Android Application Design

Software System Design

Zhu Hongjun

Session 6: Testing

- Basics
- Unit Testing
- Monkey Runner
- Test Case Design
- Conclusions

Basics

- Software Testing
  - is any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results
  - is the process of executing a program or system with the intent of finding errors
  - is more than just debugging
  - is a trade-off between budget, time and quality

Basics

- Software Quality Factors
  - exterior quality (Functionality)
    - Correctness, Reliability, Usability, Integrity
  - interior quality (Engineering)
    - Efficiency, Testability, Documentation, Structure
  - future quality (Adabtability)
    - Flexibility, Reusability, Maintainability

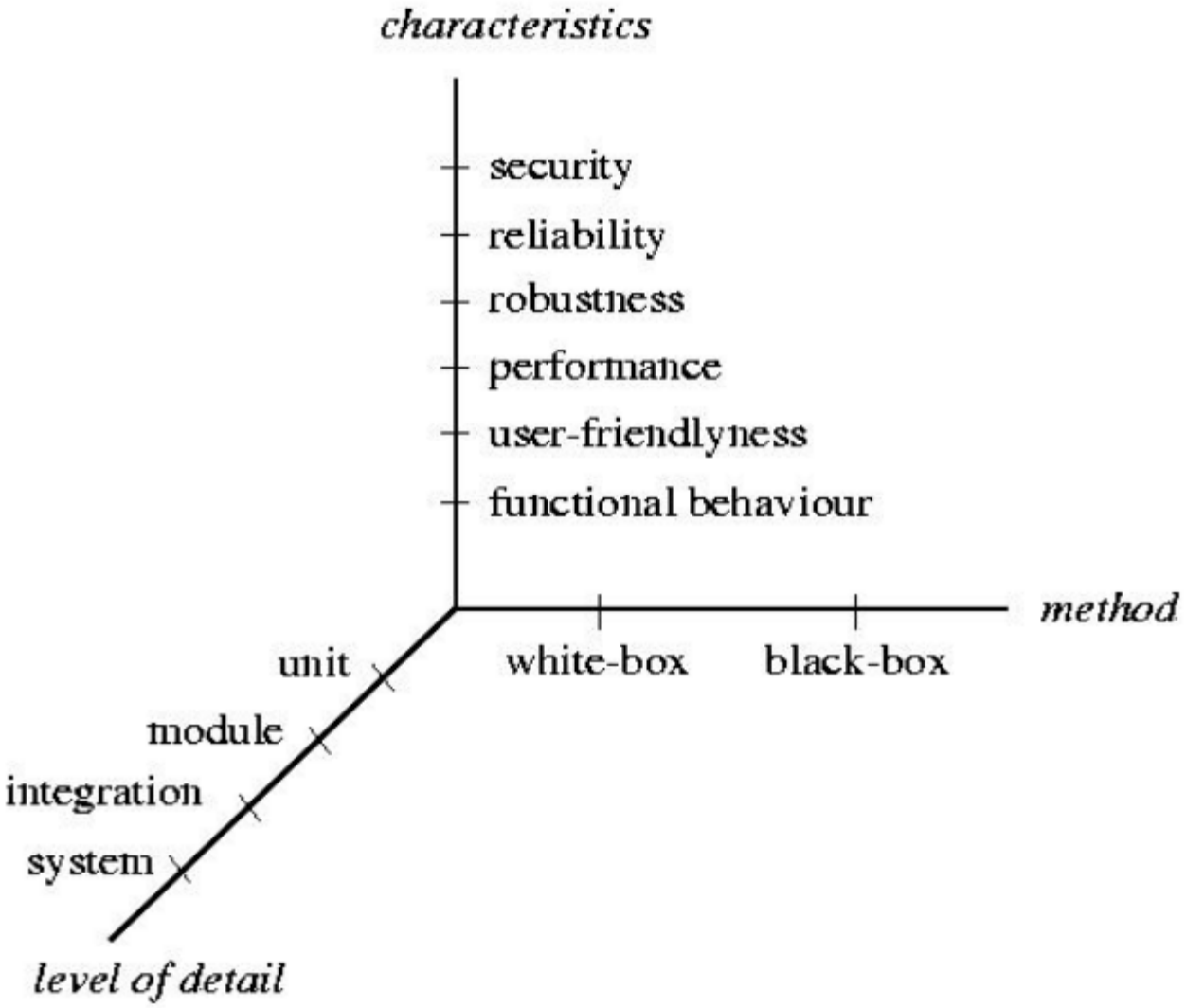
Basics

- Defects and failures
  - a defect/fault is defined as an abnormal condition or defect at the component, equipment, or sub-system level which may lead to a failure
  - not all defects will necessarily result in failures
  - not all software defects are caused by coding errors
    - One common source of expensive defects is requirement gaps
    - Unlike most physical systems, most of the defects in software are design errors

Basics

- Software Testing can be Classified
  - by purpose
    - correctness testing, performance testing, reliability testing and security testing
  - by life-cycle phase
    - requirements phase testing, design phase testing, program phase testing, evaluating test results, installation phase testing, acceptance testing and maintenance testing
  - by scope
    - unit testing, component testing, integration testing, and system testing

Types of testing

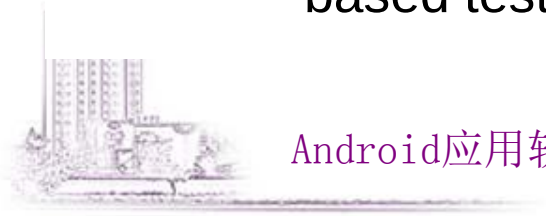


Basics

- Correctness testing and reliability testing are two major areas of testing
  - correctness testing
    - is the minimum requirement of software, the essential purpose of testing
  - reliability testing
    - software reliability refers to the probability of failure-free operation of a system

Basics

- Testing Approaches
  - Black-box testing
    - is a testing method in which test data are derived from the specified functional requirements without regard to the final program structure
    - also termed data-driven, input/output driven, or requirements-based testing
  - White-box testing
    - the structure and flow of the software under test are visible to the tester
    - also called glass-box testing, logic-driven testing or design-based testing



Basics

- Black-box testing
  - can be applied to virtually every level of software testing: unit, integration, system and acceptance
  - test cases are built around specifications and requirements
  - test design techniques include:
    - Equivalence partitioning, Boundary value analysis, etc.

Basics

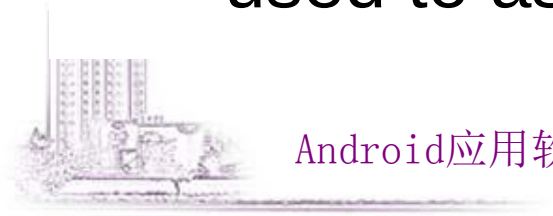
- White-box testing
  - can be applied at the unit, integration and system levels of the software testing process
  - in white-box testing an internal perspective of the system, as well as programming skills, are used to design test cases
  - test design techniques include:
    - Control flow testing, Data flow testing, etc.

Unit Testing

- It is a software testing method by which
  - individual units of source code,
  - sets of one or more computer program modules together with associated control data,
  - usage procedures,
  - operating procedures
- are tested to determine if they are fit for use

Unit Testing

- In object-oriented programming, a unit is often an entire interface, such as a class, but could be an individual method
- Ideally, each test case is independent from the others
  - substitutes such as method stubs, mock objects, fakes, and test harnesses can be used to assist testing a module in isolation



Unit Testing

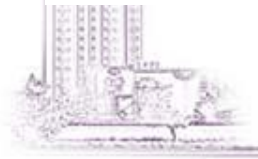
- Unit tests are typically written and run by software developers to ensure that code meets its design and behaves as intended
- Unit testing will not catch every error in the program
- Unit testing takes time and its investment may not be worth the effort
- Unit testing is difficult to set up realistic and useful tests

Unit Testing

- Unit Testing Frameworks
  - help simplify the process of unit testing, having been developed for a wide variety of languages
    - JUnit
      - a unit testing framework for the Java programming language
      - It is an instance of the xUnit architecture for unit testing frameworks

Unit Testing

- Test Runner
  - is an executable program that runs tests implemented using an xUnit framework and reports the test results
- Test Case
  - is the most elemental class. All unit tests are inherited from here
- Test Suite
  - is a set of tests that all share the same fixture
- Assertions
  - is a function or macro that verifies the behavior (or the state) of the unit under test



Unit Testing

- Test Fixture
  - is all the things that must be in place in order to run a test and expect a particular outcome
  - four phases of a test in xUnit
    - Set up -- Setting up the test fixture
    - Exercise -- Interact with the system under test
    - Verify -- Determine whether the expected outcome has been obtained
    - Tear down -- Tear down the test fixture to return to the original state

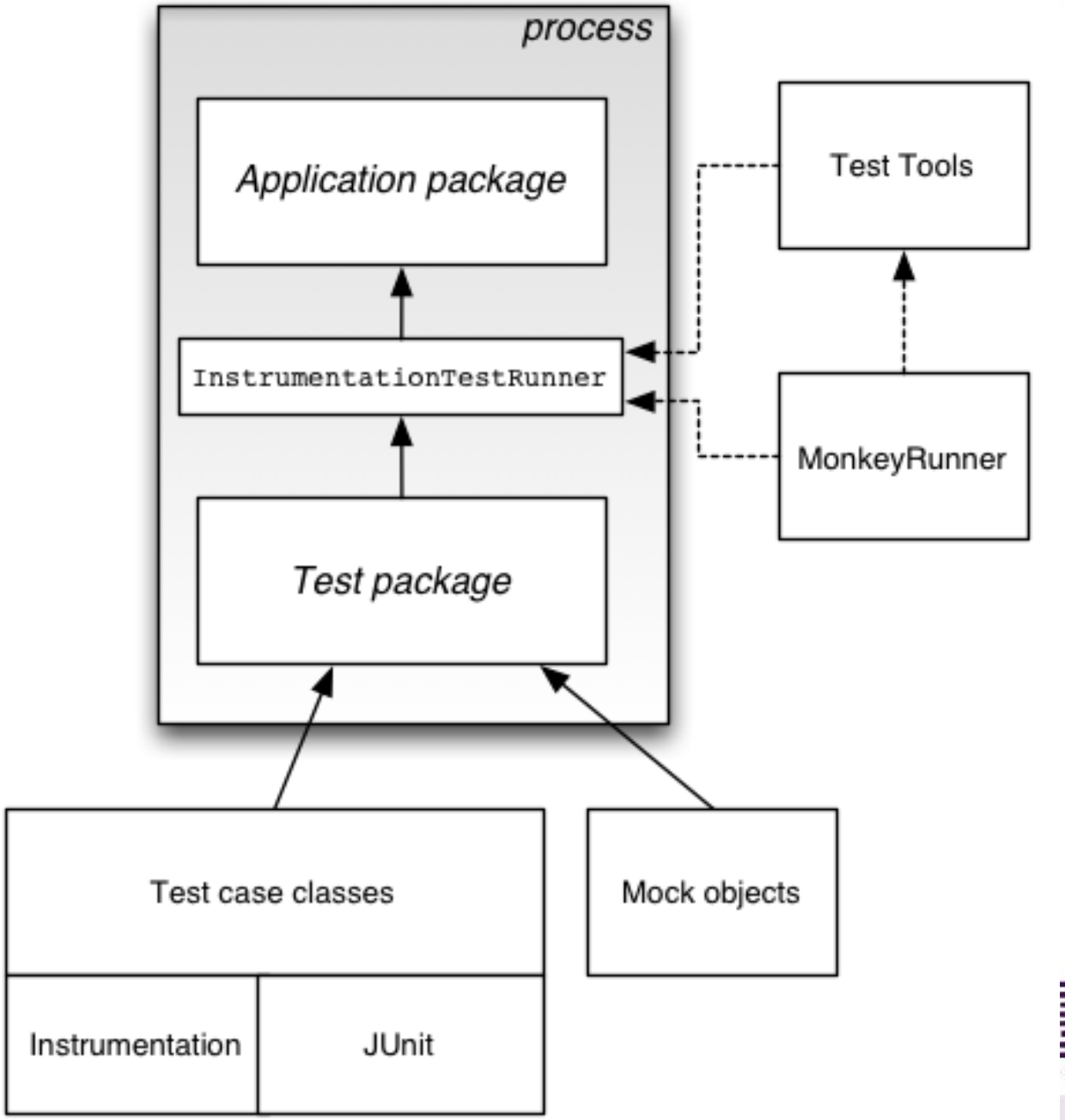
Example of JUnit test fixture

```
import org.junit.*;

public class TestFoobar {
    @BeforeClass
    public static void setUpClass() throws Exception {
        // Code executed before the first test method
    }

    @Before
    public void setUp() throws Exception {
        // Code executed before each test
    }

    @Test
    public void testOneThing() {
        // Code that tests one thing
    }

    @Test
    public void testAnotherThing() {
        // Code that tests another thing
    }

    @Test
    public void testSomethingElse() {
        // Code that tests something else
    }

    @After
    public void tearDown() throws Exception {
        // Code executed after each test
    }

    @AfterClass
    public static void tearDownClass() throws Exception {
        // Code executed after the last test method
    }
}
```

Unit Testing

- Android application testing framework
  - Android test suites are based on JUnit
  - You can use plain JUnit to test a class that doesn't call the Android API
  - Or Android's JUnit extensions to test Android components
  - The Android JUnit extensions provide component-specific test case classes

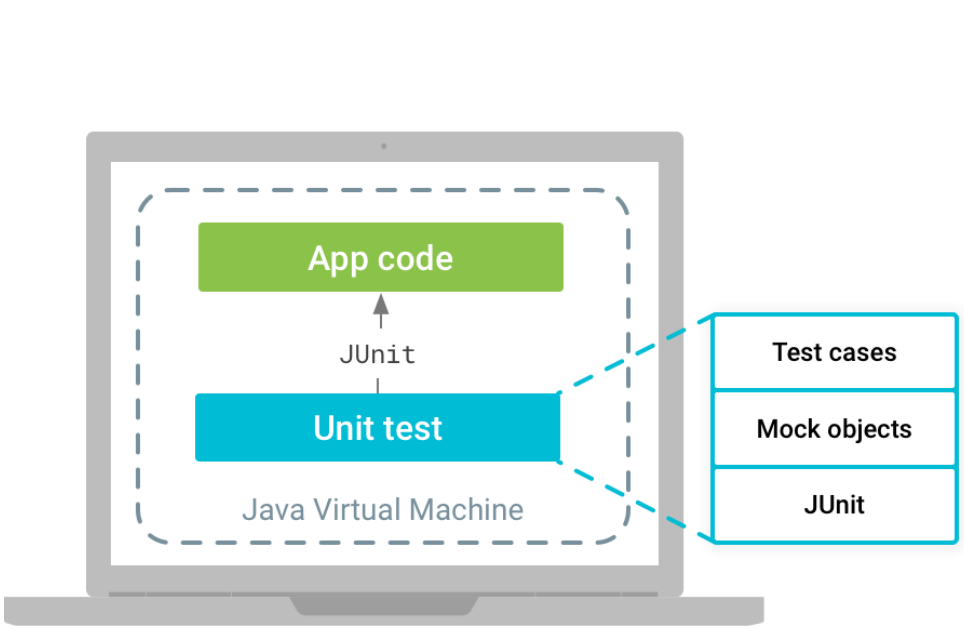
Android application testing framework



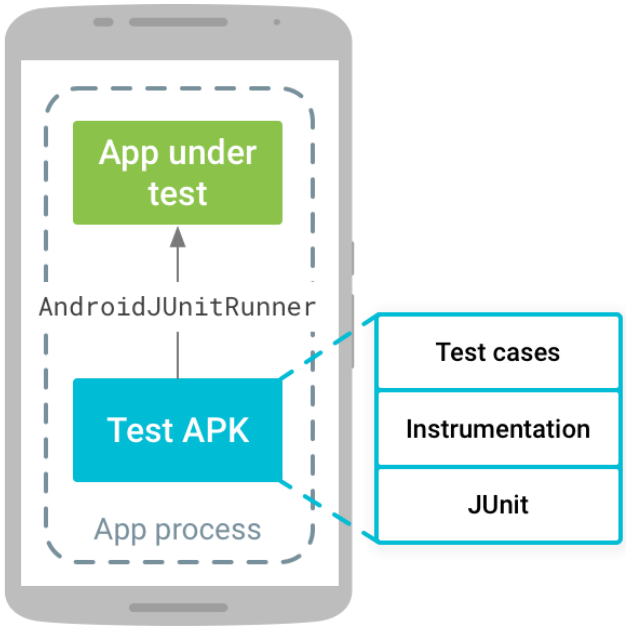
Unit Testing

- Local Unit Tests
  - run on the local JVM and do not have access to functional Android framework APIs
- Instrumented Tests
  - all tests that must run on an Android hardware device or an Android emulator

Android application testing framework

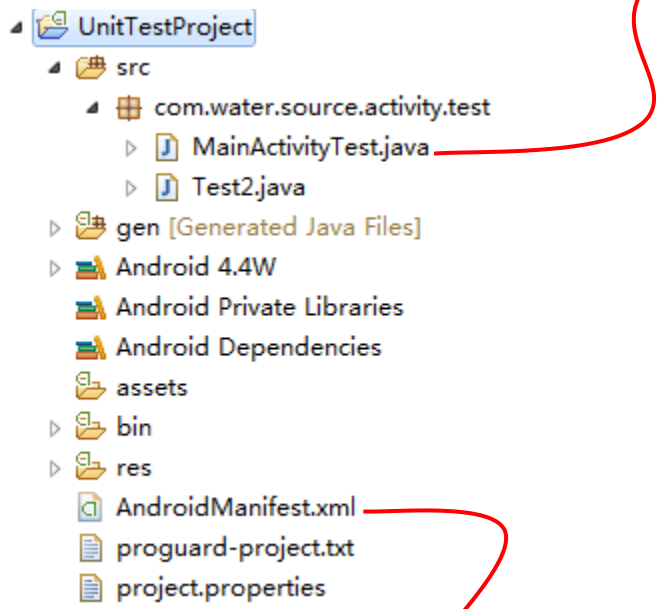


Local unit test
src/test/java/



Instrumented test
src/androidTest/java/

Activity testing




```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.water.source.activity.test"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk android:minSdkVersion="8" />

    <instrumentation
        android:name="android.test.InstrumentationTestRunner"
        android:targetPackage="com.water.source.activity" />

    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name" >
        <uses-library android:name="android.test.runner" />
    </application>

</manifest>
```

```java
public class MainActivityTest extends
        ActivityInstrumentationTestCase2<MainActivity> {

    private MainActivity mActivity = null;

    private TextView mtv = null;

    private Instrumentation mIns = null;

    public MainActivityTest() {
        super(MainActivity.class);
        // TODO Auto-generated constructor stub
    }

    @Override
    protected void setUp() throws Exception {
        // TODO Auto-generated method stub
        super.setUp();
        mIns = getInstrumentation();
        this.setActivityInitialTouchMode(false);
        mActivity = this.getActivity();
        mtv = (TextView) mActivity
                .findViewById(com.water.source.activity.R.id.tv);
    }

    public void testTextViewInit() {
        getActivity().runOnUiThread(new Runnable() {
            @Override
            public void run() {
                mtv.setText("teste");
            }

        });
        mIns.waitForIdleSync();
        assertEquals(mtv.getText(), "teste");
    }

    @Override
    protected void tearDown() throws Exception {
        // TODO Auto-generated method stub
        super.tearDown();
        mtv = null;
        mActivity = null;
        mIns = null;
    }
}
```

Monkey Runner

- The monkeyrunner tool provides an API for writing programs that control an Android device or emulator from outside of Android code
- With monkeyrunner, you can write a Python program that
  - installs an Android application or test package
  - runs an application or test package
  - sends keystrokes to an application
  - takes screenshots of an user interface and stores screenshots on the workstation

Monkey Runner

- The monkeyrunner tool provides
  - Multiple device control
  - Functional testing
  - Regression testing
  - Extensible automation
- The monkeyrunner tool uses Jython, a implementation of Python that uses the Java programming language

Monkey Runner

- The monkeyrunner API
  - Monkey Runner
    - a class of utility methods for monkeyrunner programs
    - provides a method for connecting monkeyrunner to a device or emulator
  - Monkey Device
    - represents a device or emulator
    - provides methods for installing and uninstalling packages, starting an Activity, and sending keyboard or touch events to an application
  - Monkey Image
    - represents a screen capture image

A Simple monkeyrunner Program

```
# Imports the monkeyrunner modules used by this program
from com.android.monkeyrunner import MonkeyRunner, MonkeyDevice

# Connects to the current device, returning a MonkeyDevice object
device = MonkeyRunner.waitForConnection()

# Installs the Android package. Notice that this method returns a boolean, so you can test
# to see if the installation worked.
device.installPackage('myproject/bin/MyApplication.apk')

# sets a variable with the package's internal name
package = 'com.example.android.myapplication'

# sets a variable with the name of an Activity in the package
activity = 'com.example.android.myapplication.MainActivity'

# sets the name of the component to start
runComponent = package + '/' + activity

# Runs the component
device.startActivity(component=runComponent)

# Presses the Menu button
device.press('KEYCODE_MENU', MonkeyDevice.DOWN_AND_UP)

# Takes a screenshot
result = device.takeSnapshot()

# Writes the screenshot to a file
result.writeToFile('myproject/shot1.png','png')
```

Test Case Design

- A test case, in software engineering, is a set of conditions or variables under which a tester will determine whether an application, software system or one of its features is working as it was originally established for it to do
  - http://en.wikipedia.org/wiki/Test_case

Test Case Design

- Test Case acts as the starting point for the test execution,
- and after applying a set of input values, the application has a definitive outcome
- and leaves the system at some end point or also known as execution postcondition

Test Case Design

- Typical Test Case Parameters

- test case ID
- test case description
- test step or order of execution number
- related requirement(s)
- depth
- test category
- author
- check boxes for whether the test can be or has been automated
- pass/fail
- remarks

- Test Case ID
- Test Scenario
- Test Case Description
- Test Steps
- Prerequisite
- Test Data
- Expected Result
- Test Parameters
- Actual Result
- Environment Information
- Comments

Test case example

Scenario	Test Step	Expected Result	Actual Outcome
Verify that the input field that can accept maximum of 10 characters	Login to application and key in 10 characters	Application should be able to accept all 10 characters.	Application accepts all 10 characters.
Verify that the input field that can accept maximum of 11 characters	Login to application and key in 11 characters	Application should NOT accept all 11 characters.	Application accepts all 10 characters.

Test case example

Test Suite ID	TS001
Test Case ID	TC001
Test Case Summary	To verify that clicking the Generate Coin button generates coins.
Related Requirement	RS001
Prerequisites	1. User is authorized. 2. Coin balance is available.
Test Procedure	1. Select the coin denomination in the Denomination field. 2. Enter the number of coins in the Quantity field. 3. Click Generate Coin.
Test Data	1. Denominations: 0.05, 0.10, 0.25, 0.50, 1, 2, 5 2. Quantities: 0, 1, 5, 10, 20
Expected Result	1. Coin of the specified denomination should be produced if the specified Quantity is valid (1, 5) 2. A message 'Please enter a valid quantity between 1 and 10' should be displayed if the specified quantity is invalid.
Actual Result	1. If the specified quantity is valid, the result is as expected. 2. If the specified quantity is invalid, nothing happens; the expected message is not displayed
Status	Fail
Remarks	This is a sample test case.
Created By	John Doe
Date of Creation	01/14/2020
Executed By	Jane Roe
Date of Execution	02/16/2020
Test Environment	▪ OS: Windows Y ▪ Browser: Chrome N

Test Case Design

- Conventional Black-Box Testing
  - Specification of test cases and tests

#	Description	Input	Expected Output
1	test for empty list	an empty list	same as input
2	test for single element list	a single element list	same as input
3	test for duplicate elements	a list containing duplicate elements	the list with all duplicate elements removed
4	test for no duplicate element	a list in which all elements are distinct	same as input

(a) Purge program test case specification

#	Description	Input	Expected Output (EO)	Actual Output (AO)	Passing Criteria	Test Result
1	test for empty list	()	()	TBD	EO=AO	TBD
	test for single element list	(10)	(10)	TBD	EO=AO	TBD
	test for duplicate elements	(1,2,2,3,4,4,5)	(1,2,3,4,5)	TBD	EO=AO	TBD
	test for no duplicate element	(1,2,3,4,5)	(1,2,3,4,5)	TBD	EO=AO	TBD

(b) Tests produced from the test case specification

Test Case Design

- Equivalence Partitioning
  - divies the input and output domains into a number of disjoint subsets
  - and select one test case from each of these disjoint subsets
  - the key is to identify an equivalence relation among the elements of an input domain

Rules of partitioning the input space

Case #	Input / Output		Partitions		Example	
	Type	Domain	Valid	Invalid	Valid	Invalid
1	Numeric	a range of values [n1, n2]	[n1, n2]	[-MIN, n1-1], [n2+1, MAX]	[1, 12]	[-MIN, 0], [13, MAX]
	Nonnumeric	a range of consecutive strings S1-S2	S1-S2	All strings except S1-S2	CS4300-CS4399	All strings except CS4300-CS4399
2	Numeric	ranges of values [n1, n2], [n3, n4], $n_i<n_{i+1}$	[n1, n2], [n3, n4]	[-MIN, n1-1], [n2+1, n3-1], [n4+1, MAX]	[1, 5], [8, 12]	[-MIN, 0], [6, 7], [13, MAX]
	Nonnumeric	ranges of consecutive strings S1-S2, S3-S4	S1-S2, S3-S4	All strings except S1-S2, S3-S4	CS430-CS439, CS450-CS459	All strings except CS430-CS439, CS450-CS459
	Nonnumeric	regular sets	regular sets	All strings except elements in the regular sets	CS4[35][0-9]	All strings except elements in CS4[35][0-9]
3	Numeric	a single value n	[n, n]	[-MIN, n-1], [n+1, MAX]	[3.0, 3.0]	[-MIN, 3.0), (3.0, MAX]
	Nonnumeric	a single string S	{S}	All strings except S	"USA"	All strings except "USA"
4	Numeric	an enumeration of discrete values {n1, ..., nk}	[n1, n1], ..., [nk, nk]	All integers except n1, ..., nk	{1, 5, 10, 25}/ [1, 1], [5, 5], [10, 10], [25, 25]	All integers excluding 1, 5, 10, 25
	Nonnumeric	an enumeration of strings {S1, ..., Sn}	{S1}, ..., {Sn}	All strings except S1, ..., Sn	{"jan", ..., "dec"}/ {"jan"}, ..., {"dec"}	All strings except "jan", ..., "dec"
5	Boolean	{True, False}	{True}, {False}		{True}, {False}	

Test Case Design

- Boundary Value Analysis
  - selects test cases at and near the boundaries of the equivalence classes

Case #	Input / Output		Test Cases
	Type	Domain	
1	Numeric	a range of values [n1, n2]	n1-1, n, n1+1, n2-1, n2, n2+1
	Nonnumeric	a range of consecutive strings S1-S2	S1, S2, null string, empty string, very long string
2	Numeric	ranges of values [n1, n2], [n3, n4], $n_i<n_{i+1}$	same as 1 but for each range
	Nonnumeric	ranges of consecutive strings S1-S2, S3-S4	S1, S2, S3, S4, null string, empty string, very long string
3	Numeric	a single value n	n-1, n, n+1
	Nonnumeric	a single string S	S, null string, empty string, very long string
4	Numeric	an enumeration of discrete values {n1, ..., nk}	nj-1, nj, nj+1, j=1, 2, ..., k
	Nonnumeric	an enumeration of strings {S1, ..., Sn}	S1, ..., Sn, null string, empty string, very long string
		a set of strings	null string, empty string, one-char string, very long string
5	Boolean	{True, False}	{True}, {False}
6	Container	instances of Container	(1) a null container reference, (2) an empty container, (3) a one element container, (4) one less than the maximum size of the container, (5) maximum number of elements, (6) if possible, one more element than the maximum size of container, (7) boundary test cases for container elements.

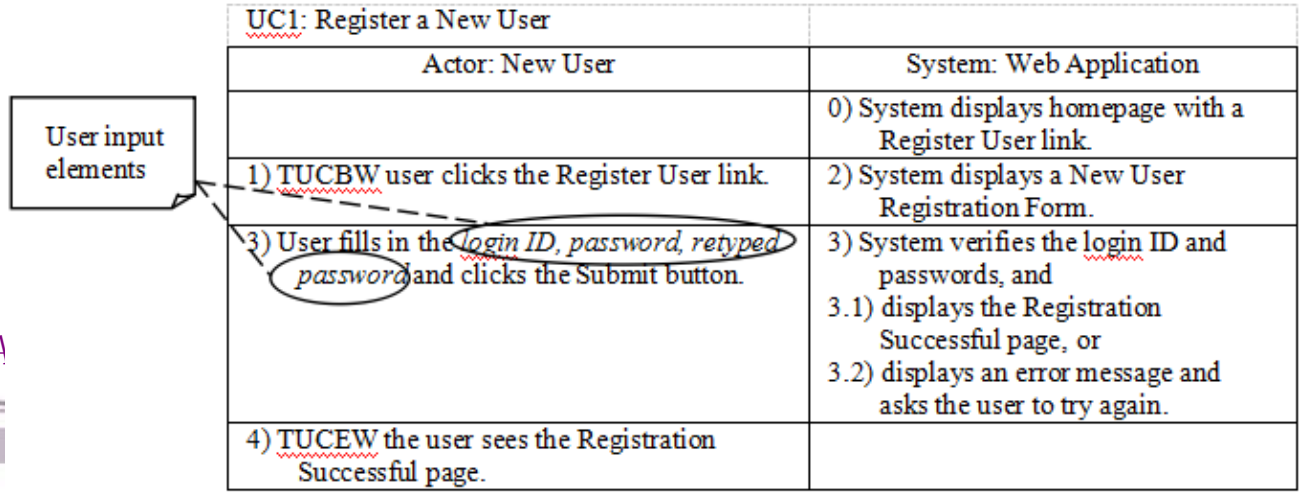
Test Case Design

- Decision Table
  - also referred to as a ’cause-effect’ table
    - provides a systematic way of stating complex business rules

Conditions	Rule 1	Rule 2	Rule 3	Rule 4	Rule 5	Rule 6	Rule 7	Rule 8
New customer (15%)	T	T	T	T	F	F	F	F
Loyalty card (10%)	T	T	F	F	T	T	F	F
Coupon (20%)	T	F	T	F	T	F	T	F
Actions								
Discount (%)	X	X	20	15	30	10	20	0

Test Case Design

- Use case-based testing
  - drives test cases from the use case specifications
    - identify actor input and actor actions
    - determine input values
    - generate test cases



UC1: Register a New User

Actor: New User	System: Web Application
	0) System displays homepage with a Register User link.
1) TUCBW user clicks the Register User link.	2) System displays a New User Registration Form.
3) User fills in the *login ID, password, retyped password* and clicks the Submit button.	3) System verifies the login ID and passwords, and 3.1) displays the Registration Successful page, or 3.2) displays an error message and asks the user to try again.
4) TUCEW the user sees the Registration Successful page.	

Identifying input values for use case-based testing

Input Element	Type	Value Specification	Valid	Invalid	Exceptional Cases
login ID	string	length must be between 8 and 20 characters	valid login not exists in system	invalid login, or login already exists in system	• strings with length=0, 1, or a very large number; or • strings with one or more spaces, control characters, or special characters
password	password	length must be between 8 and 12 characters, and contain at least one alphabet, numeric, and special character	password satisfies the password rule on the left	password not satisfies the password rule	• passwords with length=0, 1, or a very large number; • passwords contain one or more spaces or control characters
retyped password	password	same as password	match with password	retyped password does not match password, or is entered using copy-and-paste	

Test case generation for use case-based testing

Test Case	Login ID	Password	Retyped Password	Expected Outcome
1	Valid	Valid	Valid	show Registration Successful page
2	Valid	Valid	Invalid	show Error Message
3	Valid	Invalid	Valid	show Error Message
~~4~~	~~Valid~~	~~Invalid~~	~~Invalid~~	
5	Valid	Exceptional	Valid	show Error Message
~~6~~	~~Valid~~	~~Exceptional~~	~~Invalid~~	
7	Invalid	Valid	Valid	show Error Message
~~8~~	~~Invalid~~	~~Valid~~	~~Invalid~~	
~~9~~	~~Invalid~~	~~Invalid~~	~~Valid~~	
~~10~~	~~Invalid~~	~~Invalid~~	~~Invalid~~	
~~11~~	~~Invalid~~	~~Exceptional~~	~~Valid~~	
~~12~~	~~Invalid~~	~~Exceptional~~	~~Invalid~~	
13	Exceptional	Valid	Valid	show Error Message
~~14~~	~~Exceptional~~	~~Valid~~	~~Invalid~~	
~~15~~	~~Exceptional~~	~~Invalid~~	~~Valid~~	
~~16~~	~~Exceptional~~	~~Invalid~~	~~Invalid~~	
~~17~~	~~Exceptional~~	~~Exceptional~~	~~Valid~~	
~~18~~	~~Exceptional~~	~~Exceptional~~	~~Invalid~~	

Use case-based tests for register new user example

Test Case	Login ID	Password	Retyped Password	Expected Result
1	"newuser@hmail.com"	"xft123%PLM"	"xft123%PLM"	show Registration Successful page
2	"newuser@hmail.com"	"xft123%PLM"	"yyyyyyyy"	show Error Message
3	"newuser@hmail.com"	"zzzzzz"	"xft123%PLM"	show Error Message
5	"newuser@hmail.com"	"x"	"xft123%PLM"	show Error Message
7	"olduser@hmail.com"	"xft123%PLM"	"xft123%PLM"	show Error Message
13	"new user@hmail.com"	"xft123%PLM"	"xft123%PLM"	show Error Message

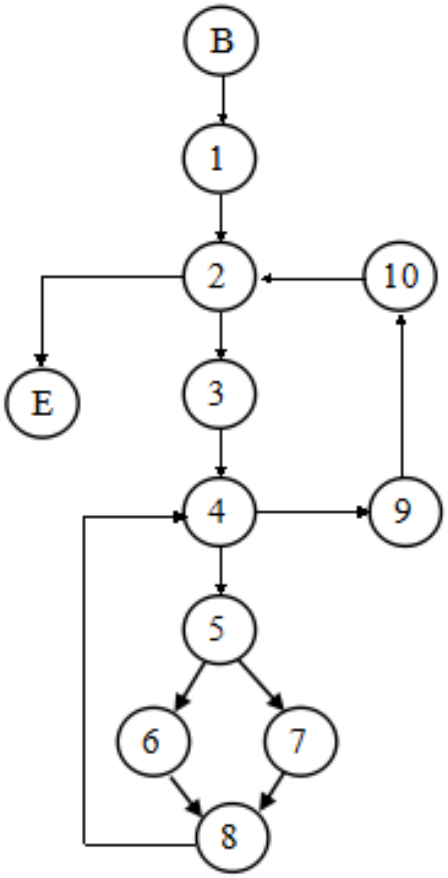
Test Case Design

- Conventional White-box Testing
  - control flow testing
  - data flow testing
  - branch testing
  - statement coverage
  - decision coverage
  - prime path testing
  - path testing

Purge program and its flow graph

```
      public void purge(LinkedList list) {
        Item p, q;
(1)     p=(Item)list.getFirst();
(2)     while (p != null) {
(3)       q=(Item)p.getNext();
(4)       while (q !=null) {
(5)         if (p.compareTo(q)==0)
(6)           list.remove (q);
(7)         else q=(Item)q. getNext ();
(8)       }
(9)       p=(Item)p. getNext ();
(10)    }
(11)  }
```

(a) the purge function

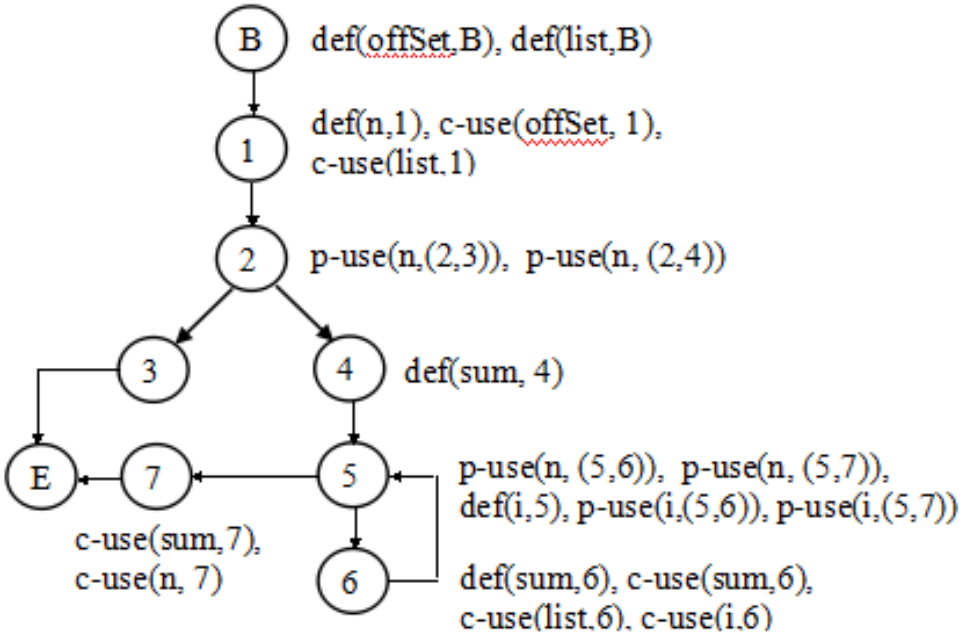


(b) flow graph for the purge program

Identifying data define-use

```
      public int average(int offSet, int[] list) {
(1)     int n=min(offSet, list.length);
(2)     if (n <= 0)
(3)       return 0;
(4)     int sum=0;
(5)     for (int i=0; i<n; i++)
(6)       sum += list[i];
(7)     return (int)(sum/n);
      }
```

(a) sample program



(b) identifying data define-use

Intraprocedural data flow define-use chains

c-use: computation use

p-use: predicate use

Variable	c-use	p-use
offSet	(B, 1)	
list	(B, 1), (B, 6)	
n	(1, 7)	(1, (2,3)), (1, (2,4)), (1, (5,6)), (1, (5,7))
sum	(4, 6), (6, 6), (6, 7)	
i	(5, 6)	(5, (5,6)), (5, (5,7))

Conclusions

- Basics
- Unit Testing
- Monkey Runner
- Test Case Design