# TQS Lab activities

v2023-03-23

**TQS Lab activities** ........ **1**
**Introductory notes and setup** ........ **1**
**Lab 1 Unit testing (with JUnit 5)** ........ **2**
1.1 Stack contract ........ 2
1.2 EuroMillions ........ 4

**Lab 2 Mocking dependencies (for unit testing)** ........ **5**
Context and key points ........ 5
2.1 Stocks portfolio ........ 6
2.2 Geocoding ........ 7
2.3 Integration tests (with maven failsafe plugin) ........ 7

**Lab 3 Multi-layer application testing (with Spring Boot)** ........ **8**
Context and key points ........ 8
3.1 Employee manager example ........ 8
3.2 Cars service ........ 10
3.3 Integration test ........ 11

**Lab 4 Acceptance testing with web automation (Selenium)** ........ **12**
Context and key points ........ 12
4.1 WebDriver starter ........ 12
4.2 Selenium IDE recorder ........ 12
4.3 Page Object pattern ........ 13
4.4 Browser variations ........ 14

**Lab 5 Behavior-driven development (Cucumber)** ........ **14**
Context and key points ........ 14
5.1 Getting started ........ 14
5.2 Books ........ 15
5.3 Web automation ........ 16

**Lab 6 Static Code analysis (with SonarQube)** ........ **16**
Context and key points ........ 16
6.1 Local analysis ........ 16
6.2 Technical debt (Cars) ........ 17
6.3 Custom QG ........ 18
6.4 IDE ........ 18

## Introductory notes and setup

### Work submission

You should create a personal (git) repository for your TQS **individual portfolio** in which you will be including your solutions for the labs (e.g.: **tqs_123567** , the number being your student number). Keep a **clean organization** that maps the exercise structure, e.g.: 🗀lab1/lab1_1; 🗀lab1/lab1_2; 🗀lab2/lab2_1; 🗀lab2/lab2_2…
You are expected to keep your repo (portfolio) up to date and complete. Teachers will select a few exercises later for assessment [not all, but representative samples].

**Lab activities**

Be sure that your developer environment meets the following requirements:

— Java development environment (JDK; v11 or v17 suggested). 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. Check with:

```
$ mvn --version
```

— Java capable IDE, such as IntelliJ IDEA (version "Ultimate" suggested) or VS Code.

# Lab 1 Unit testing (with JUnit 5)

**Learning objectives**

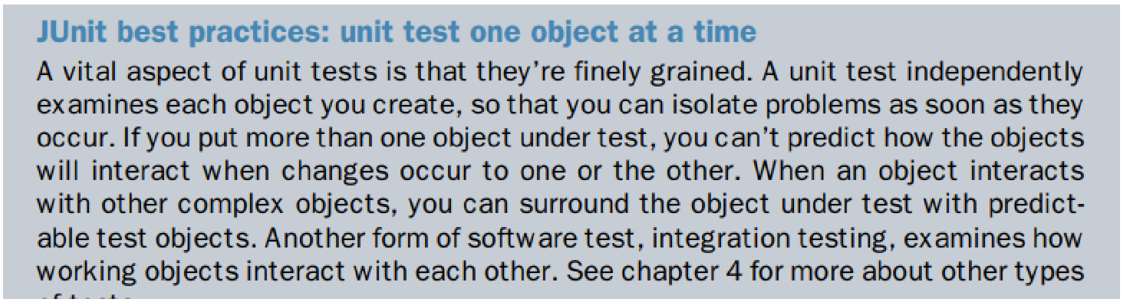
— Identify relevant unit tests to verify the contract of a module.

— Write and execute unit tests using the JUnit framework.

— Link the unit tests results with further analysis tools (e.g.: code coverage)

**Key points**

- Unit testing is when you (as a programmer) write test code to verify units of (production) code. A unit is a small, coherent subset of a much larger solution. A true "unit" should not depend on the behavior of other (collaborating) modules.
- 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 in the code.
- 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

## 1.1 Stack contract

In this exercise, you will implement a stack data structure (TqsStack) with appropriate unit tests. Be sure to adopt a **write-the-tests-first** workflow:

a) Create a new project (**maven project** for a Java standard application). You may need to update the Java version in the POM.xml and other dependencies. You may "clone" from a sample project:

- Adapt from the quick start project for Maven[1].

b) Add the required dependencies to run JUnit 5 tests[2]. Example:

- sample content for POM.xml (note the elements: **junit-jupiter** and **maven-surefire-plugin** )

```xml
<!-- ... -->
<dependencies>
    <!-- ... -->
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter</artifactId>
        <version>5.9.2</version>
        <scope>test</scope>
    </dependency>
    <!-- ... -->
</dependencies>
<build>
    <plugins>
        <plugin>
            <artifactId>maven-surefire-plugin</artifactId>
            <version>3.0.0-M7</version>
        </plugin>
        <plugin>
            <artifactId>maven-failsafe-plugin</artifactId>
            <version>3.0.0-M7</version>
        </plugin>
    </plugins>
</build>
<!-- ... -->
```

c) Create the required class definition (**just the "skeleton"**, do not implement the methods body yet!). The **code should compile**, but the **implementation is yet incomplete** (you may need to add dummy return values).

d) Write 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. [Mixing JUnit 4 and JUnit 5 dependencies will prevent the test methods to run as expected!]

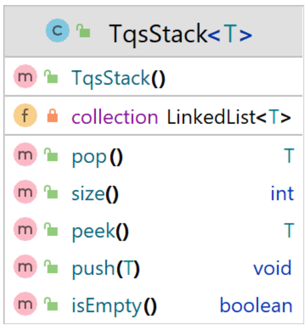
Your tests will verify several assertions that should evaluate to true for the test to pass.

e) Run the tests and prove that TqsStack implementation is not valid yet (the tests should **run** and **fail** for now, the first step in Red-Green-Refactor).

f) Correct/add the missing implementation to the TqsStack;

g) Run the unit tests.

h) Iterate from steps d) to f) and confirm that all tests are passing.



```
TqsStack<T>
TqsStack()
collection LinkedList<T>
pop()          T
size()         int
peek()         T
push(T)        void
isEmpty()      boolean
```

Suggested stack contract:

— 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**[3]:

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

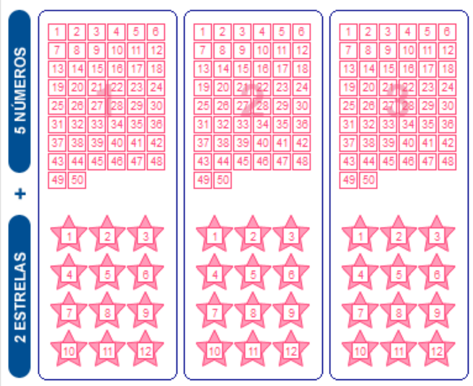
f) If the size is n, then after n pops, the stack is empty and has a size 0

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

[1] Delete the "pom-SNAPSHOT.xml", if you are cloning the project to use as a quick starter.
[2] If using IntelliJ: you may skip this step and ask, later, the IDE to fix JUnit imports.
[3] Adapted from http://cs.lmu.edu/~ray/notes/stacks/

i) For bounded stacks only: pushing onto a full stack does throw an IllegalStateException

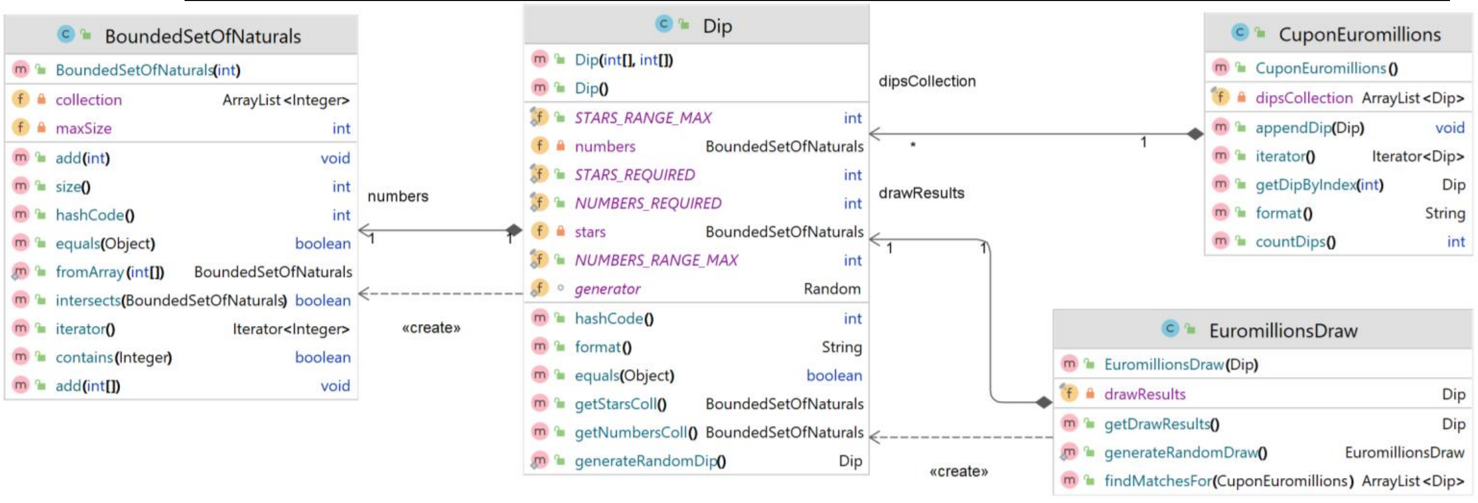


## 1.2 EuroMillions

Let us consider the "Euromilhões" use case.

**2a**/ Pull the "euromillions-play" project
The supporting implementations is visualized in the class diagram that follows.
Get familiar with the solution and existing tests.

| Class | Purpose |
|---|---|
| BoundedSetOfNaturals | Reusable set data structure<br>no duplicates allowed (it is a Set)<br>only natural numbers in the range [1, +∞].<br>the max size of the set (count of elements) is bounded to a limit<br>allows set operations (contains element?, append element, calculate intersection with another set,…) |
| Dip | A collection of 5 "numbers" and 2 "stars" (a "column" in the Euromillions playing coupon) |
| CouponEuromillion | One or more Dips, representing a bet from a player. |
| EuromillionsDraw | Holds the winning dip and can find matched for a given player coupon. |



**2b**/ Make the necessary changed for the existing (non-disabled) unit tests pass.

| For the (failing) test: | You should: |
|---|---|
| testConstructorFromBadRanges | Change Dip implementation.<br>Be sure to raise the expected exception if the arrays have invalid numbers (out of range numbers) |

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

**2c/ Assess the coverage** level in project "Euromillions-play".

Configure the maven project to run Jacoco analysis, if needed.
Run the maven "test" goal and then "jacoco:report" goal. You should get an HTML report under *target/jacoco*.

```
$ mvn clean test jacoco:report
```

Analyze the results accordingly. Which classes/methods offer less coverage? Are all possible [decision] branches being covered?
**Collect evidence** of the coverage for "BoundedSetOfNaturals".

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.

**2c/**
Consider the class BoundedSetOfNaturals and its expected contract.
What kind of unit test are worth writing for proper validation of BoundedSetOfNaturals?
Complete the project, adding the tests you have identified. (You may also enhance the implementation of BoundedSetOfNaturals, if necessary.)

**2d/**
Run Jacoco coverage analysis and compare with previous results. In particular, compare the "before" and "after"for the BoundedSetOfNaturals class.

#### Troubleshooting some frequent errors

➔ "Test are run from the IDE but not from command line."
  Be sure to configure the Surefire plug-in in Maven (example).

#### Explore

- JetBrains Blog on Writing JUnit 5 tests (with vídeo).
- Book: JUnit in Action. Note that you can access it from the OReilly on-line library.
- Book: "Mastering Software Testing with JUnit 5" and associated GitHub repository with examples
- JUnit 5 cheat sheet.

# Lab 2 Mocking dependencies (for unit testing)

## Context and key points

#### Learning objectives

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

#### Preparation

Get familiar with sections 1 to 3 in the Mockito (Javadoc) documentation.

#### Explore

- There is a recent book on JUnit and Mockito available from OReilly. The lessons are available as short videos too.

## 2.1 Stocks portfolio

Consider the example in Figure 1: the StocksPortfolio holds a collection of Stocks; the current value of the *portfolio* depends on the current condition of the *Stock Market*. **StockPortfolio#getTotalValue()** method calculates the value of the portfolio (by summing the current value of owned stock, looked up in the stock market service).

**1a/**
Implement (at least) one test to verify the implementation of **StockPortfolio#getTotalValue().** Given that test should have predictable results, you need to address the problem of having non-deterministic answers from the **IStockmarketService** interface.

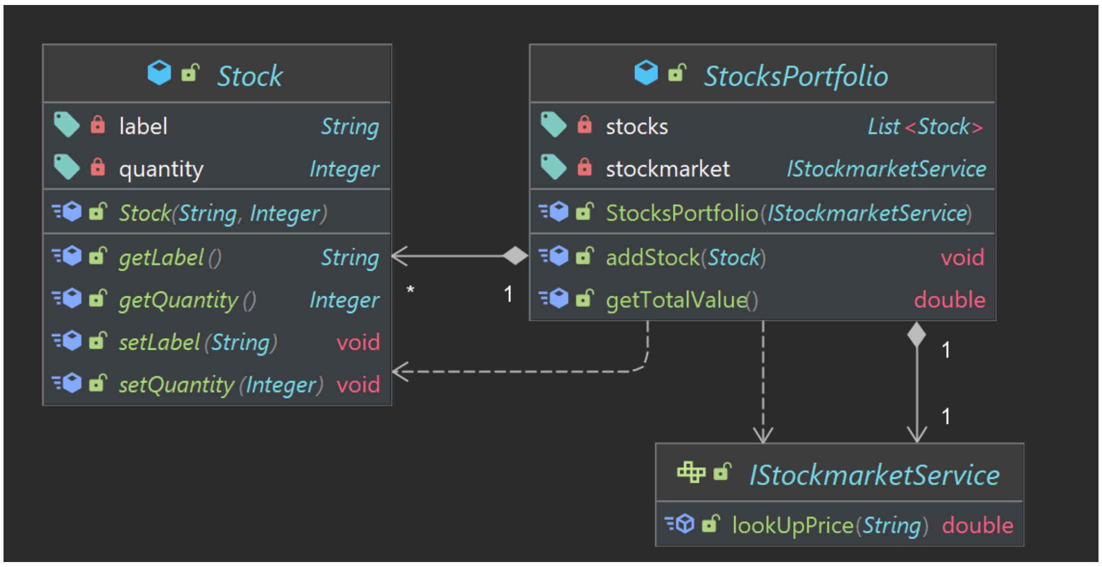


*Figure 1: Classes for the StocksPortfolio use case.*

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.
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:

— Consider use these Maven dependencies for your POM (JUnit5, Mockito).

— Mind the JUnit version. For JUnit 5, you should use the @ExtendWith annotation to integrate the Mockito framework.

```
@ExtendWith(MockitoExtension.class)
class StocksPortfolioTest { … }
```

— See an example of the main syntax and operations.

**1b/** Instead of the JUnit core asserts, you may use the Hamcrest library to create more human-readable assertions. Replace the "Assert" statements in the previous example, to use Hamcrest constructs. See example.

## 2.2 Geocoding

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 assisted by public APIs (e.g.: using the MapQuest API).
Let as create a simple application to perform (reverse) geocoding and set a few tests.

a) Create the objects represented in Figure 1. At this point, **do not implement TqsBasicHttpClient**; in fact, you should provide a substitute for it.

b) Consider that we want to test the behavior of AddressResolver#findAddressForLocation, which invokes a remote geocoding service, available in a REST interface, passing the site coordinates.
Which is the SuT (subject under test)? Which is the service to *mock*?

c) To create a test for findAddressForLocation, you will need to mimic **the exact (JSON) response of the geocoding service for a request**. Study/try the MapQuest API. See sample of response.

d) Implement a test for AddressResolver#findAddressForLocation (using mocks where required).

e) Besides the "success" case, consider also testing for alternatives (e.g.: bad coordinates;…). You may need to change the implementation.

This getting started project [gs-mockForHttpClient] can be used in your implementation.

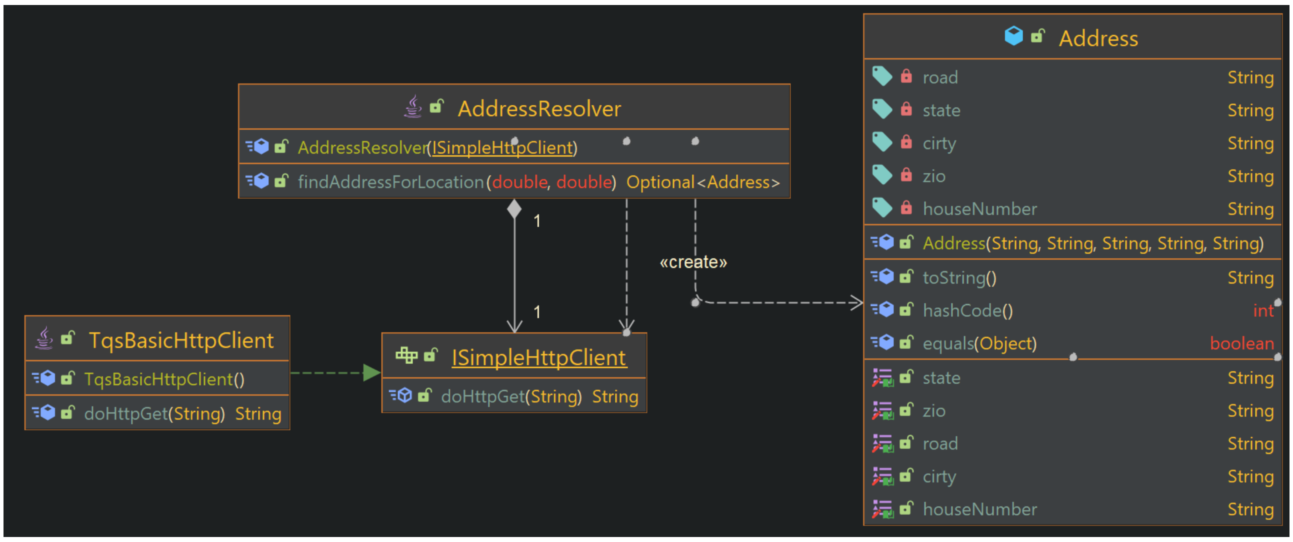


*Figure 2: Classes for the geocoding use case.*

## 2.3 Integration tests (with maven failsafe plugin)

Consider you are implementing an **integration test** for the previous example, and, in this case, you would use the real implementation of the module, not the mocks, in the test.
(This section can be included with the previous, continuing the same project.)

Create a test class (AddressResolver**IT**), in a separate test package, and be sure its name ends with "**IT**". Copy the tests from the previous exercise into this new test class but remove any support for mocking (no Mockito imports in this test).
Correct/complete the test implementation so it uses the real HttpClient implementation.
Run your test (and confirm that the remote API is invoked in the test execution).

Be sure the "failsafe" maven plugin is configured.
You should **get different results** with the following cases (try with and without internet connection):

```
$ mvn test
and
$ mvn install failsafe:integration-test
```

(Note the number of tests and the time required to run the tests…).

# Lab 3 Multi-layer application testing (with Spring Boot)

## Context and key points

**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**

- @SpringBootTest annotation loads whole application context, but it is better (faster) to limit application contexts 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.
- 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.

**Explore**

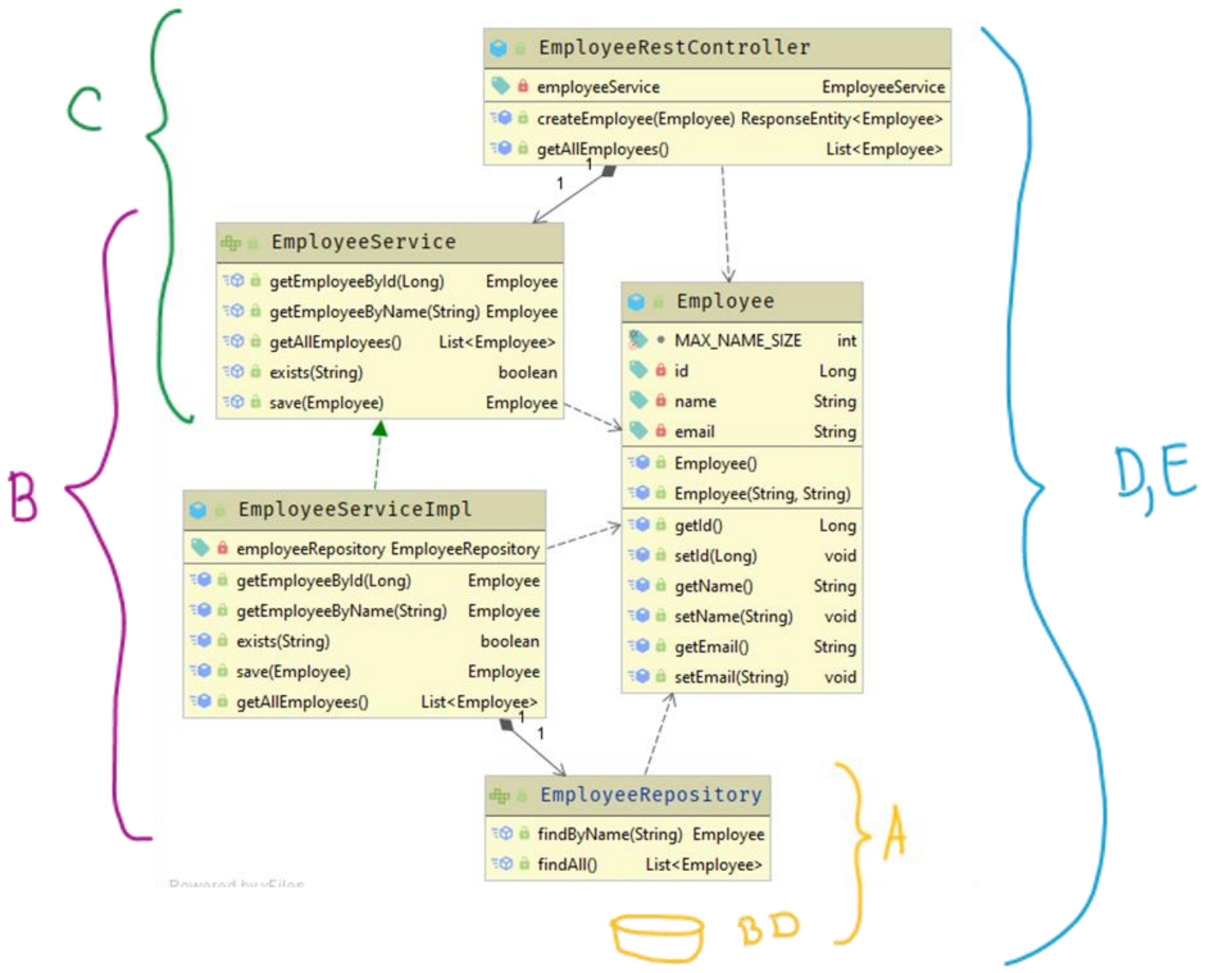
- AssertJ library to create expressive assertions in tests: https://assertj.github.io/doc/
- Talk on Spring Boot tests (by Pivotal): https://www.youtube.com/watch?v=Wpz6b8ZEgcU

## 3.1 Employee manager example

**Study the example** concerning a simplified Employee management application (project: gs-employee-manager).
This application follows the Spring Boot style to structure the 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; custom queries should be declared, if needed.
- EmployeeService and EmployeeServiceImpl: define the interface and its implementation (@Service) of a service related to the "business 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 already contains a set of tests.



Look up and study the following test scenarios:

| **Purpose/scope** | **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 this object to write to the database directly (no caches involved). | @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. [*EmployeeService_UnitTest*] | Often can be achieved with unit tests, given one mocks the repository. 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); just the controller behavior. [*EmployeeController_ WithMockServiceTest*] | Run the tests in a simplified and 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 (@MockBean); the repository component will not be involved. | 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. In principle, no database is involved. |

| | | |
|---|---|---|
| **D**/ Verify the boundary components (controllers). Load the full Spring Boot application. 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). Load the full application. Test the REST API with explicit HTTP client. [*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 convenient servlet entry point for tests, uses an API client (so request and response un/marshaling will be involved). |

Note 1: both D/ and E/ load the full Spring Boot Application (auto scan, etc…). The main difference is that in D/ one accesses the server context through a special testing servlet (MockMvc object), while in E/ the requester is a REST client (TestRestTemplate).

Note 2: you may run individual tests using maven command line options. E.g.:

```
$ mvn test -Dtest=EmployeeService*
```

Review questions: [answer in a **readme.md** file, in /lab3_1 folder]

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

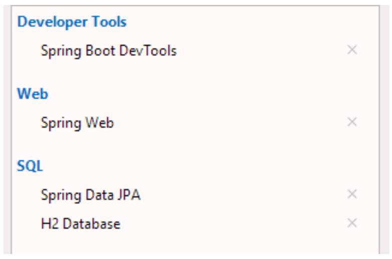
b) Identify an example in which you mock the behavior of the repository (and avoid involving a database).

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?

e) the sample project demonstrates three test strategies to assess an API (C, D and E) developed with SpringBoot. Which are the main/key differences?

## 3.2 Cars service


Developer Tools
Spring Boot DevTools
Web
Spring Web
SQL
Spring Data JPA
H2 Database

Consider the case in which you will develop an API for a car information system (as a Spring Boot application).
Consider using the Spring Boot Initializr to create the new project (either online or may be integrated in your IDE);
Add the dependencies (*starters*) for: Developer Tools, Spring Web, Spring Data JPA and H2 Database.

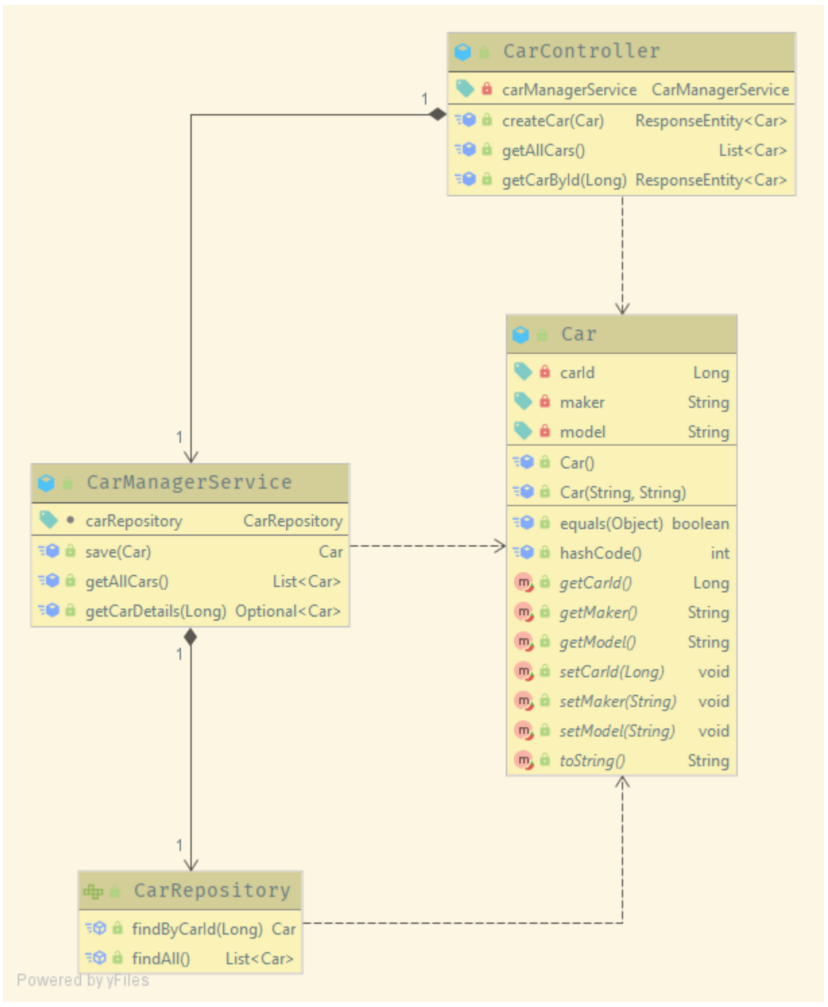
Use the structure modeled in the class diagram as a (minimal) reference.
In this exercise, **try to force a TDD approach**: write the test first; make sure the project can compile without errors; **defer the actual implementation of production code as much as possible**.
This approach will be encouraged if we try to write the tests in a top-down approach: **start from the controller, then the service, then the repository**.

a) Create a test to verify the Car [Rest]Controller (and mock the CarService bean), as "resource efficient" as possible. 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 an 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 approach "**E/**" discussed in the previous project (Employees).



Note:
Although the Service in this example is quite trivial (just delegates to repository), in a larger application, we would expect the "services" to implement more complex, interesting, and mission-critical business logic, e.g.: "find a car that provides a *suitable replacement* for some given car [e.g.: as a courtesy car]".

## 3.3 Integration test

[Continue in the same project of the previous exercise.]
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 [optionally remove H2].
- Add the connection properties file in the resources of the "test" part of the project (see the application-integrationtest.properties in the sample project)
- Use the @TestPropertySource and deactivate the @AutoConfigureTestDatabase.

# Lab 4 Acceptance testing with web automation (Selenium)

## Context and key points

**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 engine).

— Selenium is an umbrella project for a range of tools and libraries that enable and support the automation of web browsers.

— The test script can easily get "messy" and hard to read. To improve the code (and its maintainability) we could apply the Page Objects Pattern.

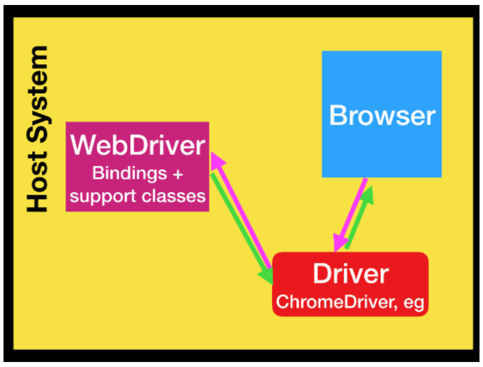
— Web browser automation is also very handy to implement "smoke tests".

**Explore**

- Book "Hands-On Selenium WebDriver with Java"
- Another Page Object Model example. Criticism on the Page Object Pattern for modern web apps (and alternatives).

## 4.1 WebDriver starter

Selenium WebDriver offers a concise programming interface (i.e., API) to drive a (web) browser, as if a real user is operating the browser.



A) Note the "getting started" section available from Selenium documentation (dependencies, instantiating the browser binding, etc…).

Use method #2 (better) or #3 explained in "Install drivers" page. For #3, you need to download the driver implementation for your browser and add driver implementation to the system PATH.

B) Implement the example discussed in the "hello world" section of B. Garcia's book (Example 2-1).

Main dependencies for POM.xml:

- `org.seleniumhq.selenium:`**`selenium-java`**
- `io.github.bonigarcia:` **`selenium-jupiter`**
- `org.junit.jupiter:`**`junit-jupiter`**

## 4.2 Selenium IDE recorder

Usually, you can use the Selenium IDE to prepare/record your tests interactively and to explore the "locators" (e.g.: id for a given web element).

A) Record the test interactively
Install the Selenium IDE plug-in/add-on for your browser.

Using the https://blazedemo.com/ dummy travel agency web app, record a test in which you select a and buy a trip. Be sure to add relevant "asserts" or verifications to your test.
Replay the test and confirm the success. Also experiment to break the test (e.g.: by explicitly editing the parameters of some test step).

Add a new step, at the end, to assert that the confirmation page contains the title "BlazeDemo Confirmation". Enter this assertion "manually" (in the editor, **not** recording).

Be sure to save you Selenium IDE test project (it creates a *.side file, to be included in your git).

B) Export and run the test using the Webdriver API
Export the test from Selenium IDE into a Java test class and include it in the previous project.
Refactor the generated code to be compliant with JUnit 5. [Note: adapt from exercise 1]
Run the test programmatically (as a JUnit 5 test).

C) Refactor to use Selenium extensions for JUnit 5
JUnit 5 allows the use of extensions which may provide annotation for dependency injection (as seen previously for Mockito). This is usually a more compact and convenient approach.

The Selenium-Jupiter extension provides convenient defaults and dependencies resolution to run Selenium tests (WebDriver) on JUnit 5 engine.
Note that this library will ensure several tasks:

- enable dependency injection with respect to the WebDriver implementation (automates the use of WebDriverManager to resolve the specific browser implementation). You do not need to pre-install the WebDriver binaries; they are retrieved on demand.
- if using dependency injection, it will also ensure that the WebDriver is initialized and closed.

Refactor your project to use the Selenium-Jupiter extension (@ExtendWith(SeleniumJupiter.class; use a dependency injection to get a "browser" instance; no explicit "quit").

## 4.3 Page Object pattern

Consider the example discussed here (or, for a more in depth discussion, here).
Note: the target web site implementation may have changed from the time the article was written and the example may require some adaptations to run (i.e., pass the tests). However, it is not mandatory to have the example running.
"Page Object model is an **object design pattern** in Selenium, where webpages are represented as classes, and the various elements [of interest] on the page are defined as variables on the class. All possible user interactions [on a page] can then be implemented as methods on the class."

A) Implement the "Page object" design pattern for a cleaner and more readable test using the same application problem from exercise 4.2. In a new project, refactor the previous implementation to use the Selenium-Jupiter extension and Page Object pattern.

## 4.4 Browser variations

Consider using a browser that is not installed in your system. You may resort to a Docker image very easily (see Docker browsers section).
Note that, in this case, the WebDriver will connect to a remote browser (no longer direct communication) and you should Docker installed in your system.

# Lab 5 Behavior-driven development (Cucumber)

## Context and key points

You may configure IntelliJ to offer extended support to run Cucumber.

### Key Points

- The Cucumber framework enables the concept of "executable specifications": with Cucumber we use **examples** to specify what we want the software to do. The (feature) **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.

### Explore

- Cucumber school: guided exercises, video lessons,…
- BDD with Cucumber – video presentation.

## 5.1 Getting started

a) Get the "Java Skelton" (maven) project proposed by the Cucumber team.
Remember to make the necessary adjustments:
- change the Artifact group and Id
- review the compiler version in the POM (1.8 → 11 or 17)

Note some elements I the POM:

| | |
|---|---|
| <dependencyManagement> with *-bom entries. | There are "bills of materials". Their use in the Dependency management section is useful to enforce the coherent use of the versions of related artifacts. Note that in the <dependencies> section, the related artifacts omit the version specification. |
| maven-compiler-plugin | Sometimes, to use more recent constructions, you may need to control the maven compiler version. |

b) Note the project structure, identify the location of the features and the tests.

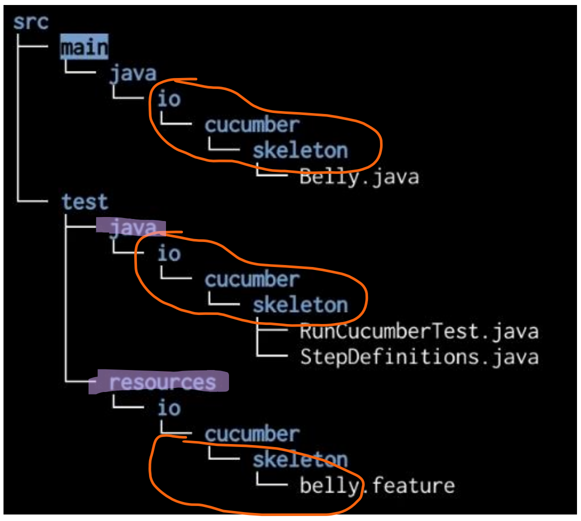
A feature will typical match a "test runner" class that will activate the test steps (in the same package).

The "test runner" class (that runs the test) should be annotated with:

@Suite

@IncludeEngines("cucumber")

@SelectClasspathResource("*my/package/structure*")



```
src
├── main
│   └── java
│       └── io
│           └── cucumber
│               └── skeleton
│                   └── Belly.java
└── test
    ├── java
    │   └── io
    │       └── cucumber
    │           └── skeleton
    │               ├── RunCucumberTest.java
    │               └── StepDefinitions.java
    └── resources
        └── io
            └── cucumber
                └── skeleton
                    └── belly.feature
```

c) Run the tests.

The tests will not pass because they still are incomplete.

Note the "clues" in the output, giving suggestions to implement the missing steps (i.e., test methods). Use these suggestions to create the missing steps. Note that, for this purpose, the "Belly" class implementation can just log information.

d) Extend your project to include also the Calculator example discussed in "Cucumber in a Nutshell" section in B. Garcia's book[4].

Note1: the example uses old-style integration with JUnit. You should adopt the strategy for the previous ["skeleton"] example. **No new imports or dependencies required**!

Note2: in this last example, the steps matching uses regular expressions. The best practice, however, is to use "cucumber expressions". **Be sure to "upgrade"** the sample code. E.g.:

| (old) regular expressions style → | (**better**) Cucumber expressions style |
|---|---|
| @When("^I add (\\d+) and (\\d+)$") | @When("I add {int} and {int}") |

## 5.2 Books

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 author, by category, etc).

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

Notes:

- The article is based on old-style Cucumber constructions. Ignore the dependencies and annotations suggested in the text: use the "Skeleton" exercise as a reference for Cucumber dependencies and test annotations.
- Write the features before the test steps. Steps can be partially generated from features.
- Prefer the "cucumber expressions" (instead of regular expressions) to write the steps definitions.
- To handle dates, consider using a ParameterType configuration. This defines a new custom parameter type to use in the matching expressions . [→ partial snippet]. The dates in the feature description need also to match the date mask used (**aaaa-mm-dd**).
- To handle data tables in the feature description (as in the Salary manager example), consider using a DataTable mapping and access you data as a *list of maps* (use headings in the data).

[4] solution code available, if needed.

## 5.3 Web automation

Cucumber id often used with Selenium WebDriver to write expressive automation tests.

Consider the example available in B. Garcia's repository concerning the integration of Cucumber and Selenium [junit5-cucumber-selenium] and implement it.
Adapt from this example to create a test scenario related to the "BlazeDemo" application (recall Lab 4, exercise 2).

# Lab 6 Static Code analysis (with SonarQube)

## Context and key points

### Key Points

Static code quality can assess a code base to produce quality metrics. These metrics are based on the occurrence of known "bed smells" and weaknesses. In this kind of analysis, the solution is not deployed, nor the code is executed (thus the name static analysis).
Static code analysis can be run locally using a "linter" and modern IDE will typically include support for code inspection. But it would be even more import to implement code analysis in the team develop infrastructure, i.e., at the continuous integration pipeline, using specialized services.
Key code quality 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 needs tools to obtain realistic information on the technical debt.

### Explore

— public projects on Sonar cloud that you can browse and learn.

## 6.1 Local analysis

a) Copy a Maven-based, Java application project to use (with tests passing). You may reuse one from previous labs, for example, the Euromillions from Lab 1.2.

b) Prepare a local instance of SonarQube server (using the Docker image). For this lab, you do not need to configure a production database (an embedded database is used by default; for a production scenario, a more demanding configuration is required).

   Note1: you may get a **conflict on port 9000** in your host, as it is also commonly picked for other services (e.g.: Portainer); pick another, if needed.

   Note2: there were known problems to run the provided docker image on Mac M1 (Apple silicon). Consider the following alternatives: use the .zip file version; use an alternative docker image.

c) Confirm that you can access the Sonar dashboard (default : http://127.0.0.1:9000) and change the default credentials (admin / admin).

d) Complete the "Analyzing a Project steps": create a project "manually"; set for "local analysis"; generate a named token. **Take note of the generated user token**! You will need it later several times.

e) Include the Sonar Maven plugin in your POM.

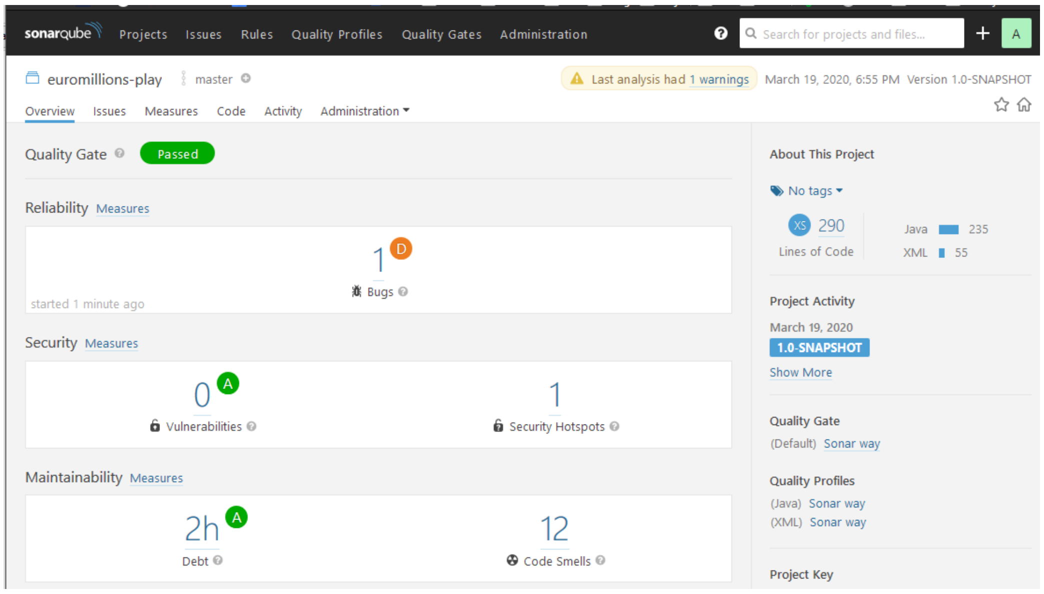
From the command line, run the code analysis (highlighted parts should be changed as needed):

```
$ mvn verify sonar:sonar -Dsonar.host.url=http://localhost:9009
-Dsonar.projectKey=lab6_1 -Dsonar.login=053bd3d423525e6df97b6bfd06b8a7ecd5bb7e
```

Note: optionally, you can save part of the Sonar configuration as "global settings".

f) Confirm that Sonar analysis was executed. Access the SonarQube dashboard (default : http://127.0.0.1:9000).

Has your project passed the defined quality gate? Elaborate your answer (in **Readme** document/markdown file, along with the code project).



sonarqube Projects Issues Rules Quality Profiles Quality Gates Administration

euromillions-play master — Last analysis had 1 warnings — March 19, 2020, 6:55 PM Version 1.0-SNAPSHOT

Overview Issues Measures Code Activity Administration

Quality Gate Passed

Reliability Measures — 1 D Bugs — started 1 minute ago

Security Measures — 0 A Vulnerabilities — 1 Security Hotspots

Maintainability Measures — 2h A Debt — 12 Code Smells

About This Project — No tags — XS 290 Lines of Code — Java 235, XML 55

Project Activity — March 19, 2020 — 1.0-SNAPSHOT — Show More

Quality Gate — (Default) Sonar way

Quality Profiles — (Java) Sonar way — (XML) Sonar way

Project Key

g) Explore the analysis results and complete with a few sample issues, as applicable. (Place your response in a **Readme** file, either html/md/pdf…).

| **Issue** | **Problem description** | **How to solve** |
|---|---|---|
| Bug | … | … |
| Vulnerability | | |
| Code smell (major) | | |

## 6.2 Technical debt (Cars)

Make a copy of the "Cars" project from Lab #3.2.
Be sure you are using a project with JUnit **tests implemented and passing**.

a) Analyze this project with SonarQube. Remember to create a new project in your Sonar instance and get a token.

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

Document the analysis findings with a screenshot (of the sonar dashboard for this project).

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

   Note: if you used the Entity data type as parameter in the API methods, you will likely get the vulnerability "Persistent entities should not be used as arguments".

c) Code coverage reports requires the Jacoco plugin. Be sure to use a project with unit tests and configured code coverage (e.g.: add the jacoco plugin to maven and update the plugin version).

d) Run the static analysis and observe/explore the coverage values on the SonarQube dashboard.

   How many lines are "not covered"? And how many conditions?

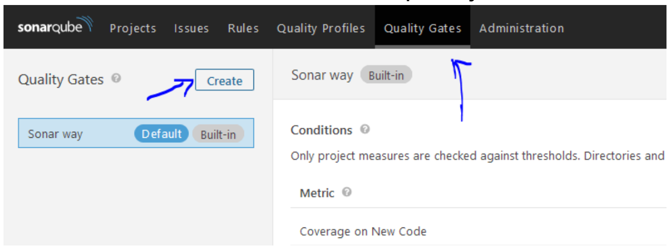
## 6.3 Custom QG

For this exercise, it would appropriate to use **a larger project**. Consider using your (group) project from IES[5]. Alternatively, you may get an open-source project (maven-based, Java project). Otherwise, continue with the project from previous exercise.

Note: **do not** to submit this project to your TQS personal Git repo! Focus on providing evidence that you complete the tasks and discuss the outcomes in a Readme.md file.

a) If possible, collaborate with other colleagues to define a custom quality gate to this project (especially if you are using the IES project, try to work with the former IES team…).

Feel free to mix the metrics but explain your chosen configuration.



Only project measures are checked against thresholds. Directories and

b) Add an increment to the source code. You may try to introduce some "bad smells"; in fact, try to break the quality gate.

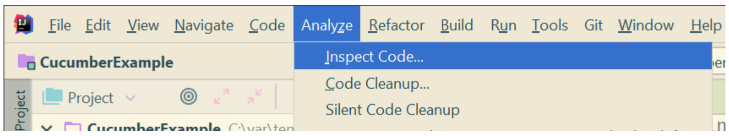
   Run the analysis and analyze the results.

## 6.4 IDE

[Optional. No submission required.]
In the previous tasks, we assume the static code analysis as a service, likely to be integrated at the Continuous Integration pipeline. You will also benefit from having code inspection as you develop, automatically integrated in the IDE.

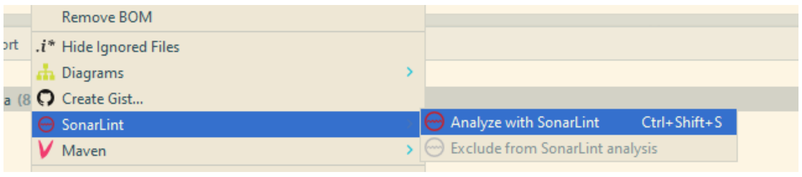
Note that:

a) IntelliJ (and most IDE) already integrates a code inspection tool. If you have not used before, try it.



b) The Sonar Qube analysis can be integrated in IntelliJ through the SonarLint plug-in.

[5] More specifically, the backend/API subproject, if applicable.


Remove BOM
Hide Ignored Files
Diagrams
Create Gist...
SonarLint
Maven
Analyze with SonarLint Ctrl+Shift+S
Exclude from SonarLint analysis