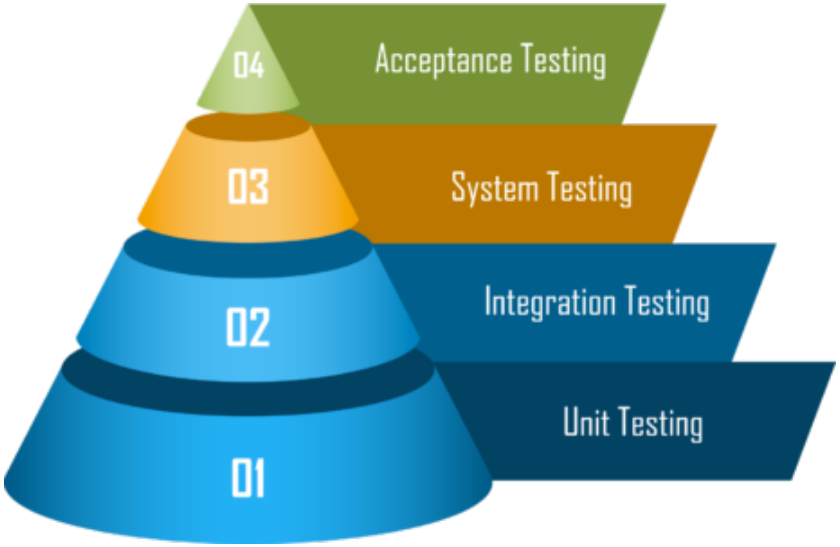
Florida Institute of Technology
SWE 5425
Advanced Software Testing

Software Testing Levels

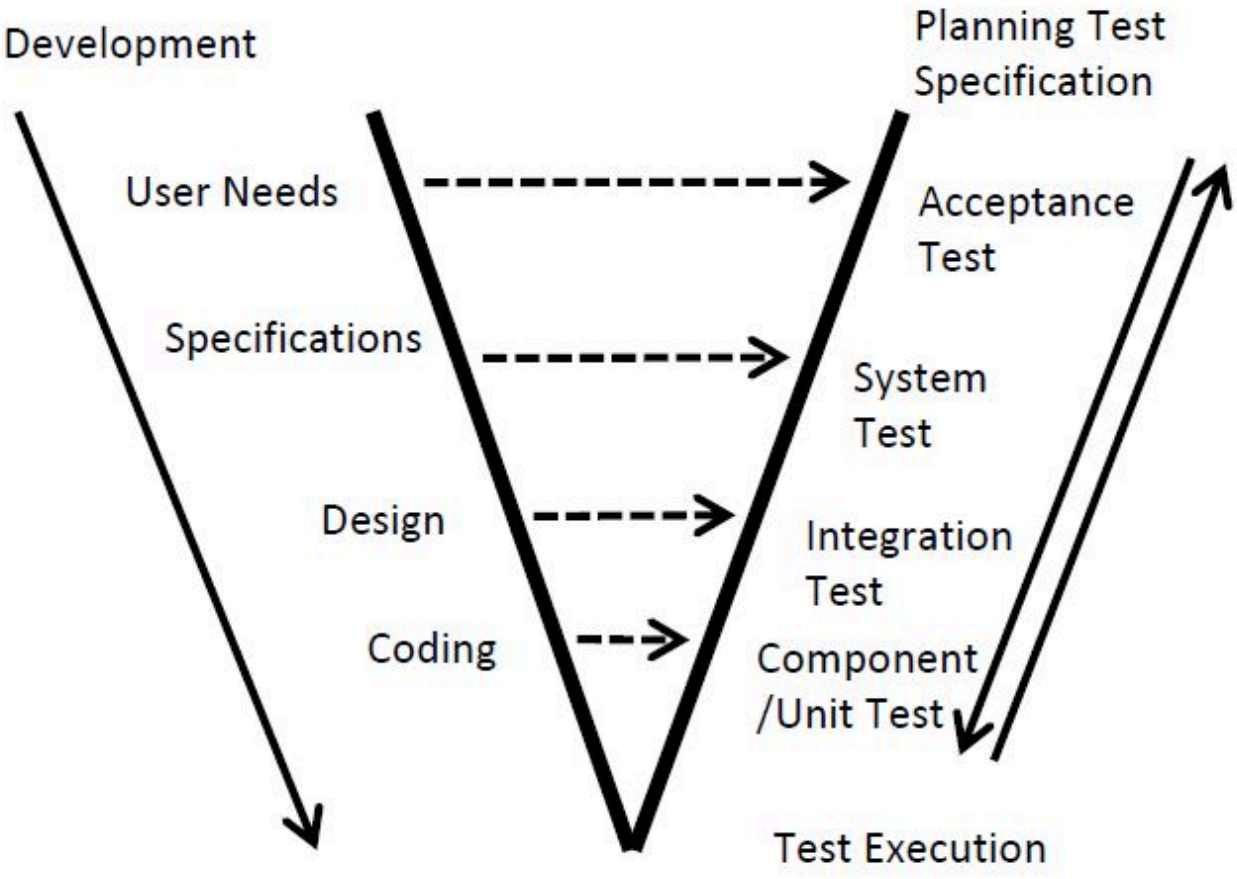


Lecturer: Khaled Slhoub, Ph.D.
Email: kslhoub@fit.edu

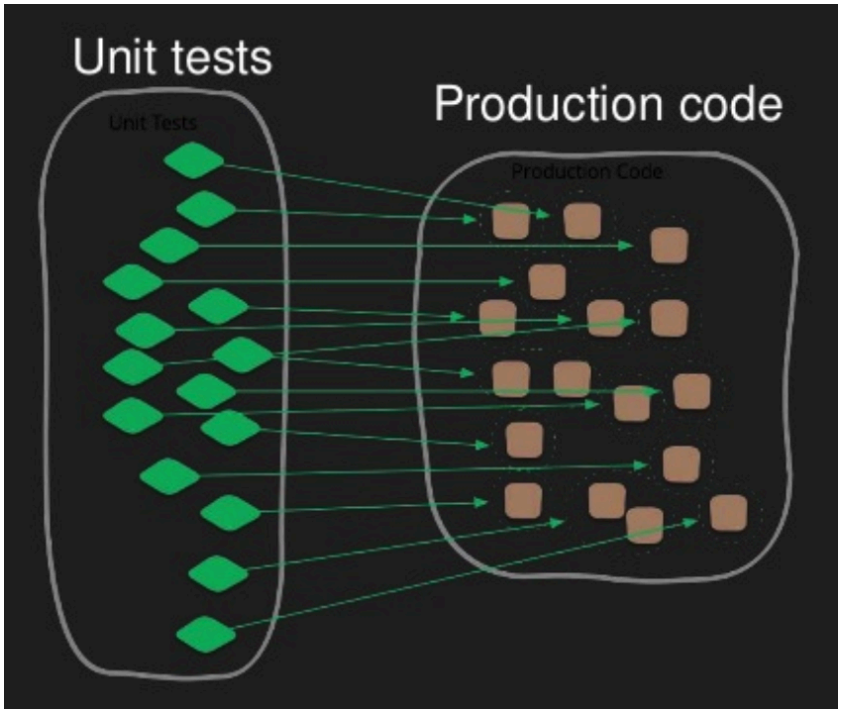
Software Testing Levels

- There are generally 4 different levels during the process of software testing:
  - **Unit Testing**
  - **Integration Testing**
  - **System Testing**
  - **Acceptance Testing**
- Software testing levels are the different stages of the software development lifecycle where testing is conducted.
- Each of these testing levels has a specific purpose and includes different methodologies and techniques that can be used while conducting software testing

The V model of software testing



Unit Testing



Unit Testing

- Generally, the code is written in component parts, or units. The units are usually constructed in **isolation**, for integration at a later stage. Units are also called modules or components.
- Unit testing is intended to ensure that the code written for the unit meets its specification, prior to its integration with other units. It verifies the unit behavior independently from other code units.
- A unit test is a piece of code that tests a unit of work – a method, class, or cluster of classes that implement a single logical operation
- Unit testing is an essential instrument in the toolbox of any serious software developer.

Unit Testing

- You need two types of information when designing unit testing: A specification and unit source code.
- Unit/Module/Component testing is largely white-box oriented.
- Unit tests have the following properties:
  - They are fully automated
  - They are self-verifying
  - They are repeatable and consistent
  - They test a single logical concept
  - They are fast

"The testing of individual software components." ***ISTQB***

Unit Testing Stages

- Arrange.

  arrange everything needed to run the unit test

- Act.

  the actual code unit (unit under test) is invoked by the unit test and the result is captured.

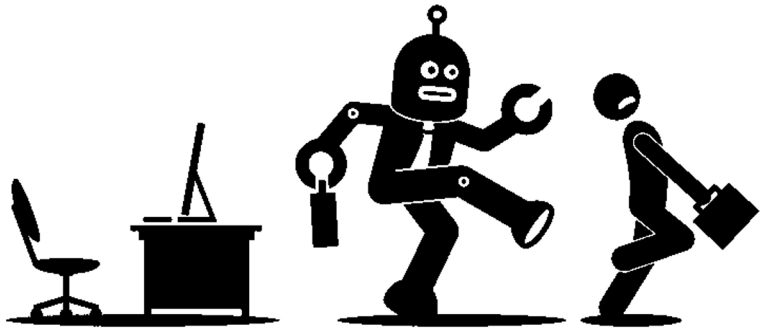
- Assert.

  The resulting behavior is observed. If it is consistent with the expectations, the unit test passes, otherwise, it fails

The three stages are simply called AAA

Software Automated Testing

- Why Software Testing?
  - Find and fix bugs
  - Analyze risk
  - Help judge product quality
  - Answer critical questions like:
    - Will customer requirements be met?
    - Will the business goals be met?
    - Is the product ready for shipping?



Manual Testing

- Performed by a human who carries out all the actions on the tested application manually

Manual Testing is hard and tedious job. Why?

- Takes much time to perform
- Consumes much effort
- Needs to high concentration
- Does not go in depth
- Allows for humans' mistakes
- Boring
- Cost much
- Less reliable
- Less efficient
- Needs skilled people
- Needs to write a lot

Automated Testing

- The use of software to control the execution of tests, the comparison of actual outcomes to predicted outcomes, the setting up of test preconditions, and other test control and test reporting functions
  - Reduces cost
  - Reduces human error
  - Reduces variance in test quality between different individuals
  - Significantly reduces the cost of regression testing

Automated Testing

Automated tests execute a sequence of actions without human intervention – Automated Testing is automating the manual testing process

- Requirements to write test scripts:
  - Detailed test cases (testing plan)
  - Test Database

  "Ad hoc testing cannot be automated"
- The main advantages is to *automate regression testing*
- Automation of the testing process is not only desirable, but in fact is a necessity

When Should Testing be Automated?

- Is automating this test case and running it once cheaper than simply running it manually?
- What is the lifetime of an automated test? Is this test likely to die sooner or later?
- What changes in software would cause that automated test be useless in the future?
- During the lifetime of an automated test, what are the possibilities that it will be able to find additional bugs?

Principal of Automated Testing

1. Speed

   - Manual testing takes long time execute a few hundred or thousand test cases?

   - Automated Testing run test cases 10, 100, or 1000 times faster

   Automation consumes much less time

Principal of Automated Testing

2. Efficiency

- Testers manually take longer to run test cases
  - Cannot complete
  - Cannot plan for more new test cases
- Automated Testing run test cases 10, 100, or 1000 times faster
  - Can complete the essential test cases
  - Move on to plan and execute more test cases
  - Automation covers more tested paths

Automation increases Productivity, and more Reliable and Dependable

Principal of Automated Testing

3. Accuracy and Precision

- Manual testing is an exhaustive process
  - Humans can get tired
  - Lose concentration
  - Make mistakes
- Automated Testing can solve these problems easily

Automation increases Accuracy

Principal of Automated Testing

4. Performance

- In manual testing, human capabilities are limited
  - Testers require a break after a while
  - Testers easily miss bugs
  - Testers cannot keep record for every test case
- Automated tools never get tired or give up
  - Test suites can easily keep records of every test case being executed

Automation increases Performance

Goals of Automated Testing

- Saves time and money
- Improves test accuracy
- Increases test coverage
- Performs tests not possible with manual testing
- Helps developers as well as testers
- More reliable
- Make life easier

- Does Automation Replace Testers?
  - An automated testing tool cannot replace testers
  - Increases productivity and efficiency

  "They just help software testers perform their job in better way" (Patton)

  e.g. Some tools do not generate test cases; they just execute them

- Knowing Limits While Automating?
  - Over-automation is trying to automate everything
    - Not finding bugs
    - Not adding any advantages of doing so
    - Cost-consuming
    - Hard to adapt to changes

Factors to Consider before Selecting a Test Automation Framework

1. Capability
   - Search for the tool that has all significant features
   - The tool must perform what vendors claim
2. Reliability
   - The tool works for long period
   - How this tool is reliable based on its popularity and customers feed back

"Many test tools are developed by small companies that do a poor job of testing them" (James)

3. Capacity
   - Can the tool handle large test scripts that run for hours?
   - Can work on one computer or many?
4. Learnability
   - The tool can be easily learned
   - Provides great help (manuals, online, and tutorials)
5. Operability
   - The tool communicates with users easily
   - Must be flexible to conform to changes

6. Performance
   - How quickly can one develop test cases?
   - How quickly can one get the results from test cases?
   - How much time will tests save?
7. Compatibility
   - What languages can the tool support?
   - What platforms/environments can the tool support

Skills That Testers Have to Learn

- The testers need to have basic knowledge of programming.
- Testers need to be familiar with the environment of the tool

"The tool is never as easy as the tool vendor claims it to be" (Dustin)

Challenges of Automated Testing

- The requirements are always changing, and new features are added very late
- There is no substitute for the human eye and intuition
- Developing an automated test is more time-consuming than running it once manually
- Humans can notice some bugs where automation tests cannot. Sometimes using a manual test is more effective
- Difficulties with automating the modern software applications (e.g. smart applications)
- Testing must be well organized and planned

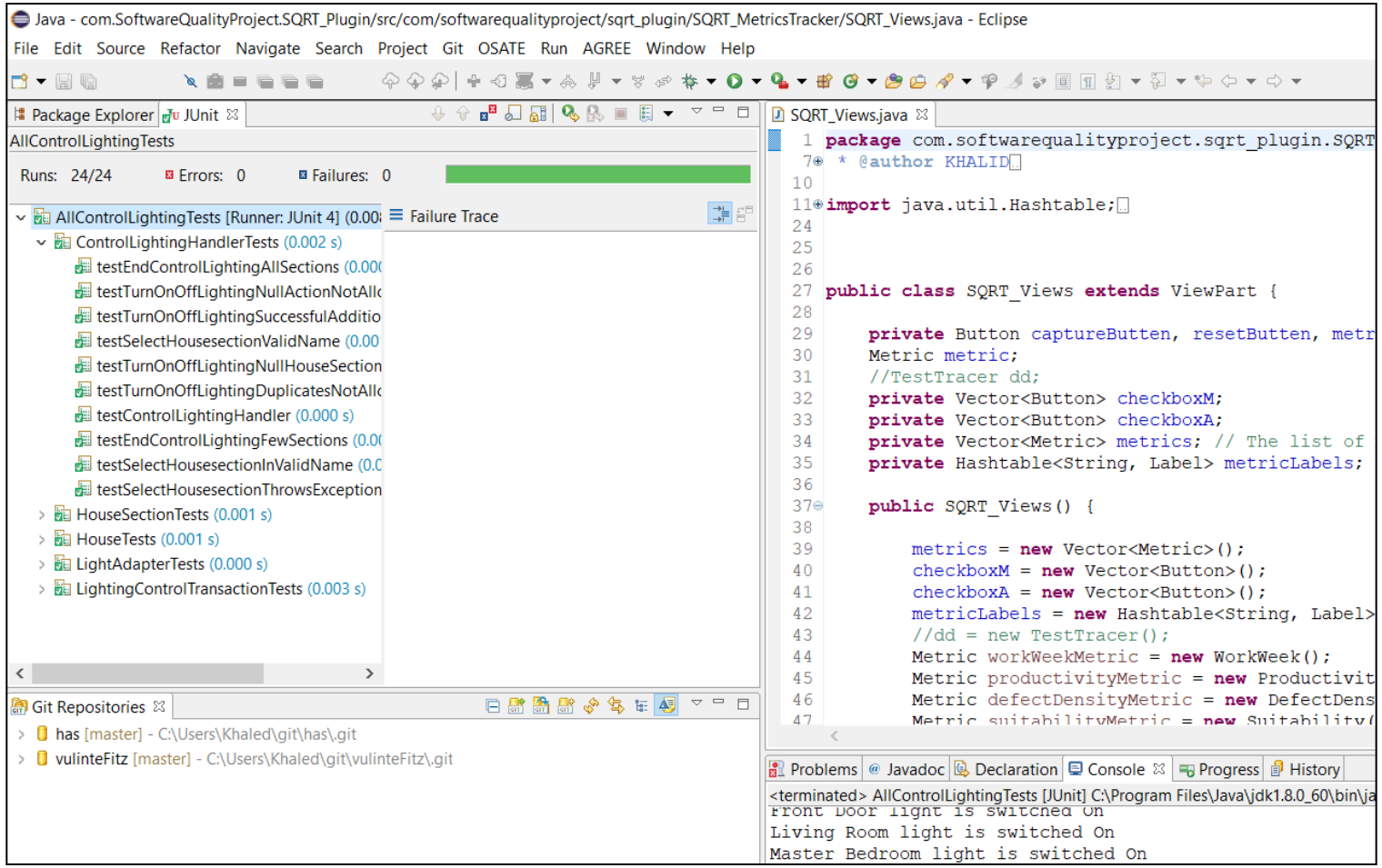
Misconceptions about Automation

- Test automation is the answer to our testing problems
- Since we have limited time to test, let's use test automation
- The testers cannot get involved in the project before the developers release the first build
- Automation is always cheaper
- Automation replaces humans in performing software testing

Conclusion

- Automation is a great idea to improve software testing
- Automation needs to be carefully planed
- Not to try automated all testing, but to choose areas that have the biggest playback
- Automation is a crucial piece to building a continuous delivery pipeline, but it must be balanced with manual testing to achieve optimal results.
- Hopes are raised by test automation to solve tough situations, but

JUint


Java - com.SoftwareQualityProject.SQRT_Plugin/src/com/softwarequalityproject/sqrt_plugin/SQRT_MetricsTracker/SQRT_Views.java - Eclipse
File Edit Source Refactor Navigate Search Project Git OSATE Run AGREE Window Help
Package Explorer
JUnit
AllControlLightingTests
Runs: 24/24
Errors: 0
Failures: 0
AllControlLightingTests [Runner: JUnit 4] (0.00
ControlLightingHandlerTests (0.002 s)
testEndControlLightingAllSections (0.00
testTurnOnOffLightingNullActionNotAllo
testTurnOnOffLightingSuccessfulAdditio
testSelectHousesectionValidName (0.00
testTurnOnOffLightingNullHouseSection
testTurnOnOffLightingDuplicatesNotAllo
testControlLightingHandler (0.000 s)
testEndControlLightingFewSections (0.0
testSelectHousesectionInValidName (0.0
testSelectHousesectionThrowsException
HouseSectionTests (0.001 s)
HouseTests (0.001 s)
LightAdapterTests (0.000 s)
LightingControlTransactionTests (0.003 s)
Failure Trace
Git Repositories
has [master] - C:\Users\Khaled\git\has\.git
vulinteFitz [master] - C:\Users\Khaled\git\vulinteFitz\.git
SQRT_Views.java
1 package com.softwarequalityproject.sqrt_plugin.SQRT
7 * @author KHALID
10
11 import java.util.Hashtable;
24
25
26
27 public class SQRT_Views extends ViewPart {
28
29 private Button captureButten, resetButten, metr
30 Metric metric;
31 //TestTracer dd;
32 private Vector<Button> checkboxM;
33 private Vector<Button> checkboxA;
34 private Vector<Metric> metrics; // The list of
35 private Hashtable<String, Label> metricLabels;
36
37 public SQRT_Views() {
38
39 metrics = new Vector<Metric>();
40 checkboxM = new Vector<Button>();
41 checkboxA = new Vector<Button>();
42 metricLabels = new Hashtable<String, Label>
43 //dd = new TestTracer();
44 Metric workWeekMetric = new WorkWeek();
45 Metric productivityMetric = new Productivit
46 Metric defectDensityMetric = new DefectDens
Problems Javadoc Declaration Console Progress History
<terminated> AllControlLightingTests [JUnit] C:\Program Files\Java\jdk1.8.0_60\bin\ja
Front Door light is switched On
Living Room light is switched On
Master Bedroom light is switched On

What is JUnit

- Component or unit testing is one of the important software testing levels practiced within software development projects.
- It is defined as the testing of individual software components such as classes and modules in isolation.
- A unit test is a piece of code that executes a specific functionality in the code under test
- Unit tests ensure that code is working as intended and validate that the code is still operating after making changes.
- JUnit is a member of the xUnit testing framework family, and it is actually the standard testing framework for Java.

What is JUnit

- JUnit is an API that enables developers to easily create Java test cases.
- It is a simple open source testing framework used to write and run repeatable automated tests.
- It was designed to run each test case independently and report its result automatically
- It provides a comprehensive *assertion* facility to verify *expected* versus *actual* results
- The more common IDEs, such as **Eclipse** and **IntelliJ**, already have JUnit functionality installed by default.

Eclipse Junit Testing

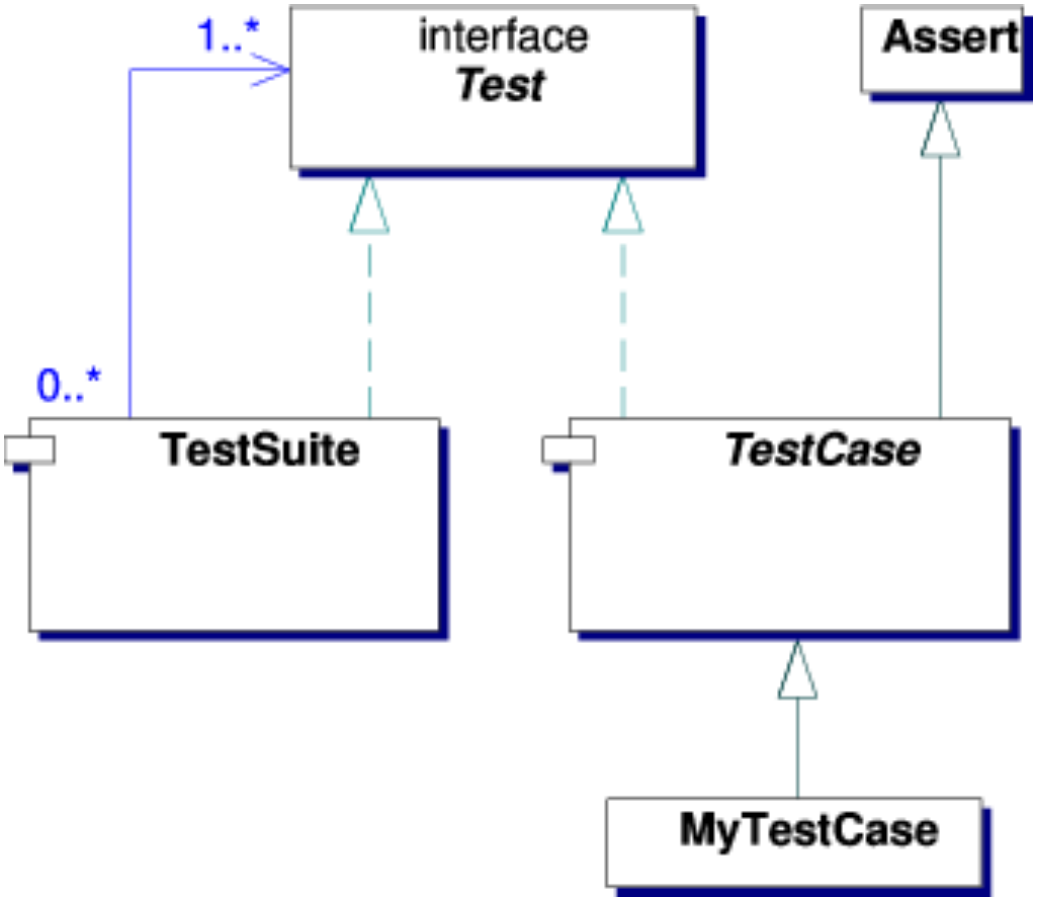
JUnit features include:

- Assertions for testing expected results
- Test fixtures for sharing common test data
- Test suites for easily organizing and running tests
- Graphical and textual test runners

JUnit is primarily intended for unit and integration testing, not system testing

JUnit can be used as **stand alone** Java programs (from the command line) or **within an IDE** such as Eclipse

Eclipse Junit Framework



Assertions (condition)

The **org.junit.Assert class** offers different flavors of assertions defined as static methods. These methods are useful in determining Pass or Fail status of a test case.

Statement	Description
fail(String)	Let the method fail. Might be used to check that a certain part of the code is not reached. Or to have failing test before the test code is implemented.
assertTrue(true) assertTrue(false)	Will always be true / false. Can be used to predefine a test result, if the test is not yet implemented.
assertTrue([message], boolean condition)	Checks that the boolean condition is true.
assertsEquals([String message], expected, actual)	Tests that two values are the same. Note: for arrays the reference is checked not the content of the arrays.
assertsEquals([String message], expected, actual, tolerance)	Test that float or double values match. The tolerance is the number of decimals which must be the same.
assertNull([message], object)	Checks that the object is NULL.
assertNotNull([message], object)	Checks that the object is not NUL.
assertSame([String], expected, actual)	Checks that both variables refer to the same object.

https://junit.org/junit4/javadoc/4.8/org/junit/Assert.html

Annotations

lines of code that are inserted in the program code to control how the testing methods run.

Annotation	Description
@Test	The annotation @Test identifies that a method is a test method.
@Before @BeforeEach	Will execute the method before each test. This method can prepare the test environment (e.g. read input data, initialize the class).
@After @AfterEach	Will execute the method after each test. This method can cleanup the test environment (e.g. delete temporary data, restore defaults).
@BeforeClass	Will execute the method once, before the start of all tests. This can be used to perform time intensive activities, for example to connect to a database.
@AfterClass	Will execute the method once, after all tests have finished. This can be used to perform clean-up activities, for example to disconnect from a database.
@Ignore	Will ignore the test method. This is useful when the underlying code has been changed and the test case has not yet been adapted. Or if the execution time of this test is too long to be included.

Test Fixtures

- A test fixture is useful if you have two or more tests for a common set of objects. Using a test fixture avoids duplicating the test code necessary to initialize and cleanup those common objects for each test.

  setUp() method to initialize common objects

  tearDown() method to cleanup those objects

- The JUnit framework automatically invokes the setUp() method before each test method is executed and the tearDown() method after each test method is executed.

TestSuite

- If you have several test cases and you want to run them all together, you could run the tests one by one at a time. But you would quickly get tired of that.
- Instead, JUnit provides an object ***TestSuite*** which runs any number of test cases together.

- The suite method is like a main method that is specialized to run test cases.

Your next project assignment should include:

- ***Assert statements like:***
  - ✓ *assertEquals()*
  - ✓ *assertTrue() / assertFalse()*
  - ✓ *assertNull([message], object)*
  - ✓ *assertNotNull([message], object)*
  - ✓ *assertSame() / assertNotSame()*
  - ✓ *assertThat()*
- ***Annotations***
- ***Fixtures***
- ***TestSuite()***

Exercise

- Write a small Java calculator program that only executes the basic arithmetic operations like +, -. Then create a set of different test cases to validate the operations.

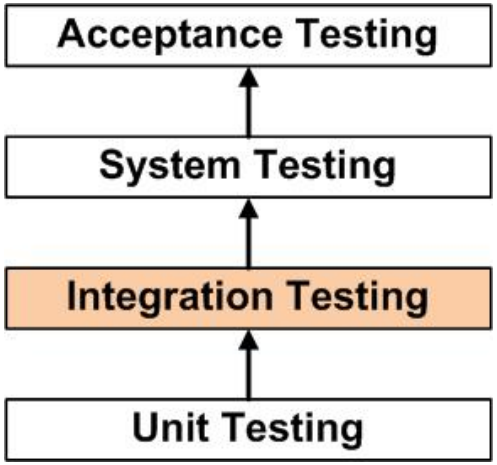
Integration Testing

The Mars Climate Orbiter Mission

- mission failed in September 1999
- completed successful flight: 416,000,000 miles (665.600.600 km)
- 41 weeks flight duration
- lost at beginning of Mars orbit
- An integration fault: Lockheed Martin used English units for acceleration calculations (pounds), and Jet Propulsion Laboratory used metric units (newtons).
- NASA announced a US$ 50,000 project to discover how this happened.

Purpose of Integration Testing

- Presumes previously tested units
- Individual software modules are integrated logically and tested in groups.
- Not system testing
- Tests functionality "between" unit levels – focusing on testing data communication amongst these units.



"Testing performed to expose defects in the interfaces and in the interactions between integrated components or systems." ***ISTQB***

Testing Level Assumptions and Objectives

- Unit assumptions
  - units are complete
  - Compiles correctly
- Integration assumptions
  - Unit testing complete
- System assumptions
  - Integration testing complete
  - Tests occur at port boundary

- Unit goals
  - Correct unit function
  - Coverage metrics satisfied
- Integration goals
  - Interfaces correct
  - Correct function across units
  - Fault isolation support
- System goals
  - Correct system functions
  - Non-functional requirements tested
  - Customer satisfaction.

Approaches to Integration Testing

- Functional Decomposition (most commonly described in the literature)
  - Top-down
  - Bottom-up
  - Sandwich/Hybrid
  - "Big bang"
- Call graph
  - **Pairwise integration**
  - Neighborhood integration
- Paths
  - MM-Paths

Incremental Approaches

- Testing is done by joining two or more modules/units that are logically related. Then the other related modules are added and tested for the proper functioning.
- This process is carried out by using **dummy modules** called ***Stubs*** and ***Drivers***. Stubs and Drivers do not implement the entire programming logic of the software module but just simulate data communication with the calling or called module/unit.
  - Stub is called by the module under test.
  - Driver calls the module to be tested.
- Incremental Approach is carried out by many different methods
  - Bottom-up (uses Drivers to simulate calling units)
  - Top Down (uses Stubs to simulate called units))
  - Sandwich/Hybrid

Stubs and Drivers Techniques

- Stubs and drivers are used to replace the missing software and simulate the interface between the software components in a simple manner.
- Stubs and drivers both are dummy modules and are only created for test purposes.
- **Stubs** are used in ***top down*** testing approach, when you have the major module ready to test, but the sub modules are still not ready yet. Stubs are "called" programs.
- **Drivers** are used in ***bottom up*** testing approach, when the sub modules are ready, but the main module is still not ready yet. Drivers are the "calling" programs .

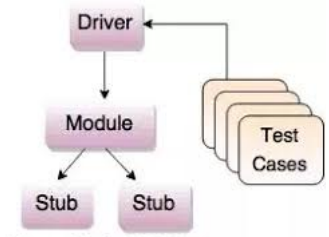
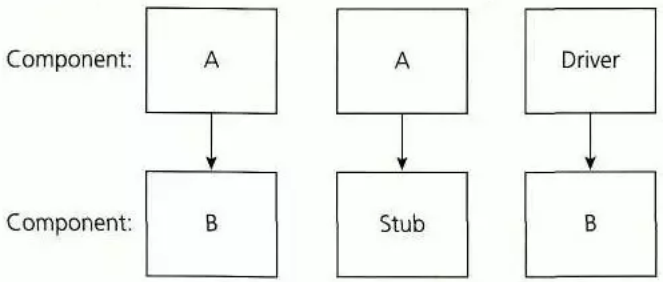


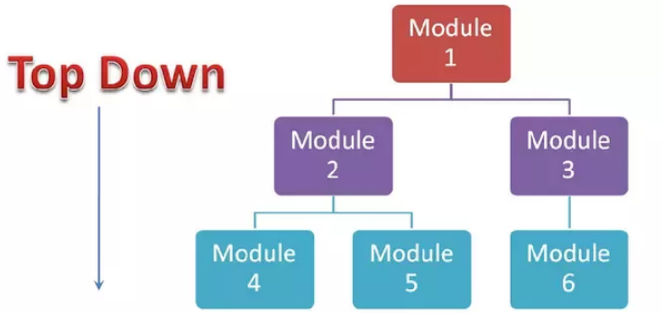
Fig. - Stub & Driver



http://www.professionalqa.com/

Top Down Integration

- Top-level units are tested first, and lower level units are tested step by step after that (following the control flow or architectural structure (e.g. starting from the GUI or main menu). Test Stubs are needed to simulate lower level.
- This approach is taken when top-down development is followed.
- Advantages
  - Fault localization is easier.
  - Possibility to obtain an early prototype.
  - Critical Modules are tested on priority; major design flaws could be found and fixed first.



- Disadvantages
  - Needs many Stubs.
  - Modules at lower level are tested inadequately.

Example—Calendar Program

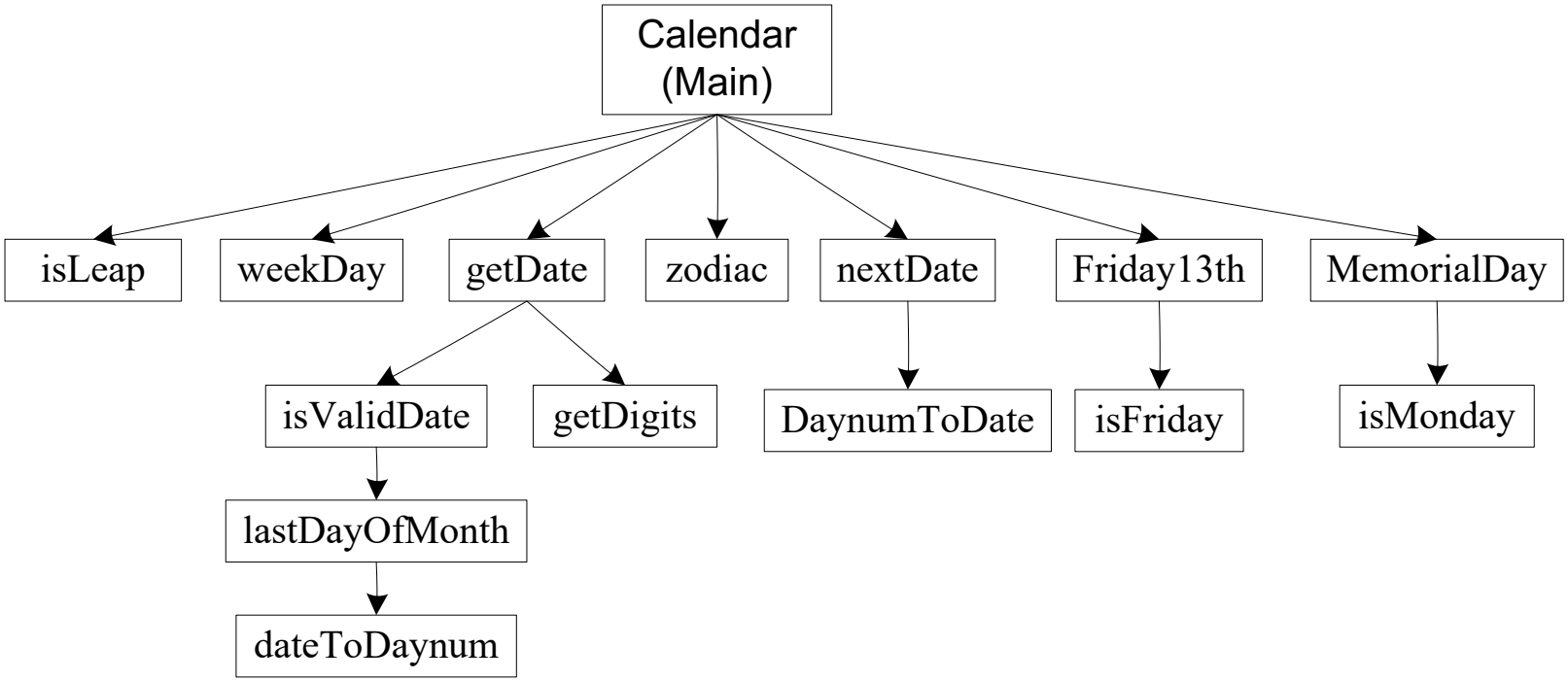
- (see text for pseudo-code version)
- Date in the form mm, dd, yyyy
- Calendar functions
  - the leap year
  - the date of the next day (NextDate)
  - the day of the week corresponding to the date
  - the zodiac sign of the date
  - the most recent year in which Memorial Day was celebrated on May 27
  - the most recent Friday the Thirteenth

Calendar Program Units

Main Calendar

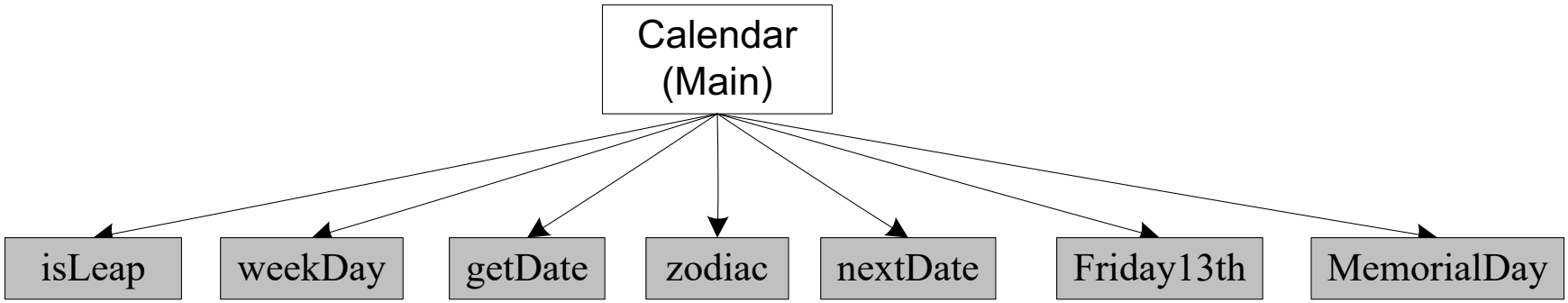
- Function isLeap
- Function weekDay
- Function getDate
  - Function isValidDate
    - Function lastDayOfMonth
      - Function dateToDaynum
    - Function getDigits
- Function zodiac
- Function nextDate
  - Function dayNumToDate
- Function friday13th
  - Function isFriday
- Function memorialDay
  - Function isMonday

Functional Decomposition of Calendar



A functional decomposition tree reflects the lexicological inclusion of units, in terms of the order in which they need to be compiled

First Step in Top-Down Integration



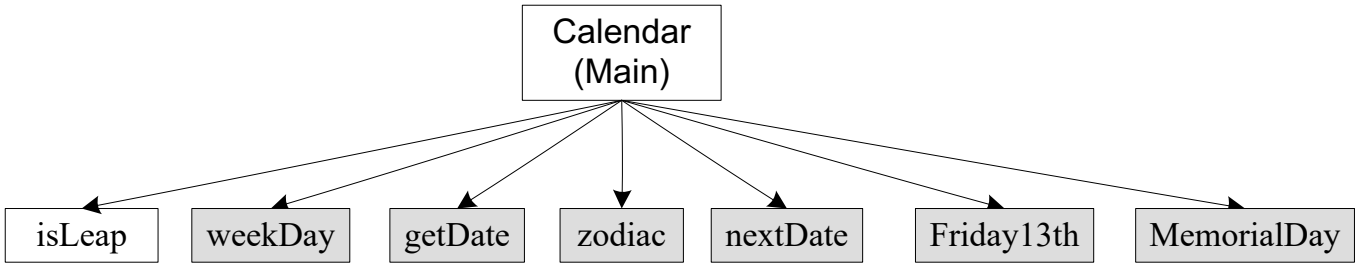
"Grey" units are stubs that return the correct values when referenced. This level checks the main program logic.

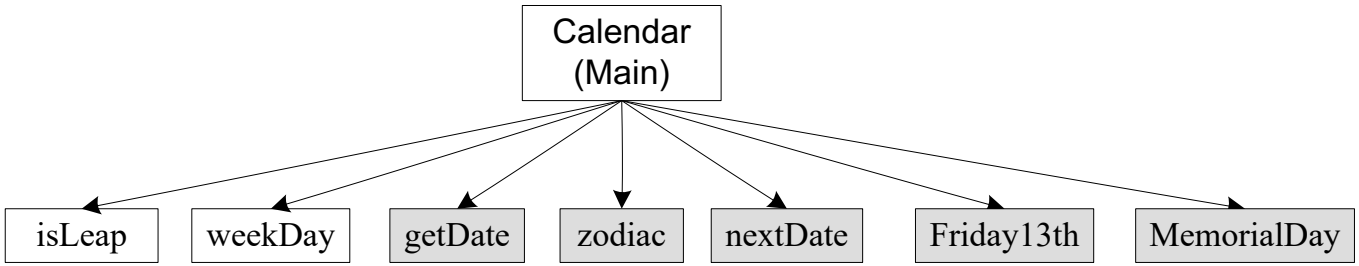
weekDayStub

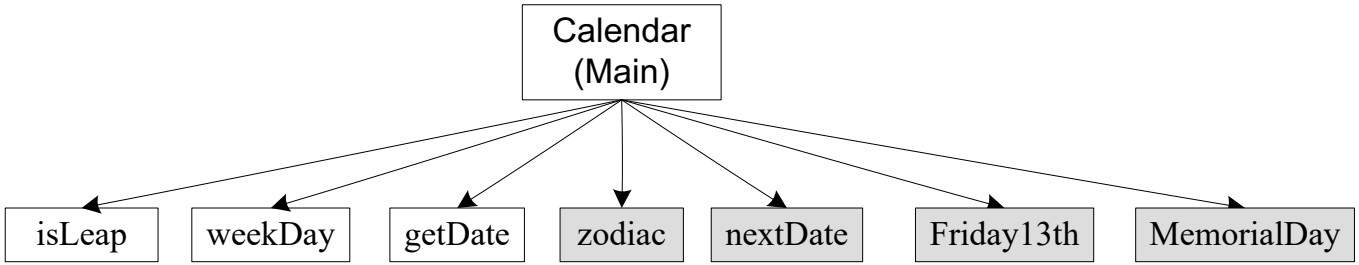
```
Function weekDayStub(mm, dd, yyyy, dayName)
If ((mm = 10) AND (dd = 28) AND (yyyy = 2013))
        Then dayName = “Monday”
EndIf
.
.
.
If ((mm = 10) AND (dd = 30) AND (yyyy = 2013))
        Then dayName = “Wednesday”
EndIf
```

Next Three Steps

(replace one stub at a time with the actual code.)





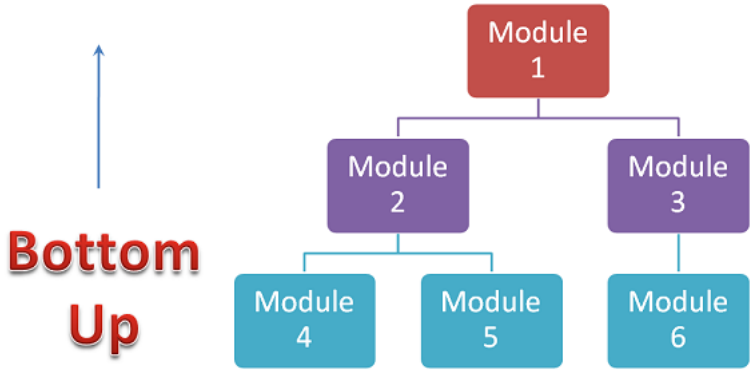


Bottom-Up Integration

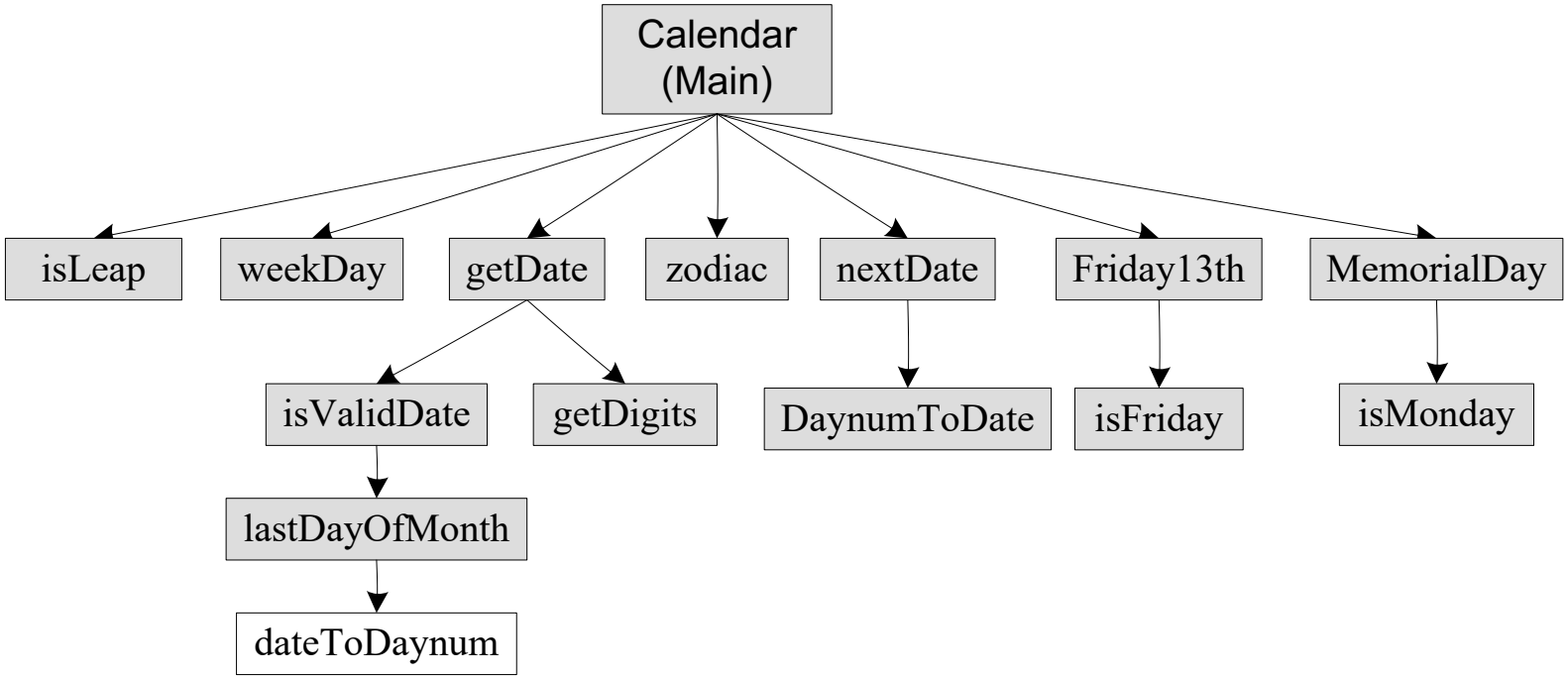
- Bottom level units are tested first and upper-level units step by step are tested later. Drivers are used to simulate calling units.
- This approach is applied when bottom-up development approach is followed.

- Advantages
  - Fault localization is easier.
  - Unlike Big-bang approach, no time is wasted waiting for all modules to be developed.

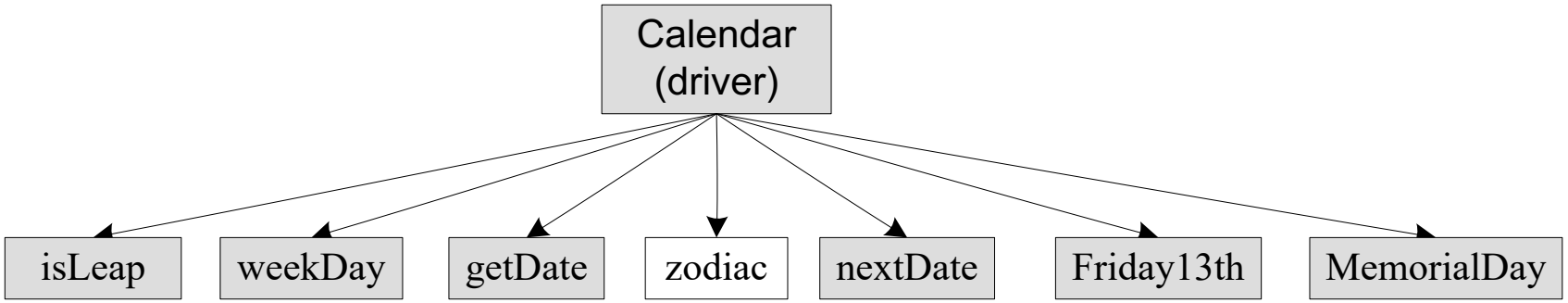
- Disadvantages
  - Critical modules (at the top level of software architecture) which control the flow of application are tested last and may be prone to defects.
  - Early prototype is not possible.



Starting Point of Bottom-Up Integration



Bottom-Up Integration of Zodiac



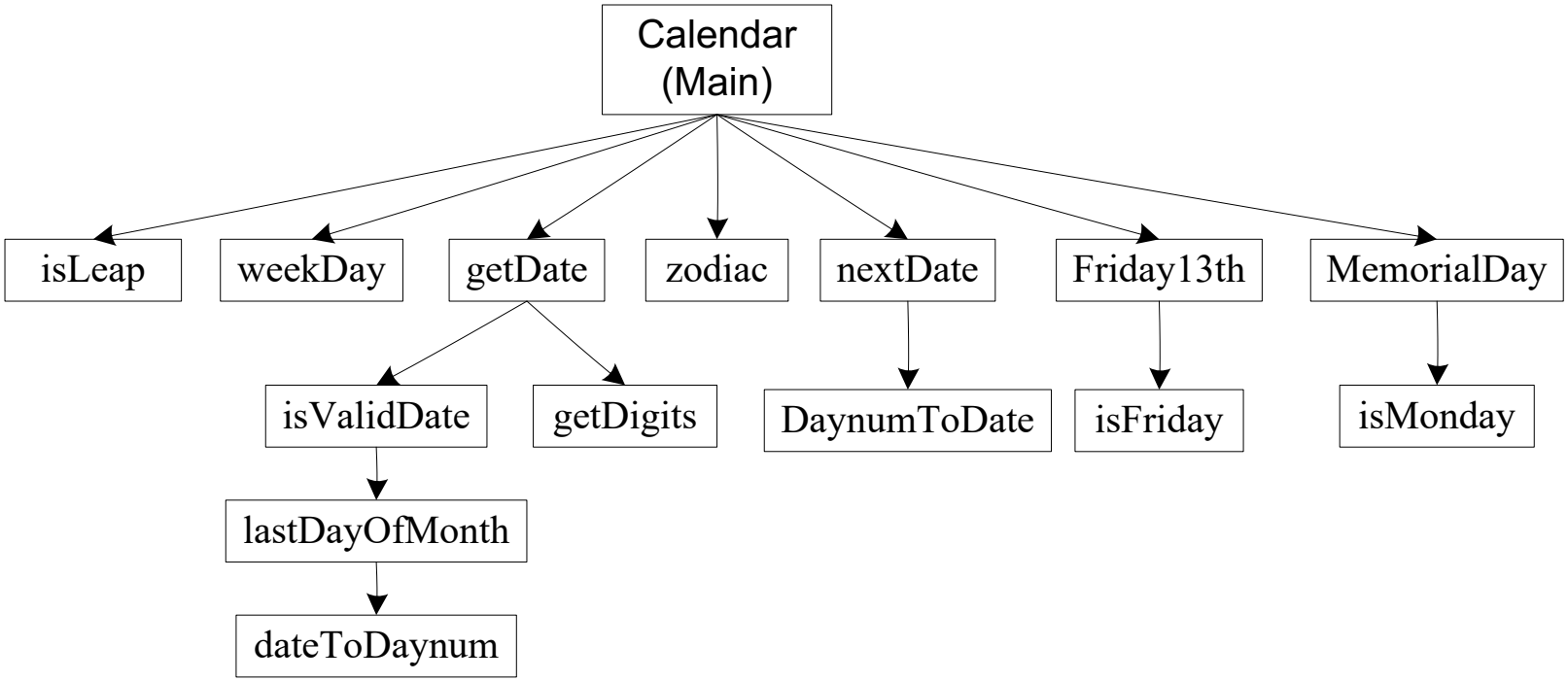
Big Bang Approach

- All or most of the components, units, are combined altogether and tested at once.
- This approach is applied when the testing team receives the entire software in a bundle, so everything is tested as a whole.
- This approach is generally executed by those developers who follows the ‘Run it and see’ approach
- Advantages of Big Bang Approach:
  - convenient for small systems and with well-understood requirements
  - everything is finished before integration testing starts
  - prevents wastage of extra efforts and time and makes the testing process cost effective.
  - testers can enjoy fast testing and can effortlessly conduct further testing

Big Bang Approach

- Disadvantages of Big Bang Approach:
  - difficult to trace the cause of failures as the modules are integrated late. If any bug is found, it becomes difficult to detach all the modules on order to find out its root cause.
  - quite challenging and risky, as all the modules and components are integrated together in a single step.
  - Since the integration testing can commence only after "all" the modules are designed, testing team will have less time for execution in the testing phase.
  - The chances of having critical testing failures are more because of integrating all the components together at same time.
  - there is a high probability of missing some crucial defects, which might pop up in the production environment.

Big Bang Integration



Big Bang Integration

- No...
  - stubs
  - drivers
  - strategy
- And very difficult fault isolation
- (Named after one of the theories of the origin of the Universe)
- This is the practice in an agile environment with a daily run of the project to that point.

Pros and Cons of Decomposition-Based Integration

- Pros
  - intuitively clear
  - “build” with proven components
  - fault isolation varies with the number of units being integrated

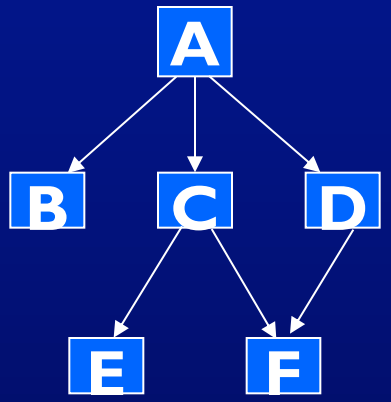
- Cons
  - based on lexicographic inclusion (a purely structural consideration)
  - some branches in a functional decomposition may not correspond with actual interfaces.
  - stub and driver development can be extensive

Graph Coverage for Software Design

- Use of data abstraction and OO software has led to an increased emphasis on modularity and reuse
- Testing design relationships is more important than before
- Graphs are based on the connections among the software pieces (components)
  - Connections are dependency relations, also called couplings

Call Graph

- The most common graph used for structural design coverage is the call path
- Nodes : represent Units (in Java – methods) (in C – functions)
- Edges : represent method Calls to units



Example call graph

Node coverage : call every unit at least once (method coverage)

Edge coverage : execute every call at least once (call coverage)

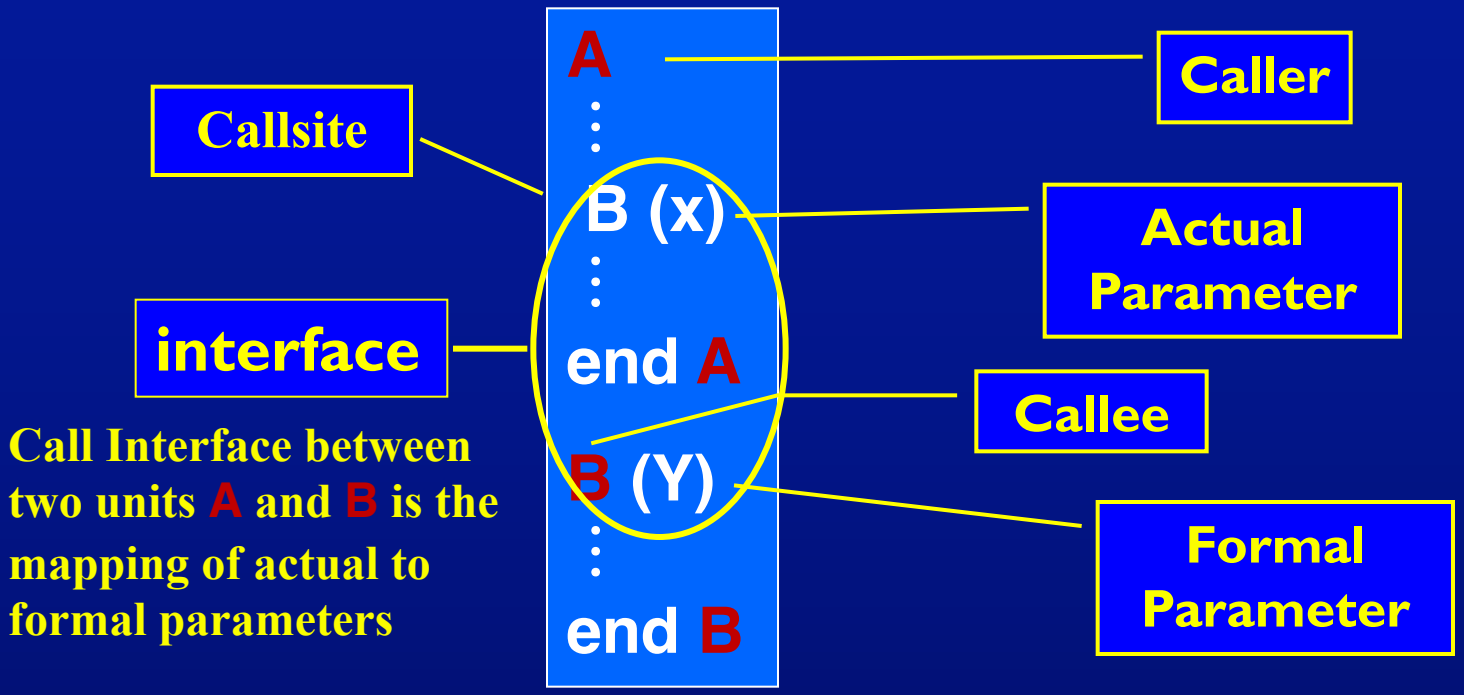
Preliminary Definitions

- Caller : A unit that invokes another unit
- Callee : The unit that is called
- Callsite : Statement or node where the call appears
- Actual parameter : Variable in the caller (argument)
- Formal parameter : Variable in the callee (parameter)

Pairwise/DU Pairs integration

Using CFG graph is very helpful

Example Call Site



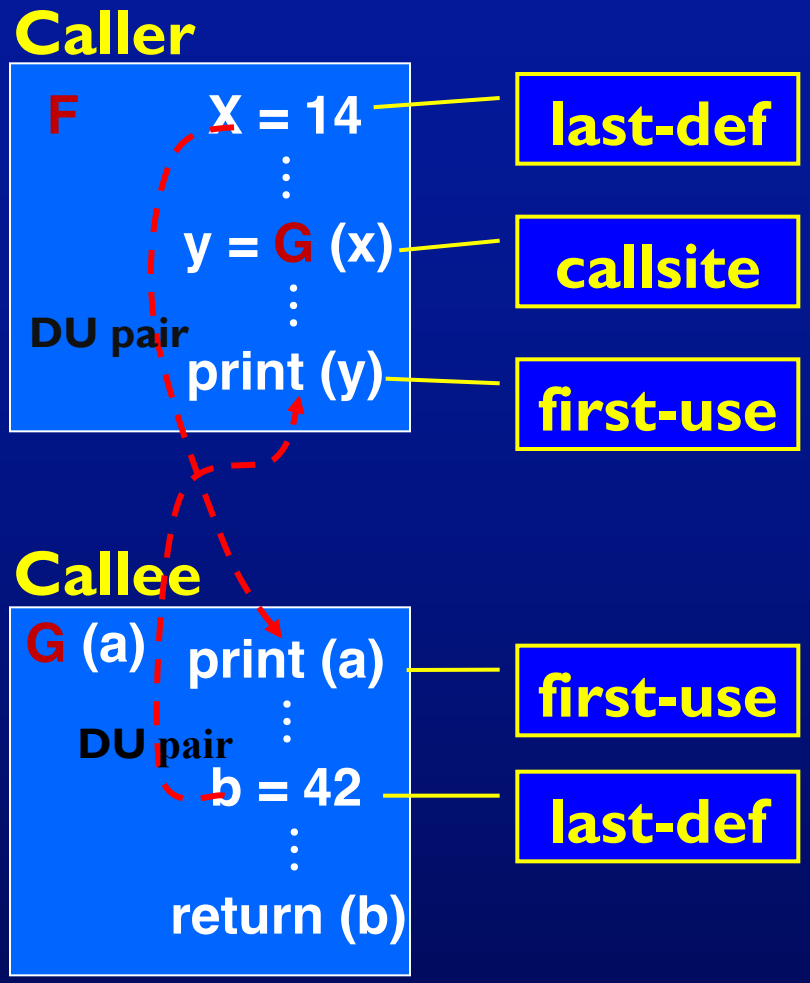
- This is integration testing, and we really only care about the interface …

Pairwise/DU Pairs integration

Pairwise/DU Pairs

- If we focus on the interface, then we just need to consider the last definitions of variables before calls, returns, first uses inside units and after calls
- Last-def : The set of nodes that define a variable *x* and has a def-clear path from the node through a callsite to a use in the other unit
  - Can be from caller to callee (parameter or shared variable) or from callee to caller as a return value
- First-use : The set of nodes that have uses of a variable *y* and for which there is a def-clear and use-clear path from the callsite to the nodes

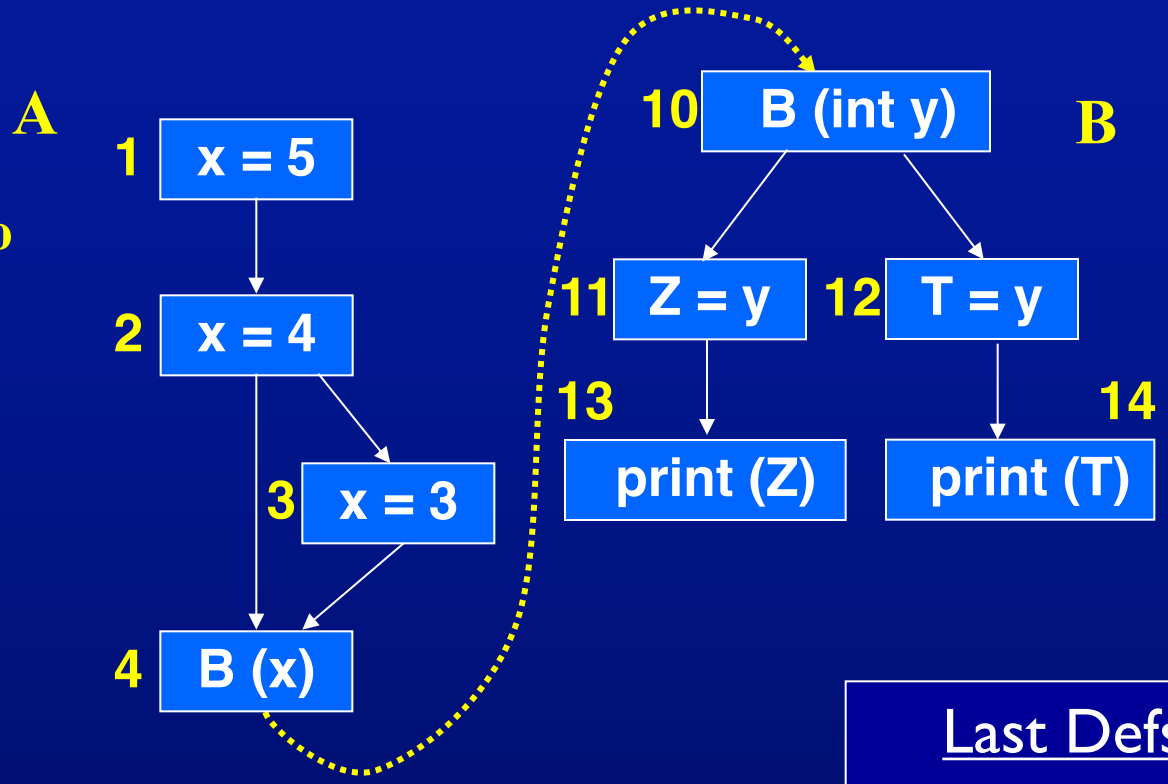
Inter-DU Pairs Example



Inte-DU Pairs Example

CFG is used to simplify units

Surrounding the calls between methods



DU Pairs

(A, x, 2)—(B, y, 11)
(A, x, 2)—(B, y, 12)
(A, x, 3)—(B, y, 11)
(A, x, 3)—(B, y, 12)

Last Defs

2, 3

First Uses

11, 12

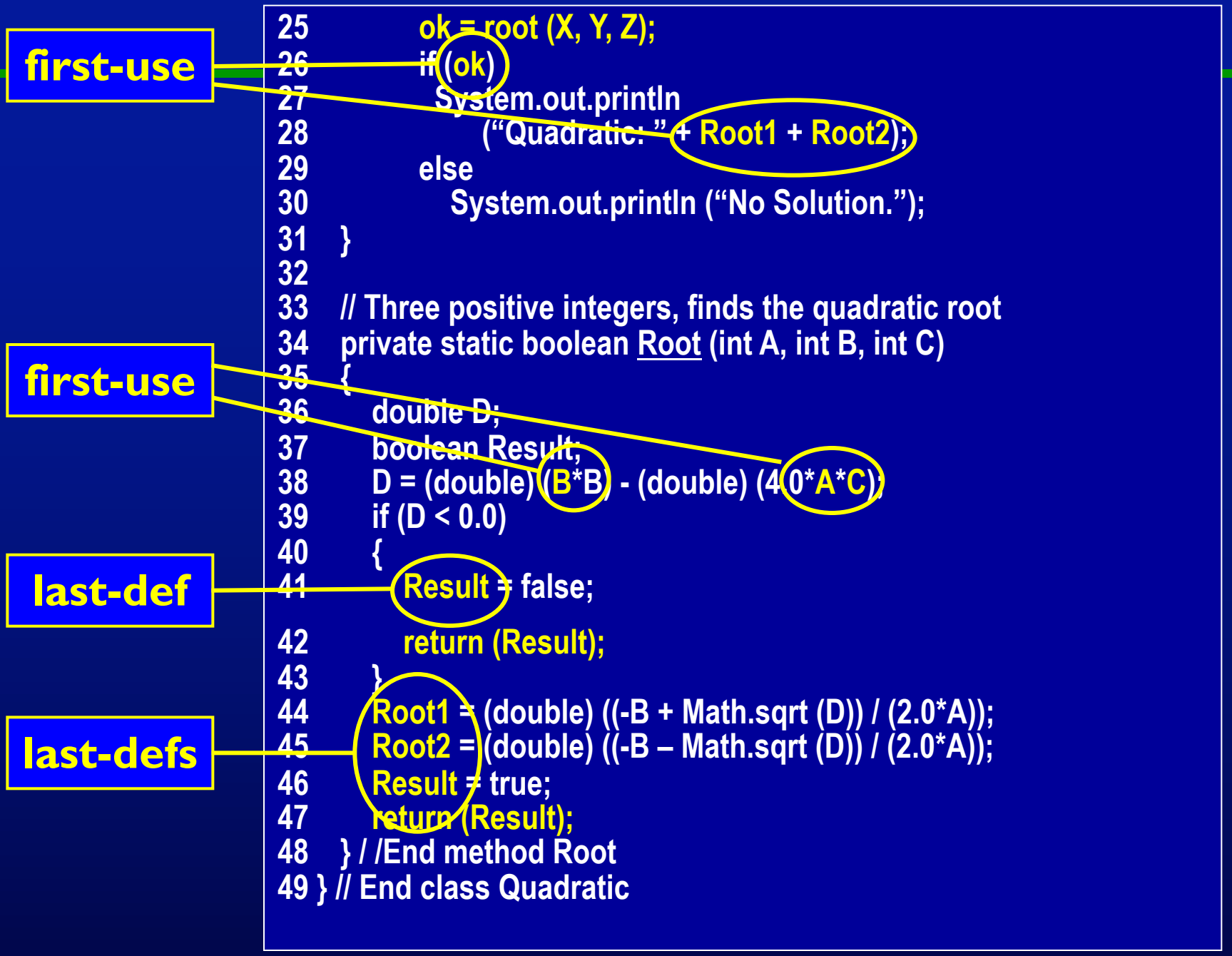
Exercise – Quadratic

```
1 // Program to compute the quadratic root for two
numbers
2 import java.lang.Math;
3
4 class Quadratic
5 {
6  private static float Root1, Root2;
7
8  public static void main (String[] argv)
9  {
10    int X, Y, Z;
11    boolean ok;
12    int controlFlag = Integer.parseInt (argv[0]);
13    if (controlFlag == 1)
14    {
15       X = Integer.parseInt (argv[1]);
16       Y = Integer.parseInt (argv[2]);
17       Z = Integer.parseInt (argv[3]);
18    }
19    else
20    {
21       X = 10;
22       Y = 9;
23       Z = 12;
24    }
25       ok = root (X, Y, Z);
26       if (ok)
27         System.out.println
28            ("Quadratic: " + Root1 + Root2);
29       else
30          System.out.println ("No Solution.");
31  }
32
33 // Three positive integers, finds quadratic root
34   private static boolean root (int A, int B, int C)
35   {
36      double D;
37      boolean Result;
38      D = (double) (B*B) -  (double) (4.0*A*C );
39      if (D < 0.0)
40      {
41         Result = false;
42         return (Result);
43      }
44      Root1 = (double) ((-B + Math.sqrt(D))/(2.0*A));
45      Root2 = (double) ((-B – Math.sqrt(D))/(2.0*A));
46      Result = true;
47      return (Result);
48   } // End method Root
49 } // End class Quadratic
```

```
1 // Program to compute the quadratic root for two numbers
2 import java.lang.Math;
3
4 class Quadratic
5 {
6  private static float Root1, Root2;
7
8  public static void main (String[] argv)
9  {
10     int X, Y, Z;
11    boolean ok;
12    int controlFlag = Integer.parseInt (argv [0]);
13    if (controlFlag == 1)
14    {
15       X = Integer.parseInt (argv [1]);
16       Y = Integer.parseInt (argv [2]);
17       Z = Integer.parseInt (argv [3]);
18    }
19    else
20    {
21       X = 10;
22       Y = 9;
23       Z = 12;
24    }
```

shared variables

last-defs


first-use
first-use
last-def
last-defs
25 ok = root (X, Y, Z);
26 if (ok)
27 System.out.println
28 ("Quadratic: " + Root1 + Root2);
29 else
30 System.out.println ("No Solution.");
31 }
32
33 // Three positive integers, finds the quadratic root
34 private static boolean Root (int A, int B, int C)
35 {
36 double D;
37 boolean Result;
38 D = (double) (B*B) - (double) (4.0*A*C);
39 if (D < 0.0)
40 {
41 Result = false;
42 return (Result);
43 }
44 Root1 = (double) ((-B + Math.sqrt (D)) / (2.0*A));
45 Root2 = (double) ((-B – Math.sqrt (D)) / (2.0*A));
46 Result = true;
47 return (Result);
48 } / /End method Root
49 } // End class Quadratic

Quadratic – Coupling DU-pairs

Pairs of locations: method name, variable name, statement

(main (), X, 15) – (Root (), A, 38)

(main (), Y, 16) – (Root (), B, 38)

(main (), Z, 17) – (Root (), C, 38)

(main (), X, 21) – (Root (), A, 38)

(main (), Y, 22) – (Root (), B, 38)

(main (), Z, 23) – (Root (), C, 38)

(Root (), Root1, 44) – (main (), Root1, 28)

(Root (), Root2, 45) – (main (), Root2, 28)

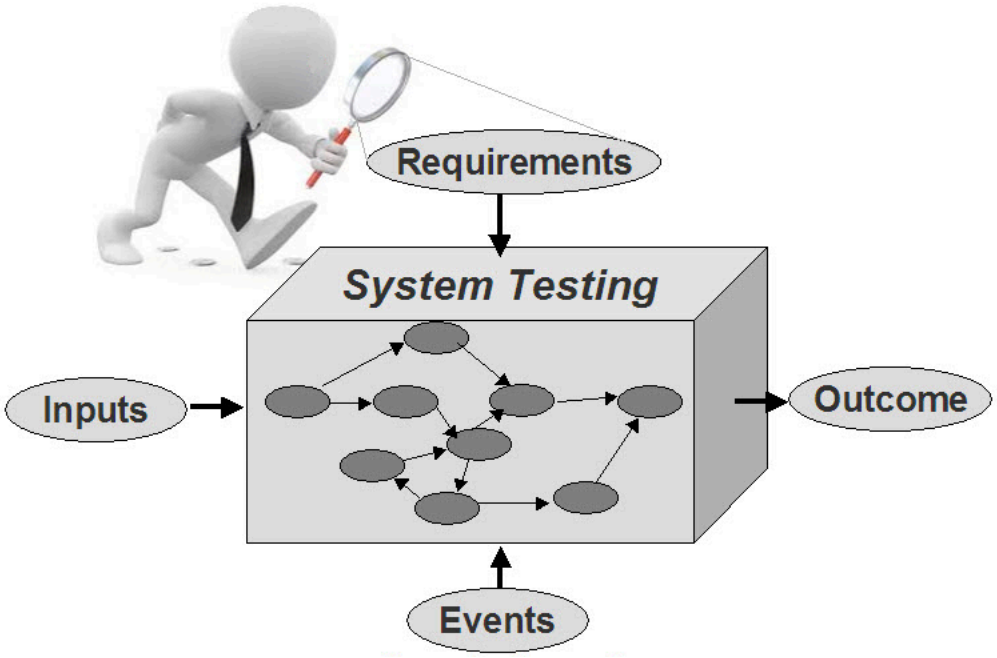
(Root (), Result, 41) – (main (), ok, 26)

(Root (), Result, 46) – (main (), ok, 26)

Summary—What Works?

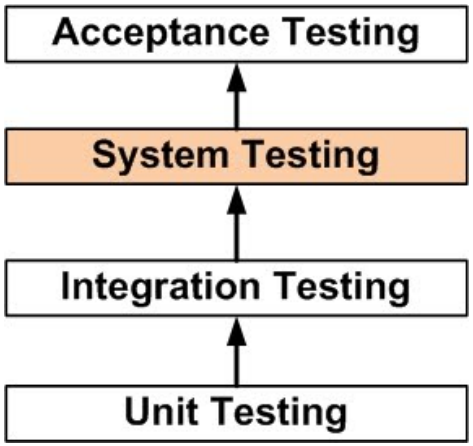
- Call graphs are common and very useful ways to design integration tests
- The Pairwise technique is relatively easy to compute and results in effective integration tests
- The ideas for testing OO software are preliminary and have not been used much in practice

System Testing

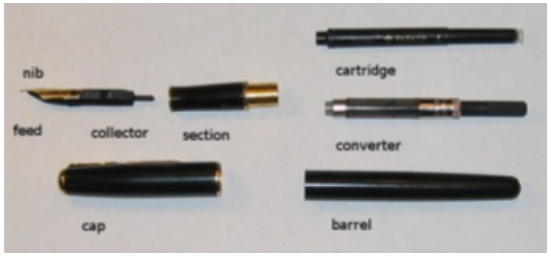


What is System Testing?

- A level of software testing to monitor and assess the behavior of the complete and fully-integrated software product, on the basis of pre-decided specifications and functional requirements.



- System testing tries to answer the question "whether the complete system functions in accordance to its pre-defined requirements (SRS)?"



"The process of testing an integrated system to verify that it meets specified requirements." ***ISTQB***

What is System Testing?

- It usually comes under black box testing i.e. only external working features of the software are evaluated during this testing.
- It does not require any internal knowledge of the coding, programming, design, etc., and is completely based on users-perspective.
- It is carried out only after Integration Testing is completed where both Functional & Non-Functional requirements are verified.
- In software system testing, testers are concentrated on finding bugs/defects based on software application behavior and expectation of end-user. The goal is not to find faults but to demonstrate correct behavior–to ensure that software does what customers want it to do.

Tests Performed During System Testing

- **Installation Testing**:- It is used to check desired functioning of the software, after its successful installation, along with, all necessary requirements
- **Functional Testing**:- A type of black-box testing, that enables to assess the integrated system according to its requirements specification.
- **Recoverability Testing**:- It is achieved by, deliberate failure or crash of the software, to assess its response to presence of errors or loss of data.
- **Interoperability Testing**:- It ensures, the ability of software to get compatible and interact with other software or system and their components.

Tests Performed During System Testing

- **Performance Testing**:- to examine the response times, stability, capacity (volume), and other quality metrics of the software, under different workloads.
- **Scalability Testing**:- Software should be scalable, along with the increase in load, number of concurrent users, data size, etc.
- **Reliability Testing**:- To assess the system ability to operate without failure under given condition for a given time interval.
- **Regression Testing**:- It guarantees the original functionality of the software, after each modification in it.

Tests Performed During System Testing

- **Documentation Testing**:- This involves, evaluation of documentation artifacts, prepared before and during the testing phase. Also documentation standards need to be checked.

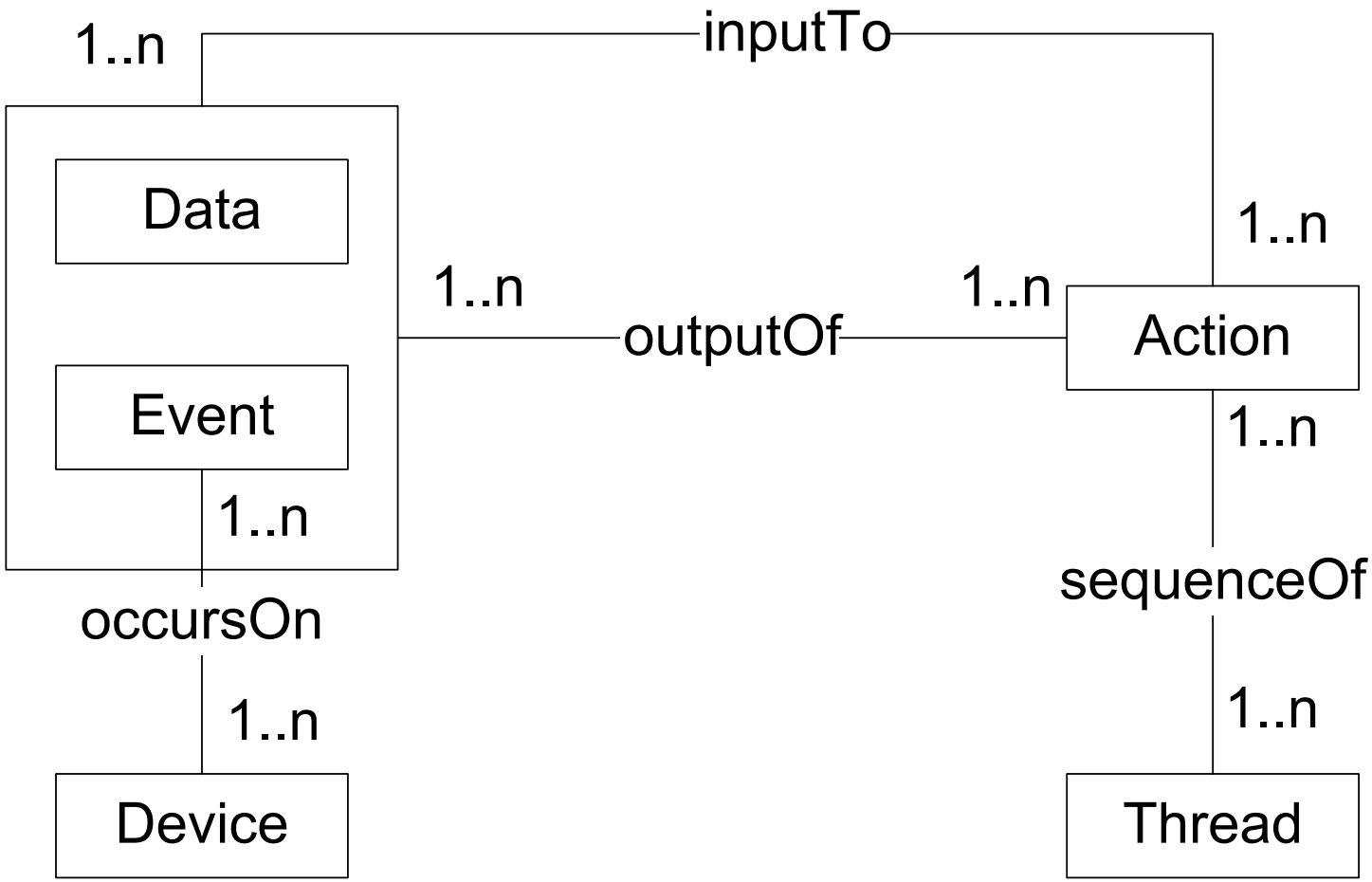
- **Security Testing**:- To assess, the security features of the software, so as to ensure, protection, authenticity, confidentiality and integrity of the information and data.

- **Usability Testing**:- Ensures software's user-friendliness feature and prevents end-users from issues or problems, while using and handling the software product.

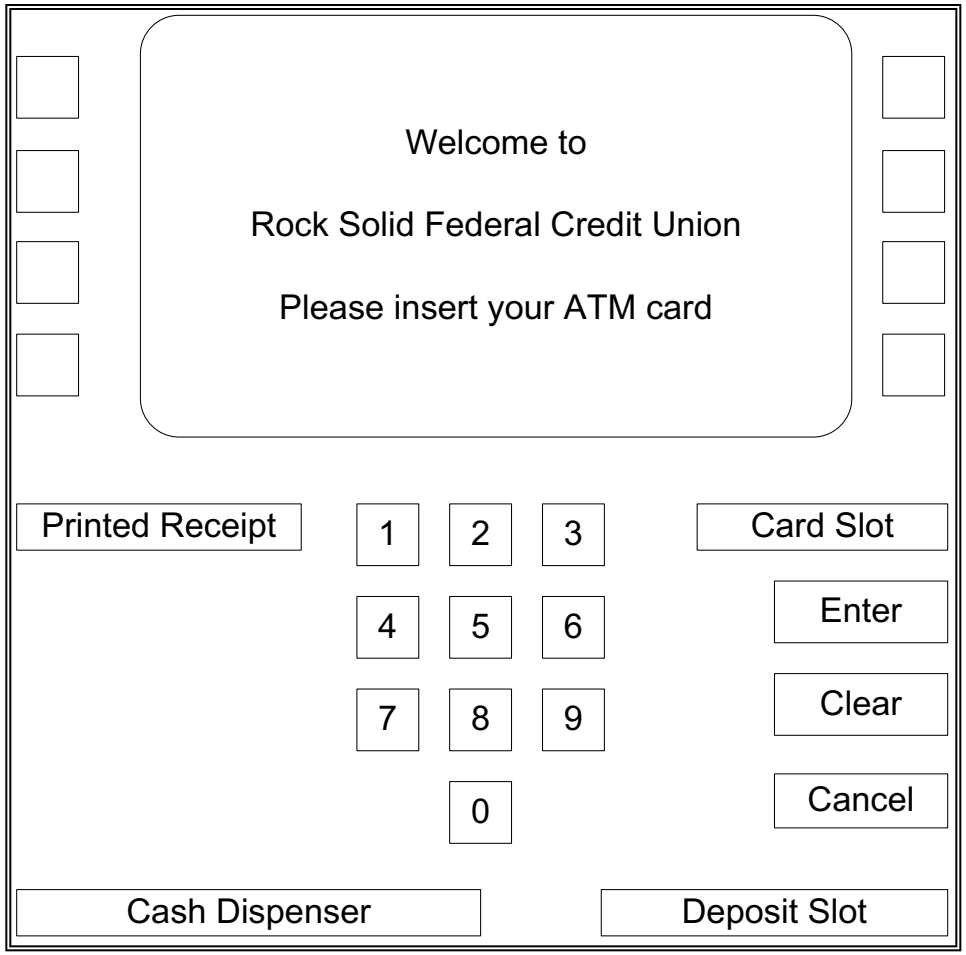
Basis Concepts for Requirements Specification

- All of requirements specification models are developed on these basis concepts.
- Data
  - Inputs to actions
  - Outputs of actions
- Events
  - Inputs to actions
  - Outputs of actions
- Actions
- Threads (sequences of actions)
- Devices

E/R Model of Basis Concepts



ATM System User Interface



ATM System Screens

Screen 1

Welcome
Please insert your ATM card

Screen 2

Please enter your PIN
_ _ _ _

Screen 3

Your PIN is incorrect. Please try again.

Screen 4

Invalid ATM card. It will be retained.

Screen 5
Select transaction:
balance >
deposit >
withdrawal >

Screen 6

Balance is $dddd.dd

Screen 7

Enter amount. Withdrawals must be multiples of $10

Screen 8

Insufficient Funds! Please enter a new amount

Screen 9

Machine can only dispense $10 notes

Screen 10

Temporarily unable to process withdrawals. Another transaction?

Screen 11

Your balance is being updated. Please take cash from dispenser.

Screen 12

Temporarily unable to process deposits. Another transaction?

Screen 13

Please insert deposit into deposit slot.

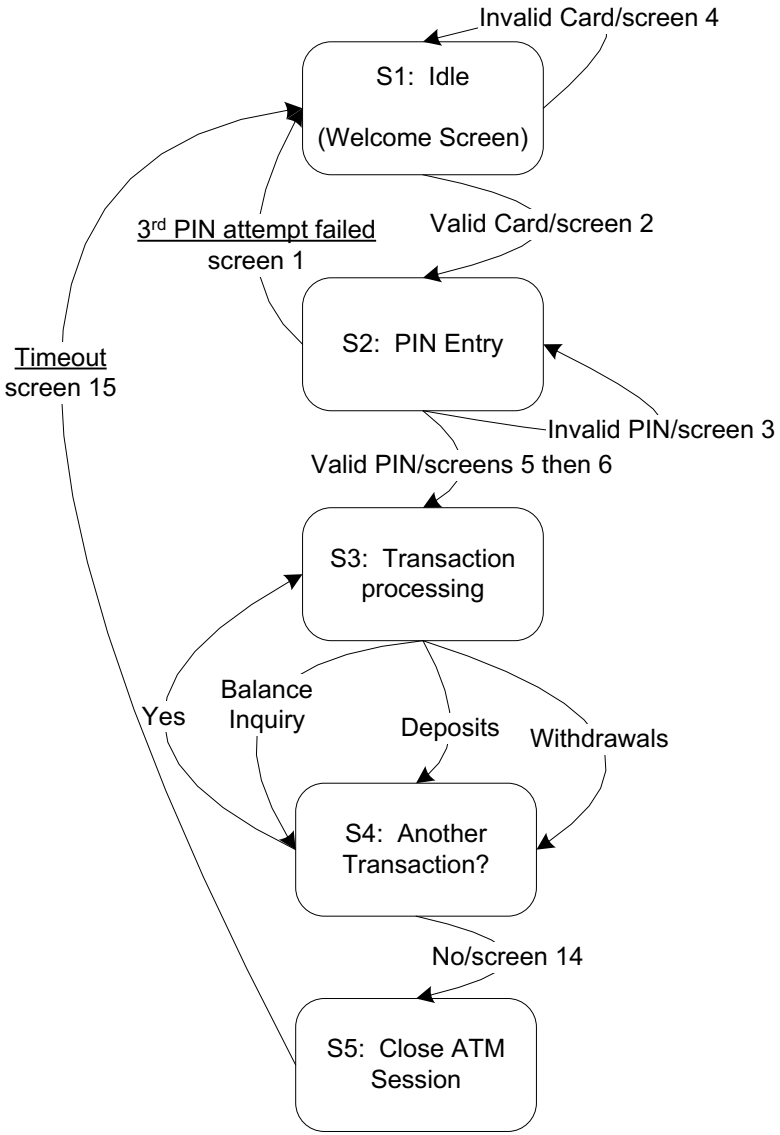
Screen 14

Your new balance is being printed. Another transaction?

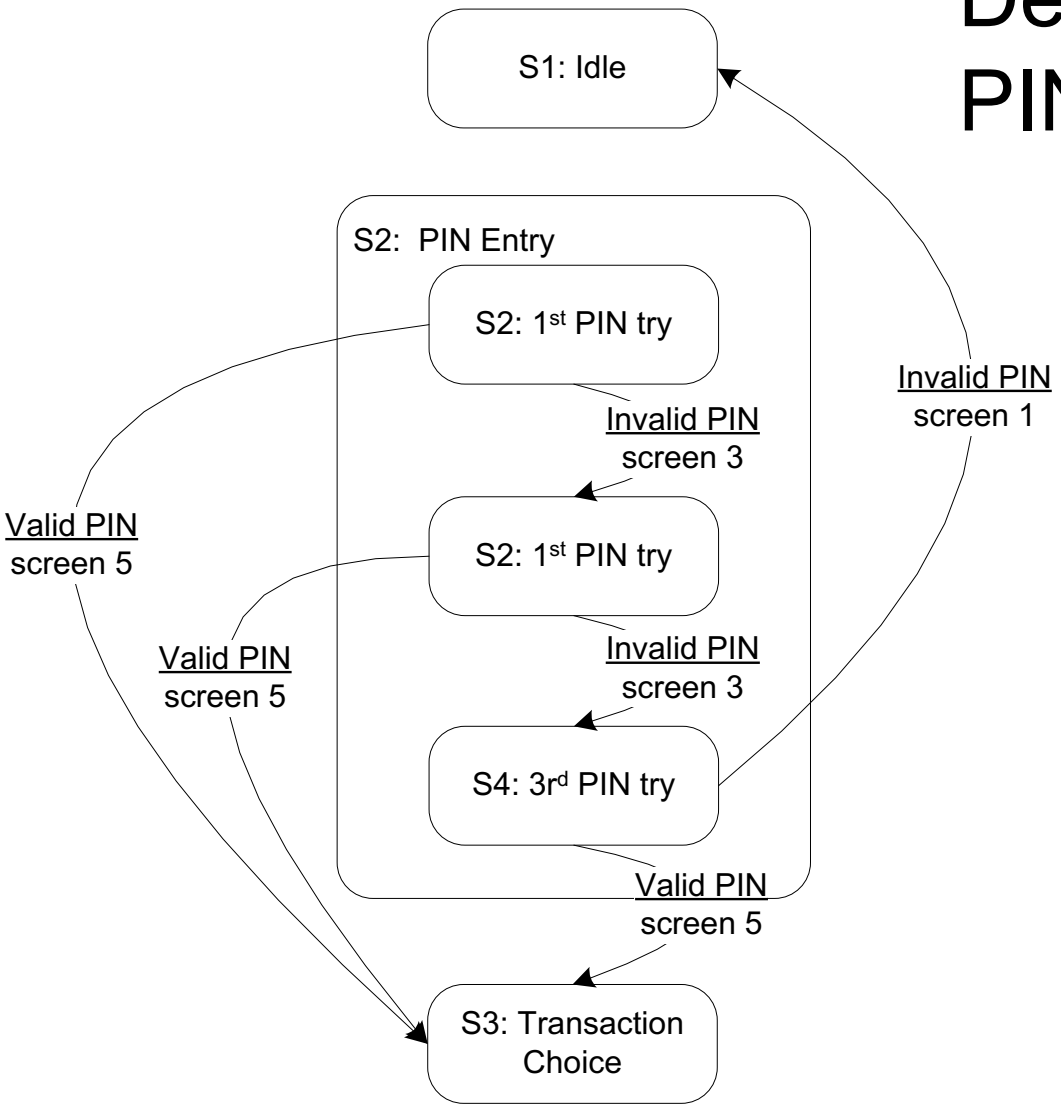
Screen 15

Please take your receipt and ATM card. Thank you.

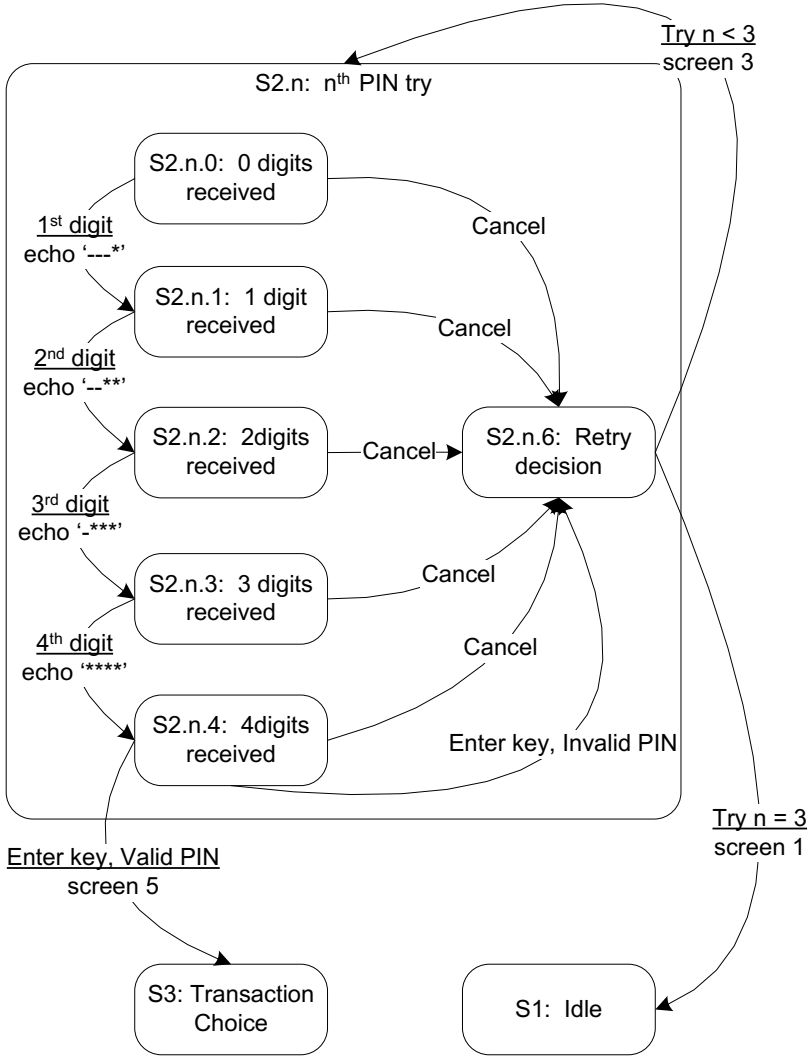
Uppermost Level of the ATM Finite State Machine



Details of ATM PIN Entry State



Details of ATM PIN Try State



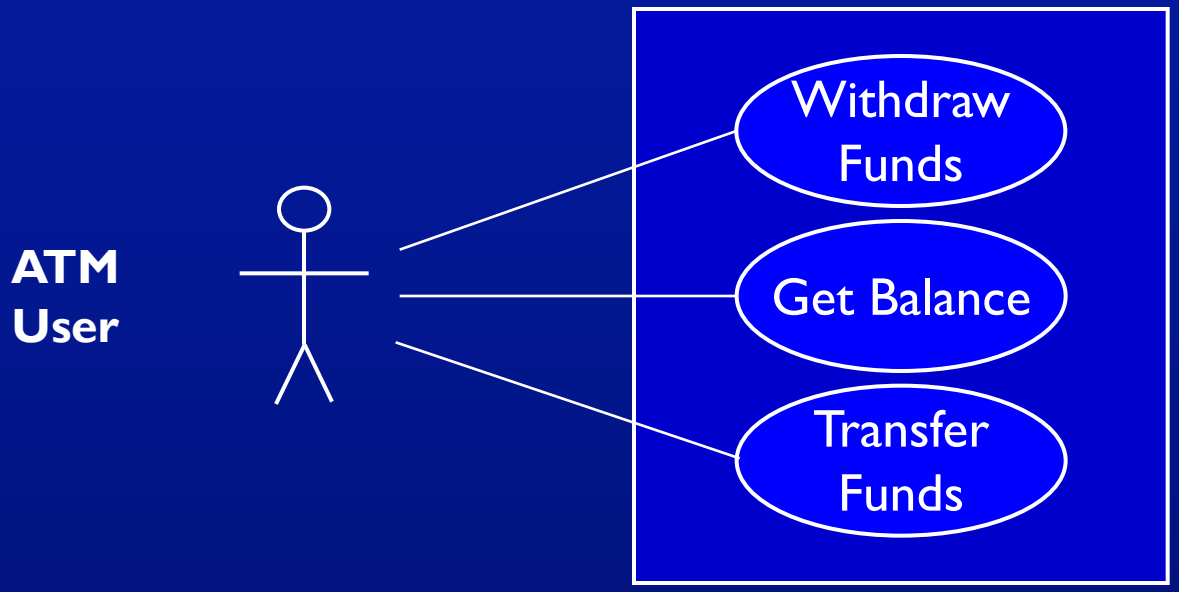
Paths in the ATM PIN Try State

- Correct PIN on first try state sequence
  - <S2.n.0, S2.n.1, S2.n.2, S2.n.3, S2.n.4, S3>
- Port Event Sequence
  - 1st digit, echo "- - - *"
  - 2nd digit, echo "- - * *"
  - 3rd digit, echo "- * * *"
  - 4th digit, echo "* * * *"
  - Enter
- Failed PIN on first try state Sequences
  - <S2.n.0, S2.n.6>
  - <S2.n.0, S2.n.1, S2.n.6>
  - <S2.n.0, S2.n.1, S2.n.2, S2.n.6>
  - <S2.n.0, S2.n.1, S2.n.2, S2.n.3, S2.n.6>
  - <S2.n.0, S2.n.1, S2.n.2, S2.n.3, S2.n.4, S2.n.6>

UML Use Cases

- UML use cases are often used to express software requirements
- They help express computer application workflow
- We won't teach use cases, but show examples

Simple Use Case Example



- Actors : Humans or software components that use the software being modeled
- Use cases : Shown as circles or ovals
- Node Coverage : Try each use case once …

Use case graphs, by themselves, are not useful for testing

Elaboration

- Use cases are commonly elaborated (or documented)
- Elaboration is first written textually
  - Details of operation
  - Alternatives model choices and conditions during execution

Elaboration of ATM Use Case

- Use Case Name : Withdraw Funds
- Summary : Customer uses a valid card to withdraw funds from a valid bank account.
- Actor : ATM Customer
- Precondition : ATM is displaying the idle welcome message
- Description :
  - Customer inserts an ATM Card into the ATM Card Reader.
  - If the system can recognize the card, it reads the card number.
  - System prompts the customer for a PIN.
  - Customer enters PIN.
  - System checks the card's expiration date and whether the card has been stolen or lost.
  - If the card is valid, the system checks if the entered PIN matches the card PIN.
  - If the PINs match, the system finds out what accounts the card can access.
  - System displays customer accounts and prompts the customer to choose a type of transaction. There are three types of transactions, Withdraw Funds, Get Balance and Transfer Funds. (The previous eight steps are part of all three use cases; the following steps are unique to the Withdraw Funds use case.)

Elaboration of ATM Use Case–(2/3)

- Description (continued) :
  - Customer selects Withdraw Funds, selects the account number, and enters the amount.
  - System checks that the account is valid, makes sure that customer has enough funds in the account, makes sure that the daily limit has not been exceeded, and checks that the ATM has enough funds.
  - If all four checks are successful, the system dispenses the cash.
  - System prints a receipt with a transaction number, the transaction type, the amount withdrawn, and the new account balance.
  - System ejects card.
  - System displays the idle welcome message.

Elaboration of ATM Use Case–(3/3)

- Alternatives :
  - If the system cannot recognize the card, it is ejected and the welcome message is displayed.
  - If the current date is past the card's expiration date, the card is confiscated and the welcome message is displayed.
  - If the card has been reported lost or stolen, it is confiscated and the welcome message is displayed.
  - If the customer entered PIN does not match the PIN for the card, the system prompts for a new PIN.
  - If the customer enters an incorrect PIN three times, the card is confiscated and the welcome message is displayed.
  - If the account number entered by the user is invalid, the system displays an error message, ejects the card and the welcome message is displayed.
  - If the request for withdraw exceeds the maximum allowable daily withdrawal amount, the system displays an apology message, ejects the card and the welcome message is displayed.
  - If the request for withdraw exceeds the amount of funds in the ATM, the system displays an apology message, ejects the card and the welcome message is displayed.
  - If the customer enters Cancel, the system cancels the transaction, ejects the card and the welcome message is displayed.
- Postcondition :
  - Funds have been withdrawn from the customer's account.

Wait A Minute ...

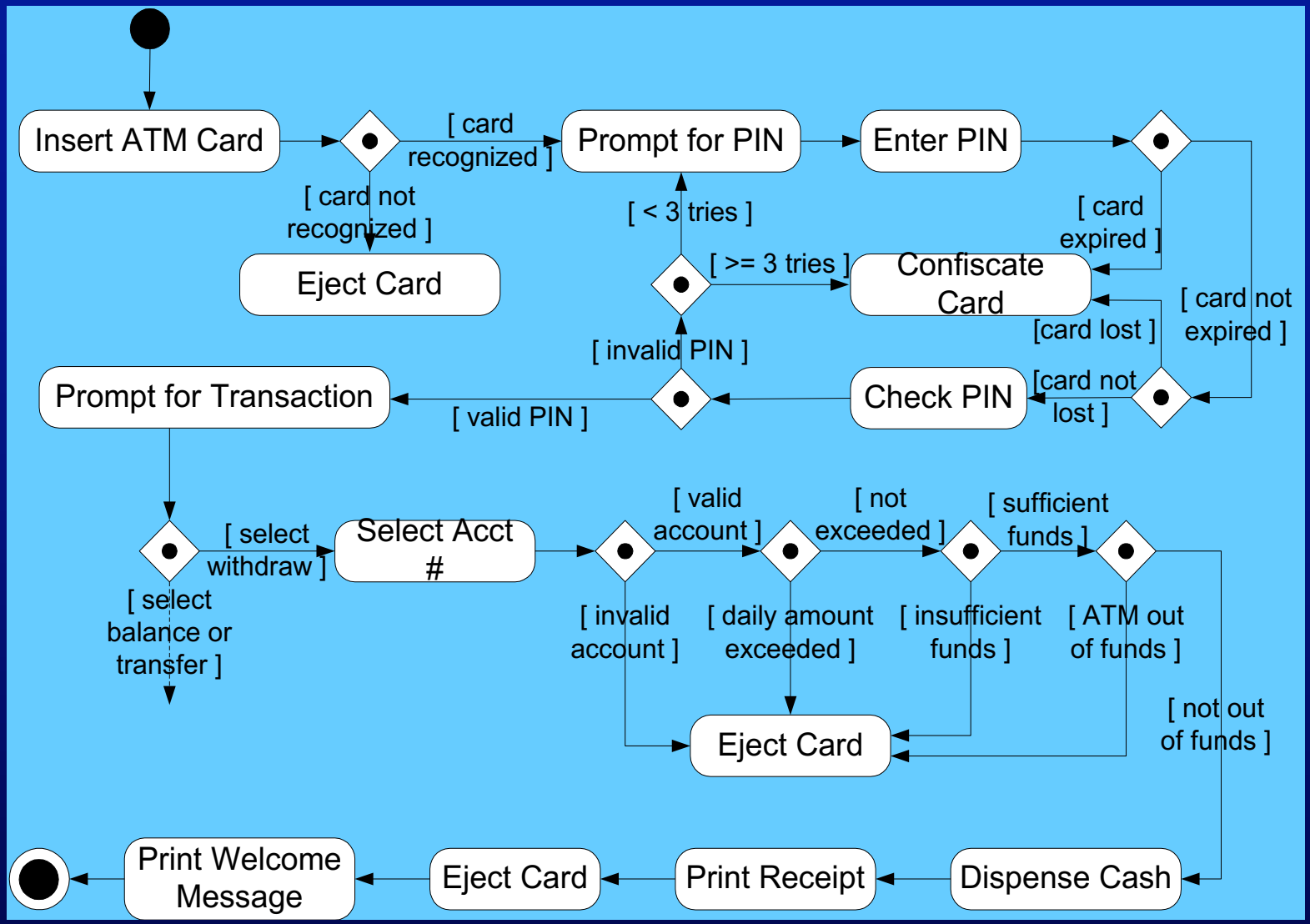
- What does this have to do with testing ?
- Specifically, what does this have to do with graphs ???
- Remember our admonition : Find a graph, then cover it!
- Beizer suggested "Transaction Flow Graphs" in his book
- UML has something very similar :

Activity Diagrams

Use Cases to Activity Diagrams

- Activity diagrams indicate flow among activities
- Activities should model user level steps
- Two kinds of nodes:
  - Action states
  - Sequential branches
- Use case descriptions become action state nodes in the activity diagram
- Alternatives are sequential branch nodes
- Flow among steps are edges
- Activity diagrams usually have some helpful characteristics:
  - Few loops
  - Simple predicates

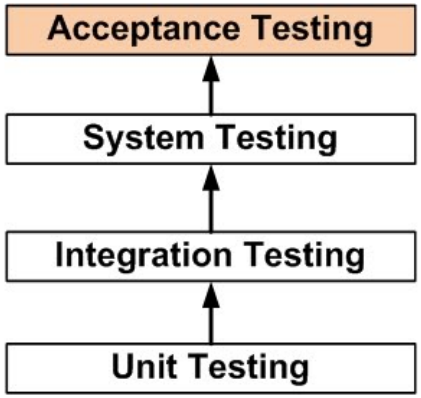
ATM Withdraw Activity Graph



Acceptance Testing

What is Acceptance Testing?

- The goal is to demonstrate system is ready for operational use, also known as user acceptance testing (UAT) and end-user testing.
- A way to check if a previously defined contract between the supplier and the customers is still on track.
- The purpose is to evaluate the system's compliance with business requirements and assess whether it is ready for delivery.



"Formal testing with respect to user needs, requirements, and business processes conducted to determine whether or not a system satisfies the acceptance criteria and to enable the user, customers or other authorized entity to determine whether or not to accept the system." ***ISTQB***

What is Alpha Testing?

- Internal acceptance testing performed mainly by the in-house software QA and testing teams–not accessible by the end-users/market.
- It is the last testing activity done by the test teams at the development site and before releasing the software for Beta testing.
- In the first phase of alpha testing, the software is tested by in-house developers during which the goal is to catch bugs quickly. In the second phase of alpha testing, the software is given to the software QA team for additional testing.
- It is carried out in a lab environment and usually testers are internal employees of the organization.
- Alpha testing can also be done by inviting some potential end-users of the application. But still, this is a form of in-house acceptance testing.

What is Alpha Testing?

- Stakeholders of Alpha testing phase are usually the engineers (in-house developers), quality assurance teams, and the product management teams.
- Encourage different sectors within the organization (sales, management, etc.) to use the product and experience it. They are also encouraged to provide feedback, suggestions to improve the product which actually results in increasing the quality of the product
- Bugs directly coming from the later testing phases (Beta Testing phase/after Production launch) leave bad remarks on the product and on organization reputation that developed the product.
- These bugs also cause delays in production launch, more effort (both time and resource) is required in fixing them.

What is Beta Testing?

- Also known as field testing, takes a place in the customer's environment.
- It involves extensive tests done by a group of end-users who examine the system in their environment. These Beta testers then provide feedback, which in turn leads to improve the software application.
- This is the final testing phase where the companies release the product to external user groups outside the organisation. Most companies gather users' feedback in this release.
- **In short, Beta testing can be defined as – the testing carried out by real users of the software application in a real environment.**

What is Beta Testing?

- The selection of beta test groups can be done based on the supplier company needs.
- Beta version is released to a limited number of end-users of the product to obtain feedback on the product quality. It might be also released openly to be examined by any user.
- Fixing the issues in the beta release can significantly reduce the development cost as most of the minor glitches get fixed before the final release.
- Until now many big companies have successfully used beta versions of their most anticipated applications.
- It is the final test before officially shipping a product to the customers.

Types of Beta Testing

- **Traditional Beta testing:** a product is distributed to the target market, and related data is gathered in all aspects. This data is used to improve the product.
- **Public Beta Testing:** a product is publicly released to the outside world via online channels and data can be gathered from anyone. Based on feedback, product improvements can be done. For example, Microsoft conducts large Beta tests for its OS – Windows 8 and 10 before officially releasing them.
- **Technical Beta Testing:** a product is released to an internal group of organization employees and gather feedback/data from these employees.
- **Focused Beta:** a product is released to the market for gathering feedback on specific features of the program. For example, important functions of the product.

Ad-hoc Testing

- An informal testing type with an aim to break the system. This testing is usually an unplanned activity. Also, it does not follow any test design techniques to create test cases.
- Ad hoc testing does not follow any structured way of testing and it is randomly done on any part of application. The main goal of this testing is to find defects by just random checking.
- Ad hoc testing can be achieved with the testing technique called Error Guessing. Error guessing can be done by the people having enough experience on the system and the application domain to "guess" the most likely source of errors.
- **Monkey testing** is often defined as a type of ad-hoc testing that deals with random inputs. If a monkey uses a computer, it will randomly perform actions on the system. Similarly, testers acts like monkeys by applying random test cases on the system to find bugs/errors without predefining any test plan.

How is Acceptance Testing different from Functional Testing?

- **Acceptance Testing** consists of a set of testing steps, which verify if specific requirements are working for the customers. If the customer and the supplier agree, then the delivery processes of software product can start legally and practically.

- **Functional testing**, on the other hand, tests specific requirements and functions of the software. It lacks the user involvement. It could conclude that the software meets its specifications. However, it does not verify if it actually works for the actual user.

Example: say that Facebook launches a new feature, allowing Facebook users to send postcards to family & friends. Technically, the implemented solution works. Testers also can use it – however due to lack of interest and need, no one wants to send printed postcards. Functional testing would go well, usability tests would go fine too, but the user acceptance testing would probably fail as Facebook users do not demand to send postcards within Facebook.

Additional Types of Acceptance Testing

- **Contract Acceptance Testing:** it ensures that a developed product is tested against certain criteria and specifications which are predefined and agreed upon in a contract. The testers defines the relevant criteria and specifications in the products that match the contract content.
- **Regulation Acceptance Testing:** it examines whether the software complies with the regulations. This includes governmental and legal regulations. Also known as compliance acceptance testing.
- **Operational Acceptance:** it ensures there are workflows in place to allow the software or system to be used. This should include workflows for backup and recovery plans, user training, and various maintenance processes and security checks. Also it known as production acceptance testing.
- **Black Box Testing:** it is often categorized as functional testing, but can, to some extent, be seen as a type of user acceptance testing