Lecture 3

Black-box Testing

See Ammann and Offutt Chapter 6

Black-box Testing

Test cases are constructed without reference to the code structure

+ Can test the requirements not the code
+ Can overcome combinatorial explosions
+ Complementary to glass-box testing
+ Insensitive to code refactoring
- Hard to find test verdicts – aka oracle problem
- Hard to define coverage
- Large volume of testing – user profiles?
- Use cases are an excellent source of tests

Some Methods

- Random testing
- Boundary value testing
- N-wise and pairwise testing
  - (combinatorial testing, Amman and Offutt 6.2)
- Testing from use cases
  - (A&O 4.2.2)

and a practical framework: Junit (A&O 3.2)

Random Testing

- Generate input vectors (or sequences) at random, fire into system and observe.
- Easy or tricky to implement
  - Low level data types … e.g. Int … easy
  - High level data types … e.g. graphs … tricky
- High volume of test cases … but is that good structural coverage? Don't know
- Good for low input dimension 1-5?
  - But poor for high input dimension how many input variables

Random Testing

- Oracle step is difficult to automate without precise requirements
- Set up and tear down must also be considered
- Does random distribution match expected distribution?
- Are some data combinations meaningless? Data interdependencies and constraints!
  - Example consider calendar combinations:
  - Year/month/day/day of week
  - 1961/02/29/Wednesday … is this legal or not?
- Can try to filter out bad data but this can be very slow

The Curse of Dimensionality

- Most programs take a large amount of input data.
- Define problem dimension D to be the total number of input variables to the SUT.
- Question: how to count structured variables e.g. arrays?
- Example:
  - 32 bit integers,
  - 10 integer input variables
  - 2^{320} possible input values

no difference in practice between large finite number and infinite number

- This is the curse of (all) high dimensional problems
- c.f. multivariate integration in physics!

1st solution: Boundary Value Testing

- Choose a small set of ***boundary values*** for each variable

  e.g. $B_i = \{ max^+, max, max^-, min^-, min, min^+ \}$,

  -: little bit below

  +:little bit above

  6 in this case

  say C choices for each input variable v_i
- Boundary values might be chosen with reference to user requirements.
- Test suite TS is all combinations of $B_1, \ldots, B_D$
- Cartesian Product $TS = B_1 \times \ldots \times B_D$
- Test suite size $|TS| = |B_1| \times \ldots \times |B_D| = C^D$

  where D is the total no. of variables and C are the choices of each variable.

  - e.g. 6^D for B_i above.

- Still face exponential explosion!

Example

- Suppose $D = 3$, $C = 2$, then $C^D = 2^3 = 8$. Suppose
- $V^1 = \{$ Mon, Sun $\}$
- $V^2 = \{$ A, Z $\}$
- $V^3 = \{$ 0, Maxint $\}$ (positive integers)
- TS = {
  (Mon, A, 0), (Sun, A, 0), (Mon, Z, 0), (Sun, Z, 0),
  (Mon, A, Maxint), (Sun, A, Maxint), (Mon, Z, Maxint),
  (Sun, Z, Maxint) }

Clearly TS has size 8
In general TS has size C^D which is exponential in the problem dimension D.

2nd solution – n-Wise Testing

(Ammann and Offutt Chapter 6.3)

Let D be the total number of input variables (i.e. dimension).

Let $V_i = \{ t_i : C \geq i \geq 1\}$ be set of C typical values for the input variable v_i, for each $D \geq i \geq 1$.

should only have one default value

Let $d_i \in V_i$ be some default value for v_i , for each $D \geq i \geq 1$

An n-wise test case is an input vector of typical values

$$(i_1, \ldots, i_D) \in V_1 \times \ldots \times V_D$$

containing exactly n non-default values.

An n-wise test suite TS is a set of test cases, such that:

1. Each test case $tc \in TS$ is at most an n-wise test case
2. Each n-tuple of typical values appears in at least one test case $tc \in TS$.

if we are doing pair-wise testing, we include 0-wise and 1-wise testing

Example: 1-wise testing

- Choose $D=3$, $n = 1$, and we get 1-wise testing of a 3-dimensional testing problem.
- Choose defaults $d_1 = \text{Sat}$, $d_2 = A$, $d_3 = 1$, and $k=2$ typicals

$V_1 = \{ \text{Sat}, \text{Sun} \}$

$V_2 = \{ A, B \}$

$V_3 = \{ 1, 2 \}$

Let $TS1 = \{\underbrace{(\text{Sat}, A, 1)}_{\text{0-wise test}}, \underbrace{(\text{Sun}, A, 1)}_{\text{1-wise test}}, \underbrace{(\text{Sat}, B, 1)}_{\text{1-wise test}}, \underbrace{(\text{Sat}, A, 2)}_{\text{1-wise test}} \}$

Clearly TS1 has size at most $((k-1)*D) +1$ which is <u>linear</u> in D.

(c.f. k^D) So TS1 is small but has limited coverage. Notice, because $D>1$, (Sat, A, 1) is a 0-wise test that is redundant.

Example: 2-wise testing (aka *all-pairs* or *pairwise testing*)

- For the same problem suppose n = 2
- Choose same defaults d_1 = Sat, d_2 = A, d_3 = 1 and typicals
- V_1 = { Sat, Sun }
- V_2 = { A, B }
- V_3 = { 1, 2 }
- TS2 =

  {(Sat, A, 1), [0-wise]
  (Sun, A, 1), (Sat, B, 1), (Sat, A, 2), [1-wise]
  (Sun, B, 1), (Sun, A, 2), (Sat, B, 2) [2-wise]}

Clearly TS2 has size 7 which is not much bigger than TS1.
In general n+1-wise test suites are much larger than n-wise test suites
Pairwise test suites grow at most quadratically $O(D^2)$, which is still much slower than C^D.

D: number of input variables, here is it V

Why Pairwise Testing?

- Bugs involving interactions between three or more parameters are progressively less common[2].

large increase in coverage

- NASA database application. 67 percent of the failures were triggered by only a single parameter value, 93 percent by two-way combinations, and 98 percent by three-way combinations [13].

not that large increase

- 10 UNIX commands. Cohen et al. showed that the pairwise tests gave over 90 percent block coverage [9].
- Medical software devices. Only 3 of 109 failure reports indicated that more than two conditions were required to cause the failure [14].

That's why pairwise testing is quite a reasonable approach to combinatorial testing

Why Pairwise Testing

- Browser and server. More than 70 percent of bugs were detected with two or fewer conditions (75 percent for browser and 70 percent for server) and approximately 90 percent of the bugs reported were detected with three or fewer conditions (95 percent for browser and 89 percent for server) [13].
- User interface software at Telcordia. Studies [8] showed that most field faults were caused by either incorrect single values or by an interaction of pairs of values. Their code coverage study also indicated that pairwise coverage is sufficient for good code coverage.
- Established tools, e.g. PICT
- See www.pairwise.org

Test Cases from Use Cases

- Instantiate a scenario with concrete data values, and expected results.
- Different flows lead to different use cases
  - vanilla scenarios exception
  - Sunny day and rainy day scenarios robustness testing, i.e don't crash when people use it in the wrong way
- Use graph coverage to measure use case coverage
- Structured and easy to use
- Natural focus on most significant use cases Pareto principle
- Good approach to system and acceptance testing, but may be difficult and unit and integration levels

UseCaseName: PurchaseTicket

Precondition: The passenger is standing in front of ticket distributor and has sufficient money to purchase a ticket.

Sequence:

1. The passenger selects the number of zones to be travelled, If the passenger presses multiple zone buttons, only the last button pressed is considered by the distributor.
2. The distributor displays the amount due
3. The passenger inserts money

4. If the passenger selects a new zone before inserting sufficient money, the distributor returns all the coins and bills inserted by the passenger

teardown -> going back to default state

5. If the passenger inserted more money than the amount due the distributor returns excess change.

5 before 6 : Delayed gratification

6. The distributor issues ticket.

7. The passenger picks up the change and ticket.

Postcondition

The passenger has an appropriate ticket and change or else their original amount of money.

TestCaseName: PurchaseTicket_SunnyDay

Precondition: The passenger is standing in front of ticket distributor and has two 5€ notes and 3 * 10 Cent coins

Sequence:

1. The passenger presses in succession the zone buttons 2, 4, 1 and 2.
2. The distributor should display in succession the fares 1.25€ 2.25€, 0.75€ and 1.25€
3. The passenger inserts a 5€ note
4. The distributor returns 3*1€ coins, 75Cent

   and a 2-zone ticket.

Postcondition

The passenger has a 2-zone ticket and right change.

The path exercised through this use-case (as a graph) is:

1, 1, 1, 1, 2, 3, 5, 6, 7.

We could also derive test cases that exercise other paths through the use-case i.e. rainy day scenarios (when something goes wrong) to test robustness.

Unit Testing with Junit
(Ammann and Offutt Chapter 3.3)

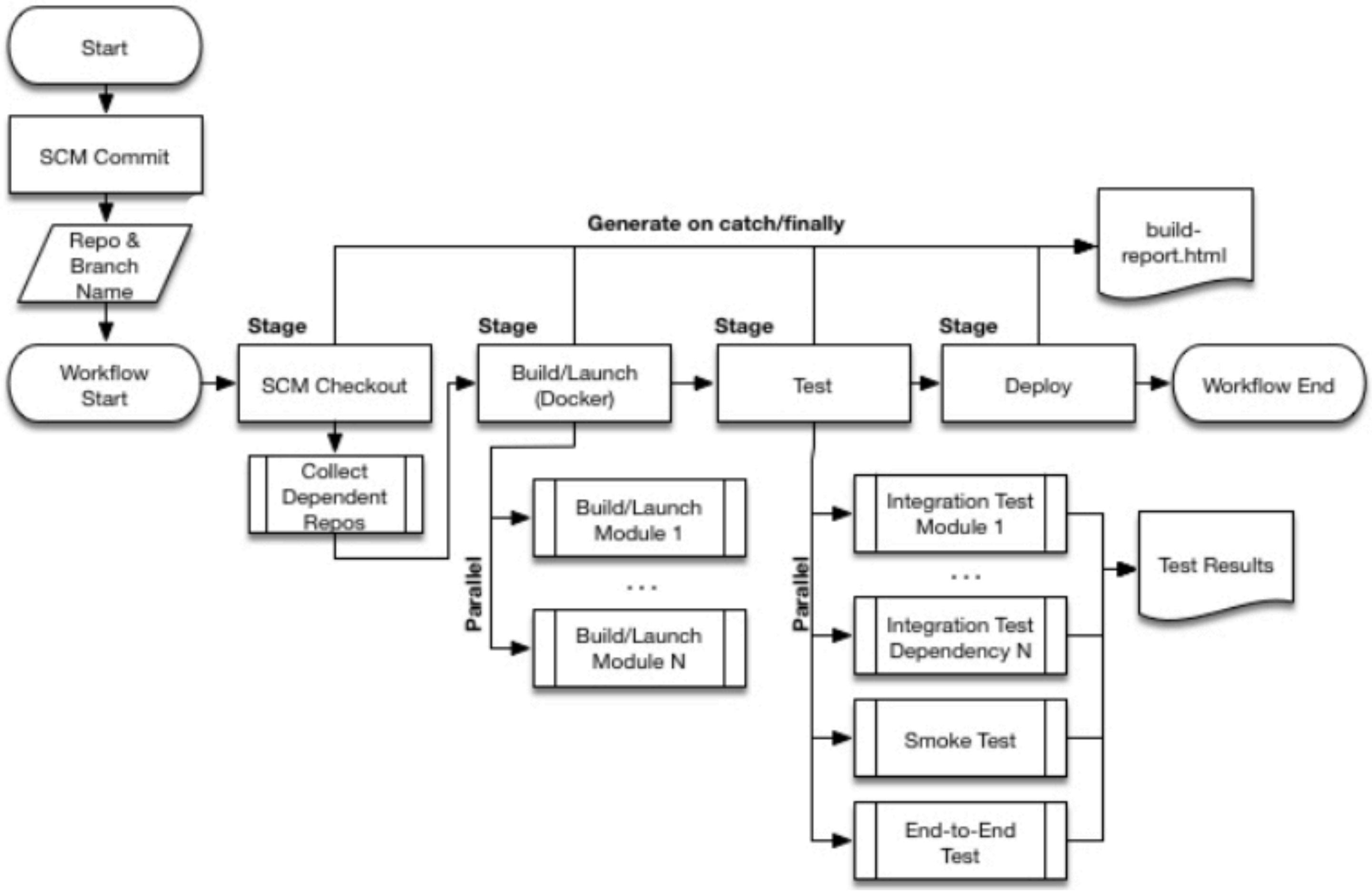
- Developed by the XP community 2002
- Framework for automating the execution of unit test for Java classes
- Write new test cases by subclassing the TestCase class
- Organise TestCases into TestSuites
- Automates testing process
- Built around Command and Composite patterns

Test Driven development -> Help in automation

Why use Junit?

- Junit tightly integrates development and testing, supports the XP approach
- Allows you to write code faster while increasing quality (???)
  - Can refactor code without worrying about correctness
- JUnit is simple.
  - Easy as running the compiler on your code

Xunit in DevOps (Jenkins)

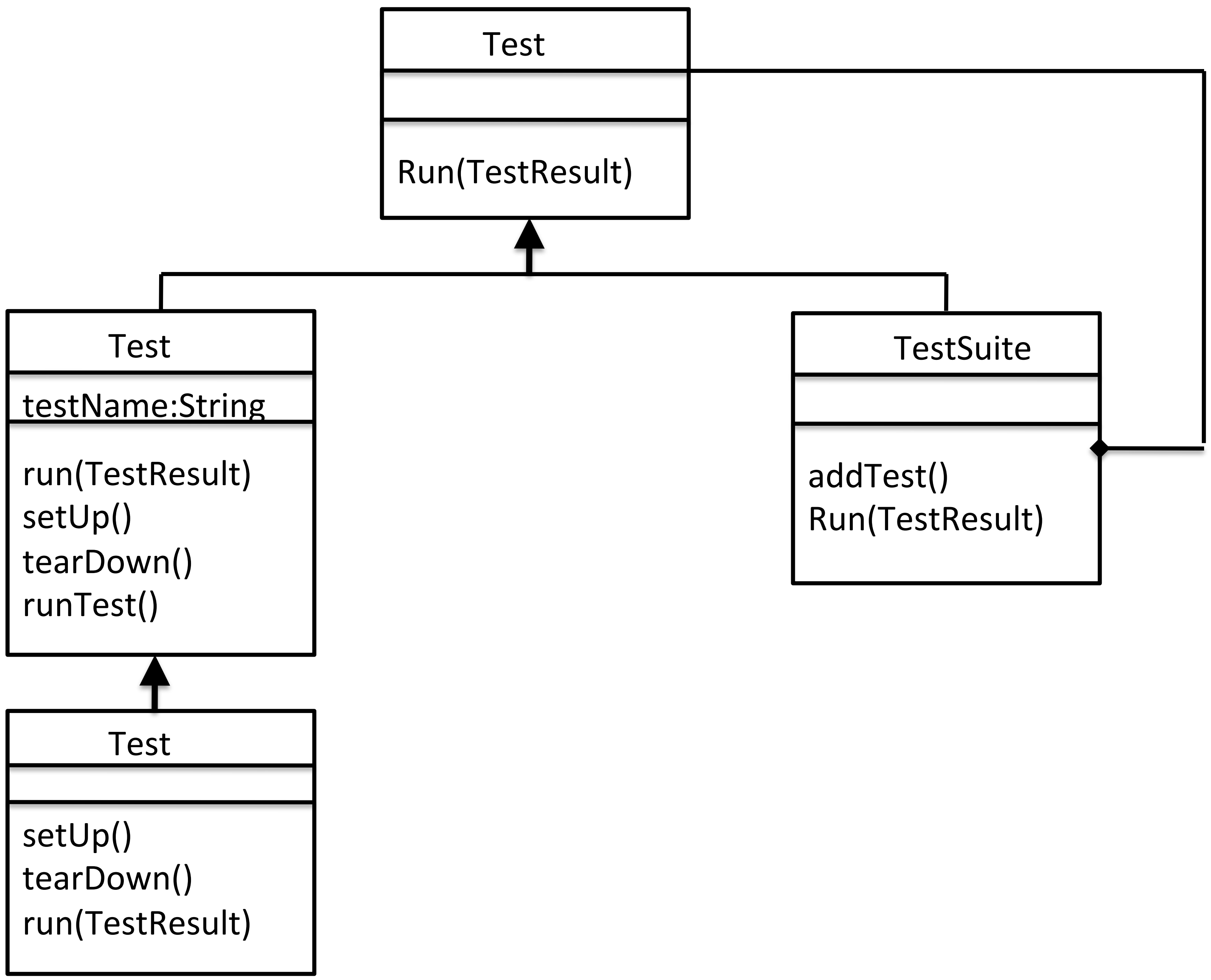


- JUnit tests check their own results (oracle step) and provide immediate feedback
  - No manual comparison of expected with actual
  - Simple visual feedback
- JUnit tests can be composed into a hierarchy of test suites
  - Can run tests for any layer in the hierarchy
- Writing JUnit tests is inexpensive
  - No harder than writing a method to exercise the code
- JUnit tests increase the stability of software
  - More tests = more stability

- Junit tests are developer tests
  - Tests fundamental building blocks of system
  - Tests delivered with code as a certified package
- Junit tests are written in Java
  - Seamless bond between test and code under test
  - Test code can be refactored into software code and vice-versa
  - Data type compatibility (float, double etc.)
- Junit is free

Junit Design

- A TestCase is a Command object
- A class of test methods subclasses TestCase
- A TestCase has public testXXX() methods
- To check expected with actual output invoke assert() method
- Use setUp() and tearDown() to prevent side effects between subsequent testXXX() calls


Test
Run(TestResult)
Test
testName:String
run(TestResult)
setUp()
tearDown()
runTest()
TestSuite
addTest()
Run(TestResult)
Test
setUp()
tearDown()
run(TestResult)

- TestCase objects can be composed into TestSuite hierarchies. Automatically invoke all the testXXX() methods in each object
- A TestSuite is composed of TestCase instances or other TestSuite instances
- Nest to arbitrary depth
- Run whole TestSuite with a single pass/fail result
- Get your own installation instructions

Writing a Test Case

- Define a subclass of TestCase
- Override the setUp() method to intialise object(s) under test
- Optionally override the tearDown() method to release objects under test
- Define 1 or more public testXXX() methods hat exercise the object(s) under test and assert expected results

To create a test with JUnit, go through the following steps:

1. Create a subclass of TestCase: The TestCase fixture.
2. Create a constructor that accepts a String as a parameter and passes it to the superclass.
3. Add an instance variable for each part of the TestCase fixture.
4. Override setUp() to initialize the variables.
5. Override tearDown() to release any permanent resources allocated in setUp.
6. Implement a test method with a name starting with the "test" string, and using assert methods.
7. Implement a "runTest" method to define the logic to run the different tests.

```
Import junit.framework.TestCase

Public class ShoppingCartTest extends TestCase{
Private ShoppingCart cart;
Private Product book1;

Protected void setUp() {
Cart = new ShoppingCart();
Book1 = new Product(“myTitle”, “50€”);
Cart.addItem(book1) }
```

```
public classMoneyTest extends TestCase {
private Money f12SEK; private Money f24SEK;

protected void setUp() { f12SEK = new Money(12, ”SEK");
f24SEK = new Money(24, ”SEK");
}
public void testAdd() { Money f36SEK = f12SEK.add(f24SEK);
assertTrue(f36SEK.equals(new Money(36, ”SEK"));
}
public void testMultiply() { Money f24SEKa = f12SEK.multiply(new
Money(2, ”SEK"));
assertTrue(f24CHF.equals(f24SEK));
}
protected void runTest throws Throwable() {
testAdd();
testMultiply();
}
```

```
Protected void tearDown() {
   //release objects under test here if necessary
}
Public void testEmpty() {
Cart.empty(); // empty out cart
assertEquals(0,cart.getItemCount() );
}
```

```
Public void testAddItem() {
Product book2 = new Product("title2", "65€");
cart.addItem(book2);
double expectedBalance =
book1.getPrice() + book2.getPrice();
assertEquals(expectedBalance,
   cart.getBalance());
assertEquals(2, cart.getItemCount() );
}
```

```
Public void testRemoveItem() throws
productNotFoundException {
    Cart.removeItem(book1); // sunny day scenario
    assertEquals(1, cart.getItemCount() );
}
Public void testRemoveItemNotInCart() {
try{
    Product book3 = new Product("title3", "10€");
    Cart.removeItem(book3); // rainy day scenario
    fail("should raise a ProductNotFoundException");
}        fail: JUnit keyword
Catch(ProductNotFoundException expected) {
    //passed the test!
}
} // of class ShoppingCartTest
```