deti universidade de aveiro departamento de eletrónica, telecomunicações e informática

TQS Lab activities

v2024-03-20

Introductory notes and setup

Work submission

You should create a personal (git) repository for your TQS **individual portfolio** into which you will be pushing your solutions for the labs, named ***TQS_1235678*** (the number being your student number). Keep a **clean organization** that maps the exercises structure, e.g.: 🗁lab1/lab1_1; 🗁lab1/lab1_2; 🗁lab2/lab2_1; ...

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 (Long-term support JDK recommended; 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 5 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-scoped, 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.

Useful resources

- 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
- JetBrains Blog on Writing JUnit 5 tests (with vídeo).

1.1 Simple Stack contract

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

a) Create a new project (**maven project** for a Java standard application).

Add the required dependencies to run JUnit 5 tests.

- Note the elements: **junit-jupiter** and **maven-surefire-plugin** in the side illustration.
- You may adapt from a starter project for Maven.

```
<dependencies>
    <!-- ... -->
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter</artifactId>
        <version>5.10.2</version> <!-- can be omitted when using the BOM -->
        <scope>test</scope>
    </dependency>
    <!-- ... -->
</dependencies>
<build>
    <plugins>
        <plugin>
            <artifactId>maven-surefire-plugin</artifactId>
            <version>3.1.2</version>
        </plugin>
        <plugin>
            <artifactId>maven-failsafe-plugin</artifactId>
            <version>3.1.2</version>
        </plugin>
    </plugins>
</build>
```

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

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

See also: basic examples.

d) 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!]

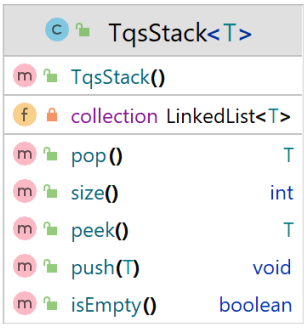
e) Your tests will verify several assertions that should evaluate to true for the test to pass.

f) 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).

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

h) Run the unit tests.

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



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[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]

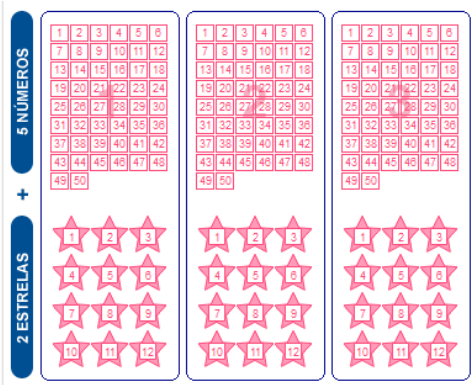
h) Peeking into an empty stack does throw a NoSuchElementException

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

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

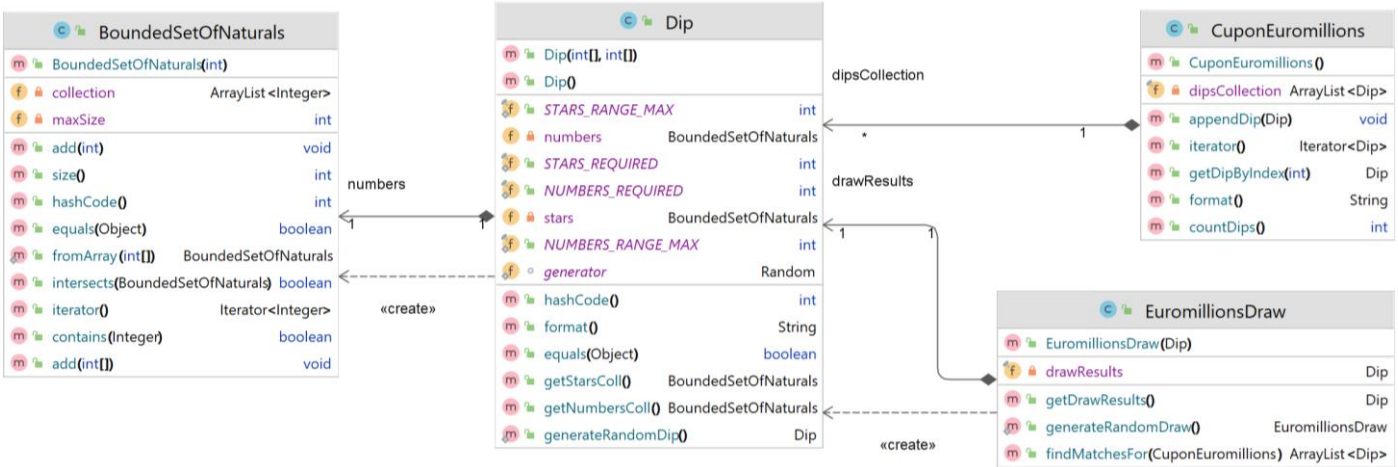
1.2 EuroMillions

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 no duplicates allowed (it is a Set) only natural numbers in the range [1, +∞]. the max size of the set (count of elements) is bounded to a limit 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. 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.
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

Problem/Symptom	Solution
Tests run from the IDE but not from command line	Be sure to configure the Surefire plug-in in Maven to a recent version (example).
My project's pom.xml is a mess! It is too long and uses old artifacts…	Check this POM and adapt (loggers are optional). Delete everything else.

Lab 2 Mocking dependencies (with Mockito)

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.

Useful resources

- There is a short course 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#totalValue()** 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#totalValue().**
Given that test should have predictable results, you need to address the problem of having non-deterministic answers from the **IStockmarketService** interface.

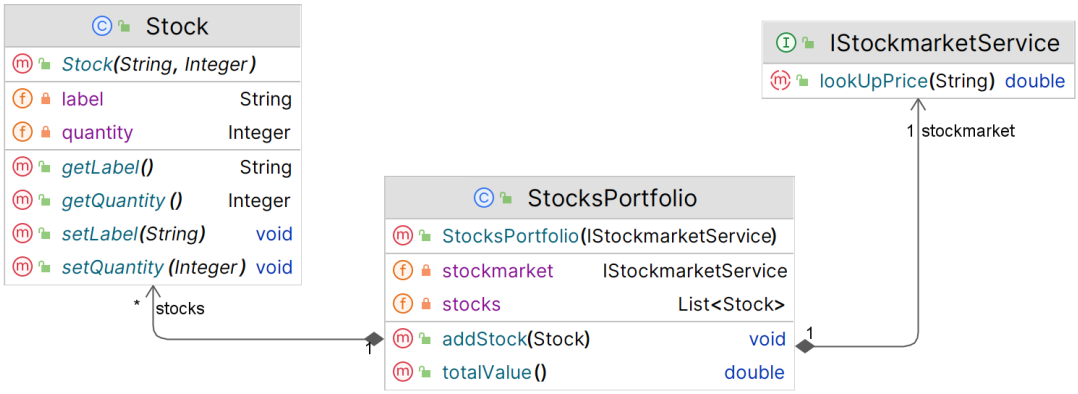


Figure 1: Classes for the StocksPortfolio use case.

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

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

1. *Prepare a mock to substitute the remote stockmarket 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 another assertions library. Consider using 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[2]).
Let us 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.

[2] To use the MapQuest API you need a valid developer API key.

b) Consider that we want to test the behavior of AddressResolverService#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 AddressResolverService#findAddressForLocation (using mocks where required).

e) Besides the "success" case, consider also testing for alternatives (e.g.: bad coordinates;...).

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 the failsafe plugin)

Consider you are implementing an **integration test** for the previous example; in this case, you would use the real implementations of the modules in the test, not the mocks.
(This section can be included in the same project as the previous one.)

Create a test class (AddressResolver**IT**), in a separate test package (be sure its name ends with "**IT**").
Copy the tests from the previous exercise into this new test class and 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 preparation
This lab is based in the Spring Boot (SB) framework. Most of students already used SB (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 the whole application context; a better practice(faster) is to focus your test and to limit application contexts only to a set of Spring components that participate in the 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. Note that Beans used by the controller under test will need to be mocked, since they will not be loaded by @WebMvcTest.
- 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

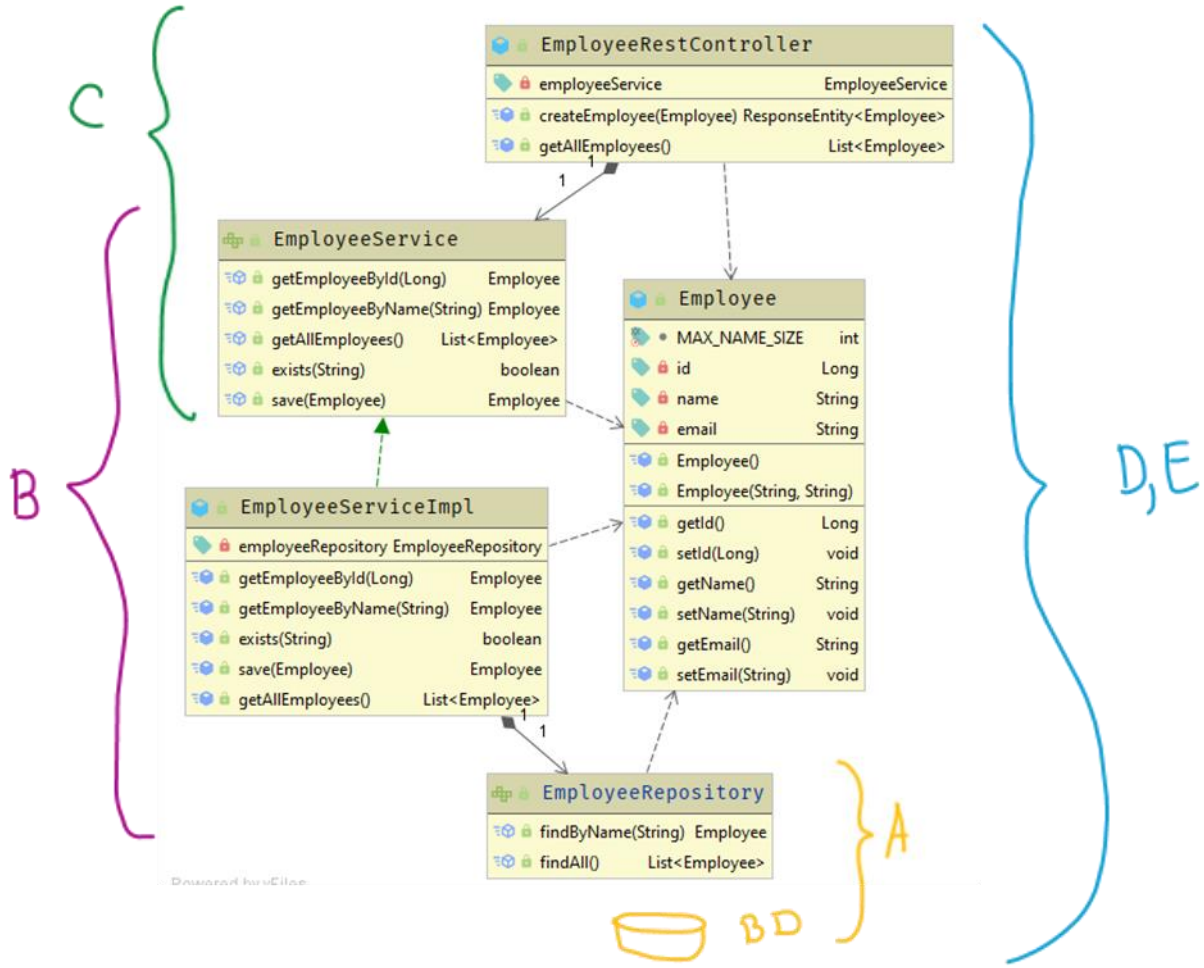
- Talk on Spring Boot tests (by Pivotal): https://www.youtube.com/watch?v=Wpz6b8ZEgcU

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” queries 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.



Find and study the following test scenarios (in the code provided):

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.

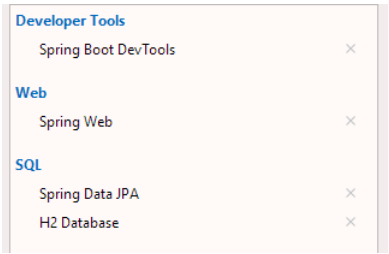
b) Identify an example in which you mock the behavior of the repository (and avoid involving a database).

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



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[3] (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 you 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. [Suggested strategy: C][4]

b) Create a test to verify the CarService (and mock the CarRepository). [Suggested strategy: B]

[3] To use Spring Boot v3.2+, you need Java v17+.
[4] Reference to the previous exercise (Employees project).

c) Create a test to verify the CarRepository persistence[5]. Be sure to include an in-memory database dependency in the POM (e.g.: H2). [Suggested strategy: A]

d) Having all the previous tests passing, implement an integration test to verify the API. [Suggested strategy: E]

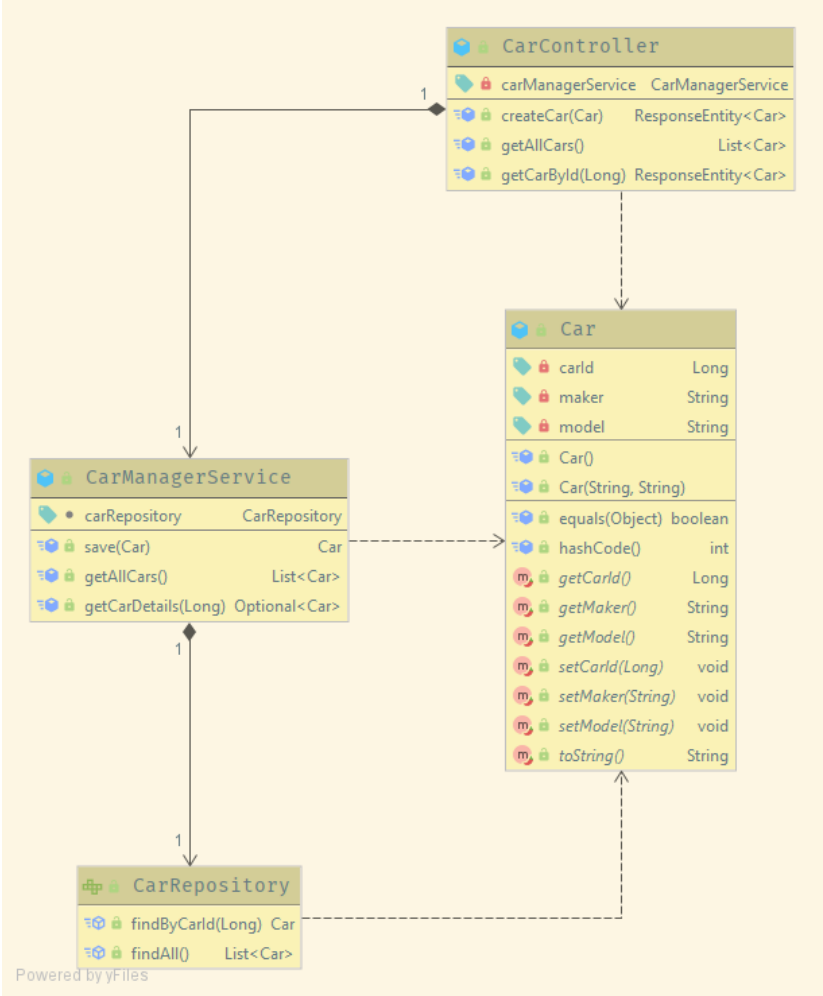


Figure 3- Classes for a Car management API.

e) [**Optional**] At this point, you should have a trivial Spring Boot application, in which the Service just delegates to repository. In a real application, we would expect the "services" to implement more complex, interesting, and mission-critical business logic.

Add a new feature (or more…) to your API that requires some handling of car-related business logic (e.g.: car selling, car renting, carpooling…). Ideally, you would have to implement some algorithm in the service. For example, "find a car that provides a *suitable replacement* for some given car [e.g.: as a courtesy car]", "create a credit plan for a given model, providing the monthly instalments list".

Be sure to add appropriate tests.

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 (use a Docker container)

[5] This will feel somewhat forced, given the Repository is quite simple. Testing the data access layer is more relevant when complex queries are involved.

- 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)

Key Points

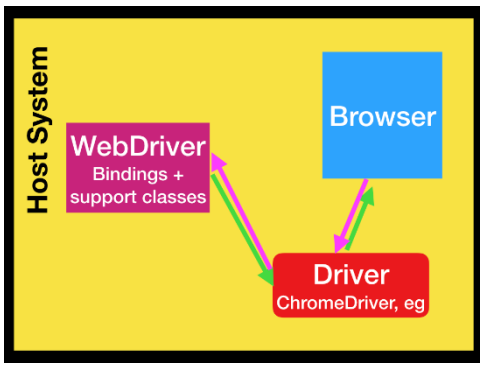
- Acceptance tests exercise the user interface of the system as if a real user was using the application. The system is treated as a black box.
- Browser automation (control the browser interaction from a script) is an essential step to implement acceptance tests on web applications. There are several 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 "hello world"

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

Adapt to a custom scenario (choose another web site,...).

Main dependencies for POM.xml:

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

b) refactor the previous example to use the approach in "Example 2-2" (which uses dependency injection to get the WebDriver instance).

4.2 Selenium IDE recorder

Often 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.
You need to 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
Be sure to use the Selenium-Jupiter extension. Note that this library will ensure several tasks:

- enable dependency injection with respect to the WebDriver implementation (i.e., hides 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 : (1) @ExtendWith(SeleniumJupiter.class); (2) use a dependency injection to get a "browser" instance; (3) no explicit "quit". You may check this related example (consider only for the parts related to the WebDriver...).

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., to 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. Suggestion: use the Page Factory approach.

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 3.4 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)

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.
- BDD plays well with user stories: the user story can be used as a unit of specification, work assignment, acceptance, and delivery. In this sense, we can have a better traceability from business goals (requirements) to code.

Explore

- You may configure IntelliJ to offer extended support to run Cucumber.
- Cucumber school: guided exercises, video lessons... Especially: Cumber for Java developers.
- BDD with Cucumber – video hands-on discussing Cumber/Junit examples.

5.1 Getting started (calculator)

a) Consider that you are implementing a Reverse Polish Notation (RPN) calculator module and, naturally, you want to supply some tests.

This example is discussed in "Cucumber in a Nutshell" section, in B. Garcia's book[6].

Start a (regular) Java project with the appropriate POM settings to run Cucumber.

b) Crete a .feature file (a regular text file with .feature extension) to describe the intended use of the calculator (related example).

Note that the *.feature* file must go into the tests resources folder, e.g.:

./src/**test/resources**/mypackage/calculator_ops.*feature*

c) Be sure to implement the required test steps. You will need two test files: one just to activate Cucumber and another with the test steps implementation (related example).

d) Run the tests. (or, better, run the feature):

- place a tag on top of your feature file (example @calc_sample), then
- mvn test -Dcucumber.filter.tags='@calc_sample'

Note: in the example from Boni García book, step 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}")

e) Add a few more test scenarios in your feature (e.g.: multiply, invalid operation...)

Now, if you try to build the project, the tests will not pass because they still are incomplete.

[6] solution code available, if needed.

Note the "clues" in the output, giving suggestions to implement the missing steps (i.e., test methods).

Implement the required test steps to have your feature specification satisfied.

5.2 Book search

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. [The Library "database" is just a regular class holding a collection of Books.]

Notes:

- Use the article to get an overall description of the use case. Concerning the implementation, however, you should try to use modern constructs. E.g.: ignore the POM from the article; consider what you have used in the previous exercise.
- 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 matching, consider using a ParameterType configuration. This defines a new custom parameter type to use in the matching expressions. [→ partial snippet].

@When("the customer searches for books published between {iso8601Date} and {iso8601Date}")

The dates in the feature description need also to match the date mask defined in the ParameterType (**aaaa-mm-dd**).

- To handle data tables in the feature description (e.g.: a "database" of books), consider using a DataTable mapping and access you data as a *list of maps* (use headings in the data).

5.3 Web automation with Cucumber

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)

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" (usually integrated in the IDE) but it would be even more import to implement code analysis in the team development infrastructure, i.e., at the continuous integration pipeline, using specialized services.
Key code quality measures include occurrences of problematic code ("bugs" in logic/flow), vulnerabilities (security/reliability concerns), code smells (bad/poor practice or coding style); coverage (ratio tested/total); and code complexity.

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.
- Webinar on static code analysis (promoted by Codacy)
- SonarQube analysis can be integrated in several IDE (IntelliJ, VS Code,...) through the SonarLint plug-in.

6.1 Local analysis

a) Copy a Maven-based, Java application project to use (with some tests). 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; but 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.

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

c) Complete the initial configuration to create a local project.

Select the "Analysis method" to "locally;then generate a named token. **Take note of the generated user token**! You will need it later several times.

d) Lock the Sonar Maven plugin version in your POM.

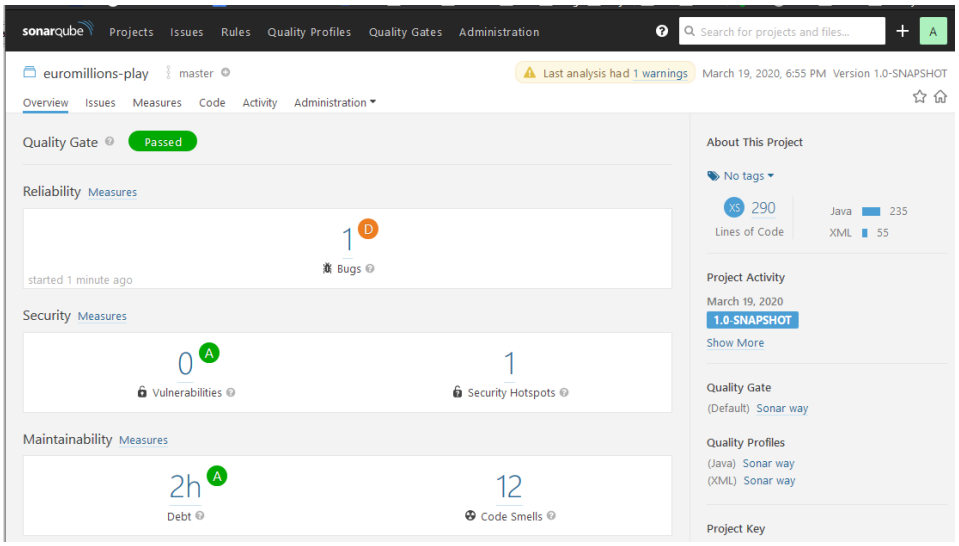
From the command line, run the code analysis (highlighted parts should be changed as needed):

```
$ mvn verify sonar:sonar -Dsonar.host.url=http://localhost:9000
-Dsonar.projectKey=lab6_1 -Dsonar.login=053bd3d423525e6df97b6bfd06b8a7ecd5bb7e
```

Note: alternatively, you can save part of the Sonar configuration as "global settings", shortning the required maven command.

e) 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 (prepare a **Readme** document/markdown file, along with the code project).



f) 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).

   Run the static analysis and discuss the coverage values on the SonarQube dashboard (how many lines are "not covered"? And how many conditions? Are the values good?...)

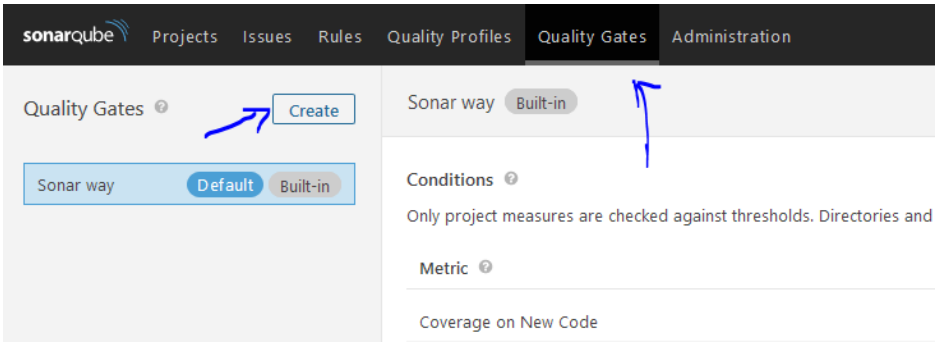
6.3 Custom Quality-Gate

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! Work in a separate area and just provide 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.



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.