DD2459 Software Reliability
Lecture 3

Black-box Testing

(see Amman and Offut, Chapters 3, 4, 6)

Part 1 Random, Boundary Value and N-wise Testing

Black-box Testing

Definition

A testing method is a black-box method if test cases are constructed without reference to the internal code structure

Advantages

+ Can focus on testing the requirements
+ Can overcome combinatorial explosions (maybe)
+ Complementary capabilities to glass-box testing
+ Insensitive to code refactoring (c.f. glass-box)

Black-box Disadvantages

- Slightly harder to find test verdicts (why?)
  - Aka. the oracle problem
- Much harder to define coverage
- Lacks some capabilities of glass-box testing

Five Black-box Methods

1. Random testing
2. Boundary value testing
3. N-wise and pairwise testing

   (combinatorial testing, Amman and Offutt Section 6.2)
4. Testing from use cases
   – (A&O Section 4.2.2)
5. A practical framework: JUnit
   – (A&O Section 3.2)

Method 1:
Random Testing

- Generate input vectors at random, load into SUT and observe results.
- Can be easy or hard to implement
  - Low level data types … e.g. `int` … easy
  - High level data types … e.g. `MyGraph` … hard
- High volume of test cases …
  - but is this good structural coverage?
- Good for low input dimension 1-5?
  - Poorer for high input dimension (c.f. low discrepancy sequences)

Technical Issues

- Oracle step is difficult to automate without precise code requirements
- Set up and tear down may need to be considered (random?)
- 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

Method 2:
Boundary Value Testing

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

  e.g. $\text{Boundary}_i = \{ \max^+, \max, \max^-, \min^-, \min, \min^+ \}$,
- Suppose k choices for each of N input variables $v_1, \ldots, v_N$
- Values might be chosen from user requirements?
- Test suite TS is all combinations of

  $\text{Boundary}_1, \ldots, \text{Boundary}_N$
- i.e. Cartesian product $TS = \text{Boundary}_1 \times \ldots \times \text{Boundary}_N$

Boundary Value Example

Suppose N = 3, k = 2, then $k^N = 2^3 = 8$. Let

$Boundary_1$ = { Mon, Sun }

$Boundary_2$ = { A, Z }

$Boundary_3$ = { 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)

}

Curse of Dimensionality

- Most programs take a large amount of input data, per execution.
- Define problem dimension D to be the total number of input variables to the SUT.
- Question: how to count structured variables e.g. arrays? (c.f. Lab 2)
- Example:
  - 32 bit integers,
  - 10 integer input variables
  - 2^{320} possible input values
- This is the curse of (all) high dimensional problems
- c.f. multivariate integration in physics!

Combinatorial Explosion

- Example:

  Cartesian product

  $$TS = Boundary_1 \times \ldots \times Boundary_N$$

- Test suite size is $|TS| = k^N$
- Faces an exponential explosion!
- Need combinatorial test techniques which select a subset of all possible combinations

Method 3: n-Wise Testing
a Combinatorial Technique

See also Ammann and Offutt Chapter 6.3.

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

For each input variable let

$$Typical_i = \{ t_1 , \ldots , t_k \}$$

be set of k typical values for variable v_i.

Choose one typical value t_j to be a default value

$def_i = t_j$ for v_i

Definitions

Definition 1. An n-wise test case is an input vector of typical values

$$(y_1, \ldots, y_M) \in \text{Typical}_1 \times \ldots \times \text{Typical}_M$$

containing exactly n non-default values.

Definition 2. 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 $tc \in TS$.

Example: 1-wise testing

- Choose n = 1, and we get 1-wise testing
- Choose M = 3 and we get a 3-dimensional testing problem.
- Choose defaults def_1 = Sat, def_2 = A, def_3 = 1, and k=2 typical values

  $Typical_1$ = { Sat, Sun }

  $Typical_2$ = { A, B }

  $Typical_3$ = { 1, 2 }

  Let TS1 = {(def_1, def_2, def_3) [0-wise]

  (Sun, def_2, def_3), (def_1, B, def_3), (def_1, def_2, 2) [1-wise]

  }

Clearly TS1 has size at most ((k-1)*M) +1 which is linear in M.

TS1 is small, but has limited coverage.

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

- Choose n = 2 and we get pairwise testing
- For the same parameters as above we get …

TS2 =

{ (def_1, def_2, def_3), [0-wise]
(Sun, def_2, def_3), (def_1, B, def_3), (def_1, def_2, 2), [1-wise]
(Sun, B, def_3), (Sun, def_2, 2), (def_1, 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 $O(M^2)$, which is still much slower than k^M.

Test Compaction

Important Note: many authors recommend test compaction

This means putting several n-wise tests into same test vector if M is large enough, (i.e. M >= 2n) e.g.

(t_1, t_2, def_3, def_4) and (def_1, def_2, t_3, t_4) two [2-wise] tests

can be compacted into

$(t_1, t_2\ t_3, t_4)$ (one test)

This disagrees with Definition 1 but is the same as just using Part 2 of Definition 2.

Greedy compaction algorithms are typically used, e.g. IPOG (In-Parameter-Order-General) - see DD2459 *Course Literature* web page

Is test compaction really *justified*?

Why Pairwise Testing?
Optimal n-value giving good coverage

- All references from www.pairwise.org
- Bugs involving interactions between three or more parameters are progressively less common[2].
- 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].
- Medical software devices. Only 3 of 109 failure reports indicated that more than two conditions were required to cause the failure [14].

Pairwise coverage?

- 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.
- 10 UNIX commands. Cohen et al. showed that the pairwise tests gave over 90 percent block coverage [9].
- So n = 2 seems to be an *optimal value*
- Open source tools, e.g. PICT

DD2459 Software Reliability
Lecture 3

Black-box Testing

(see Amman and Offut, Chapters 3, 4, 6)

Part 2 Use Case and Junit Testing

Method 4:
Test Cases from Use Cases

- Instantiate a use case model with concrete data values, and expected results.
- Different flows lead to different use case stories
  - Sunny day and rainy day stories
- Use graph coverage to measure use case coverage
- Structured and easy to use
- Sequential test cases (a time dimension exists)
- Stories are event driven rather than data driven
- Natural focus on most significant use cases
- Good approach to *system* and *acceptance testing*
- Difficult for *unit* and *integration testing*, if models are lacking

UseCaseName: PurchaseTicket

Precondition: The passenger is standing in front of ticket distributor and has sufficient money to purchase a ticket. (Needs a setup sequence?)

Event 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
5. If the passenger inserted more money than the amount due the distributor returns excess change.
6. The distributor issues ticket.
7. The passenger picks up the change and ticket.

Postcondition (test oracle for verdict)

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

Event 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 (i.e. event labels) 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.

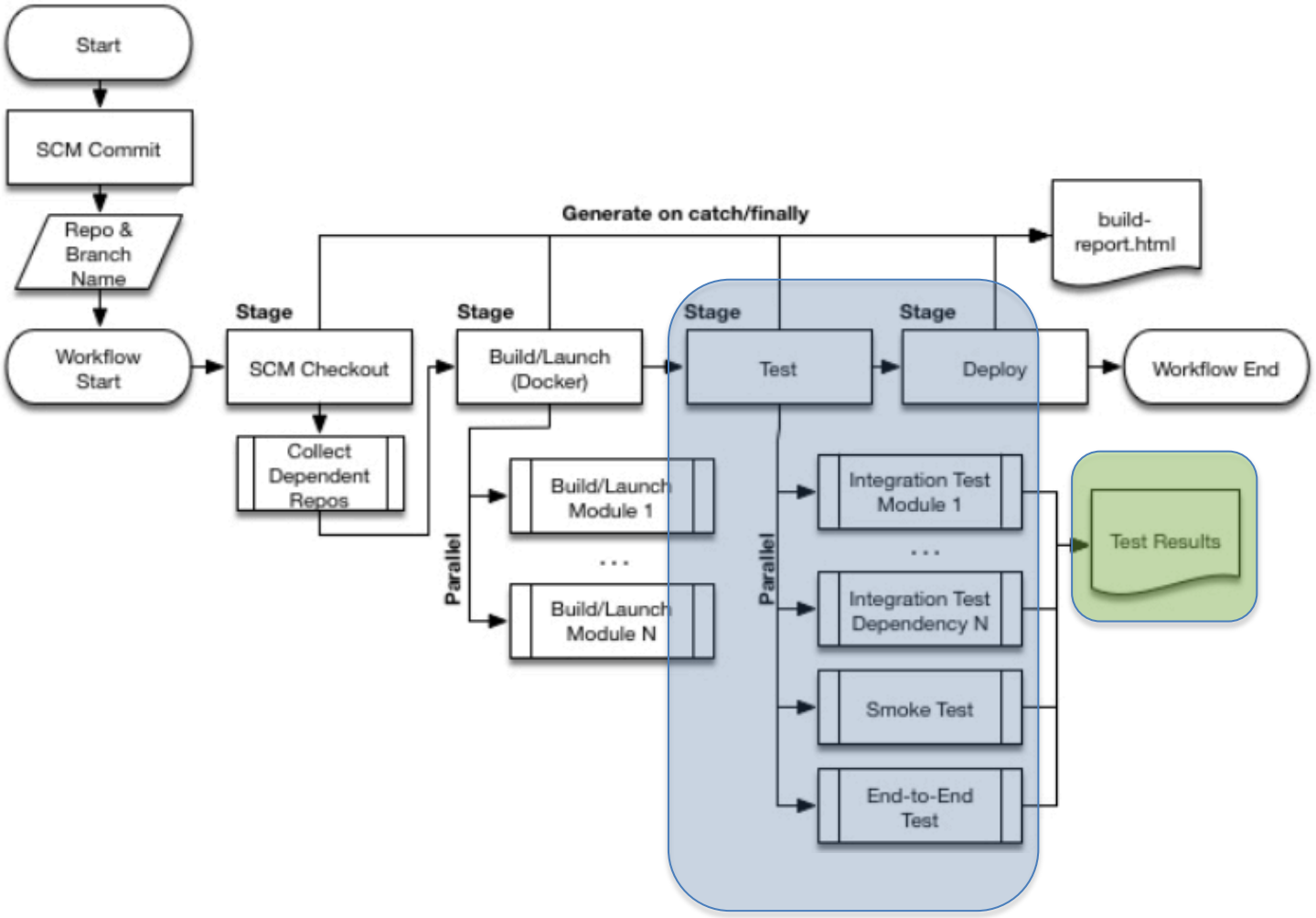
Method 5:
Unit Testing with JUnit

- See Ammann and Offutt Chapter 3.3
- Developed by the XP community 2002
- Framework for automating the execution of unit tests for Java classes
- Write new test cases by subclassing the TestCase class
- Organise TestCases into TestSuites
- Automates black-box unit testing

Why use JUnit?

- JUnit tightly integrates development and testing
- Can refactor code without worrying about correctness
- JUnit is simple.
  - Easy as running the compiler on your code

XUnit in DevOps (Jenkins)

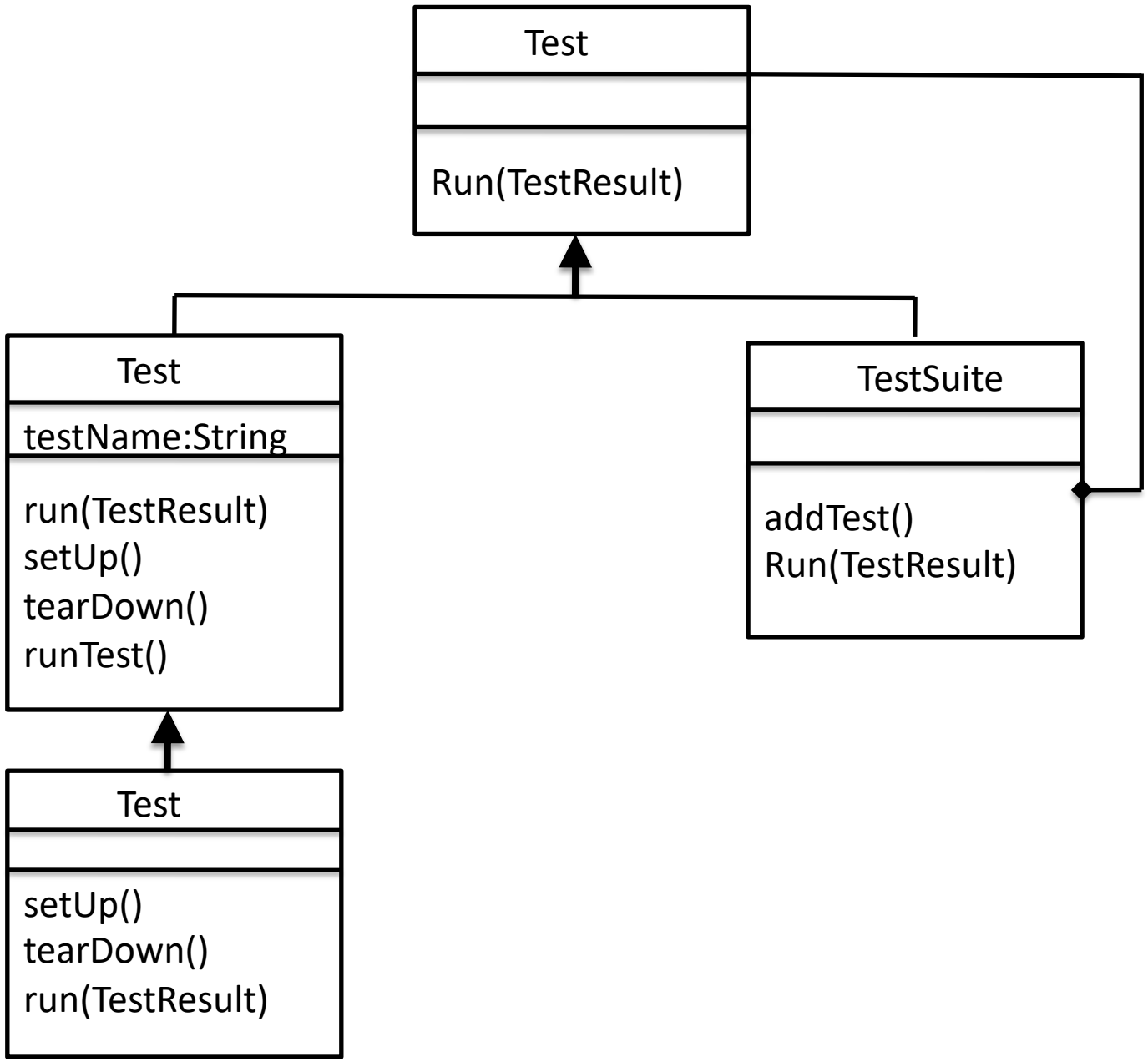


- JUnit tests check their own results
- Oracle step provides immediate feedback
  - No manual comparison of expected with actual
  - Simple visual feedback
- JUnit tests can be composed into a hierarchy of test suites
  - Can run tests from any layer in the hierarchy
- Writing JUnit tests is inexpensive
  - No harder than writing a method to exercise code

- 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

- `MyTestingClass` subclasses `TestCase`
- `MyTestingClass` has public `testXXX()` methods
- Compare expected output against actual output using `assert()` method (oracle step)
- Use `setUp()` and `tearDown()` to prevent side effects between subsequent `testXXX()` calls (i.e test infection)


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.
- Automatic invocation of all `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

JUnit Example

```
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)
}
```

```
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");
   }
Catch(ProductNotFoundException expected) {
   //passed the test!
}
} //of class ShoppingCartTest
```

Lecture Summary

- Have seen advantages and disadvantages of black-box testing (compared with glass-box)
- Have seen 5 black-box testing methods
  - Random testing
  - Boundary value testing
  - N-wise and pairwise testing
  - Testing from use case models
  - JUnit
- Have considered related coverage concepts and test suite size