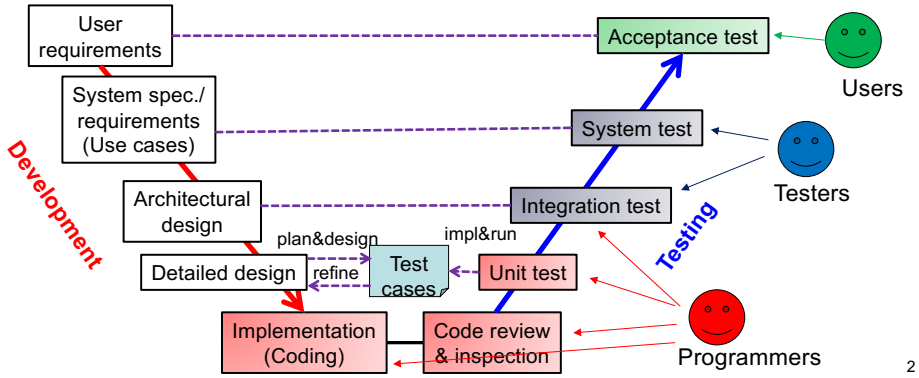
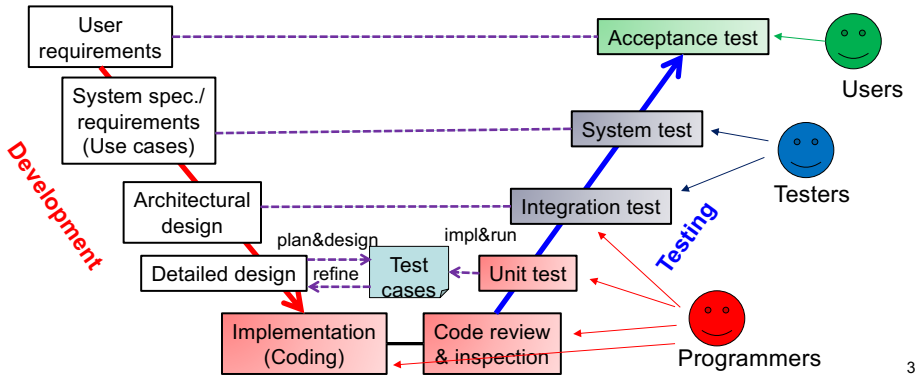
Unit Testing

Unit Tests

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



- Test case
  - Describes a test (a set of actions) to verify a tested class in accordance with the system specification.



Who Does it?

- You!
  - as a programmer
- Test cases are written *as programs* from a programmer's perspective.

- 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 (classes) in the least amount of efforts.

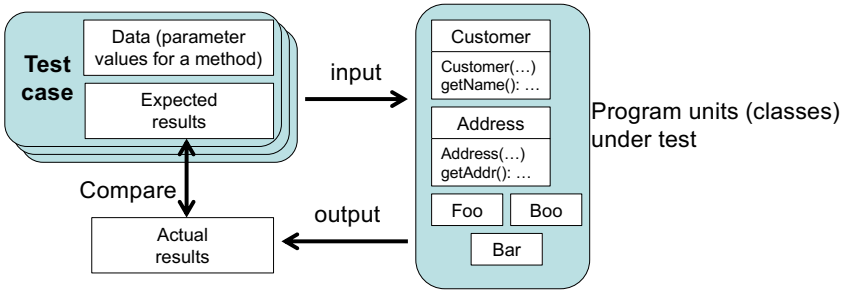
What to Do in Unit Testing?

- 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 performs a specific task successfully (as you expect)
  - 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

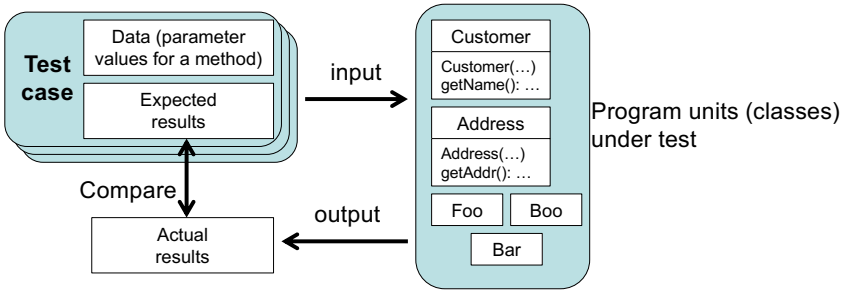
Confirmation Test in Unit Testing

- Re-testing
  - 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.

Unit Testing with JUnit

Functional Test in Unit Testing

- Ensure that each method of a tested class performs a specific task successfully (as you expect)
  - 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 of a tested class 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 throw an exception
  - Annotated with **`@Test`**
    - `org.junit.jupiter.api.Test`
  - One (or more) test method(s) implements one test case.

Assertions

- Each test method is implemented to verify 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 passwd = user.getPassword();`
      Assertion: `passwd.length() > 10`
- 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 annotated with @Test.
    - It runs them in a random order. It doesn't follow the order of test method definitions.
- **`@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`** are 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 worries about redundancy in/among test methods.

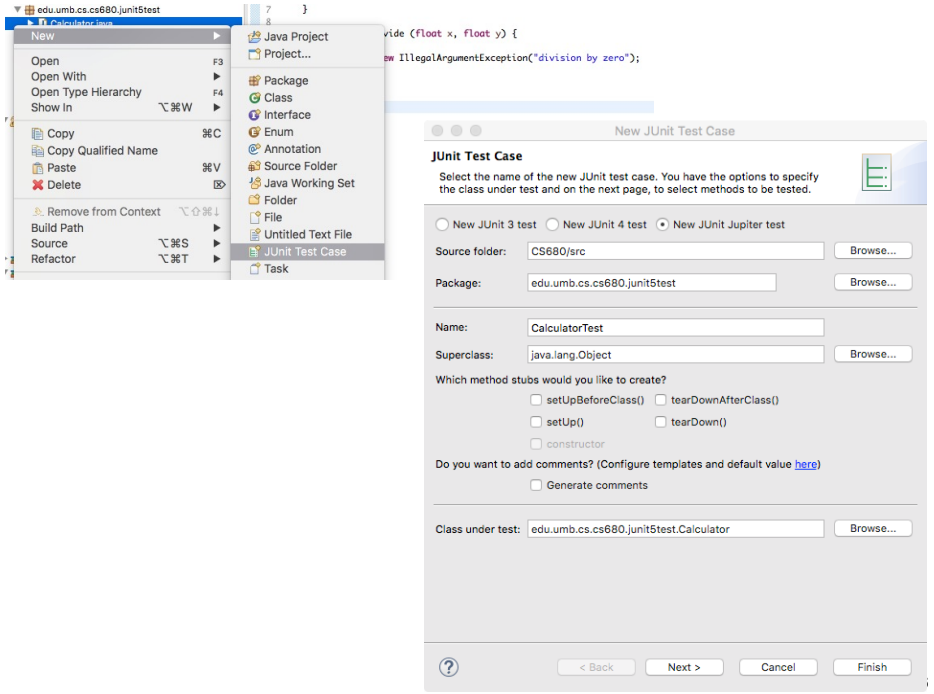
- No need to include the prefix "test" in each test method
  - Use @Test.

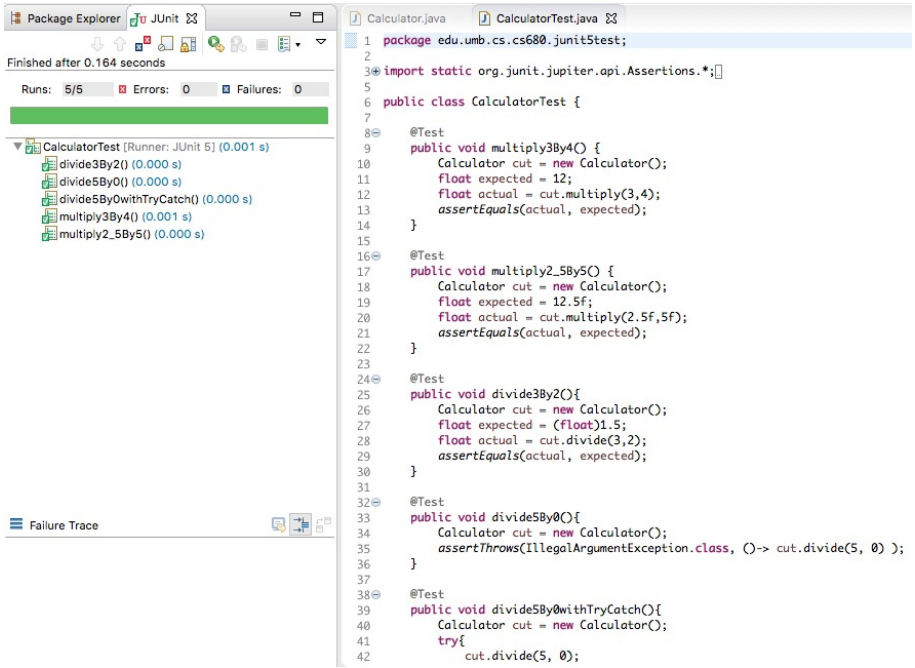
```
@Test
public void divide3By2(){
    ...
}
public void testDivide3By2(){
    ...
}
```


- 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"

Test Runners

- How to run JUnit?
  - From command line
  - 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 in build.xml





At This Point…

- You do NOT have to use Ant to run JUnit.
- 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
  - Compile Java code and run it on your IDE
  - Compile and run test cases with JUnit on your IDE

In CS680…

- Your tests should never rely on `println()` calls.
  - Use the assertion methods of JUnit instead.
  - If your HW solution uses println() for testing, it will incur some penalty.
- Your tests should never rely on `main()`.
  - Write your tests in tests methods.
  - If your HW solution contains main(), it will incur some penalty.

Why Not Just Use System.out.println() for Testing?

- Your code gets cluttered with println() statements. They will be packaged into the production code.
- You usually scan println() outputs manually every time your code runs to ensure that it behaves as expected.
- It is often hard to understand/remember the intent of each println()-based test.
  - What is tested? What is expected?

Why Not Just Write main() for Testing?

- Your classes get cluttered with test code in main(). The test code will be packaged into the production code.
- main() tends to test multiple features, which makes it hard to understand/maintain.
- If one method call fails, subsequent method calls are not executed.

```
calc.divide(5, 0); // This call fails.
calc.divide(10, 2);// This is not executed.
```


In CS680…

- In principle, you should write a test case(s) for every public method of your class.
- However, methods with obvious functionalities/behaviors do not need unit tests.
  - e.g. simple getter and setter methods
- Write a unit test whenever you feel you need to comment the behavior of a method.

- Write as many test cases as possible.
  - Try to test all public methods in principle.
  - If your HW solution tests simple getter/setter methods only and skips testing more important methods, it will lose most points in the testing portion of your work.
  - If your HW solution tests major/important methods but skip testing simple getter/setter methods, it will never incur penalty.

Use Test Cases as Documentation

- Lasting, runnable and reliable documentation on the capabilities of the classes you write.
- Can serve as sample code to use your classes/methods.
  - Useful when you forgot how to use a class/method you implemented.
  - Useful when you use a class/method that someone else implemented.
  - No need to write sample use cases and sample code in API documentation and other docs.
- Can replace a lot of comments.
  - Cannot completely replace comments, but you often do not have to write a long documentation (with javadoc, for example)

Keep Test Cases Simple

- Write *single-purpose* tests.
  - Have each test case (test method) focus on a distinctive external behavior of a tested class
  - Do not test multiple behaviors in a single test case
    - e.g., divide5by0, divide3by2, multiply3By4
      - Rather than testCalculator, testDivision
- Give a specific name to your test method, so others (and you) can understand what is being tested.
- In CS680, if you use vague method names, you will lose some points in the testing portion of your work.
  - Long, weird-looking method names are fine!

Many Naming Conventions Exist for Test Methods

- Simple naming convention
  - e.g., divide5by4, multiply3By4, divide5By0
- More expressive conventions:
  - Not only suggesting what ***context*** to be tested, suggest what ***happens*** as well by invoking some ***behavior***.
    - *doingSomethingGeneratesSomeResult*
    - divide5By0 v.s. divisionBy0GeneratesIllegalArgumentException

- Suggest what ***happens*** by invoking some ***behavior*** under a certain ***context***.
  - *doingSomethingGeneratesSomeResult*
    - divisionBy0GeneratesIllegalArgumentException
  - *someResultOccursUnderSomeCondition*
    - illegalArgumentExceptionOccursUnderDivisionBy0
  - *given*SomePrecondition*When*DoingSomething*Then*SomeResultOccurs
    - given**TwoNumbers**When**DivisionBy0**Then**IllegalArgumentExceptionOccurs**
      - divide(5,0)
    - given**TwoStrings**When**DivisionBy0**Then**IllegalArgumentExceptionOccurs**
      - divide("5", "0")
    - "Given-When-Then" style
    - "*givenSomePrecondition" can be dropped → doingSomethingGeneratesSomeResult*

- 7 popular conventions
  - https://dzone.com/articles/7-popular-unit-test-naming
- No single "correct" (or "one-size-fits-all") naming convention
  - What's good/correct depends on personal tastes, project requirements, project history, etc. etc.

- Like to include the name of a tested method?
  - divide5By0GeneratesIllegalArgumentException
    - v.s. divisionBy0GeneratesIllegalArgumentException
  - isAdultFalseIfAgeLessThan18
    - v.s. isNotAnAdultIfAgeLessThan18
  - Like to explicitly state which method is tested?
  - Like to focus on a behavior/feature that a method under test implements, not method name itself?
  - What if it is renamed?
    - Often need to rename test methods manually.
    - Method calls in test code can be automatically refactored through.

- Like to use underscores (_)?
  - given**TwoStrings**When**DivisionBy0**Then**IllegalArgumentExceptionOccurs**
  - given_TwoStrings_When_DivisionBy0_Then_IllegalArgumentExceptionOccurs
- Like to keep the name of a test method as short as possible?
  - Up to 7 or so words?