# SOFTWARE ARCHITECTURE DESIGN

Requirement G26-T7 Documentation

Pedro Zahonero Mangas
Pablo García Fernández
Manuel Guerrero García

## Index

## Introduction

The Gamification Game project includes a key requirement of implementing a test compiler/executor functionality. This feature allows users to submit their own test cases written in Java and have them compiled and executed by the game engine. The functionality receives two text files as input (the class to test and the test class) and launches the compiler and the test executor. The output returned by the compiler and executor is processed to extract relevant information for the purposes of the game.

To implement this functionality, the project uses Java as the backend language, JUnit as the testing framework, and Maven for build automation. The Java Compiler API is used to compile the submitted Java files, and JUnit is used to execute the compiled test cases.

The outcome of the test execution is processed to provide feedback to the user and to track their progress. For example, the presence of compilation errors is identified and reported to the user, and the coverage of the code is calculated and displayed to the user. This functionality is a critical component of the game, as it allows users to practice writing their own test cases and to receive immediate feedback on their code.

The project documentation provides a detailed overview of the test compiler/executor functionality, including its requirements, design, and implementation details. The documentation serves as a valuable reference for developers and stakeholders involved in the project and ensures that the feature is implemented correctly and effectively.

## Scrum

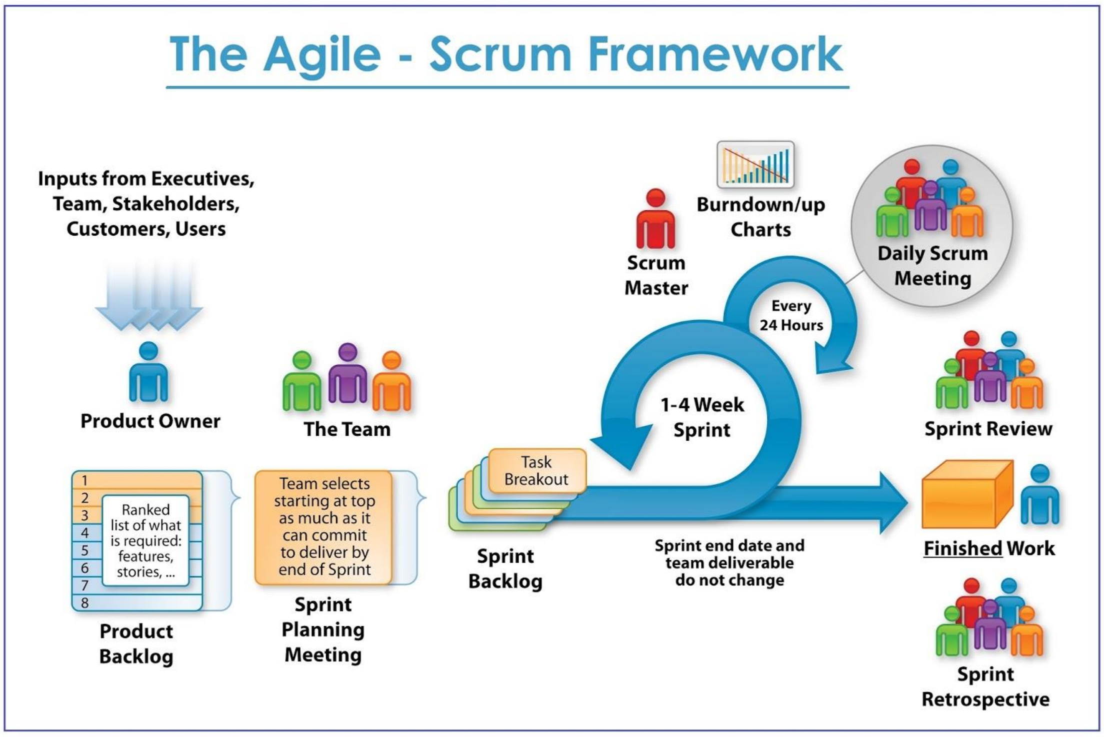
Scrum is an agile framework commonly used in software development projects to manage and deliver products efficiently. It promotes collaboration, flexibility, and iterative development. Here's an explanation of how we use Scrum in our software application development process:

- **Scrum Team**: We have a cross-functional team consisting of different roles, including a Product Owner, Scrum Master, and Development Team members.
- **Product Owner**: The Product Owner represents the stakeholders and ensures that the product meets their requirements. They prioritize and manage the product backlog.
- **Scrum Master**: The Scrum Master is responsible for facilitating the Scrum process, removing obstacles, and ensuring the team adheres to Scrum principles and practices.
- **Development Team**: The Development Team is a self-organizing group responsible for delivering the product increment. They estimate, plan, and execute the work required for each sprint.
- **Product Backlog**: The Product Backlog is a prioritized list of user stories, enhancements, and bugs. The Product Owner maintains this backlog, constantly refining and reprioritizing it based on feedback and changing requirements.
- **Sprint**: A Sprint is a time-boxed iteration where the Development Team works to deliver a potentially shippable product increment. Our sprints typically last two weeks, but the duration can vary based on the project's needs.
- **Sprint Planning**: At the beginning of each sprint, the Scrum Team holds a Sprint Planning meeting. The Product Owner discusses the highest-priority items from the Product

Backlog, and the Development Team determines what they can accomplish during the sprint.

- **Daily Scrum**: The Daily Scrum is a short daily meeting where the Development Team synchronizes its activities. Each team member shares their progress, plans for the day, and any obstacles they're facing. The Scrum Master ensures the meeting remains focused and time boxed.
- **Sprint Review**: At the end of each sprint, we conduct a Sprint Review meeting. The Development Team demonstrates the completed work to stakeholders, and the Product Owner provides feedback. This session helps us gather valuable insights and make necessary adjustments.
- **Sprint Retrospective**: Following the Sprint Review, we hold a Sprint Retrospective meeting. The Scrum Team reflects on the sprint, identifies areas for improvement, and discusses actionable steps to enhance the process in future sprints.
- **Increment**: The outcome of each sprint is a potentially shippable product increment. It should be in a usable state and adhere to the Definition of Done agreed upon by the team.
- **Continuous Improvement**: Scrum encourages continuous improvement. We regularly review and refine our processes, seeking ways to enhance productivity, quality, and customer satisfaction.

By utilizing Scrum, we aim to foster collaboration, transparency, and flexibility in our software development process. It helps us deliver value incrementally, respond to changing requirements, and improve the overall efficiency and effectiveness of our team.


The Agile - Scrum Framework
Inputs from Executives, Team, Stakeholders, Customers, Users
Product Owner
The Team
Scrum Master
Burndown/up Charts
Daily Scrum Meeting
Every 24 Hours
1-4 Week Sprint
Sprint Review
Task Breakout
1
2
3
4
5
6
7
8
Ranked list of what is required: features, stories, ...
Team selects starting at top as much as it can commit to deliver by end of Sprint
Sprint Backlog
Sprint end date and team deliverable do not change
Finished Work
Product Backlog
Sprint Planning Meeting
Sprint Retrospective

## Requirements of the project

- The application should allow to the user to compile their code and receive feedback of the result of the compilation.
- The application should allow to execute the code and get the results of the test.
- The application will show the coverage of the execution of the test.
- The application will show you the error of compilation if it occurs.
- The execution of the code will be not allowed if there is no compilation or is a compilation error.

## Glossary

- **Application**: The software program or system being developed.
- **User**: The individual or entity interacting with the application.
- **Compile**: The process of translating the source code into machine-readable instructions.
- **Code**: The set of instructions or statements written in a programming language that the application processes.
- **Feedback**: Information or responses provided by the application to the user regarding the compilation or execution of the code.
- **Execute**: The action of running or carrying out the code within the application.
- **Test Results**: The output or outcome of executing the code against predefined tests, showing the success or failure of each test.
- **Coverage**: The extent or percentage of code that has been executed during the testing process.

## Diary of effort

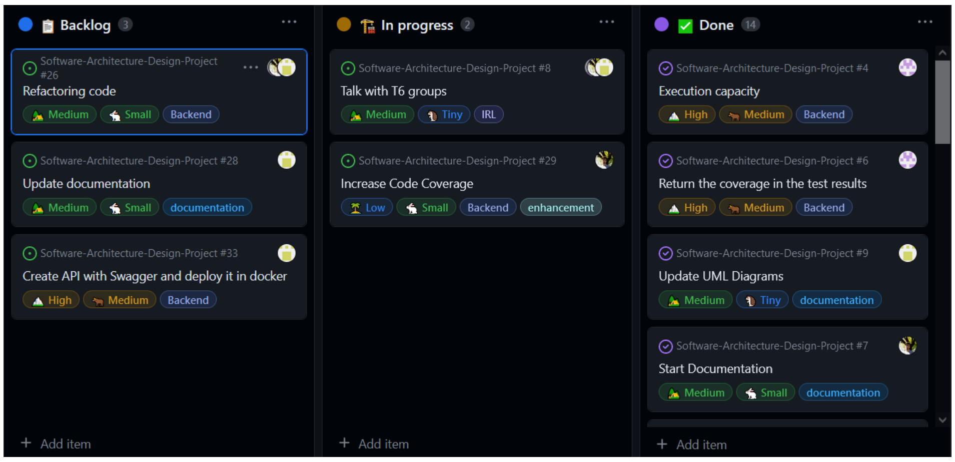
- 7 of April - 3h – Research on how to compile and execute code from txt.
- 9 of April - 2h – Research information.
- 10 of April - 1h – Organization and split tasks.
- 11 of April - 2.5h – UMLs.
- 11 of April - 1h – Prototype Design.
- 13 of April - 0.5h – PowerPoint presentation.
- 15 of April - 1h – Programming.
- 16 of April - 3h – Programming.
- 17 of April - 3h – Programming & PowerPoint.
- 22 of April – 1.5h – Programming & Research.
- 25 of April – 2h – Programming & Research.
- 26 of April – 1h – UMLs.
- 2 of May – 2h – Programming.
- 6 of May – 1h – Documentation.
- 7 of May – 0.5 PowerPoint.

- 13 of May - 1h – Programming.
- 18 of May - 2h – Programming & Research.
- 24 of May – 1.5h – Programming.
- 25 of May – 1h – UMLs.
- 30 of May – 1.5h – Programming.
- 3 of June – 3h - Programming & Research.
- 5 of June – 3h - Programming & Research.
- 6 of June – 1h - Documentation.
- 6 of June – 0.5h – PowerPoint.
- 25 of June – 2h – Programming.
- 26 of June – 2h – Programming.
- 27 of June – 1h – Programming
- 29 of June – 0.5 – Programming
- 4 of July – 3h – Documentation
- 5 of July – 4h – Programming
- 6 of July – 1h – Programming

## Organization techniques

For the team organization we use the following techniques:

- **GitHub Projects (Kanban)**: We utilize GitHub Projects, specifically the Kanban feature, to organize and track our development tasks. Kanban boards provide a visual representation of our workflow, allowing us to create and manage different stages or columns that represent the various stages of our development process. Each task or user story is represented as a card that can be moved across the board as it progresses from one stage to another. This technique helps us visualize, know the difficulty and priority on what to work, collaborate effectively, and ensure transparency within the team.



Backlog 3

Software-Architecture-Design-Project #26
Refactoring code
Medium · Small · Backend

Software-Architecture-Design-Project #28
Update documentation
Medium · Small · documentation

Software-Architecture-Design-Project #33
Create API with Swagger and deploy it in docker
High · Medium · Backend

+ Add item

In progress 2

Software-Architecture-Design-Project #8
Talk with T6 groups
Medium · Tiny · IRL

Software-Architecture-Design-Project #29
Increase Code Coverage
Low · Small · Backend · enhancement

+ Add item

Done 14

Software-Architecture-Design-Project #4
Execution capacity
High · Medium · Backend

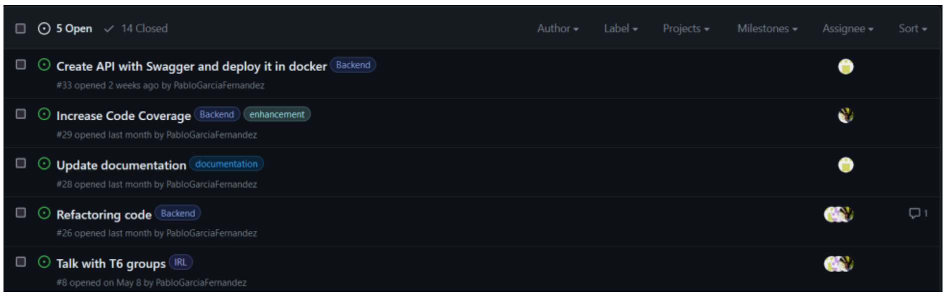
Software-Architecture-Design-Project #6
Return the coverage in the test results
High · Medium · Backend

Software-Architecture-Design-Project #9
Update UML Diagrams
Medium · Tiny · documentation

Software-Architecture-Design-Project #7
Start Documentation
Medium · Small · documentation

+ Add item

- **GitHub Issues**: GitHub Issues serve as a centralized platform for tracking and managing tasks, bugs, feature requests, and other project-related items. Issues allow us to create, assign, and track individual work items or tasks. We can provide detailed descriptions, assignees, labels, and milestones to each issue, facilitating clear communication and accountability within the team. Additionally, issues provide a thread of comments and discussions related to a specific task, making it easy to collaborate and keep track of progress.



5 Open ✓ 14 Closed

Create API with Swagger and deploy it in docker Backend
#33 opened 2 weeks ago by PabloGarciaFernandez

Increase Code Coverage Backend enhancement
#29 opened last month by PabloGarciaFernandez

Update documentation documentation
#28 opened last month by PabloGarciaFernandez

Refactoring code Backend
#26 opened last month by PabloGarciaFernandez
1

Talk with T6 groups IRL
#8 opened on May 8 by PabloGarciaFernandez

- **Discord**: Discord is a communication platform that we use for real-time collaboration, discussions, and quick communication within our team. We leverage Discord channels to facilitate ongoing conversations related to the project, including general discussions, brainstorming, feedback sharing, and informal updates. Discord allows us to communicate in both text and voice formats, making it a versatile tool for quick queries, team synchronization, and ad-hoc conversations.
- **WhatsApp**: WhatsApp serves as an additional communication channel that we use to facilitate quick and direct communication within the team. While Discord is primarily used for project-related discussions, WhatsApp provides a convenient platform for instant messaging, informal updates, and notifications. We use WhatsApp to share urgent or time-sensitive information, coordinate activities, and stay connected when a more immediate response is required.

By utilizing these organization techniques, we aim to streamline our project management, enhance collaboration, and maintain effective communication within the team. These tools enable us to organize and track tasks efficiently, discuss ideas and issues in real-time, and ensure everyone is aligned and working towards the common goals of the project.

## Technological background

The test compiler/executor functionality requires a set of tools and technologies to implement, including:

- **Java**: The backend language used for the development of the project.
- **JUnit**: The testing framework used for executing unit tests.
- **Maven**: The build automation tool used for the project. Maven is a widely used build automation tool that simplifies the process of building and deploying Java applications.
- **GitHub**: The version control system and platform used for managing the source code. GitHub is a web-based platform for hosting and sharing Git repositories. Here is where we hold the Sprint Backlog in the project part.

- **Jacoco**: Jacoco is a code coverage tool that can be used to measure the amount of code that is covered by tests. It is a valuable tool for the Gamification Game project as it helps to ensure that the test compiler/executor functionality is robust and provides reliable feedback to users on the quality of their tests.
- **Spring Framework**: Spring MVC is a module of the Spring Framework that provides a robust and flexible framework for building web applications. It allows to separate the concerns of the application into different components, including the presentation layer (view), business logic (controller), and data (model).
- **Docker**: Docker is a containerization platform that allows applications to be packaged along with their dependencies into containers.
- **Swagger**: Swagger is an open-source framework used to design, build, document, and consume RESTful APIs. It provides a set of tools and specifications that enable developers to define the structure and behavior of APIs.

In addition to these technologies, the test compiler/executor functionality also requires a solid understanding of software engineering principles, particularly in the areas of software testing and automation. This includes knowledge of software design patterns, code quality and maintainability, and best practices for software testing and debugging.

## Functionality and technological characteristics

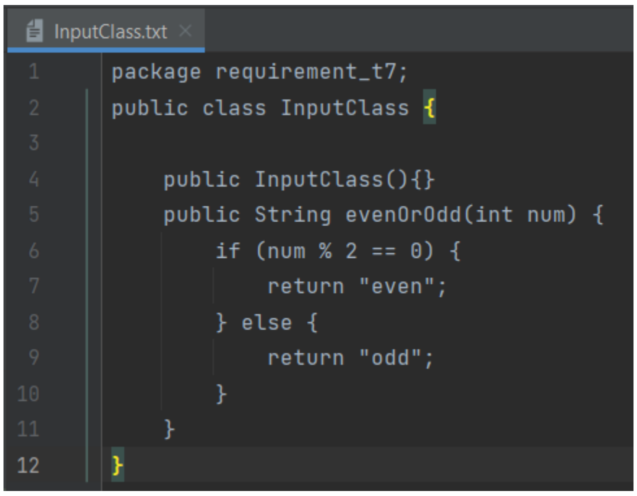
### Example case

**Test Case Tittle**: Compile and execute the given .txt files.

**Context**: We have a Gamification application, and this part of the application receives a file with the code of the class to be tested and the test class code.

**Inputs**:

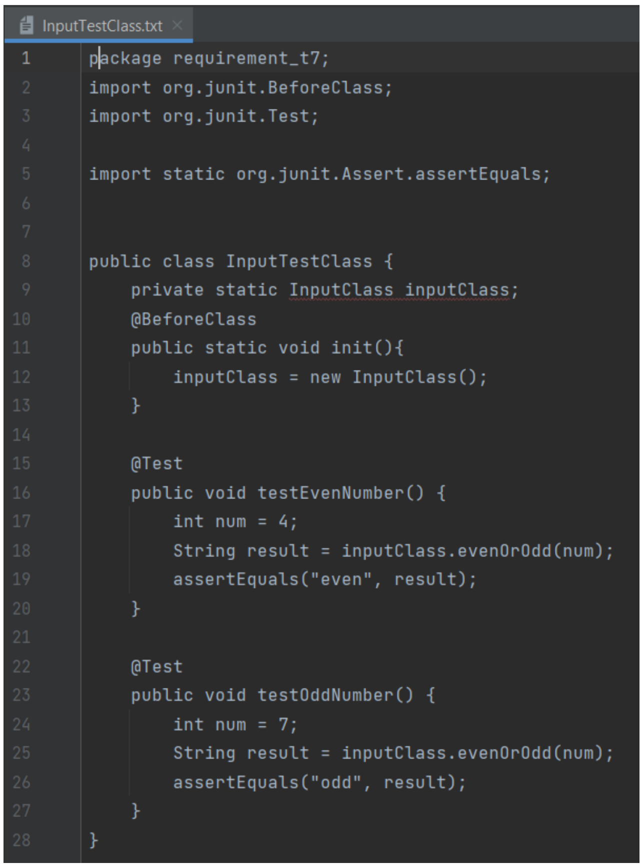
- InputClass.txt:



InputClass.txt

```
package requirement_t7;
public class InputClass {

    public InputClass(){}
    public String evenOrOdd(int num) {
        if (num % 2 == 0) {
            return "even";
        } else {
            return "odd";
        }
    }
}
```

- InputTestClass.txt:


InputTestClass.txt

```
package requirement_t7;
import org.junit.BeforeClass;
import org.junit.Test;

import static org.junit.Assert.assertEquals;


public class InputTestClass {
    private static InputClass inputClass;
    @BeforeClass
    public static void init(){
        inputClass = new InputClass();
    }

    @Test
    public void testEvenNumber() {
        int num = 4;
        String result = inputClass.evenOrOdd(num);
        assertEquals("even", result);
    }

    @Test
    public void testOddNumber() {
        int num = 7;
        String result = inputClass.evenOrOdd(num);
        assertEquals("odd", result);
    }
}
```

**Expected Output**: In this case both files should compile and be able to be executed.

- Compile: Compiled
- Execute: Tests run: 2, Failures: 0, Errors: 0, Skipped: 0 Total Lines: 4 Covered Lines: 4 Missed Lines: 0 Coverage Percentage: 100.0%

**Conclusion**: The code received had no errors so it could be compiled, and the execution showed that all tests were executed successfully, and the coverage is 100% of the InputClass.

## UML Diagrams

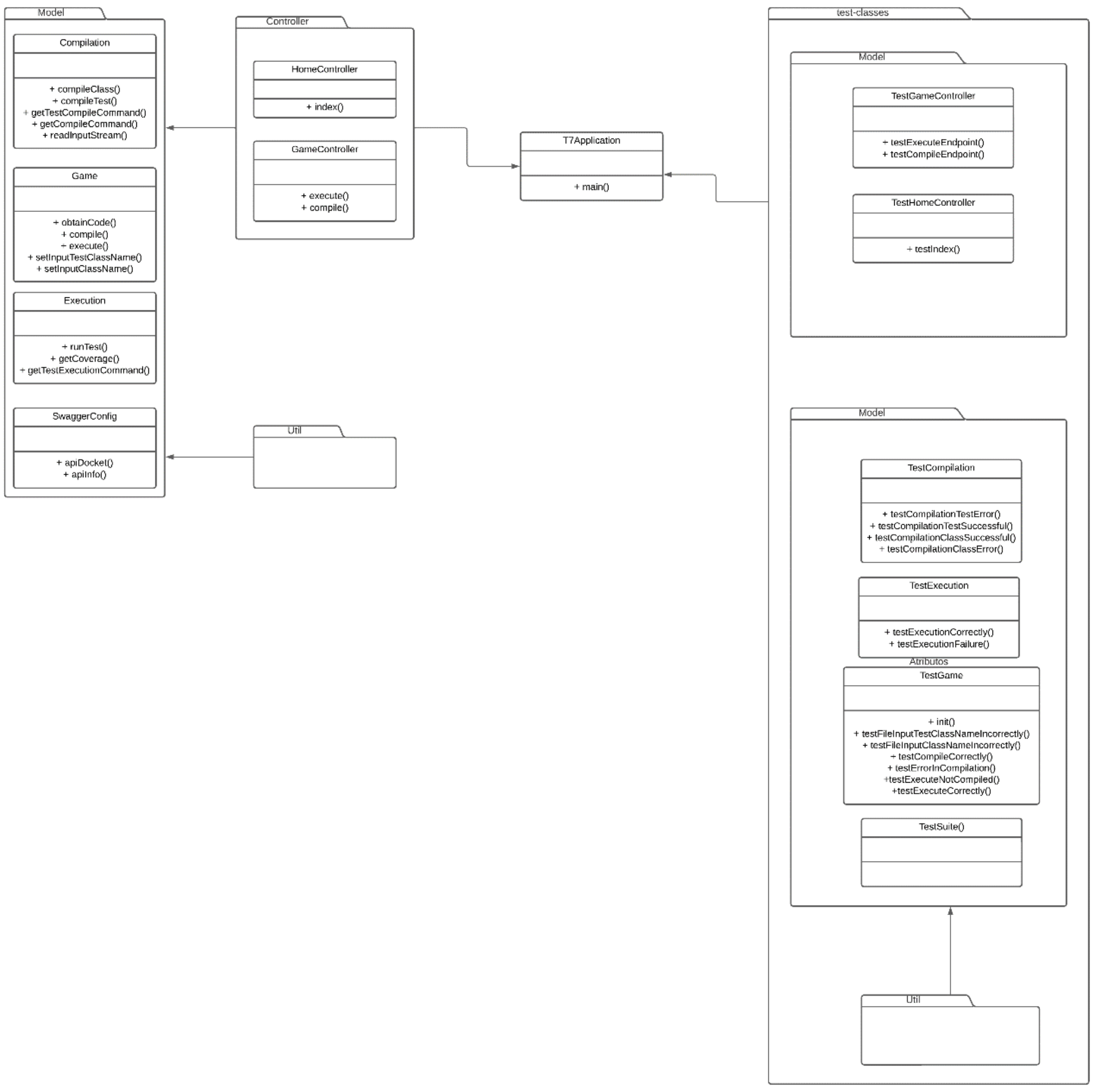
UML diagrams play a crucial role in the development of software projects, including the Gamification Game project. Since the project is built entirely in Java, UML diagrams are used to represent the various components of the project and illustrate their interactions with one another. These diagrams provide a visual representation of the project's software architecture, enabling developers and stakeholders to easily comprehend the relationships between different classes and components.

To represent the different aspects of the project's architecture, the Gamification Game project employs a variety of UML diagrams, including class diagrams, sequence diagrams, and use-case diagrams. These diagrams are indispensable tools for understanding the project's structure and design and provide a valuable resource for anyone who needs to understand the architecture of the project.

## UML Class diagram

UML class diagrams are an indispensable means of comprehending the structure and architecture of a software system. These diagrams furnish a graphic depiction of the system's design, enabling a clear understanding of the connections between various components and classes.

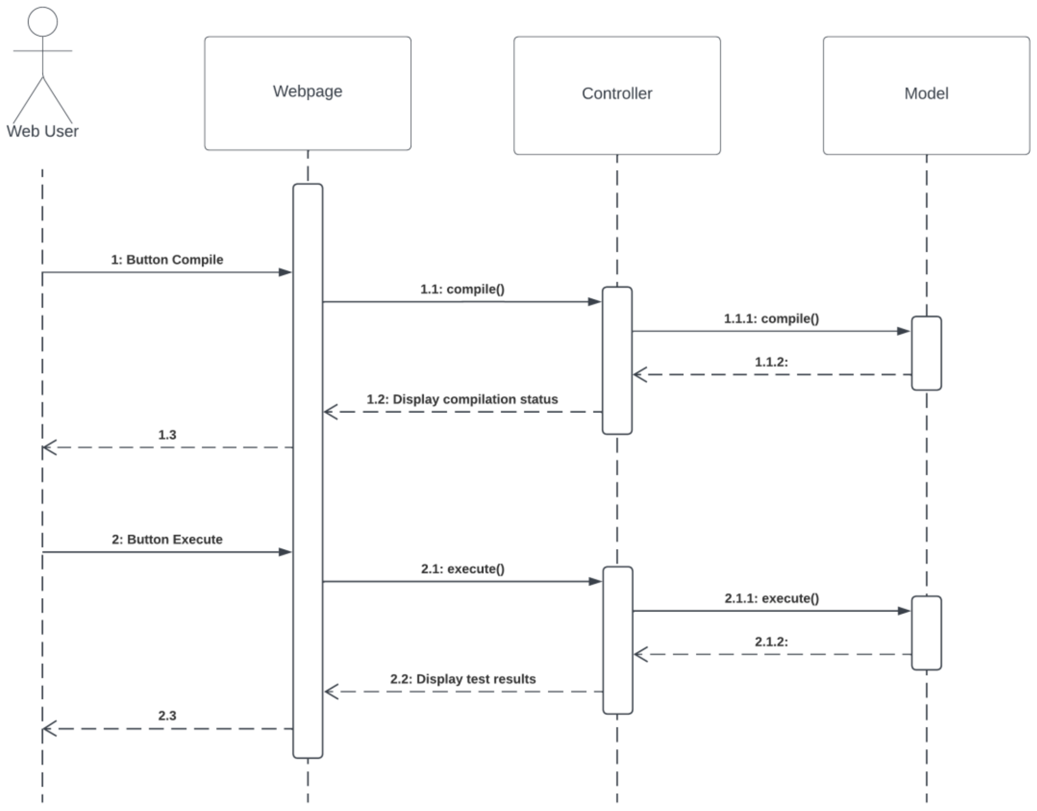
This is the UML class diagram for this project:



## UML Sequence diagram

A UML sequence diagram is a critical component of UML, used to represent the interactions between objects or components in a system over time. These diagrams offer a visual representation of the sequence of messages exchanged between objects, their chronological order, and the objects' lifecycle involved in the interactions.

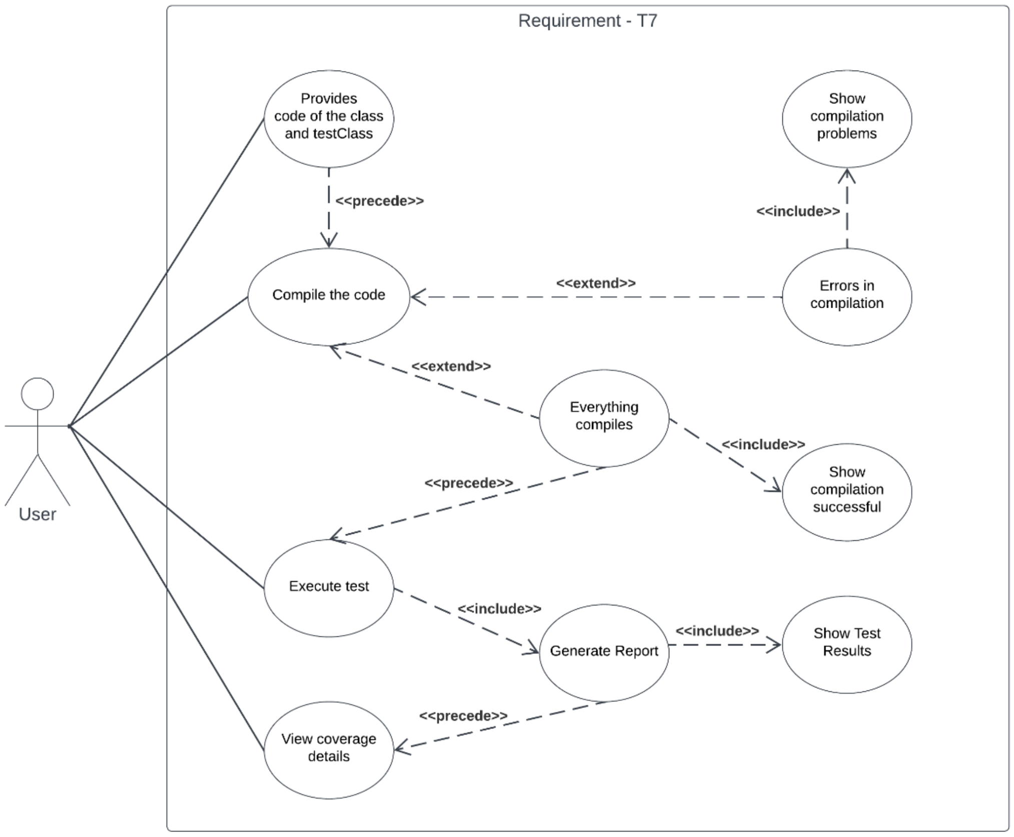
This is the UML sequence diagram of this project:



## UML Use-case diagram

A UML use-case diagram is a pivotal UML diagram used to represent the functions provided by a system or component. The diagram illustrates the various use cases, or distinct ways in which the system can be employed, and the actors, or external entities that interact with the system. Additionally, the diagram depicts the connections between use cases and actors, thereby providing a comprehensive representation of the system's functionality.

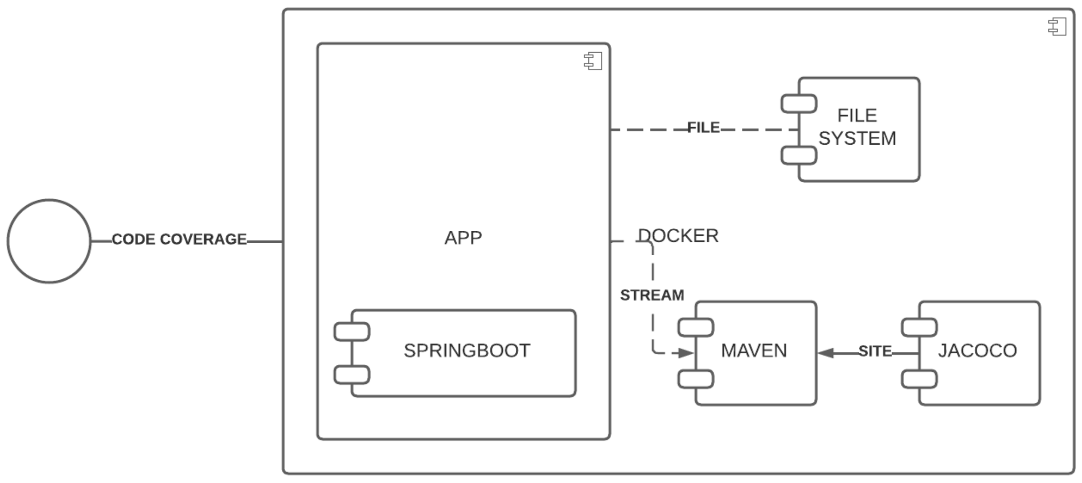
This is the UML use-case diagram of this project:



### UML Component diagram

A UML component diagram is a type of structural diagram that represents the high-level structure and relationships of the components within a system. It provides an overview of the system's architecture and shows how the various components interact and collaborate to fulfill the system's functionality.

This is the UML component diagram of this project:



## Code analysis

### Project main design pattern

MVC, which stands for Model-View-Controller, is a design pattern commonly used in web application development to separate concerns and improve code organization. It provides a structured approach to building applications by dividing them into three interconnected components: the model, the view, and the controller.

Model: The model represents the data and business logic of the application. It encapsulates the application's data structure, handles data validation and manipulation, and interacts with the database or any other data source. The model component does not depend on the other components and operates independently.

View: The view represents the user interface of the application. It is responsible for presenting the data to the user and receiving user input. Views are typically implemented using HTML, CSS, and JavaScript in web applications. They retrieve data from the model and format it for display to the user. Multiple views can be associated with a single model, allowing different ways of presenting the same data.

Controller: The controller acts as an intermediary between the model and the view components. It receives user input from the view, processes it, and updates the model accordingly. The controller handles user actions, such as button clicks or form submissions, and determines the appropriate actions to take. It updates the model based on the user's input and may trigger changes in the view to reflect the updated data.

The MVC pattern promotes separation of concerns, making it easier to maintain and modify code. It allows developers to work on different parts of the application independently. For example, a front-end developer can focus on the view without worrying about the underlying data or business logic, while a back-end developer can work on the model and controller without concerning themselves with the specific UI details.

In a web application, the MVC pattern is typically implemented with the following interactions:

- The user interacts with the view, triggering events or actions.
- The view notifies the controller about the user's actions.
- The controller updates the model based on the user's actions and business logic.
- The model updates its state and notifies the view of any changes.
- The view retrieves updated data from the model and updates the UI, accordingly, reflecting the changes to the user.

By following the MVC pattern, code organization becomes more modular, scalable, and maintainable. It enables easier testing and reusability of components, as each component has a specific role and responsibility within the application.

## Project class code analysis

### GameController.java

The GameController class is a RESTful controller that handles HTTP requests related to game execution and compilation. The class has two methods: execute() and compile().

- The execute() method executes a test class within the game and returns the result. It logs the execution process using a Logger class and utilizes a Game object to perform the execution.
- The compile() method compiles the game's classes. It accepts two parameters, the names of the input class and the input test class. It validates the input class names and returns error messages if they are invalid. Otherwise, it sets the input class names in the Game object and invokes the compile() method to compile the classes.

The class also includes a private method, isValidInputClassName(), which validates the input class name to ensure it does not contain certain characters.

Overall, the GameController class acts as a controller for handling HTTP requests related to game execution and compilation, utilizing the Game model object and a Logger utility for logging purposes.

### HomeController.java

It is annotated with **@Controller**, indicating that it serves as a controller component in a Spring MVC application.

The **HomeController** class has a single method **index()**, which is annotated with **@RequestMapping("/")**. This method handles requests to the root URL ("/") of the application.

When a request is made to the root URL, the **index()** method is invoked, and it returns the string "home". This indicates that it is mapping to a view called "home" which will be resolved by the view resolver configured in the application.

In summary, this controller class is responsible for handling requests to the root URL and returning the "home" view as the response.

### Compilation.java

This class handles the compilation tasks for Java classes and test classes.

- Static Methods:
    - compileClass(String inputClassName, String inputClassCode): This method takes the class name and code as parameters, creates a Java file from the provided code, compiles it, and returns the result of the compilation as a String. It uses a Logger to log the execution process and file-related utility classes.
    - compileTest(String inputTestClassName, String inputTestClassCode): This method is similar to compileClass(), but it handles the compilation of test classes. It takes the test class name and code as parameters, creates a test file, compiles it, and returns the compilation result. It also uses a Logger and file-related utility classes.
- Private Methods:
    - getCompileCommand(): This method determines the compile command based on the operating system and returns it as an array of strings.
    - getTestCompileCommand(String testClassName): This method determines the compile command for test classes based on the operating system and the test class name. It returns the command as an array of strings.
    - readInputStream(BufferedReader reader): This method reads the console log of the command execution and summarizes it. It logs the execution process using a Logger and returns the summarized output as a String.

The Compilation class provides functionality for compiling Java classes and test classes. It utilizes file handling, command execution, and logging utilities to carry out the compilation process and provide the compilation results.

### Execution.java

This class is responsible for executing test classes and obtaining code coverage information.

- Static Methods:
    - runTest(String inputClassName, String inputTestClassName): This method takes the class name and test class name as parameters, executes the specified test class using Maven, and returns the output of the test execution as a String. It uses a Logger to log the execution process and a CommandExecution class to execute the test command and capture the output. It also includes code to extract coverage information using JaCoCo and appends it to the result.
    - getCoverage(String inputClassName): This method takes the class name as a parameter, retrieves the coverage information for the specified class from the JaCoCo XML report, and returns it as a String. It uses the Jsoup library to parse the XML document, and then iterates through the XML elements to extract the coverage details (covered lines, missed lines, total lines, and coverage percentage). It appends this information to the result and returns it.
- Private Methods:
    - getTestExecutionCommand(String inputTestClassName): This method determines the command for executing the test class based on the operating system and the test class name. It returns the command as an array of strings.

The Execution class provides functionality for executing test classes and obtaining code coverage information. It utilizes Maven for executing the tests, the JaCoCo XML report for coverage information, and the Jsoup library for parsing the XML document. The class also includes logging statements to track the execution process.

### Game.java

This class is a service component annotated with @Service and represents a game.

- Instance Variables:
  - inputTestClassName and inputClassName: These variables store the names of the test class and the main class of the game, respectively.
  - compiled: This boolean variable indicates whether the classes of the game have been compiled.
- Private Methods:
  - obtainCode(String input): This method takes a file name as input, reads the contents of the corresponding text file, and returns the code as a String. It uses a Logger to log the execution process and handles any IOException that occurs during file reading.
- Public Methods:
  - compile(): This method compiles the classes of the game by obtaining the code from text files, invoking the Compilation.compileClass() and Compilation.compileTest() methods to compile the main class and test class, respectively. It appends the compilation results to a StringBuilder and sets the compiled flag accordingly. If the compilation is successful, it returns the string "Compiled" as the result.
  - execute(): This method executes the classes of the game by invoking the Execution.runTest() method to run the test class. If the classes have not been compiled (compiled is false), it returns a message indicating that execution is not possible. After execution, it deletes the compiled class files and resets the compiled flag.
  - setInputTestClassName(String inputTestClassName): This method sets the name of the test class for the game.
  - setInputClassName(String inputClassName): This method sets the name of the main class for the game.

The Game class provides functionality for compiling and executing the classes of a game. It reads the code from text files, invokes the Compilation class for compilation, and uses the Execution class for executing tests. The class also includes logging statements to track the execution process.

### SwaggerConfig.java

This class is responsible for configuring and enabling Swagger for API documentation.

- Class Declaration:
  - The class is named SwaggerConfig.
  - It is annotated with @Configuration, indicating that it is a configuration class.
  - It is also annotated with @EnableSwagger2, enabling Swagger support in the application.
- Bean Method:

    - apiDocket(): This method is annotated with @Bean and configures the Docket bean, which represents the main Swagger configuration.
    - It uses the DocumentationType.SWAGGER_2 type for Swagger 2 documentation.
    - The select() method is used to specify the API endpoints to include in the documentation.
    - RequestHandlerSelectors.basePackage("requirement_t7.controller") is used to select the controllers in the specified package to be included in the documentation. You should replace "requirement_t7.controller" with the package where your controllers are located.
    - The paths(PathSelectors.any()) method is used to include all paths in the documentation.
    - Finally, the apiInfo() method is invoked to provide additional API information, and the build() method is called to build the Docket configuration.
- Private Method:
    - apiInfo(): This method is responsible for creating an ApiInfo object that contains information about the API documentation.
    - It sets the title, description, and version of the API documentation.

The SwaggerConfig class provides the necessary configuration to enable and customize Swagger for API documentation. It uses the Docket bean to define the API endpoints to include, sets the API information using the ApiInfo object, and enables Swagger functionality with the @EnableSwagger2 annotation.

### CommandExecution.java

This class is responsible for executing commands in the command line.

- Method:
    - executeCommand(String[] command): This method takes an array of strings as a parameter representing the command to be executed in the command line.
    - Inside the method, an InputStream variable named inputStream is declared and initialized as null.
    - The method attempts to execute the provided command by using the ProcessBuilder class and redirecting the error stream to the input stream.
    - If the execution is successful, the input stream of the process is obtained using process.getInputStream().
    - If an IOException occurs during the execution, an error message is logged using the Logger class from the same package.
    - The method returns the obtained input stream, which can be used to read the output of the executed command.

The CommandExecution class provides a convenient way to execute commands in the command line. It uses the ProcessBuilder class to create a process and obtain the input stream to read the command's output.

### FileCreator.java

This class is responsible for creating a Java file.

- Method:

- createFile(String name, String code): This method takes two parameters: the name of the Java file to create and the code to be written in the file.
- Inside the method, a File object named file is created with the specified name and ".java" extension.
- The method attempts to create the file and write the provided code into it using a BufferedWriter and FileWriter.
- If the file creation and writing are successful, an informational message is logged using the Logger class from the same package.
- If an IOException occurs during the file creation or writing process, an error message is logged using the Logger class.
- The method returns the created File object.

The FileCreator class provides a convenient way to create a Java file and write code into it. It uses the BufferedWriter and FileWriter classes to perform the file writing operation. The @Component annotation indicates that this class can be managed as a bean by the Spring framework.

### FileDeletor.java

The provided code represents a utility class named FileDeletor in the requirement_t7.model.util package. This class is responsible for deleting a file. Let's analyze the code:

- Method:
    - deleteFile(String path): This method takes a parameter representing the path of the file to be deleted.
    - Inside the method, a File object named file is created with the specified path.
    - The method checks if the file exists using the exists() method of the File class.
    - If the file exists, the method attempts to delete the file using the delete() method of the File class.
    - If the file is successfully deleted, an informational message is logged using the Logger class from the same package.
    - If the file deletion fails, an informational message is logged indicating the failure.
    - If the file does not exist, an informational message is logged indicating that the file does not exist.

The FileDeletor class provides a convenient way to delete a file given its path. It uses the File class to represent the file and perform deletion operations. The @Component annotation indicates that this class can be managed as a bean by the Spring framework.

### Logger.java

The provided code represents a logger utility class named Logger in the requirement_t7.model.util package. This class is responsible for logging messages. It has been designed using the **Singleton design pattern**.

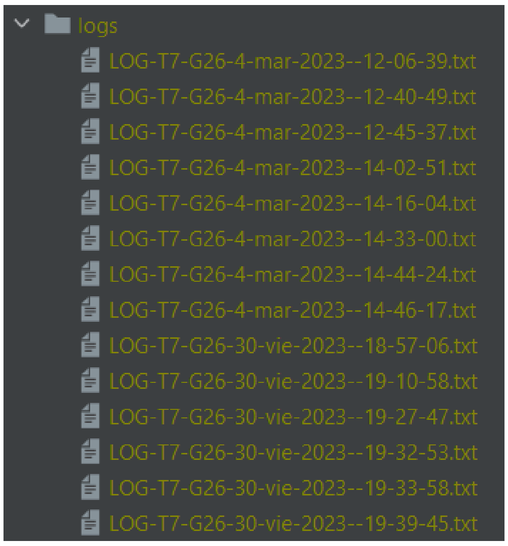
Let's analyze the code:

- Class Declaration:
    - The class is named Logger.
    - It is declared as final, indicating that it cannot be subclassed.

  - The class has a private constructor, indicating that instances of this class can only be created from within the class itself.
  - The class maintains a single instance of Logger using the Singleton design pattern.
  - The class has a path field representing the path of the log file.
- Constants:
  - The class defines constants ERROR, RUNNING, and INFO representing different types of log messages.
- Methods:
  - getInstance(): This method returns the instance of the Logger class. It follows the Singleton pattern by creating a new instance if one doesn't exist or returning the existing instance.
  - createLog(String filePath): This private method creates the log file at the specified filePath. It creates the necessary directory structure using Files.createDirectories(). It then creates an empty file at the specified path using FileWriter. If any error occurs, an error message is logged.
  - getFormattedDateTime(String type): This private method returns a formatted date and time string based on the specified type. It uses SimpleDateFormat to format the current date and time.
  - log(String type, String message): This method logs the specified message with the specified type. It opens the log file using FileWriter, writes the formatted log message, and closes the file. If any error occurs, an error message is logged.
  - getFormattedMessage(String type, String message): This private method returns the formatted log message based on the specified type and message. It includes the formatted date and time, type, and message.

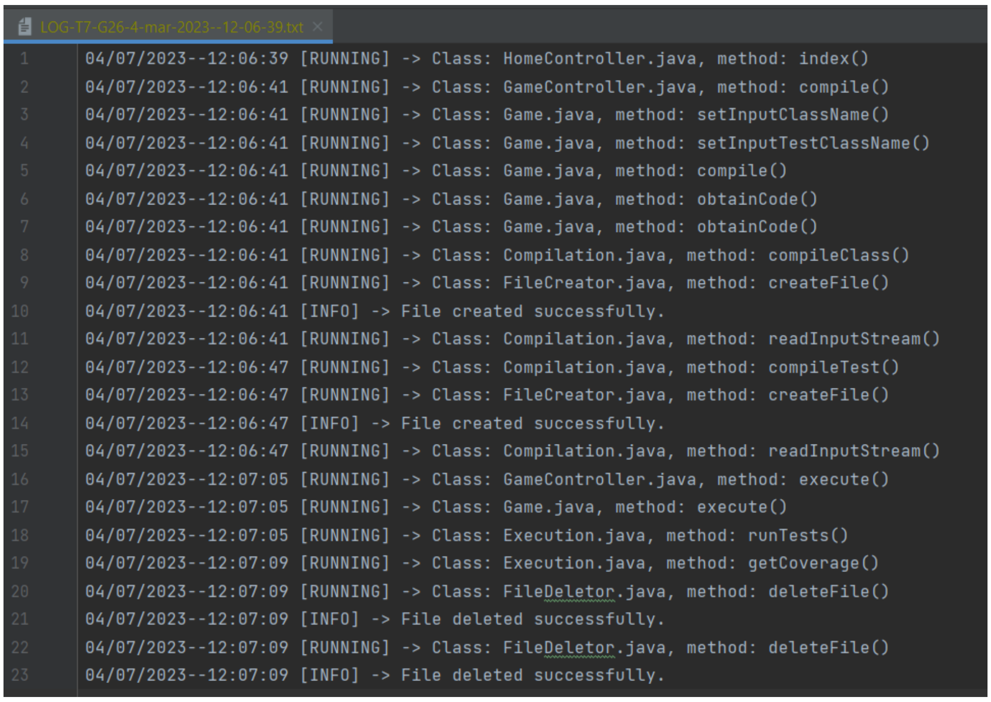
The Logger class provides a way to log messages to a file. It ensures that only one instance of the Logger class exists and allows logging messages of different types. It also creates the log file with a formatted name and directory structure.

Folder containing logs:



```
logs
    LOG-T7-G26-4-mar-2023--12-06-39.txt
    LOG-T7-G26-4-mar-2023--12-40-49.txt
    LOG-T7-G26-4-mar-2023--12-45-37.txt
    LOG-T7-G26-4-mar-2023--14-02-51.txt
    LOG-T7-G26-4-mar-2023--14-16-04.txt
    LOG-T7-G26-4-mar-2023--14-33-00.txt
    LOG-T7-G26-4-mar-2023--14-44-24.txt
    LOG-T7-G26-4-mar-2023--14-46-17.txt
    LOG-T7-G26-30-vie-2023--18-57-06.txt
    LOG-T7-G26-30-vie-2023--19-10-58.txt
    LOG-T7-G26-30-vie-2023--19-27-47.txt
    LOG-T7-G26-30-vie-2023--19-32-53.txt
    LOG-T7-G26-30-vie-2023--19-33-58.txt
    LOG-T7-G26-30-vie-2023--19-39-45.txt
```

Log output example:



LOG-T7-G26-4-mar-2023--12-06-39.txt

```
04/07/2023--12:06:39 [RUNNING] -> Class: HomeController.java, method: index()
04/07/2023--12:06:41 [RUNNING] -> Class: GameController.java, method: compile()
04/07/2023--12:06:41 [RUNNING] -> Class: Game.java, method: setInputClassName()
04/07/2023--12:06:41 [RUNNING] -> Class: Game.java, method: setInputTestClassName()
04/07/2023--12:06:41 [RUNNING] -> Class: Game.java, method: compile()
04/07/2023--12:06:41 [RUNNING] -> Class: Game.java, method: obtainCode()
04/07/2023--12:06:41 [RUNNING] -> Class: Game.java, method: obtainCode()
04/07/2023--12:06:41 [RUNNING] -> Class: Compilation.java, method: compileClass()
04/07/2023--12:06:41 [RUNNING] -> Class: FileCreator.java, method: createFile()
04/07/2023--12:06:41 [INFO] -> File created successfully.
04/07/2023--12:06:41 [RUNNING] -> Class: Compilation.java, method: readInputStream()
04/07/2023--12:06:47 [RUNNING] -> Class: Compilation.java, method: compileTest()
04/07/2023--12:06:47 [RUNNING] -> Class: FileCreator.java, method: createFile()
04/07/2023--12:06:47 [INFO] -> File created successfully.
04/07/2023--12:06:47 [RUNNING] -> Class: Compilation.java, method: readInputStream()
04/07/2023--12:07:05 [RUNNING] -> Class: GameController.java, method: execute()
04/07/2023--12:07:05 [RUNNING] -> Class: Game.java, method: execute()
04/07/2023--12:07:05 [RUNNING] -> Class: Execution.java, method: runTests()
04/07/2023--12:07:09 [RUNNING] -> Class: Execution.java, method: getCoverage()
04/07/2023--12:07:09 [RUNNING] -> Class: FileDeletor.java, method: deleteFile()
04/07/2023--12:07:09 [INFO] -> File deleted successfully.
04/07/2023--12:07:09 [RUNNING] -> Class: FileDeletor.java, method: deleteFile()
04/07/2023--12:07:09 [INFO] -> File deleted successfully.
```

### T7Application.java

This Java class, **T7Application**, is located in the **requirement_t7** package. It serves as the entry point for the application.

The class is annotated with **@SpringBootApplication**, which enables auto-configuration and component scanning based on the classpath. It configures and starts the Spring Boot application.

The **main** method is the starting point of the application. It calls **SpringApplication.run** with the **T7Application** class and any command-line arguments. This method launches the Spring Boot application and starts the embedded web server.

By running this class, the Spring Boot application defined in the project will be started.

## Test architecture and Developed test cases

### Maven

The project is developed using Maven for introducing the dependencies and it is also used in the test execution.

### pom.xml

The pom.xml file is an XML file that is used in Maven-based Java projects to define project dependencies, build profiles, and other information that is necessary for building and managing the project. It is typically located in the root directory of the project and contains information such as the project's name, version, and a list of dependencies that are needed to build the project.

### Build

Here's a summary of the build configuration in the POM:

- **<sourceDirectory>**: Specifies the directory where the main Java source code is located. In this case, it is set to **src/main/java**.

- **<plugins>**: This section contains a list of plugins used during the build process. The plugins are executed in the order they are defined.
    - **maven-compiler-plugin**: This plugin is responsible for compiling the Java source code. It is configured with a specific version (**3.8.1**) and sets the Java release version to **17**.
    - **maven-surefire-plugin**: This plugin is used for executing tests. It is configured to include the **TestSuite.java** test class during the test execution.
    - **jacoco-maven-plugin**: This plugin is used for code coverage analysis. It generates code coverage reports using JaCoCo. It is configured to execute during the **test** phase of the build lifecycle.
- Overall, the build configuration sets up the necessary plugins for compiling the code, running tests, and generating code coverage reports. It ensures that the project is built correctly and includes code quality analysis.

### Dependencies

Additionally, the POM file defines the project's dependencies, including:

- **org.springframework.boot:spring-boot-starter-web**: Starter for building web applications using Spring MVC.
- **org.springframework.boot:spring-boot-starter-tomcat**: Starter for using Apache Tomcat as the embedded servlet container.
- **org.springframework.boot:spring-boot-starter-test**: Starter for testing Spring Boot applications.
- **javax.servlet:jstl**: JavaServer Pages Standard Tag Library for JSP-based views.
- **org.springframework.boot:spring-boot-starter-thymeleaf**: Starter for integrating Thymeleaf as the template engine for Spring MVC.
- **javassist:javassist:3.12.1.GA**: Dependency for bytecode manipulation used by Spring framework.
- **org.junit.jupiter:junit-jupiter-api:5.9.3**: JUnit Jupiter API for unit testing.
- **org.junit.platform:junit-platform-suite:1.9.3**: JUnit Platform Suite for organizing and executing multiple test cases.
- **org.junit.vintage:junit-vintage-engine:5.9.3**: JUnit Vintage Engine for running JUnit 3 and 4 tests in JUnit 5.
- **com.github.hazendaz.jsoup:jsoup:1.15.1**: Java HTML parser used for web scraping.
- **com.theoryinpractise:halbuilder-xml:4.1.3**: XML support for HAL (Hypertext Application Language) responses.
- **io.springfox:springfox-swagger-ui:3.0.0**: Swagger UI for visualizing and interacting with RESTful APIs.
- **io.springfox:springfox-swagger2:3.0.0**: Swagger core library for generating API documentation.
- **io.springfox:springfox-boot-starter:3.0.0**: Starter for integrating Swagger with Spring Boot applications.

### Properties

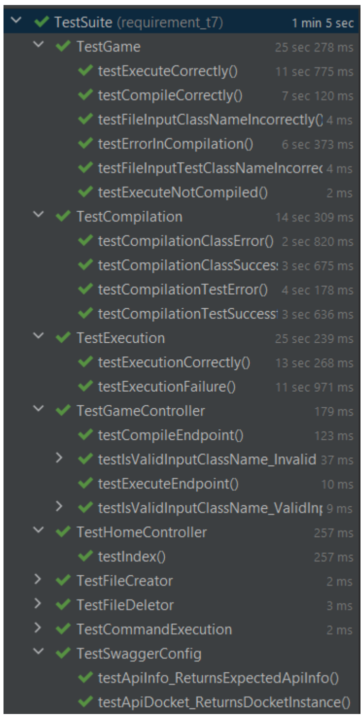
**<properties>** in the given POM file sets the project-level properties used for configuring various aspects of the build and dependencies. It includes the following:

- **maven.compiler.source** and **maven.compiler.target**: Specifies the Java version used for compiling the source code (set to 11).
- **sonar.organization** and **sonar.host.url**: Configures the SonarCloud properties for static code analysis.

## Test Execution

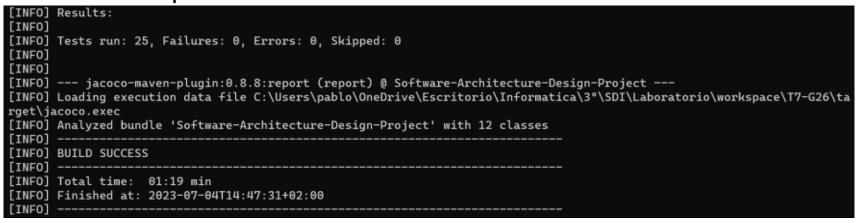
When having the project in local, there are two ways to execute the tests of the project.

- The first one is directly from IntelliJ.
  1. In the package explorer you right click on the TestSuite class.
  2. Select the option Run 'TestSuite'.
  3. This should be the output.



```
✓ TestSuite (requirement_t7)                     1 min 5 sec
  ✓ TestGame                                   25 sec 278 ms
    ✓ testExecuteCorrectly()                   11 sec 775 ms
    ✓ testCompileCorrectly()                    7 sec 120 ms
    ✓ testFileInputClassNameIncorrectly(        4 ms
    ✓ testErrorInCompilation()                  6 sec 373 ms
    ✓ testFileInputTestClassNameIncorre         4 ms
    ✓ testExecuteNotCompiled()                  2 ms
  ✓ TestCompilation                            14 sec 309 ms
    ✓ testCompilationClassError()               2 sec 820 ms
    ✓ testCompilationClassSucces                3 sec 675 ms
    ✓ testCompilationTestError()                4 sec 178 ms
    ✓ testCompilationTestSuccess                3 sec 636 ms
  ✓ TestExecution                              25 sec 239 ms
    ✓ testExecutionCorrectly()                 13 sec 268 ms
    ✓ testExecutionFailure()                   11 sec 971 ms
  ✓ TestGameController                          179 ms
    ✓ testCompileEndpoint()                     123 ms
    ✓ testIsValidInputClassName_Invalid         37 ms
    ✓ testExecuteEndpoint()                     10 ms
    ✓ testIsValidInputClassName_ValidIn         9 ms
  ✓ TestHomeController                          257 ms
    ✓ testIndex()                               257 ms
  ✓ TestFileCreator                             2 ms
  ✓ TestFileDeletor                             3 ms
  ✓ TestCommandExecution                        2 ms
  ✓ TestSwaggerConfig
    ✓ testApiInfo_ReturnsExpectedApiInfo()
    ✓ testApiDocket_ReturnsDocketInstance()
```

- The second one is using the cmd in windows for example.
  1. Executing this command inside the cmd in the folder where the project is stored: "mvn -B test"
  2. This is the output:



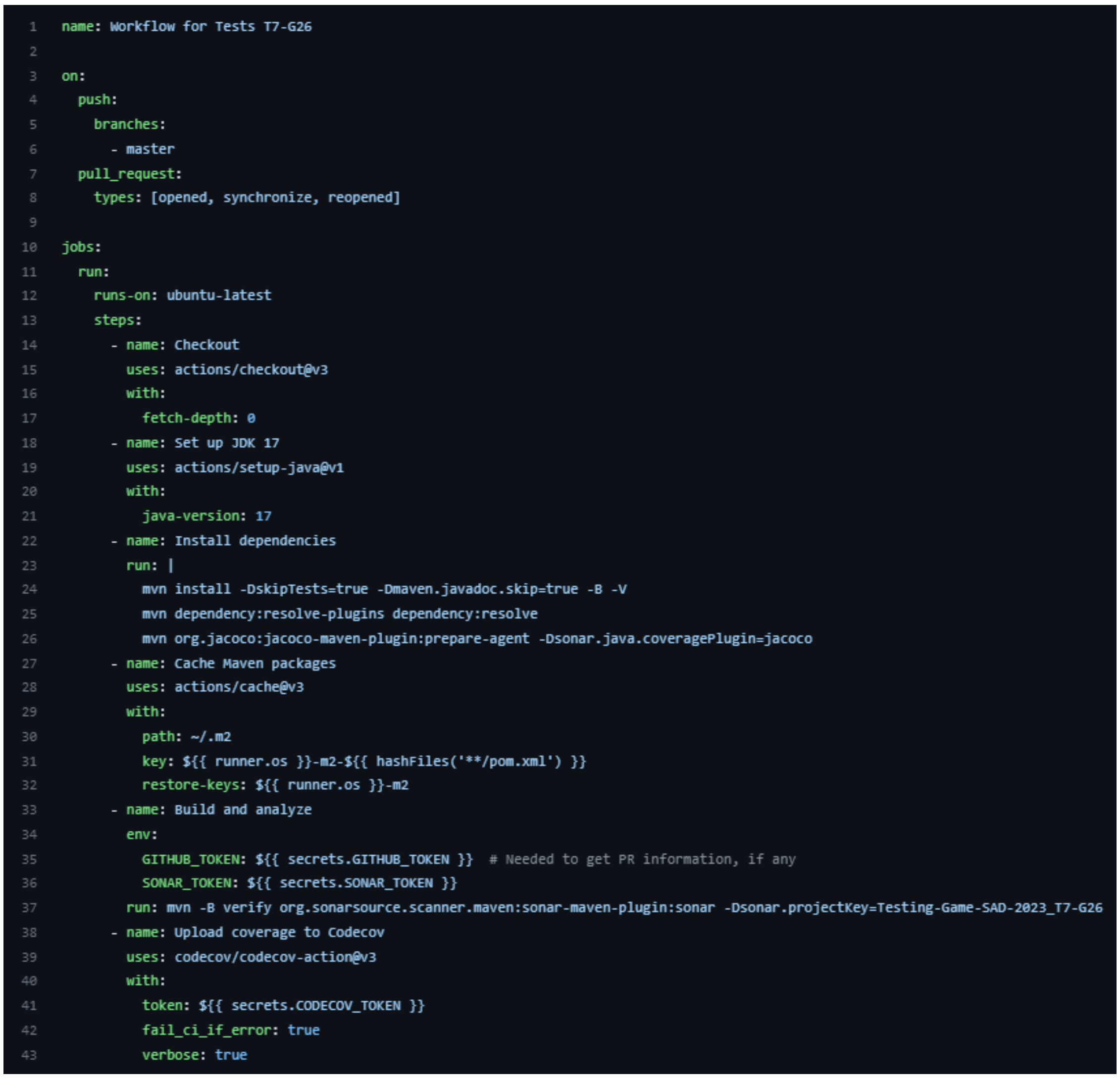
```
[INFO] Results:
[INFO]
[INFO] Tests run: 25, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO]
[INFO] --- jacoco-maven-plugin:0.8.8:report (report) @ Software-Architecture-Design-Project ---
[INFO] Loading execution data file C:\Users\pablo\OneDrive\Escritorio\Informatica\3°\SDI\Laboratorio\workspace\T7-G26\ta
rget\jacoco.exec
[INFO] Analyzed bundle 'Software-Architecture-Design-Project' with 12 classes
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  01:19 min
[INFO] Finished at: 2023-07-04T14:47:31+02:00
[INFO] ------------------------------------------------------------------------
```

## Continuous Integration

A key aspect of DevOps is the practice of continuous integration, which refers to automatically integrating code changes from multiple contributors into a single software project. With this approach, every time a pull request is made to merge changes from the 'develop' branch into the 'master' branch, a workflow is automatically executed to provide feedback on the changes.

## GitHub Actions

GitHub Actions is a tool that automates your software development pipeline, including building, testing, and deploying. It enables you to set up workflows that automatically build and test changes to your codebase whenever a pull request is made and can also deploy those changes to production after they are merged.
In this project the only workflow that I am using is called ci.yml and it has this code.



```yaml
name: Workflow for Tests T7-G26

on:
  push:
    branches:
      - master
  pull_request:
    types: [opened, synchronize, reopened]

jobs:
  run:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout
        uses: actions/checkout@v3
        with:
          fetch-depth: 0
      - name: Set up JDK 17
        uses: actions/setup-java@v1
        with:
          java-version: 17
      - name: Install dependencies
        run: |
          mvn install -DskipTests=true -Dmaven.javadoc.skip=true -B -V
          mvn dependency:resolve-plugins dependency:resolve
          mvn org.jacoco:jacoco-maven-plugin:prepare-agent -Dsonar.java.coveragePlugin=jacoco
      - name: Cache Maven packages
        uses: actions/cache@v3
        with:
          path: ~/.m2
          key: ${{ runner.os }}-m2-${{ hashFiles('**/pom.xml') }}
          restore-keys: ${{ runner.os }}-m2
      - name: Build and analyze
        env:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}  # Needed to get PR information, if any
          SONAR_TOKEN: ${{ secrets.SONAR_TOKEN }}
        run: mvn -B verify org.sonarsource.scanner.maven:sonar-maven-plugin:sonar -Dsonar.projectKey=Testing-Game-SAD-2023_T7-G26
      - name: Upload coverage to Codecov
        uses: codecov/codecov-action@v3
        with:
          token: ${{ secrets.CODECOV_TOKEN }}
          fail_ci_if_error: true
          verbose: true
```

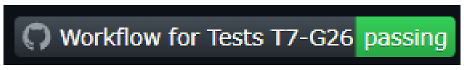
This GitHub Actions workflow, named "Workflow for Tests T7-G26," is triggered by push events to the master branch and pull request events that are opened, synchronized, or reopened.

The workflow runs on the latest version of Ubuntu. It consists of several steps:

- **Checkout**: This step checks out the repository using the actions/checkout@v3 action.
- **Set up JDK 17**: This step sets up Java Development Kit (JDK) version 17 using the actions/setup-java@v1 action.
- **Install dependencies**: This step installs project dependencies and performs other necessary setup tasks using Maven commands.
- **Cache Maven packages**: This step caches Maven packages to improve build performance using the actions/cache@v3 action.

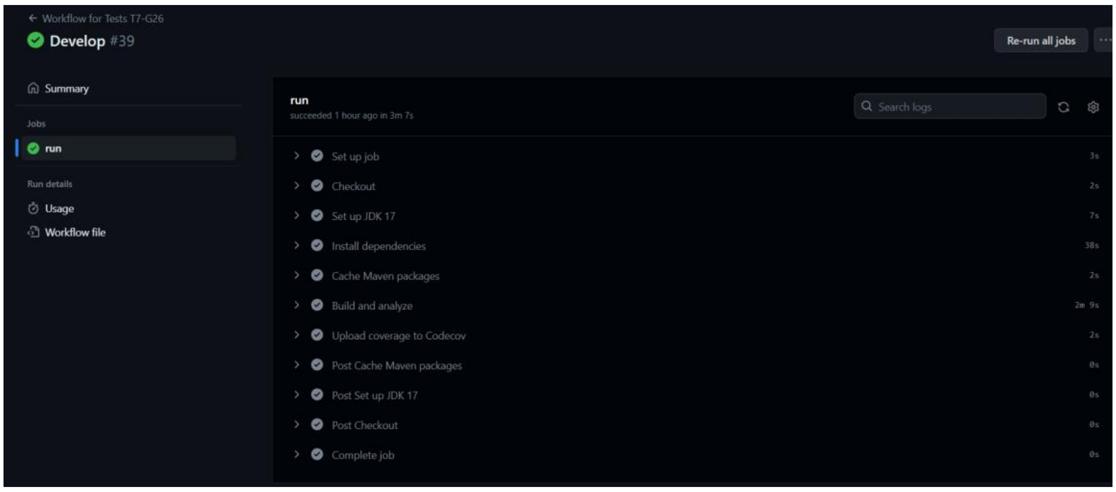
- **Build and analyze**: This step builds the project, runs tests, and analyzes the code using SonarQube. It sets environment variables for the GitHub token and SonarQube token to access relevant information.
- **Upload coverage to Codecov**: This step uploads code coverage information to Codecov using the codecov/codecov-action@v3 action. It specifies the token for authentication and sets options to fail the CI process if there are any errors and provide verbose output.

Through the badge in the README.md it can be accessed the workflow of the project:



Workflow for Tests T7-G26 passing

Photo of the last workflow in the repo:



← Workflow for Tests T7-G26

Develop #39

Summary

Jobs

run

Run details

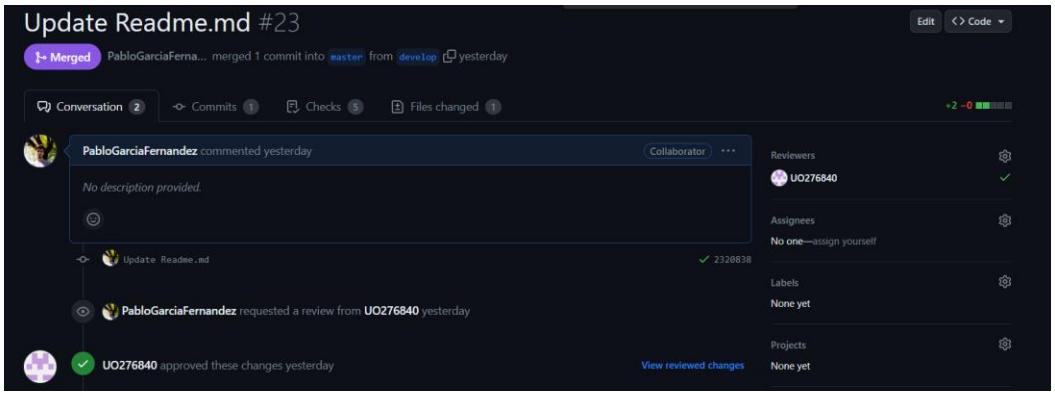
Usage

Workflow file

run
succeeded 1 hour ago in 3m 7s

```
Set up job                      3s
Checkout                        2s
Set up JDK 17                   7s
Install dependencies            38s
Cache Maven packages            2s
Build and analyze               2m 9s
Upload coverage to Codecov      2s
Post Cache Maven packages       0s
Post Set up JDK 17              0s
Post Checkout                   0s
Complete job                    0s
```

Photo of how it looks like a pull request:



Update Readme.md #23

Merged PabloGarciaFerna... merged 1 commit into master from develop yesterday

Conversation 2 · Commits 1 · Checks 5 · Files changed 1 · +2 −0

PabloGarciaFernandez commented yesterday · Collaborator

*No description provided.*

Update Readme.md ✓ 2320838

PabloGarciaFernandez requested a review from UO276840 yesterday

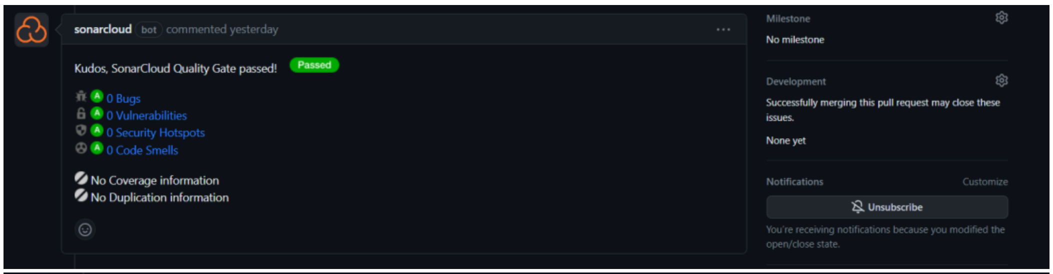
UO276840 approved these changes yesterday · View reviewed changes

Reviewers: UO276840 ✓

Assignees: No one—assign yourself

Labels: None yet

Projects: None yet


sonarcloud bot commented yesterday

Kudos, SonarCloud Quality Gate passed! Passed

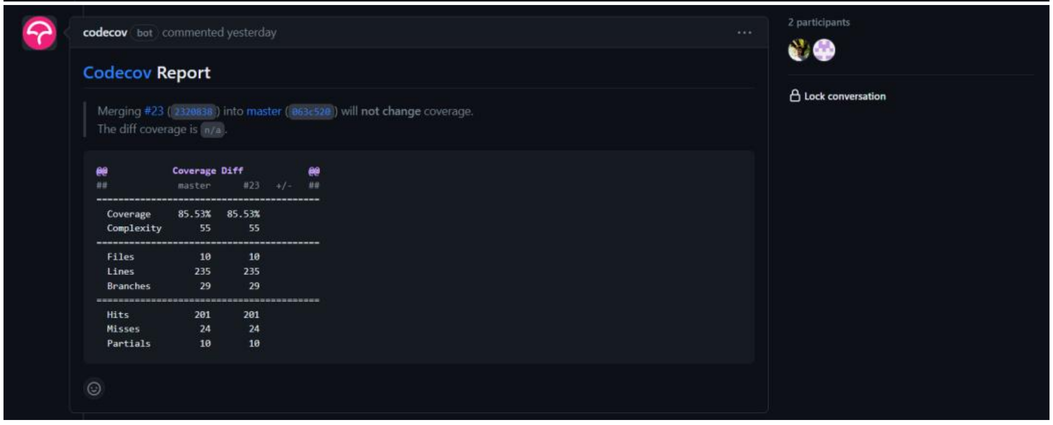
A 0 Bugs
A 0 Vulnerabilities
A 0 Security Hotspots
A 0 Code Smells

No Coverage information
No Duplication information

Milestone
No milestone

Development
Successfully merging this pull request may close these issues.
None yet

Notifications Customize
Unsubscribe
You're receiving notifications because you modified the open/close state.


codecov bot commented yesterday

Codecov Report

Merging #23 (2320838) into master (063c520) will not change coverage.
The diff coverage is n/a.

```
@@           Coverage Diff          @@
##            master     #23   +/-   ##
=======================================
  Coverage    85.53%  85.53%
  Complexity      55      55
=======================================
  Files           10      10
  Lines          235     235
  Branches        29      29
=======================================
  Hits           201     201
  Misses          24      24
  Partials        10      10
```

2 participants

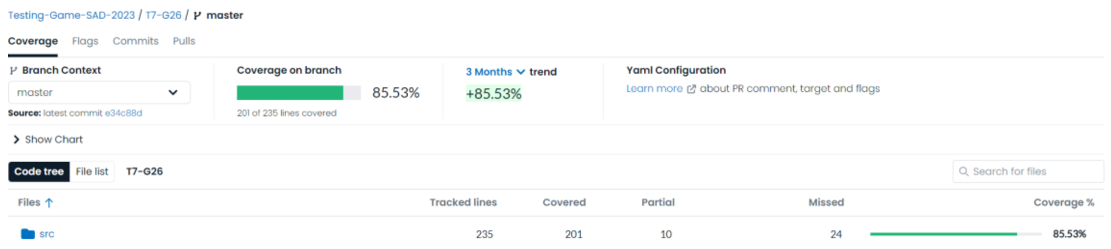
Lock conversation

## Codecov

Codecov.io is a code coverage analysis tool that helps developers to understand how much of their code is being tested when integrated with GitHub. It provides developers with a detailed report of the percentage of code that is covered by tests, as well as the lines of code that were executed during the tests. This information can be used to identify areas of the codebase that are not being adequately tested, and to make more informed decisions about where to focus additional testing efforts.

Through the badge in the README.md it can be accessed the coverage of the project:

This is the view inside Codecov:


Testing-Game-SAD-2023 / T7-G26 / master

Coverage | Flags | Commits | Pulls

Branch Context: master
Source: latest commit e34c88d

Coverage on branch: 85.53%
201 of 235 lines covered

3 Months trend: +85.53%

Yaml Configuration
Learn more about PR comment, target and flags

Show Chart

Code tree | File list | T7-G26 | Search for files

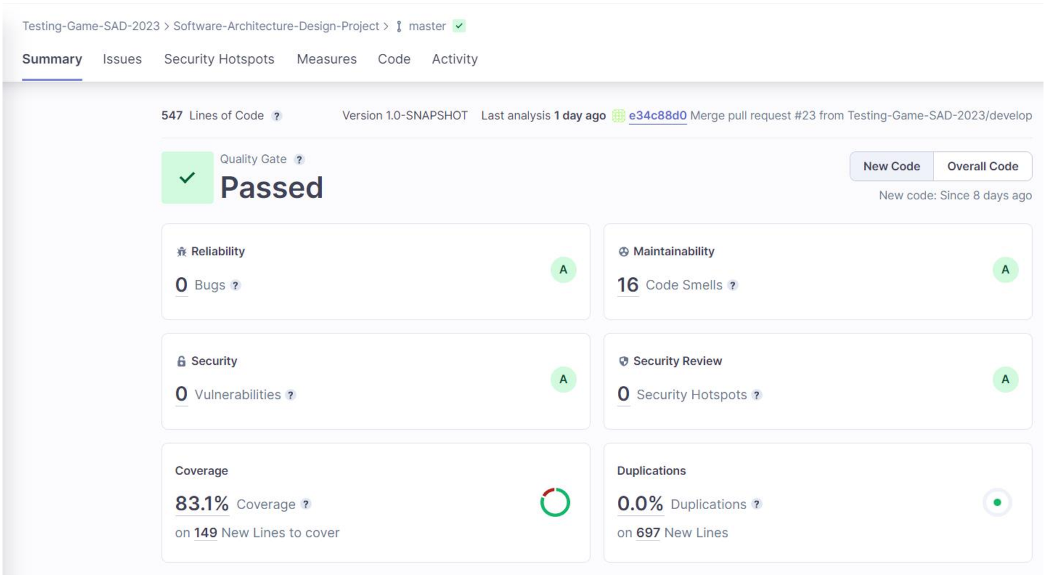
| Files | Tracked lines | Covered | Partial | Missed | Coverage % |
|---|---|---|---|---|---|
| src | 235 | 201 | 10 | 24 | 85.53% |

## SonarCloud

SonarCloud is a cloud-based code quality management platform that helps Java developers to improve the quality of their code when integrated with GitHub. It automatically analyzes Java code and provides developers with actionable insights on how to improve it. SonarCloud uses static code analysis to identify bugs, vulnerabilities, and security issues in the code, as well as providing metrics such as code coverage, technical debt, and maintainability.

Through the badge in the README.md it can be accessed the quality gate of the project:

This is the view inside Sonarcloud:



Testing-Game-SAD-2023 > Software-Architecture-Design-Project > master

Summary | Issues | Security Hotspots | Measures | Code | Activity

547 Lines of Code   Version 1.0-SNAPSHOT   Last analysis 1 day ago   e34c88d0 Merge pull request #23 from Testing-Game-SAD-2023/develop

Quality Gate
Passed

New Code | Overall Code
New code: Since 8 days ago

Reliability: 0 Bugs — A
Maintainability: 16 Code Smells — A
Security: 0 Vulnerabilities — A
Security Review: 0 Security Hotspots — A
Coverage: 83.1% Coverage on 149 New Lines to cover
Duplications: 0.0% Duplications on 697 New Lines

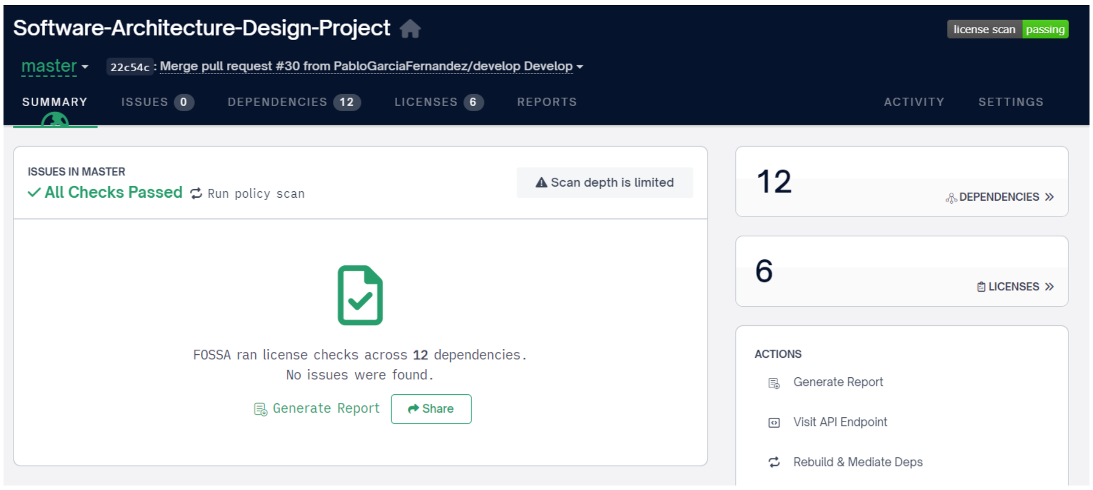
## FOSSA

FOSSA is a tool that helps developers manage and track open-source software licenses used in their projects. It integrates with platforms like GitHub and scans code to identify OSS components and licenses and provides a central interface for managing and reporting on them. It helps to ensure compliance and identify potential licensing risks.

Through the badge in the README.md it can be accessed the license scan of the project:

license scan | passing

This is the view inside FOSSA:



Software-Architecture-Design-Project   license scan passing

master   22c54c : Merge pull request #30 from PabloGarciaFernandez/develop Develop

SUMMARY | ISSUES 0 | DEPENDENCIES 12 | LICENSES 6 | REPORTS | ACTIVITY | SETTINGS

ISSUES IN MASTER
All Checks Passed   Run policy scan   Scan depth is limited

FOSSA ran license checks across 12 dependencies.
No issues were found.

Generate Report   Share

12 DEPENDENCIES

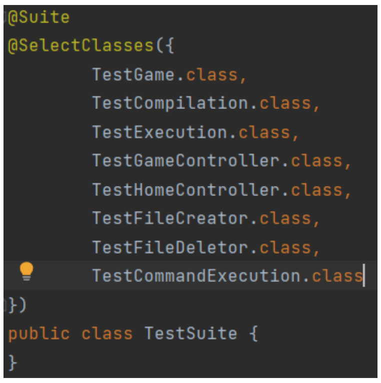
6 LICENSES

ACTIONS
Generate Report
Visit API Endpoint
Rebuild & Mediate Deps

## Test cases

### TestSuite.java

In charge of executing all the test cases and is used by Jacoco to generate the report of the code coverage.

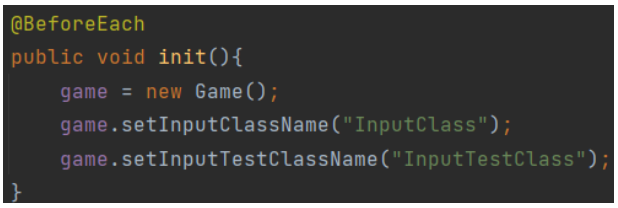

```java
@Suite
@SelectClasses({
        TestGame.class,
        TestCompilation.class,
        TestExecution.class,
        TestGameController.class,
        TestHomeController.class,
        TestFileCreator.class,
        TestFileDeletor.class,
        TestCommandExecution.class
})
public class TestSuite {
}
```

### TestGame.java

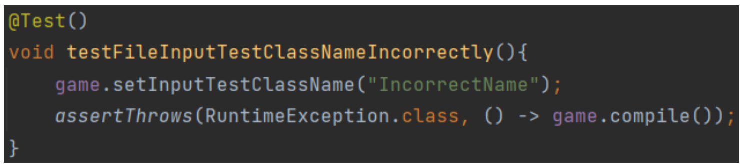
Test class in charge of testing Game.java class.

Developed tests:

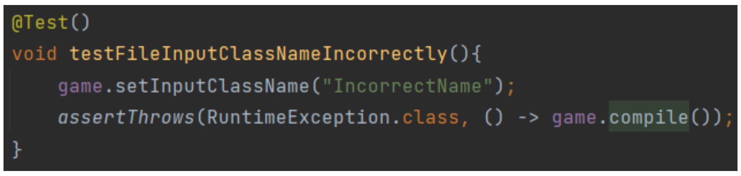
- @BeforeEach **init()**: Before each test a new Game is set with the corresponding class names.


```java
@BeforeEach
public void init(){
    game = new Game();
    game.setInputClassName("InputClass");
    game.setInputTestClassName("InputTestClass");
}
```

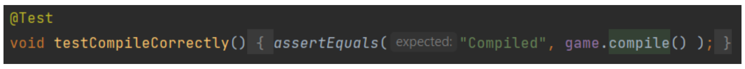
- @Test **testFileInputTestClassNameIncorrectly()**: Using an incorrect test class name will throw a RuntimeException because the file does not exist.


```java
@Test()
void testFileInputTestClassNameIncorrectly(){
    game.setInputTestClassName("IncorrectName");
    assertThrows(RuntimeException.class, () -> game.compile());
}
```

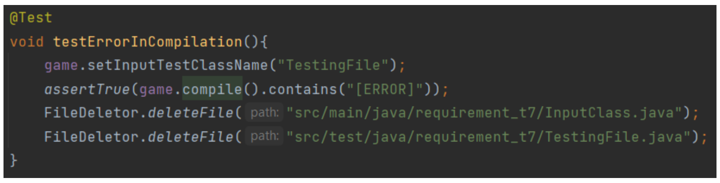
- @Test **testFileInputClassNameIncorrectly()**: Using an incorrect class name will throw a RuntimeException because the file does not exist.


```java
@Test()
void testFileInputClassNameIncorrectly(){
    game.setInputClassName("IncorrectName");
    assertThrows(RuntimeException.class, () -> game.compile());
}
```

- @Test **compileCorrectly()**: Compiling with default variables set in the init() compiles correctly.

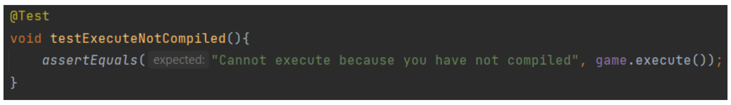

```java
@Test
void testCompileCorrectly() { assertEquals( expected: "Compiled", game.compile() ); }
```

- @Test **testErrorInCompilation()**: Receives a txt file that exists but the content should not compile.



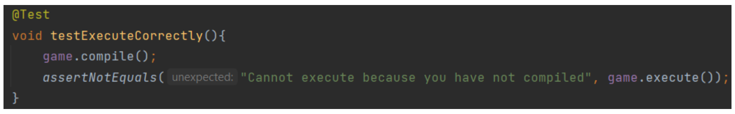
```java
@Test
void testErrorInCompilation(){
    game.setInputTestClassName("TestingFile");
    assertTrue(game.compile().contains("[ERROR]"));
    FileDeletor.deleteFile( path: "src/main/java/requirement_t7/InputClass.java");
    FileDeletor.deleteFile( path: "src/test/java/requirement_t7/TestingFile.java");
}
```

- @Test **testExecuteNotCompiled()**: Trying to execute a test class without having compiled the class to be tested.



```java
@Test
void testExecuteNotCompiled(){
    assertEquals( expected: "Cannot execute because you have not compiled", game.execute());
}
```

- @Test **testExecuteCorrectly()**: Normal execution of a test.



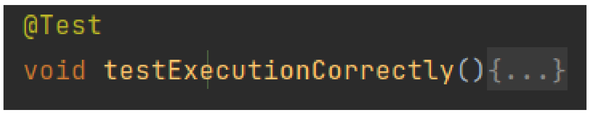
```java
@Test
void testExecuteCorrectly(){
    game.compile();
    assertNotEquals( unexpected: "Cannot execute because you have not compiled", game.execute());
}
```

## TestExecution.java

Test class in charge of testing Execution.java class.

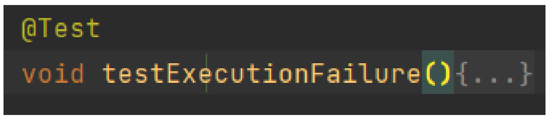
Developed tests:

- @Test **testExecutionCorrectly()**: We compile two class and confirm that the tests pass.



```java
@Test
void testExecutionCorrectly(){...}
```

- @Test **testExecutionFailure()**: We compile two class and confirm that the tests fail.



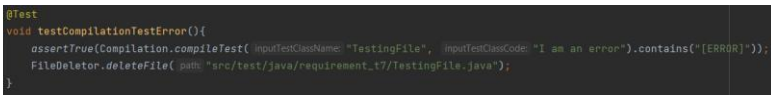
```java
@Test
void testExecutionFailure(){...}
```

## TestCompilation.java

Test class in charge of testing Compilation.java class.

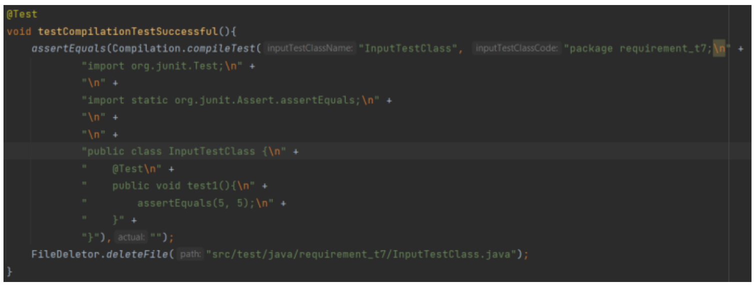
Developed tests:

- @Test **testCompilationTestError()**: Try to compile a test that has code that does not compile.



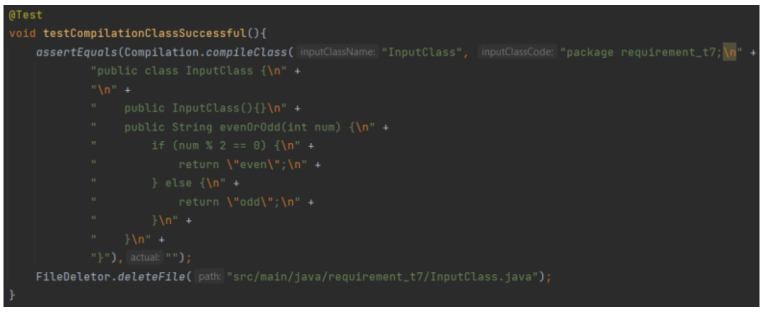
```java
@Test
void testCompilationTestError(){
    assertTrue(Compilation.compileTest( inputTestClassName: "TestingFile",  inputTestClassCode: "I am an error").contains("[ERROR]"));
    FileDeletor.deleteFile( path: "src/test/java/requirement_t7/TestingFile.java");
}
```

- @Test **testCompilationTestSuccessful()**: Compile successfully test code that compiles.



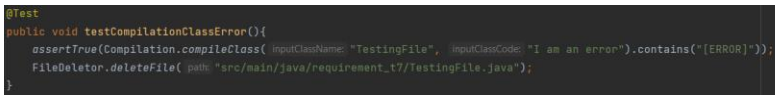
```java
@Test
void testCompilationTestSuccessful(){
    assertEquals(Compilation.compileTest( inputTestClassName: "InputTestClass",  inputTestClassCode: "package requirement_t7;\n" +
            "import org.junit.Test;\n" +
            "\n" +
            "import static org.junit.Assert.assertEquals;\n" +
            "\n" +
            "\n" +
            "public class InputTestClass {\n" +
            "    @Test\n" +
            "    public void test1(){\n" +
            "        assertEquals(5, 5);\n" +
            "    }" +
            "}"), actual: "");
    FileDeletor.deleteFile( path: "src/test/java/requirement_t7/InputTestClass.java");
}
```

- @Test **testCompilationClassSuccessful()**: Compile successfully class code that compiles.



```java
@Test
void testCompilationClassSuccessful(){
    assertEquals(Compilation.compileClass( inputClassName: "InputClass",  inputClassCode: "package requirement_t7;\n" +
            "public class InputClass {\n" +
            "\n" +
            "    public InputClass(){}\n" +
            "    public String evenOrOdd(int num) {\n" +
            "        if (num % 2 == 0) {\n" +
            "            return \"even\";\n" +
            "        } else {\n" +
            "            return \"odd\";\n" +
            "        }\n" +
            "    }\n" +
            "}"), actual: "");
    FileDeletor.deleteFile( path: "src/main/java/requirement_t7/InputClass.java");
}
```

- @Test **testCompilationClassError()**: Try to compile a class that has code that does not compile.



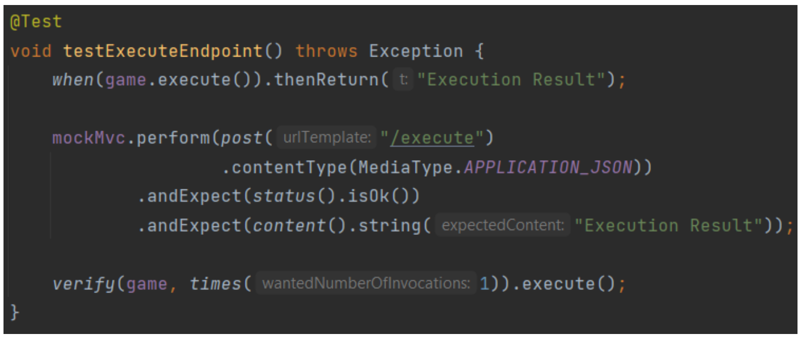
```java
@Test
public void testCompilationClassError(){
    assertTrue(Compilation.compileClass( inputClassName: "TestingFile",  inputClassCode: "I am an error").contains("[ERROR]"));
    FileDeletor.deleteFile( path: "src/main/java/requirement_t7/TestingFile.java");
}
```

### TestGameController.java

Test class in charge of testing GameController.java class.

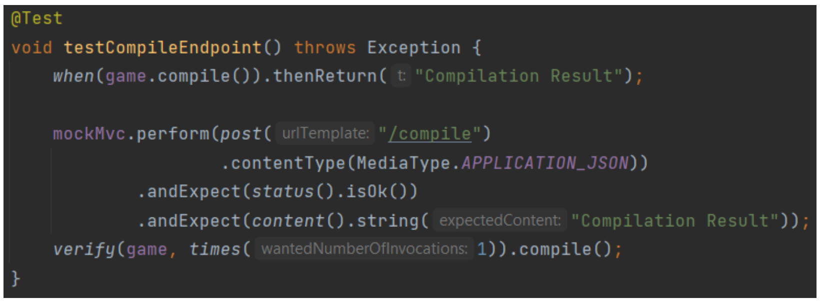
Developed tests:

- @Test **testExecuteEndpoint()**: Test trying /execute endpoint.



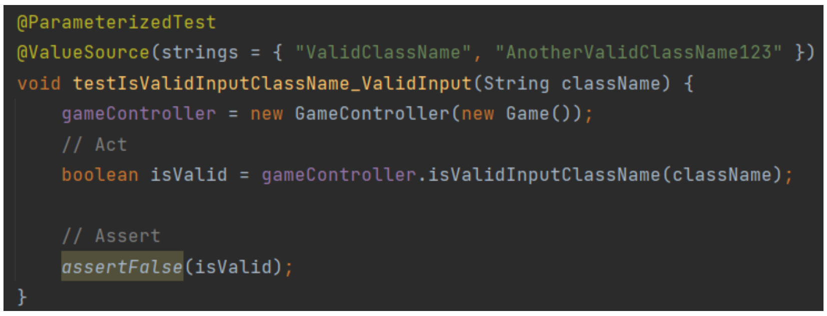
```java
@Test
void testExecuteEndpoint() throws Exception {
    when(game.execute()).thenReturn( t: "Execution Result");

    mockMvc.perform(post( urlTemplate: "/execute")
                    .contentType(MediaType.APPLICATION_JSON))
            .andExpect(status().isOk())
            .andExpect(content().string( expectedContent: "Execution Result"));

    verify(game, times( wantedNumberOfInvocations: 1)).execute();
}
```

- @Test **testCompileEndpoint()**: Test trying /compile endpoint.



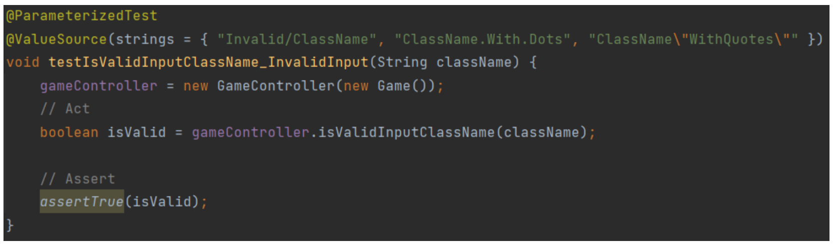
```java
@Test
void testCompileEndpoint() throws Exception {
    when(game.compile()).thenReturn( t: "Compilation Result");

    mockMvc.perform(post( urlTemplate: "/compile")
                    .contentType(MediaType.APPLICATION_JSON))
            .andExpect(status().isOk())
            .andExpect(content().string( expectedContent: "Compilation Result"));
    verify(game, times( wantedNumberOfInvocations: 1)).compile();
}
```

- @ParameterizedTest **testIsValidInputClassName_ValidInput()**: Parameterized test for checking if a name input is valid.



```java
@ParameterizedTest
@ValueSource(strings = { "ValidClassName", "AnotherValidClassName123" })
void testIsValidInputClassName_ValidInput(String className) {
    gameController = new GameController(new Game());
    // Act
    boolean isValid = gameController.isValidInputClassName(className);

    // Assert
    assertFalse(isValid);
}
```

- @ParameterizedTest **testIsValidInputClassName_InvalidInput()**: Parameterized test for checking if a name input is valid.



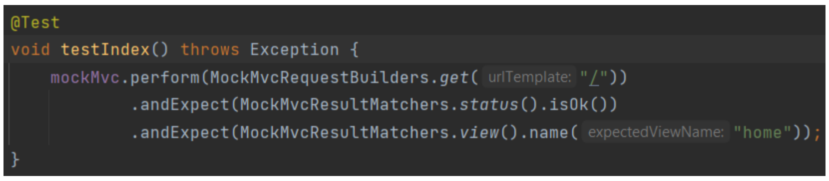
```java
@ParameterizedTest
@ValueSource(strings = { "Invalid/ClassName", "ClassName.With.Dots", "ClassName\"WithQuotes\"" })
void testIsValidInputClassName_InvalidInput(String className) {
    gameController = new GameController(new Game());
    // Act
    boolean isValid = gameController.isValidInputClassName(className);

    // Assert
    assertTrue(isValid);
}
```

## TestHomeController.java

Test class in charge of testing HomeController.java class.

Developed test:

- @Test **testIndex()**: Test if endpoint / is working properly.



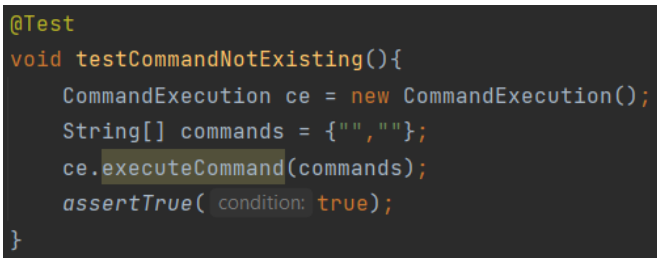
```java
@Test
void testIndex() throws Exception {
    mockMvc.perform(MockMvcRequestBuilders.get( urlTemplate: "/"))
            .andExpect(MockMvcResultMatchers.status().isOk())
            .andExpect(MockMvcResultMatchers.view().name( expectedViewName: "home"));
}
```

## TestCommandExecution.java

Test class in charge of testing CommandExecution.java class.

Developed test:

- @Test **testCommandNotExisting()**: Test that checks that a command not exist.



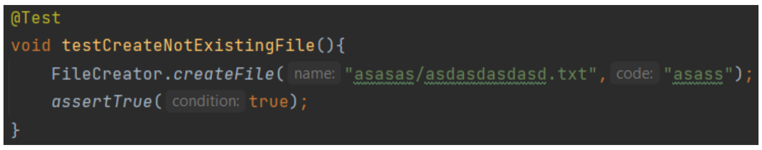
```java
@Test
void testCommandNotExisting(){
    CommandExecution ce = new CommandExecution();
    String[] commands = {"",""};
    ce.executeCommand(commands);
    assertTrue( condition: true);
}
```

## TestFileCreator.java

Test class in charge of testing FileCreator.java class.

Developed test:

- @Test **testCreateNotExistingFile()**: Test to try to create a nonexistent file.



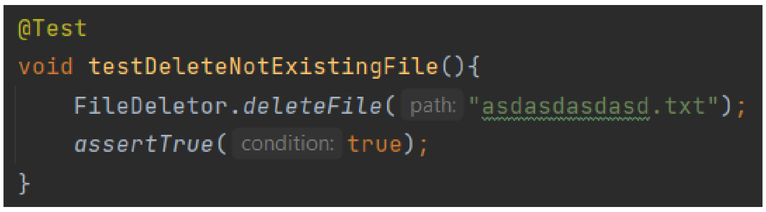
```java
@Test
void testCreateNotExistingFile(){
    FileCreator.createFile( name: "asasas/asdasdasdasd.txt", code: "asass");
    assertTrue( condition: true);
}
```

### TestFileDeletor.java

Test class in charge of testing FileCreator.java class.

Developed test:

- @Test **testDeleteNotExistingFile()**: Test to try to delete a nonexistent file.



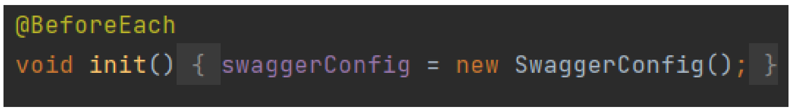
```java
@Test
void testDeleteNotExistingFile(){
    FileDeletor.deleteFile( path: "asdasdasdasd.txt");
    assertTrue( condition: true);
}
```

### TestSwaggerConfig.java

Test class in charge of testing FileCreator.java class.

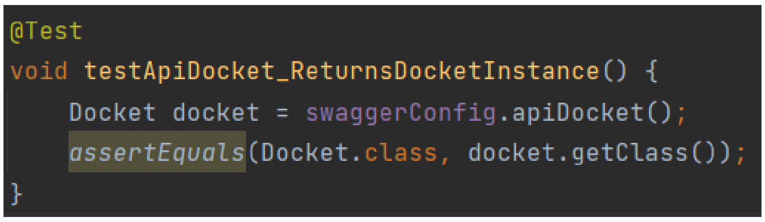
Developed tests:

- @BeforeEach **init()**: Each time a test is executed it creates a SwaggerConfig.



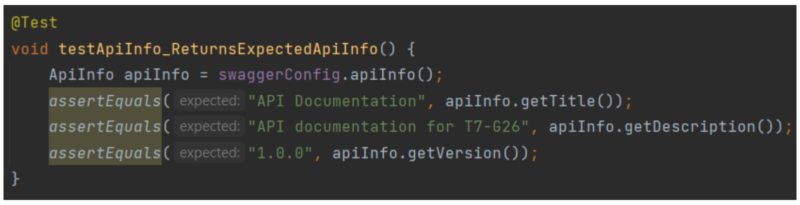
```java
@BeforeEach
void init() { swaggerConfig = new SwaggerConfig(); }
```

- @Test **testApiDocket_ReturnsDocketInstance()**: Tests that we are getting a docket class.



```java
@Test
void testApiDocket_ReturnsDocketInstance() {
    Docket docket = swaggerConfig.apiDocket();
    assertEquals(Docket.class, docket.getClass());
}
```

- @Test **testApiInfo_ReturnsExpectedApiInfo()**: Test that checks that we are getting the correct API info back.



```java
@Test
void testApiInfo_ReturnsExpectedApiInfo() {
    ApiInfo apiInfo = swaggerConfig.apiInfo();
    assertEquals( expected: "API Documentation", apiInfo.getTitle());
    assertEquals( expected: "API documentation for T7-G26", apiInfo.getDescription());
    assertEquals( expected: "1.0.0", apiInfo.getVersion());
}
```

## API

An API is a set of rules and protocols that enables software applications to communicate and interact with each other. It serves as a bridge that allows different systems, services, or applications to exchange data and perform specific tasks in a standardized and efficient manner.

By defining how software components should interact, APIs facilitate seamless integration and interoperability, enabling developers to build powerful and interconnected applications.

In this case we would be able to connect our code with other groups code by just using our API.

## Swagger

Swagger is a tool used to efficiently design, document, and test APIs. It facilitates the creation of interactive and readable documentation for developers, making it easy to understand and use the API. Additionally, Swagger provides a graphical interface (Swagger UI) for testing API endpoints without the need for external tools.

## Docker

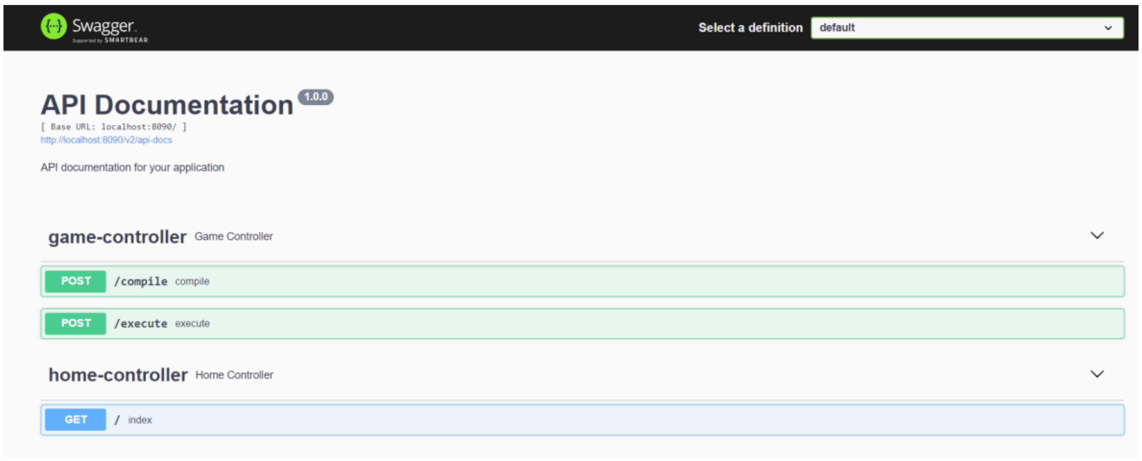
Docker is an open-source platform that allows developers to create, distribute, and run applications in containers. These containers are isolated and portable environments that include everything needed to run an application, such as code, dependencies, and configurations. Docker simplifies the deployment and management of applications, enabling consistent and efficient deployment across different environments.

## Documentation

This Swagger view represents the API documentation for your application. It provides information about the available endpoints and their corresponding operations.

- **game-controller**: This section represents the Game Controller, which handles requests related to game functionality.
  - **POST /compile**: This endpoint is used to compile the game. It expects a POST request and triggers the compilation process.
  - **POST /execute**: This endpoint is used to execute the game. It expects a POST request and triggers the execution process.
- **home-controller**: This section represents the Home Controller, which handles requests related to the home page.
  - **GET /**: This endpoint is used to retrieve the home page. It expects a GET request and returns the index page.

These endpoints and their associated operations allow users to interact with your application's functionality as documented in the Swagger view.



API Documentation 1.0.0

[ Base URL: localhost:8090/ ]

http://localhost:8090/v2/api-docs

API documentation for your application

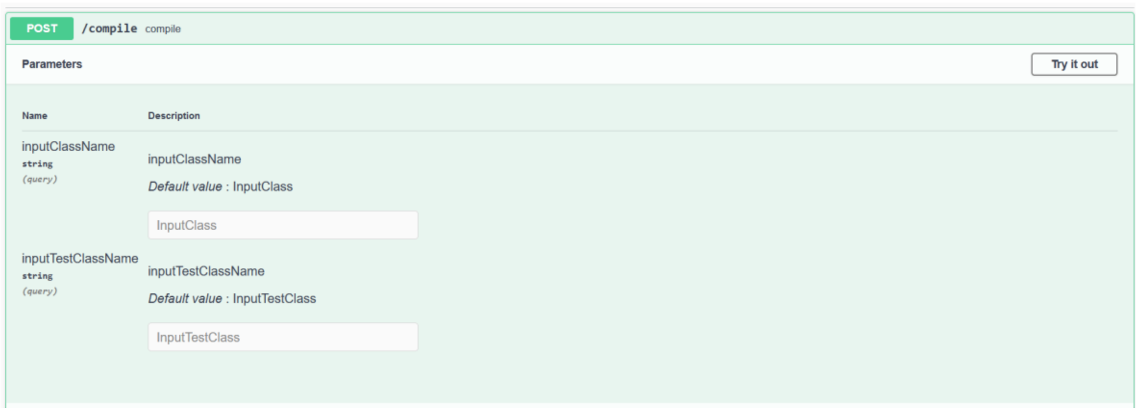
game-controller Game Controller

POST /compile compile

POST /execute execute

home-controller Home Controller

GET / index



inputClassName string (query) — inputClassName — *Default value* : InputClass — InputClass

inputTestClassName string (query) — inputTestClassName — *Default value* : InputTestClass — InputTestClass

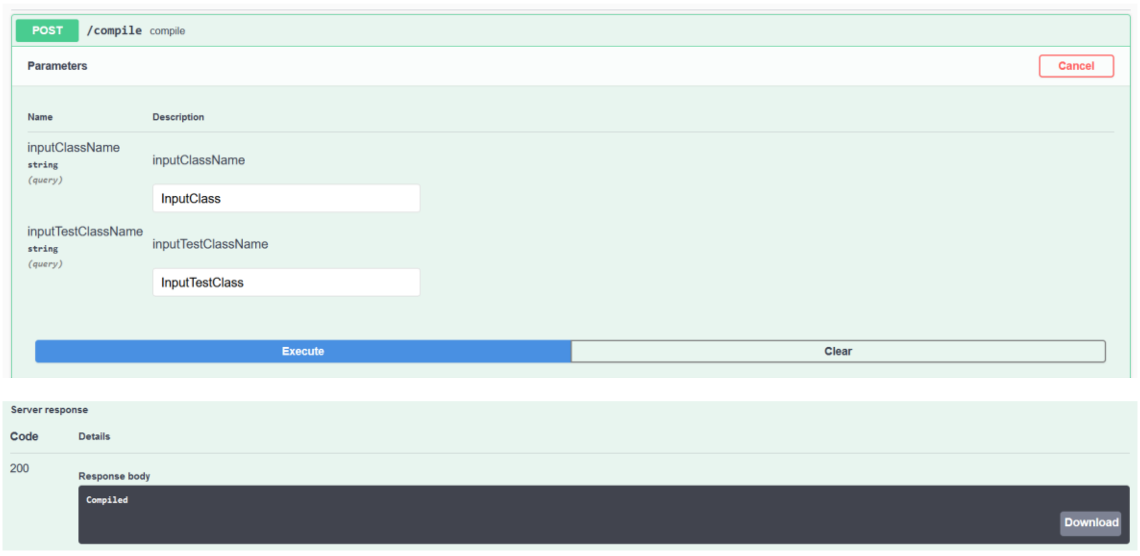
## Application usage

### Swagger

To run it in Docker, you need to follow these steps:

1. Have installed the docker desktop application.
2. Build the Docker image by running the following command in the terminal:
   - docker build -t image-name .
3. Replace image-name with the name you want to give to your Docker image.
4. Once the image is built, run the Docker container with the following command:
   - docker run -p 8090:8090 image-name
5. This command will start the container and map port 8090 of the container to port 8090 of your local machine.
6. Open your preferred web browser and navigate to:
   - http://localhost:8090/swagger-ui.html

- **/compile**:



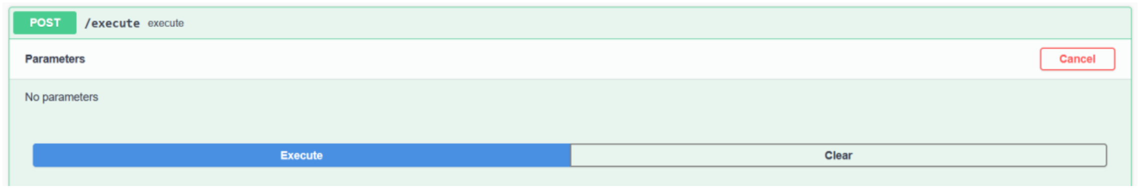
inputClassName string (query) — inputClassName — InputClass

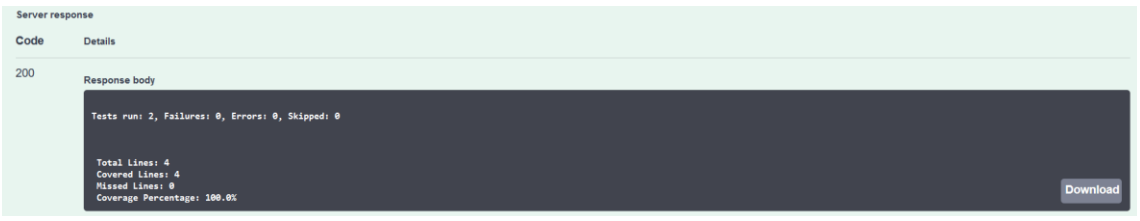
inputTestClassName string (query) — inputTestClassName — InputTestClass

200

```
Compiled
```

- **/execute**:

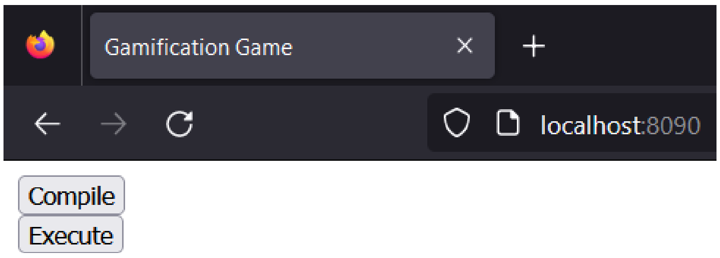




```
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0


 Total Lines: 4
 Covered Lines: 4
 Missed Lines: 0
 Coverage Percentage: 100.0%
```

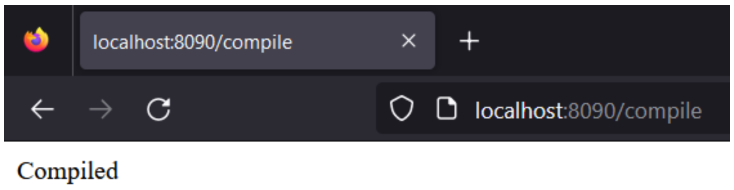
## Browser

To use our application in the browser it must be launched using the **T7Application.java** and then we can go to [http://localhost:8090/](http://localhost:8090/).



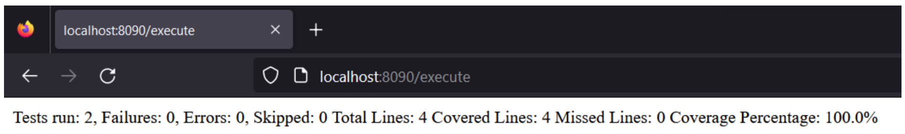
Then we can compile or execute pressing the buttons:

- **/compile**:



```
Compiled
```

- **/execute**:



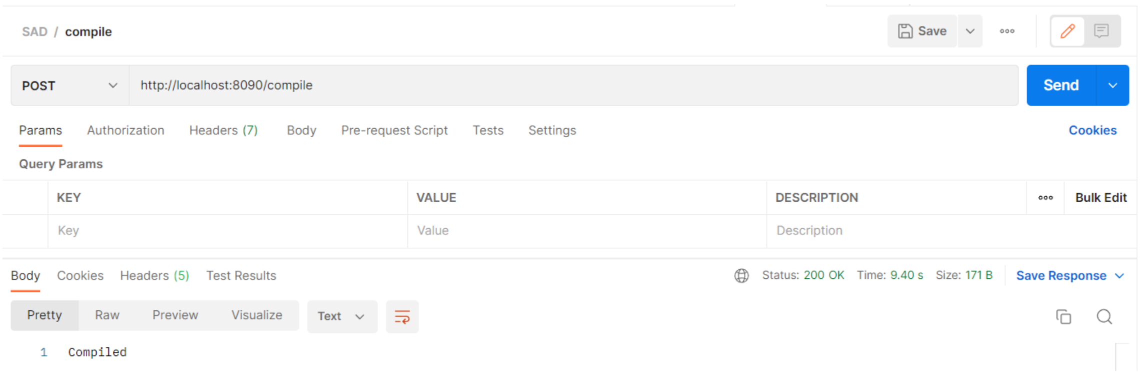
```
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0 Total Lines: 4 Covered Lines: 4 Missed Lines: 0 Coverage Percentage: 100.0%
```

## Postman

Postman is a versatile API development and testing tool to simplify the process of working with APIs. It provides a user-friendly interface that allows developers to design, build, document, and test APIs effortlessly.

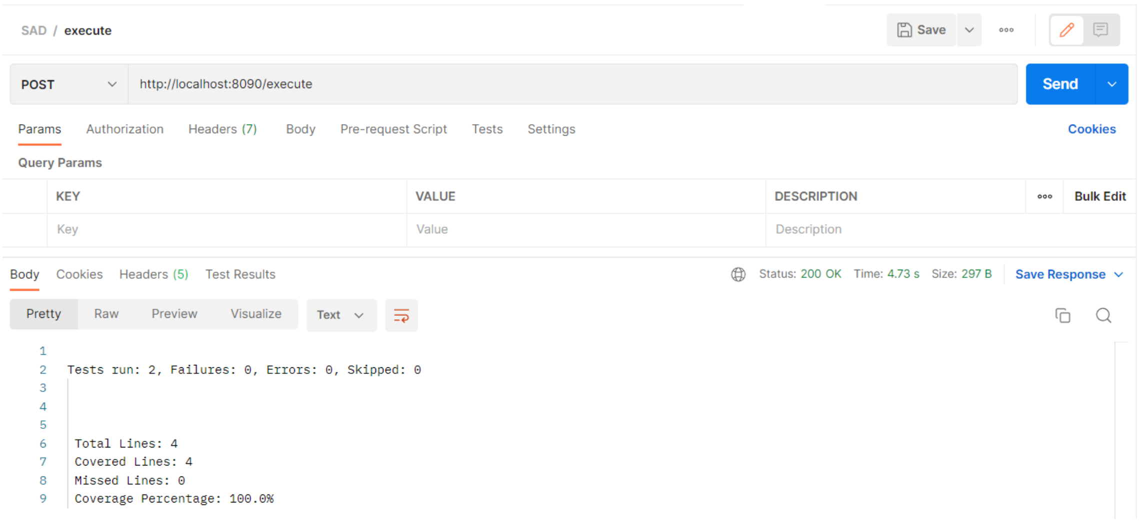
With Postman we can execute the two main endpoints using this configuration, first of all the application has to be launched and then we can use this:

- **/compile**:



```
Compiled
```

- **/execute**:



```

Tests run: 2, Failures: 0, Errors: 0, Skipped: 0



 Total Lines: 4
 Covered Lines: 4
 Missed Lines: 0
 Coverage Percentage: 100.0%
```

## Achieved Results

The project successfully achieved the following results:

1. **Fully Functional Application**:
   - Developed and implemented a fully functional application with comprehensive execution and compile functions.
   - Ensured that the application performs as expected and delivers the intended functionalities, including messages related to error or successful executions.
2. **Code Functionality Tests**:
   - Designed and executed tests to thoroughly validate the functionality of the entire codebase.
   - The tests covered various scenarios and edge cases, ensuring the reliability and correctness of the application.
3. **Generation of logs**:
   - Developed and integrated a logging mechanism within the application.
   - Implemented a custom logger that captures and records details of every execution in a text file.
   - The logger enhances the application's monitoring and debugging capabilities, providing valuable insights into the execution flow and aiding in troubleshooting.
4. **Continuous Integration with SonarCloud**:

- Integrated the project with SonarCloud, a powerful code quality and analysis platform.
- Achieved continuous integration, enabling automated code inspection, detecting issues, and ensuring code quality throughout the development process.

5. **Testing with Codecov**:
   - Leveraged codecov.io for comprehensive code coverage analysis.
   - Conducted tests to measure the extent to which the codebase is exercised, ensuring a high level of test coverage.
6. **Docker with Swagger**:
   - Utilized Docker to containerize the application, ensuring easy deployment and scalability.
   - Integrated Swagger, a popular API documentation tool, to provide clear and interactive documentation for the application's APIs.

By accomplishing these milestones, the project achieved a fully functional application with robust testing, continuous integration, code quality analysis, and documentation, thereby ensuring a reliable and maintainable software solution.