TQS Lab activities

v2020-03-20

Lab 1: Unit testing with JUnit 5 ..... **1**
Learning objectives 1
Preparatory readings 1
Key points 2
Lab activities 2
Explore 4
Lab 2: Mocking dependencies in unit testing **5**
Learning objectives 5
Lab activities 5
Explore 7
Lab 3: Functional testing with web automation **7**
Prepare 7
Key Points 7
Lab 7
Explore 8
Lab 4: Behavior-driven development (Cucumber in Java) **8**
Prepare **Error! Bookmark not defined.**
Key Points 8
Lab 9
Lab 5: Multi-layer application testing with Spring Boot **10**
Prepare 10
Key Points 10
Lab 10
Lab 6: Static Code analysis with Sonar Qube **13**
Key Points 13
Lab 13

Lab 1: Unit testing with JUnit 5

Learning objectives

- Identify relevant unit tests to verify the contract of a module.
- Write and execute unit tests using the JUnit framework.
- Link the unit tests results with further analysis tools (e.g.: code coverage)

Preparatory readings

- The test piramyd concept.
- Optional: TDD & Unit testing in IntelliJ tutorial.

Key points

- Unit testing is when you (as a programmer) write test code to verify units of (production) code. A unit represents some small subset of a much larger end-to-end-behavior. A true "unit" does not have dependencies on other (external) components.
- Unit tests help the developers to (i) understand the module contract (what to construct); (ii) document the intended use of a component; (iii) prevent regression errors; (iv) increase confidence on the code.
- When following a TDD approach, typically you go through a cycle of Red-Green-Refactor. You'll run a test, see it fail (go red), implement the simplest code to make the test pass (go green), and then refactor the code so your test stays green and your code is sufficiently clean.
- JUnit and TestNG are popular frameworks for unit testing in Java.

> **JUnit best practices: unit test one object at a time**
>
> A vital aspect of unit tests is that they're finely grained. A unit test independently examines each object you create, so that you can isolate problems as soon as they occur. If you put more than one object under test, you can't predict how the objects will interact when changes occur to one or the other. When an object interacts with other complex objects, you can surround the object under test with predictable test objects. Another form of software test, integration testing, examines how working objects interact with each other. See chapter 4 for more about other types

Lab activities

Be sure that your developer environment meets the following requirements:

— Java development environment (JDK), v8 or v11. Note that you should install it into a path without spaces or special characters (e.g.: avoid \Users\José Conceição\Java).

— Maven configured to run in the command line.

— Java capable IDE, such as IntelliJ IDEA.

Implement a stack data structure (TqsStack) with appropriate unit tests. Be sure to adopt a **write-the-tests-first** workflow:

a) Create a new **maven-based,** Java standard application.

   Note: use the IDE features. If you are not sure if the IDE can generate a maven-compatible structure, consider using this starter project.

b) Create the required classes definition (**just the "skeleton"**, not the methods body; you may need to add dummy return values). The code should compile, though the implementation is incomplete.

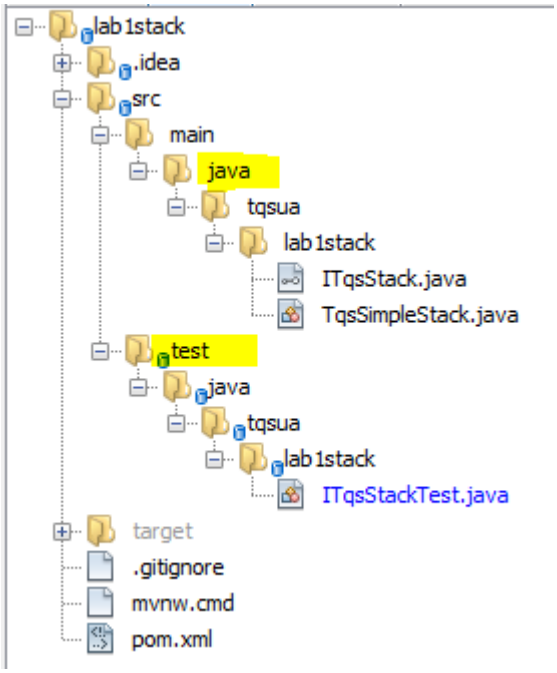
c) Write the unit tests that will verify the TqsStack contract.

   You may use the IDE features to generate the testing class; note that the IDE support will vary. Be sure to use JUnit 5.x.

   Important: for maven based projects, check the proper POM.xml dependencies for JUnit 5. If you get an error saying that the 1.5 version is no longer supported, specify the target compiler version in the POM.

   Your tests will verify several assertions that should evaluate to true for the test to pass. See some examples.

d) Run the tests and prove that TqsStack implementation is not valid yet (the tests should fail for now, the first step in Red-Green-Refactor).
e) Correct/add the missing implementation to the TqsStack;
f) Run the unit tests.
g) Iterate from steps d) to f) and confirm that all tests pass.



Stack operations:

- push(x): add an item on the top
- pop: remove the item at the top
- peek: return the item at the top (without removing it)
- size: return the number of items in the stack
- isEmpty: return whether the stack has no items

What to test[1]:

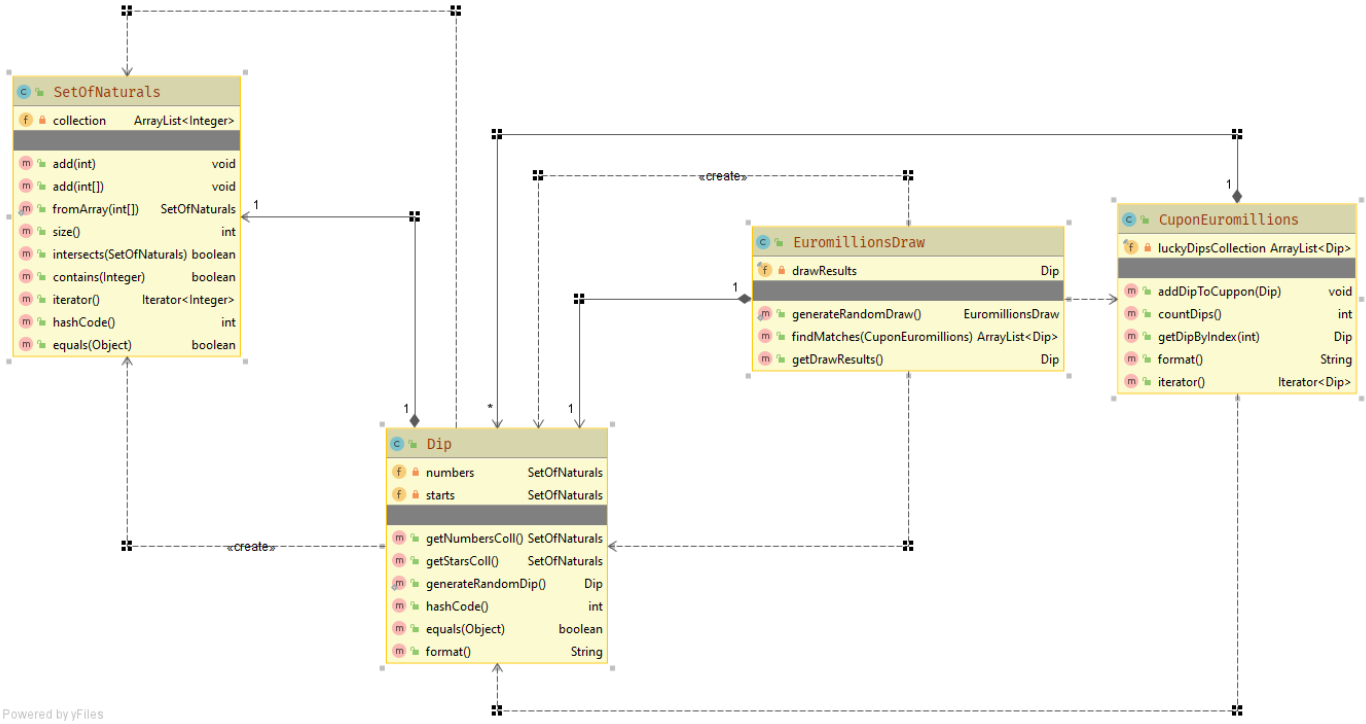
a) A stack is empty on construction.
b) A stack has size 0 on construction
c) After n pushes to an empty stack, n > 0, the stack is not empty and its size is n
d) If one pushes x then pops, the value popped is x.
e) If one pushes x then peeks, the value returned is x, but the size stays the same
f) If the size is n, then after n pops, the stack is empty and has a size 0
g) Popping from an empty stack does throw a NoSuchElementException [You should test for the Exception occurrence]
h) Peeking into an empty stack does throw a NoSuchElementException
i) For bounded stacks only, pushing onto a full stack does throw an IllegalStateException

2a/ Pull the "euromillions-play" project and correct the code (or the tests themselves, if needed) to have the existing unit tests passing.

For test:	**You should:**
testFormat	Correct the implementation of Dip#format so the tests pass.
testConstructorFromBadArrays	Implement new test logic to confirm that an exception will be raised if the arrays have invalid numbers (wrong count of numbers of starts)

Note: you may suspend temporary a test with the @Disabled tag (useful while debugging the tests themselves).

[1] Adapted from http://cs.lmu.edu/~ray/notes/stacks/



2b/ The class SetOfNaturals represents a set (no duplicates should be allowed) of integers, in the range [1, +∞]. Some basic operations are available (add element, find the intersection…).
What kind of unit test are worth writing for the entity SetOfNaturals?
Complete the project, adding the new tests you identified.

2c/ Note that the provided code includes “magic numbers” (2 for the number of stars, 50 for the max range,…). Refactor the code to extract constants and eliminate the “magic numbers” bad-smell.

2d/ Assess the coverage level in project “Euromillions-play”.
Configure the maven project to run Jacoco analysis.
Run the maven “test” goal and then “jacoco:report” goal. You should get an HTML report under target/jacoco.
Interpret the results accordingly. Which classes/methods offer less coverage? Are all possible decision branches being covered?

Note: IntelliJ has an integrated option to run the tests with the coverage checks (without setting the Jacoco plugin in POM). But if you do it at maven level, you can use this feature in multiple tools.

Explore

- Book: JUnit in Action.
- Vogel’s tutorial on JUnit. Useful to compare between JUnit 4 and JUnit 5.
- Working effectively with unit testing (podcast).

Lab 2: Mocking dependencies in unit testing

Learning objectives

- Prepare a project to run unit tests (JUnit 5) and mocks (Mockito 3.x), with mocks injection (@Mock).
- Write and execute unit tests with mocked dependencies.
- Play with mock behaviors: strict/lenient verifications, advanced verifications, etc.

Lab activities

1a/ Implement the test case illustrated with the following classes, with respect to the **StockPortfolio#getTotalValue()** method. The method is expected to calculate the value of the portfolio by summing the current value (looked up in the stock market) of the owned stocks.
Be sure to use:

- Maven-based Java application project;
- Mockito framework (mind the maven dependencies).

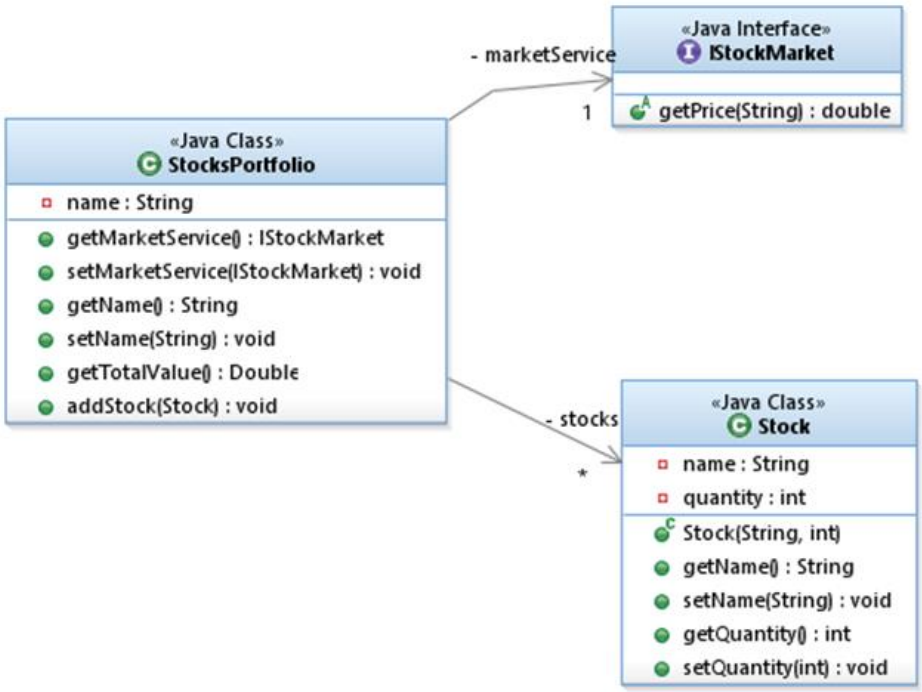


Figure 1: Classes for the StocksPortfolio use case.

a) Create the classes. You may write the implementation of the services before or after the tests.

b) Create the test for the getTotalValue(). As a guideline, you may adopt this outline:

```
1. Prepare a mock to substitute the remote service (@Mock annotation)
2. Create an instance of the subject under test (SuT) and use the mock to set the
   (remote) service instance (you may prefer to use @InjectMocks)
3. Load the mock with the proper expectations (when...thenReturn)
4. Execute the test (use the service in the SuT)
5. Verify the result (assert) and the use of the mock (verify)
```

Notes:

- Mind the JUnit version. For JUnit 5, you should use the @ExtendWith annotation to integrate the Mockito framework.
- Some IDE may not support JUnit 5 integration; you may need to further configure the POM.
- See a quick reference of Mockito syntax and operations.

1b/ Instead of the JUnit core asserts, you may use the Hamcrest library to create more human-readable assertions. Consider using this library in the previous example, in particular, assertThat(), is().

2/ Consider an application that needs to perform reverse geocoding to find a zip code for a given set of GPS coordinates. This service can be obtained in the Internet (e.g.: using the MapQuest API).

a) Create the objects represented in Figure 1. TqsHttpClient represents a service to initiate HTTP requests to remote servers. **You don't need to implement TqsHttpBasic**; in fact, you should provide a substitute for it.

b) Consider that we want to verify the AddressResolver#findAddressForLocation, which invokes a remote geocoding service, available in a REST interface, passing the site coordidates. Which is the service to fake?

c) To create a test for findAddressForLocation, you will need to know the exact response of the geocoding service for a sample request. Assume that we will use the MapQuest API. Use the browser or an HTTP client to try some samples so you know what to test for (example 1).

d) Implement a test for AddressResolver#findAddressForLocation using a mock.

e) Besides de "success" case, consider also testing for alternatives (e.g.: invalid coordinates should raise an exception).

This getting started project can used in your implementation.

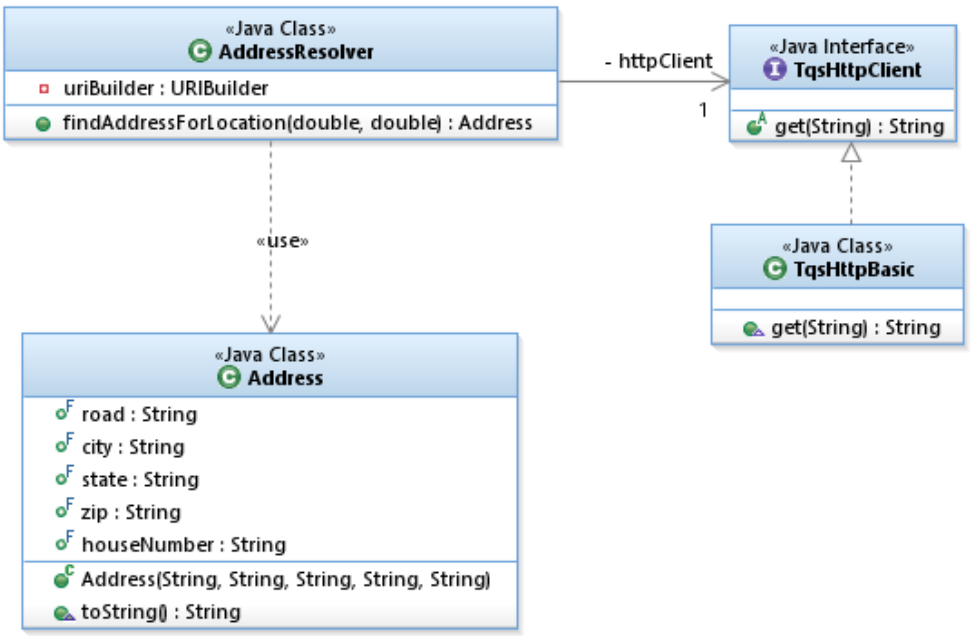


Figure 2: Classes for the geocoding use case.

3/ Consider you are implementing an integration test, and, in this case, you would use the real implementation of the module, not the mocks, in the test.
Create new test class and be sure its name end with "IT" (e.g.: GeocodeTestIT).
Copy the tests from the previous exercise into this new test class.
Remove all support for mocking (no dependencies on Mockito imports).
Correct the test implementation, so it used the real module.

If the “failsafe” maven plugin is configured, you should get different results with:
$ mvn test
$ mvn package failsafe:integration-test

Explore

JUnit 5 cheat sheet.

Lab 3: Functional testing with web automation

Prepare

— Selenium and separation of concerns

Key Points

— Acceptance tests (or functional test) exercise the user interface of the system, as if a real user was using the application. The system is treated as a black box.
— Browser automation (control the browser interaction from a script) is an essential step to implement acceptance tests on web applications. There are several frameworks for browser automation (e.g.: Puppeteer); for Java, the most used framework is the WebDriver API, provided by Selenium (that can be used with JUnit or TestNG engines).
— The test script can easily get "messy" and hard to read. To improve the code (and its maintainability) we could apply the Page Objects patterns.
— Web browser automation is also very handy to implement “smoke tests”.

Lab

Selenium works with multiple browsers but, for sake of simplicity, the samples will be discussed with respect to Chrome/Chromium; you may adapt for Firefox.

Suggested setup:
Install Chrome/Chromium in your system (if needed), using the default installation paths.
Download the ChromeDriver and make sure it is available available in the PATH. (GeckoDriver for Firefox)
Install the “Selenium IDE” browser plugin. (or, alternatively, the Katalon Recorder).

In this lab:

1. Create a web automation with Selenium IDE recorder
2. Run the test as a Java project (JUnit 5 + Selenium)
3. Use the Web Page Object pattern

1/ Create a web automation with Selenium IDE recorder

Access the Redmine demo site and create a temporary account (http://demo.redmine.org)
Be sure to logout before recording the test.
Create (record) an automation macro with the Selenium IDE recorder tool to test login (a quick start for Katalon is available, which is similar to Selenium IDE):

a) Open http://demo.redmine.org
b) Sign-in with your credentials
c) Assert that you have successfully logged in (by verifying the presence of the username)
d) Logout

... and Stop recording. Test your macro (replay).

Add a new step, at the end, to confirm that, after logout, the home shows the "Sign in" option present. Enter this assertion "manually" (in the editor, but not recording).

2a/ Run the test as a Java project (JUnit 5, Selenium)

Prepare a (new) project to run JUnit tests and Selenium (sample POM.xml available; alternative site).

Take note of the information in this page under "quick reference"; then, in the section "Local browsers", pick the example that suites your setup and run the test (as you usually do with JUnit). You will have to deploy the WebDriver implementation (binary) for you browser [→ download browser driver]. Be sure to include in the system PATH.

2b/ Export and run the test (Webdriver)

Export the test from Selenium IDE into a Java test class and include it in the previous project. Refactor the code that was generated to be compliant with JUnit 5 and the Selenium-Jupiter extension.

Run the test (programmatically).

3/ Use the Web Page Object pattern

Consider the example discussed here.
Implement the "Page object pattern" for a cleaner and more readable test, as suggested.

Notes:
Execute the suggested steps to interactively record the test case. Export it to Java and run the test with JUnit automation, refactoring for JUnit 5.
The text in the tutorial is somewhat old. You may **need to adapt to the current implementation** of the site under test (e.g.: IDs of page elements,...).
You should **stop** at "Increased readability" with Cucumber.

Explore

- Puppeteer - a Node library which provides a high-level API to control headless Chrome/Chromium.
- Another, more recent, Page Object Model example.

Lab 4: Behavior-driven development (Cucumber in Java)

Key Points

— The Cucumber framework enables the concept of "executable specifications": with Cucumber we use concrete examples to specify what we want the software to do. **Scenarios are written before production code**.
— Cucumber executes features (test scenarios) written with the Gherkin language (readable by non-programmers too).

— The steps included in the feature description (scenario) must be mapped into Java test code by annotating test methods with matching "expressions". Expressions can be (traditional) regular expressions or the (new) Cucumber expressions.

Lab

Bear in mind that the integration of JUnit 5 and Cumcumber is still very recent and most of the samples available in the internet are still based on JUnit 4.

In this lab you will:
1/ Create a simple cumcumber-enabled project
2/ Book search example
3/ Integrate Cucumber with Selenium Webdriver

1/ Create a simple cucumber-enabled project

This example assumes that you have Maven available in the path and you can invoke mvn from the command line. Check with:

Go through the Cucumber getting started tutorial for Java.
The example uses the command line and the IntelliJ IDEA (you may use other IDE, but you will need to adapt the instructions).

Note that:
→ Start the project from the suggested Maven archetype io.cucumber:cucumber-archetype
→ You need a test class that activates the Cucumber runner; that is the purpose of the "RunCucumberTest" file automatically included in this archetype.
→ the features are stored under src/test/resources/ and the folder structure (under resources) must mirror the package hierarchy under src/test/java). In this sample: src/test/resources/hellocumber/ and src/test/java/hellocumcumber/
→ IntelliJ recognizes the .feature file type. You can even select "Run all features" to run the Cucumber tests.

2/ Book search example

To get into the "spirit" of BDD, partner with a colleague, and jointly write a couple of features to verify a book search user story. Consider a few search options (by year, etc).

Take the approach discussed in this example, and write your own tests. Feel free to add different scenarios/features.

The sample uses Cucumber v2, but you should write the test steps using Cucumber 3x. Some changes are required:
In the "book search" story:

— You will need to change the parameters placeholder in the steps definition. Prefer the "cucumber expressions" (instead of regular expressions). [→ partial snippet]

— migrate the date formatter option, by creating a new datatype in a "Configurer" class, which should be placed in the same package as the test steps [→ possible solution snipet]. This defines a new custom parameter type ("`date_iso_local_date_time`")

— The dates in the feature description need also to match the date mask used (aaaa-mm-dd).

In the "salary" story:

— Adapt the data table definition for the Salary use case [→ possible solution snipet]. This allows to extract a List<Employees> from the feature definition and use it as a parameter.

```
public void
the_salary_management_system_is_initialized_with_the_following_data(final
List<Employee> employees)
```

Notes:
→ you may build on the previous project or start with a new one, but stick with the base archetype (io.cucumber:cucumber-archetype).

3/ Integrate Cucumber with Selenium Webdriver

Implement the brief sample using cucumber and selenium webdriver to declare expressive web automation tests.
You may now extend this strategy to revisit the example (Weather forecast) of lab 3.

Lab 5: Multi-layer application testing with Spring Boot

Prepare

This lab is based on Spring Boot. Most of students already used the Spring Boot framework (in IES course).
If you are new to Spring Boot, then you need to develop a basic understanding or collaborate with a colleague. Learning resources are available at the Spring site.

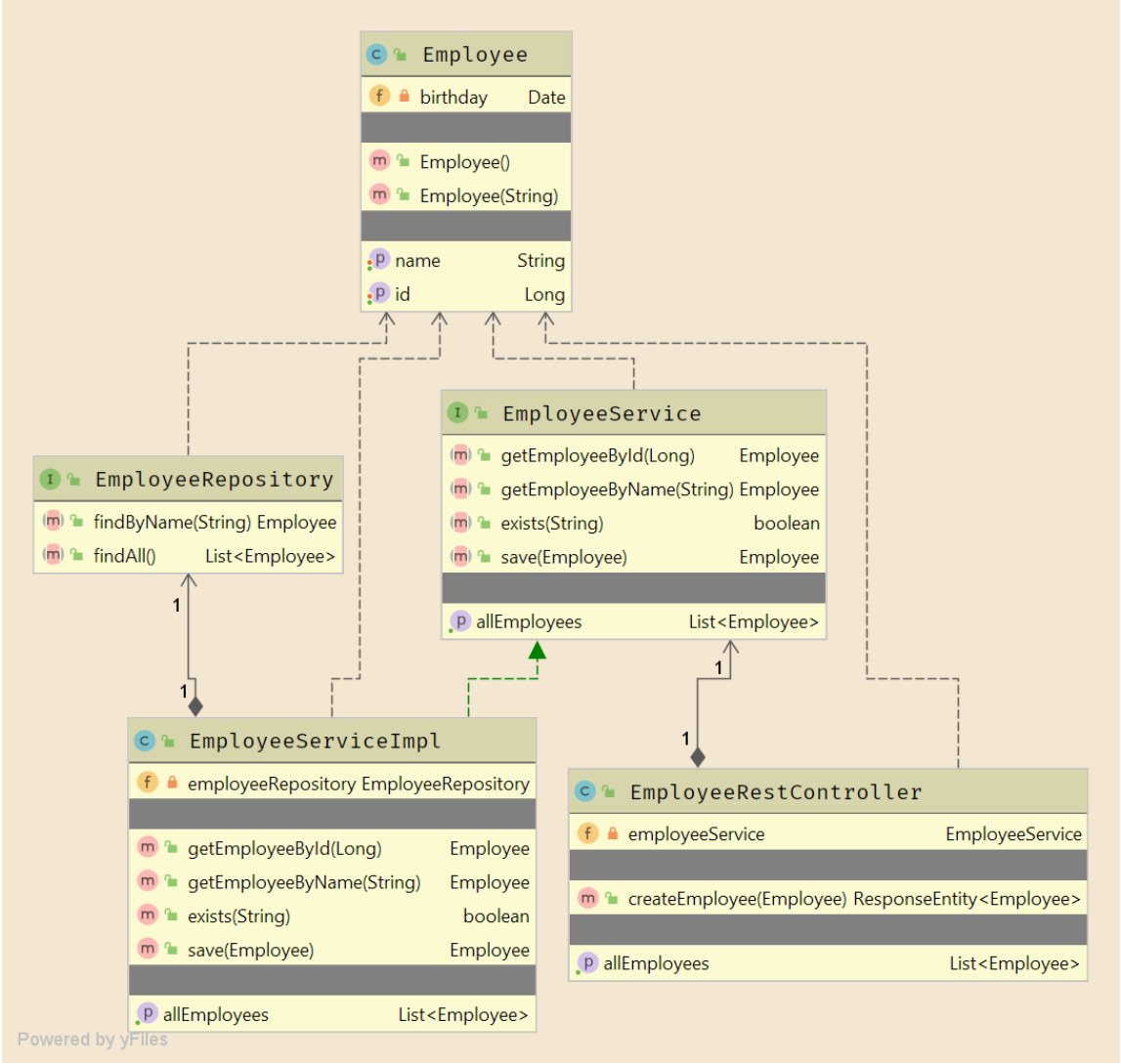
Key Points

- Isolate the functionality to be tested by limiting the context of loaded frameworks/components. For some use cases, you can even test with just standard unit testing.
- @SpringBootTest annotation loads whole application, but it is better to limit Application Context only to a set of spring components that participate in test scenario
- @DataJpaTest only loads @Repository spring components, and will greatly improve performance by not loading @Service, @Controller, etc.
- Use @WebMvcTest to test rest APIs exposed through Controllers. Beans used by controller need to be mocked.

Lab

1/
Study the example available concerning a simplified Employee management application (gs-employee-manager).
This application follows commons practices to implement a Spring Boot solution:

- Employee: entity (@Entity) representing a domain concept.
- EmployeeRepository: the interface (@Repository) defining the data access methods on the target entity, based on the framework JpaRepository. “Standard” requests can be inferred and automatically supported by the framework (no additional implementation required).
- EmployeeService and EmployeeServiceImpl: define the interface and its implementation (@Service) of a service related to the “bizz logic” of the application. Elaborated decisions/algorithms, for example, would be implemented in this component.
- EmployeeRestController: the component that implements the REST-endpoint/boundary (@RestController): handles the HTTP requests and delegates to the EmployeeService.



The project contains a set of tests. Take note of the following test scenarios:

Purpose	Strategy	Notes
A/ Verify the data access services provided by the repository component. [EmployeeRepositoryTest]	Slice the test context to limit to the data instrumentation (@DataJpaTest) Inject a TestEntityManager to access the database; use directly this object to write to the database.	@DataJpaTest includes the @AutoConfigureTestDatabase. If a dependency to an embedded database is available, an in-memory database is set up. Be sure to include H2 in the POM.
B/ Verify the business logic associated with the services implementation. [EmployeeServiceImplUnitTest]	Can be achieved with a unit test, if we mock the repository behavior. Rely on Mockito to control the test and to set expectations and verifications.	Relying only in JUnit + Mockito makes the test a unit test, much faster that using a full SpringBootTest. No database involved.
C/ Verify the boundary components (controllers). No need to test the real HTTP-REST framework; just the controller behavior. [EmployeeControllerIT]	Run the tests in a simplified light environment, simulating the behavior of an application server, by using @WebMvcTest mode. Get a reference to the server context with @MockMvc. To make the test more localized to the controller, you may mock the dependencies on the service	MockMvc provides an entry point to server-side testing. Despite the name, is not related to Mockito. MockMvc provides an expressive API, in which methods chaining is expected.

	(@MockBean); the repository component will not be involved.	
D/ Verify the boundary components (controllers). Test the REST API on the server-side; no API client involved. [EmployeeRestControllerIT]	Start the full web context (@SpringBootTest, with Web Environment enabled). The API is deployed into the normal SpringBoot context. Use the entry point for server-side Spring MVC test support (MockMvc).	This would be a typical integration test in which several components will participate (the REST endpoint, the service implementation, the repository and the database).
E/ Verify the boundary components (controllers). Test the REST API with explicit HTTP client involved. [EmployeeRestControllerTemplateIT]	Start the full web context (@SpringBootTest, with Web Environment enabled). The API is deployed into the normal SpringBoot context. Use a REST client to create realistic requests (TestRestTemplate)	Similar to the previous case, but instead of assessing a server entry point for tests, start a API client (so request and response un/marshaling will be involved).

Review questions:

a) Identify a couple of examples on the use of AssertJ expressive methods chaining.

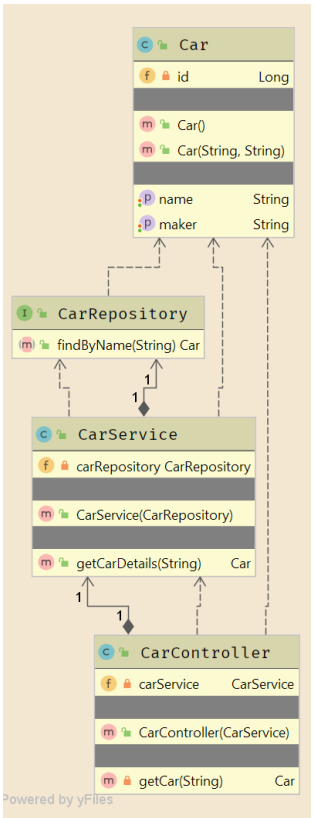
b) Identify an example in which you mock the behaviour of the repository (and avoid involvind a database).

c) What is the difference between standard @Mock and @MockBean?

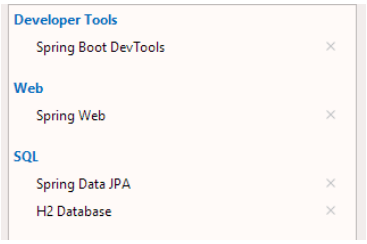
d) What is the role of the file "application-integrationtest.properties"? In which conditions will it be used?

2/

Consider the case in which you will develop an API for a car information system.



Implement this scenario, as a Spring Boot application. Consider using the Spring Boot Initializr to create the new project (integrated in IntelliJ or online); add the "starters" for Developer Tools, Spring Web, Spring Data JPA and H2 Database.



Take the structure modeled in the classes diagram as a (minimal) reference.

In this exercise, try to force a TDD approach: write the test first; make sure the project can compile without errors; defer the actual implementation of production code as much as possible. This will be forced if we try to write the tests in a top-down approach: start from the controller, than the service, than the repository.

a) Create a test to verify the CarController (and mock the CarService bean). Run the test.

b) Create a test to verify the CarService (and mock the CarRepository). This can be a standard unit test with mocks.

c) Create a test to verify the CarRepository persistence. Be sure to include a in-memory database dependency in the POM (e.g.: H2).

d) Having all the previous tests passing, implement an integration test to verify the API. Suggestion: use the E/ approach discussed for the previous (Employees) example.

e) Adapt the integration test to use a real database. E.g.:

- Run a mysql instance and be sure you can connect (for example, using a Docker container)

- Change the POM to include a dependency to mysql;
- Add the connection properties file in the resources or the “test” part of the project (see the application-integrationtest.properties in the sample project)
- Use the @TestPropertySource and deactivate the @AutoConfigureTestDatabase.

Lab 6: Static Code analysis with Sonar Qube

Prepare

You will find a lot of demos in Youtube under for static code analysis in Java with SonarQuebe.
It is not required but you may choose to have a look.

Key Points

Static code quality can be inspected to obtain quality metrics on the code base. These metrics are based on the occurrence of known weaknesses, a.k.a. “code smells”. The code is not executed (thus static analysis).
Key measures include the occurrence of problems likely to produce errors, vulnerabilities (security/reliability concerns) and code smells (bad/poor practice or coding style); coverage (ratio tested/total); and code complexity assessment.
The estimated effort to correct the vulnerabilities is called the **technical debt**. Every software quality engineer must use specialized tools to obtain realistic technical debt information.

Lab

In this lab:
Task 1: Analyze an existing project (maven-based)
Task 2: Include tests and coverage
Task 3: Define and apply quality gates

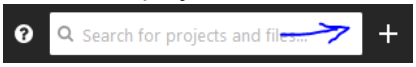
Task 1: Analyze an existing project (maven-based)

a/ Install the SonarQube server either as a local service, or by using the docker image (see instructions in the link).
For the purpose of this lab, you don’t need to configure a production database (the embedded H2 database is used by default). If, however, you want a production-like setup, you should consider using a persistent database (for example, like this configuration; not required for the class).

b/ Configure the environment for Sonar Scanner for Maven. Be sure to adapt the <sonar.host.url> for your setup (e.g.: http://127.0.0.1:9000)

c/ Select a Maven-based, Java application project to use. You may reuse one from previous labs, for example, the Euromillions from Lab 1 (part 2b), with tests passing.
Configure the POM to integrate the Sonar scanner plugin (section #2, in this guide)

d/ Create a project in the Sonar Qube dashboard (default : http://127.0.0.1:9000).

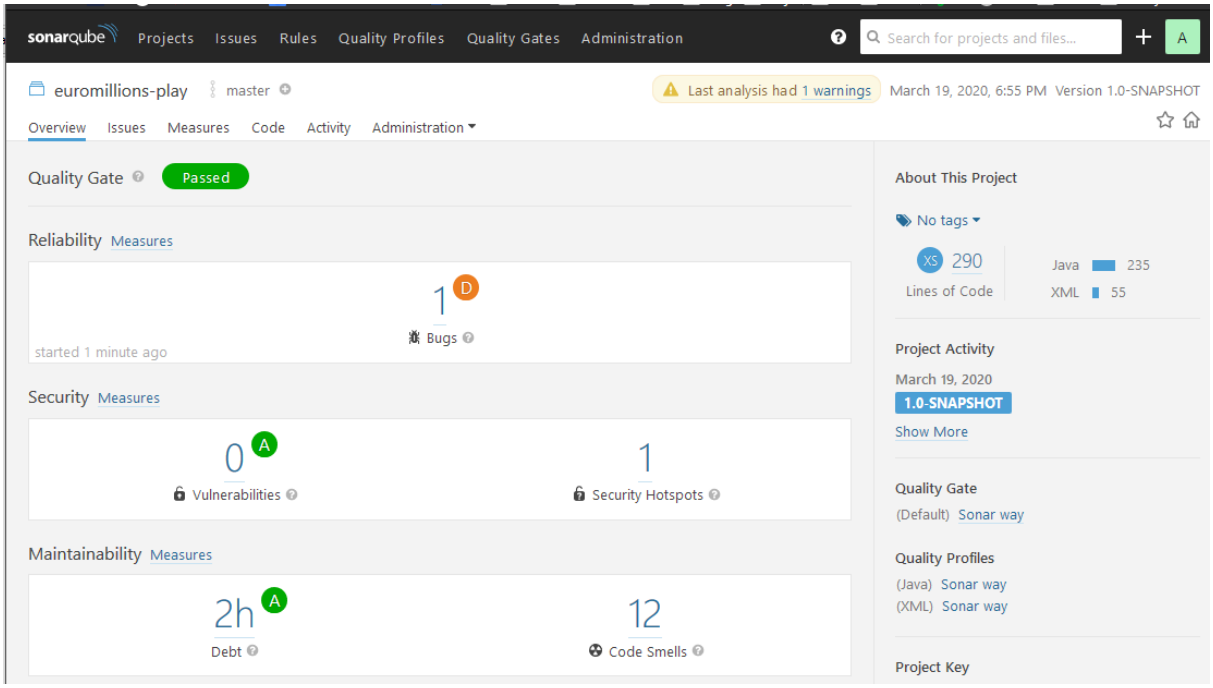


Be sure to define the **project key equal to the maven** GroupId:ArtifactId elements.

Then, generate a token for the project and take note for later use.

Then, in the command line, run the code analysis (adapt as needed):
$ mvn sonar:sonar -Dsonar.projectKey=tqslabs:euromillions-play -
Dsonar.host.url=http://127.0.0.1:9000 -Dsonar.login=9d0ca979aff1cacd052c669072275f70ba2bd512

e/ observe the Sonar dashboard. Has the project passed the defined quality gate?



Explore the analysis results and complete with a few sample issues (if applicable):

Issue (be severity)	**Problem description**	**How to solve**
Bug	...	...
Vulnerability (severe)		
Code smell (severe)		

f/ Describe the metrics assessed in the Quality Gate that is being used (Sonar Dashboard --> top menu --> Quality Gates view):

Metric	**Conditions**	**Explanation (in your own words)**
Coverage on New Code	Fails if < 80	How much of the UPDATED source code has been covered by the unit tests? Mixes Line coverage and Condition coverage.
...		

Task 2: Include tests and coverage

For this part, be sure you are using a project with JUnit tests implemented and passing. Let us assume the Euromillions project.

a/ Take note of the technical debt found. Explain what this value means.

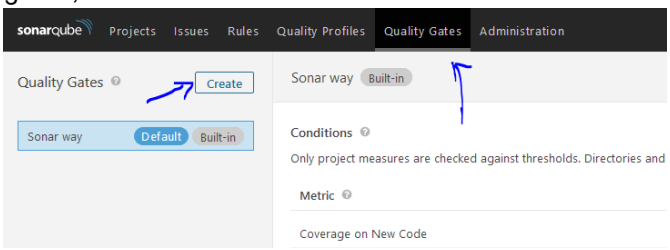
b/ Analyze the reported problems and be sure to **correct the severe** code smells reported (critical and major).

c/ Code coverage reports require an additional plugin. Be sure to use a project with unit tests and configured code coverage (e.g.: add the jacoco plugin to maven). You may have already did it in Lab 2 (2d).

d/ Run the static analysis and observe/explore the coverage values on the Sonar Qube dashboard. How many lines are "uncovered"? And how many conditions?

Task 3: Define and apply quality gates

a/ Define a custom quality gate to this project. Consider using: coverage %, Maintainability Rating grade, unit tests success %. Feel free to mix other metrics.



b/ Add an increment to the source code. You may try to introduce some "bad smell".
Does the updated project pass the quality gate?

Explore:

- Use the SonarQube to inspect a project of your own, maybe from another course. Note that free analyzers are not available for all languages.
- A related tutorial by Baeldung.
- public projects on Sonar cloud that you can browse and learn.