuition(): float +getName():String …

	Functional test	Non-functional test	Structural test	Confirmation test
Acceptance test	X	X		
System test	X	X		X
Integration test	X	?	X	X
Unit test	X	?	X	X
Code rev&insp.	X	?	X	X

Test Types: Non-Functional Test

- Focuses on the non-functional quality characteristics of tested code.
  - Driven by the descriptions specified in the specification.
  - Security test
    - Check if security vulnerability exists in tested code.
  - Usability test
    - Ease of use/browse/comprehension, intuitive page/screen transition
  - Efficiency test
    - Performance (e.g. response time, throughput), resource utilization (e.g. memory, disk, bandwidth, energy/battery)

  - Reliability test
    - Stress test (load test)
      - How does tested code behave under an excessive load?
        - » Example loads: huge data inputs, numerous network connections
    - Long-run test
      - Does performance degrade when tested code runs for a long time?
    - High frequency test
      - How does tested code behave when it repeats a certain task at excessively high frequency?
    - Fault-tolerance test
      - Can a tested code continue its operation under a fault?
    - Recoverability test
      - How can a tested code recover its operation and data after a disaster (e.g. physical damages of hardware, blackout)?
    - Compliance test
      - Data retention, access control, logging, etc.

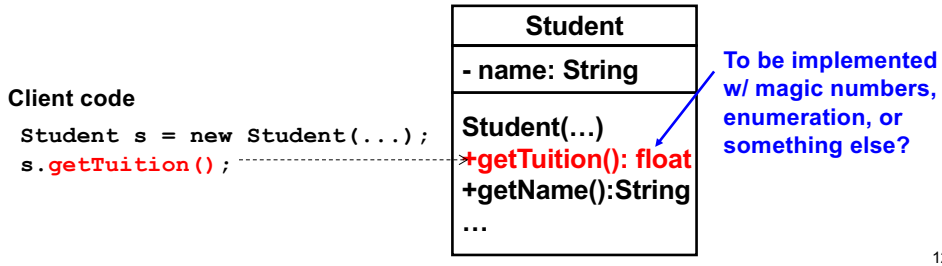
- Environmental test
  - Configuration/compatibility test
    - Can the tested code be installed on certain OS(es) and HW(s)?
    - How does the tested code behave on certain OS(es) and HW(s)?
    - How does the tested code interact with an external required service(s)?
      - » Does it work with Version X of the service? How about Version Y?
  - Co-existence test
    - Can the tested code run correctly when other software/services run on the same machine?

	Functional test	Non-functional test	Structural test	Confirmation test
Acceptance test	X	X		
System test	X	X		X
Integration test	X	?	X	X
Unit test	X	?	X	X
Code rev&insp.	X	?	X	X

Test Types: Structural Test

- Testing the internal structure of an individual module or a set of (integrated) modules.
- Revise the structure, if necessary, to improve maintainability, flexibility and extensibility.
  - Refactoring
    - » e.g. Replacing conditionals with polymorphism, replacing a magic number with a symbolic constant.
    - » Revising the interfaces of modules if an integration test fails.
      - » Interface: How modules interact with each other
  - Use of design patterns
    - » e.g., Replacing conditionals with the *State* design pattern

- White-box testing
  - Treat tested code as a white-box
    - Testing *with* the knowledge about the internals of the tested code
  - Fine-grained testing: Taking care of internal behaviors of tested code



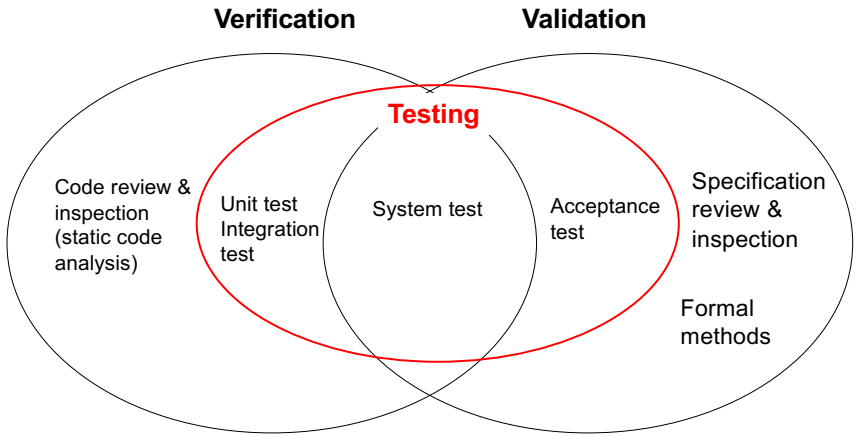
	Functional test	Non-functional test	**Structural test**	Confirmation test
Acceptance test	X	X		
System test	X	X		X
Integration test	X	?	**X**	X
Unit test	X	?	**X**	X
Code rev&insp.	X	?	**X**	X

Test Types: Confirmation Test

- Re-testing
  - When a test fails, detect a defect and fix it. Then, execute the test again
    - To confirm that the defect has been fixed.
- Regression testing
  - In addition to re-testing, execute ALL tests to confirm that the tested code has not regressed.
    - That is, it does not have extra defects as a result of fixing a bug.
    - Verifying that a change in the code has not caused unintended negative side-effects and it still meets the specification.

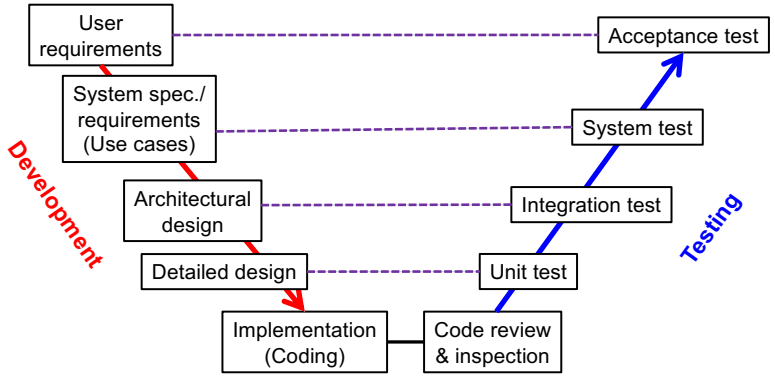
	Functional test	Non-functional test	Structural test	**Confirmation test**
Acceptance test	X	X		
System test	X	X		**X**
Integration test	X	?	X	**X**
Unit test	X	?	X	**X**
Code rev&insp.	X	?	X	**X**

V/V Methods

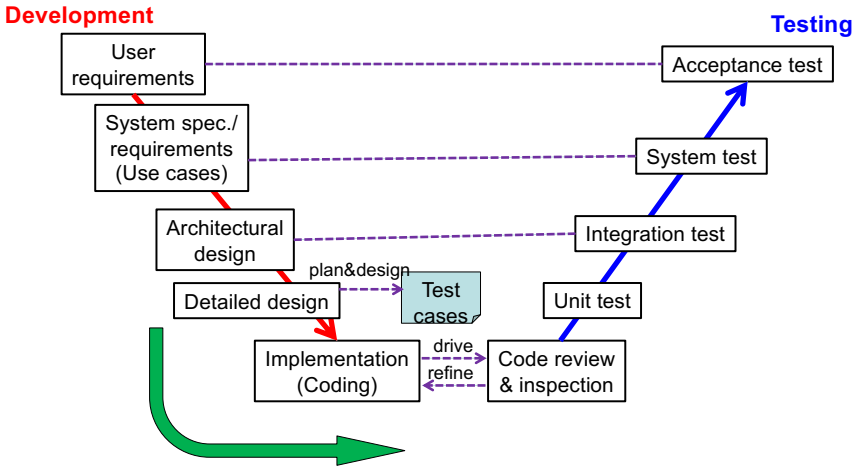


V-Model and Development Process

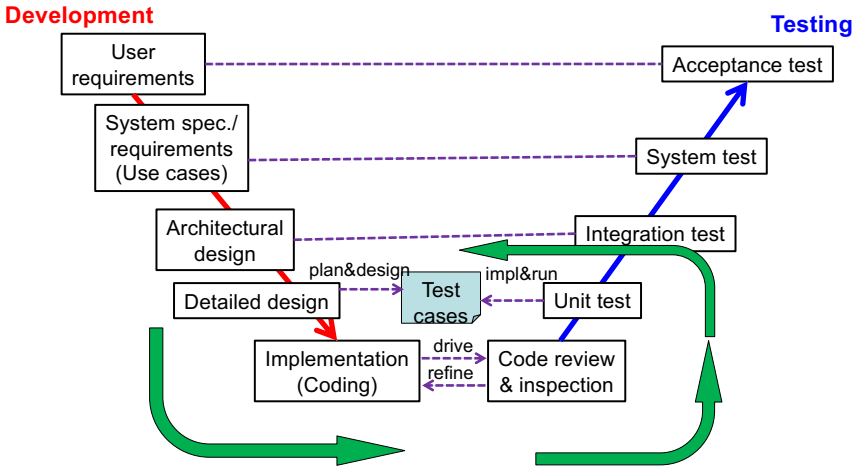
- V-Model can be used to define various iterative development processes,
  - beyond waterfall process



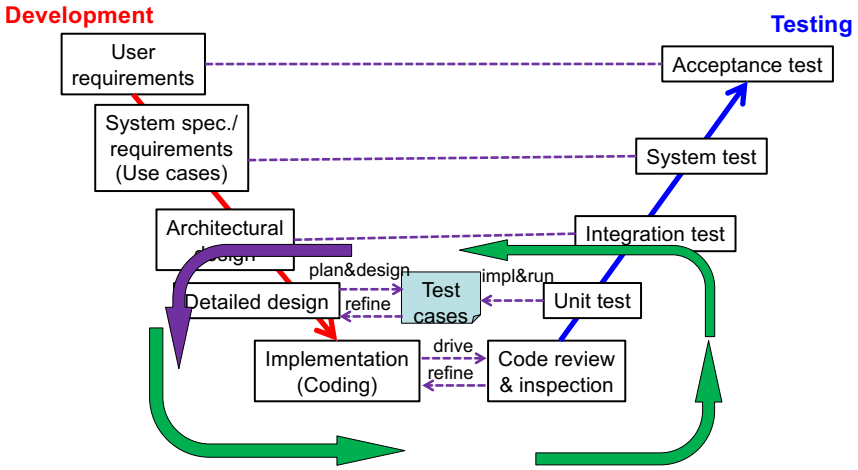
17



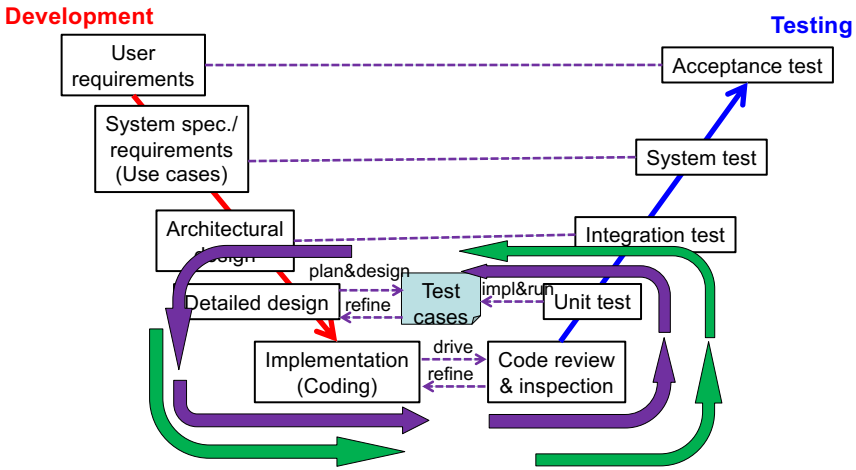
18

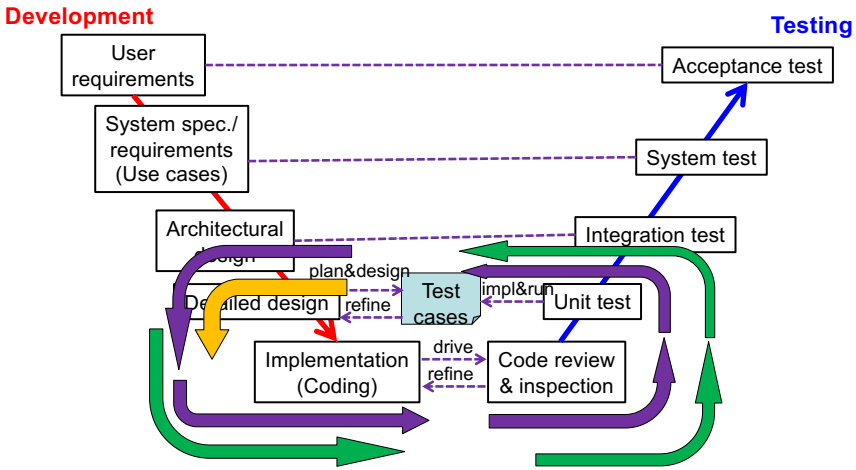


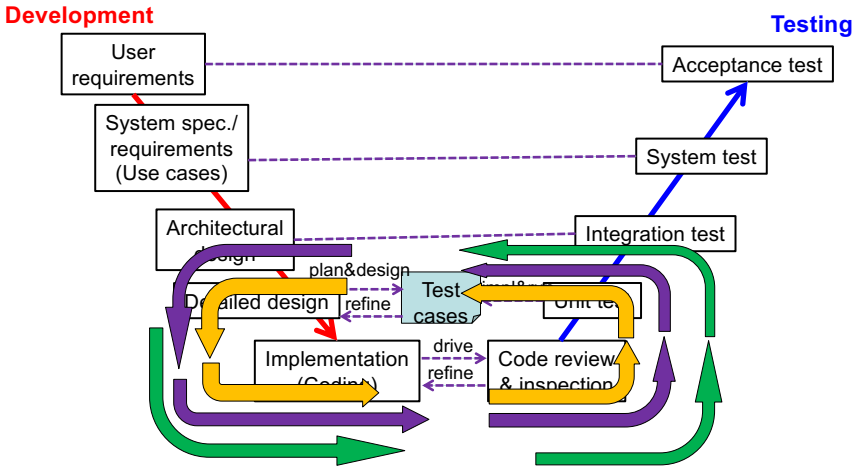
19



20



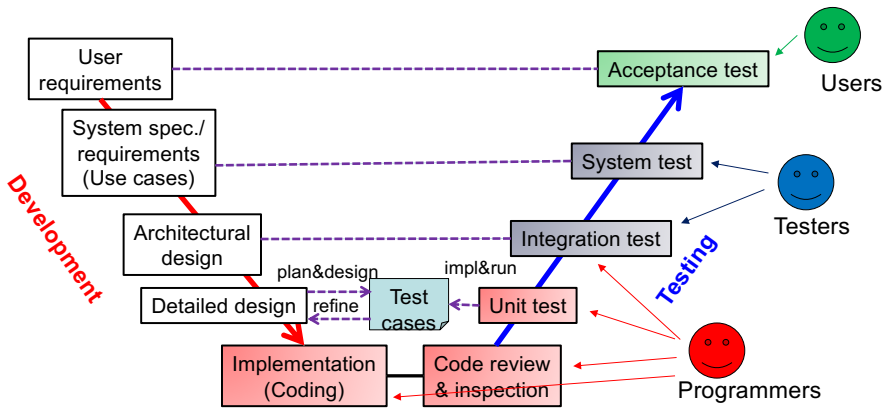




Unit Testing

Unit Tests

- Verify that each *program unit* (program module) works as expected along with the system specification.
  - Units to be tested: classes (methods in each class) in OOPLs



Who Does it?

- You!
  - as a programmer
- Test cases are written *as programs* from a programmer's perspective.
  - A test case describes a test to verify a tested class in accordance with the system specification.

Who Does it?

- Programmers and unit testers are no longer separated in most projects as
  - Useful tools have made unit testing a lot easier and less time-consuming.
  - Programmers can write the best test cases for their own code in the least amount of efforts.

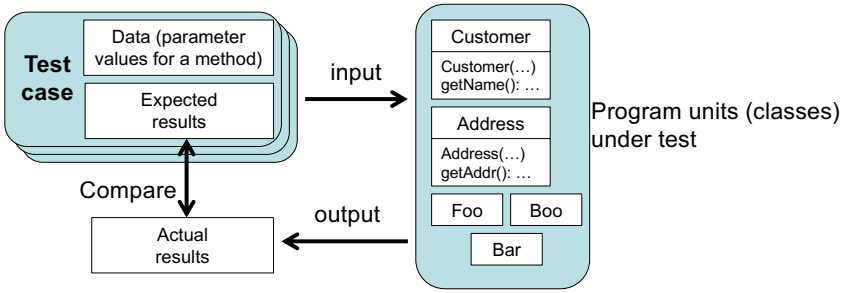
What to Do in Unit Testing?

- 4 possible test types
  - CS680 will focus on 3 of them: *functional*, *structural* and *confirmation* tests.

	Functional test	Non-functional test	Structural test	Confirmation test
Acceptance test				
System test				
Integration test				
Unit test	**X (B-box)**		**X (W-box)**	**X (Reg test)**
Code rev&insp.				

Functional Test in Unit Testing

– Ensure that each method of a tested class successfully performs a set of specific tasks.
  - Each test case confirms that a method produces the expected output when a known input is given.
    – Black-box test
  - Well-known techniques: equivalence test, boundary value test, etc.



Structural Test in Unit Testing

- Verify the structure of each class.
- Revise the structure, if necessary, to improve maintainability, flexibility and extensibility.
  – White-box test

- To-dos
  – Refactoring
  – Use of design pattern
  – Control flow test
  – Data flow test

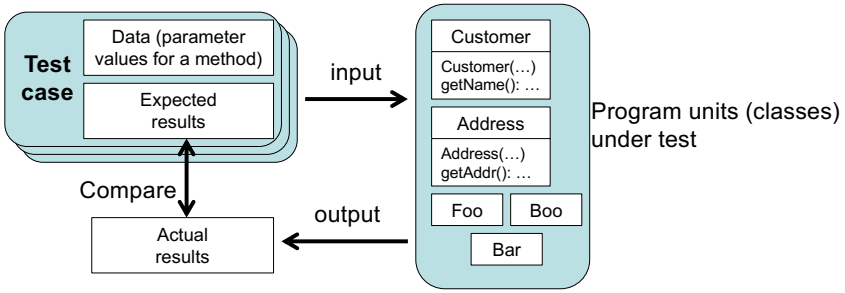
Confirmation Test in Unit Testing

- Re-testing
- Regression testing

Unit Testing with JUnit

Functional Test in Unit Testing

– Ensure that each method of a class under test successfully performs a set of specific tasks.
  • Each test case confirms that a method produces the expected output when given a known input.



JUnit

• A unit testing framework for Java code
  – Defines the format of each test case
    • Test case: a program to verify a method(s) of a class under test with a set of inputs/conditions and expected results.
  – Provides APIs to write and run test cases
  – Reports test results
  – Makes unit testing as easy and automatic as possible.
• Version 5.x, http://junit.org/junit5/

Test Classes and Test Methods

• Test class
  – A public class that has a set of test methods
  – Common naming convention: **`XYZTest`**
    • **`XYZ`** is a class under test.
  – One test class for one class under test, in principle
• Test method
  – A public method in a test class.
    • Takes no parameters
    • Returns no values (“void”)
    • Can have a “throws” clause
  – Annotated with **`@Test`**
    • `org.junit.jupiter.api.Test`
  – One test method implements one test case.

Assertions

• Each test method verifies one or more *assertions*.
  – Assertion: a statement that a predicate (Boolean expression) should be always true at a particular point in code.
    • `String line = reader.readLine();`
      Assertion: `line != null`
    • `String str = foo.getPassword();`
      Assertion: `str.length() > 20`
• In JUnit, running a unit test means verifying an assertion(s) described in a test method.

An Example

- Class under test

```
public class Calculator{

 public float multiply(float x, float y){
   return x * y;
 }

 public float divide(float x, float y){
   if(y==0){ throw new IllegalArgumentException(
                                          "division by zero");}
   return x/y;
 }

}
```


- Class under test

```
public class Calculator{
 public float multiply(float x,
                       float y){
   return x * y;
 }
 public float divide(float x,
                     float y){
   if(y==0){ throw
    new IllegalArgumentException(
     "division by zero");}
   return x/y;
 }
}
```

- Test class

```
import static org.junit.jupiter.api.
Assertions.*;

import org.junit.jupiter.api.Test;

public class CalculatorTest{
 @Test
 public void multiply3By4(){
   Calculator cut = new Calculator();
   float actual = cut.multiply(3,4);
   float expected = 12;
   assertEquals(expected, actual); }

 @Test
 public void divide3By2(){
   Calculator cut = new Calculator();
   float actual = cut.divide(3,2);
   float expected = 1.5f;
   assertEquals(expected, actual)); }
}
```


- Class under test

```
public class Calculator{
 public float multiply(float x,
                       float y){
   return x * y;
 }
 public float divide(float x,
                     float y){
   if(y==0){ throw
    new IllegalArgumentException(
     "division by zero");}
   return x/y;
 }
}
```

- Test class

```
import static org.junit.jupiter.api.
Assertions.*;

import org.junit.jupiter.api.Test;

public class CalculatorTest{
 @Test
 public void divide5By0(){
   Calculator cut = new Calculator();
   try{
     cut.divide(5, 0);
     fail("Division by zero");
   }
   catch(IllegalArgumentException ex){
     assertEquals("division by zero",
                  ex.getMessage() );
   }
 }
}
```


Key APIs

- **`org.junit.jupiter.api.Assertions`**
  - Utility class (i.e., a set of static methods) to write assertions.
    - **`assertEquals(expected, actual)`**
      - Asserts that *expected* and *actual* are equal.
    - **`fail(String message)`**
      - Fails a test with the given failure message.
    - **`assertNotNull(Object actual)`**
      - Asserts that *actual* is not null.
    - **`assertNull(Object actual)`**
      - Asserts that *actual* is null.
    - **`assertTrue(Boolean actual)`**
      - Asserts that *actual* is true.
    - **`assertFalse(Boolean actual)`**
      - Asserts that *actual* is false.

Key Annotations

- `@Test`
  - `org.junit.jupiter.api.Test`
  - Denotes that an annotated method is a test method.
  - JUnit runs all the methods that are annotated with @Test.
- `@Disabled`
  - `org.junit.jupiter.api.Disabled`
  - Used to disable a test class or test method
  - JUnit skips running all disabled methods and classes.
    - No need to comment out the entire test method/class.

Static Imports

- Static methods of `Assertions` is often imported via *static import*.
  - `import static org.junit.jupiter.api.Assertions.*;`
  - With static import
    - `assertEquals(expected, actual);`
      - asserting that "actual" is equal to "expected"
      - `assertEquals()` is a static method of `Assertions`.
  - With normal import
    - `Assertions.assertEquals(expected, actual);`

Things to Test

- Methods
- Exceptions
- Constructors

```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;

public class StudentTest{
  @Test
  public void constructorWithName(){
    Student cut = new Student("John");
    assertEquals("John", cut.getName());
    assertNull(cut.getAge());
}
 @Test
  public void constructorWithNameAndAge(){
    Student cut = new Student("John", 10);
    assertEquals("John", cut.getName());
    assertNonNull(cut.getAge());
}
 @Test
  public void constructorWithoutName(){
    Student cut = new Student();
    ...
}
```


Principles in Unit Testing

- Define one or more *fine-grained* test cases (test methods) for each method in a class under test.
- Give a *concrete/specific* and *intuitive* name to each test method.
  - e.g. "divide5by4"
  - Avoid something like "testDivide"
- Use specific values and conditions, and detect design and coding errors.
  - Be detail-oriented. The devil resides in the details!
- No need to worry about redundancy in/among test methods.

- No need to include the prefix “test” in each test method
  - Use @Test.
    - `@Test`
      `public void divide3By2(){`
      `...`
      `}`
    - `public void testDivide3By2(){`
      `...`
      `}`

- Write simple, short and easy to understand test cases
  - Try to write many simple test cases, rather than a fewer number of complicated test cases.
    - Make it clear what is being tested for yourself and your team mates.
    - Avoid a test case that performs multiple tasks.
    - You won’t feel bothered/overwhelmed by the number of test cases as far as they have intuitive names.
      - e.g. “divide5by4”

Expected Directory Structure

- Place all test classes under the “test” directory
  - [project top directory]
    - src [source code directory]
      - edu/umb/cs680/hw01/Foo.java
      - edu/umb/cs680/hw02/Boo.java
    - bin [byte code directory]
      - edu/umb/cs680/hw01/Foo.class
      - edu/umb/cs680/hw02/Boo.class
    - test [test code directory]
      - src
        - » edu/umb/cs680/hw01/FooTest.java
        - » edu/umb/cs680/hw02/BooTest.java
      - bin
        - » edu/umb/cs680/hw01/FooTest.class
        - » edu/umb/cs680/hw02/BooTest.class

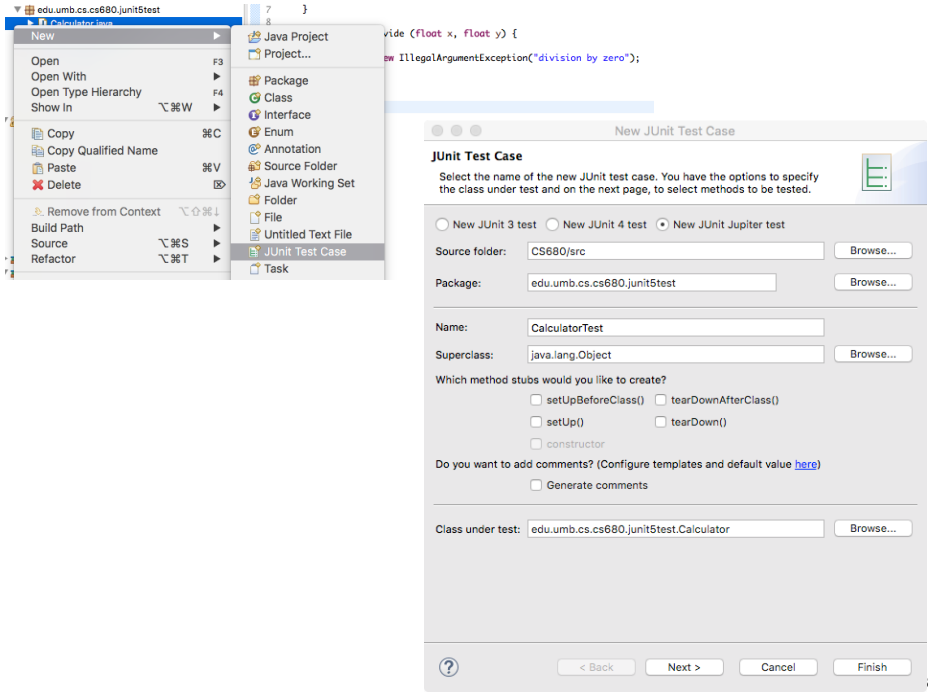
- Clearly separate the directories for source code and binary code.
- In every HW, I will **NOT** grade your work if your source and binary code are placed in the same directory.
  - [project directory]
    - edu/umb/cs680/hw01/Foo.java
    - edu/umb/cs680/hw01/Foo.class

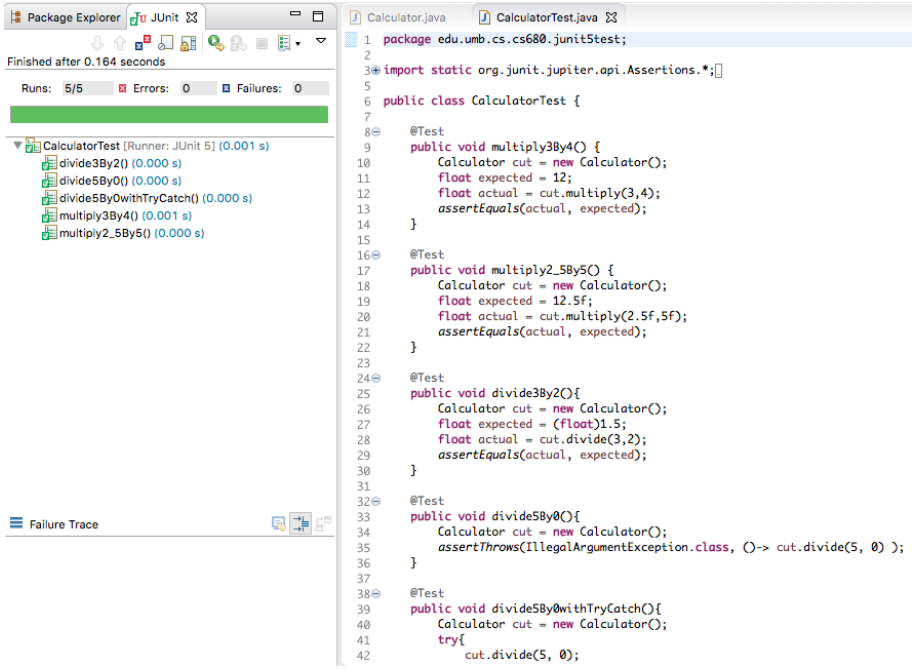
- Clearly separate the directories for test code and tested code.
- Production code should not include tests when it is delivered to the customer.
- In every HW, I will **NOT** grade your work if your test code and tested code are placed in the same directory.
  - [project directory]
    - src [source code directory]
      - edu/umb/cs680/hw01/Foo.java
      - edu/umb/cs680/hw01/FooTest.java
    - bin [byte code directory]
      - edu/umb/cs680/hw01/Foo.class
      - edu/umb/cs680/hw01/FooTest.class

- In every HW, I will **NOT** grade your work's testing portion if you use a prior version of JUnit.
  - Use JUnit 5, not version 4 or 3.

Test Runners

- How to run JUnit?
  - From command line
    - "ant" command
  - From IDEs
    - Eclipse, etc.
    - Get used to this option first.
      - How to import JUnit to your project? How to run test cases with JUnit?
  - From Ant
    - <junitlauncher> task





At This Point…

- You do NOT have to use Ant. Just get familiar with how to use JUnit on your IDE.
- Make sure to…
  - Write Java code (tested code; e.g., Calculator) and test cases (e.g., CalculatorTest) with your IDE
  - Follow the expected directory structure.
  - Compile Java code and run it on your IDE
  - Compile and run test cases with JUnit on your IDE