

SOFTWARE PROJECT MANAGEMENT AND EVOLUTION

Sliding Puzzle Documentation



Pedro Zahonero Mangas
Pablo García Fernández

Index

Introduction.....	1
Organization techniques.....	1
Technological background	2
Functionality and technological characteristics	3
Example execution	3
UML Diagrams	4
UML Class diagram	4
UML Sequence diagram	5
UML Use-case diagram.....	6
Code analysis.....	6
Before	6
After.....	7
SlidingPuzzle.java	7
PrincipalWindow.java	8
Logger.java	9
Main.java	10
Test architecture and Developed test cases	11
Maven.....	11
pom.xml.....	11
Build.....	11
Dependencies.....	11
Properties	11
Test Execution.....	11
Continuous Integration.....	12
GitHub Actions	12
Codecov	15
SonarCloud	16
FOSSA	16
Test cases.....	17
TestSuite.java.....	17
TestPrincipalWindow.java.....	17
TestSlidingPuzzle.java	18
Achieved Results	20

Introduction

In this project, our primary focus is to upgrade the existing codebase, which is currently characterized by its lack of sophistication and cleanliness. Our goal is to transform it into an optimal, well-documented, and clean codebase that adheres to industry best practices.

The Sliding Puzzle game is a classic brain-teasing game that challenges players to rearrange numbered tiles within a grid to form a specific pattern or sequence. Our objective is not only to enhance the gameplay and user interface but also to improve the underlying code structure and organization.

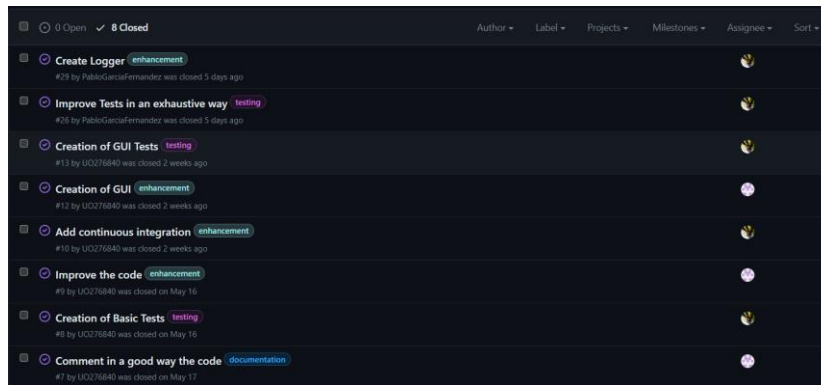
The current state of the codebase has various shortcomings, including poor modularity, limited scalability, and an absence of clear documentation. These issues make it challenging to maintain, extend, and collaborate effectively.

To achieve these goals, we will implement a series of optimizations and code refactoring techniques. This includes improving algorithmic efficiency, eliminating redundant code, enhancing code readability, and applying design patterns where appropriate. Moreover, we will place a strong emphasis on writing comprehensive and well-structured documentation, which will serve as a valuable resource for both current and future developers working on the project.

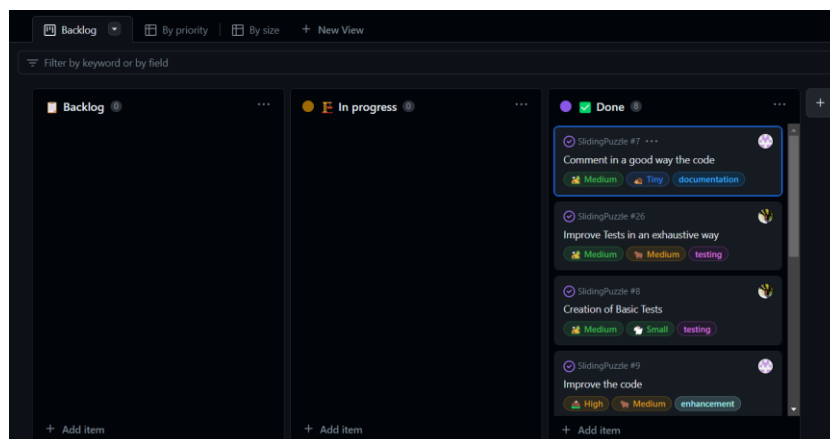
Organization techniques

Effective organization techniques are essential for streamlining the development process and ensuring smooth collaboration among team members. In our project, we have employed various tools and methodologies to enhance organization and communication. This section will outline the techniques we utilized, including Discord, WhatsApp, presential meetings, GitHub Issues, and the Project feature on GitHub.

- **Discord:** It is a communication platform that offers text, voice, and video channels, providing a central hub for team collaboration.
- **WhatsApp:** It is served as an additional communication channel, particularly for quick updates and urgent notifications. We utilized WhatsApp to share important announcements, coordinate schedules, and address immediate concerns efficiently. However, we ensured that critical discussions and decision-making were consolidated on more appropriate platforms, such as Discord or presential meetings.
- **Presential Meetings:** Face-to-face meetings played a crucial role in fostering effective communication and deeper collaboration within the team. We organized regular presential meetings to discuss project progress, set milestones, and brainstorm solutions to challenges.
- **GitHub Issues:** GitHub Issues served as a centralized platform for issue tracking, bug reporting, and feature requests. This streamlined the process of issue triaging and allowed us to prioritize tasks effectively.



- GitHub Project:** The Project feature on GitHub provided a visual representation of our project's workflow and progress. We organized tasks into different columns, such as "Backlog", "In Progress" and "Done," allowing team members to track the status of each task easily. This feature helped us monitor the overall project progress, identify bottlenecks, and distribute workload efficiently. We could see the priority and the weight of each issue.



Technological background

Our software project for the Sliding Puzzle game is built using a combination of technologies that enable efficient development, testing, and deployment. In this section, we will discuss the key technologies utilized, including Java, Maven, Mockito, JaCoCo, JUnit 5, and GitHub Actions.

- Java:** Java is a widely adopted, versatile programming language known for its simplicity, portability, and robustness. We chose Java as the primary programming language for our Sliding Puzzle game due to its extensive libraries, strong community support, and compatibility with various platforms. Java enables us to develop a scalable and platform-independent application that can run on different operating systems.
- Maven:** Maven is a popular build automation tool used for managing project dependencies, compiling source code, and packaging applications. We utilized Maven as our build tool to streamline the development process and simplify dependency management. Maven's declarative approach allows us to define project configurations and dependencies in a standardized manner, facilitating easy project setup and ensuring consistent builds across different environments.

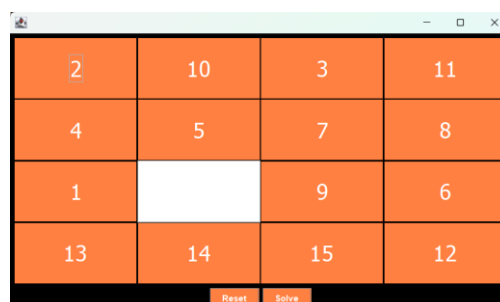
- **Mockito:** Mockito is a widely used Java framework for writing unit tests and creating mock objects. It enables us to isolate dependencies and simulate external components, ensuring focused and reliable unit testing. By utilizing Mockito, we can effectively verify the behavior of our code and increase test coverage, leading to more robust and bug-free software.
- **JaCoCo:** JaCoCo is a code coverage library that measures the extent to which our test suite covers the application's source code. It provides valuable insights into the effectiveness of our testing efforts and helps identify areas of the code that require additional testing. By integrating JaCoCo into our project, we can ensure a high level of code coverage and maintain the quality and reliability of our Sliding Puzzle game.
- **JUnit 5:** JUnit 5 is a powerful testing framework for Java applications. It provides a comprehensive set of features and annotations for writing unit tests, parameterized tests, and test suites. We utilized JUnit 5 to design and execute test cases, validate expected behavior, and ensure the correctness of our code. JUnit 5's modular and extensible architecture empowers us to create comprehensive and maintainable test suites that enhance the overall quality of our software.
- **GitHub Actions:** GitHub Actions is an automation platform integrated with the GitHub repository. It allows us to automate various development workflows, such as building, testing, and deploying our Sliding Puzzle game. By configuring GitHub Actions, we can define custom workflows that automatically trigger specific actions whenever a code change is pushed, or a pull request is submitted. This automation streamlines our development process, improves efficiency, and ensures consistent software builds.

By leveraging these technologies, we establish a strong technological foundation for our Sliding Puzzle game project. Java serves as the core programming language, while Maven handles dependency management and builds. Mockito and JUnit 5 enable us to create comprehensive and reliable test suites, ensuring the correctness of our code. JaCoCo provides insights into code coverage, and GitHub Actions automates key development workflows. Together, these technologies contribute to the development of a high-quality, well-tested, and efficiently managed software solution.

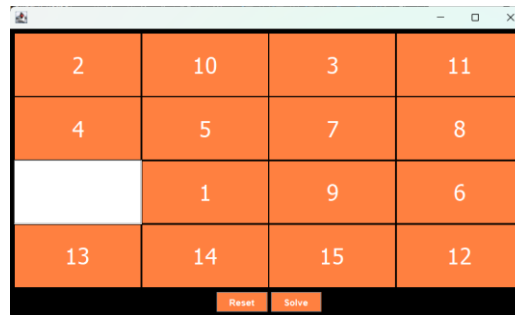
Functionality and technological characteristics

Example execution

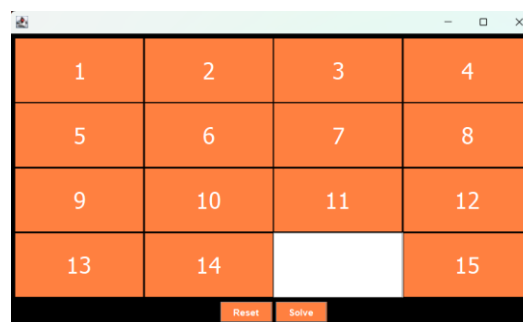
1. The project can be started by executing the Main.java file.
2. Once executed the game will appear in a new window like this:



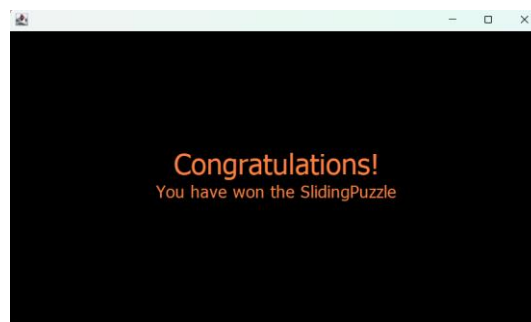
3. The objective of the game is to obtain the numbers in this positions {1,2,3,...,13,14,15,0} where 0 is the blank space. Whenever you click to an adjacent tile to the blank space both will be exchanged, for example now we are clicking the 1:



4. This is how it looks just before solving, we need to change the blank with the 15:



5. After doing it we have won the game:



6. There are also two extra options to reset or solve the puzzle during the game.

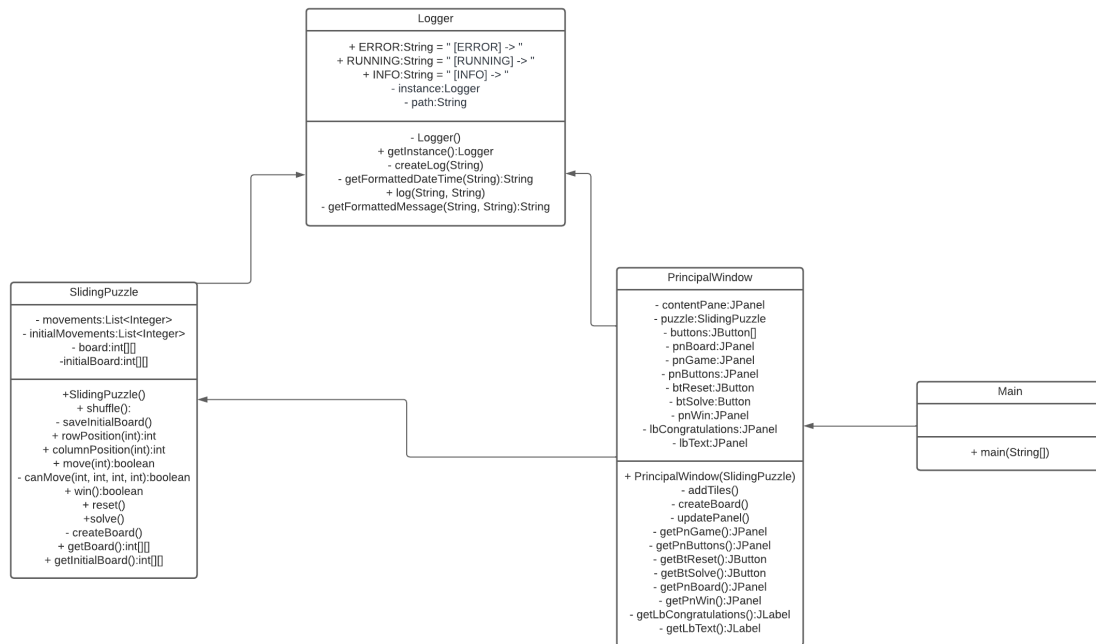
UML Diagrams

To enhance the understanding of the architecture and functionality of our Sliding Puzzle game, we have created three key UML (Unified Modeling Language) diagrams: Class Diagram, Sequence Diagram, and Use Case Diagram. These diagrams serve as visual representations that provide a clear overview of the system's structure, interactions, and user interactions.

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.

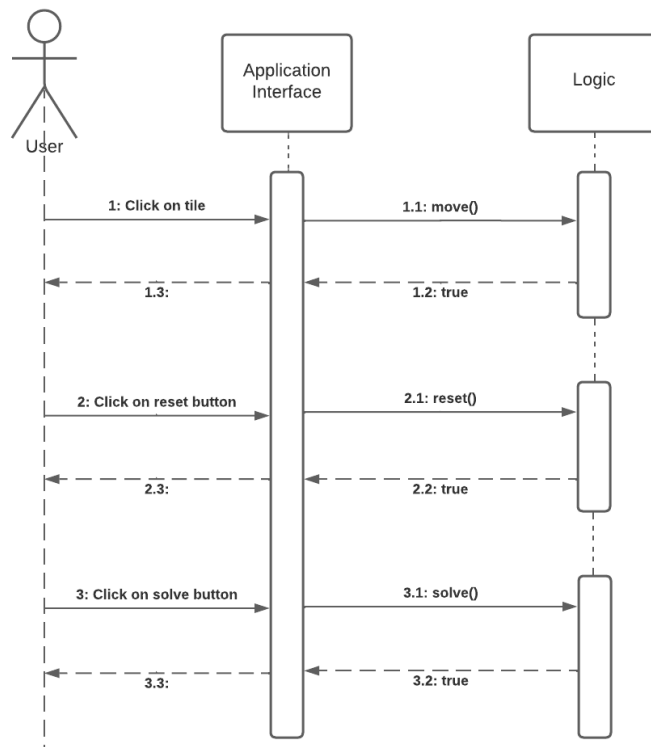
This is the UML class diagram for this project:



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.

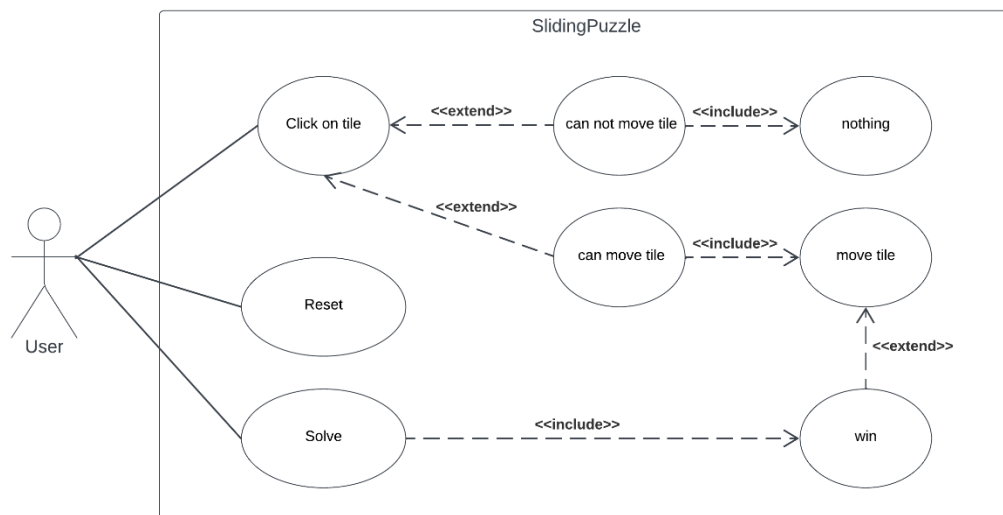
This is the UML sequence diagram of this project:



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:



Code analysis

Before

The project starts with only one class called Prueba3 that is the logic of a sliding puzzle game.

The class contains various methods for shuffling, moving, displaying, and checking the win condition of a sliding puzzle.

The **shuffle()** method takes a 2D array of tiles and shuffles their positions randomly using the Fisher-Yates algorithm.

The **posI()** and **posJ()** methods return the row and column indices of a given number in the tile grid.

The **move()** method takes a 2D array of tiles and a number to move. It checks the adjacent positions of the empty tile (0) and the specified number and swaps their positions if they are adjacent. The method handles various edge cases based on the positions of the empty tile and the specified number.

The **display()** method prints the current state of the tile grid to the console.

The **win()** method checks if the tile grid represents a winning configuration where the numbers are in ascending order with the empty tile (0) in the bottom-right corner.

The **reset()** method creates a new instance of the Prueba3 class, effectively resetting the game.

The **start()** method is the main logic for playing the sliding puzzle. It takes a 2D array of tiles and an array of movements. It displays the initial state of the puzzle, prompts the user to enter a number, moves the tiles accordingly, and checks for a win condition. The user can enter "17" to reset the game or "18" to display the solution.

The code uses JOptionPane for input and message dialogs.

Overall, the Prueba3 class provides functionality for playing a sliding puzzle game, but the code seems to be incomplete or lacking proper structure. This code is hardly readable, maintainable, and inefficient. It has no graphical interface and the reset and solve methods do not work correctly, in addition to not having any test.

After

After the code refactoring, we can see that now we have a readable, maintainable, and efficient code. We have four classes: SlidingPuzzle (logic of the application), PrincipalWindow (graphical user interface), Logger (class to log the activity) and Main. We have also added tests to the project and a version control system (GitHub)

SlidingPuzzle.java

This class represents the logic for a sliding puzzle game.

The package name is logic, and the code is organized into a single class called SlidingPuzzle. The class has two private instance variables: **movements** and **initialMovements**, which are both of type `List<Integer>`. These lists store the movements made during the game. The class also has two 2D array variables: `board` and **initialBoard**, both of type `int[][]`. These arrays represent the current state and the initial state of the game board, respectively.

The constructor **SlidingPuzzle()** initializes the board by calling the **createBoard()** method. It also logs the creation of the instance using a Logger class.

The **shuffle()** method shuffles the tiles on the game board. It generates random numbers to determine the movements and saves them in the **initialMovements** list. The initial state of the board is also saved using the **saveInitialBoard()** method.

The **rowPosition()** and **columnPosition()** methods determine the row and column positions of a given number on the board.

The **move()** method attempts to move a tile to the blank location if possible. It swaps the positions of the tile and the blank tile on the board and adds the movement to the `movements` list.

The **canMove()** method checks if a movement is possible by comparing the positions of the blank tile and the target tile.

The **win()** method checks if the player has won the game by comparing the current state of the board with the winning state.

The **reset()** method resets the game by restoring the initial state of the board and clearing the `movements` list.

The **solve()** method solves the game by reversing the movements stored in the `movements` list and applying them to the board.

The **createBoard()** method initializes the `'board'` array with a sorted sequence of numbers from 1 to 15, with 0 representing the blank tile. It also initializes the `'movements'` and **initalMovements** lists.

The class provides getter methods **getBoard()** and **getInitialBoard()** to retrieve the current state and the initial state of the board.

The code uses a **Logger** class to log messages at different stages of execution, such as the creation of the instance, shuffling, moving, etc. However, the **Logger** class itself is not provided in the given code.

Overall, the code represents the logic for a sliding puzzle game, including methods for shuffling the tiles, making moves, checking for a win, resetting the game, and solving the puzzle.

[PrincipalWindow.java](#)

This class represents the GUI (Graphical User Interface) for a sliding puzzle game.

The package name is `gui`, and the code is organized into a class called `PrincipalWindow`, which extends `JFrame` to create the main window for the game.

The class has several instance variables representing different GUI components such as panels, buttons, labels, and the **SlidingPuzzle** object.

The constructor **PrincipalWindow(SlidingPuzzle puzzle)** initializes the GUI components, sets the layout, and adds the panels to the content pane.

The **addTiles()** method creates the tiles of the game board based on the state of the puzzle object. It creates buttons for each tile and adds them to the **pnBoard** panel. It also assigns an action listener to each button to handle tile movements when clicked.

The **createBoard()** method initializes the game board by shuffling the tiles using the **shuffle()** method of the puzzle object. It calls **addTiles()** to create and add the tiles to the **pnBoard** panel.

The **updatePanel()** method updates the game board by removing all the components from the **pnBoard** panel and calling **addTiles()** to recreate the tiles based on the current state of the puzzle object.

The class includes getter methods for various components, such as **getBtReset()** and **getBtSolve()** for the reset and solve buttons, **getPnBoard()** for the panel containing the game board, and **getPnWin()** for the panel displayed when the player wins.

The **getBtReset()** and **getBtSolve()** methods assign action listeners to the reset and solve buttons, respectively, which call the corresponding methods of the puzzle object and update the game board.

The **getPnGame()** method returns the panel where the game is displayed, and the **getPnButtons()** method returns the panel containing the reset and solve buttons.

The **getPnWin()** method returns the panel displayed when the player wins, containing congratulatory labels.

The GUI is styled with various colors, fonts, and layouts to create an appealing and user-friendly interface.

Overall, the code provides a graphical representation of the sliding puzzle game, allowing the player to interact with the game board through button clicks and providing visual feedback when the game is won.

Logger.java

This class represents a Logger class used for logging messages in the application.

The package name is `logic.util`, and the code is organized into a class called `Logger`.

The class implements a singleton pattern, meaning that only one instance of the `Logger` class can exist.

The class has several instance variables, including `instance` (to hold the singleton instance), and `path` (to store the path of the log file).

The constructor is private, ensuring that the `Logger` class cannot be instantiated directly from outside the class.

The **`getInstance()`** method returns the singleton instance of the `Logger` class. If the instance is null, it creates a new instance and returns it.

The **`createLog(String filePath)`** method is a private helper method that creates the log file specified by the `filePath` parameter. It uses the **`Files.createDirectories()`** method to create the necessary directory structure, and then creates an empty log file using a **`FileWriter`**.

The **`getFormattedDateTime(String type)`** method returns a formatted date and time string based on the `type` parameter. It uses a **`SimpleDateFormat`** object to format the date and time.

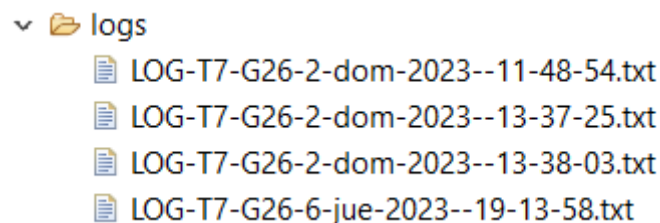
The **`log(String type, String message)`** method is used to log messages. It takes the `type` (e.g., `ERROR`, `RUNNING`, `INFO`) and `message` as parameters. It opens the log file using a **`FileWriter`**, writes the formatted message to the file, and then closes the file.

The **`getFormattedMessage(String type, String message)`** method returns the formatted log message by concatenating the formatted date and time, the `type`, and the `message`.

The `Logger` class follows a simple log format: `[datetime] [type] -> message`, where `[datetime]` is the formatted date and time, `[type]` is the type of the message (e.g., `ERROR`, `RUNNING`, `INFO`), and `message` is the content of the log message.

Overall, the provided `Logger` class allows for logging messages to a log file with timestamps and different message types. It creates the log file if it doesn't exist and appends new log messages to it. It also creates the log file with a formatted name and directory structure.

Folder containing logs:



```

v logs
LOG-T7-G26-2-dom-2023--11-48-54.txt
LOG-T7-G26-2-dom-2023--13-37-25.txt
LOG-T7-G26-2-dom-2023--13-38-03.txt
LOG-T7-G26-6-jue-2023--19-13-58.txt

```

Log output example:

```

1 02/07/2023--11:48:54 [INFO] -> Execution of main() at Main.java
2 02/07/2023--11:48:54 [INFO] -> Creation of instance of SlidingPuzzle.java
3 02/07/2023--11:48:54 [RUNNING] -> Class: SlidingPuzzle.java , method: createBoard()
4 02/07/2023--11:48:54 [INFO] -> Creation of instance of PrincipalWindow.java
5 02/07/2023--11:48:54 [RUNNING] -> Class: PrincipalWindow.java , method: createTiles()
6 02/07/2023--11:48:54 [RUNNING] -> Class: SlidingPuzzle.java , method: shuffle()
7 02/07/2023--11:48:54 [RUNNING] -> Class: SlidingPuzzle.java , method: move()
8 02/07/2023--11:48:54 [RUNNING] -> Class: SlidingPuzzle.java , method: rowPosition()
9 02/07/2023--11:48:54 [RUNNING] -> Class: SlidingPuzzle.java , method: columnPosition()
10 02/07/2023--11:48:54 [RUNNING] -> Class: SlidingPuzzle.java , method: rowPosition()
11 02/07/2023--11:48:54 [RUNNING] -> Class: SlidingPuzzle.java , method: columnPosition()
12 02/07/2023--11:48:54 [RUNNING] -> Class: SlidingPuzzle.java , method: canMove()
13 02/07/2023--11:48:54 [RUNNING] -> Class: SlidingPuzzle.java , method: move()
14 02/07/2023--11:48:54 [RUNNING] -> Class: SlidingPuzzle.java , method: rowPosition()
15 02/07/2023--11:48:54 [RUNNING] -> Class: SlidingPuzzle.java , method: columnPosition()
16 02/07/2023--11:48:54 [RUNNING] -> Class: SlidingPuzzle.java , method: rowPosition()
17 02/07/2023--11:48:54 [RUNNING] -> Class: SlidingPuzzle.java , method: columnPosition()
18 02/07/2023--11:48:54 [RUNNING] -> Class: SlidingPuzzle.java , method: canMove()
19 02/07/2023--11:48:54 [RUNNING] -> Class: SlidingPuzzle.java , method: move()
20 02/07/2023--11:48:54 [RUNNING] -> Class: SlidingPuzzle.java , method: rowPosition()
21 02/07/2023--11:48:54 [RUNNING] -> Class: SlidingPuzzle.java , method: columnPosition()
22 02/07/2023--11:48:54 [RUNNING] -> Class: SlidingPuzzle.java , method: rowPosition()
23 02/07/2023--11:48:54 [RUNNING] -> Class: SlidingPuzzle.java , method: columnPosition()
24 02/07/2023--11:48:54 [RUNNING] -> Class: SlidingPuzzle.java , method: canMove()
25 02/07/2023--11:48:54 [RUNNING] -> Class: SlidingPuzzle.java , method: move()
26 02/07/2023--11:48:54 [RUNNING] -> Class: SlidingPuzzle.java , method: rowPosition()
27 02/07/2023--11:48:54 [RUNNING] -> Class: SlidingPuzzle.java , method: columnPosition()
28 02/07/2023--11:48:54 [RUNNING] -> Class: SlidingPuzzle.java , method: rowPosition()
29 02/07/2023--11:48:54 [RUNNING] -> Class: SlidingPuzzle.java , method: columnPosition()
30 02/07/2023--11:48:54 [RUNNING] -> Class: SlidingPuzzle.java , method: canMove()

```

Main.java

The provided code represents the main entry point of the application. Here's an analysis of the code:

The package name is main, and the code is organized into a class called Main.

The main method is the starting point of the application. It is called when the program is executed.

The main method starts by logging the execution of the main method using the Logger class.

Inside the main method, the application is launched using the **EventQueue.invokeLater()** method. This method ensures that the GUI-related code is executed on the Event Dispatch Thread (EDT) to ensure proper thread-safety.

Inside the **run()** method of the Runnable interface, the following steps are performed:

An instance of the **SlidingPuzzle** class is created.

An instance of the **PrincipalWindow** class is created, passing the **SlidingPuzzle** instance as a parameter.

The **setVisible(true)** method is called on the frame object to display the application window.

If any exception occurs during the execution of the application, it is caught in the catch block, and the error message is logged using the Logger class.

Overall, the Main class serves as the entry point of the application. It initializes the necessary components, including the **SlidingPuzzle** object and the main application window (PrincipalWindow), and starts the GUI event loop to display the application to the user.

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

- **<sourceDirectory>**: Specifies the directory containing the main Java source code (src/main/java).
- **<plugins>**: Contains the list of plugins used for the build process.
- **maven-compiler-plugin**: Configures the Maven Compiler Plugin to set the Java release version to 17.
- **maven-surefire-plugin**: Configures the Maven Surefire Plugin to execute tests, specifically including the TestSuite.java file.
- **jacoco-maven-plugin**: Configures the JaCoCo Maven Plugin, which enables code coverage analysis. It excludes the main source code (**/main/*) from the coverage report. It defines two executions: prepare-agent to set up JaCoCo agent, and report to generate the coverage report during the test phase.

Dependencies

- **junit-jupiter-api**: The JUnit Jupiter API dependency is used for writing and executing JUnit 5 tests.
- **junit-platform-suite**: The JUnit Platform Suite dependency enables the creation of test suites in JUnit 5.
- **mockito-core**: The Mockito Core dependency provides support for creating mock objects and performing mocking operations during testing.

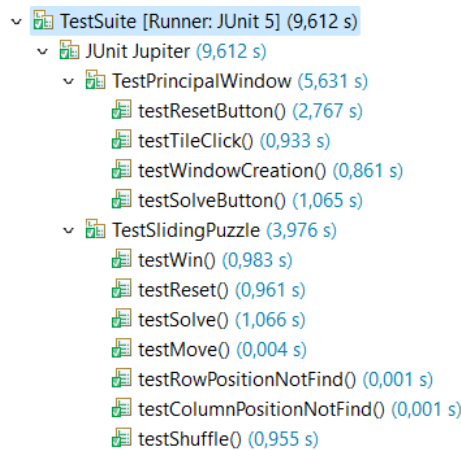
Properties

- **sonar.organization**: Specifies the organization key for SonarCloud integration.
- **sonar.projectKey**: Sets the project key for SonarCloud integration.
- **sonar.host.url**: Specifies the URL of the SonarCloud instance.
- **sonar.branch**: Sets the branch to be analyzed by SonarCloud.
- **maven.javadoc.skip**: Skips the generation of Javadoc during the build process.

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 the Eclipse IDE.
 1. In the package explorer right click on the TestSuite class.
 2. Select the option Run As JUnit test.
 3. This should be the output:



- 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] -----
[INFO] T E S T S
[INFO] -----
[INFO] Running tests.TestSuite
[INFO] Running tests.gui.TestPrincipalWindow
Java HotSpot(TM) 64-Bit Server VM warning: Sharing is only supported for boot loader classes because bootstrap classpath
has been appended
File 'logs/LOG-T7-G26-6-jue-2023--20-08-54.txt' created successfully.
[INFO] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 7.949 s - in tests.gui.TestPrincipalWindow
[INFO] Running tests.logic.TestSlidingPuzzle
[INFO] Tests run: 7, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 4.215 s - in tests.logic.TestSlidingPuzzle
[INFO] Tests run: 0, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 12.365 s - in tests.TestSuite
[INFO] Results:
[INFO] Tests run: 11, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] --- jacoco-maven-plugin:0.8.8:report (report) @ SlidingPuzzle ---
[INFO] Loading execution data file C:\Users\pablo\git\SlidingPuzzle\target\jacoco.exec
[INFO] Analyzed bundle 'SlidingPuzzle' with 6 classes
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 17.684 s
[INFO] Finished at: 2023-07-06T20:09:04+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 we are using is called ci.yml and it has this code.

```

1  name: Workflow for SPME-SlidingPuzzle
2
3  on:
4    push:
5      branches:
6        - master
7    pull_request:
8      types: [opened, synchronize, reopened]
9
10 jobs:
11   run:
12     runs-on: windows-latest
13     steps:
14       - name: Checkout
15         uses: actions/checkout@v3
16         with:
17           fetch-depth: 0
18       - name: Set up JDK 17
19         uses: actions/setup-java@v1
20         with:
21           java-version: 17
22       - name: Install dependencies
23         run: mvn install -DskipTests=true -B -V
24       - name: Cache Maven packages
25         uses: actions/cache@v3
26         with:
27           path: ~/.m2
28           key: ${{ runner.os }}-m2-${{ hashFiles('**/pom.xml') }}
29           restore-keys: ${{ runner.os }}-m2
30       - name: Build and analyze
31         env:
32           GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }} # Needed to get PR information, if any
33           SONAR_TOKEN: ${{ secrets.SONAR_TOKEN }}
34         run: mvn -B verify org.sonarsource.scanner.maven:sonar-maven-plugin:sonar
35       - name: Upload coverage to Codecov
36         uses: codecov/codecov-action@v3
37         with:
38           token: ${{ secrets.CODECOV_TOKEN }}
39           fail_ci_if_error: true
40           verbose: true

```

This GitHub Actions workflow is designed to automate the build, analysis, and code coverage reporting process for the SPME-SlidingPuzzle project.

- **name:** Specifies the name of the workflow, which is "Workflow for SPME-SlidingPuzzle."
- **on:** Defines the triggering events for the workflow.
 - **push:** Triggers the workflow when there is a push to the master branch.
 - **pull_request:** Triggers the workflow when a pull request is opened, synchronized, or reopened.
- **jobs:** Contains the list of jobs to be executed as part of the workflow.
 - **run:** Represents the main job named "run," which runs on the latest version of Windows.
- **runs-on:** Specifies the operating system environment for the job, which is windows-latest.

- **steps:** Contains a list of steps to be executed within the job.
 - **Checkout:** Checks out the source code using the actions/checkout action.
 - **Set up JDK 17:** Sets up JDK 17 using the actions/setup-java action.
 - **Install dependencies:** Installs project dependencies using the Maven command `mvn install -DskipTests=true -B -V`.
 - **Cache Maven packages:** Caches Maven packages in the `~/.m2` directory using the actions/cache action. Caching is done based on the content hash of the `pom.xml` file.
 - **Build and analyze:** Builds the project and performs analysis using SonarCloud. It uses the Sonar Maven plugin and SonarCloud tokens specified as environment variables.
 - **Upload coverage to Codecov:** Uploads the code coverage report to Codecov using the codecov/codecov-action action. It uses the Codecov token specified as a secret.

The workflow is triggered on push events to the master branch and on pull request events (opened, synchronized, and reopened). It checks out the code, sets up JDK 17, installs dependencies, caches Maven packages, builds and analyzes the project using SonarCloud, and uploads the code coverage report to Codecov.

This workflow enables continuous integration and automated quality analysis for the SPME-SlidingPuzzle project, ensuring that code changes are thoroughly tested, analyzed, and assessed for code coverage.

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

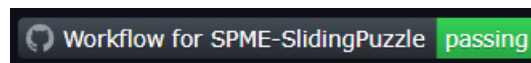


Photo of the last workflow in the repo:

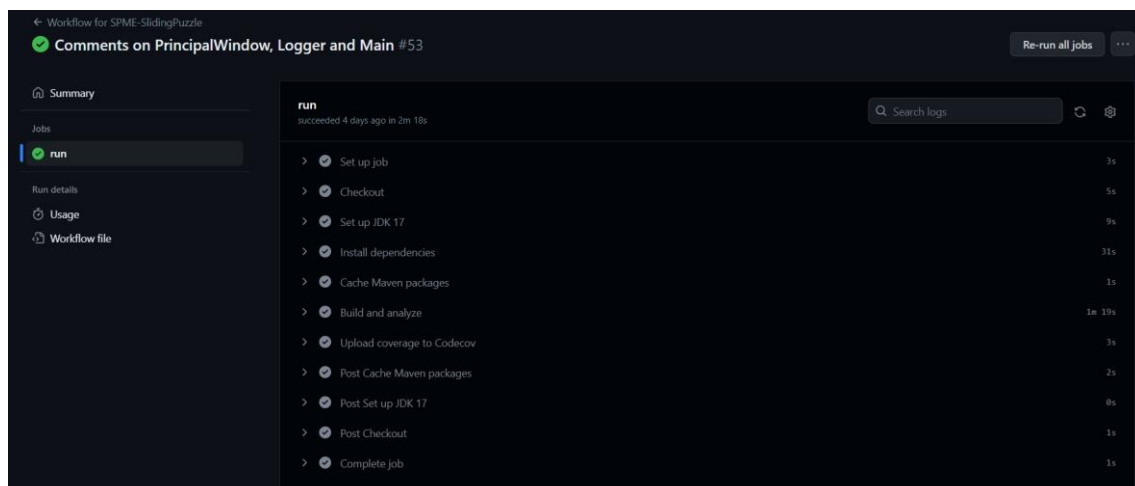


Photo of how it looks like a pull request:

Update README.md #31 Edit <> Code

Merged PabloGarciaFerna... merged 1 commit into `main` from `PabloGarciaFernandez-patch-1` 5 days ago

Conversation 2 · Commits 1 · Checks 5 · Files changed 1 +1 -1

PabloGarciaFernandez commented 5 days ago Owner ...

No description provided.

Update README.md Verified ✓ a7434d1

PabloGarciaFernandez requested a review from **UO276840** 5 days ago

UO276840 approved these changes 5 days ago View reviewed changes

sonarcloud bot commented 5 days ago ...

Kudos, SonarCloud Quality Gate passed! Passed

- 0 Bugs
- 0 Vulnerabilities
- 0 Security Hotspots
- 0 Code Smells
- No Coverage information
- No Duplication information

Milestone
No milestone

Development
Successfully merging this pull request may close these issues.

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

codecov bot commented 5 days ago ...

Codecov Report

Patch coverage has no change and project coverage change: **+2.07**

Comparison is base `(cb6db30)` 91.70% compared to head `(a7434d1)` 93.77%.

► Additional details and impacted files

[View full report in Codecov by Sentry.](#)

Do you have feedback about the report comment? [Let us know in this issue.](#)

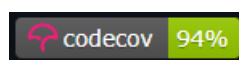
PabloGarciaFernandez merged commit `dc802b6` into `main` 5 days ago View details Revert
6 checks passed

Pull request successfully merged and closed Delete branch
You're all set—the `PabloGarciaFernandez` branch can be safely deleted.

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:



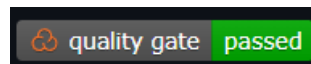
This is the view inside Codecov:



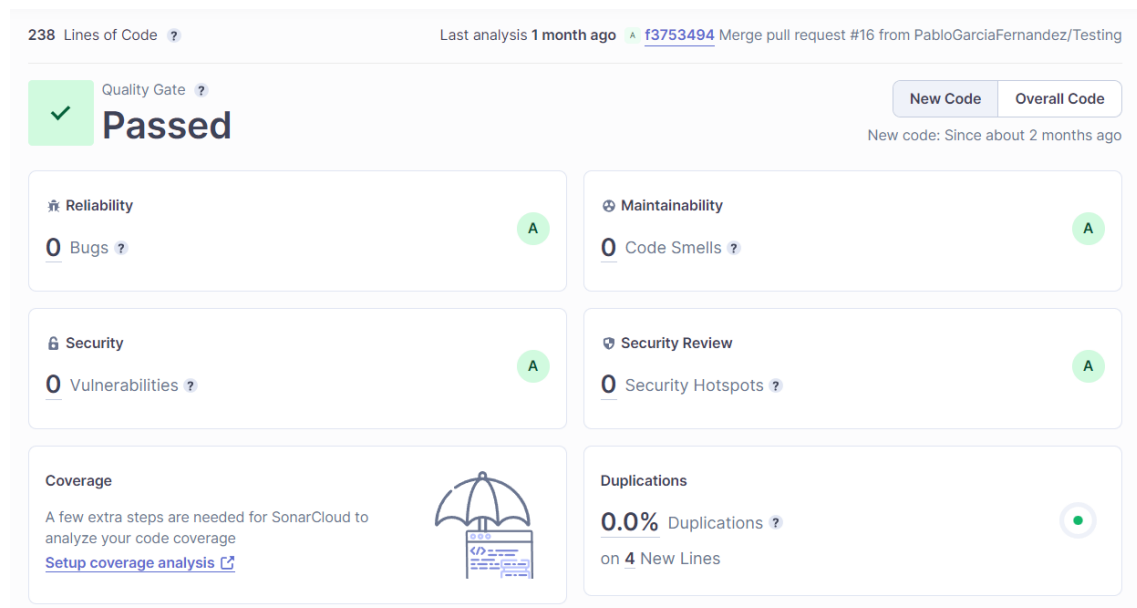
SonarCloud

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:



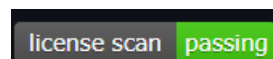
This is the view inside Sonarcloud:



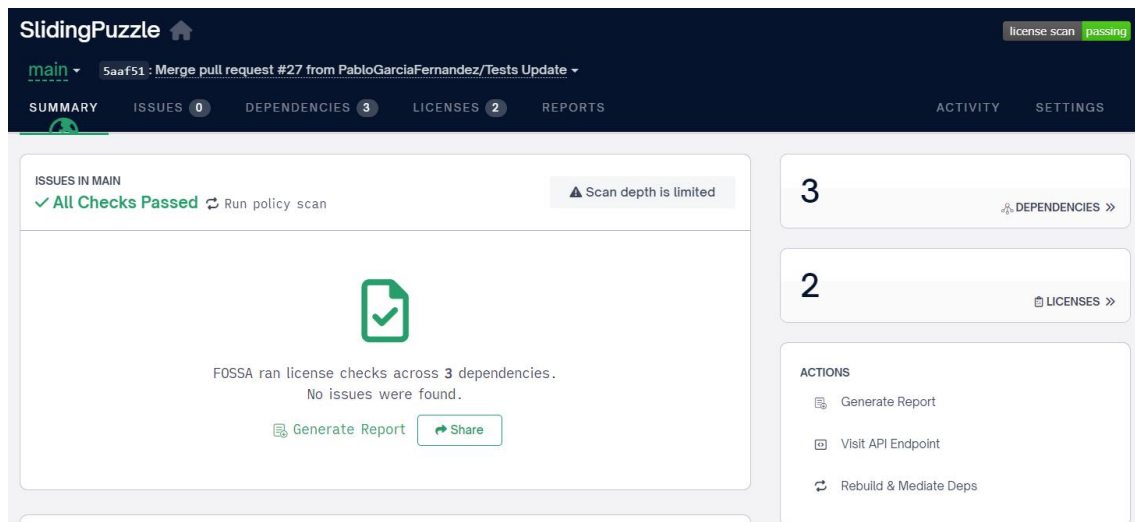
FOSSA

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:



This is the view inside FOSSA:



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.

```
@Suite
@SelectClasses({ TestPrincipalWindow.class, TestSlidingPuzzle.class })
public class TestSuite {

}
```

TestPrincipalWindow.java

Test class in charge of testing PrincipalWindow.java class.

All developed test use Mockito in order to be able to be executed in the workflow because it is a headless container.

Developed tests:

- @BeforeEach **setUp()**: Before each it creates a new SlidingPuzzle and a PrincipalWindow.

```
@BeforeEach
void setUp() {
    MockitoAnnotations.openMocks(this);
    puzzle = new SlidingPuzzle();
    window = new PrincipalWindow(puzzle);
}
```

- @Test **testWindowCreation()**: Test that checks if a window is created.

```
@Test
void testWindowCreation() {
    assertNotNull(window);
}
```

- @Test **testResetButton()**: Test that checks if the reset button works.

```

@Test
void testResetButton() {
    window.getBtReset().doClick();
    // Verify that the puzzle is reset
    assertFalse(puzzle.win());
}

```

- @Test **testSolveButton()**: Test that checks if the solve button works properly.

```

@Test
void testSolveButton() {
    window.getBtSolve().doClick();
    // Verify that the puzzle is solved
    assertTrue(puzzle.win());
}

```

- @Test **testTileClick()**: Test that checks if when clicking a tile it works.

```

@Test
void testTileClick() {
    // Simulate a tile click
    ((AbstractButton) window.getPnBoard().getComponent(0)).doClick();
    assertFalse(puzzle.win());
}

```

TestSlidingPuzzle.java

Test class in charge of testing SlidingPuzzle.java class.

Developed tests:

- @BeforeEach **init()**: Before each test a new SlidingPuzzle is created.

```

@BeforeEach
void init() {
    puzzle = new SlidingPuzzle();
}

```

- @Test **testShuffle()**: Test that checks that shuffle works.

```

@Test
void testShuffle() {
    int board[][] = puzzle.getBoard();
    puzzle.shuffle();

    // Check that the board is in a shuffled state.
    assertNotEquals(board, puzzle.getInitialBoard());
}

```

- @Test **testMove()**: Test that check move a tile.

```

@Test
void testMove() {
    puzzle.move(15);

    // Check that the blank tile has moved to the left.
    assertEquals(15, puzzle.getBoard()[3][3]);
    assertEquals(0, puzzle.getBoard()[3][2]);
}

```

- @Test **testWin()**: Test that checks the win conditions.

```

@Test
void testWin() {
    puzzle.shuffle();
    puzzle.solve();
    assertTrue(puzzle.win());
}

```

- @Test **testReset()**: Test that checks that the board has been reset.

```

@Test
void testReset() {
    puzzle.shuffle();
    for (int i = 1; i <= 15; i++) {
        puzzle.move(i);
    }
    puzzle.reset();

    // Check that the board has been reset to the initial state.
    assertEquals(puzzle.getBoard(), puzzle.getInitialBoard());
}

```

- @Test **testSolve()**: Test that checks the solve works.

```

@Test
void testSolve() {
    puzzle.shuffle();
    puzzle.solve();

    // Check that the game has been solved.
    assertTrue(puzzle.win());
}

```

- @Test **testRowPositionNotFind()**: Check that tries to break the application.

```

@Test
void testRowPositionNotFind() {
    assertEquals(-1, puzzle.rowPosition(100));
}

```

- @Test **testColumnPositionNotFind()**: Check that tries to break the application.

```

@Test
void testColumnPositionNotFind() {
    assertEquals(-1, puzzle.columnPosition(100));
}

```

Achieved Results

Throughout the course of our project to upgrade the Sliding Puzzle game's codebase, we have successfully achieved several notable results. These outcomes are a testament to our commitment to optimization, documentation, and clean code practices. Let's explore the achievements we have accomplished:

1. **Optimal Codebase:** We have transformed the previous unoptimized and inefficient code into an optimal codebase. Through careful refactoring and optimization techniques, we have improved algorithmic efficiency, eliminated redundant code, and enhanced overall performance. The optimized code ensures a smoother and more responsive gameplay experience for users.
2. **Well-documented Code:** Our commitment to clean code practices includes comprehensive documentation. We have extensively documented the codebase, providing clear explanations of classes, methods, and their functionalities. The documentation serves as a valuable resource for current and future developers, facilitating easier understanding, maintenance, and collaboration on the project.
3. **Clean Code Practices:** We have diligently followed clean code practices, such as meaningful variable and method names, proper code formatting, adherence to coding standards, and the elimination of code smells. By implementing clean code practices, we have improved code readability, reduced technical debt, and made the codebase more maintainable and extensible.
4. **Generation of logs:** We have 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.
5. **Test Coverage and Quality Assurance:** We have achieved high test coverage by implementing a comprehensive suite of unit tests using JUnit 5 and Mockito. The test suite covers critical code paths and ensures the correctness and reliability of the Sliding Puzzle game. Through continuous integration and code analysis with tools like SonarCloud, we have maintained a high level of code quality and reduced the likelihood of bugs and regressions.
6. **Automation and Workflow Efficiency:** By leveraging GitHub Actions, we have automated various aspects of our development workflow. Continuous integration, code analysis, and code coverage reporting are seamlessly integrated into our development process. This automation has improved the efficiency of our workflow, allowing for faster feedback loops, quicker bug identification, and smoother collaboration among team members.
7. **Integration with External Services:** We have successfully integrated our project with external services like SonarCloud and Codecov. SonarCloud provides continuous code quality analysis, identifying potential issues, and suggesting improvements. Codecov allows us to track and monitor code coverage, ensuring that our tests effectively cover the codebase. These integrations contribute to the overall quality and reliability of our Sliding Puzzle game.

The achieved results demonstrate our commitment to transforming the Sliding Puzzle game's codebase into an optimal, well-documented, and clean code solution. The optimized code, comprehensive documentation, adherence to clean code practices, high test coverage, automation, and integration with external services collectively enhance the quality, maintainability, and user experience of the Sliding Puzzle game. We are proud of the progress made and the successful outcomes achieved throughout the project.