# UNIT TESTING

## LEARNING OBJECTIVES

- Produce testable code
- Design, structure, and automate tests
- Know how to apply equivalence partitioning and boundary value analysis in tests
- Know general features and usage of xUnit Framework API
- Implement high quality automated tests with xUnit Framework

## INTRODUCTION

### WHY MAKE TESTS?

Leads to better code and design
Generates code targeted at tests
Reduces manual testing
Ensures basic functionality is independently scrutinized for proper operation regularly
Creates control of code working as assumed throughout project process
Requires more consideration when refactoring and redesigning is done
Catches regression defects when they are introduced

### WHEN TO WRITE TESTS?

After the code is written
- Verify code is working as intended
- Tests might be influenced by implementation

While writing the code
- Co-design a unit and its tests together, in an iterative fashion
- Things gradually become more clear

Before any code is written
- Helps clarifying the needed functionality
- Becomes a design process
- Tests are not influenced by implementation knowledge

### WHAT CODE TO TEST?

| | |
|---|---|
| Getters / Setters? | Not so interesting |
| Logic? | More interesting |

## WHAT MAKES CODE TESTABLE?

Easily testable code guide lines…

- Follow the SOLID rules
- Follow the single responsibility principle
- Create simple constructors
- Use new with care
- Use inversion of control and dependency injection
- Reduce dependencies
- Avoid hidden dependencies
- Avoid logic in constructors
- Avoid static methods
- Avoid singletons
- Avoid too many parameters in methods
- Avoid too large methods
- Favor generic methods
- Favor composition over inheritance
- Favor polymorphism over conditionals

Untestable code should be refactored

## WHAT MAKES A TEST A UNIT TEST?

Smallest testable parts
Individual units of work
Separate components
Class methods

No external dependencies
Does not communicate with database, across networks or use file system

# DESIGN / STRUCTURE / AUTOMATE

## HOW TO DESIGN TESTS

1. Identify which conditions to test
2. Design test cases that cover conditions

Dynamic testing = Tests are executed on running code

- Black box techniques
  Implementation is not known
  High level / Testers / No programming knowledge
  External / Based on requirements documentation
  Equivalence partitioning / Boundary value analysis / Decision tables / State transition
- White box techniques
  Implementation is known
  Low level / Developers / Programming knowledge
  Internal / Based on design documentation

## EQUIVALENCE PARTITIONING

Good all round technique to use first

Aims at minimizing the number of test cases
Divide set of test conditions into groups that can be considered the same and handled equivalently
Input values are partitioned into equivalence classes if they result in the same program behavior and find the same errors
Only need to test one condition from each partition, since it is assumed that all conditions in a partition will be treated the same way and either work or not work
Identify partitions and write a test case for each partition
Both valid and invalid partitions need to be considered
Invalid inputs are separate equivalence classes
Equivalence partitions = Equivalence classes

Divide a set of test conditions into groups or sets that can be considered the same
Identify classes / Select class value
Test only one condition from each partition
No need to test many values that act the same way only one

Think about possible valid and invalid classes
Equivalence partitioning can be applied to both input and output

## BOUNDARY VALUE ANALYSIS

Equivalence partitioning and boundary value analysis are closely related
Based on testing at the boundaries between partitions
Off by one errors often show at the boundaries between equivalence classes
Boundary value is on the edge of an equivalence partition
Choose minimum and maximum values from an equivalence class together with first and last value respectively in adjacent equivalence classes

**Open boundaries**
Open boundaries are when one of the sides in a equivalence class is left open
Open boundaries are more difficult to test, but there are ways to approach them
Best solution to open boundaries is to find out what the boundary should be specified as
Examine specification or investigate other related areas of the system
If no boundaries can be found, then use intuition or experience-based approach

**2 or 3 value approach**
Two value approach is on boundary value and one side, where three value approach is on boundary value and two sides

Technically, because every boundary is in some partition, if you did only boundary value analysis you would also have tested every equivalence partition, but…

If the boundary values fail:
Is it only the boundary values which failed or did the partition fail?
If one boundary value fails but not the other:
Does the partition then fail?

**Middle value**
Also choose a ”normal” input in the equivalence class
Not only extreme values should be tested but also more normal values which are not boundary values

# STRUCTURING TESTS

## HOW TO STRUCTURE TESTS?

**Naming conventions**

- BDD
  - Premises and assumptions
  - What should be done in a given situation
  - Should do something
  - Prefix test methods with “Should”
  - Example: ”ShouldAlertUserIfAccountBalanceIsExceeded” / ”ShouldFailForNegativeAmount”

- Unit of work / State under test / Expected behavior
  - unit test may exercise more than a single method
    1. First part correspond to methods or classes
    2. Second part describe the action performed
       - What’s being done?
       - What’s passed in?
    3. Third part convey an expectation
       - What’s supposed to happen?
       - What result is expected
  - Example: “Atm_NegativeWithdrawal_FailsWithMessage” / “Divide_DenominatorIsZero_ExThrown”

Think about conditions and outcomes
What is tested? / What is performed? / What is expected?

**Organizing conditions**

Arrange-Act-Assert

1. Set up test objects and arrange data
2. Execute the test code
3. Verify the outcome and assert that result is as expected

The Arrange Act Assert style obstructs repeating each of the steps several times

There are other styles of structuring tests, such as…

- Operate-Check
- Given-When-Then
- Setup-Execute-Verify-Teardown

**Asserts**

One test case – Many asserts
- Fewer test cases are needed
- Good enough for some test cases

One test case – One test assert
- More test cases are needed
- If a test fails it is easier to find out why

One liners might make it harder to identify problems

Have a concentrated focus in each test

## AUTOMATING TESTS

### WHY AUTOMATE TESTS?

Manual testing is tiresome / Make sure new code works / Make sure new code doesn't break existing code

### WHY NOT USE SYSTEM OUT?

Needs human interpretation / Confusing with many old messages / Slows down code

### HOW TO AUTOMATE TESTS?

xUnit frameworks and build tools can be used to automate execution of tests

## GOOD TESTS

### WHAT MAKES A TEST A GOOD UNIT TEST?

- Must have a clear intention
- Must be easy to understand
- Must use descriptive test names
- Must be based on good test case design
- Must consider invalid conditions
- Must follow the style of arranging, acting and asserting
- Must only have a single assert in each test
- Must not be dependent on environment
- Must not be dependent on external factors
- Must not be dependent on other tests
- Must provide understandable feedback
- Must provide same inputs and expect same result each time
- Must be reliable and consistent
- Must be repeatable and automated
- Must be executed quickly

## MAVEN

Build automation tool used primarily for Java projects
Standard way to build project / Clear definition of what project consists of / Easy way to publish project information / Way to share JARs across several projects

Default lifecycle comprises of the following phases

- validate - validate the project is correct and all necessary information is available
- compile - compile the source code of the project
- test - test the compiled source code using unit testing framework and code not being packaged or deployed
- package - take the compiled code and package it in its distributable format, such as a JAR.
- verify - run any checks on results of integration tests to ensure quality criteria are met
- install - install the package into the local repository, for use as a dependency in other projects locally
- deploy - done in the build environment, copies the final package to the remote repository for sharing with other developers and projects.

Dependencies and plugins can be added to pom.xml file
Helps building project infrastructure in nice way
Unit tests are usually included within the build process, which means they are run by a build tool like Maven.
Used to handle project dependencies via the pom.xml and execute commands in console

## XUNIT (JUNIT / NUNIT / CPPUNIT / PHPUNIT / JSUNIT)

Group of testing frameworks
Used for test automation

Architecture components

- Test cases
- Test suites
- Test fixture
- Test runner
- Assertions

## JUNIT

A testing framework which uses annotations to identify methods that specify a test
Executes methods with test annotation
A JUnit TEST, called a TEST CLASS, is a method contained in a class which is only used for testing.
To write a JUnit test you annotate a method with the `@org.junit.Test` annotation.
Test methods execute code during test and ASSERT methods, called ASSERTS or ASSERT STATEMENTS, provided by JUnit or another assert framework, checking an expected result with the actual result.
Meaningful messages are provided in assert statements, so that it is easy to identify and fix problems.

JUnit is a regression testing framework
Used by developers to implement unit tests
"Framework" because subclasses are created from abstract classes
"Regression" since tests can be kept and run frequently

**Project structure**
Junit best practice: Put test classes in the same package as the class they test but in a parallel directory structure.

**Usage**
Create test classes of JUnit's with test methods
Run test classes directly from main() or with test running utility, such as JUnit TestRunners
JUnit assumes all test methods can be executed in an arbitrary order
Most IDE's has built in functionality in GUI for executing JUnit tests

**Create / Update tests**
Options: IntegrationTest / Levels / Generate / Comments

**Naming**
Mandatory prefixes and postfixes for tests to be executed during build (Test* / *Test / *TestCase)
Can be used to disable tests or might be the cause why tests are not run

## JUNIT4

### MAVEN – JUNIT4 DEPENDENCY

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.12</version>
</dependency>
```

## JUNIT5

JUnit 5 = JUnit Platform + JUnit Jupiter + JUnit Vintage

Architecture
4 separate modules:

**JUnit Platform**
: For launching testing frameworks

**JUnit Jupiter**
: Programming and extension model for writing tests and extensions

**JUnit Runner**
: Describe, collect, run and analyze tests

**JUnit Vintage**
: Support for JUnit 3 and 4 tests

## MAVEN – JUNIT5 DEPENDENCY

```
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter-engine</artifactId>
  <version>5.3.2</version>
</dependency>
```

## MAVEN – JUNIT4 + JUNIT5 DEPENDENCIES

```
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter-engine</artifactId>
  <version>5.3.2</version>
</dependency>
<dependency>
  <groupId>org.junit.platform</groupId>
  <artifactId>junit-platform-runner</artifactId>
  <version>1.3.2</version>
</dependency>
<dependency>
  <groupId>org.junit.vintage</groupId>
  <artifactId>junit-vintage-engine</artifactId>
  <version>5.3.2</version>
</dependency>
```

# JUNIT FEATURES

## ANNOTATIONS

| | |
|---|---|
| @Test | (Test methods) |
| @BeforeAll / @AfterAll | (Methods executed before and after all test methods) |
| @BeforeEach / @AfterEach | (Methods executed before and after each test method) |
| @Disabled | (Classes / Methods) |
| @DisplayName | (Overwrite test methods default name) |

## ASSERTIONS

fail(message)
assertEquals(expected, actual, message)
assertNotEquals(expected, actual, message)
assertTrue(boolean, message)
assertFalse(boolean, message)

assertNull(object, message)
assertNotNull(object, message)
assertSame(expected, actual, message)
assertNotSame(expected, actual, message)
assertArrayEquals(expectedArray, resultArray, message)
assertAll()
assertThrows()

**Fixtures** Fixed state of a set of objects used as a baseline for running tests
- Do not use constructors and destructors, use test fixture methods instead
- Initializers / Clean up methods
- Advantage: Eliminates need for repeating test code
- Disadvantage: Separates test code into multiple different methods
- Split test classes if only some of the test methods need fixtures or need different fixtures or possibly refactor code if it is poorly designed to begin with

**Tests** Defines test cases to be executed
- Results in either passed or failed
- Tests can be however also be skipped or time out

**Suites** Several test classes can be combined into a test suite
- Running a test suite executes all test classes in that suite in the specified order
- A test suite can also contain other test suites
- A test suite is simply a logical grouping of test classes that can be run together as a group

**Assertions** Communicates success or failure and expresses outcome of test
- Any failing assertions will result in a failed run of the tests by the testing framework
- Use the assertions that best communicate the intent
- Be careful when setting up actual and expected data
- Remember order of parameters actual and expected

**Runners** Used for executing test cases
- JUnit provides a set of different test runners to execute tests, including runners for backward compatibility
- Normally JUnit will detect the right test runner to use without asking, based on the test cases
- Test runner is given a list of test classes
- For each test class:
  - Create an instance of the test class
  - Run constructor, setups, checks and teardowns
  - If checks fails, assertions are thrown and test method fails
  - Produce test report

**Latest features in JUnit5…**
Nesting / Tagging / Assumptions / Extensions / Conditions / Lambdas / Parameters