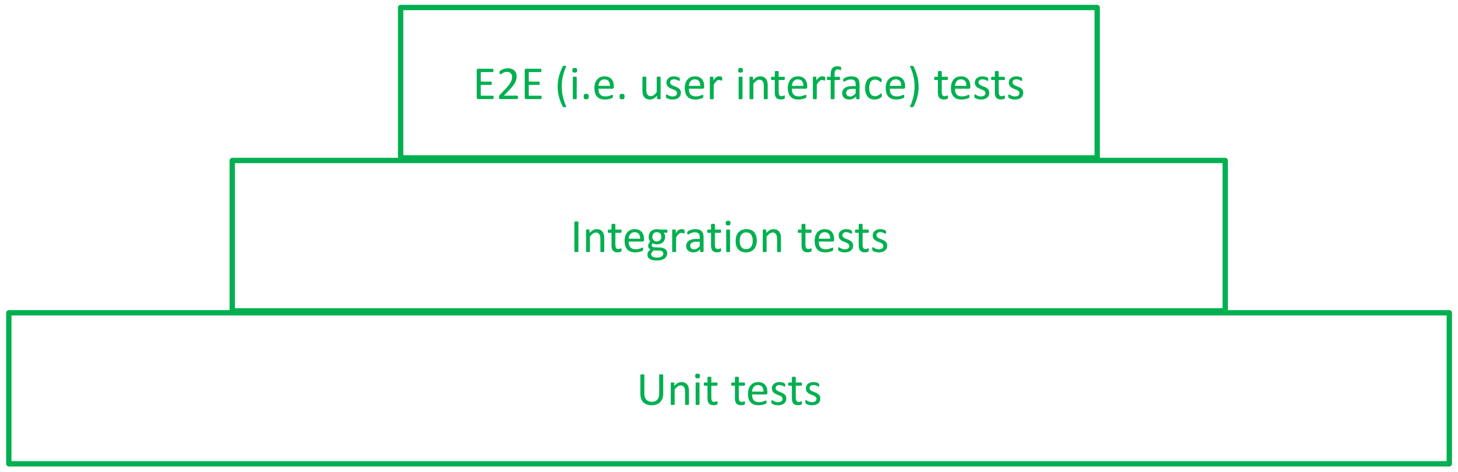
Unit testing

Agenda

- Unit testing overview
- JUnit framework
- TestNG framework
- Mocking frameworks

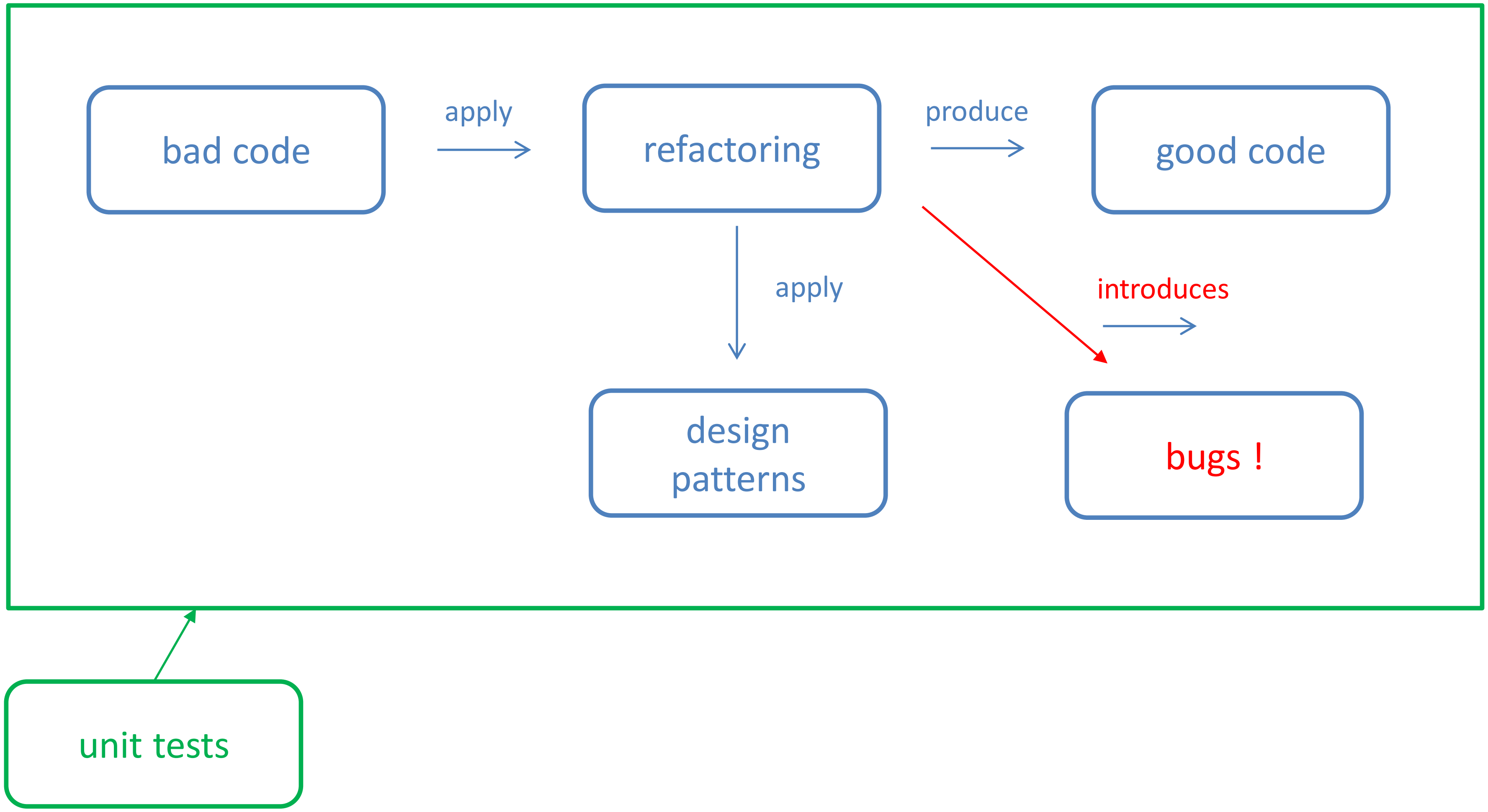
Software testing

The software testing pyramid



Unit testing overview

The big picture



Unit testing

- Unit testing refers to the practice of testing certain “units” of the source code

- It has a twofold purpose:

  - To provide a mechanism to test separate units (methods) of the code in isolation

  - To prevent regression bugs caused by any modifications/refactoring of the source code thus increasing product quality

Unit tests

- Unit tests are typically implemented by developers and are inseparable part of software development
- Unit tests aim to cover non-trivial functionality with confidence that it works for both base and corner cases
- In many cases time and budgeting constraints prevent the writing of unit tests

Unit of testing

- In Java the basic unit of testing are **public** methods which provide the public API of a class

- Protected and package-level methods might also be covered with tests but in many cases that is an extra activity

- Private methods should typically not be covered with unit tests in most cases

Trivial methods (like getters and setters) should not be covered with unit tests

Unit testing frameworks

- Different unit testing frameworks throughout the variety of programming languages

- In Java two of the most popular testing frameworks used in practice are JUnit and TestNG

- Testing frameworks are further supplemented by mocking frameworks that provide extra utilities for writing unit tests

In earlier days when unit testing framework were not that popular tests tend to be written as simple test classes separate from the main source code

Unit testing frameworks

- Testing frameworks facilitate further the process of unit testing by providing additional features such as:
  - parametrized tests that eliminate duplication of similar tests
  - mechanisms creating test lifecycle with pre/post-process phases
  - Logical grouping of tests (i.e. per package, class or custom group)
  - enhanced test execution strategies (i.e. running tests in parallel)
  - mechanisms to indicate success or failure of a test (asserts)

Unit testing frameworks

- IDEs like Eclipse and IntelliJ provide good integration for running JUnit and TestNG tests from the IDE
- Build systems like Maven or Gradle also provide mechanisms to execute unit tests
- In a continuous integration environment like Jenkins unit tests are typically executed once the source code is compiled
- If a test fails the entire build fails (unless the test is skipped)

Test-driven development

- Some disciplines such as test-driven development (TDD) promote the writing of unit tests before source code is written
- The main rationale here is that developers focus on short development cycle turning business requirements directly into unit tests
- Once unit tests pass development is considered done

Rules of thumb

- General guidelines in writing unit tests include:
  - tests should be self-contained
  - tests should not depend on each other or on system state affected by other tests
  - slowly running tests should generally be executed separately from the main application build cycle in order to avoid slowing down drastically build time of the application

JUnit framework

JUnit framework

- JUnit is by far the most popular unit testing framework for Java applications

- Apart from the main test execution framework JUnit provides an API for writing more specialized test execution frameworks

- Latest version of the framework is JUnit 5 that requires at least JDK 8

JUnit tests

- JUnit tests are organized in test class

- A test classes contains one or more test methods annotated with @Test

- Different naming conventions can be adopted for test methods (i.e. starting with **should** or in a **given-when-then** format)

Test classes are typically included in a separate source folder and not part of the application distribution

In JUnit 5 the @Test annotation comes from the org.junit.jupiter.api.Test package

JUnit tests

- Test classes can also provide lifecycle methods during unit test execution such as:
  - methods that are executed before the test methods from the class are executed (annotated with @BeforeAll)
  - methods that are executed after the test methods from the class are executed (annotated with @AfterAll)
  - methods that are executed before any test method from the class is executed (annotated with @BeforeEach)
  - methods that are executed after any test method from the class is executed (annotated with @AfterEach)

In JUnit 4 these annotations are @BeforeClass/@AfterClass/@Before/@After

JUnit tests

```
public class RingBufferTest {
        private RingBuffer buffer;

        @BeforeEach
        public void setUpBuffer() {
                buffer = new RingBuffer(10);
        }

        @Test
        public void shouldContainElement() {
                buffer.add(10);
                assertFalse(buffer.isEmpty());
                assertEquals(buffer.get(0), 10);
        }

        @AfterEach
        public void tearDownBuffer() {
                buffer = null;
        }
}
```

Executing JUnit tests

- JUnit tests can be executed in a variety of ways:
  - from the IDE using the JUnit integration for the target IDE
  - from the build system being used such as Ant, Maven or Grade
  - from the command line using the **junit-platform-console-standalone** utility for JUnit 5 tests

JUnit and Maven

- Maven dependencies can have a special **test** scope that indicates they are used only for unit tests and not included in the final distribution

- Maven uses a plug-in called Surefire to execute unit tests during a Maven build

- Surefire recognizes test classes under the **src/test/java** directory or classes starting or ending with **Test** or ending with **Tests** or **TestCase**

Ignoring tests

- Certain test methods or the test class might be ignored using the @Disabled annotation

- This might be required if the test is expected to fail or temporary fails and needs to be ignored

```
@Disabled
@Test
public void shouldContainElement() {
        buffer.add(10);
        assertFalse(buffer.isEmpty());
        assertEquals(buffer.get(0), 10);
}
```

In JUnit 4 this is the @Ignore annotation

Ignoring tests

- Additional criteria for running test cases can be specified using the ExecutionCondition API

- There are build in annotations used for conditional execution

```
@EnabledOnOs(LINUX)
```

```
@DisabledOnOs(WINDOWS)
```

```
@EnabledOnJre(JAVA_8)
```

```
@DisabledOnJre(JAVA_9)
```

```
@EnabledIfSystemProperty(named = “nginx.enabled",
       matches = “true")
```

```
@EnabledIfEnvironmentVariable(named = “TEST_SERVER",
       matches = “true")
```

Test timout

- A certain timeout can be specified for a unit test
- This might be required if the unit-under-test is expected to completed under certain time constraints

```
@Timeout(value = 5, unit = TimeUnit.SECONDS)
@Test
public void shouldContainElement() {
       buffer.add(10);
       assertFalse(buffer.isEmpty());
       assertEquals(buffer.get(0), 10);
}
```

Asserting exceptions

- To assert that an exception is thrown a special **assertThrows** assertion can be used in JUnit 5

```
@Test
void shouldThrowException() {
      Exception exception = assertThrows(Exception.class,
             () -> {throw
               new Exception("some exception ...");});
      assertEquals("some exception ...",
             exception.getMessage());
}
```

In JUnit 4 the exception was specified in an exception field of the @Test annotation:
@Test(expectedExceptions = SomeException.class)

Assert methods

- The following outlines some of the assert methods that are provided by the JUnit 5 framework as static methods of the **org.junit.jupiter.api.Assertions** class:

assertEquals/assertNotEquals	Asserts if two objects/primitives are/are not equivalent
assertTrue/assertFalse	Asserts if a condition is true/false
assertThrows	Asserts that a block throws an exception
assertTimeout	Asserts a block finishes execution within a timeout
assertNull/assertNotNull	Asserts if an object is/is not null
fail	Fails the unit test

Third party libraries such as AssertJ, Hamcrest and Truth provide additional assertions

Display name

- Test classes may have a **@DisplayName** annotation that provides display name for the unit tests and test classes in reports

```
@DisplayName("Ring buffer data structure tests")
public class RingBufferTest {
       private RingBuffer buffer;

       @Test
       @DisplayName("Ring buffer add operation test")
       public void shouldContainElement() {
              …
       }
}
```

Grouping tests

- Test classes and methods can be grouped for discovery and execution by tag using a @Tag annotation

```
@Tag(“DATA_STRUCTURES")
public class RingBufferTest {
          …
}
```

- Test cases can be grouped into test suites using the @SelectPackages and @SelectClasses annotations on a target Java class (that represents the test suite)

Repeated tests

- A test can be repeated multiple times with the @RepeatedTest annotation

```
@RepeatedTest(10)
@Test
public void shouldContainElement() {
       ...
}
```

Parameterized tests

- Parameterized tests provide the possibility to run a test multiple times with different arguments
- The @ParameterizedTest annotation is used instead of the @Test annotation
- There are a number of ways to specify parameters for a parametrized test
- The simplest mechanism is to use the @ValueSource annotation

Parameterized tests

```
@ParameterizedTest
@ValueSource(ints = {10, 20, 30})
public void shouldContainElement(int value) {
       buffer.add(value);
       assertFalse(buffer.isEmpty());
       assertEquals(buffer.get(0), value);
}
```

Parameterized tests

- Other types of value sources include:
  - @NullSource
  - @EmptySource
  - @EnumSource
  - @MethodSource
  - @CsvSource
  - @CsvFileSource
  - @ArgumentsSource

Dynamic tests

- Dynamic tests are test generated by a test factory method annotated with @TestFactory
- The return type of an @TestFactory method must be a DynamicTest instance or a collection/stream/iterable/iterator/array of DynamicTest instances
- A different return type results in a JUnitException

Dynamic tests

```
@TestFactory
List<DynamicTest> dynamicTestsForRingBuffer() {
      return Arrays.asList(
            dynamicTest("1st ring buffer test", () -> {
                  buffer.add(10);
                  assertFalse(buffer.isEmpty());
                 }),
            dynamicTest("2nd ring buffer test", () -> {
                  buffer.add(10);
                  assertEquals(buffer.get(0), 10);
                 })
            );
}
```

TestNG framework

TestNG framework

- TestNG is another popular unit testing framework inspired by JUnit and NUnit (unit testing framework for the .NET framework)

- TestNG provides additional features on top of the ones provided by JUnit such as:
  - dependent tests (thus creating dependencies between tests)
  - more capabilities for integration and GUI testing built in the framework

In JUnit 4 and earlier there were more features of TestNG lacking in JUnit but JUnit 5 filled the gap

TestNG framework

```
public class RingBufferTestNGTest {
	private RingBuffer buffer;

	@BeforeTest
	public void setUpBuffer() {
		buffer = new RingBuffer(10);
	}

	@Test
	public void shouldAddValue() {
		buffer.add(10);
		assertFalse(buffer.isEmpty());
		assertEquals(buffer.get(0), 10);
	}

	@AfterTest
	public void tearDownBuffer() {
		buffer = null;
	}
}
```

Grouping tests

- Tests can be grouped by specifying one or more groups as part of the @Test annotation

```
@Test(groups = {"data_structured"})
public void shouldAddValue() {
       buffer.add(10);
       assertFalse(buffer.isEmpty());
       assertEquals(buffer.get(0), 10);
}
```

Grouping tests

- Tests to be executed can also be grouped in a XML file

```
<test name="SampleTestSuite">
  <groups>
    <run>
      <exclude name="algorithms"  />
      <include name="data_structures"  />
    </run>
  </groups>

  <classes>
    <class name="RingBufferTestNGTest">
      <methods>
        <include name="shouldAddValue" />
      </methods>
    </class>
  </classes>
</test>
```

Parameterized tests

- Tests can be parametrized using the **dataProvider** element of the @Test annotation and creating a data provider method annotated with @DataProvider

```
@DataProvider(name = "valueProvider")
public Integer[] createValues() {
       return new Integer[] {10, 20, 30};
}

@Test(dataProvider = "valueProvider")
public void shouldAddValue(Integer value) {
       buffer.add(value);
       assertFalse(buffer.isEmpty());
       assertEquals(buffer.get(0), value);
}
```

Test dependencies

- A test can depend on other tests specified with the **dependsOnMethods** element of the @Test annotation

```
@Test
public void shouldNotBeEmpty() {
       buffer.add(10);
       assertFalse(buffer.isEmpty());
}

@Test(dependsOnMethods = { "shouldNotBeEmpty" } )
public void shouldAddValue() {
       buffer.add(10);
       assertEquals(buffer.get(0), 10);
}
```

TestNG and JUnit

- TestNG framework interoperates with JUnit by allowing for the execution of JUnit tests through TestNG

```
<test name="JUnitTestSuite" junit="true">
  <groups>
    <run>
      <exclude name="algorithms"  />
      <include name="data_structures"  />
    </run>
  </groups>
```

Mocking frameworks

The need of mocking

- In many cases the unit-under-tests depends on third-party systems (such as a relational database) or services that are not available during the test execution

- In order to avoid this problem we need to replace the third-party system or service with a fake object during the test execution

Types of mock objects

- To replace a certain object during the execution of a test the following types of mock objects can be created:
  - **dummy objects**: passed around but never really used
  - **fake objects**: have a working logic but are not suitable for production (like an in-memory database)

Types of mock objects

- To replace a certain object during the execution of a test the following types of mock objects can be created:
  - **stubs**: provide results to some operations being called (only those required by the test typically)
  - **spies**: a variant of stubs whereby some state is also preserved in addition by the spy object
  - **mocks**: an object on which certain expectations are set when an operations is called

Mocking frameworks

- To simplify the creation of mock objects there are number of frameworks that can be used on top of test frameworks such as JUnit and TestNG

- In Java the most popular mocking frameworks include:

  - Mockito
  - PowerMock
  - EasyMock

EasyMock

- EasyMock library provides a simple mechanism to create mock objects
- Mock objects can be created:
  - directly using the **org.easymock.EasyMock#mock()** method
  - using the @Mock annotation
- EasyMock provides a JUnit 5 extension with the EasyMockExtension class for running unit tests with EasyMock

```
@ExtendWith(EasyMockExtension.class)
```

EasyMock

```
@ExtendWith(EasyMockExtension.class)
public class RingBufferMockitoTest {

      @Mock
      private RingBuffer buffer;

      @BeforeEach
      public void setUpBuffer() {

      EasyMock.expect(buffer.isFull()).andReturn(true);
      }

      @Test
      public void shouldBeFull() {
            replay(buffer);
            assertTrue(buffer.isFull());
      }
}
```

Mockito

- Mockito library expands the capabilities of the EasyMock library

- Mockito provide several ways to create mock objects:
  - directly using the **org.mockito.Mockito#mock()** method
  - using the @Mock annotation

- Mockito provides a JUnit 5 extension with the MockitoExtension class for running unit tests with Mockito

```
@ExtendWith(MockitoExtension.class)
```

Mockito

- Once an object is created stubs and spies can be created on the mock object

- At a basic form the **when** and **then** methods are used to define he behavior of the mock object when a certain operation is called

```
Mockito.when(mock.method()).thenReturn(result);
```

```
Mockito.doThrow(new Exception()).when(mock).method();
```

- A mock object can be verified for a certain operation

```
mock.method();
verify(mock).method();
```

Mockito

```
@ExtendWith(MockitoExtension.class)
public class RingBufferMockTest {

       @Mock
       private RingBuffer buffer;

       @BeforeEach
       public void setUpBuffer() {

       Mockito.when(buffer.isFull()).thenReturn(true);
       }

       @Test
       public void shouldBeFull() {
              assertTrue(buffer.isFull());
       }
}
```

PowerMock

- PowerMock provides enhanced mocking capabilities on top of EasyMock and Mockito thus eliminating some limitations of the frameworks such as:
  - accessing final classes and methods
  - mocking private and static methods of a class
  - partial mocking of methods
  - mocking object creation

PowerMock

```
PowerMockito.mockStatic(SomeClass.class);
Mockito.when(SomeClass.staticMethod(param)).
      thenReturn(value);
```

At present PowerMock (also called PowerMockito) does not provide support for JUnit 5 and if needed JUnit 4 unit tests should be written instead

Questions ?