2025 — 2026

1TI – 1ACS



Back-End Development

Maven, Test Driven Development

N. Jacobs, J. Pieck, E. Steegmans, B. Van Impe, S. Van Peborgh, V. Witters

CONTENTS

- Testing
- Unit Testing
- Test Driven Development
- Maven

Testing

Unit Testing

Test Driven Development

Maven

Testing

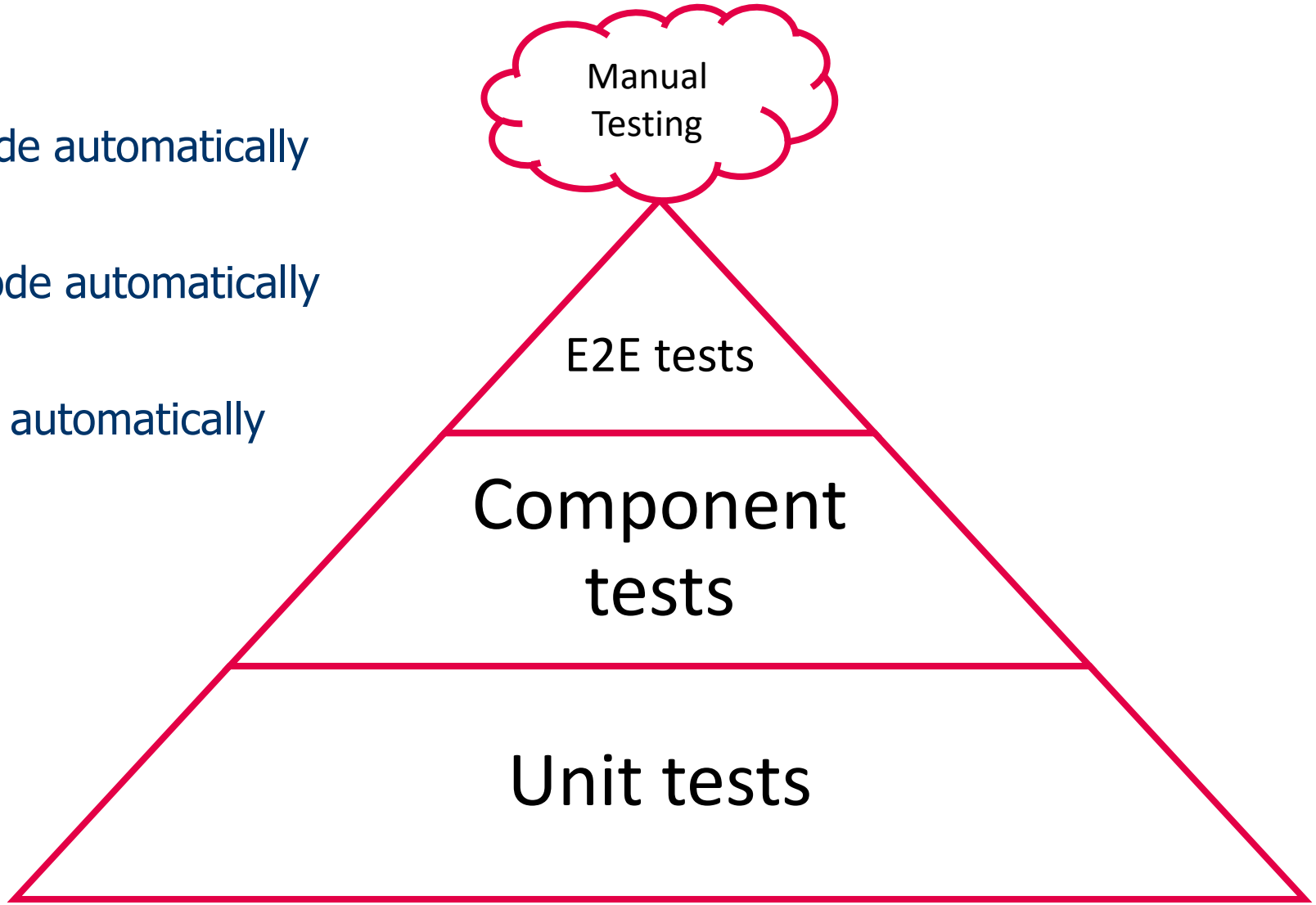
WHY WRITE TESTS?

- Detect bugs early
  - Cheaper to fix a bug on your computer than in an application that is being used
- You can run the tests again at any time
  - Remaining certain that everything works properly after making adjustments
- Makes you think about the design of your code
  - Code that cannot be tested is often badly designed code

- In short, tests provide automated Quality Control!

TEST TAXANOMY

- Unit tests
  - Tests a small piece of code automatically
- Component tests
  - Tests a large chunk of code automatically
- End2End tests
  - Tests the entire program automatically
- Manual tests



Testing

Unit Testing

Test Driven Development

Maven

Unit Testing

- What is a unit test?
- Getting started with tests

UNIT TESTING

- In Unit Testing, small pieces of code (called "units") are tested individually
- The code under test should be small so that the test executes quickly

UNIT TESTING

```java
public class Application {

    public static void main(String[] args) {
        // ...
    }

    public static int incrementByThree(int number) {
        return number + 3;
    }
}
```

```java
class ApplicationTest {

    @Test
    void incrementByThreeWorksCorrectly() {
        int result = Application.incrementByThree(3);
        Assertions.assertEquals(6, result);
    }
}
```

UNIT TESTING

```
class ApplicationTest {

    @Test
    void incrementByThreeWorksCorrectly() {
        int result = Application.incrementByThree(3);
        Assertions.assertEquals(6, result);
    }
}
```

- @Test indicates that this is a test
- A test is a method that can be executed
  - It is an entry-point to a piece of your code

UNIT TESTING

```
class ApplicationTest {

    @Test
    void incrementByThreeWorksCorrectly() {
        int result = Application.incrementByThree(3);
        Assertions.assertEquals(6, result);
    }
}
```

- The *Assertions* class contains methods to check the result

UNIT TESTING

```
public static int calcLength(String text) {
    return text.length();
}
```

```
@Test
public void calcLengthOnTextOf5Characters() {
    Assertions.assertEquals(5, Application.calcLength("12345"));
}
```

```
@Test
public void calcLengthOnInvalidInput() {
    Assertions.assertEquals(0, Application.calcLength(null));
}
```

- What happens with the second test?

UNIT TESTING

```
public static int calcLength(String text) {
    if (text == null) {
        throw new IllegalArgumentException(
            "Argument should not be null");
    }

    return text.length();
}
```

```
@Test
public void calcLengthOnTextOf5Characters() {
    Assertions.assertEquals(5, Application.calcLength("12345"));
}
```

```
@Test
public void calcLengthOnInvalidInput() {
    Exception ex = Assertions.assertThrows(
            IllegalArgumentException.class,
            () -> Application.calcLength(null));

    Assertions.assertEquals("Argument should not be null", ex.getMessage());
}
```

- It is important to test all cases for a method
  - Happy and unhappy paths
- Take note of the syntax when using “assertThrows”

UNIT TESTING — ASSERTIONS

	Description
assertEquals(expected, actual)	Is true if “expected” and “actual” contain the same value
assertNotEquals(expected, actual)	Is true if “expected” and “actual” contain a different value
assertTrue(boolean condition)	Is true if the “condition” is true
assertFalse(boolean condition)	Is true if the “condition” is false
assertThrows(Class<T> expectedType, Executable executable)	Is true if the code ran via the functional interface “executable” throws the exception specified by “expectedType”

- The class contains much more methods to check outcomes
  - The ones listed above cover most common cases

UNIT TESTING — IN PRACTICE

- Create a new file called "…Test" in which you write your tests
- Name of the test method should be human readable and indicative of what is being tested
  - If the test fails, you should know the error by checking the name of the failing test
  - Use the "given..._when..._then..." format for naming your tests
    - For example, "givenAString_whenCalcLengthOnNullArgument_thenAnExceptionIsThrown"

- Writing unit tests requires JUnit, an external library
  - We will manage external libraries with Maven

UNIT TESTING — IN PRACTICE

- Use the same given-when-then structure every time you write a test

```
@Test
public void givenACalculator_whenSumCalledWithOneAndThree_ThenFourIsReturned() {
    //given
    Calculator calculator = new Calculator();

    //when
    int sum = calculator.sum(1,3)

    //then
    assertEquals(4, sum);

}
```

Set up everything you need for the test

Perform the action you are testing

Check whether the result is what you were expecting

Testing

Unit Testing

Test Driven Development

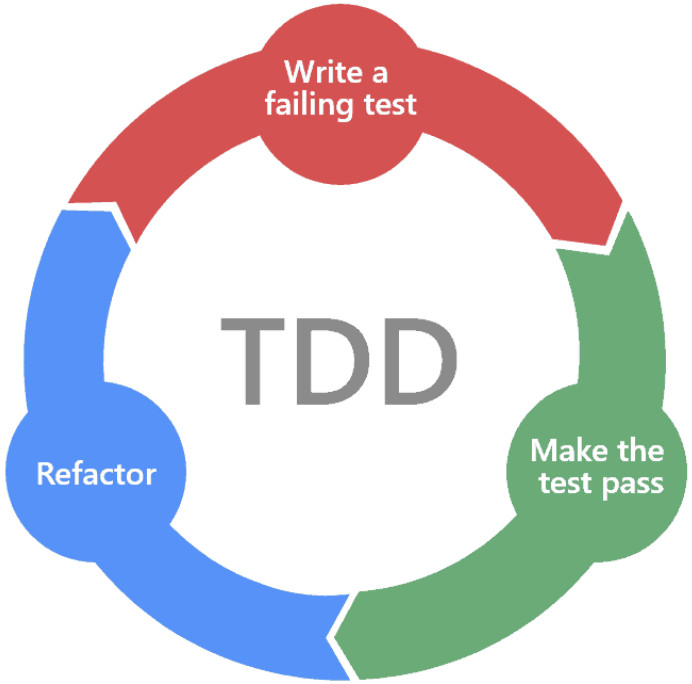
Maven

Test Driven Development

TEST DRIVEN DEVELOPMENT

- **T**est **D**riven **D**evelopment (TDD) is a software development process that relies on software requirements being converted to test cases before software is fully developed

- In practice, TDD means that you write new functionality in three steps:
  1. Write a failing test for the new functionality
  2. Make the test pass by implementing the functionality
  3. Reorganise (*refactor*) your code if necessary



EXERCISES @CLASS

- Exercise 1: Test-Driven development
  - Download *exercise-tdd.zip* from Toledo
  - With this exercise, you will practice the implementation and refactoring of code when tests are provided.
  - In the provided folder, you find an existing project with a test class for User.
  - Uncomment each test one by one and refactor your code until the test runs green.
  - You can also find this assignment in the README.md file.



EXERCISES @HOME

- Exercise 2: StudentTest
  - Uncomment the code in Student.java.
  - Implement tests for the constructor (one happy case and unhappy case for the studies field: null and empty string).
  - Implement tests for all methods overridden from the User superclass.
- Exercise 3: ParentTest
  - Uncomment the code in Parent.java.
  - Add an extra field *maxNumberOfChildren* to the class Parent.
  - This field is set when a new Parent is created. The parent cannot have more children than this parameter.
  - Implement this functionality according to the principles of TDD: first write tests (for the *constructor* with the extra field and *addChild()*), implement code afterwards.



Testing

Unit Testing

Test Driven Development

Maven

Maven

- What is Maven?
- Project Structure
- Dependency Management
- Lifecycle Management

WHAT IS MAVEN

- (Apache) Maven is an open-source tool that helps with structuring, building, releasing and maintaining (large) Java programs

- Maven provides a standardized project structure
- Maven provides a way to manage *dependencies*
  - Dependencies are external libraries that are necessary for running your code
- Maven provides a standardized way to build, test and release your code
  - Your IDE uses maven behind the scenes when you perform these actions



PROJECT STRUCTURE

- Maven provides a standardized project structure:
  - root-project-folder
    - src
      - main
        - java
        - resources
      - test
        - java
        - resources
    - target
    - readme.md
    - **pom.xml**

- The pom file contains all maven configuration
  - pom stands for **P**roject **O**bject **M**odel
  - The pom file is an XML file

PROJECT STRUCTURE

- root-project-folder
  - src
    - main
      - **java** → contains all source code (.java) files organized in packages
      - **resources** → contains supporting files necessary for the source code
    - test
      - **java** → contains all test (*Test.java) files organized in packages
      - **resources** → contains supporting files necessary for the test code
  - **target** → contains the compiler output (.class files)
  - readme.md
  - pom.xml

PROJECT STRUCTURE — POM

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>be.ucll</groupId>
    <artifactId>backend</artifactId>
    <version>0.0.1-SNAPSHOT</version>

    <name>A backend project</name>
    <description>A backend project</description>

    <properties>
        <java.version>21</java.version>
    </properties>
</project>
```

- The pom file contains all configuration for the project
- A maven codebase is uniquely identified by its *coordinates*
  - *groupId:* what organization is developing this codebases
  - *artifactId:* identifier of the codebase
  - *version:* the version currently being worked on

PROJECT STRUCTURE — POM

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>be.ucll</groupId>
    <artifactId>backend</artifactId>
    <version>0.0.1-SNAPSHOT</version>

    <name>A backend project</name>
    <description>A backend project</description>

    <properties>
        <java.version>21</java.version>
    </properties>
</project>
```

- *name* & *description* are optional descriptive properties
- *properties* contain key-value pairs that
  - Are sometimes used by maven in pre-configured *lifecycles* and *plugins*
  - Can be referenced in other sections of the pom

DEPENDENCY MANAGEMENT

- Almost every codebase uses dependencies
  - Without dependencies, every piece of functionality needs to be written by hand using standard java constructs
  - Dependencies contain code bundled by other people, often grouped to perform a specific function
  - Recall that for testing, we will need a dependency that provides us with the building blocks to test our code

- Maven provides
  - A system to host dependencies (called *registries*)
  - A system to download dependencies from registries
    - The .m2 directory is the place where Maven downloads the dependency jar files

DEPENDENCY MANAGEMENT

- A maven project **can be** packaged and then deployed to a maven registry
  - You can use maven to manage your project without ever deploying it
  - The deployed code is identified by its coordinates
  - The deployed code is called *a maven artifact*

- A maven registry is a software product that is used to store maven artifacts and make them available for download
  - A maven registry can be self-hosted or public
  - A popular choice is Sonatype Nexus

- For more information about deploying projects to maven registries, refer to the course Software Engineering

DEPENDENCY MANAGEMENT

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <!-- ... -->

    <dependencies>
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter-engine</artifactId>
            <version>5.6.2</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
</project>
```

- The public "maven repository" is configured by default and is the source for all listed dependencies
- Additional registries can be configured in the pom file

DEPENDENCY MANAGEMENT

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <!-- ... -->

    <dependencies>
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter-engine</artifactId>
            <version>5.6.2</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
</project>
```

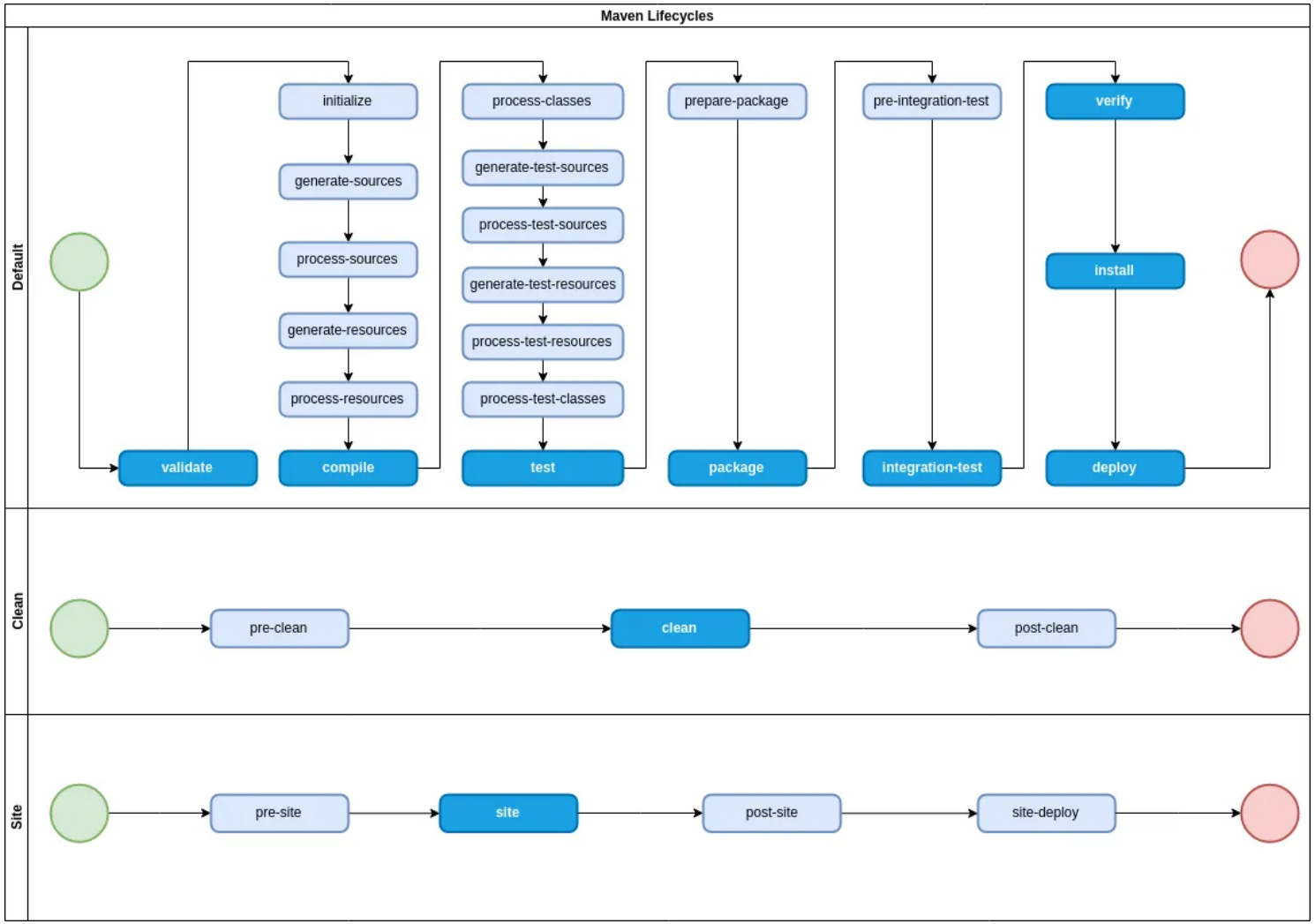
- You can list dependencies for your project in the *dependencies* section
- Specified dependencies are downloaded from maven registries

LIFECYCLE MANAGEMENT

- Maven provides *lifecycles* to manage your code
  - The default lifecycle to build and deploy your output
  - The clean lifecycle to remove all build output
  - (The site lifecycle to generate documentation)

- Every *lifecycle* consists of multiple *phases*
  - You can run the full lifecycle or one or more phases separately
  - You can execute a phase on your project with the command 'mvn <phase-name>'
  - Running a phase of a lifecycle first runs all previous phases from the lifecycle

LIFECYCLE MANAGEMENT



LIFECYCLE MANAGEMENT — IMPORTANT PHASES

Phase	Lifecycle	Description
clean	clean	Removes all .class files by deleting the “target” folder
compile	default	Compiles all code and stores .class files in the “target” folder
test	default	Runs the compile phase and then executes all tests
package	default	Runs the compile and test phases, then packages the code according to instructions so it can be deployed to a registry
deploy	default	Runs the compile, test and package phases, then deploys the packaged code to a registry according to configuration

- Only the clean, compile and test phases are relevant for this course
- We will execute these phases via our IDE most of the time

LIFECYLE MANAGEMENT

- Behind the screens, phases consist of a series of plugins that are executed in order
- The default phases are pre-configured with default plugins
  - For example, the compile phase runs a plugin that executes the java compiler
- All phases can be extended with existing or custom plugins
  - For example, the Spring framework will modify the package phase so that it produces a runnable artifact

LIFECYLE MANAGEMENT

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <!-- ... -->

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>
</project>
```

- Lifecycles, phases and plugins are configured in the *build* section

EXERCISES @CLASS

- Exercise 4: Create a new Maven project
  - Open IntelliJ: Click on the button “NewProject”
  - Select “New Project“
    - Enter a name for your project: “DemoUsers”
    - Select a destination folder.
    - Select “Java” as the language
    - Select “Maven” as the build system
    - For the GroupId, it is standard convention to enter "be.ucll".
    - ArtifactId: “DemoUsers”
  - Add the JUnit dependency to your pom.xml file.
  - In the src/main/java/be/ucll folder, create the domain and repository packages.
  - Ensure that you create these packages in the same directory as the class containing the main method.



EXERCISES @HOME

- Exercise 5
  - In the project you've created in exercise 4, create the Profile class.
    - A profile has the fields bio, location, and interests (all must be non-null and non-empty Strings).
    - Implement a constructor, getters, and setters including validation for all fields.Implement the equals(), hashCode(), and toString() methods.
    - Throw a RuntimeException when validation fails.
    - Follow the Test Driven Development approach.



