TVVS – Software Testing Verification and Validation

Unit Testing

Ana Paiva

apaiva@fe.up.pt

//web.fe.up.pt/~apaiva

Homework

Build concrete test cases for

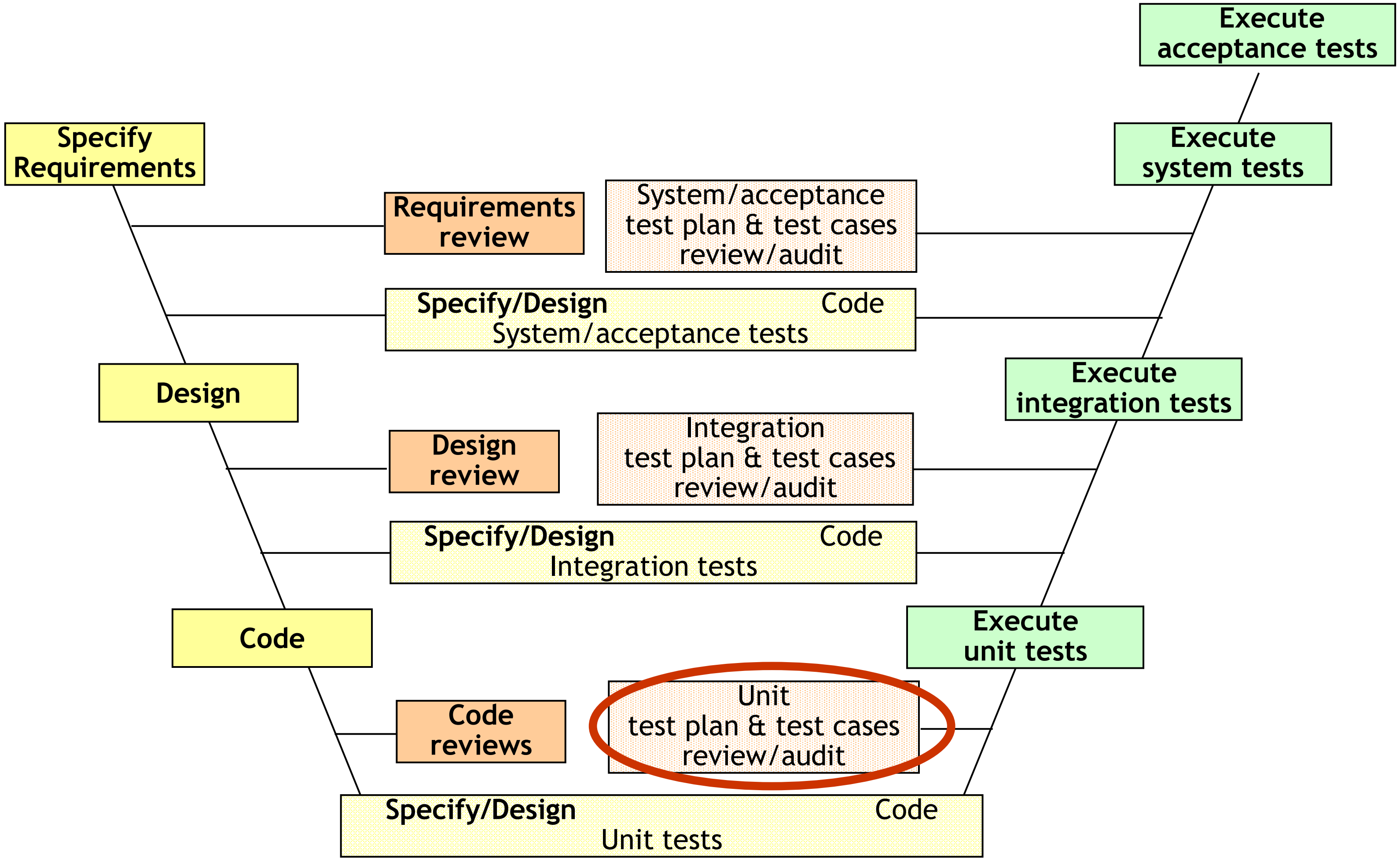
- Concrete Test Case 3 (Verify Password Reset Functionality)
  - **Precondition**: be on //www.url.pt
  - **Actions**:
    - Enter *login* ANA
    - Click reset_password;
    - Enter *email_address* apaiva@fe.up.pt;
    - Click on link sent by email to apaiva@fe.up.pt;
    - Enter *new_password* "NEWPass";
    - Enter *confirm_password* "NEWPass"
    - Click submit
  - **Result**: Verify if message "password reset" on the screen
  - **Post-condition**: Password of ANA is now "NEWPass"

Homework

Build concrete test case for

- Concrete Test Case 4 (Verify Account Lockout) "… when a user exceeds the maximum number of login attempts…"
  - **Precondition**: be on //www.url.pt
  - **Actions**:
    - Enter *login* "ANA"; Enter *password* "WrongPass"
    - Click Login
    - Enter *login* "ANA"; Enter *password* "WrongPass"
    - Click Login
    - Enter *login* "ANA"; Enter *password* "WrongPass"
    - Click Login
  - **Result**: Verify if message "ANA Account Lockout" on the screen
  - **Post-condition**: ANA Account *in state* Locked

Unit testing



Execute acceptance tests
Specify Requirements
Execute system tests
Requirements review
System/acceptance test plan & test cases review/audit
Specify/Design Code System/acceptance tests
Design
Execute integration tests
Design review
Integration test plan & test cases review/audit
Specify/Design Code Integration tests
Code
Execute unit tests
Code reviews
Unit test plan & test cases review/audit
Specify/Design Code Unit tests

The extended V-model of software development [I.Burnstein]

Unit testing – definition

- **Unit testing**: Testing of individual hardware or software units or groups of related units [IEEE 90].

- **Unit testing** is a development procedure where programmers create tests as they develop software. The tests are simple short tests that test functionality of a particular unit or module of their code, such as a class or function.

- **Unit testing** is a software development process in which the smallest testable parts of an application, called units, are individually and independently scrutinized for proper operation. This testing mode is a component of **Extreme Programming (XP)**, a pragmatic method of software development that takes a meticulous approach to building a product by means of continual testing and revision.

Unit testing

- Defining a ‘unit’ is challenging and highly dependent on the context. A unit can be just one method or can consist of multiple classes. Here is a definition for unit testing by Roy Osherove
- “A unit test is an automated piece of code that invokes a unit of work in the system. And a unit of work can span a single method, a whole class or multiple classes working together to achieve one single logical purpose that can be verified.”

The anatomy of a unit test

(Arrange, Act, Assert - AAA)

```
public class CalculatorTest {
    @Test
    public void test_sum_of_two_numbers() {
        // Arrange
        double first = 10;
        double second = 20;
        var calculator = new Calculator();

        // Act
        double result = calculator.sum(first, second);

        // Assert
        org.junit.Assert.assertEquals(30, result);
    }
}
```

Arrange: This is where you prepare and initialize all variables and objects that are needed by the system under test to work

The anatomy of a unit test
(Arrange, Act, Assert - AAA)

```
public class CalculatorTest {
    @Test
    public void test_sum_of_two_numbers() {
        // Arrange
        double first = 10;
        double second = 20;
        var calculator = new Calculator();

        // Act
        double result = calculator.sum(first, second);

        // Assert
        org.junit.Assert.assertEquals(30, result);
    }
}
```

Act: In this section you will actually invoke the method that you are testing

The anatomy of a unit test
(Arrange, Act, Assert - AAA)

```
public class CalculatorTest {
    @Test
    public void test_sum_of_two_numbers() {
        // Arrange
        double first = 10;
        double second = 20;
        var calculator = new Calculator();

        // Act
        double result = calculator.sum(first, second);

        // Assert
        org.junit.Assert.assertEquals(30, result);
    }
}
```

Arrange: This is where you prepare and initialize all variables and objects that are needed by the system under test to work

Act: In this section you will actually invoke the method that you are testing

Assert: This section is used to validate the return value received from the sum() method. If the returned value is what is expected, then the test method will pass. If the returned value is not what was expected, then the test method fails.

The anatomy of a unit test
(Arrange, Act, Assert - AAA)

```
public class CalculatorTest {
    @Test
    public void test_sum_of_two_numbers() {
        // Given
        double first = 10;
        double second = 20;
        var calculator = new Calculator();

        // When
        double result = calculator.sum(first, second);

        // Then
        org.junit.Assert.assertEquals(30, result);
    }
}
```

Unit testing

- The goal of unit testing is to exercise ONE unit at a time
- Unit tests vs. Integration tests
  - Unit tests do not touch real dependencies; Integration tests touch concrete dependencies
  - Unit tests are fast; Integration tests do not need to be
  - Unit tests do not touch databases, web services, etc.; Integration tests do
- What if a Unit depends on other things?
  - A network? A database? A servlet engine?
  - Other parts of the system?
  - We do not want to initialize lots of components just to get the right context for one test to run.

Separation of interface from implementation

- Because some classes may have references to other classes, testing a class can frequently spill over into testing another class.
  - Ex.: classes that depend on a database: in order to test the class, the tester often writes code that interacts with the database.
- This is a mistake
  - unit test should never go outside of its own class boundary.
- Abstract an interface around database connection and implement it with your own **mock objects**
  - the independent unit can be more thoroughly tested
  - this results in a higher quality unit that is also more maintainable.

Benefits of unit testing

- Facilitates change
  - Allows **refactor** code at a later date and make sure the module still works correctly
- Simplifies integration
  - Helps to eliminate uncertainty in the units themselves and can be used in a bottom-up testing style approach. By testing the parts of a program first and then testing the sum of its parts, **integration testing** becomes much easier
- Documentation
  - Provides a sort of **living documentation** of the system. Developers looking to learn what functionality is provided by a unit and how to use it can look at the unit tests to gain a basic understanding of the unit API

“Good” unit tests

- What properties a unit test should have?
  - It is automated and repeatable
  - It is easy to implement
  - Once it is written, it stays on for the future
  - Anyone can run it
  - It runs at the push of a button
  - It runs quickly
- Unit tests are code …
  - Maintanability
  - Readability
  - Correctness
  - Documentation
  - …

“Good” unit tests

- **Runs fast, runs fast, runs fast.** If the tests are slow, they will not be run often.
- **Separates or simulates environmental dependencies such as databases, file systems, networks, queues, and so on.** Tests that exercise these will not run fast, and a failure does not give meaningful feedback about what the problem actually is.
- **Is very limited in scope.** If the test fails, it is obvious where to look for the problem. Use few Assert calls so that the offending code is obvious. It is important to only test one thing in a single test.

[http://msdn.microsoft.com/en-us/library/aa730844.aspx#guidelinesfortdd_topic3]

“Good” unit tests (2)

- **Runs and passes in isolation.** If the tests require special environmental setup or fail unexpectedly, then they are not good unit tests. Change them for simplicity and reliability. Tests should run and pass on any machine. The "works on my box" excuse doesn't work.
- **Often uses stubs and mock objects.** If the code being tested typically calls out to a database or file system, these dependencies must be simulated, or mocked. These dependencies will ordinarily be abstracted away by using interfaces.
- **Clearly reveals its intention.** Another developer can look at the test and understand what is expected of the production code.

[http://msdn.microsoft.com/en-us/library/aa730844.aspx#guidelinesfortdd_topic3]

Six rules of unit testing

- Write the test first
- Never write a test that succeeds the first time
- Start with the null case, or something that doesn't work
- Don't be afraid of doing something trivial to make the test work
- Loose coupling and testability go hand in hand
- Use mock objects

[http://radio.weblogs.com/0100190/stories/2002/07/25/sixRulesOfUnitTesting.html]

Unit testing - best practices

- Keep unit tests small and fast
  - Ideally the entire test suite should be executed before every code check in.
- Unit tests should be fully automated and non-interactive
  - The test suite is noemally executed on a regular basis and must be fully automated to be useful. If the results require manual inspection the tests are nto proper unit tests.
- Make unit tests simple to run
  - Configure the development environment so that single tests and test suites can be run by single command or a one button click

Unit testing – best practices

- Keep testing at unit level
  - Unit testing is about testing units. Units should be tested in isolation. Avoid temptation to test an entire workflow using a unit testing framework, as such tests are slow and hard to maintain. Workflow testing may have its place, but it is not unit testing and it must be set up and executed independently
- Start of simple
  - One simple test is infinitely better than no tests at all. A simple test class will verify the presence and correctness of both the build environment, the unit testing environment, the execution environment and the coverage analysis tool, etc.

Unit testing – best practices

- Keep tests independent
  - To ensure testing robusteness and simplify maintenance, tests should never rely on other tests nor should they depend on the ordering in which tests are executed
- Name tests properly
  - Make sure each test method test one distinct feature of the unit being tested and name the test methods accordingly. The typical naming convention is test[what] such as testSaveAs(), testAddListener(), testDeleteProperty(), etc.

Unit testing – best practices

- Keep tests close to the unit being tested
  - If you are testing a class Foo, the test class should be called FooTest and keep in the same package (directory) as Foo.
- Think black-box
  - Act as a 3rd party class consumer, and test if the class fulfills its requirements. As try to tear it apart.
- Think white-box
  - After all, the test programmer also wrote the class being tested, and extra effort should be put into testing the must complex logic.

Unit testing – best practices

- Test the trivial cases too
  - It is sometimes recommended that all non-trivial cases should be tested and that trivial methods like simple setters and getters can be omitted. However, there are several reasons why trivial cases should be tested too:
    - Trivial is hard to define.It may mean different things to different people.
    - The trivial cases can contain errors too, often as a result of copy-paste operations
- Focus on execution coverage
  - Ensure that the code is actually executed on some input parameters
- Cover boundary cases
  - Make sure the parameter boundary cases are covered, e.g., test negatives, 0, positive, smallest, larger, etc

Unit testing - best practices

- Provide a random generator
  - When boundary are covered, you may use random generated parameters.
- Use explicit asserts
  - Always prefer assertEquals(a,b) to assertTrue(a==b) as the fomer gives more useful information
- Provide negative tests
  - Negative tests intentionally misuse the code and verify robusteness and appropriate error handling

Unit testing - best practices

- Design code with testing in mind
  - Writing and maintaining unit tests are costly, so reducing cyclomatic complexity in the code are ways to reduce this cost
- Know the cost of testing
  - Not writing unit tests is costly, but writing unit tests is costly too. There is a trade-off between the two, and in terms of execution coverage the typical industry standard is at about 80%
- Prioritize testing
  - If there is not enough resources to test all parts you should establish a priopitization

Unit testing - best practices

- Prepare test code for failures
  - If a test fails, the remaining tests will not be exected. Always prepare for test failure so that the failure of a single test does not bring down the entire test suite execution
- Write tests to reproduce bugs
  - When a bug is reported, write a test to reproduce the bug (i.e., failing test) and use this test as a success criteria when fixing the code
- Know the limitations
  - Unit tests can never prove the correcteness of code.
- ...
- ...

Limitations of unit testing

- Testing, in general, cannot be expected to catch every error in the program. Unit **tests can only show the presence of errors; it cannot show the absence of errors.**
- It only tests the functionality of the units themselves.
- It may not catch integration errors, performance problems, or other system-wide issues.
- Unit testing is more effective if it is used in conjunction with other software testing activities.

Limitations of unit testing

- Software testing is a combinatorial problem.
  - For example, every **boolean decision** statement requires at least **two tests**:
    - one with an outcome of "true" and one with an outcome of "false".
  - As a result, for **every line of code** written, programmers often need **3 to 5 lines of test code**. Therefore, it is unrealistic to test all possible input combinations for any non-trivial piece of software without an automated characterization test generation tool such as JUnit Factory used with Java code or many of the tools.

Limitations of unit testing

- To obtain the intended benefits from unit testing use of a **version control system** is essential
  - If a later version of the unit fails a particular test that it had previously passed, the version-control software can provide a list of the source code changes (if any) that have been applied to the unit since that time.
- It is also essential to **implement a sustainable process** for ensuring that test case failures are reviewed daily and addressed immediately
  - If such a process is not implemented and ingrained into the team's workflow, the application will evolve out of sync with the unit test suite — increasing false positives and reducing the effectiveness of the test suite.

Unit testing frameworks

- Help simplify the process of unit testing
  - Log test cases that fail
  - Automatically flag and report in a summary these failed test cases.
  - Depending upon the severity of a failure, the framework may halt subsequent testing.
- Examples
  - JUnit
  - JTest
  - NUnit
  - MbUnit
  - …

[http://en.wikipedia.org/wiki/List_of_unit_testing_frameworks]

Setup and teardown (1)

- You want to avoid duplicated code when several tests share the same initialization and clean-up code.
- … before and after each test method inside a test class
  - **JUnit**: Use the **setUp( )** and **tearDown( )** methods. Both of these methods are part of the **junit.framework.TestCase** class.
  - **NUnit**: build methods and annotate them with **SetUp** and **TearDown** tags

Setup and teardown (2)

- …. One-time set up and tear down
  - You want to run some setup code one time and then run several tests. You only want to run your clean-up code after all of the tests are finished, ex.: establish a database connection.
  - **JUnit**: Use the **junit.extensions.TestSetup** class to define **test suites**
    - Pass a TestSuite to the TestSetup constructor. This means that TestSetup's setUp( ) method is called once before the entire suite, and tearDown( ) is called once afterwards.

      TestSetup setup = new TestSetup(new TestSuite(TestPerson.class)) {…}
  - **NUnit**: methods annotated with **TestFixtureSetup** and **TestFixtureTearDown** attributes.

Mock objects

- Mock objects are simulated objects that mimic the behavior of real objects in controlled ways.
- Useful when a real object is impractical or impossible to incorporate into a unit test. If an object has any of the following characteristics, it may be useful to use a mock object in its place:
  - supplies non-deterministic results (e.g., the current time or the current temperature);
  - has states that are difficult to create or reproduce (e.g., a network error);
  - is slow (e.g., a complete database, which would have to be initialized before the test);
  - does not yet exist or may change behavior;
  - would have to include information and methods exclusively for testing purposes (and not for its actual task).

Mock objects (example)

- An alarm clock program which causes a bell to ring at a certain time might get the current time from the outside world. To test this, the **test must wait** until the alarm time to know whether it has rung the bell correctly. If a mock object is used in place of the real object, it can be programmed to provide the bell-ringing time (whether it is actually that time or not) so that the alarm clock program can be tested in isolation.

Steps to use mocks for testing

The code under test only refers to the object by its interface, so it have no idea whether it is using the real object or the mock object. Both use the same interface.

1. Use an interface to describe the object
2. Implement the interface for production code
3. Implement the interface in a mock object for testing

Example: real and mock object

// Interface describing the object

```
public interface Environmental {
    public long getTime();

    // other methods omitted...
}
```

// Implementation for production code

```
public class SystemEnvironment implements Environmental {
    public long getTime() {
        return System.getCurrentTimeMillis();
    }
    // other methods omitted...
}
```

Example: real and mock object

// Implementation of a mock object for testing

```
public class MockSystemEnvironment implements Environmental {
    private long currrentTime;
    public void setTime(long aTime) { currentTime = aTime; }
    public long getTime() {
        return currentTime;
    }
    private Boolean bellRung = false;
    public void ringBell(string s) { bellRung = true;}
    public void resetBell() { bellRung = false;}
    public Boolean wasBellRung() {return bellRung;}
}
```

Example: real and mock object

// Implementation of a timer

```
public Timer {
    private Environmental env;
    Timer(Environmental env) { this.env = env; …}

    public void reminder() {
        long t = env.getTime();
        if (laterThanSpm(t)) env.ringBell("ringing.wav");
    }
}
```

Example: real and mock object

// Testing code

```
public TestClassUnderTest {
     @Test
     public void testACaseOfMethodUnderTest() {
          MockSystemEnvironment env = new MockSystemEnvironment();
          Calendar cal = Calendar.getInstance();
          cal.set…. // to the 16:59:00, December 2012
          long t = cal.getTimeMillis();
          env.setTime(t);
          Timer timer = new Timer(env);

          timer.reminder();
          assertFalse(env.wasBellRung());  // too early to ring

          t += (5*60*100); // advance the timer 5 minutes
          env.setTime(t);
          env.resetBell(t);
          timer.reminder();
          assertTrue(env.wasBellRung());
     }
}
```

Mock Frameworks

- Facilitates the construction of Mock Objects
- Allow the construction based on
  - Interfaces
  - Classes
- Allow to define expectations
  - Number of method calls
  - Return values
  - Values of the parameters
  - Throw exceptions
- Integrate with JUnit (Java), NUnit (.Net), etc.
- Usually provide extensible APIs

Mock Frameworks

- Java
  - **jMock** -- http://jmock.org
  - **Mockito --** http://code.google.com/p/mockito
  - **rMock** – http://rmock.sourceforge.net
  - **MockCreator** – http://mockcreator.sourceforge.net
  - **MockLib** – http://mocklib.sourceforge.net
- C#
  - **MockLib** – http://sourceforge.net/projects/mocklib
  - **Rhino Mocks** – http://www.ayende.com
  - **Nmock** – http://nmock.org
- Ruby
  - **Mocha** – http://mochs.rubyforge.org
  - **Rspec** – http://rspec.rubyforge.org
- …