ITSS SOFTWARE DEVELOPMENT

08. UNIT TEST

Content

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

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

Software Quality Assurance (QA)

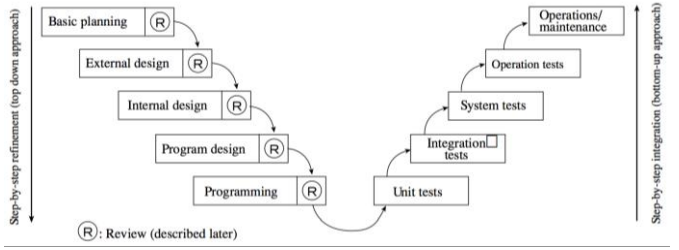
Testing plus other activities including

- 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

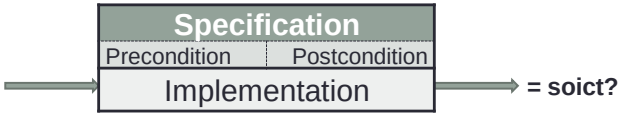
- Unit Testing: Does each unit (class, method, etc.) do what it supposed to do?
  - Smallest programming units
  - Strategies: Black box and white box testing
  - Techniques, Tools
- Integration Testing: do you get the expected results when the parts are put together?
  - Strategies: Bottom-up, top-down 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 approaches
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
Must choose inputs *without knowledge* of the implementation

White box
Can choose inputs *with knowledge* of the implementation

It's not black-and-white, but…

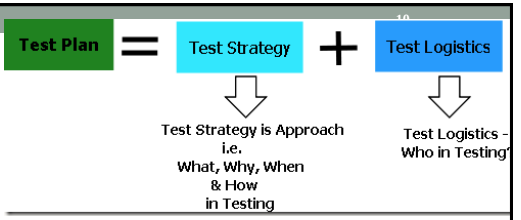
Black box
Must choose inputs *without knowledge* of the implementation

- 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

White box
Can choose inputs *with knowledge* of the implementation

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

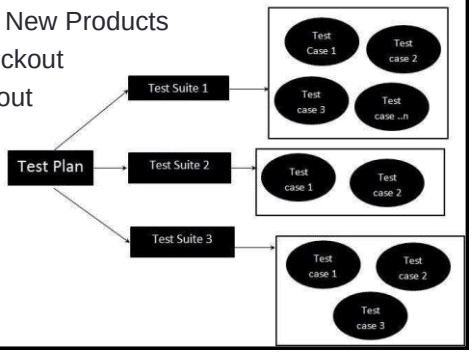
Terms



- 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 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. Blackbox Testing Techniques
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

13

Example: Examination Judgment Program

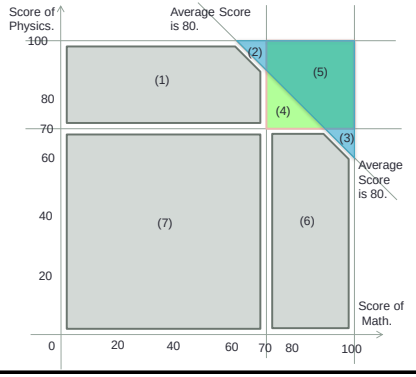
- Program Title: “Examination Judgment Program”
- Subject: Two subjects as Mathematics, and Physics Judgment
- Specification:
  - Passed if
    - scores of both mathematics and physics are greater than or equal to 70 out of 100

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

14

Equivalence Partitioning of Input space and test cases

- 7 equivalence classes => at least 7 test cases/data



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

15

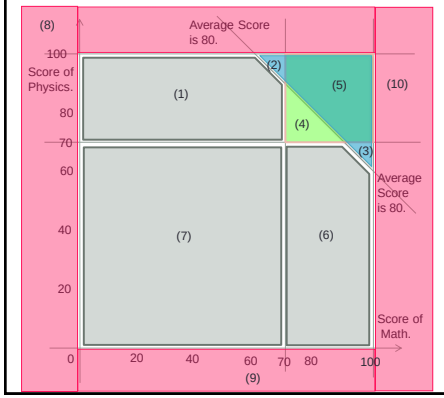
Equivalence Partitioning
Discussions and additional analysis

- Are we successful?
  - No we don’t! Why?
  - →One thing is missing!
- The scope of input space analyzed is not enough!
- We must add “Invalid value” as the test data.
  - For example, some patterns of “Invalid value”.
    - (8) Math = -15, Physics = 120 Both score are invalid.
    - (9) Math = 68, Physics = -66 Physics score is invalid.
    - (10) Math = 118, Physics = 85 Math score is invalid.

16

More equivalent classes

- Additional 3 test cases/data



Some invalid data are added.

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

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. Blackbox Testing Techniques
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:

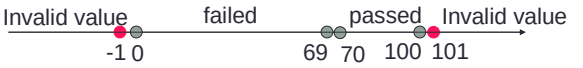
```
if (mathscore > 70) {
      ..................
}
```

- Instead of the following correct code;

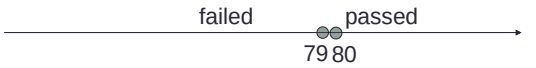
```
if (mathscore >= 70) {
      ..................
}
```


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. Blackbox Testing Techniques
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

21

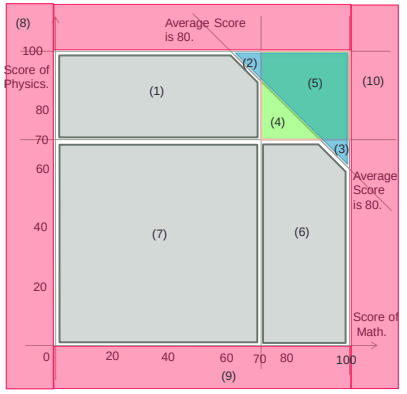
Example: Decision table

- 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	-

22

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

23

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

24

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?

25

2.2. Blackbox Testing Techniques
2.2.4. Testing for Use case

- ???
- E.g. Decision table for Login
  - Conditions
    - ???
  - Results
    - ???
- E.g. Boundary Value Analysis
  - ?

26

Test cases for "Log in"

- "Thành công"
  - Mã PIN đúng
- "Thất bại"
  - Mã PIN sai và số lần sai < 3
- "Khoá tài khoản"
  - Mã PIN sai và số lần sai = 3

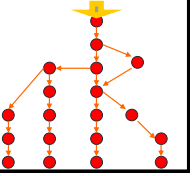
Mã PIN đúng	**Y**	**Y**	**N**	**N**
Số lần sai < 3	**Y**	**N**	**Y**	**N**
"Thành công"	x	N/A	-	-
"Thất bại"	-	N/A	x	-
"Khoá tài khoản"	-	N/A	-	x

- Phân tích vùng biên? Số lần sai = 2, 4 (?)

27

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



28

2.3. White Box Testing Techniques

- Test cases should cover all processing structure in code

=> Typical test coverage

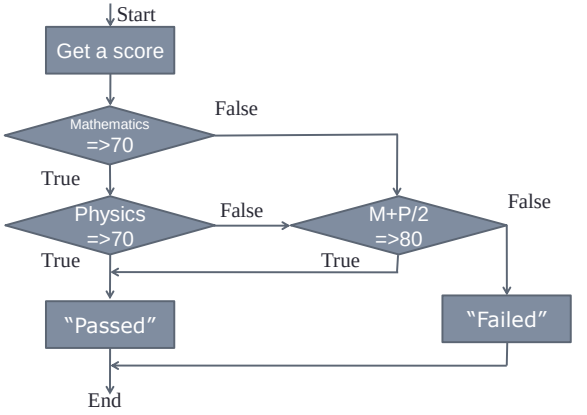
- C0 measure: Executed statements #/all statements #
  - C0 measure at 100% means “all statements are executed”
- C1 measure: Branches passed #/all blanches #
  - C1 measure at 100% means “all branches are executed”

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

=> Cannot detect functions which aren't implemented

29

E.g. Control flow test of “Examination Judgment Program”

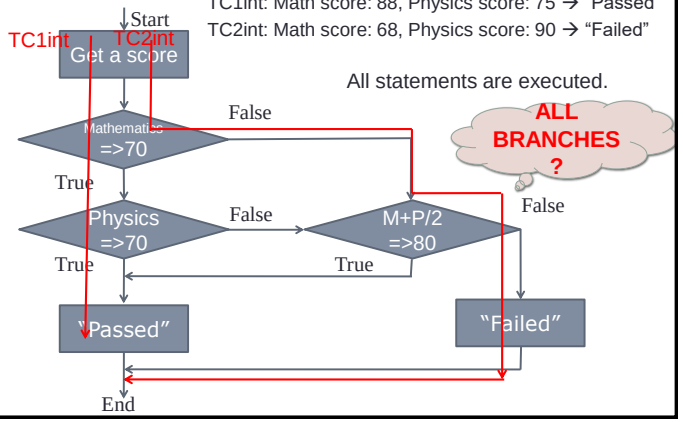


30

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.

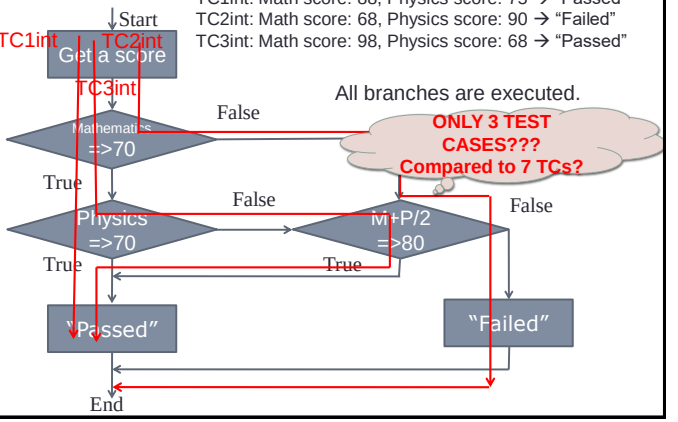


31

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.



32

Decision Table for “Examination Judgement”

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

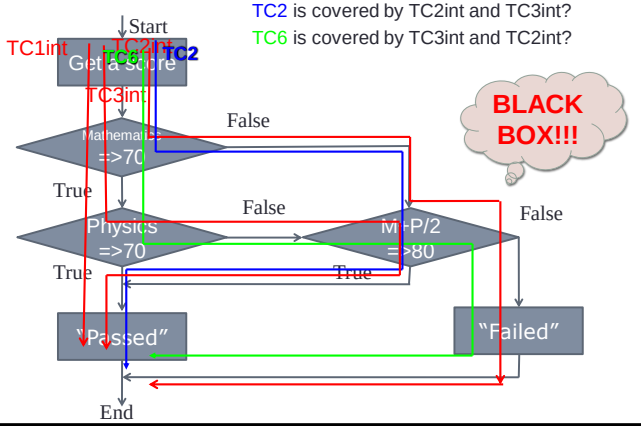
	TC5	TC4	TC3	TC6	TC2	TC1	TCNG	TC7
Condition1	True	True	True	True	False	False	False	False
Condition2	True	True	False	False	True	True	False	False
Condition3	True	False	True	False	True	False	True(none)	False
“Passed”	Yes	Yes	Yes	---	Yes	---	N/A	--
“Failed”	---	---	---	Yes	---	Yes	N/A	Yes

- 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

33

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?

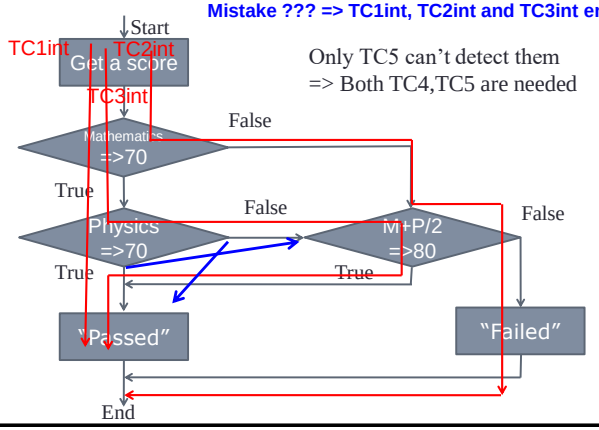


34

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

Mistake ??? => TC1int, TC2int and TC3int enough?

Only TC5 can’t detect them
=> Both TC4,TC5 are needed



35

Data/message path test for integrated test

- Execute white box test using sequence chart for integration test.

⇒Execute every message path/flow

⇒100% message path/flow coverage

- Can apply to other data/message path/flow charts or diagrams

36

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.
  - Only one pass 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:

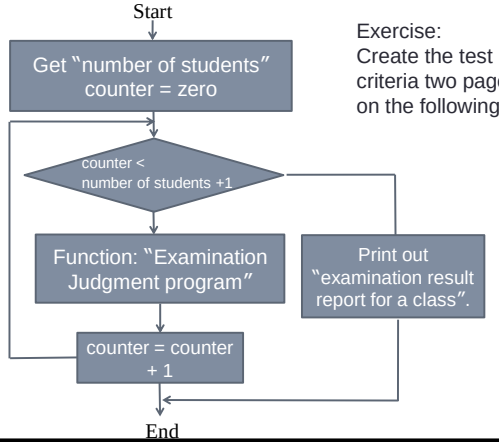
37

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.

38

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.

39

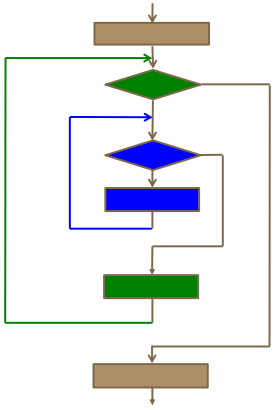
Examples for "Examination Judgment Program"

Loop test cases of the module are; n = 50.

"number of students" = 0,
"number of students" = 1,
"number of students" = 20,
"number of students" = 49,
"number of students" = 50,
"number of students" = 51 → Invalid.

40

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.

41

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"

42

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

43

2.5. JUnit

- A tool for test-drivent development (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#, …)

44

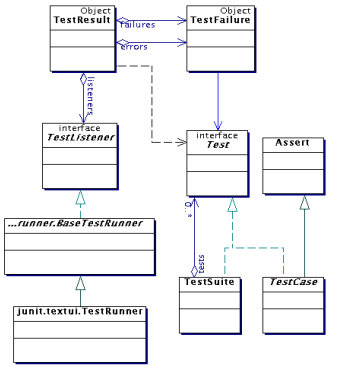
Why create a test suite?

- 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

45

Architectural overview

- JUnit test framework is a package of classes that lets you write tests for each method, then easily run those tests
- **TestRunner** 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



46

Writing a TestCase

- To start using JUnit, create a subclass of *TestCase*, to which you add test methods
- Here's a skeletal test class:

```
import junit.framework.TestCase;
public class TestBowl extends TestCase {

} //Test my class Bowl
```

- Name of class is important – should be of the form ***Test***MyClass or MyClass***Test***
- This naming convention lets TestRunner automatically find your test classes

47

Writing methods in TestCase

- Pattern follows ***programming by contract*** paradigm:
  - Set up **preconditions**
  - Exercise functionality being tested
  - Check **postconditions**
- Example:

```
public void testEmptyList() {
    Bowl emptyBowl = new Bowl();
    assertEquals("Size of an empty list should be zero.",
                0, emptyList.size());
    assertTrue("An empty bowl should report empty.",
    emptyBowl.isEmpty());
}
```

- Things to notice:
  - Specific method signature – public void ***test***Whatever()
    - Allows them to be found and collected automatically by JUnit
  - Coding follows pattern
  - Notice the assert-type calls…

48

Assert methods

- Assert methods dealing with floEach assert method has parameters like these: *message, expected-value, actual-value*
- ating 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

49

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*)

50

More stuff in test classes

- Suppose you want to test a class Counter
- public class CounterTest
  extends junit.framework.TestCase {
  - This is the unit test for the Counter class
- public CounterTest() { } //Default constructor
- protected void setUp()
  - Test *fixture* creates and initializes instance variables, etc.
- protected void tearDown()
  - Releases any system resources used by the test fixture
- public void testIncrement(), public void testDecrement()
  - These methods contain tests for the Counter methods increment(), decrement(), etc.
  - Note capitalization convention

51

JUnit tests for Counter

```
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);
   }
}
```

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

52

TestSuites

- 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 static Test suite() {
        suite.addTest(new TestBowl("testBowl"));
        suite.addTest(new TestBowl("testAdding"));
        return suite;
}
```

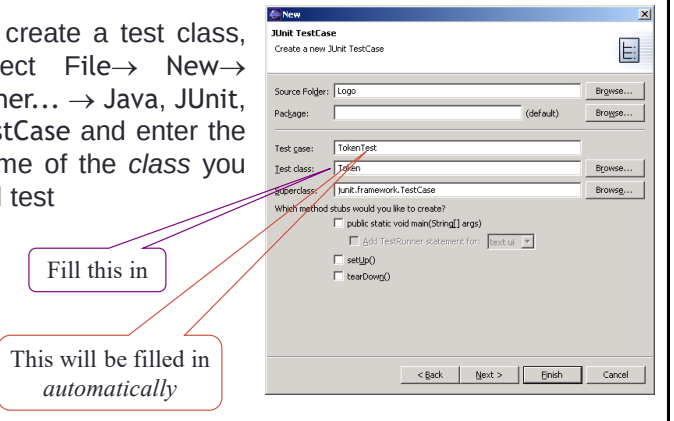
- Should seldom have to write your own TestSuites as each method in your TestCase should be independent of all others
- Can create TestSuites that test a whole package:

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

53

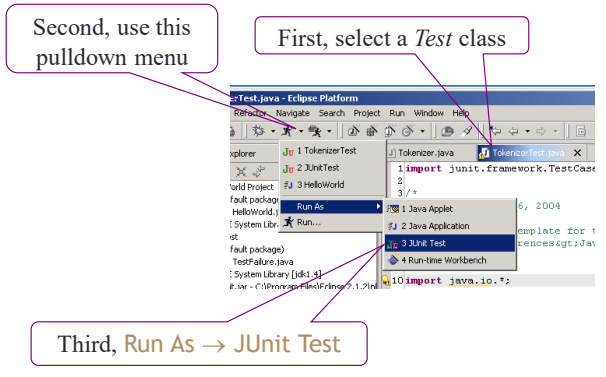
JUnit in Eclipse

- To create a test class, select File→ New→ Other… → Java, JUnit, TestCase and enter the name of the *class* you will test



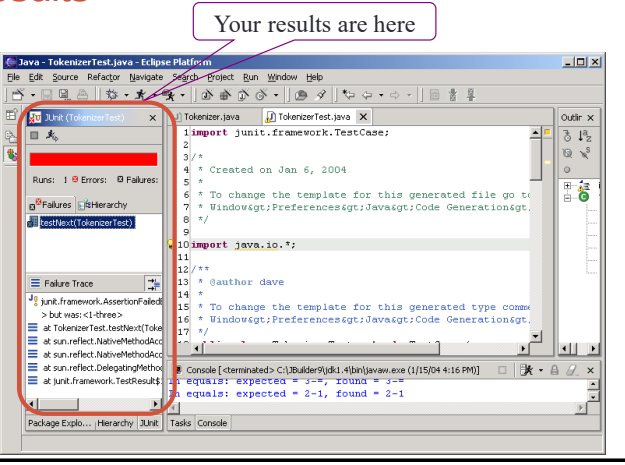
54

Running JUnit



55

Results



56

Unit testing for other languages

- Unit testing tools differentiate between:
  - Errors (unanticipated problems caught by exceptions)
  - Failures (anticipated problems checked with assertions)
- Basic unit of testing:
  - *CPPUNIT_ASSERT(Bool)* examines an expression
- CPPUnit has variety of test classes (e.g. *TestFixture*)
  - Inherit from them and overload methods

Another example: sqrt

```
// throws: IllegalArgumentException if x < 0
// returns: approximation to square root of x
public double sqrt(double x)
```

What are some values or ranges of `x` that might be worth testing

- `x < 0` (exception thrown)
- `x ≥ 0` (returns normally)
- around `x = 0` (boundary condition)
- perfect squares (`sqrt(x)` an integer), non-perfect squares
- `x < sqrt(x)`, `x > sqrt(x)`
- Specific tests: say `x = {-1, 0, 0.5, 1, 4}`

Subdomains

- Many executions reflect the same behavior – for **`sqrt`**, for example, the expectation is that
  - all `x < 0` inputs will throw an exception
  - all `x ≥ 0` inputs will return normally with a correct answer
- By testing any element from each *subdomain*, the intention is for the single test to represent the other behaviors of the subdomain – *without testing them!*
- Of course, this isn't so easy – even in the simple example above, what about when **`x`** overflows?

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

61

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" (doesn't execute if exception thrown)	`assertTrue(true);` `}`

Tests should have descriptive (often very long) names

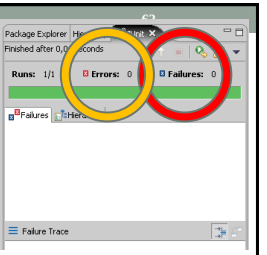
A unit test is a (stylized) program! When you're writing unit tests (and many other tests), you're programming!

61

62

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)

62

63

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");
}
```

63

64

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

64

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 = 1; 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)

JUnit assertion methods

…causes the current test to fail…	
`fail()`	`immediately`
`assertTrue(tst)`	`if tst is false`
`assertFalse(tst)`	`if test is true`
`assertEquals(expected, actual)`	`if expected does not equal actual`
`assertSame(expected, actual)`	`if expected != actual`
`assertNotSame(expected, actual)`	`if oracle == actual`
`assertNull(value)`	`if value is not null`
`assertNotNull(value)`	`if value is null`

- Can add a failure message: `assertNull("Ptr isn't null", value)`
- **`expected`** is the oracle – remember this is the first (leftmost) param
- The table above only describes when to fail – what happens if an assertion succeeds? Does the test pass?

`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

More JUnit

- Timeouts – don't want to wait forever for a test to complete
- Testing for exceptions

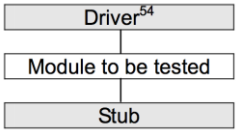
```
@Test(expected = ArrayIndexOutOfBoundsException.class)
public void testBadIndex() {
  ArrayIntList list = new ArrayIntList();
  list.get(4); // this should raise the exception
}              // and thus the test will pass
```

- Setup [teardown] – methods to run before [after] each test case method [test class] is called

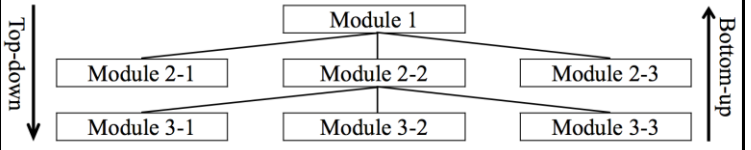
Content

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

3. Integration test



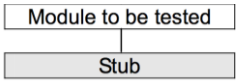
- Examine the interface between modules as well as the input and output
- Stub/Driver:
  - A program that simulates functions of a lower-level/upper-level module

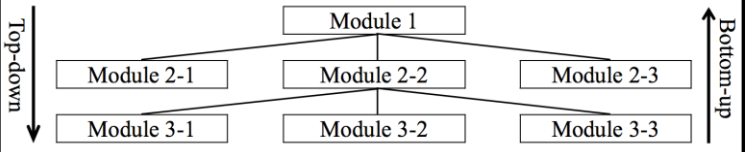


77

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)



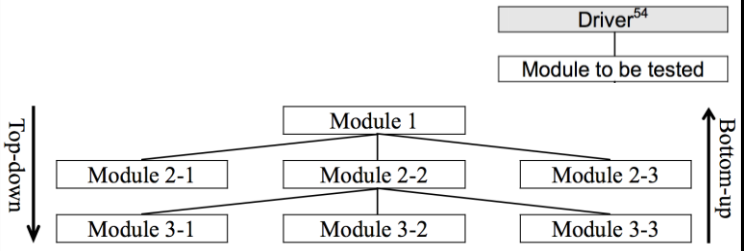


77

78

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)



78

79

3.3. Other integration test techniques

- 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
- Sandwich test
  - Where lower-level modules are tested bottom-up and higher-level modules are tested top-down

79

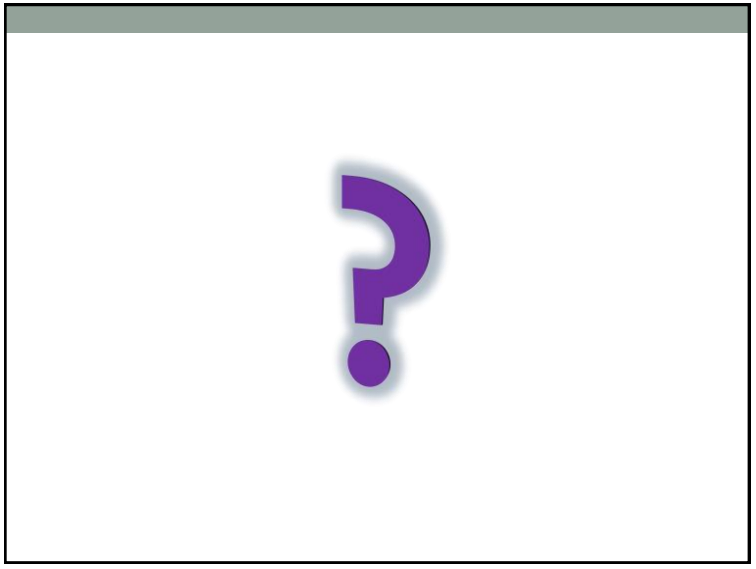
80

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

80



81