Key API: Assert

- `org.junit.Assert`
  - Contains a series of *static* "assertion methods."
    - » `assertThat( Object, org.hamcrest.Matcher )`
      - » A primitive-type value (for the 1st param) to be *autoboxed*.
        - »
```
float expected = 12;
float actual = cut.multiply(3,4);
assertThat(actual, is(expected));
```
      - » Just returns if two values (expected and actual values) match.
      - » Throws an `AssertionError` if two values do not match.
    - » `fail( String message )`
      - » Force to fail a test with a message.
      - » Throws an `AssertionError`.
    - » `assertTrue(boolean condition), assertFalse(boolean condition)`
      - » Asserts a condition is true/false.
      - » `assertTrue(account.getBalance()>0)`

Just in case… Auto-boxing and Auto-unboxing

- Automatic conversion between a primitive type value and a wrapper class instance
  -
```
int numInt = 10;
Integer numInteger = numInt;
```
  - No need to write…
  -
```
int numInt = 10;
Integer numInteger = new Integer(numInt);
// OR
Integer numInteger = Integer.valueOf(numInt);
```
  -
```
Integer numInteger = new Integer(10);
int numInt = numInteger;
```
  - No need to write…
  -
```
Integer numInteger = new Integer(10);
int numInt = numInteger.intValue();
```

Primitive type	Wrapper class
boolean	Boolean
byte	Byte
char	Character
float	Float
int	Integer
long	Long
short	Short
double	Double

- `assertEquals()` is deprecated in JUnit version 4.
  - It was a major assertion method until JUnit version 3.
  - Use `assertThat()` instead.
- Never use `assertEquals()` in your HW solutions.

Key API: CoreMatchers

- `org.hamcrest.CoreMatchers`
  - Contains static methods, each returning a matcher object that performs matching logic.
  - `is()`
    - » `assertThat(actual, is(expected))`
      - » Asserts "actual" and "expected" are equal.
    - » `assertThat(actual, is(nullValue()))`
    - » `assertThat(actual, is(notNullValue()))`
    - » `assertThat(actual, is(not(expected))`
      - » Asserts "actual" and "expected" are not equal.
    - » `assertThat(actual, is(sameInstance(expected)))`
      - » Asserts "actual" and "expected" are identical instance with the same object ID.
    - » `assertThat(actual, is(instanceOf(Foo.class)))`
      - » Asserts "actual" is an instance of Foo.
        - » Foo may be a super class of "actual"'s class.

```
» assertThat(actual, containsString("foo"))
» assertThat(actual, startsWith("foo"))
» assertThat(actual, endsWith("foo"))

» assertThat(actual, allOf(
                      notNullValue(),instanceOf(Foo.class)))
```

» Asserts all assertions hold.

```
» assertThat(actual, anyOf(
                      containsString("HTTP/1.0"),
                      containsString("HTTP/1.1")))
```

» Asserts any of the assertions (at least one of the assertions) hold.

- `hasItem()`
  - » `ArrayList<String> actual = …;`
    `assertThat(actual, hasItem("Hello"));`
- `hasItems()`
  - » `assertThat(actual, hasItems("Hello", "World");`
- `everyItem()`
  - » `assertThat(actial, everyItem("Hello");`

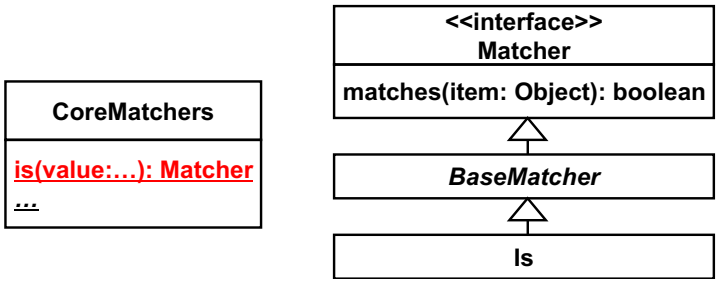
```
String[] str = {"UMass Boston", "UMass Amherst"};
ArrayList<String> actual = Arrays.asList(str);
assertThat(actual, hasItem( containsString("UMass") ));
assertThat(actual, hasItem( endsWith("Boston") );
assertThat(actual, everyItem( containsString("UMass"));
```

– It is important to learn what methods are available in `CoreMatchers` and what parameters the methods accept.

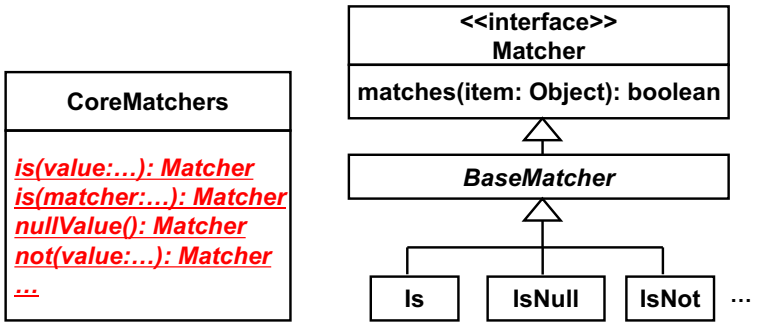
» c.f. Javadoc API documentation.

Class Structure of JUnit APIs

- `org.junit.Assert`
  - `assertThat(Object, org.hamcrest.Matcher)`
- `org.hamcrest.CoreMatchers`
  - Contains static methods, each returning a matcher object that performs matching logic.
  - `is()`
    - » `assertThat(actual, is(expected))`



- `org.junit.Assert`
  - `assertThat( Object, org.hamcrest.Matcher )`
- `org.hamcrest.CoreMatchers`
  - `assertThat(actual, is(expected))`
  - `assertThat(actual, is(nullValue()))`
  - `assertThat(actual, is(notNullValue()))`
  - `assertThat(actual, is(not(expected))`

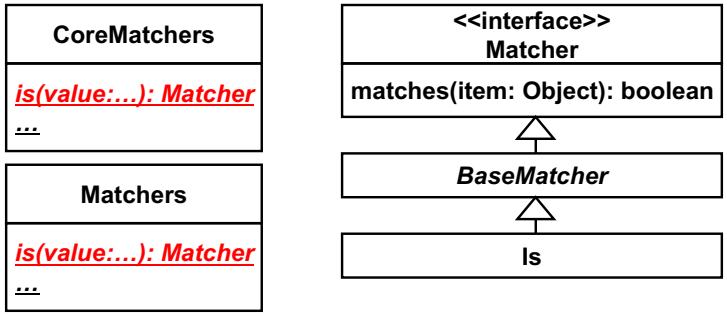


JUnit and Hamcrest

- Hamcrest provides many useful matchers for JUnit
  - junit.jar and hamcrest-core.jar from http://junit.org
  - hamcrest-all.jar from http://hamcrest.org
  - hamcrest-all.jar is a superset of hamcrest-core.jar.
    - If you use hamcrest-all.jar, you don't have to use hamcrest-core.jar.
      - No need to set both to CLASSPATH.

hamcrest-all.jar

- `org.hamcrest.CoreMatchers`
- `org.hamcrest.Matchers`
  - Contains static methods, each returning a matcher object that performs matching logic.
  - `Matchers` is a superset of `CoreMatchers`.



Matchers in hamcrest-all.jar

- Examine String data

```
assertThat("UMass Boston", containsString("UMass"))
assertThat("UMass Boston", startsWith("UMass"))
assertThat("UMass Boston", endsWith("Boston"))
assertThat("UMass Boston", equalToIgnoringCase("UMASS BOSTON"))
assertThat("  UMass", equalToIgnoringWhiteSpace("UMass"))
assertThat(actual, isEmptyOrNullString())
assertThat("", isEmptyString())
assertThat("UMass Boston", stringContainsInOrder(
                           Arrays.asList("UM", "Boston"));
```


- Examine numbers

```
assertThat(10, is(10));
assertThat(10.3, closeTo(10, 0.3)); // PASS
assertThat(10, is(greaterThan(9));
assertThat(10, is(greaterThanOrEqualTo(10));
assertThat(10, is(lessThan(11));
assertThat(10, is(lessThanOrEqualTo(10));
```

- Examine arrays
  - `String[] strArray = {"UMass","Boston"};`
    `assertThat(strArray, array( is("UMass"),`
    `startWith("B")));`
  - `assertThat(strArray, arrayContaining("UMass","Boston");`
  - `assertThat(strArray, arrayContainingInAnyOrder("Boston","UMass");`
  - `assertThat(strArray, arrayWithSize(2));`
  - `assertThat(strArray, is(not(emptyArray())));`
  - `assertThat(strArray, hasItemInArray("UMass"));`
  - `assertArrayEquals(strArray,strArray);`

- Examine collections
  - `String[] strArray = {"UMass","Boston"};`
    `ArrayList<String> actual = Array.asList(strArray);`
    `String[] expected = {"UMass","Boston"};`
  - `assertThat(actual, contains(expected));`
  - `assertThat(actual, contains("UMass", "Boston");`
  - `assertThat(actual, contains( is("Umass"),`
    `startWith("B")));`
  - `String[] expected2 = {"Boston","UMass"};`
    `assertThat(actual, containsInAnyOrder(expected2));`
  - `assertThat(actual, hasItem("UMass"));`
  - `assertThat(actual, hasItems("Boston")));`
  - `assertThat(actual, hasItems("UMass", "Boston")));`
  - `assertThat(actual, hasItem( containsString("Mass") ));`
    `assertThat(actual, hasItem( endsWith("Mass") );`
    `assertThat(actual, everyItem( containsString("s"));`

  - `String[] strArray = {"UMass","Boston"};`
    `ArrayList<String> actual = Array.asList(strArray);`
    `String[] expected = {"UMass","Boston"};`
  - `assertThat(actual, hasSize(2));`
  - `assertThat("UMass", isIn(strArray));`
  - `assertThat(actual, not(empty());`

- Examine maps
  - `HashMap<String, Integer> actual = new HashMap<String, Integer>();`
    `actual.put("foo", 0);`
    `actual.put("boo", 10);`
    `actual.put("bar", 100);`
  - `assertThat(actual, hasEntry("foo", 0));`
  - `assertThat(actual, hasEntry( endWith("oo"), greaterThan(5)) );`
  - `assertThat(actual, hasKey("bar"));`
  - `assertThat(actual, hasKey( startWith("b") ));`
  - `assertThat(actual, hasValue(0));`
  - `assertThat(actual, hasValue( lessThanOrEqualTo(100) ));`

Identity and Equality

- `assertThat(actual, is(sameInstance(expected)))`
  - Asserts "actual" and "expected" are ***identical*** instance with the same object ID.
    - `assertThat(new Foo(), is(sameInstance(new Foo())));`
    - `Foo f = new Foo();`
      `assertThat(f, is(sameInstance(f)));`
- `assertThat(actual, is(expected))`
  - A shortcut of `assertThat(actual, is(equalTo(expected)))`
- `assertThat(actual, is(not(expected)))`
  - A shortcut of `assertThat(actual, is(not(equalTo(expected))))`
  - Asserts "actual" is ***logically equal*** to "expected," as determined by calling `Object.equals(java.lang.Object)` on "actual."
    - `actual.equals(expected);`

- `String str1 = "umb";`
  `String str2 = "umb0".substring(0,2); // "umb0" -> "umb"`
- `assertThat(actual, is(sameInstance(expected))); // FAIL`
  `assertThat(actual, is(equalTo(expected)));  // PASS`
- `equalTo()` calls `String.equals()` on "actual."
  - `String.equals()` overrides `Object.equals()` and returns true if two string values match.
  - c.f. Java API doc
- Note: `Object.equals(java.lang.Object)`
  - Implements the most discriminating possible equivalence relation on objects.
    - Returns true if two objects refer to the same instance (x == y has the value true): identity check.
    - c.f. Java API doc
  - Most pre-defined (API-defined) classes override equals() to perform appropriate equality check.

- `Date d1 = new Date(); //java.util.Date`
  `Date d2 = new Date();`
- `assertThat(actual, is(sameInstance(expected))); // FAIL`
  `assertThat(actual, is(equalTo(expected)));  // PASS, most likely`
- `equalTo()` calls `Date.equals()` on "actual."
  - `Date.equals()` overrides `Object.equals()` and returns true if two Date objects represent the same timestamp in millisecond.
  - c.f. Java API doc
- Most pre-defined (API-defined) classes override equals() to perform appropriate equality check.
- You need to override equals() in your own (i.e. user-defined) class, if you want to do equality check.

Equality Check for a User-defined Class

- `Person p1 = new Person("John","Doe");`
  `Person p2 = new Person("John","Doe");`
  `Person p3 = new Person("Jane", "Doe");`
- `assertThat(p1, is(sameInstance(p1))); // PASS`
  `assertThat(p1, is(sameInstance(p2))); // FAIL`
  `assertThat(p1, is(equalTo(p2)));  // FAIL`
- Person just inherits equals() from Object. The method just do identity check.
  - You need to override equals() in Person if you want equality check.

Person
- firstName: String - lastName: String
+ Person(first:String, last:String) + getFirstName(): String + getLastName(): String

```
Person p1 = new Person("John","Doe");
Person p2 = new Person("John","Doe");
Person p3 = new Person("Jane", "Doe");
```

```
assertThat(p1, is(sameInstance(p2))); // FAIL
assertThat(p1, is(equalTo(p2)));      // PASS
assertThat(p1, is(sameInstance(p3))); // FAIL
assertThat(p1, is(equalTo(p3)));      // FAIL
```

Person
- firstName: String - lastName: String
+ Person(first:String, last:String) + getFirstName(): String + getLastName(): String + equals(anotherPerson:Object): boolean

```
if( this.firstName.equals(((Person)anotherPerson).getFirstName())
    && this.lastName.equals(((Person)anotherPerson).getLastName())){
  return true;
}
else{
  return false;
}
```


Alternatively…

```
Person p1 = new Person("John","Doe");
Person p2 = new Person("John","Doe");
Person p3 = new Person("Jane", "Doe");
```

```
assertThat(p1, is(sameInstance(p2)));                              // FAIL
assertThat(p1.getFirstName(), is(equalTo(p2.getFirstName())));     // PASS
assertThat(p1.getLastName(), is(equalTo(p2.getLastName())));       // PASS

assertThat(p1, is(sameInstance(p3)));                              // FAIL
assertThat(p1.getFirstName(), is(equalTo(p3.getFirstName())));     // FAIL
assertThat(p1.getLastName(), is(equalTo(p3.getLastName())));       // FAIL
```


One More Method in Assert

- `org.junit.Assert`
  - `assertArrayEquals(expecteds, actuals)`
    - » Assert two arrays are equal (i.e. all element values are equal in the two arrays)
      - » Can accept an primitive-type arrays and Object arrays
        - » Primitive types: boolean, byte, char, double, float, int, long, short
        - » You can pass an array of arrays (multi-dimensional array) to `assertArrayEquals(Object[], Object[])`.
  - Assert has some extra methods, but you don't really have to learn/use them.


```
int[] i1 = {2,0,0,0};
int[] i2 = {2,0,0,0};
assertArrayEquals(i1, is(i2)); // PASS
```

```
String[] str1 = {"UMass","Boston"};
String[] str1 = {"UMass","Amherst"};
assertArrayEquals(str1, is(str2)); // FAIL
```

```
Person[] persons1 = {new Person("Mickey", "Mouse"),
                     new Person("Minnie", "Mouse")};
Person[] persons2 = {new Person("Mickey", "Mouse"),
                     new Person("Hanna", "Suzuki")};
```

```
assertArrayEquals(persons1, persons2); // PASS if...
```

```
assertThat(new Person("Mickey","Mouse"),
           is(new Person("Mickey","Mouse"))); // PASS if ...
```

Person
- firstName: String - lastName: String
+ Person(first:String, last:String) + getFirstName(): String + getLastName(): String

Tips for Unit Testing

What to Test?

- In principle, you should write a unit test(s) for every public method of your class.
- However, methods with very 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.

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 looooong Javadoc comments.

Keep Test Cases Simple

- Write *single-purpose* tests.
  - Have each test case (test method) focus on a distinctive behavior of a tested class
  - Do not test multiple/many behaviors in a single test case
    - e.g., divide5by4, multiply3By4
      - Rather than testCalculator, testCalculation

Use Specific and Meaningful Names

- Give a specific and meaningful name to each test case.
  - e.g., divide5by4, multiply3By4, divide5By0
    - Rather than testDivide (or testDivision), testMultiply (or testMultiplication)
    - Do not even name it like ATest, BasicTest or ErrorTest.
  - 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*

Many Many Naming Conventions Exist

- 7 popular conventions
  - https://dzone.com/articles/7-popular-unit-test-naming
- No single “correct” way exists to name test methods.
  - Personal taste, project history…

- 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.

- Like to use underscores (_)?
  - givenTwoStringsWhenDivisionBy0ThenIllegalArgumentExceptionOccurs
  - 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?

- No need to include the prefix "test" in each test method
  - JUnit now encourages you to use @Test.

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


Testing Exceptions to be Thrown

- *Positive* tests
  - Verifying tested code runs without throwing exceptions
- *Negative* tests
  - Testing is not always about ensuring that tested code runs without errors/exceptions.
  - Sometimes need to verify that tested code throws an exception(s) when expected.

Positive Tests

```
@Test
public void writeToTestFile(){
 BufferedWriter writer = new BufferedWriter(
                          new FileWriter("test.txt");
 try{
   writer.write("test data");
 }
 catch(IOException ex){
   fail();
 }
 finally{
   writer.close();
 }
}
```

- When `write()` throws an `IOException`, this test case fails with `fail()`. Otherwise, the test case passes.
- Clear, logic-wise, but try-catch-finally blocks can clutter a test case.

- Alternative strategy
  - Have a test case *re-throw* an exception
    - rather than *catching* it.

```
@Test
public void writeToTestFile() throws IOException {
 BufferedWriter writer = new BufferedWriter(
                         new FileWriter("test.txt");
 writer.write("test data");
 writer.close();
}
```

- JUnit's test runner (i.e. the client code that calls `readFromTestFile()`) will catch an `IOException`.
  - `write()` throws it originally, and `readFromTestFile()` re-throws it.

Negative Tests

- Verify that tested code throws an exception(s) when expected.
  - Understand the conditions that cause tested code to throw each exception and test those conditions in test cases

- 3 Common ways
  - Specify an expected exception(s) with @Test
  - Write a test case with try-catch blocks
  - Specify an expected exception(s) with @Rule


```
@Test(expected=illegalArgumentException.class)
public void divide5By0(){
 Calculator cut = new Calculator();
 cut.divide(5,0); }
}
```

```
public void divide5By0(){
 Calculator cut = new Calculator();
 try{
   cut.divide(5,0);
   fail();
 }
 catch(IllegalArgumentException ex){
   assertThat(ex.getMessage(),
              equalTo("division by zero"));
 }
}
```



```
@Rule
public ExpectedException thrown = ExpectedException.none();

public void divide5By0(){
 thrown.expect(illegalArgumentException.class);
 thrown.expectMessage("division by zero");
 Calculator cut = new Calculator();
 cut.divide(5,0);
}
```

- `@Rule`
  - `org.junit.Rule`
  - Used to annotate data fields that reference *rules*.

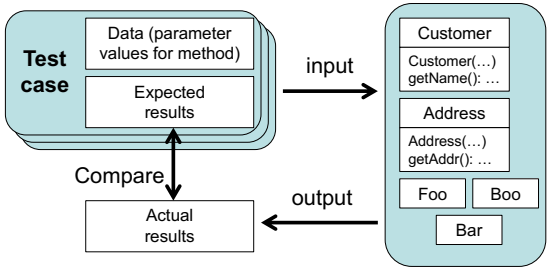
- `org.junit.rules.ExpectedException`
  - Used to verify that tested code throws a specific exception.
  - `none()`: Returns a rule that expects no exception to be thrown (identical to behavior without this rule).

```
@Rule
public ExpectedException thrown = ExpectedException.none();

public void divide5By0(){
 thrown.expect(illegalArgumentException.class);
 thrown.expectMessage("division by zero");
 Calculator cut = new Calculator();
 cut.divide(5,0);
}
```

- `org.junit.rules.ExpectedException`
  - Used to verify that tested code throws a specific exception.
  - `expect()`: Specify an exception to be thrown.
- The test case passes if the specified exception is thrown at some point when running the rest of the test.
  - It fails otherwise.

Test Fixtures

- Fixture
  - An instance of a class under test
    - A state of the class instance
  - An instance of another class that the class under test depends on
    - A state of that class instance
  - Input data
  - Expected result(s)
  - Set up of a file(s) and other resources
    - e.g., Socket
  - Set up of external systems/frameworks
    - e.g. Database, web server, web app framework, emulator (e.g. Android emulator)



Setting up Fixtures

- Class under test

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

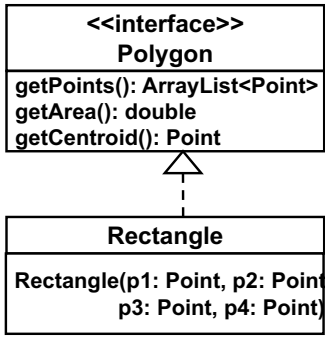
- Test class

```
import static org.junit.Assert.*;
import static org.hamcrest.CoreMatchers.*;
import org.junit.Test;

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

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

@Test(expected=illegalArgumentException.class)
 public void divide5By0(){
   Calculator cut = new Calculator();
   cut.divide(5,0); }
}
```

Setting up fixtures

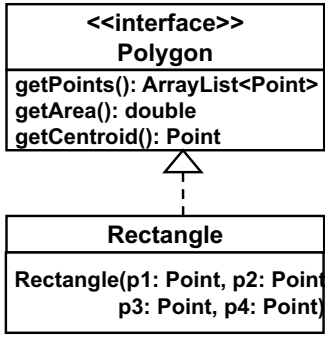
Inline Setup

```
import static org.junit.Assert.*;
import static org.hamcrest.CoreMatchers.*;
import org.junit.Test;

public class RectangleTest{
 @Test
 public void constructorWith4Points(){
   Rectangle cut = new Rectangle(
                       new Point(0,0),
                       new Point(2,0),
                       new Point(2,2),
                       new Point(0,2) );
   assertThat(cut.getPoints(), contains(...));  }

 @Test
 public void getSquareArea2By2(){
   Rectangle cut = new Rectangle(
                       new Point(0,0),
                       new Point(2,0),
                       new Point(2,2),
                       new Point(0,2) );
   assertThat(cut.getArea(), is(4)); }
}
```



Implicit Setup

```
import static org.junit.Assert.*;
import static org.hamcrest.CoreMatchers.*;
import org.junit.Test;
import org.junit.Before;
import org.junit.After;

public class RectangleTest{
 private Rectangle cut;

 @Before
 public void setUp(){
   cut = new Rectangle( new Point(0,0),
                        new Point(2,0),
                        new Point(2,2),
                        new Point(0,2) );  }

 @Test
 public void constructorWith4Points(){
   assertThat(cut.getPoints(), contains(...));  }

 @Test
 public void getSquareArea2By2(){
   assertThat(cut.getArea(), is(4)); }

 @After
 public void releaseResources(){...}  }
```



- Implicit setup makes a test class less redundant.
- Flow of execution
  - @Before setUp()
  - @Test constructorWith4Points()
  - @Before setUp()
  - @Test getSquareArea2By2()
  - The @Before method runs before every test method.
  - JUnit may run the test methods in an order different from their ordering in source code.

Continuous Unit Testing

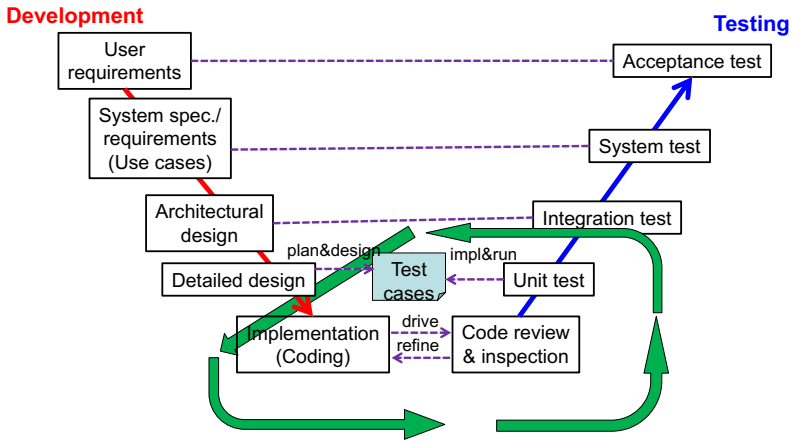
Benefits of Unit Tests

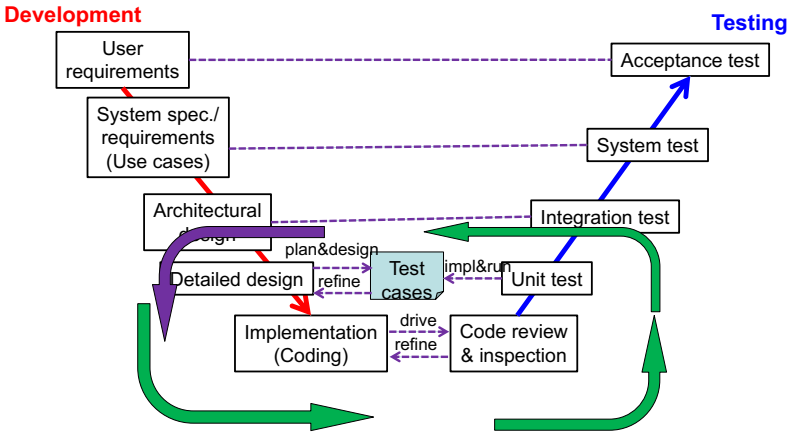
- Can test classes and their methods thoroughly.
  - Provide you a great confidence and in turn satisfaction.
- Can trigger/motivate design changes
  - You as a programmer are the first “user” of your own code.
  - If you feel your class/method is not easy to use, that encourages you to revise the current design.

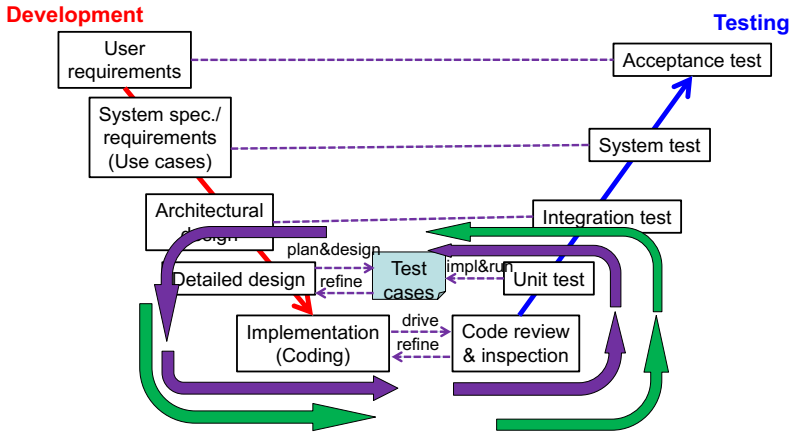
Continuous Unit Testing

- You as a programmer do it in a *continuous and step-by-step* manner.
  - as you write code and whenever you revise existing code
  - Code-test-code-test, rather than code-code-code-test
  - Test-code-test-code
    - "Test first": Test-driven development (TDD)
- Goal: Continuously make sure that your code works as expected and gain strong confidence and a peace of mind about your code.

Code-Test-Code-Test







- Test your code *early*, *automatically* and *repeatedly*.
  - To maximize the benefits of unit testing.
- Early testing
  - Do coding and unit testing at the same time.
- Automated testing
  - Run ALL test cases in an automated way.
    - Use an automated build tool (e.g. Ant). Never think of selecting and running test cases by hand.
- Repeated testing
  - Run ALL test cases whenever changes are made in the code base.

Benefits of Continuous Testing

- Can perform *regression testing* through continuous unit testing
  - Regression
    - A bug that emerges as a by-product in making changes in the code base
      - e.g., fixing a bug, adding new code to the code base, or revising existing code in the code base.
  - Regression testing
    - Uncovering regressions after changes are made in the code base
- *Immediate*ly giving feedback on regressions to development project members and fix them.
  - DO: Code → test → small regression fixes → test
  - DON'T: Code → code → code → test → big regression fixes
    - The amount of regressions (and the cost to fix them) can exponentially increase as time goes without continuous testing.

What to Do in Unit Testing?

- 4 tests (test types)
  - We 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.				

Coverage of Unit Tests

Code Coverage

- How much code is *executed* by test cases.
  - Higher coverage means/implies…
    - You have executed (~ tested) your code more thoroughly.
    - You have lower chances to have bugs in your code.
- Metrics to calculate coverage
  - Line coverage
    - Each line has been executed at least once?
  - Branch coverage
    - Each branch of each control structure (e.g. if, switch, try-catch structures) has been executed at least once?
  - Condition coverage
    - Each combination of true-false conditions has been executed at least once?

Example Coverage Calculation

- Class under test

```
public class Calculator{
 public int multiply(int x, int y){
   return x * y;
 }
}
```

- Test class

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

- Line coverage=100% (1/1)
- Branch coverage=100% (1/1)

- Class under test

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

- Test class

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

- Line coverage=66% (2/3)
- Branch coverage=50% (1/2)

- Class under test

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

- Test class

```
public class CalculatorTest{

@Test(expected=illegalArgumentException.class)
 public void divide5By0(){
   Calculator cut = new Calculator();
   cut.divide(5,0); }
}
```

- Line coverage=66% (2/3)
- Branch coverage=50% (1/2)

- Class under test

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

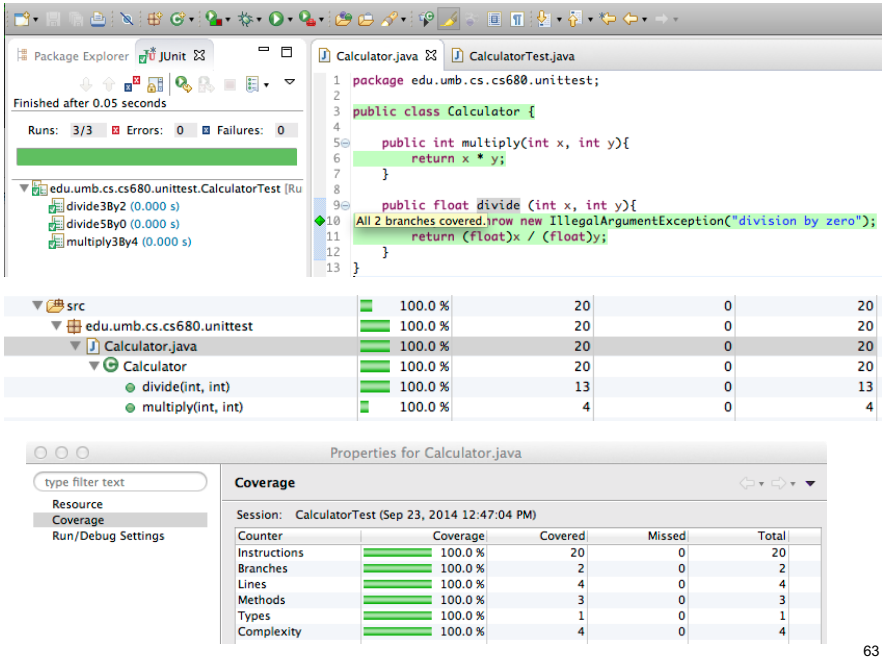
- Test class

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


@Test(expected=illegalArgumentException.class)
 public void divide5By0(){
   Calculator cut = new Calculator();
   cut.divide(5,0); }
}
```

- Line coverage=100% (3/3)
- Branch coverage=100% (2/2)

EclEmma: A Code Coverage Tool

- A code coverage tool for Eclipse
  - http://eclemma.org/
- Can examine how much code JUnit test cases cover/execute.
- Metrics
  - Line coverage
  - Instruction coverage
  - Branch coverage
  - Method coverage
    - How many methods are executed at least once per class.
    - Useful to find which methods are not tested yet.
  - Type coverage
    - How many classes are executed with 100% method coverage.
    - Useful to find which classes are not fully tested yet.



- Integration with Ant
  - Use a coverage measurement engine, JaCoCo, which is a part of EclEmma
    - http://www.eclemma.org/jacoco/
  - Jacoco provides Ant tasks
    - e.g., <coverage> and <report>
    - http://www.eclemma.org/jacoco/trunk/doc/integrations.html

How to do Code Coverage?

- Rule of thumb: Keep maintaining a reasonably high coverage
  - Need to seek 100% coverage in all metrics? No.
    - ~100% for the method and class coverage metrics
    - 80-90% in the line and branch coverage metrics
    - Depends on the nature of a project, the use of external libraries (e.g., Swing and DBs), etc.
      - c.f. DBUnit
  - You as a programmer is responsible for that.
    - How often?
      - Whenever code is written/revised, ideally.
      - Everyday, once a week, twice a week, etc.
      - When the coverage goes below a threshold.
    - Coverage can decrease very fast.
    - It can be time-consuming to recover it.

Is Coverage Maintenance Effective for Quality Assurance?

- Yes, as far as you have "good" test cases.
  - This test case can yield 100% method coverage for multiply(), but it doesn't actually test anything.

```
Calculator cut = new Calculator();
int expected = 12;
int actual = cut.multiply(3,4);
//assertThat(actual, is(expected));
```

- Note: 100% coverage does NOT mean bug-free.
  - It simply means that test cases have run your code thoroughly.
  - It's not a quality indicator.
- Your goal is not reaching the coverage of 100%.

Some Notes

- Utility class
  - Provide a series of utility methods.
    - e.g., java.lang.Math, java.util.Collections
  - Not intended to be instantiated.

```
final public class SomeUtils{
  private SomeUtils(){}
  public static String aUsefulUtilMethod(int n){
    ...
  } }
```

  - The private constructor is defined to prevent a Java compiler from implicitly inserting a public constructor when no constructors are explicitly defined.
  - No test cases can call it. Coverage decreases.
  - Forget about it.
    - There are some tricks to call it from a test case, but it wouldn't be worth doing that.

- Some exceptions may rarely occur.
  - e.g. IOExcepetion for file I/O operations
  - Test cases may not be able to reproduce all error cases to throw all exceptions. Coverage decreases.
  - Forget about it.
- Branching may be decided at random.

```
If( Math.random() >= 0.5 ){ do this }else{ do that }
```

  - Both branches may not be covered by running a test case twice.
  - It may be possible to cover all branches by repeating the test case multiple times, but…

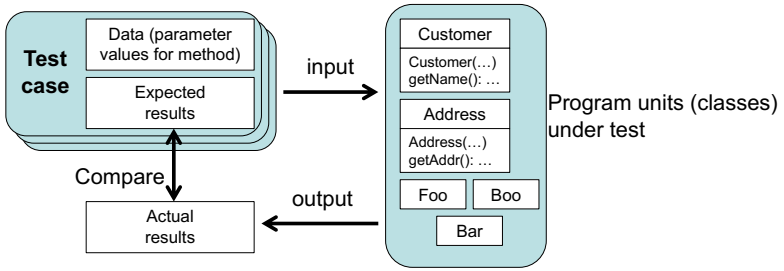
What to Do in Unit Testing?

- 4 tests (test types)
  - CS680 focuses 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**
Code rev&insp.				

Functional Test in Unit Testing

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

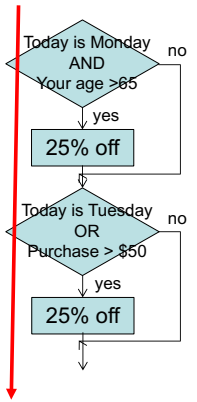


Structural Test in Unit Testing

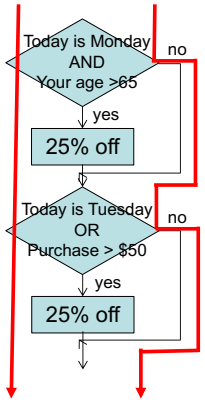
- Verify the structure of each class.
- Revise the structure, if necessary, to improve maintainability, flexibility and extensibility.
  - White-box test
- To-dos
  - Refactoring
  - Use of design pattern
  - Control flow test
  - Data flow test

Control Flow Test

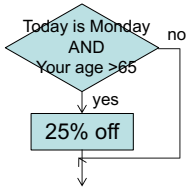
- Verify the flow of program execution
  - White-box test
- Need to decide the coverage metric to be used.
  - Line coverage
  - Branch coverage
  - Condition coverage
- To reach 100% line coverage, use a case where
  - Monday? Y, Age? > 65, Purchase > $50



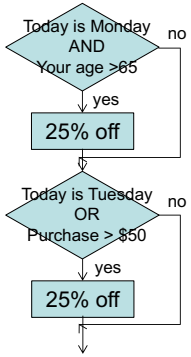
- To reach 100% branch coverage, use 2 cases
  - For example:
    - Monday? Y, Age > 65, Tue? N, Purchase > $50
    - Monday? N, Age > 65, Tue? Y, Purchase > $50



- Condition coverage
  - How many combinations of true-false conditions have been executed at least once?
  - EclEmma does not support it.
    - Need to manually keep track of it.

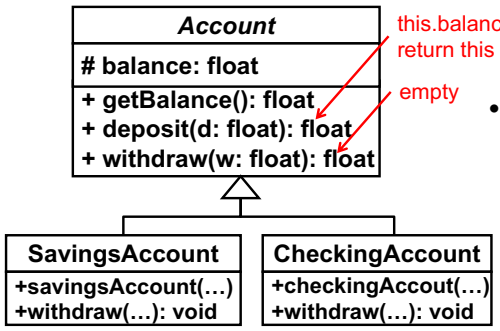


- Monday?, >65?
  - 4 true-false combinations
    - Y-Y, Y-N, N-Y, N-N
- Need 4 tests to reach 100% condition coverage
  - 4 tests may be in a single test case or 4 different test cases
- 2 tests required for branch coverage
  - Condition coverage requires more tests than branch coverage
    - Condition > branch > line



- Monday?, >65?
  - 4 true-false combinations (Y_1-Y_1, Y_1-N_1, N_1-Y_1, N_1-N_1)
- Tuesday?, >$50?
  - 4 true-false combinations (Y_2-Y_2, Y_2-N_2, N_2-Y_2, N_2-N_2)
- Need 8 tests to reach 100% condition coverage
  - Y_1-Y_1, Y_1-N_1, N_1-Y_1, N_1-N_1
  - Y_2-Y_2, Y_2-N_2, N_2-Y_2, N_2-N_2
- Just need 4 tests in fact.
  - Mon, >65, >$50: Y_1-Y_1, N_2-Y_2
  - Mon, >65, <=$50: Y_1-Y_1, N_2-N_2 (redundant)
  - Mon, <=65, >$50: Y_1-N_1, N_2-Y_2 (redundant)
  - Mon, <=65, <=$50: Y_1-N_1, N_2-N_2
  - Tue, >65, >$50: N_1-Y_1, Y_2-Y_2
  - Tue, >65, <=$50: N_1-Y_1, Y_2-N_2 (redundant)
  - Tue, <=65, >50: N_1-N_1, Y_2-Y_2 (redundant)
  - Tue, <=65, <=$50: N_1-N_1, Y_2-N_2

HW 4: Implement and Test This.



- SavingsAccount's withdraw()
  - If this.getBalance() >= w, withdraw the money.
  - if this.getBalance() < 0, throw an InsufficientFundsException.

- CheckingAccount's withdraw()
  - If this.getBalance() > w, withdraw the money.
  - If savingsAccount.getBalance() + this.getBalance() >= w, withdraw the money and charge a $50 penalty.
  - If savingsAccount.getBalance() + this.getBalance() < w, throw an InsufficientFundsException.
  - If this.getBalance() < 0, throw an InsufficientFundsException.

- Implement the class diagram.
  - It is not complete. You can complete it as you like.
  - Follow the specified rules to implement withdraw().
- Test all methods including constructors with JUnit.
  - You can use any naming convention for test method.
- Measure and report coverage with JaCoCo
  - Seek 100% coverage in all metrics.
  - Seek 100% condition coverage for withdraw().
- Have your Ant script to
  - compile Java code
  - invoke JUnit to run all test cases
  - invoke JaCoCo to generate a coverage report in HTML in the “test/reports/coverage” directory.

- Use <batchtest> to have Ant search test classes in your project directory and run all of them (RectangleTest and TriangleTest).
  - <junit …>

    …

    <batchtest …>

    ...

    </batchtest>

    ...

    </junit>

  - c.f. JUnit documentation

- [Top directory]
  - build.xml
  - src
    - edu/umb/cs/cs680/hw/SavingsAccount.java
  - bin
    - edu/umb/cs/cs680/hw/SavingsAccount.class
  - test
    - src
      - edu/umb/cs/cs680/hw/SavingsAccountTest.java
    - bin
      - edu/umb/cs/cs680/hw/SavingsAccountTest.class
    - reports
      - junit
        - » TEST-edu.umb.cs.cs680.hw.SavingsAccountTest.xml
        - » TESTS-TestSuites.xml
        - » HTML files
      - coverage
        - » Reports by JaCoCO

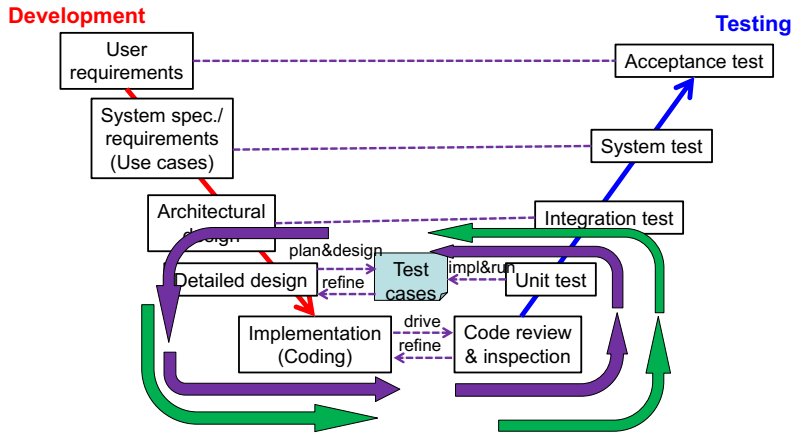
- Use <junit> to generate test reports in XML to test/reports/junit
  - TEST-…SavingsAccountTest.xml
- Use <junitreport> to
  - aggregate TEST-…xml files to a single XML file (TESTS-TestSuites.xml)
  - generate test reports in HTML to test/junit

- <mkdir dir="test/reports/junit" />

```
<junit … printsummary="yes">
   …
   <formatter type="xml"/>
   <batchtest … todir="test/reports/junit">
   ...
   </batchtest>
   ...
</junit>

<junitreport todir="test/reports/junit">
   ...
   <report todir="test/reports/junit"/>
</junitreport>
```

- **FIRM DUE:** November 7 (Tue) midnight

Code Review/Inspection



Code Inspection (Static Code Analysis)

- Analyzing code without actually executing it.
  - Dynamic analysis: analysis performed by executing code
- Static code analyzers
  - Can automatically detect particular (bad) code smells
  - Various tools available for various languages
    - http://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis

Static Code Analyzer: FindBugs

- Looks for bugs in Java programs based on bug patterns, which are code idioms that are often error.
  - http://findbugs.sourceforge.net/
  - http://findbugs.sourceforge.net/bugDescriptions.html
  - Can analyze code compiled for any version of Java, from 1.0 to 1.8
  - Can be used from Ant and Eclipse
  - Plug-ins are available for IntelliJ, Maven, Gradle, Jenkins, etc.

Static Code Analyzer: PMD

- Looks for potential problems such as:
  - Possible bugs
    - Empty try/catch/finally/switch statements
  - Dead code
    - Unused local variables, parameters and private methods
  - Suboptimal code
    - Wasteful String/StringBuffer usage
  - Overcomplicated expressions
    - Unnecessary if statements, for loops that could be while loops
  - Duplicate code
    - Copied/pasted code means copied/pasted bugs
  - http://pmd.sourceforge.net/
  - Requires JRE 1.6 or higher to run
  - Can be used from Ant
  - Plug-ins are available for Maven, Eclipse, etc.

Other Static Code Analyzers

- CheckStyle
  - Checks coding conventions
    - http://checkstyle.sourceforge.net/

FAQs

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.
- If you have many classes to test, you need to run main() in each of them.
- If one method fails, subsequent method calls are not executed.
  - `calc.divide(5, 0); // This call fails.`
    `calc.divide(10, 2);// This is not executed.`
- When you join a project, you may see a completely different testing practice with main(). Extra learning time/efforts. Few things are standardized.

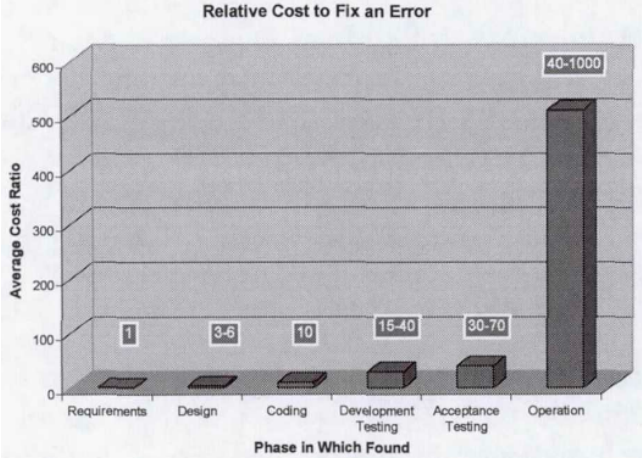
Why Not Just Use a Debugger for Testing?

- A debugger can be used for unit testing. However, it is designed for *manual* (or step by step) program execution.
  - i.e. for *manual* debugging and *manual* unit testing.
- JUnit (or any other unit testing frameworks) is designed for *automated* unit testing.

Costs of Unit Testing

- Unit testing requires programmers extra costs (time and efforts).
  - Not free. The costs are not that low
    - although various tools have dropped the costs dramatically.
  - Unit testing cost can be at least 2 times higher than coding cost.
    - The amount of unit testing code > the amount of production code
    - 2 to 5 times, in my experience
    - Costs get higher as you write more tests and more comprehensive tests.
      - Higher condition and branch coverage
      - More detailed boundary tests

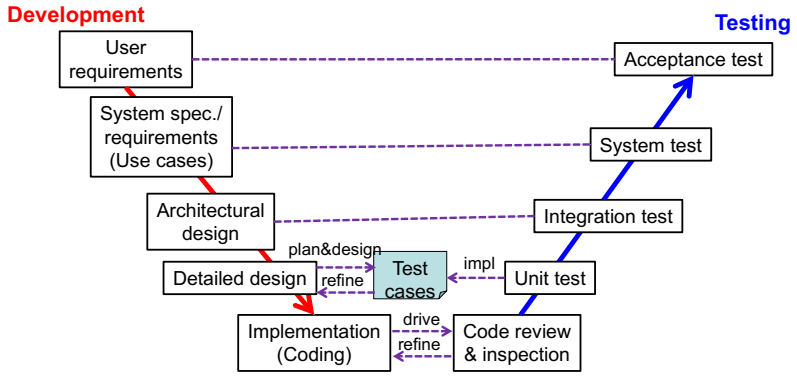
- Need to decide what to test.
  - e.g., Avoid unit tests for GUI and test GUI in a system test.
  - Skip unit tests for getter and setter methods.
- Unit testing costs can be paid off gradually throughout the product lifecycle
  - ONLY IF TESTS ARE EXECUTED REPEATEDLY AND AUTOMATEDLY
- Defect correction cost grows exponentially as product lifecycle goes.
  - Early defect corrections is much less expensive than late corrections.



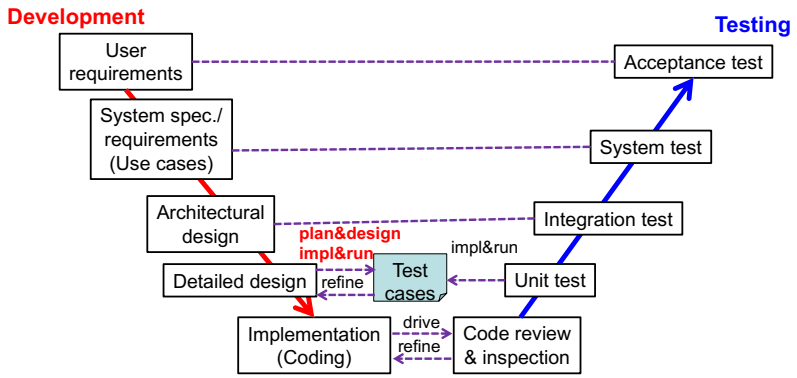
"Error Cost Escalation Through the Project Life Cycle" Stecklein et al., 2004
Software Engineering Economics, Boehm, Prentice-Hall, 1981.

Pushing "Early Testing" to "Test First"

- Early testing
  - Plan and design test cases while designing a class
  - Write and run test cases once the class is implemented.



- Test first development (TDD)
  - Write and run test cases
  - once the class is implemented.



- One of benefits of unit testing
  - Verify the ease of use for a class under test by using test cases as sample code
- This benefit can be maximized by TDD
  - You write sample code earlier than writing a class under test.

Suggested Reading

- Error Cost Escalation Through the Project Life Cycle
  - Stecklein et al., 2004
  - http://ntrs.nasa.gov/search.jsp?R=20100036670
- Software defect reduction Top 10 List
  - Boehm et al. 2001
  - https://www.cs.umd.edu/projects/SoftEng/ESEG/papers/82.78.pdf
- Understanding and Controlling Software Costs
  - Boehm et al., 1988
  - http://sunset.usc.edu/TECHRPTS/1986/usccse86-501/usccse86-501.pdf