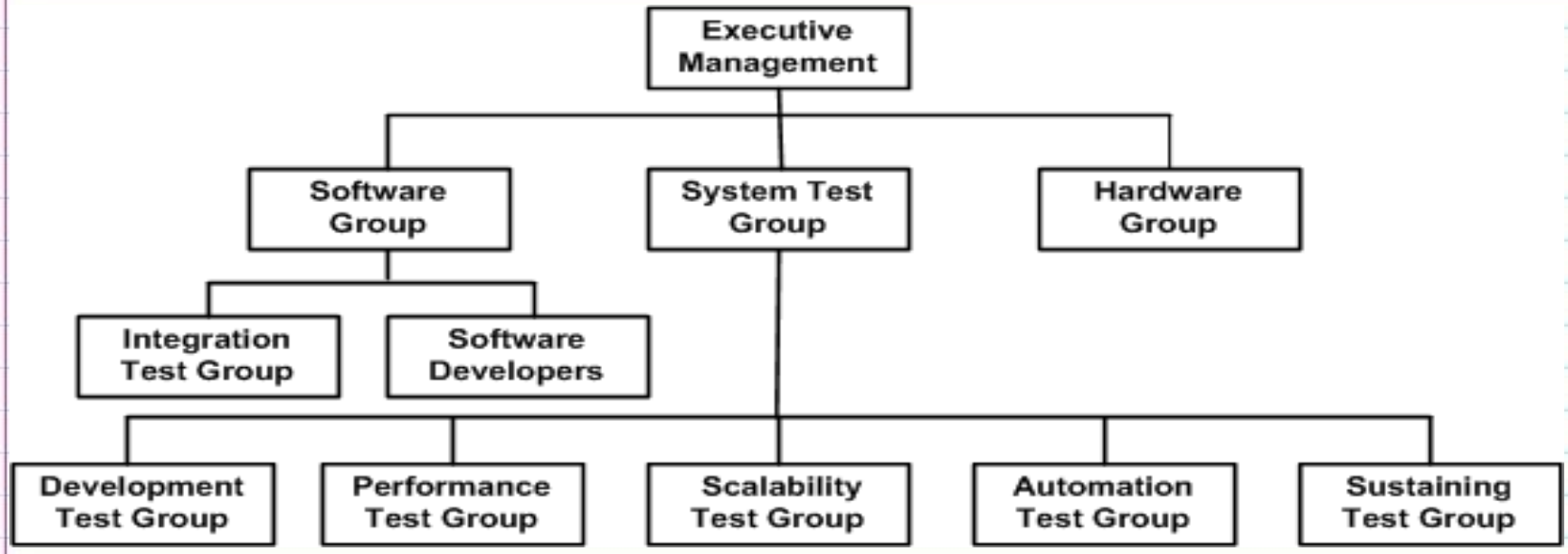
Software Testing Overview - Unit, Integration and System Testing

Dr. John H Robb, PMP
UTA Computer Science and Engineering

Test Team Organization and Management



A Notional Organization Structure of test groups

- Unit and Integration testing is many times performed by the developers. System level and Acceptance testing is typically performed by independent organizations
- Technology obsolescence in software development and test is very high - new training must be continually provided to maintain an efficient work force
- To retain test engineers management must recognize the importance of testing efforts at par with development effort

Test Documentation

- Testing is a very complicated task
  - Mature organizations have a test process, test standard, and test plan
  - These address all test activities Unit, Integration, and System testing including regression tests
- Test Process
  - Test methodologies (specification based, code based, etc.)
  - Test organization (how tests are organized, where kept, how changed, etc.)
  - Who approves various test plans, documents, and artifacts.
  - Organizational roles and responsibilities
  - Tools and automation
  - Training of personnel (new and existing)
- Test Standard
  - Describes when testing is complete and the various activities used to support test completion

Test Plan

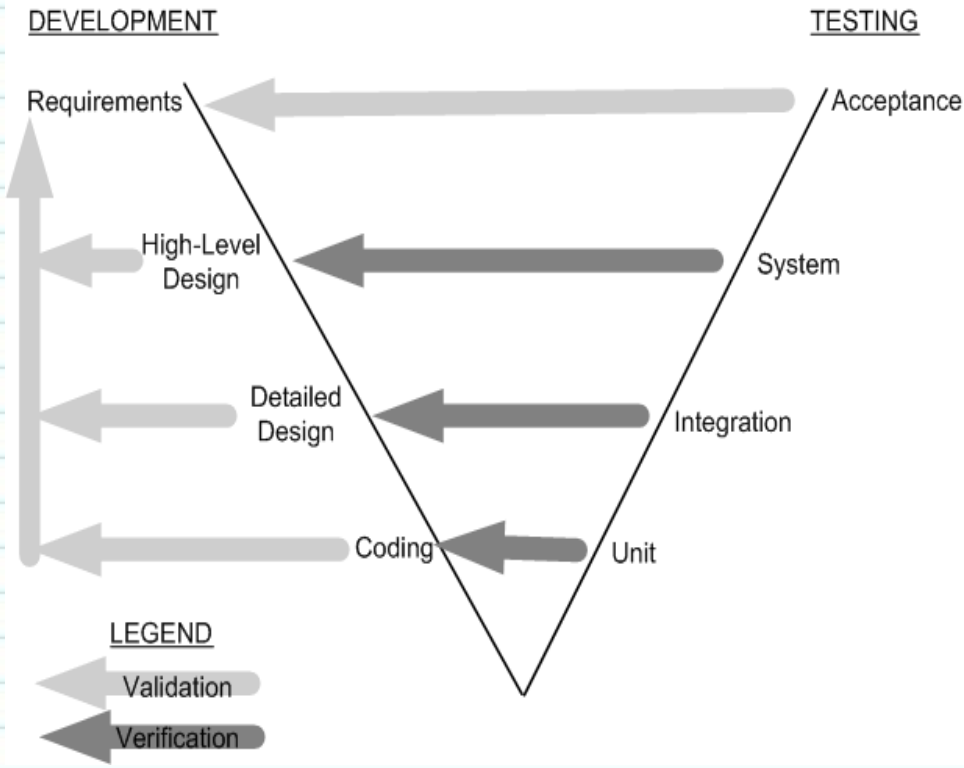
- The purpose is to get ready and organized for test execution
- A test plan provides a:
  - Framework
    - A set of ideas, facts or circumstances within which the tests will be conducted
  - Scope
    - The domain or extent of the test activities
  - Details of resource needed
  - Effort required
  - Schedule of activities
  - Budget
- Test objectives are identified from different sources
- Each test case is designed as a combination of modular test components called test steps
- Test steps are combined together to create more complex tests

Test Results

- These are the results of the test execution across all test activities Unit, Integration, and System testing including regression tests
- They are typically tied to a specific product configuration (such as a specific test or delivery milestone)
- These are used to demonstrate the software function correctly works as specified by the requirements
- Test results may indicate one or more deficiencies as described earlier in the course
- Test results in this course are the JUnit passing green bar (expanded) and JaCoCo coverage. It is typical to have to capture this in the CM system for each build.

Testing Level

- Unit testing
  - Individual program units are tested in isolation. Unit testing is the lowest level of testing performed.
- Integration testing
  - Units are assembled to construct larger groups and tested
- System testing
  - Includes wide spectrum of testing such as functionality, and stress
- Acceptance testing
  - Customer's expectations from the system



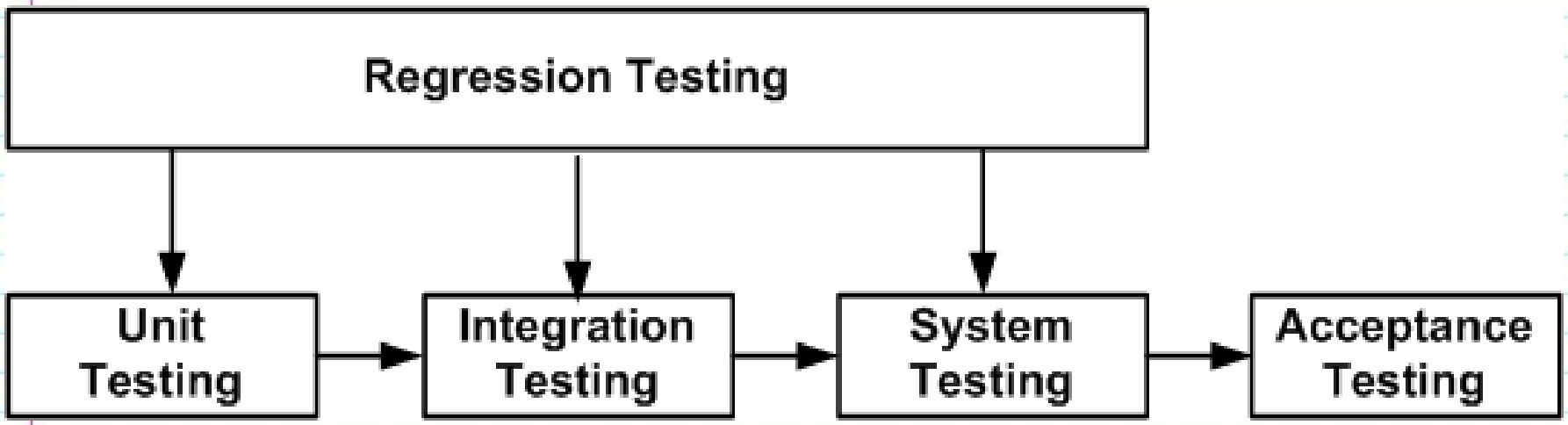
Development and testing phases in the V model

During product evolution two kinds of testing occurs

1) testing of new features

2) testing to ensure that the change didn't impact some previously working function - next slide "Regression testing"

Re-Testing Level



- Regression testing is performed at software testing levels - many times in parallel
- Regression testing is performed to ensure that the software function wasn't impacted because of
  - Unforeseen linkages/impacts between software components
  - Subtle timing or tasking issues that cannot be verified other than test
  - Configuration management or build problems

Test Areas of Focus

- Each area of test has a specific focus

Test Level	Area of focus
Unit	1) execution of basis paths 2) BVs/ECPs 3) Mathematical testing
Integration	1) testing of interface between classes 2) testing of behavior between classes
System	1) end to end functionality 2) performance testing

- Unit testing can find the most defects, but it cannot be used to find defects between classes and performance issues

Unit Testing - Testing in Isolation

Unit Testing

- Involves testing a single isolated unit
- Note that unit testing allows us to isolate the errors to a single unit
  - we know that if we find an error during unit testing it is in the unit we are testing
- Units in a program are not isolated, they interact with each other. Possible interactions:
  - calling procedures in other units
  - receiving procedure calls from other units
  - sharing variables (possible but DON'T DO THIS)
- For unit testing we need to isolate the unit we want to test, we do this using
  - test drivers
  - stubs/mock objects

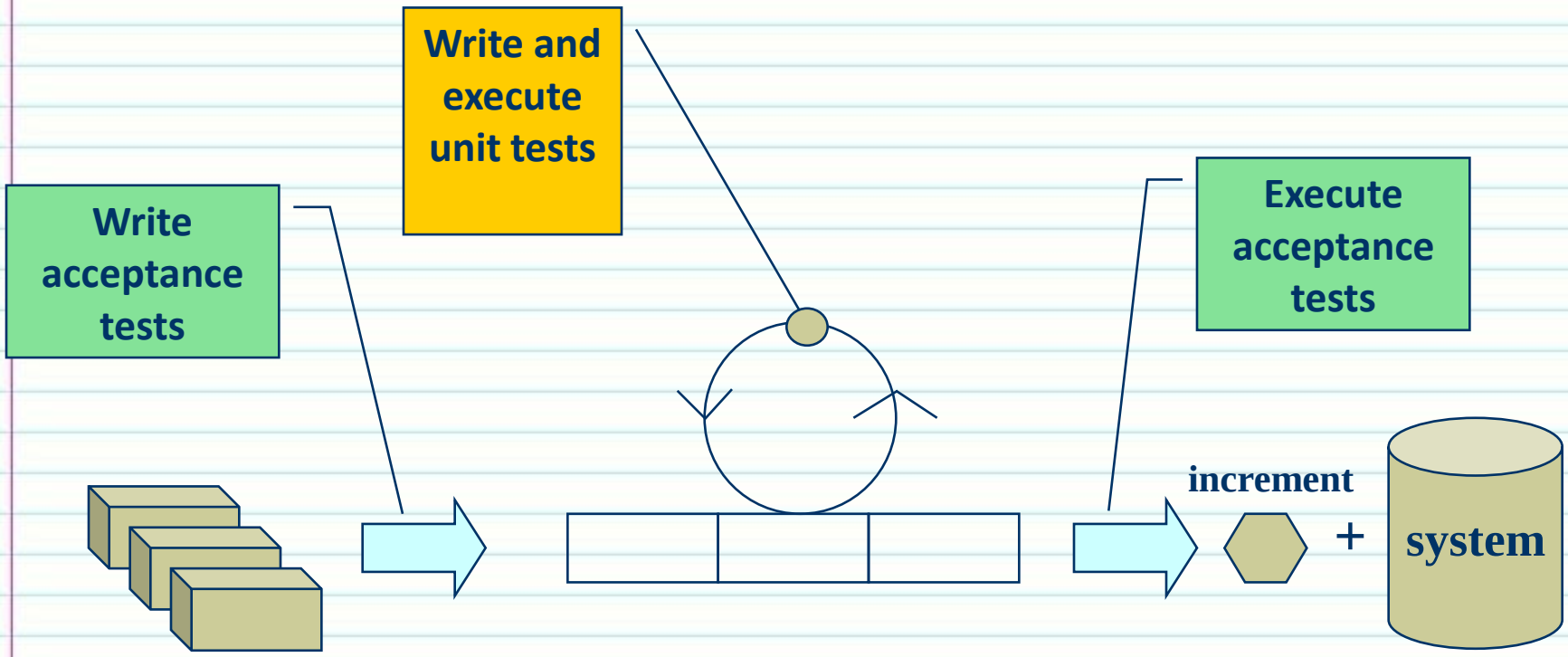
Advantages of Unit Testing

- Advantages of unit testing (testing in isolation):
  - Faster Troubleshooting
    - Unit tests smaller amounts of code, easily isolating points of error and narrowing the scope for errors.
  - Faster Development
    - Less time troubleshooting unit tests
    - Encourage aggressive refactoring resulting in better design and easier maintenance.
    - Adding enhancements can be performed with confidence that existing behavior of the system continues to operate in the same manner.
  - Better Design
    - Unit tests make developers focus more on the contracts (methods) of a class and those classes that might use it

Advantages of Unit Testing (cont.)

- Excellent Regression Tool
  - Major refactoring and restructuring of the code can be performed.

Iterative Software Development



Prioritized functionalities

"Written *before*"

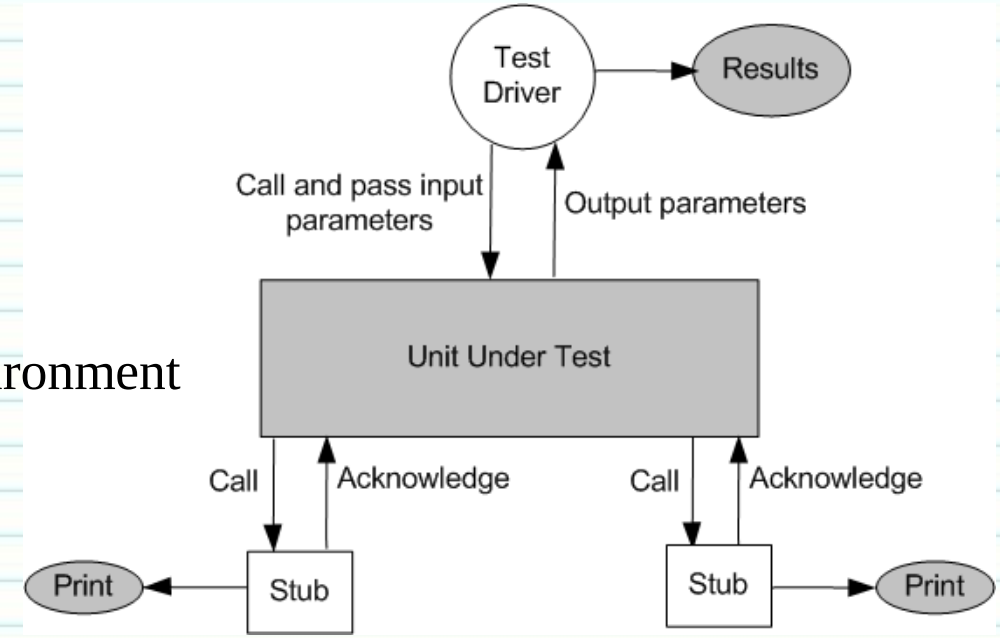
"Executed after the development"

Unit Testing is the key to most testing efforts

Unit Testing Framework

- The caller of the unit under test (UUT) is known as *test driver*
  - A *test driver* is a program that invokes the UUT
  - It provides input data to unit under test and report the test result
- The emulation of the units called by the UUT are called *stubs or mocks*
  - It is a dummy program
- The *test driver*, the *stubs*, and *mock objects* are together called *scaffolding* - this term is not used that much
- The low-level design document provides guidance for selection of input test data

Unit test environment



JUnit is a test driver

Unit Testing Framework (cont.)

- Typically the driver is a stand-alone class/file that is not part of the delivered software.
  - Normally for a product there are two trees in the CM system - one for the production build and another for test
  - It is stand-alone to make it easily separatable from the deliverable software.

- When the UUT calls another method outside its class it is typically stubbed or mocked out
  - This means that the test case supplies the correct return value from that method (instead of actually calling it and getting the return)
  - This helps to isolate one unit from another so that they can independently change
  - As the units mature they may either be un-stubbed or stay stubbed

Why use a testing framework?

- Each class must be tested when it is developed
- Each class needs a regression test
- Regression tests need to have standard interfaces
- Thus, we can build the regression test when building the class and have a better, more stable product for less work

Drivers, Stubs, and Mock Objects for Unit Testing

- Driver
  - A method (or main) that passes test case data to the component being tested, verifies the returned results, and summarizes pass/fail outcomes
  - JUnit is such a driver, some drivers are developed by hand without tools
- Test double
  - Substitution for production code during test - typically either a mock or stub (fakes and spies are also used)
  - Both stubs and mocks are used to force the method being called to return the needed value to the UUT
- Stubs and Mocks are interchangeable - new this semester
  - The literature is not very consistent on what the differences are so I am now saying they are interchangeable - any difference is subtle
  - Dynamic "wrapper" used to intercept a method call/return using a tool
  - They use methods exact interface and are replaced by production code
- Drivers, stubs, and mock objects both represent overhead
  - This in addition to production code

Advantages of JUnit

- The **TestCase** class is where each of the unit tests are defined. These create stand-alone classes distinct from the developed product.

- The **TestSuite** class is used to manage unit tests. A TestSuite can embed a single, or multiple tests, and even single or multiple test suites. The recursive nature allows a hierarchy of unit tests to be build. It also provides a way for developers to quickly specify the unit tests that they may want to execute locally.

- The **TestDecorator** class is part of the extensions to the junit framework. These classes provide additional classes that are not considered essential to writing unit tests, but may provide useful functionality in writing tests. Test decorator subclasses can be introduced to add behaviour before or after a test is run.' An example of the TestSetup is provided in the document.

Disadvantages of JUnit

- **GUIs** typically require a human to drive the process and analyze the outcome. There are a number of techniques for validating GUI classes - JUnit is still limited to manual checks. There are a number of programs available that allow recording and execution of scripts against a GUI.

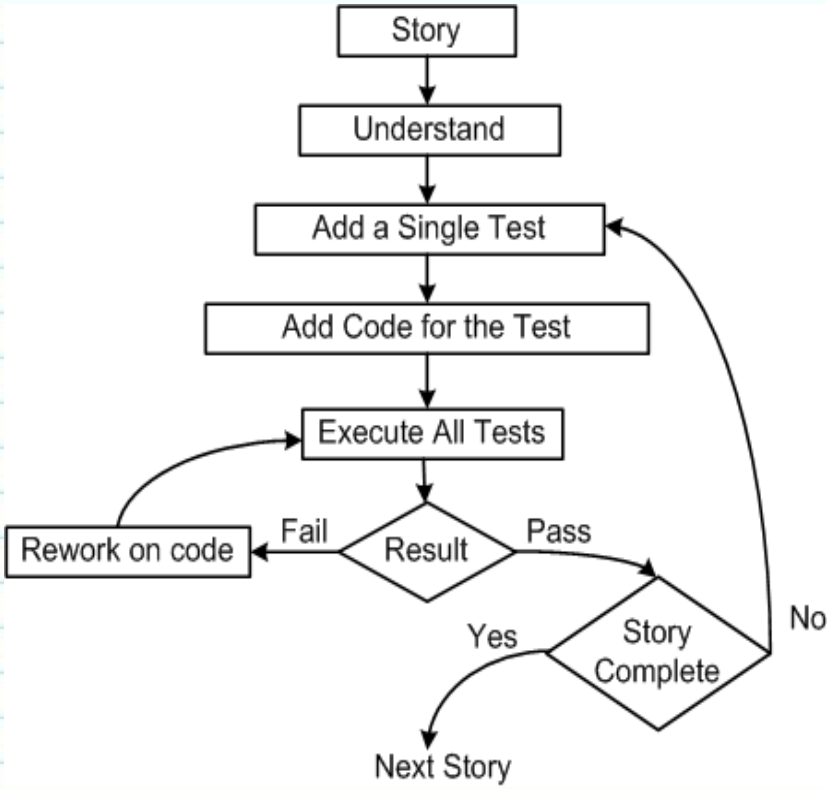
- **Test Reporting - Artima** provides an extension to Junit by enhancing the reporting mechanism of JUnit. It allows for multiple reports to be generated from a single unit test execution and builds upon a number of other issues that JUnit has in certain environments.
- http://www.artima.com/suiterunner/

Execution of Unit Tests

- Some approaches (Scrum) perform nightly unit testing
  - There are pros and cons associated with this approach
  - Pros: ensures that each interface (method) still works especially as we are working toward a new build
  - Cons: busywork - I prefer to test a unit when the code for that unit is changed only (this includes changes to the interface or inheritance structure).

Unit Testing in eXtreme Programming

1. Pick a requirement, i.e., a story
2. Write a test case that will verify a small part of the story and assign a fail verdict to it
3. Write the code that implement particular part of the story to pass the test
4. Execute all test
5. Rework on the code, and test the code until all tests pass
6. Repeat step 2 to step 5 until the story is fully implemented



Test-first process in XP

Unit Testing in TDD

Three laws of Test Driven development (TDD)

- One may not write production code unless the first failing unit test is written
- One may not write more of a unit test than is sufficient to fail
- One may not write more production code than is sufficient to make the failing unit test pass

EasyMock Examples

Why Mock Objects?

- When we have a method in one class that calls a method in another class
  - The other method may not yet exist
  - It may not be tested yet
  - So we don't want to actually call that method

- We want to "intercept" that call to that method and return the value(s) we need for our unit test
  - The production code will still have a call to that method, but we will do something to intercept that call with a dummy method
  - The dummy method has traditionally been called a mock (or stub)

- The mock or stub allows us to test the unit in isolation, but with the same code actually call the method

- Use EasyMock to intercept the mock object at runtime.

Easy Mock Overview

- Framework for creating mock objects at run time.
- Open source tool available at easymock.org.
- Download the EasyMock JAR from M14 files and install in your build path

Steps to use Easy Mock

1. Create the Interface for the method being mocked
2. Create the mock object.
3. Tell the mock object how to behave when called (this is the "expect" - this intercepts the method call and returns the result you specify.
4. Activate the mock object (this is called “replay” in the EasyMock documentation)
5. Run the test(s)

How to use easy mock:

- Create a mock:

  SomeInterface mock = createMock(SomeInterface.class);

- Record Behavior:

  expect(mock.doStuff(“argument”)).andreturn(“returnValue”);

- Replay behavior:

  replay(mock);

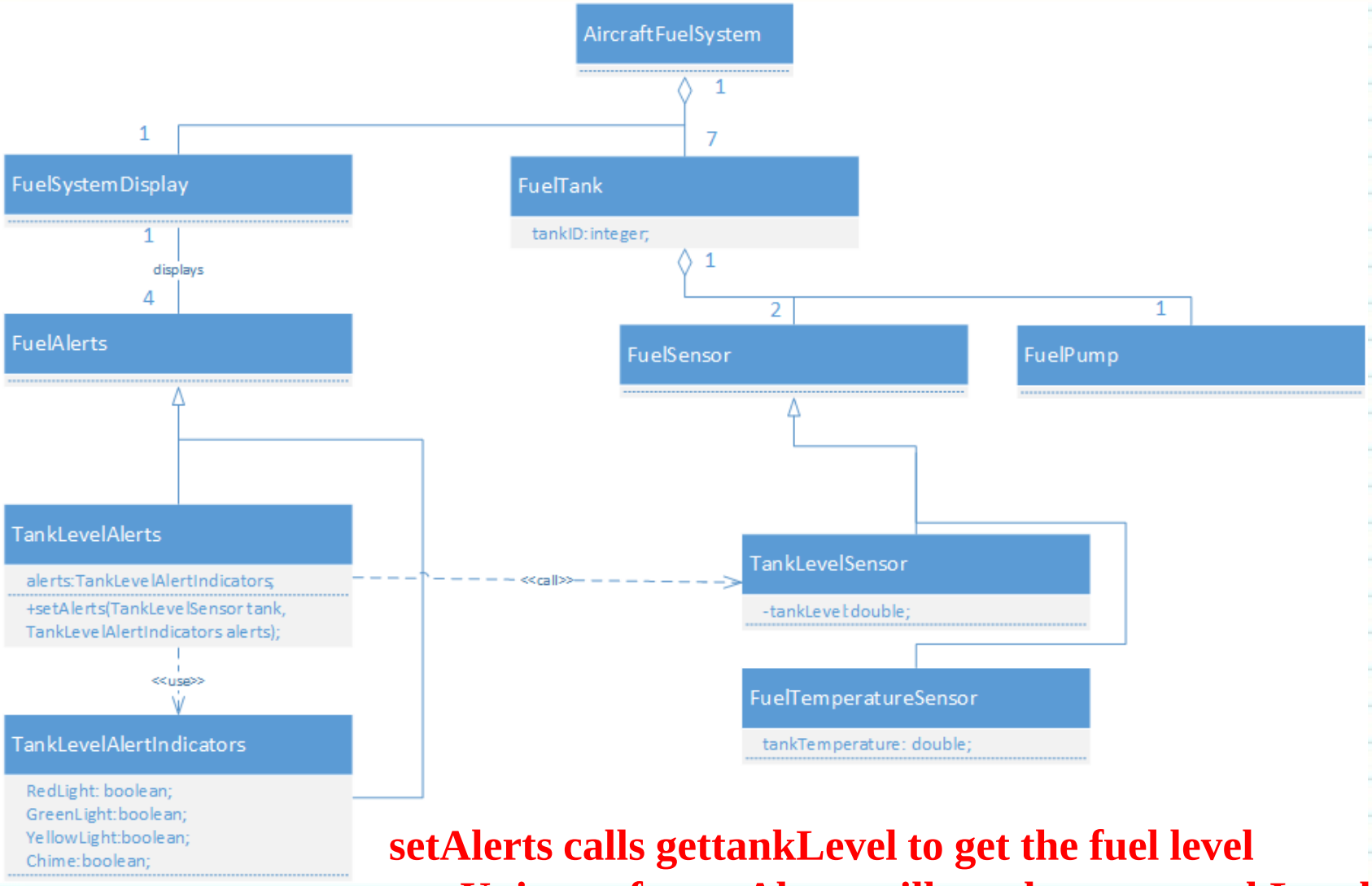
- Executing the code we want to test

  assume some method calls.

Example System

- We have an aircraft fuel system
- The aircraft fuel system has the following:
  - A display
  - 7 tanks (each with the following)
    - 2 sensors (temperature, fuelLevel)
    - Fuel Pump
- TankLevel is a double with 0.1 gallons significance and a range of
  - 0.0:100.0 gallons inclusive

Example System (UML)



setAlerts calls gettankLevel to get the fuel level
out Unit test for setAlerts will mock out gettankLevel

Example System (Java)

```
1  package FuelExampleFiles;
2
3  public class TankLevelAlerts {
4
5  public void setAlerts (TankLevelSensor tank, TankLevelAlertIndicators alerts) {
6
7          double tank_level;
8          alerts.setChime(false); alerts.setRed_light(false); alerts.setYellow_light(false); alerts.setGreen_light(false);
9
10         tank_level=tank.gettankLevel();
11         if (tank_level<30.0)
12             alerts.setRed_light(true);
13         else
14             if (tank_level<=40.0)
15                 alerts.setYellow_light(true);
16             else
17                 if (tank_level<50.0)
18                     alerts.setGreen_light(true);
19
20         if (tank_level > 15.0)
21             alerts.setChime(true);
22     }
23 }
```

mock gettankLevel()

1) we want gettankLevel() to provide the actual values we need for our Unit test without actually calling it

2) we want to preserve the call to gettankLevel() so that when it is built with the production build it WILL call the real gettankLevel()

3) EasyMock provides us with a way to do this without having to change the source code of the method under test

Example System (Java) - cont.

```java
package FuelExampleFiles;

public class TankLevelAlertIndicators {

    boolean Green_light, Yellow_light, Red_light, Chime;

    public boolean isGreen_light() {
        return Green_light;
    }

    public void setGreen_light(boolean green_light) {
        Green_light = green_light;
    }

    public boolean isYellow_light() {
        return Yellow_light;
    }

    public void setYellow_light(boolean yellow_light) {
        Yellow_light = yellow_light;
    }

    public boolean isRed_light() {
        return Red_light;
    }

    public void setRed_light(boolean red_light) {
        Red_light = red_light;
    }

    public boolean isChime() {
        return Chime;
    }

    public void setChime(boolean chime) {
        Chime = chime;
    }
}
```

TankLevelAlertIndicators.java

Example System (Java) - cont.

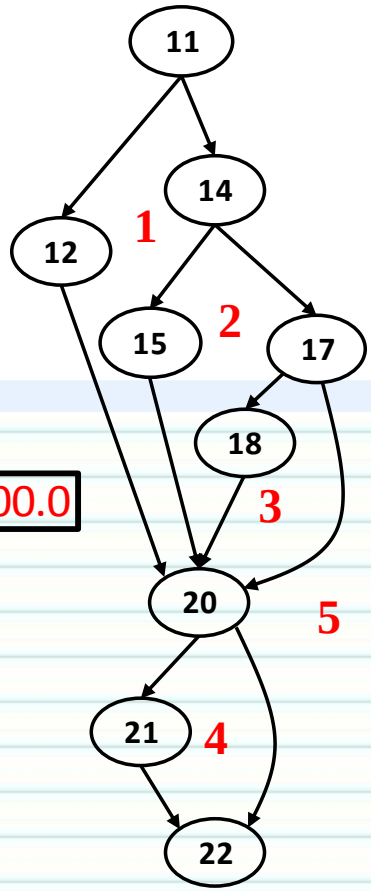
```
1  package FuelExampleFiles;
2
3  public interface TankLevelSensor {
4
5      public double gettankLevel();
6  }
```

1) gettanklevel() is the method we are mocking - it is in the class TankLevelSensor
2) we create an interface of that class with the method signature in it
3) we will create a mock object in EasyMock of this interface which will let us intercept the call
4) during a production build the real TankLevelSensor class will be used (by using the class ... implements TankLevelSensor)

Example System (CFG and ECP/BV)

```
1  package FuelExampleFiles;
2
3  public class TankLevelAlerts {
4
5  public void setAlerts (TankLevelSensor tank, TankLevelAlertIndicators alerts) {
6
7          double tank_level;
8          alerts.setChime(false); alerts.setRed_light(false); alerts.setYellow_light(false); alerts.setGreen_light(false);
9
10         tank_level=tank.gettankLevel();
11         if (tank_level<30.0)
12             alerts.setRed_light(true);
13         else
14             if (tank_level<=40.0)
15                 alerts.setYellow_light(true);
16             else
17                 if (tank_level<50.0)
18                     alerts.setGreen_light(true);
19
20         if (tank_level > 15.0)
21             alerts.setChime(true);
22     }
23 }
```

0.0 15.0	15.1 29.9	30.0 40.0	40. 1 49.9	50.0 100.0

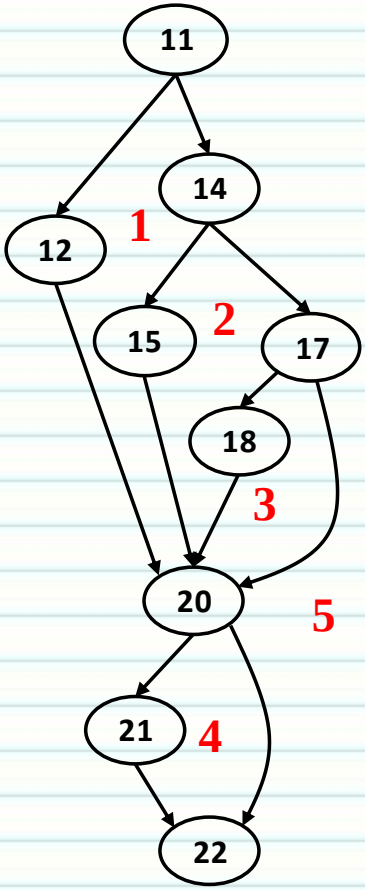


Example System (Test Case Table)

Test Case	Inputs	Expected Outputs				
	TankLevel	RedLight	YellowLight	GreenLight	Chime	Basis Path
1	29.9	TRUE	FALSE	FALSE	TRUE	11-12-20-21-22
2	40.0	FALSE	TRUE	FALSE	TRUE	11-14-15-20-21-22
3	49.9	FALSE	FALSE	TRUE	TRUE	11-14-17-18-20-21-22
4	50.0	FALSE	FALSE	FALSE	TRUE	11-14-17-20-21-22
5	15.0	TRUE	FALSE	FALSE	FALSE	11-14-17-20-22
6	0.0	TRUE	FALSE	FALSE	FALSE	-
7	15.1	TRUE	FALSE	FALSE	TRUE	-
8	30.0	FALSE	TRUE	FALSE	TRUE	-
9	40.1	FALSE	FALSE	TRUE	TRUE	-
10	100.0	FALSE	FALSE	FALSE	TRUE	-

0.0	15.0	15.1	29.9	30.0	40.0	40. 1	49.9	50.0	100.0

Red values indicate untested BVs by basis path



Example System (JUnit test code)

```
1  package FuelExampleFiles;
2
3  import static org.junit.Assert.assertEquals;
10
11 @RunWith(JUnitParamsRunner.class)
12 public class TankLevelAlertsTest {
13
14     double tankLevel;
15     TankLevelSensor mockobj;
16     TankLevelAlertIndicators alerts;
17     TankLevelAlerts tankAlerts;
18
19     @Before
20     public void setUp() {
21         alerts = new TankLevelAlertIndicators();
22         tankAlerts = new TankLevelAlerts();
23         mockobj = EasyMock.strictMock(TankLevelSensor.class);
24     }
25
26     @Test
27     @FileParameters("src/FuelExampleFiles/FuelLevelExampleTestCase.csv")
28     public void test (int testCaseNumber, double tankLevel,boolean red_light, boolean yellow_light,
29         EasyMock.expect(mockobj.gettankLevel()).andReturn(tankLevel);
30         EasyMock.replay(mockobj);
31         alerts.setRed_light(!red_light);
32         alerts.setYellow_light(!yellow_light);
33         alerts.setGreen_light(!green_light);
34         alerts.setChime(!chime);
35         // Default value of a boolean in Java is false
36         tankAlerts.setAlerts(mockobj, alerts);
37         assertEquals(red_light,alerts.isRed_light());
38         assertEquals(yellow_light,alerts.isYellow_light());
39         assertEquals(green_light,alerts.isGreen_light());
40         assertEquals(chime,alerts.isChime());
41     }
42 }
```

import org.easymock.EasyMock;

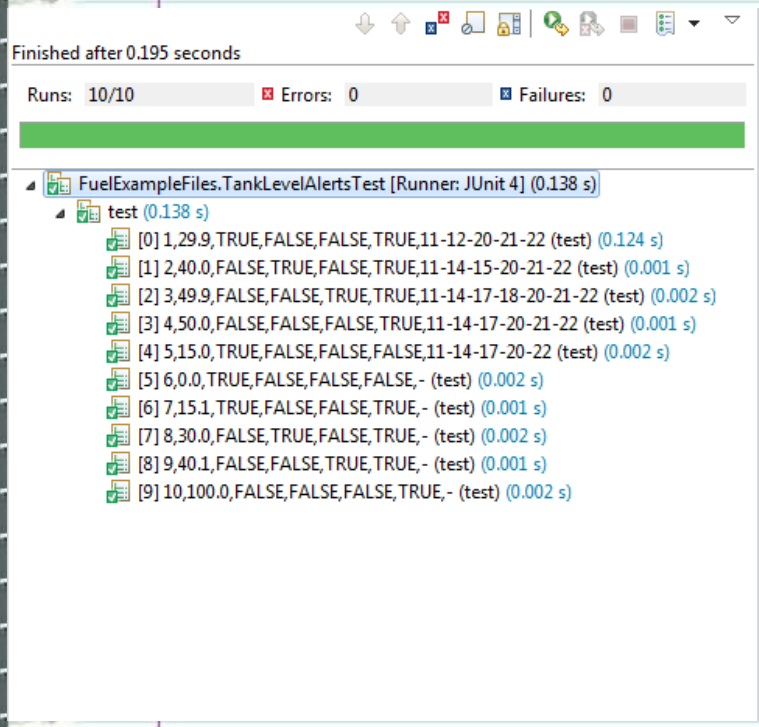
declare the mockobj

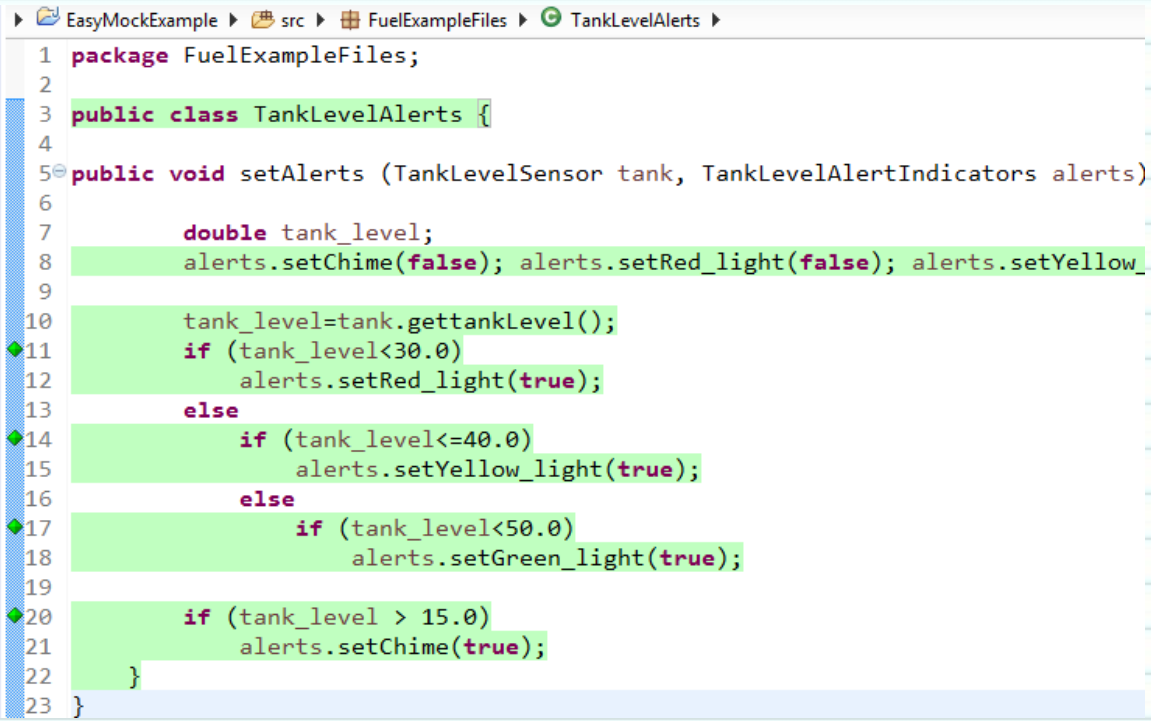
construct the mockobj

intercept the call

replay (activate) the mock

Example System (JUnit Results)





Example System (Demo)

In Eclipse

Integration Testing - Putting Classes Together

Integration Testing

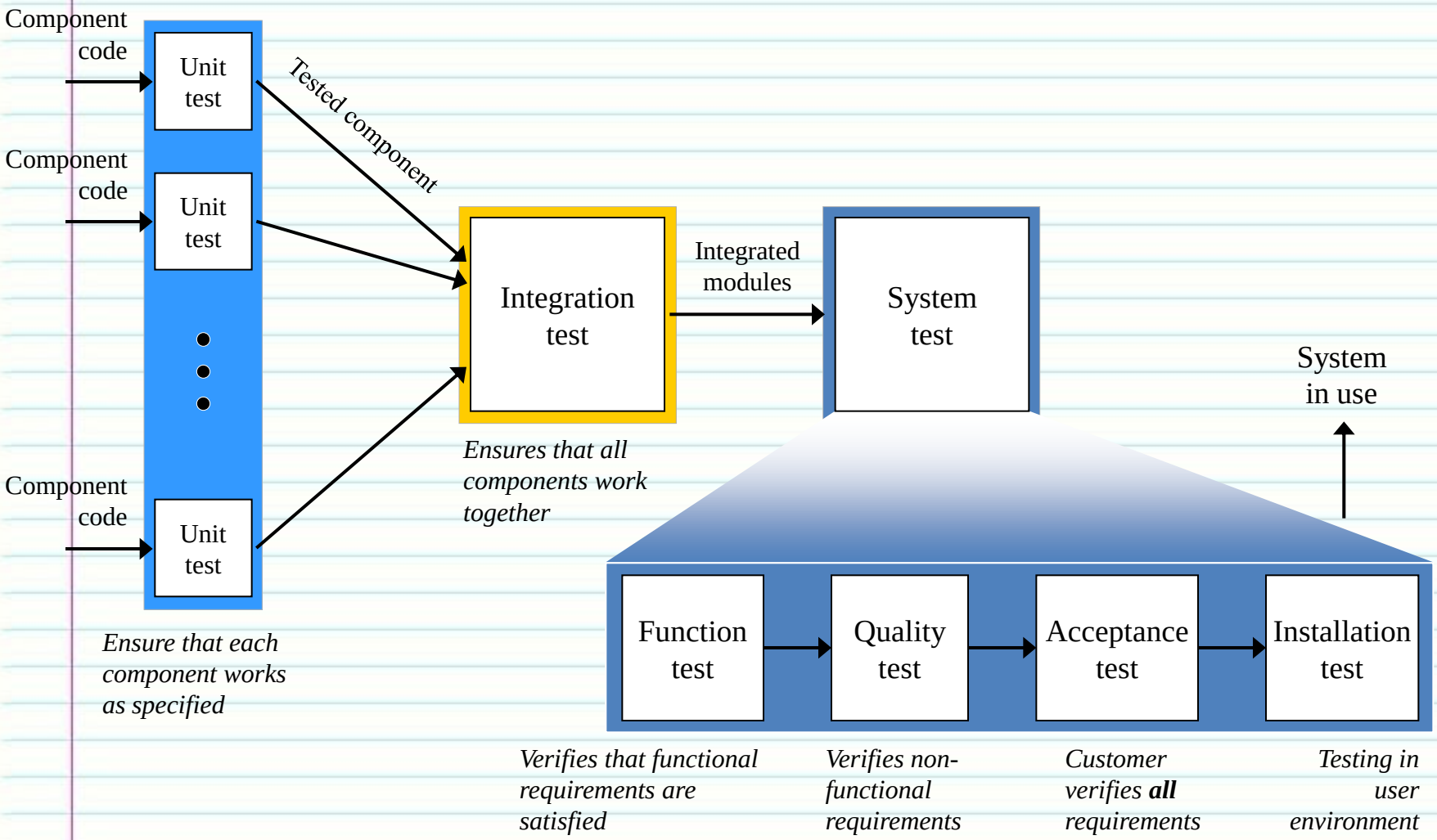
- As mentioned earlier, the object oriented approach of developing highly cohesive, low-coupled, encapsulated and information hiding code places great emphasis on integrating these classes together - this is where the complexity lays.

- ISTQB defines Integration Testing as "integration testing: Testing performed to expose defects in the interfaces and in the interactions between integrated components or systems. See also component integration testing, system integration testing.

Integration Testing (cont.)

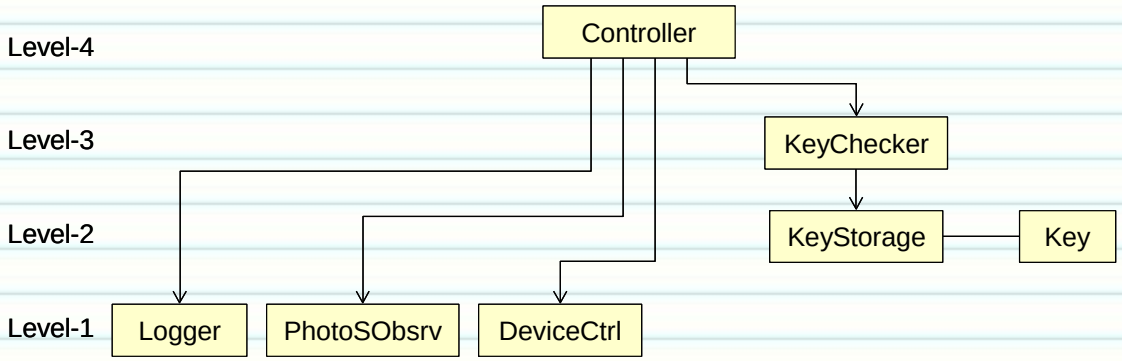
- There are several methods of performing integration testing - testing across classes
- Horizontal Integration Testing
  - "Big bang" integration
  - Bottom-up integration
  - Top-down integration
  - Sandwich integration
- Vertical Integration Testing
- As mentioned previously we need to test
  - Methods
  - Within a single class
  - Across Classes
  - As a whole
- This is what integration seeks to achieve

Logical Organization of Testing

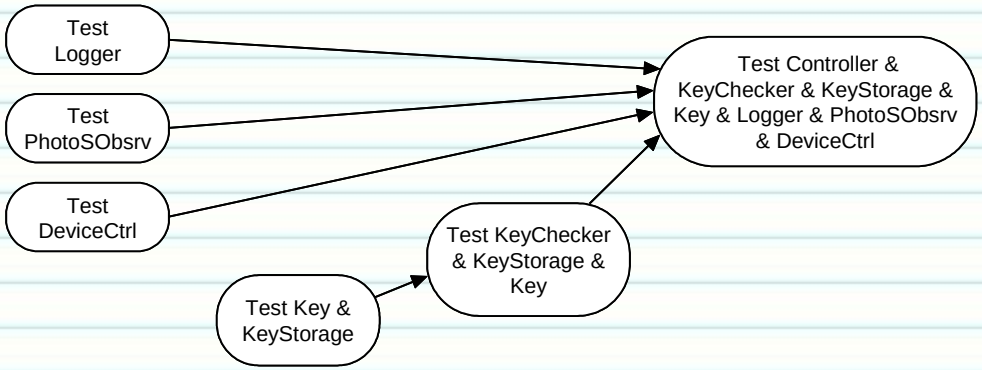


Horizontal Integration Testing

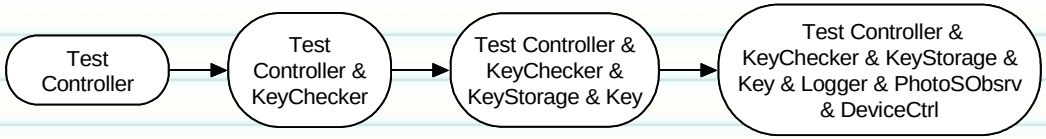
System hierarchy:



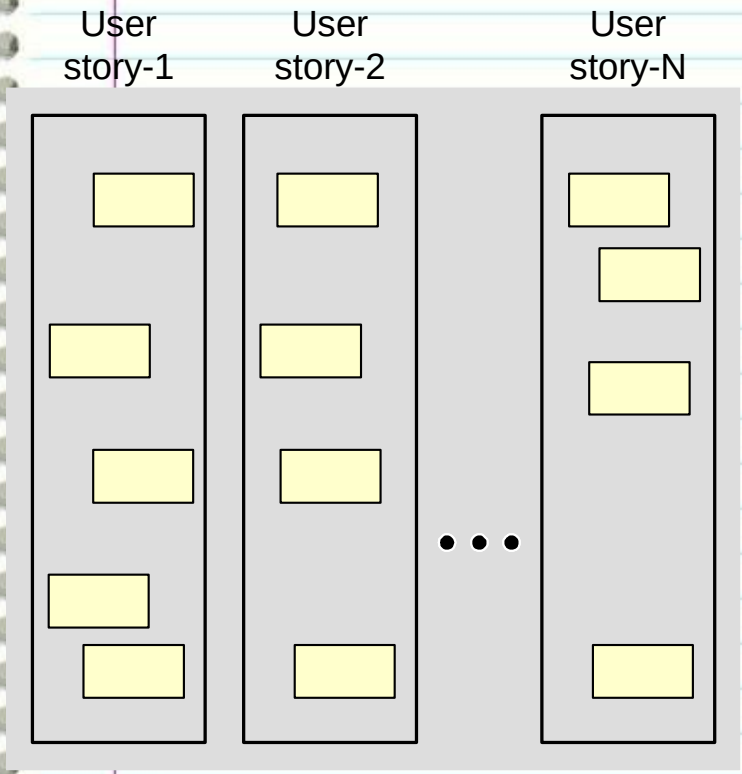
Bottom-up integration testing:



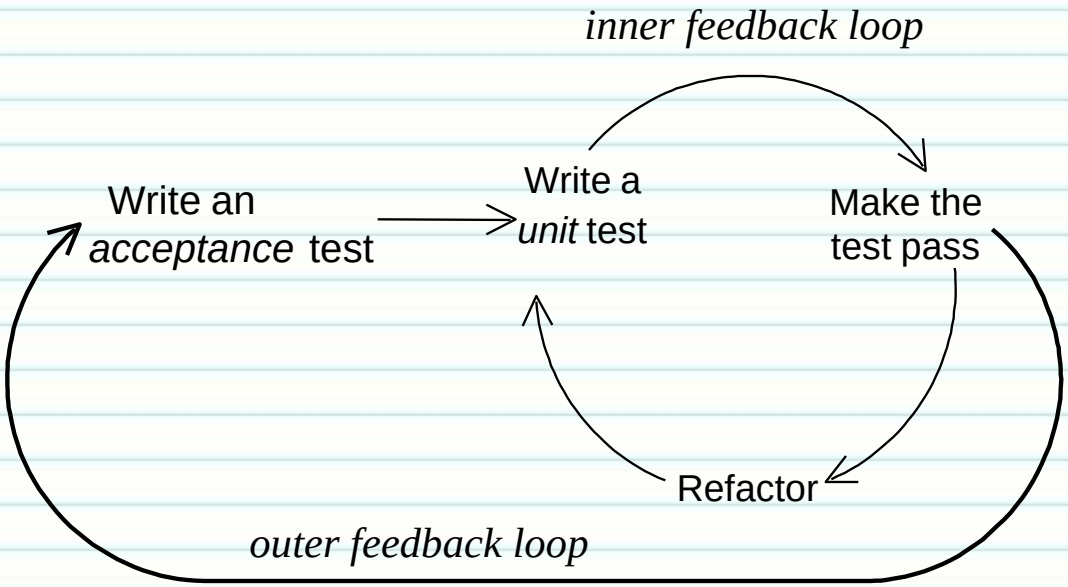
Top-down integration testing:



Vertical Integration Testing



Whole system



Developing user stories:

Each story is developed in a cycle from acceptance tests across integration level tests down to unit tests

System Testing - The Entire Application

System Testing - the Entire Application

- System Testing is where the application is built as a whole
  - It may be a stand-alone software executable or
  - It may be delivered as part of a hardware/software system
- For System Testing we test to the Software Requirements (Specification)
  - This is typically performed by an group/team independent from the development team
  - Problem reports are written and formally tracked - each problem is assessed for whether it needs to be fixed in the current release
- After System Testing some teams perform Acceptance Testing - this is typically NOT tested to the Software Requirements but at a higher level
- Prior to formal product release we may perform pre-release tests
  - Alpha testing - typically a pre-release readiness test performed internally
  - Beta testing - typically a pre-release readiness test perrformed by a limited set of external users

Other Types of Test

Risk Based Testing

Risk based testing is testing of the project based on risks.

- Uses risk to prioritize and emphasize the appropriate tests during test execution.
- Risk is the probability of occurrence of an undesirable outcome. This outcome is also associated with an impact.
- It starts early in the project, identifying risks to system quality and using those risks to guide testing planning, specification, preparation and execution.
- Given time limitations, it involves testing the functionality which has the highest impact and probability of failure.
- It involves test mitigation – testing to reduce the likelihood of defects, especially high-impact defects –
- It involves test contingency – testing to identify work-arounds to make the defects that do get past us less painful.
- It may involve measuring how well we are doing at finding and removing defects in critical areas.

Risk Based Testing (cont.)

- How risk based testing performed?
  1. Make a prioritized list of risks.
  2. Perform testing that explores each risk.
  3. As risks are mitigated and new ones emerge, adjust the test effort

Exploratory Testing

- Exploratory testing is a hands-on approach in which testers are involved
  - Emphasis is on test execution over planning
- The planning involves the creation of a test charter, a short declaration of the scope of a short (1 to 2 hour) time-boxed test effort, the objectives and possible approaches to be used.
- The test design and test execution activities are performed in parallel
  1. Typically without formally documenting the test conditions, test cases or test scripts - the tester may decide to us boundary value analysis but will informally do this without writing them down.
  2. Notes may be written during the exploratory-testing session, so that a report can be produced afterwards.
- Test logging is undertaken as test execution is performed, documenting the key aspects of what is tested, any defects found and any thoughts about possible further testing.
- It can complement other, more formal testing, helping to establish greater confidence in the software
  - e.g., as a check on the formal test process by helping to ensure that the most serious defects have been found.

Test Execution and Monitoring

Monitoring and Measuring Test Execution

- Each level of test has specific measures used to monitor and measure test executions and progress

- Metrics for monitoring test execution
  - Test cases reviewed
  - Test cases developed, debugged, and executing
  - Test cases captured in the developmental CM system

- Metrics for monitoring defects
  - Test cases passing/failing
  - Test cases blocked (awaiting a software fix)
  - Software fixes targeted for a specific release

- Test case effectiveness metrics
  - Measure the “defect revealing ability” of the test suite
  - Use the metric to improve the test design process

- Test-effort effectiveness metrics
  - Number of defects found by the customers that were not found by the tes engineers