IT4490 – SOFTWARE DESIGN AND CONSTRUCTION

12. UNIT TEST

Content

Testing

- "[T]he means by which the presence, quality, or genuineness of anything is determined; a means of trial." – dictionary.com
- A *software test* executes a program to determine whether a property of the program holds or doesn't hold
- A test ***passes*** [***fails***] if the property **holds** [**doesn't hold**] on that run

Why?

- Why testing?
  - Improve software design
  - Make software easier to understand
  - Reduce debugging time
  - Catch integration errors
- In short, to Produce Better Code
- Preconditions
  - Working code
  - Good set of unit tests

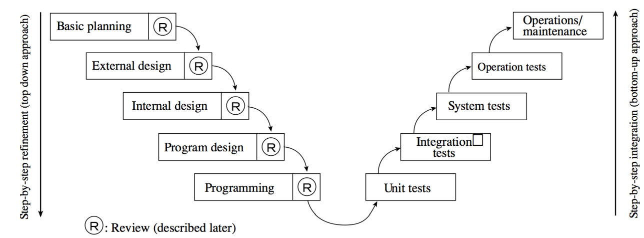
Software Quality Assurance (QA)
Testing plus other activities including

- Static analysis (assessing code without executing it)
- Proofs of correctness (theorems about program properties)
- Code reviews (people reviewing others' code)
- Software process (placing structure on the development lifecycle)
- …and many more ways to find problems and to increase confidence

No single activity or approach can guarantee software quality

V Model – Different test level

- Unit test: ONE module at a time
- Integration test: The linking modules
- System test: The whole (entire) system



Test levels

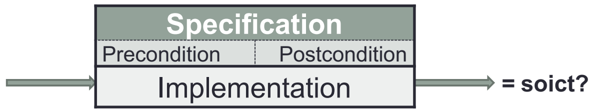
- Unit Testing: Does each unit (class, method, etc.) do what it supposed to do?
  - Smallest programming units
  - Strategies: Black box and white box testing
  - Techniques, Tools
- Integration Testing: do you get the expected results when the parts are put together?
  - Strategies: Bottom-up, top-down testing
- System Testing: does it work within the overall system?
- Acceptance Testing: does it match to user needs?

Content

1. Testing overview
2. **Unit Test**
3. Integration Test

2.1. Unit test strategies
Black box and White box testing

A. Choose input data ("test inputs")
B. Define the expected outcome ("soict")
C. Run the unit ("SUT" or "software under test") on the input and record the results
D. Examine results against the expected outcome ("soict")



Black box	White box
Must choose inputs *without knowledge* of the implementation	Can choose inputs *with knowledge* of the implementation

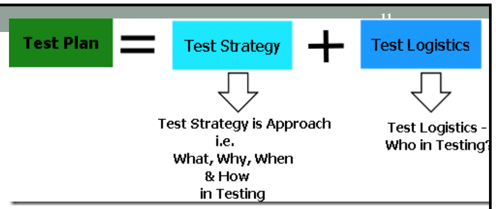
It's not black-and-white, but…

Black box	White box
Must choose inputs *without knowledge* of the implementation	Can choose inputs *with knowledge* of the implementation

- Has to focus on the behavior of the SUT
- Needs an "soict"
  - Or at least an **expectation** of whether or not an exception is thrown

- Common use: *coverage*
- Basic idea: if your test suite never causes a statement to be executed, then that statement might be buggy

Terms



- Test case
  - a set of conditions/variables to determine whether a system under test satisfies requirements or works correctly
- Test suite
  - a collection of test cases related to the same test work
- Test plan
  - a document which describes testing approach and methodologies being used for testing the project, risks, scope of testing, specific tools

Test Case Example1 (simple test)

Test Case #: 2.2 **Test Case Name:** Change PIN **Page:** 1 of 1
System: ATM **Subsystem:** PIN
Designed by: ABC **Design Date:** 28/11/2004
Executed by: **Execution Date:**
Short Description: Test the ATM Change PIN service

Pre-conditions
The user has a valid ATM card - The user has accessed the ATM by placing his ATM card in the machine
The current PIN is 1234
The system displays the main menu

Step	Action	Expected System Response	Pass/ Fail	Comment
1	Click the 'Change PIN' button	The system displays a message asking the user to enter the new PIN		
2	Enter '5555'	The system displays a message asking the user to confirm (re-enter) the new PIN		
3	Re-enter '5555'	The system displays a message of successful operation The system asks the user if he wants to perform other operations		
4	Click 'YES' button	The system displays the main menu		
5	**Check post-condition 1**			

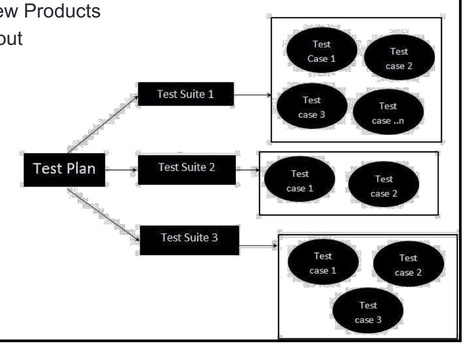
Post-conditions
1. The new PIN '5555' is saved in the database

Good test case design

- An good test case satisfies the following criteria:
  - Reasonable probability of catching an error
  - Does interesting things
  - Doesn't do unnecessary things
  - Neither too simple nor too complex
  - Not redundant with other tests
  - Makes failures obvious
  - Mutually Exclusive, Collectively Exhaustive

Test suite

- Example of test suite
  - Test case 1: Login
  - Test case 2: Add New Products
  - Test case 3: Checkout
  - Test case 4: Logout



Unit Testing techniques

- For test case design
- (2.2) Test Techniques for Black Box Test
  - Equivalence Partitioning Analysis
  - Boundary-value Analysis
  - Decision Table
  - Use Case-based Test
- (2.3) Test Techniques for White Box Test
  - Control Flow Test with C0, C1 coverage
  - Sequence chart coverage test

2.2.1. Equivalence Partitioning

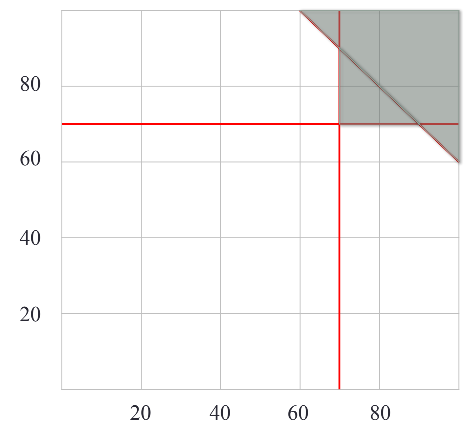
- Create the encompassing test cases by analyzing the input data space and dividing into equivalence classes
  - Input condition space is partitioned into equivalence classes
  - Every input taken from a equivalence class produces the same result

Example: Examination Judgment Program

- Program Title: “Examination Judgment Program”
- Subject: Two subjects as Mathematics, and Physics Judgment
- Specification:
  - Passed if
    - scores of both mathematics and physics are greater than or equal to 70 out of 100

      or,
    - average of mathematics and physics is greater than or equal to 80 out of 100
  - Failed => Otherwise

Equivalence Partitioning of Input space and test cases



Analysis and discussions

- We tried to create encompassing test cases based on external specification.
  - Successful? “Yes” !
- Next question. The test cases/data are fully effective?
  - We have to focus on the place in which many defects are there, don’t we?
  - Where is the place ?

→ “Boundary-value analysis”

2.2.2. Boundary-value analysis

- Extract test data to be expected by analyzing boundary input values => Effective test data
  - Boundary values can detect many defects effectively

→E.g. mathematics/physics score is 69 and 70

- The programmer has described the following wrong code:

```
if (mathscore > 70) {
    ...................
}
```

- Instead of the following correct code;

```
if (mathscore >= 70) {
    ...................
}
```

Example: Boundary-value analysis

- Boundary values of the mathematics score of small case study:
- What about the boundary value analysis for the average of mathematics and physics?

2.2.3. Decision Table

- Relations between the conditions for and the contents of the processing are expressed in the form of a table
- A decision table is a tabular form tool used when complex conditions are combined

Example: Decision Table

- The conditions for creating reports from employee files

Under age 30	Y	Y	N	N
Male	Y	N	Y	N
Married	N	Y	Y	N
Output Report 1	-	X	-	-
Output Report 2	-	-	-	X
Output Report 3	X	-	-	-
Output Report 4	-	-	X	-

Decision Table for “Examination Judgement”

- Condition1: Mathematics score >= 70
- Condition2: Physics score >= 70
- Condition3: Average of Mathematics, and Physics >= 80

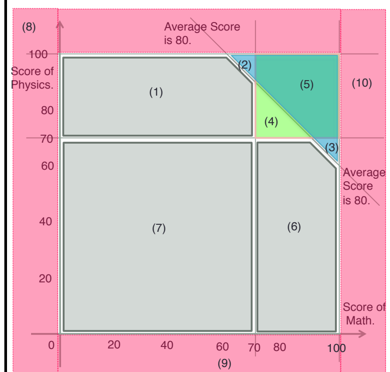
Decision Table for “Examination Judgement”

- Invalid input data (integer)
  - Condition1: Mathematics score = valid that means “0=< the score =< 100”
  - Condition2: Physics score = valid that means “0=< the score =< 100”

	TCI1	TCI2	TCI3	TCI4
Condition1	Valid	Invalid	Valid	Invalid
Condition2	Valid	Valid	Invalid	Invalid
“Normal results”	Yes	---	---	---
“Error message math”	---	Yes	---	Yes
“Error message phys”	---	---	Yes	Yes

If both of mathematics score and physics score are invalid, two messages are expected to be output. Is it correct specifications? Please confirm it?

Decision Table for “Examination Judgement”???



Test Data from Equivalence Analysis

Score	Math.	Physics	Result
(1)	55	85	Failed
(2)	67	97	Passed
(3)	96	68	Passed
(4)	77	80	Passed
(5)	85	92	Passed
(6)	79	58	Failed
(7)	52	58	Failed
(8)	15	120	Invalid
(9)	68	-66	Invalid
(10)	118	85	Invalid
(11)	-18	80	Invalid

2.2.4. Use case Testing

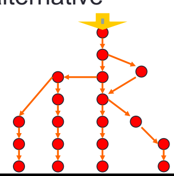
- helps us identify test cases that exercise the whole system on a transaction by transaction basis from start to finish.
- Use cases are defined in terms of the end user and not the system, use case describe what the user does and what the user sees rather than what inputs the software system expects and what the system outputs.
- Each usecase must specify any preconditions that need to be met for the use case to work. Use cases must also specify post conditions that are observable results and a description of the final state of the system after the use case has been executed successfully.

Test cases for Use case “Log in”?

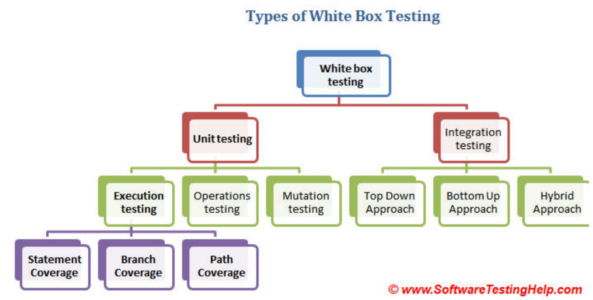
	Step	Description
Main Success Scenario **A: Actor** **S: System**	1	A: Inserts card
	2	S: Validates card and asks for PIN
	3	A: Enters PIN
	4	S: Validates PIN
	5	S: Allows access to account
Extensions	2a	Card not valid S: Display message and reject card
	4a	PIN not valid S: Display message and ask for re-try (twice)
	4b	PIN invalid 3 times S: Eat card and exit

Creating test cases from use cases

- Identify all of the scenarios for the given use case
- Alternative scenarios should be drawn in a graph fo each action
- Create scenarios for
  - a basic flow,
  - one scenario covering each alternative flow,
  - and some reasonable combinations of alternative flows
- Create infinite loops



2.3. Techniques for White Box testing



2.3. Techniques for White Box testing

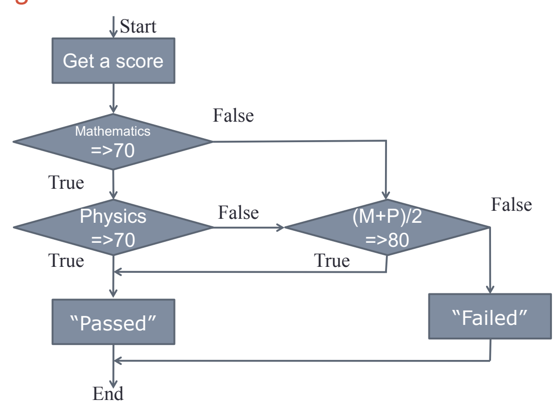
- Test cases should cover all processing structure in code

=> Typical test coverage
  - C0 measure: Executed statements # / all statements #
    - C0 measure at 100% means "all statements are executed".
  - C1 measure: Branches passed # / all blanches #
    - C1 measure at 100% means "all blanches are executed"

=> Prevent statements/blanches from being left as non-tested parts

=> Cannot detect functions which aren't implemented

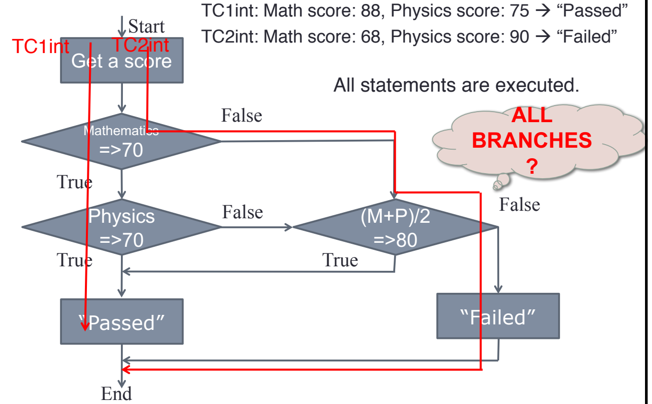
E.g. Control flow test of "Examination Judgment Program"



E.g. Control flow test of "Examination Judgment Program" – 100% C0 coverage

TC1int: Math score: 88, Physics score: 75 → "Passed"
TC2int: Math score: 68, Physics score: 90 → "Failed"

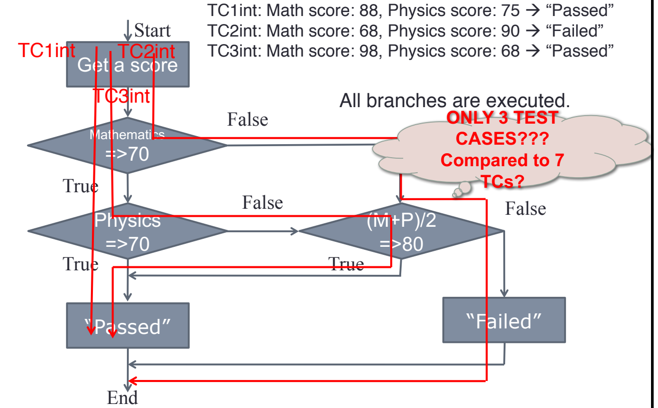
All statements are executed.



E.g. Control flow test of "Examination Judgment Program" – 100% C1 coverage

TC1int: Math score: 88, Physics score: 75 → "Passed"
TC2int: Math score: 68, Physics score: 90 → "Failed"
TC3int: Math score: 98, Physics score: 68 → "Passed"

All branches are executed.



Decision Table for "Examination Judgement"

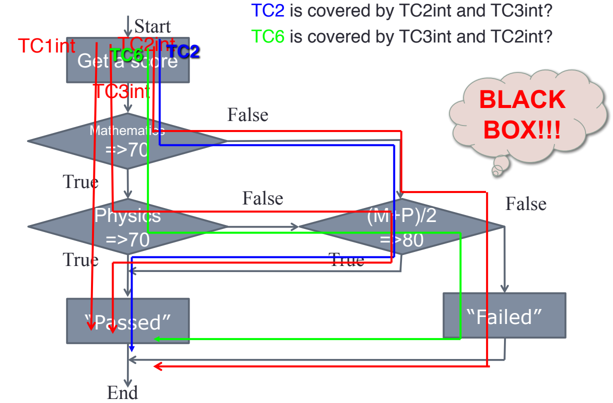
Condition1: Mathematics score >= 70
Condition2: Physics score >= 70
Condition3: Average of Mathematics, and Physics >= 80

	TC5	TC4	TC3	TC6	TC2	TC1	TCNG	TC7
Condition1	True	True	True	True	False	False	False	False
Condition2	True	True	False	False	True	True	False	False
Condition3	True	False	True	False	True	False	True(none)	False
"Passed"	Yes	Yes	Yes	---	Yes	---	N/A	--
"Failed"	---	---	---	Yes	---	Yes	N/A	Yes

- One TCxint can cover plural TCs, based on the correct control flow structure
  - TC1int covers TC5 and TC4
  - TC2int covers TC1 and TC7
  - TC3int covers to TC3.
- TC2 and TC6 are left in no execution

E.g. Control flow test of "Examination Judgment Program" – 100% C1 coverage

TC2 is covered by TC2int and TC3int?
TC6 is covered by TC3int and TC2int?



How to test a loop structure program

- For the control flow testing in the software including a loop, the following criteria are usually adopted instead of C0/C1 coverage measures.
  - Skip the loop.
  - One and two passes through the loop.
  - Typical times m passes through the loop
  - n, n-1, n+1 passes through the loop
    - n is maximum number, m is typical number (m<n)
- Example: 6 cases based on boundary-value analysis:

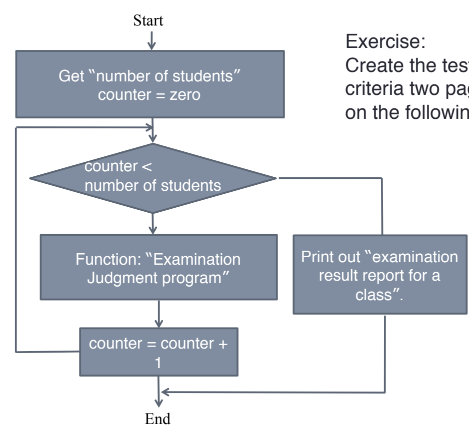
Example

```
class LoopTestExampleApp {
   public  static BufferedReader keyboardInput =
            new BufferedReader(new InputStreamReader(System.in));
   private static final int MINIMUM = 1;
   private static final int MAXIMUM = 10;

   public static void main(String[] args) throws IOException {
            System.out.println("Input an integer value:");
            int input = new Integer(keyboardInput.readLine()).intValue();

            int numberOfIterations=0;
            for(int index=input;index >= MINIMUM && index <= MAXIMUM;index++) {
                        numberOfIterations++;
            }

            System.out.println("Number of iterations = " + numberOfIterations);
   }
}
```


Examples for "Examination Judgment Program"

- Input two subjects scores, Mathematics and Physics, for each member of one class. The input form is "tabular form". Class members can be allowed only 0 (zero) through 50.
- Output/Print out the "Examination result report for a class". The output form is also "tabular form" that has the columns such as student name, scores (Math., Physics), passed or failed.

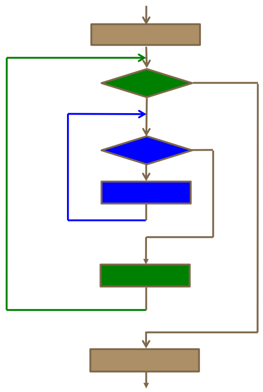
Examples for "Examination Judgment Program"



Exercise:
Create the test cases using the criteria two pages before based on the following assumptions.

1. "Examination Judgment program" are already tested.
2. Input data of this module are already checked, and valid.

How to test for nested loops structure



At first, first loop's control number is determined at typical number, and second loop is tested as a simple loop.

Next, second loop's control number is determined at typical number, and first loop is tested as a simple loop.

2.4. Combination of Black/White Box test

- Advantage of Black box
  - Encompassing test based on external specification
  - Very powerful and fundamental to develop high-quality software
- Advantage of White box
  - If any paths/flows don't appear in the written specifications, the paths/flows might be missed in the encompassing tests => White box test
    - for data of more than two years before => alternative paths
    - "0 =< score =< 100" => code: "if 0 =< score " and "if score =< 100"

How to carry out efficient and sufficient test

- First, carry out tests based on the external specifications
  - If all test cases are successful

  => All external specifications are correctly implemented
- Second, carry out tests based on the internal specifications
  - Add test cases to execute the remaining paths/flow, within external specifications
  - If all test cases are successful with coverage = 100%

  => All functions specified in the external specification are successfully implemented without any redundant codes

2.5. JUNIT

49

What is a testing framework?

- A test framework provides reusable test functionality which:
  - Is easier to use (e.g. don't have to write the same code for each class)
  - Is standardized and reusable
  - Provides a base for regression tests

50

Why use a testing framework?

- Each class must be tested when it is developed
- Each class needs a regression test
- Regression tests need to have standard interfaces
- Thus, we can build the regression test when building the class and have a better, more stable product for less work

51

Mannual testing vs. Automated testing

Manual Testing	Automated Testing
Executing a test cases manually without any tool support is known as manual testing.	Taking tool support and executing the test cases by using an automation tool is known as automation testing.
Time-consuming and tedious − Since test cases are executed by human resources, it is very slow and tedious.	**Fast** − Automation runs test cases significantly faster than human resources.
Huge investment in human resources − As test cases need to be executed manually, more testers are required in manual testing.	**Less investment in human resources** − Test cases are executed using automation tools, so less number of testers are required in automation testing.
Less reliable − Manual testing is less reliable, as it has to account for human errors.	**More reliable** − Automation tests are precise and reliable.
Non-programmable − No programming can be done to write sophisticated tests to fetch hidden information.	**Programmable** − Testers can program sophisticated tests to bring out hidden information.

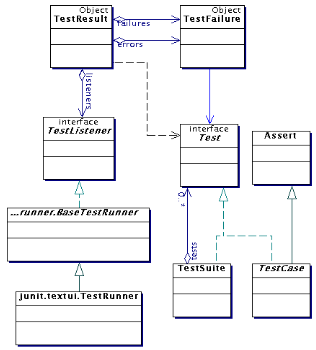
52

JUnit

- JUnit is a unit testing framework for Java programming language (junit.org)
- Junit test generators now part of many Java IDEs (Eclipse, BlueJ, Jbuilder, DrJava)
- Xunit tools have since been developed for many other languages (Perl, C++, Python, Visual Basic, C#, …)

Architectural overview

- JUnit test framework is a package of classes that lets you write tests for each method, then easily run those tests
- **Test Runner** runs tests and reports **TestResult**s
- You test your class by extending abstract class ***TestCase***
- To write test cases, you need to know and understand the **Assert** class



JUnit 3 – TestCase example (1/2)

```
public class Counter {
  int count = 0;

  public int increment() {
          return ++count;
  }

  public int decrement() {
          return --count;
  }
  public int getCount() {
          return count;
  }
}
```

JUnit 3 – TestCase example (2/2)

```
public class CounterTest extends junit.framework.TestCase {
    Counter counter1;
    public CounterTest() { }   // default constructor

    protected void setUp() {   // creates a (simple) test fixture
        counter1 = new Counter();
    }

    public void testIncrement() {
        assertTrue(counter1.increment() == 1);
        assertTrue(counter1.increment() == 2);
     }

    public void testDecrement() {
        assertTrue(counter1.decrement() == -1);
    }

    protected void tearDown() { }
}
```

Note that each test begins with a *brand new* counter

This means you don't have to worry about the order in which the tests are run

Why create a test suite?

- Obviously you have to test your code—right?
  - You can do ad hoc testing (running whatever tests occur to you at the moment), or
  - You can build a test suite (a thorough set of tests that can be run at any time)
- Disadvantages of a test suite
  - It's a lot of extra programming
    - True, but use of a good test framework can help quite a bit
  - You don't have time to do all that extra work
    - False! Experiments repeatedly show that test suites reduce debugging time more than the amount spent building the test suite
- Advantages of a test suite
  - Reduces total number of bugs in delivered code
  - Makes code much more maintainable and refactorable

JUnit 3 – TestSuite

- TestSuites collect a selection of tests to run them as a unit
- Collections automatically use TestSuites, however to specify the order in which tests are run, write your own:

```
public class TestSuiteExample {
    public static Test suite() {
            TestSuite suite = new TestSuite();
            suite.addTest(new CounterTest("testIncrement"));
            suite.addTest(new CounterTest("testDecrement"));
            return suite;
    }
}
```

- Can create TestSuites that test a whole package:

```
    public static Test suite() {
            TestSuite suite = new TestSuite();
            suite.addTestSuite(CounterTest.class);
            suite.addTestSuite(TestXXX.class);
            return suite;
    }
```


JUnit 3 – Run a TestSuite

```
public class TestRunner {
  public static void main(String[] args) {
    Result result = JUnitCore.runClasses(TestSuiteExample.class);

    for (Failure failure : result.getFailures()) {
      System.out.println(failure.toString());
    }

    System.out.println(result.wasSuccessful());
  }
}
```


JUnit 4 – Basic example (1/4)

```
public class MessageUtil {

  private String message;

  //Constructor
  //@param message to be printed

  public MessageUtil(String message){
    this.message = message;
  }

  // prints the message
  public String printMessage(){
    System.out.println(message);
    return message;
  }
}
```


JUnit 4 – Basic example (2/4)

```
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class TestJunit {

  String message = "Hello World";
  MessageUtil messageUtil = new MessageUtil(message);

  @Test
  public void testPrintMessage() {
    assertEquals(message, messageUtil.printMessage());
  }
}
```

JUnit 4 – Basic example (3/4)

```
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class TestRunner {
  public static void main(String[] args) {
    Result result = JUnitCore.runClasses(TestJunit.class);

    for (Failure failure : result.getFailures()) {
      System.out.println(failure.toString());
    }

    System.out.println(result.wasSuccessful());
  }
}
```

```
Hello World
true
```


JUnit 4 – Basic example (4/4)

```
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class TestJunit {

  String message = "Hello World";
  MessageUtil messageUtil = new MessageUtil(message);

  @Test
  public void testPrintMessage() {
    message = "New Word";
    assertEquals(message,messageUtil.printMessage());
  }
}
```

```
Hello World
testPrintMessage(TestJunit): expected:<[New Wor]d> but was:<[Hello Worl]d>
false
```


JUnit 4 – Another way to run the Testcase

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;

@RunWith(Suite.class)
@Suite.SuiteClasses({
  TestJunit.class,
  YYY.class
})
public class AllTests {
  // this class remains completely
  // empty, being used only as a
  // holder for the above
  // annotations
}
```


JUnit - Execution Procedure

```
import org.junit.After;
import org.junit.AfterClass;

import org.junit.Before;
import org.junit.BeforeClass;

import org.junit.Ignore;
import org.junit.Test;

public class ExecutionProcedureJunit {

  //execute only once, in the starting
  @BeforeClass
  public static void beforeClass() {
    System.out.println("in before class");
  }

  //execute only once, in the end
  @AfterClass
  public static void  afterClass() {
    System.out.println("in after class");
  }

  //execute for each test, before executing test
  @Before
  public void before() {
    System.out.println("in before");
  }

  //execute for each test, after executing test
  @After
  public void after() {
    System.out.println("in after");
  }

  //test case 1
  @Test
  public void testCase1() {
    System.out.println("in test case 1");
  }

  //test case 2
  @Test
  public void testCase2() {
    System.out.println("in test case 2");
  }
}
```

Assert methods

- Each assert method has parameters like these: *message, expected-value, actual-value*
- Assert methods dealing with floating point numbers get an additional argument, a tolerance
- Each assert method has an equivalent version that does not take a message – however, this use is not recommended because:
  - messages helps documents the tests
  - messages provide additional information when reading failure logs

Assert methods

- assertTrue(String *message*, Boolean *test*)
- assertFalse(String *message*, Boolean *test*)
- assertNull(String *message*, Object *object*)
- assertNotNull(String *message*, Object *object*)
- assertEquals(String *message*, Object *expected*, Object *actual*) (uses equals method)
- assertSame(String *message*, Object *expected*, Object *actual*) (uses == operator)
- assertNotSame(String *message*, Object *expected*, Object *actual*)

Assert methods

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

public class TestAssertions {

  @Test
  public void testAssertions() {
    //test data
    String str1 = new String ("abc");
    String str2 = new String ("abc");
    String str3 = null;
    String str4 = "abc";
    String str5 = "abc";

    int val1 = 5;
    int val2 = 6;

    String[] expectedArray = {"one", "two", "three"};
    String[] resultArray =  {"one", "two", "three"};

    //Check that two objects are equal
    assertEquals(str1, str2);

    //Check that a condition is true
    assertTrue (val1 < val2);

    //Check that a condition is false
    assertFalse(val1 > val2);

    //Check that an object isn't null
    assertNotNull(str1);

    //Check that an object is null
    assertNull(str3);

    //Check if two object references point to the same object
    assertSame(str4,str5);

    //Check if two object references not point to the same object
    assertNotSame(str1,str3);

    //Check whether two arrays are equal to each other.
    assertArrayEquals(expectedArray, resultArray);
  }
}
```


JUnit - Time Test

```
import org.junit.Test;

public class TimeoutTest {

   //This test will always failed :)
   @Test(timeout = 1000)
   public void infinity() {
      while (true) ;
   }

   //This test can't run more than 5 seconds, else failed
   @Test(timeout = 5000)
   public void testSlowMethod() {
      //...
   }

}
```

JUnit – Exception (1/2)

```
public class MessageUtil {
  private String message;
  public MessageUtil(String message){
    this.message = message;
  }
  public void printMessage(){
    System.out.println(message);
    int a = 0;
    int b = 1/a;
  }
  public String salutationMessage(){
    message = "Hi!" + message;
    System.out.println(message);
    return message;
  }
}
```


JUnit – Exception (2/2)

```
import org.junit.Test;
import org.junit.Ignore;
import static org.junit.Assert.assertEquals;

public class TestJunit {

  String message = "Robert";
  MessageUtil messageUtil = new MessageUtil(message);

  @Test(expected = ArithmeticException.class)
  public void testPrintMessage() {
    System.out.println("Inside testPrintMessage()");
    messageUtil.printMessage();
  }

  @Test
  public void testSalutationMessage() {
    System.out.println("Inside testSalutationMessage()");
    message = "Hi!" + "Robert";
    assertEquals(message,messageUtil.salutationMessage());
  }
}
```

```
Inside testPrintMessage()
Robert
Inside testSalutationMessage()
Hi!Robert
true
```


JUnit – Parameterized Test (1/3)

```
public class PrimeNumberChecker {
  public Boolean validate(final Integer primeNumber) {
    for (int i = 2; i < (primeNumber / 2); i++) {
      if (primeNumber % i == 0) {
        return false;
      }
    }
    return true;
  }
}
```


JUnit – Parameterized Test (2/3)

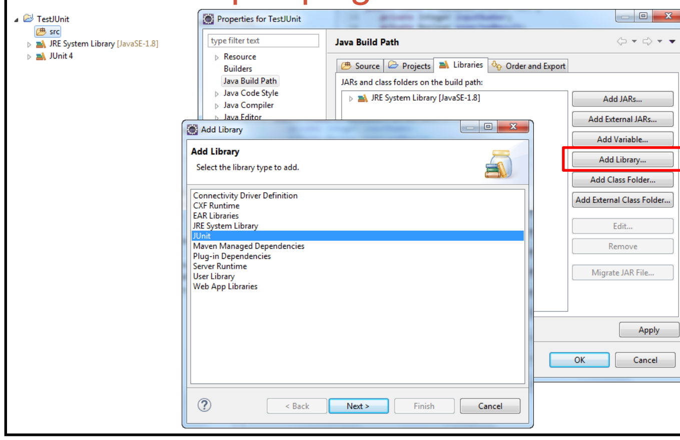
```
import java.util.*;
import java.util.Collection;

import org.junit.Test;
import org.junit.Before;

import org.junit.runners.*;
import org.junit.runners.Parameterized.*
import static org.junit.Assert.assertEquals;

@RunWith(Parameterized.class)
public class PrimeNumberCheckerTest {
  private Integer inputNumber;
  private Boolean expectedResult;
  private PrimeNumberChecker primeNumberChecker;

  @Before
  public void initialize() {
    primeNumberChecker = new PrimeNumberChecker();
  }

  public PrimeNumberCheckerTest(Integer inputNumber, Boolean expectedResult) {
    this.inputNumber = inputNumber;
    this.expectedResult = expectedResult;
  }
```

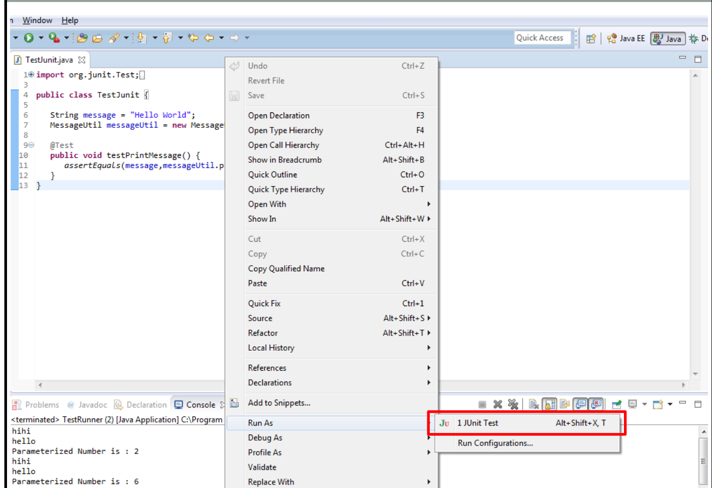
JUnit – Parameterized Test (3/3)

```
@Parameterized.Parameters
public static Collection primeNumbers() {
    return Arrays.asList(new Object[][] {
        { 2, true },
        { 6, false },
        { 19, true },
        { 22, false },
        { 23, true }
    });
}

// This test will run 5 times since we have 5 parameters defined
@Test
public void testPrimeNumberChecker() {
    System.out.println("Parameterized Number is : " + inputNumber);
    assertEquals(expectedResult,
    primeNumberChecker.validate(inputNumber));
  }
}
```

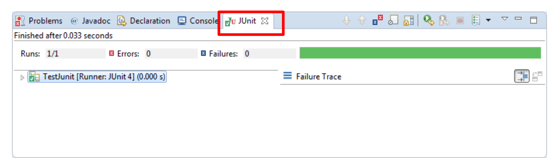
```
Parameterized Number is : 2
Parameterized Number is : 6
Parameterized Number is : 19
Parameterized Number is : 22
Parameterized Number is : 23
true
```


JUnit – Eclipse plugin

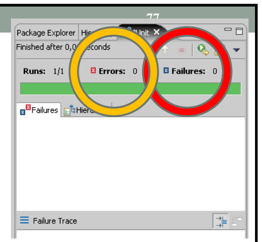




JUnit – Eclipse plugin



Running JUnit tests



- There are many ways to run JUnit test method, test classes, and test suites
- Generally, select the method, class or suite and **`Run As >> JUnit Test`**
- A green bar says "all tests pass"
- A red bar says at least one test failed or was in error
- The failure trace shows which tests failed and why

- A failure is when the test doesn't pass – that is, the oracle it computes is incorrect
- An error is when something goes wrong with the program that the test didn't check for (e.g., a null pointer exception)

Another example – Testing `RandomHello`

- "Create your first Java class with a main method that will randomly choose, and then print to the console, one of five possible greetings that you define."
- We'll focus on the method **`getGreeting`**, which randomly returns one of the five greetings
- We'll focus on ***black-box testing*** – we will work with no knowledge of the implementation
- And we'll focus on unit testing using the **JUnit framework**

- Intermixing, with any luck, slides and a demo

RandomHello class

```
import java.util.Random;

public class RandomHello {
  public static String[] greetings = new String[] {
                    "Hello World",
                    "Hola Mundo",
                    "Bonjour Monde",
                    "Hallo Welt",
                    "Ciao Mondo"
  };

  public static String getGreeting() {
          Random randomGenerator = new Random();
          return greetings[randomGenerator.nextInt(5)];
  }
}
```


Does it even run and return?

- If `getGreeting` doesn't run and return without throwing an exception, it cannot meet the specification

JUnit tag "this is a test"	`@Test`
name of test	`public void test_NoException(){`
Run `getGreeting`	`RandomHello.getGreeting();`
JUnit "test passed"	`assertTrue(true);`
(doesn't execute if exception thrown)	`}`

Tests should have descriptive (often very long) names

A unit test is a (stylized) program! When you're writing unit tests (and many other tests), you're programming!

Does it return one of the greetings?

- If it doesn't return one of the defined greetings, it cannot satisfy the specification

```
@Test
public void testDoes_getGreeting_returnDefinedGreeting() {
  String rg = RandomHello.getGreeting();
  for (String s : RandomHello.greetings) {
    if (rg.equals(s)) {
      assertTrue(true);
      return;
    }
  }
  fail("Returned greeting not in greetings array");
}
```

A JUnit test class

Don't forget that Eclipse can help you get the right `import` statements – use Organize Imports (Ctrl-Shift-O)

```
import org.junit.*;
import static org.junit.Assert.*;
public class RandomHelloTest() {
  @Test
  public void test_ReturnDefinedGreeting() {
    …
  }
  @Test
  public void test_EveryGreetingReturned() {
    …
  }
  …
}
```

- All `@Test` methods run when the test class is run
- That is, a JUnit test class is a set of tests (methods) that share a (class) name

Does it return a random greeting?

```
@Test
public void testDoes_getGreetingNeverReturnSomeGreeting() {
  int greetingCount = RandomHello.greetings.length;
  int count[] = new int[greetingCount];
  for (int c = 0; c < greetingCount; c++)
    count[c] = 0;
  for (int i = 0; i < 100; i++) {
    String rs = RandomHello.getGreeting();
    for (int j = 0; j < greetingCount; j++)
      if (rs.equals(RandomHello.greetings[j]))
        count[j]++;
  }
  for (int j = 0; j < greetingCount; j++)
    if (count[j] == 0)
      fail(j+"th [0-4] greeting never returned");
  assertTrue(true);
}
```

Run it 100 times

If even one greeting is never returned, it's unlikely to be random (~$1-0.8^{100}$)

What about a sleazy developer?

```
if (randomGenerator.nextInt(2) == 0) {
   return(greetings[0]);
} else
  return(greetings[randomGenerator.nextInt(5)]);
```

- Flip a coin and select **either** a random **or** a specific greeting
- The previous "is it random?" test will almost always pass given this implementation
- But it doesn't satisfy the specification, since it's not a random choice

Instead: Use simple statistics

```
@Test
public void test_UniformGreetingDistribution() {
  // …count frequencies of messages returned, as in
  // …previous test (test_EveryGreetingReturned)

   float chiSquared = 0f;
   float expected = 20f;
   for (int i = 0; i < greetingCount; i++)
     chiSquared = chiSquared +
             ((count[i]-expected)*
              (count[i]-expected))
                          /expected;
   if (chiSquared > 13.277) // df 4, pvalue .01
     fail("Too much variance");
}
```


A JUnit test suite

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;

@RunWith(Suite.class)
@Suite.SuiteClasses({
  RandomHelloTest.class,
  SleazyRandomHelloTest.class
})
public class AllTests {
  // this class remains completely
  // empty, being used only as a
  // holder for the above
  // annotations
}
```

- Define one suite for each program (for now)
- The suite allows multiple test classes – each of which has its own set of `@Test` methods – to be defined and run together
- Add `tc.class` to the `@Suite.SuiteClasses` annotation if you add a new test class named `tc`
- So, a JUnit test suite is a set of test classes (which makes it a set of a set of test methods)

`ArrayIntList`: example tests

```
@Test
public void testAddGet1() {
  ArrayIntList list = new
              ArrayIntList();
  list.add(42);
  list.add(-3);
  list.add(15);
  assertEquals(42, list.get(0));
  assertEquals(-3, list.get(1));
  assertEquals(15, list.get(2));
}
```

```
@Test
public void testIsEmpty() {
  ArrayIntList list = new
                ArrayIntList();
  assertTrue(list.isEmpty());
  list.add(123);
  assertFalse(list.isEmpty());
  list.remove(0);
  assertTrue(list.isEmpty());
}
```

- High-level concept: test behaviors in combination
  - Maybe `add` works when called once, but not when call twice
  - Maybe `add` works by itself, but fails (or causes a failure) after calling `remove`

A few hints: data structures

- Need to pass lots of arrays? Use array literals

```
public void exampleMethod(int[] values) { ... }
...
exampleMethod(new int[] {1, 2, 3, 4});
exampleMethod(new int[] {5, 6, 7});
```

- Need a quick `ArrayList`?

```
List<Integer> list = Arrays.asList(7, 4, -2, 3, 9, 18);
```

- Need a quick set, queue, etc.? Many take a list

```
Set<Integer> list = new HashSet<Integer>(
                        Arrays.asList(7, 4, -2, 9));
```

A few general hints

- Test one thing at a time per test method
  - 10 small tests are much better than one large test
- Be stingy with **`assert`** statements
  - The first **`assert`** that fails stops the test – provides no information about whether a later assertion would have failed
- Be stingy with logic
  - Avoid **`try/catch`** – if it's supposed to throw an exception, use **`expected=`** ... if not, let JUnit catch it

Test case dangers

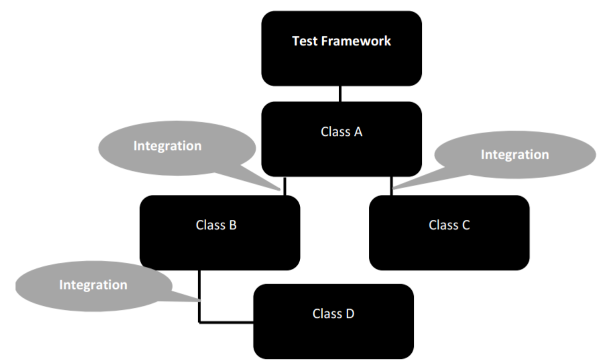
- Dependent test order
  - If running Test A before Test B gives different results from running Test B then Test A, then something is likely confusing and should be made explicit
- Mutable shared state
  - Tests A and B both use a shared object – if A breaks the object, what happens to B?
    - This is a form of dependent test order
    - We will explicitly talk about invariants over data representations and testing if the invariants are ever broken

Content

1. Testing overview
2. Unit Test
3. **Integration Test**

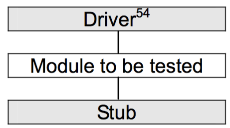
3. Integration Test

- Integration testing is a logical extension of unit testing



Stubs and Drivers in Integration testing

- **Stubs**
  - are dummy modules that are always distinguish as "called programs“
  - used when sub programs are under construction.
  - simulate the low level modules
- **Drivers**
  - dummy modules which are always distinguished as "calling programs”
  - used when main programs are under construction
  - simulate the high level modules

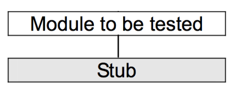


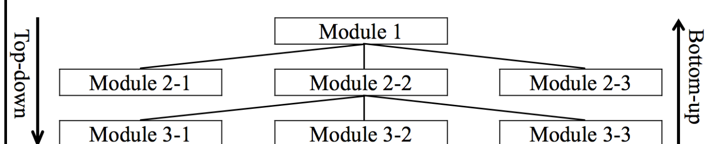
Integration testing types

- 2 groups:
  - Incremental software integration: Top-Down, Bottom-Up, Sandwich testing
  - Non Incremental software integration: Big bang integration approach

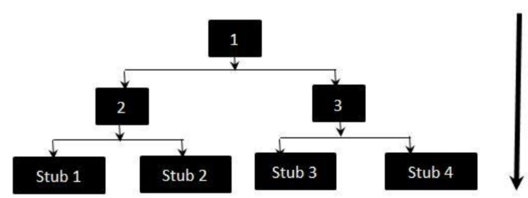
3.1. Top-down approach

- Defects based on misunderstanding of specification can be detected early
- Effective in newly developed systems
- Need test stubs (can be simply returning a value)



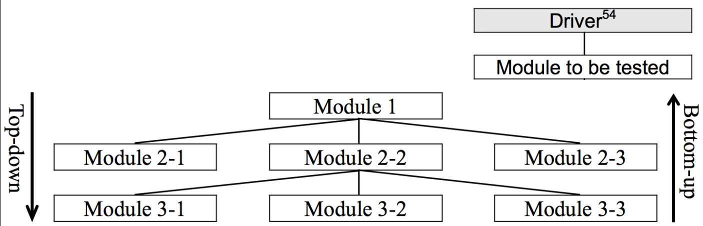


3.1. Top-down approach



3.2. Bottom-up approach

- Lower modules are independent => test independently and on a parallel
- Effective in developing systems by modifying existing systems
- Need test drivers (more complex with controlling)



3.3. Other integration test techniques

- Sandwich test
  - Where lower-level modules are tested bottom-up and higher-level modules are tested top-down
- Big-bang test
  - Wherein all the modules that have completed the unit tests are linked all at once and tested
  - Reducing the number of testing procedures in small-scale program; but not easy to locate errors

3.4. Regression test

"When you fix one bug, you introduce several new bugs"

- Re-testing an application after its code has been modified to verify that it still functions correctly
  - Re-running existing test cases
  - Checking that code changes did not break any previously working functions (side-effect)
- Run as often as possible
- With an automated regression testing tool