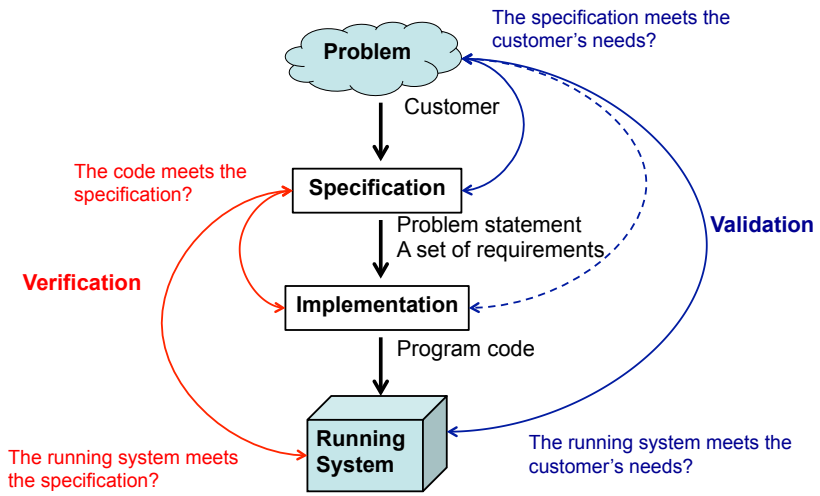
Software Testing: Verification and Validation

Verification and Validation (V&V)

- Verification
  - Testing whether a system is developed in accordance with its specification (a set of requirements).
    - Ensures you built it right.
- Validation
  - Testing whether a system meets the customer's needs.
    - Ensures you built the right thing.



- Defects found in verification
  - Occur when the implementation and/or running system fail to meet the specification.
    - The specification of a printer's firmware states that the printer stops printing when its paper tray is empty.
    - The printer doesn't stop printing when a tray is empty.
- Defects found in validation
  - Occur when the specification is wrong or misses the customer's needs.
    - The printer specification states that the printer can keep printing even when its tray is empty.
    - The specification fails to state that the printer stops printing when its tray is empty.
  - It is possible to define the specification correctly. However, it is hard (if not possible) to make the specification perfectly comprehensive.

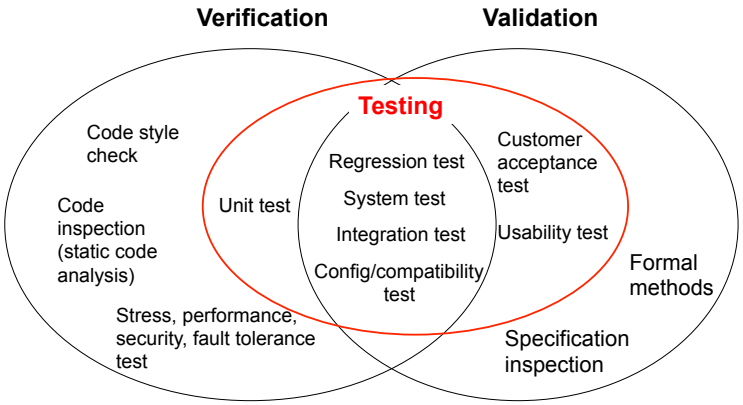
Example Defects in Validation

- Firmware for Boeing 787's generator control unit (GCU)
  - Does periodic "status check" every 10 milliseconds.
  - Implements a timer/counter (timestamp) with signed 32-bit integer.
    - 2^31=2,147,483,648
    - 10 msec * 2147483648 = 248.551 days
    - An integer overflow occurs once GCU operates for 248.551 days.
- GCUs fall into a failsafe mode if they are continuously powered on for 248 days.
  - A 787 aircraft has 4 GCUs.
  - If all of them are powered on at the same time, the aircraft can lose its control completely.

XXX-day Problems

- 248-day problem
- 494-day problem
  - Occurs if a counter/timer relies on an unsigned 32-bit integer
    - Server OSes, WiFi routers, network switches, etc. etc.
- 24-day and 49-day problems
  - Occur if a counter/timer relies on an signed 32-bit integer and its counting/timing resolution is 1 msec.
- 830-day problem
  - Occurs if a counter/timer relies on an unsigned 32-bit integer and its counting/timing resolution is 60 Hz (1/60 second; 16.67 msec)
- Year 2038 problem
  - Many OSes have a timer that counts time in second from 1970/1/1 0:00:00, using a signed integer. It will overflow at January 19 in 2018.

V/V Methods



White Box and Black Box Tests

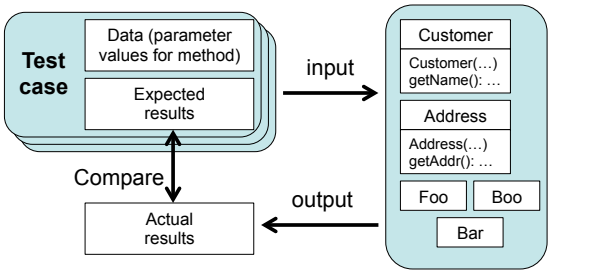
- White box testing
  - Testing a (software) system based on the knowledge about what are in the system.
    - e.g., How packages are organized, what classes and interfaces are defined in each package, what methods and data fields are defined in each class, etc.
  - Fine-grained testing
    - e.g., statement by statement, method by method
  - Two major types of white box tests
    - Control flow testing and data flow testing
  - The most basic (lowest-level) white box tests can be done through *unit testing*.
    - Higher-level white box testing: *integration* testing and *regression* testing

- Blackbox texting
  - Testing a system *without* knowing what are in the system.
  - Coarse-grained testing
    - Testing a system's external behaviors
  - Examples
    - Unit testing
      - Some blackbox tests can be done through unit testing.
    - Security testing, usability testing, fault tolerance testing
    - Stress testing, performance testing
    - Configuration testing, compatibility testing

Unit Testing

Unit Testing

- Verify that each *program "unit"* works as it is intended and expected along with given requirements.
  - Units to be tested: each class/interface and its methods
  - Test cases are often written *as programs* these days.



Who does it?

- You as a programmer do it.
- Programmers and unit testers are no longer separated in most (both large-scale and small-scale) projects as
  - it has been a lot easier and less time-consuming to write and run unit tests.
  - programmers can write the best test cases for their own code in the least amount of time.

Continuous (Unit) Testing

- You as a programmer do it *continuously* (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 intended and gain a peace of mind about your code

- Test your code *early*, *automatically* and *repeatedly*.
  - To maximize the benefits of unit testing.
- Early testing
  - You as a programmer do coding and unit testing at the same time.
- Automated testing
  - Run ALL test cases in an automated way.
    - 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., 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
  - Seamlessly integrate unit testing and regression testing
- *Immediate*ly giving feedback on regressions to development project members and fix them.
  - DO: Code → unit test → small regression fixes → unit test
  - DON'T: Code → code → code → big regression fixes
    - The amount of regressions (and the cost to fix them) can exponentially increase as time goes without continuous testing.

Unit Testing with JUnit

JUnit

- A unit testing framework for Java
  - Defines the format of a test case
    - Test case
      - Is a procedure to verify a particular feature(s)/behavior(s) with a set of inputs/conditions and expected results.
      - Describes how to perform a particular test.
  - Provides APIs to write test cases
  - Runs a set of test cases (a test suite)
  - Reports test results
- Making unit testing as easy and automatic as possible.
- Version 4.x, http://junit.org/
- Integration with Ant and Eclipse (and other IDEs)
  - <junit> task for Ant

Test Classes and Methods in JUnit

- 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
- Test method
  - A public method in a test class.
    - No parameters
    - Void return type
    - Can have a "throws" clause
  - Annotated with @Test
    - org.junit.Test
  - One test method implements one test case.

An Example

- 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); }
}
```


Static Import

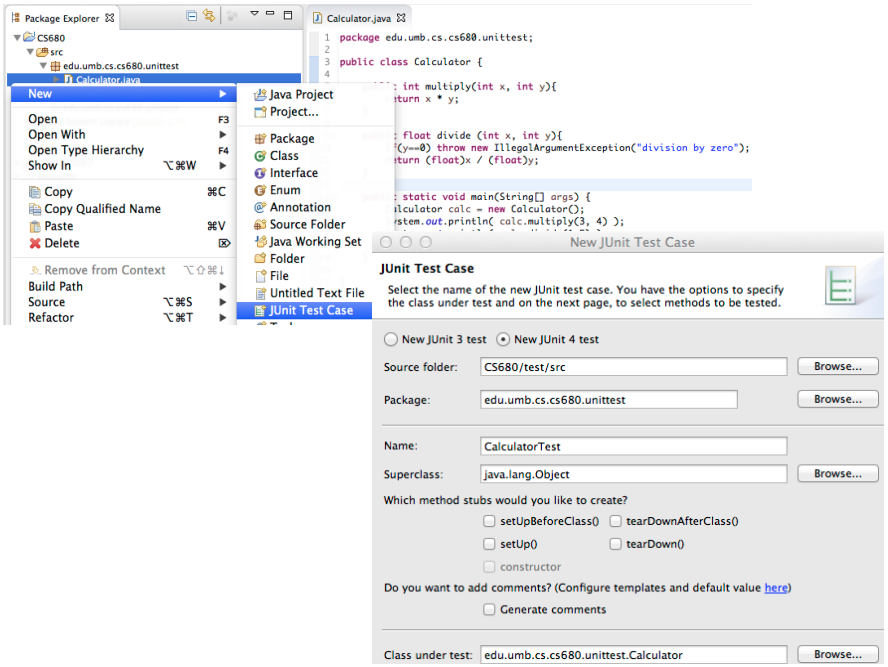
- `Assert` and `CoreMatchers` are typically referenced through *static import*.
  - With static import
    - » `assertThat(actual, is(expected);`
      - » "assert that actual is expected"
  - With normal import
    - » `Assert.assertThat(actual, CoreMatchers.is(expected));`

Principles in Unit Testing

- Define one or more fine-grained concrete/specific 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”)
- Use specific values and conditions, and detect design and coding errors.
  - Be detail-oriented. The devil resides in the details!
- Write simple, short, easy to understand test code.
  - Try to write many simple test cases, rather than a fewer number of complicated ones
    - Avoid a test case that perform multiple tasks.
    - You won’t feel bothered/overwhelmed by the number of test cases as far as they have intuitive names.
- No need to worry about redundancy in/among test methods.

Test Suite in JUnit

- A set of test classes
  - ~/code/projectX/ [project directory]
    - build.xml
    - src [source code directory]
      - edu/umb/cs/cs680/Foo.java
      - edu/umb/cs/cs680/Boo.java
    - bin [byte code directory]
      - edu/umb/cs/cs680/Foo.class
      - edu/umb/cs/cs680/Boo.class
    - test [a test suite; test classes]
      - src
        - » edu/umb/cs/cs680/FooTest.java
        - » edu/umb/cs/cs680/BooTest.java
      - bin
        - » edu/umb/cs/cs680/FooTest.class
        - » edu/umb/cs/cs680/BooTest.class



Things to Test

- Methods
- Exceptions
- Constructors

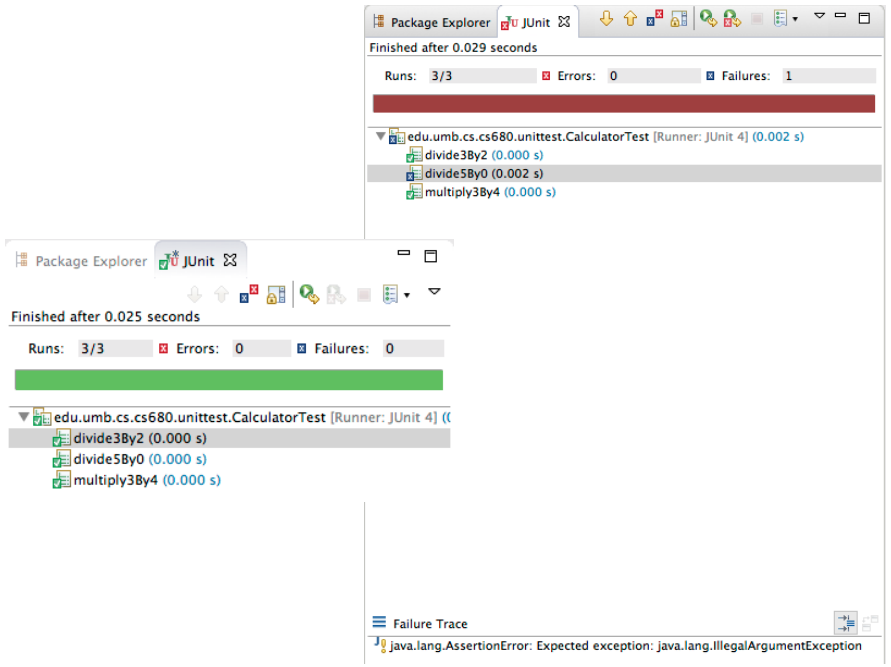
```
import static org.junit.Assert.*;
import static org.hamcrest.CoreMatchers.*;
import org.junit.Test;

public class StudentTest{
  @Test
  public void constructorWithName(){
    Student cut = new Student("John");
    assertThat(cut.getName(), is("John"));
    assertThat(cut.getAge(), is(nullValue()));
    assertThat(cut.getEmailAddr(), is(nullValue()));
  }
 @Test
  public void constructorWithoutName(){
    Student cut = new Student();
    ...
}
```

Test Runners

- How to run test classes?
  - From command line
    - `java org.junit.runner.JUnitCore edu.umb.cs.cs680.CalculatorTest`
    - `java org.junit.runner.JUnitCore edu.umb.cs.cs680.FooTest, edu.umb.cs.cs680.BooTest`
  - From IDEs
    - Eclipse, etc.
  - From Ant
    - <junit> task
- Test runners
  - org.junit.runners.JUnit4 (default runner)
  - org.junit.runners.Suite
    - Suite class

```
@RunWith(Suite.class)
@SuiteClasses( {FooTest.class, BarTest.class} )
public class AllTests{}
```

    - `java org.junit.runner.JUnitCore test.AllTests`
    - Can define a suite of suites.



Key Annotations and APIs

- Annotations
  - @Test
    - org.junit.Test
  - @Ignore
    - org.junit.Ignore
    - No need to comment out the entire test method.
- APIs
  - org.junit.Assert
    - Tests if an assertion holds
  - org.hamcrest.CoreMatchers
    - Provides a series of *matchers*, each of which performs a particular matching logic.

Key APIs: Assert

- `org.junit.Assert`
  - Contains a series of *static* "assertion methods."
    - » `assertThat( Object, org.hamcrest.Matcher )`
      - » Primitive-type value to be autoboxed.
      - » Just returns if two values (expected and actual values) match.
      - » Throws an AssertionError if two values do not match.
    - » `fail( java.lang.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.
  - Note: `assertEquals()` has been deprecated in JUnit version 4.
    - » It was a major assertion method in JUnit version 3.
    - » Use `assertThat()` instead.

- `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
  - Assert has some extra methods, but you don't 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
```

```
assertArrayEquals(new Person("Mickey","Mouse"),
                  new Person("Mickey","Mouse")); // FAIL
```

```
assertThat(new Person("Mickey","Mouse"),
           is(new Person("Mickey","Mouse"))); // FAIL
```

- Check if two Person instances are identical (i.e. if they have the same object ID)

Person
- firstName: String - lastName: String
Person(first:String last:String)

Key APIs: CoreMatchers

- `org.hamcrest.CoreMatchers`
  - Contains static methods, each returning a matcher object that performs matching logic.
  - `is()`
    - » `assertThat(actual, is(expected))`
    - » `assertThat(actual, is(nullValue()))`
    - » `assertThat(actual, is(notNullValue()))`
    - » `assertThat(actual, is(not(expected))`
    - » `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.

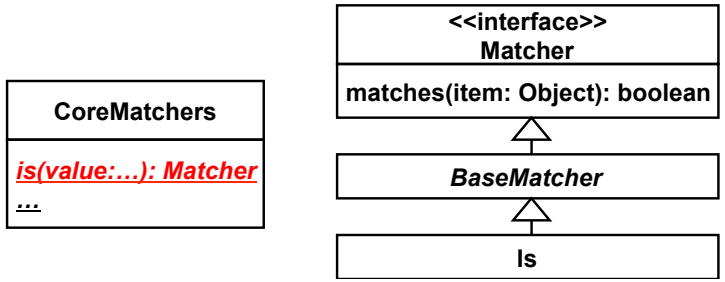
- `hasItem()`
  - » `ArrayList<String> actual = …;`
    `assertThat(actual, hasItem("Hello"));`
- `hasItems()`
  - » `assertThat(actual, hasItems("Hello", "World");`
- `everyItem()`
  - » `assertThat(actial, everyItem("Hello");`

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

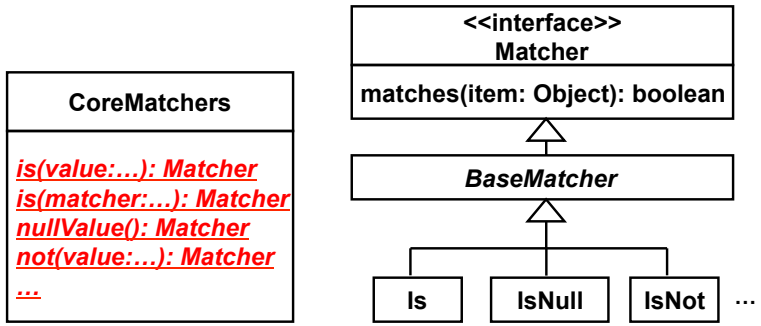
- It is important to learn what methods are available in `CoreMatchers` and what parameters the methods accept.
  - » c.f. Javadoc API documentation.

A Note on 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))`



Principles in Unit Testing

- Define one or more fine-grained concrete/specific test cases (test methods) for each method in a class under test.
- Give a concrete/specific and intuitive name to each test method.
- Use specific values and conditions, and detect design and coding errors.
  - Be detail-oriented. The devil resides in the details!
- Write simple, short, easy to understand test code.
- No need to worry about redundancy in/among test methods.

Principles in Unit Testing (cont'd)

- Write simple, short, easy to understand test cases
  - Try to write many simple test cases, rather than a fewer number of complicated test cases.
    - Avoid a test case that perform multiple tasks.
    - You won't feel bothered/overwhelmed by the number of test cases as far as they have intuitive names.
      - e.g. "divide5by4"

Extra Benefits of Unit Tests

- Besides you test classes and their methods…
- Can trigger/motivate design changes
  - You as a programmer can be 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.
- Can be useful as sample code to use your class/ method.
  - When you forgot how to use a class/method you implemented.
  - When you use a class/method that someone else implemented.

Continuous Unit Testing w/ Ant and JUnit

- Whenever you revise your code, you re-build your code base with Ant.
  - Revise test cases accordingly (if necessary)
  - Perform all test cases.
    - Code-test-code-test, rather than code-code-code-test
      - Continuous unit testing
- Ant: automated build tool
  - Maven: an alternative

HW 6-1

- Write test cases for the code you wrote in HW2-2 (polygon example)
  - Write at least one test case for every single method.
- Turn in build.xml, src and test/src for each.
  - build.xml should build all source code, run all text cases and run your app automatically.

HW 6-2

- Write test cases for the code you wrote in HW and HW3-1 (icecream examples)
  - Write at least one test case for every single method.
- Turn in build.xml, src and test/src for each.
  - build.xml should build all source code, run all text cases and run your app automatically.

- Use relative paths in build.xml.
- You can assume junit.jar and hamcrest-core.jar are specified in my CLASSPATH environment variable.