Project Report on an MVC Application:

Integrating Test-Driven Development, Automated Builds, and Continuous Integration

Project Title:	Order Worker Management System
Project Repository:	https://github.com/MeIbtihajnaeem/Project-Test-Driven-Development
Developed By:	Muhammad Ibtihaj Naeem
Email:	muhammad.naeem@edu.unifi.it
Matriculation Number:	7128359

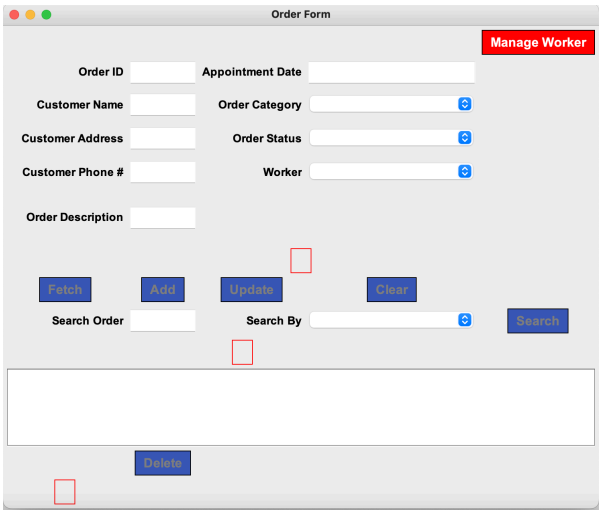
Introduction:

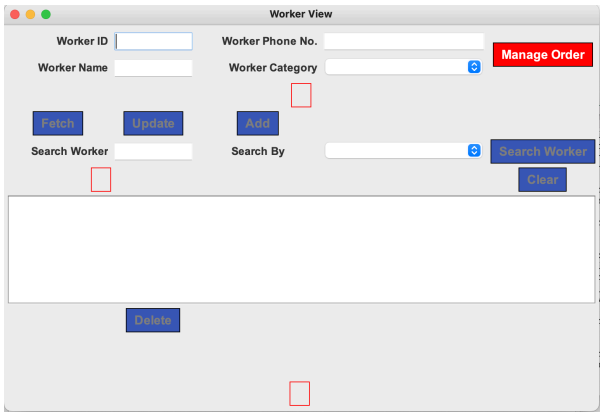
This project report details the creation of a software application utilizing the **Model-View-Controller (MVC)** architecture and incorporating the repository pattern through **Test-Driven Development**, alongside **Build Automation** and **Continuous Integration.**This project utilizes the tools and techniques described in the book "Test-Driven Development, Build Automation, Continuous Integration with Java, Eclipse and Friends" by Professor **Lorenzo Bettini.**The primary objective was to develop a robust, easily maintainable, and scalable application using modern development practices and tools. The project uses a range of technologies like **Eclipse, Maven, Docker, GitHub Actions,** and **SonarCloud**. It emphasizes thorough testing through **white-box (unit and integration tests)** and **black-box (end-to-end tests)** testing methodologies.

The "**Order Worker Management System**" facilitates the management of orders and workers, establishing relationships between them through **one-to-one** and **one-to-many** connections. Each order can be assigned to a single worker, while a worker may handle multiple orders. However, certain criteria govern the interaction between orders and workers:

1. **Order and worker categories must align** (e.g., both categorized as "Electrician").
2. **Orders are initiated with a "pending" status.**
3. **An order cannot be assigned to a worker if the worker already has a pending order.**

The main features of the project include **adding, updating, fetching, and deleting orders and workers.** Additionally, it offers a **search function with multiple options** such as ID, name, phone number, etc.





Project Structure:

Directory Structure:

Src

- **main**
  - **java**: This directory contains all the primary code of the project.
  - **resources**: This directory contains only the log4j configuration file.
- **test**
  - **java**: This directory contains all the unit tests for the project.
  - **resources:** This file contains log4j configuration and persistence.xml for postgresql database
- **it**: This directory holds the Docker database container configuration for both integration and end-to-end tests. It also contains all the integration tests.
- **e2e**: This directory contains all end-to-end tests using JUnit.
- **bdd**:
  - **Java:** This directory contains all end-to-end tests written in Cucumber and BDD style.
  - **Resources:** This directory contains all the feature files for Cucumber

Primary Code Structure:

The code is organized into **models, views, controllers, repositories, utilities, enumerations,** and the **main class** along with resources directory. Here's a detailed breakdown:

Java Code Structure:

Main Class:

OrderWorkerAssignmentSwingApp is the main class of the project. To run the main method, you must provide command line arguments for database configuration (user, password, host, database, port, etc.). If not provided, default values will be used, and the database connection may fail. Notably, the password value must be provided as it was removed by SonarCloud for security reasons.

Enumerations:

a. **OrderCategory**: Used in Worker and CustomerOrder models for assigning categories.
b. **OrderStatus**: Used in CustomerOrder model to assign each order a status.
c. **OperationType**: Used in the controller for adding or updating, as the same method handles both operations.
d. **OrderSearchOptions**: Used for searching orders in the repository and controller.
e. **WorkerSearchOptions**: Used for searching workers in the repository and controller.

Models: (Entities Used in the project)

Worker

1. **Attributes**: Worker Id, Worker Name, Worker Phone Number, Worker Category, List of Customer Orders.
2. **Enumerations**: OrderCategory (Category of the worker, data type is **OrderCategory**).
3. **Relationships**:
    a. WorkerId is the auto incrementing primary key for worker
    b. A worker can handle multiple orders (One to many relation)
    c. A worker cannot be assigned a new order if they have a pending one
    d. Worker phone numbers are unique.

CustomerOrder:

1. **Attributes**: (Order Id, Customer Name, Customer Address, Customer Phone Number, AppointmentDate, Order Description, Order Category, Order Status, Worker)
2. **Enumerations**: OrderCategory, OrderStatus.
3. **Relationships**:
    a. OrderId is the auto incrementing primary key for CustomerOrder
    b. Each order can have only one worker (One to one relation)
    c. Orders are initiated with a "pending" status
    d. Orders and worker categories must align (e.g., both categorized as "Electrician").

Repositories: (Database Repository)

Interfaces

1. OrderRepository
2. WorkerRepository

Java Classes Implementing Repositories:

1. OrderDataRepository (implements OrderRepository)
2. WorkerDataRepository (implements WorkerRepository)

Views:

Interfaces

1. OrderView
2. WorkerView

JFrame Classes Implementing Views

1. OrderSwingView (**JFrame** class implementing OrderView)
2. WorkerSwingView (**JFrame** class implementing WorkerView)

Utils:

The utils include an interface and a Java class that implements this interface. The **Worker** and **CustomerOrder** models contain some common fields that require validation in multiple places, such as validating an ID when adding or deleting an entity in the controller, therefore common validations are not part of the controller or model. To handle these common validations, methods are defined in the **ValidationConfigurations** interface. The design is loosely coupled, allowing for flexibility and interchangeability of implementations. Additionally, the use of dependency injection ensures that **ValidationConfigurations** can be easily integrated and utilized, especially when injecting mocks, ensuring consistent validation across different parts of the application.

a. **Interface:** ValidationConfigurations
b. **Java Class Implementing Interface:** ExtendedValidationConfigurations (implements ValidationConfigurations)

Controllers:

This Controller class accepts **Repository, View, and ValidationConfigurations** interfaces in the constructor, promoting loose coupling and dependency injection. This design allows any class implementing these interfaces to be assigned seamlessly.The controller uses **Log4j** for logging, with the resource files located in **src/main/resources** and **src/test/resources.**

a. **WorkerController:** accepts WorkerView, WorkerRepository, and ValidationConfigurations as parameters.
b. **OrderController:** accepts OrderRepository, OrderView, WorkerRepository, and ValidationConfigurations as parameters.

Resources Code Structure:

This directory has only 1 file named **log4j2.xml** responsible for showing logs while running main code.While the persistence.xml file is not part of github repository since it is sensitive file therefore main method accepts arguments and provide these arguments to entity manager factory as properties for database connection.

Test Code,Resource structure & Strategies:

This project is developed using **Test Driven Development (TDD)** while adhering to the **Test Pyramid**. It includes three types of tests: **unit tests, integration tests,** and **end-to-end tests.** In this project, I demonstrate two types of end-to-end tests: one using **JUnit** with AssertJ, and the other using **Cucumber** in a BDD (Behavior Driven Development) style. For a better understanding of these tests, please refer to the header comments in each test file.

Test Resource structure:

This directory has 2 files 1 for logs **log4j2.xml** responsible for showing logs while running tests. The other file is **persistence.xml** which holds the configuration for the database which is later used by **JPA** during **IT tests and End-to-End tests along with Unit tests for database repository.**

Testing strategy for Unit tests:

Unit tests consist of five types: **utils tests, controller tests, repository tests, Swing view tests,** and **race condition tests.** Each test follows a structured approach with three main phases: **Setup, Exercise, and Verify.** Additionally, I've included a fourth phase, **Mocks**, to improve the readability of the tests. All these tests achieve **100% code coverage** with **no surviving mutants.**

Utils Unit Tests:

The **Utils** unit tests were the first tests I wrote using **TDD** and **the Transformation Priority Premise**. This is because the **Utils** class is responsible for validating everything in the project. These tests verify each

validate method in the **SUT ExtendedValidationConfigurations**. The Utils tests include unit tests for all validation methods in ExtendedValidationConfigurations, utilizing the **AssertJ** library.

Controller Unit Tests:

Controller tests include unit tests for both the **worker and order controllers** using the **Mockito** library. The **OrderControllerTest** and **WorkerController Test** class has the **SUT** as **OrderController** and **WorkerController** respectively with mocked dependencies such as OrderRepository,WorkerRepository, OrderView, WorkerRepository, WorkerView, and ValidationConfigurations. These tests verify whether the controller calls the corresponding view method when a controller method is invoked.

Swing View Unit Tests:

Swing view tests include unit tests for both order and worker Swing JFrame views, utilizing the **AssertJ**, **Mockito**, and **AssertJSwingJUnitTestCase** libraries. These tests verify several aspects, including initial state verifications of labels, text fields, combo boxes, and buttons, as well as the enabling and disabling of the **Add, Update, Fetch, Search, Delete,** and **Clear buttons** based on user input. They also ensure proper delegation of user actions to the **OrderController** and **WorkerController** for tasks such as adding, updating, fetching, searching, and deleting orders. Additionally, these tests confirm the correct display of orders and workers in UI components like lists and combo boxes, and the appropriate showing of error messages in the error labels.

Repository Unit Tests:

Unit tests for the **worker and order database repositories** use the **H2 in-memory database** with **PostgreSQLDialect**, as there is no PostgreSQL Java driver similar to the **Mongo Java Driver** described in the book. These tests utilize the **persistence.xml** file located in the tests/resource directory for configuration. However, the rest of the information for the **H2 database** is provided in the **@Before** method of the test class as properties to the entity manager factory. These tests then use a local database to mimic a real PostgreSQL database. They verify all **CRUD operations**, along with other **search methods** and behaviors of the database on these inputs.

Race Condition Unit Tests:

OrderControllerRaceConditionTest and **WorkerControllerRaceConditionTest:** These tests ensure the **OrderController** and **WorkerController** functions correctly in concurrent environments, verifying that the application handles race conditions properly when multiple threads access and modify order and worker data simultaneously. The tests use Mockito for mocking dependencies and **Awaitility** for handling asynchronous operations. The methods tested include **createOrUpdateOrder**() and **createOrUpdateWorker**() for concurrent order and worker creation, respectively, and **deleteOrder**() and **deleteWorker**() for concurrent order and worker deletion.

Testing strategy for Integration and End-to-End Tests:

The most crucial component in the **integration tests** directory is the **DatabaseConfig** Java class. This class is essential because the project offers **two methods for starting PostgreSQL Docker containers** one **with test containers** and the other **without test containers** directly starting a postgresql docker container. The first method is useful for starting the Docker container during development using an IDE like **Eclipse**, while the second method is beneficial for running integration tests and end-to-end tests with a **Maven profile**. The DatabaseConfig Java class is vital because it is located in the integration tests

directory, yet end-to-end tests also utilize this class for database configuration. This setup ensures code duplication is avoided. This project includes two types of end-to-end tests: one set using **JUnit** with AssertJ, and the other using **Cucumber** in a BDD (Behavior Driven Development) style. Both sets cover the same test scenarios.These tests include setting up and tearing down the test environment using Docker containers and database configurations.

Integration tests:

Integration tests are divided into four types: **controller, MVC, race condition,** and **view integration tests.**

Controller IT Tests:

These tests verify the integration between the **OrderController** and **WorkerController** and their respective dependencies, including **OrderRepository**, **WorkerRepository**, **OrderView**, **WorkerView**, and **ValidationConfigurations**. The tests cover various scenarios such as creating, updating, fetching, deleting, and searching orders and workers. These tests verify that when a controller method is called, it correctly displays or modifies data in the view.

View IT Tests:

Integration tests for the **OrderSwingView** and **WorkerSwingView** classes cover a range of functionalities. They ensure the correct operation of creating, updating, fetching, searching, and deleting customer orders and workers through the graphical user interface. The tests verify both successful and failed operations for these actions, ensuring proper validation and error handling for invalid input data. Additionally, the tests utilize the **AssertJSwingJUnitTestCase** framework for GUI testing and **Awaitility** for asynchronous operations.

Model View Controller IT Tests:

Integration tests for the **OrderModelViewControllerIT** and **WorkerModelViewControllerIT** classes verify the integration between their respective controllers (**OrderController** and **WorkerController**), views (**OrderSwingView** and **WorkerSwingView**), and underlying repositories. These tests cover scenarios such as adding, updating, fetching, and deleting orders and workers. They utilize **AssertJ Swing** for GUI testing to ensure that the user interface behaves correctly and interacts properly with the controllers and repositories.

Race Condition IT Tests:

Integration tests for the **OrderController** and **WorkerController** classes focus on race conditions, ensuring proper functionality in concurrent environments. These tests verify that the application correctly handles race conditions when multiple threads are accessing and modifying order and worker data simultaneously. Specifically, for the OrderController, the tests cover accessing and deleting order data, while for the WorkerController, they cover accessing, adding, and deleting worker data. The tests utilize **Awaitility** for managing asynchronous operations, ensuring robust and reliable performance under concurrent conditions.

End-To-End tests (JUnit):

These tests use the same **DatabaseConfig** Java class for database connections, as defined in the testing strategy for **integration tests.** These tests involve interactions with the **Order** and **Worker views**, such as adding, updating, fetching, searching, and deleting orders and workers. It is important to note that

the Order View Form and Worker View Form are not tested together in previous tests such as integration or unit tests. Since this is an end-to-end test, the focus is on verifying the interaction between the two. The tests verify the correct display of database records in the GUI and ensure proper handling of various operations. Additionally, they ensure appropriate error handling and validation for both orders and workers. The tests utilize the **AssertJSwingJUnitTestCase** framework for GUI testing, Awaitility for asynchronous operations, and persistence for database operations. Runs all **Model, View, Controller, Repository,** and **Utils** together, along with testing functionality that integrates both worker and order views together.

End-To-End tests (Cucumber):

These tests also utilize the same **DatabaseConfig** Java class and a **ConfigSteps** file in **bdd/java** for database connections. They are end-to-end tests, covering the same scenarios as the JUnit end-to-end tests but are written to demonstrate **Cucumber**. The tests are organized into two directories: one for features, located in **bdd/resources,** and the other for the primary test code, located in **bdd/java.** The directory is named BDD because these tests are written in the Behavior Driven Development (BDD) format.Runs all **Model, View, Controller, Repository,** and **Utils** together, along with testing functionality that integrates both worker and order views together using features written in Gherkin, and executed with **Cucumber** in a **BDD** (Behavior Driven Development) style.

Comprehensive Guide to POM File: Dependencies, Plugins, and Profiles:

Dependencies:

All the versions of these dependencies are defined in the properties

Testing Frameworks:

These dependencies are used for **End-To-End Test with BDD**

- Cucumber-junit
- Cucumber-java

Mocking and Assertions:

These dependencies are used for mocking, testing & adding delays to the assertion.

- Mockito-core
- AssertJ-swing-junit
- Awaitility

Database:

These dependencies are responsible for database

- H2
- Postgresql
- Testcontainers

Persistence and ORM:

These dependencies are used for database operations

- Hibernate-core
- Javax.persistence-api

Utility Libraries:

This dependency provides a helper method for Java enums.

- Commons-lang3

Logging:

These dependencies are responsible for logging and solving the issue of logging in test containers

- Log4j-core
- Logback-classic
- Slf4j-api

Command-Line Interface (CLI):

This dependency allows main method to accept arguments (**Database, Host, Port, User, Password**) while running the jar file

- Picocli

Plugins:

Testing:

- **Surefire Plugin :** Runs unit tests.
- **Failsafe plugin**: Runs Integration tests and End-To-End tests.
- **Surefire Report Plugin:** Generate tests report

Code Quality and Analysis:

- **Jacoco plugin**: Manages code coverage
- **Coveralls plugin**: Sends coverage reports to Coveralls.
- **Pitest:** Performs mutation testing.
- **Sonar plugin:** for code quality analysis.

Documentation and Reporting:

- **Project info reports**
- **Site Plugin**

Other Plugins:

- **Build helper maven plugin:** Add **it,e2e,bdd** and **bdd/resources** to the project.
- **Assembly plugin:** Generate fat jar of the project where main class is (**OrderWorkerAssignmentSwingApp**)
- **Docker maven plugin:** Manages Docker containers and images.

Profiles:

This project consists of **4 profiles.**

Jacoco profile:

This profile runs unit tests with coverage excluding models and the main class (**OrderWorkerAssignmentSwingApp**). It utilizes the **JaCoCo** plugin, which operates on the **prepare-agent** and **report** goals. The **prepare-agent** goal sets up the JaCoCo agent to collect coverage data during the test phase, while the **report** goal generates a code coverage report after the tests have run. This profile is specifically created for the **SonarCloud** workflow, which will be discussed later.

Mutation-testing-with-coverage Profile:

This profile runs the **JaCoCo** plugin with the same configuration as mentioned above, along with the **Coveralls** plugin, which is responsible for uploading the coverage report generated by JaCoCo to the **Coveralls** website. By binding the **verify** phase with the **report** goals, **Coveralls** ensures the report is generated after all tests have run, and the **report** goal generates and sends the report to the **Coveralls** website. Additionally, this profile runs the **PIT mutation plugin**, but only on **controller** and **utils unit tests,** with **strong mutators** and a **100% mutation threshold.**

Integration-test-profile:

This profile skips unit tests run by the **Surefire** plugin. The **docker-maven-plugin** starts the Docker container with a predefined port, but other arguments such as user, password, and database are taken from properties that can be updated during runtime. The Docker container will start before integration tests and stop after completing all integration and end-to-end tests, as this plugin is bound to the **pre-integration-test** and **post-integration-test** phases with **start** and **stop** goals respectively. The database connection is checked programmatically by each test before running.

The **Failsafe** plugin in this profile has three executions: one for **integration tests**, one for **JUnit end-to-end tests**, and one for **Cucumber end-to-end tests**. This plugin is responsible for running these tests and generating reports, utilizing its two goals: **integration-test** and **verify**.

Generate_fat_jar_profile:

This profile skips unit tests and generates a fat jar using the **maven-assembly-plugin.**

Database Implementation:

The database used in this project is **PostgreSQL**, which utilizes an **EntityManagerFactory** to establish connections. The connection requires a **persistence**.**xml** file that contains all the database configuration details, such as entities annotated in the models. The **EntityManagerFactory** accepts properties that override those in the **persistence**.**xml** file if duplicated.

This project has a single **persistence**.**xml** file located in **test/resources** that holds the database configuration. However, the user, database, and password details in this file are taken from **pom.xml** properties, which are also used by the **docker-maven-plugin** to start **Docker containers**. Therefore, when running the **integration test profile**, both the **Docker container** and **persistence.xml** will use the same credentials.

In some cases, such as database repository unit tests, the persistence.xml needs to update certain details such as **JDBC URL, user,** and **password** at runtime. These updates are provided by passing **properties** to the **EntityManagerFactory**.

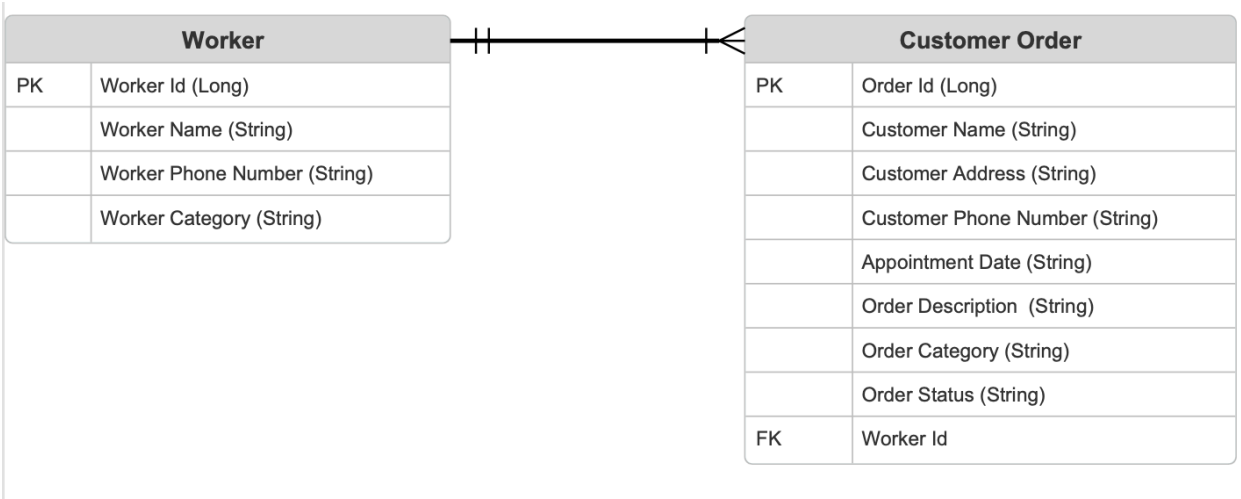
Since **integration** and **End-to-End tests** need to run both locally and in workflows, I have divided the database setup into two types: one with a profile and one without. Running the project with the profile ensures that both the project and Docker container use the configuration in the pom.xml file. Running the project without the profile starts the **test containers**, which use constant variables for configuration, updating properties dynamically and not relying on **pom.xml** properties because the host and port of these test containers can change.

To solve this problem, I created a **DBConfig** interface with three methods, implemented in two files: **MavenContainerConfig** and **TestContainerConfig**.

1. **testAndStartDatabaseConnection**: If the project runs without a profile, this method starts the test container. If it runs with a profile, it starts the Docker container without test containers and tries to connect to the database until successful.

2. **getEntityManagerFactory**: This method provides the EntityManagerFactory for all tests. If the project runs without a profile, it updates the persistence.xml configuration using properties generated by test containers. If it runs with a profile, it uses the default persistence.xml configuration.

3. **startApplication**: This method is only for End-to-End tests. If the project starts without a profile, it passes arguments to the main class configured with Picocli using test container configurations. If started with a profile, it uses the default persistence.xml configuration from pom.xml properties.



Github action workflows & Sonar Cloud:

There are four **GitHub Action workflows** defined in the **.github/workflows** directory. Each workflow follows a common set of steps: it first downloads cached dependencies, then runs tests with various configurations, and finally generates reports that are stored as **artifacts**. These workflows require sensitive information, such as tokens, which are securely stored in **GitHub Secrets**. Only the secret

variables are used in the workflows. Every workflow has badge which can be seen in readme.md file. **Sonar cloud** and **Coverall badge** is also embedded in the **readme.md** file by clicking which you can see reports on their respective websites.Each Workflow works on both push and pull requests.

WorkFlows:

1. **JAVA CI Unit tests, Mutation tests and Coveralls with Maven in Linux**
    - **Operating System:** Ubuntu
    - **Java Versions:** Java 8, Java 11 and java 17
    - **Secrets:** Coversall token
    - **Tests:** All unit tests
    - **Coverage and Mutation Testing:** Performed only on Java 8
        - **Profile:** mutation-testing-with-coverage
    - **Reports**: Unit test report , Coverage report ,Mutation Testing report and logs report
    - **Additional Actions:** Upload Coverage report to coversall
2. **JAVA CI Unit tests with Maven in Windows & MacOS**
    - **Operating System:** MacOS & Windows
    - **Java Versions:** Java 8
    - **Tests:** All unit tests
    - **Reports**: Unit test report

3. **JAVA IT, E2E tests and E2E with cucumber with Maven in Linux**
    - **Operating System:** Ubuntu
    - **Java Versions:** Java 8
    - **Secrets:** Database credentials
    - **Tests:** All Integration and End-To-End tests
        - **Profile:** integration-test-profile
    - **Reports**: FailSafeReport

4. **Java CI with Maven, Docker and SonarCloud in Linux**
    - **Operating System:** Ubuntu
    - **Java Versions:** java 17
    - **Tests:** All unit tests
    - **Secrets:** Github token & Sonar cloud token
    - **Coverage:** Performed on Java 8
        - **Profile:** Jacoco
    - **Additional Actions:** Upload sonar report to sonar cloud, Runs only on main branch

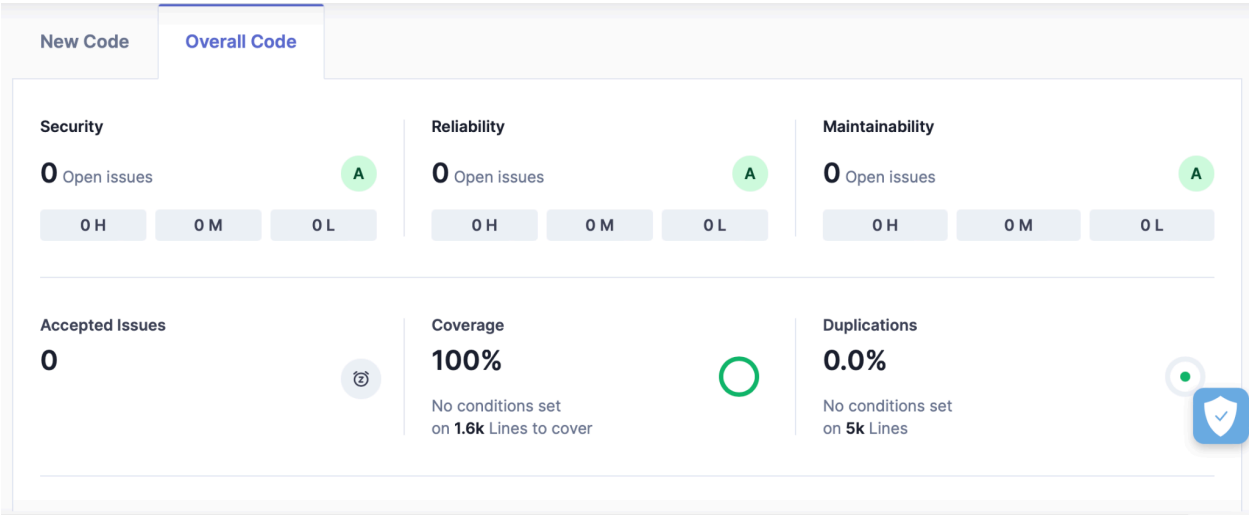
Sonar cloud:

The project is available on the **SonarCloud** website, accessible via the badge in the **README**.md file. Below is a summary of the code on the **main branch** only. In the properties section of the pom.xml file, there are ignored rules along with coverage exclusion for models and the main class (OrderWorkerAssignmentSwingApp). The SonarCloud plugin is utilized in the **"Java CI with Maven, Docker, and SonarCloud in Linux"** workflow, which employs **Java 17** due to issues encountered when running this workflow with **Java 11.** However, SonarCloud was operational with J**ava 8 and 11** when using the local Sonar Docker container, which can be run with the docker-compose.yml file in the project.

Sonar Cloud Ignored Rules (False positives):

1. The use of static access with "**javax.swing.WindowConstants**" for "**EXIT_ON_CLOSE**" in all Java classes was ignored because it is auto-generated by Windows Builder.
2. The rule "**Constructor has X parameters, which is greater than Y authorized**" was ignored in all Java classes due to the code being generated by Windows Builder and models.
3. The warning "**Add at least one assertion to this test case**" was ignored in **OrderSwingViewTest** and **WorkerSwingViewTest** since SonarQube does not recognize that these AssertJ Swing methods effectively assert.
4. The recommendation to "**Rename this fields like CUSTOMER_PHONE_1 to match the regular expression**" was ignored because the current naming provides more **readability** in the tests.
5. The suggestion to "**Rename class OrderWorkerSwingAppE2E to match the regular expression**" was ignored in the OrderWorkerSwingAppE2E test class.
6. The advice to "**Replace these 3 tests with a single Parameterized one**" in the **ExtendedValidationConfigurations** class was ignored as it warns that three methods should be merged into one.

Overall Code Sonar Cloud:



Conclusion:

In conclusion, this project showcases the successful development of a **Model-View-Controller (MVC)** application leveraging **Test-Driven Development (TDD), automated builds,** and **continuous integration.** Through a comprehensive testing strategy, robust architecture, and the effective use of tools like **Maven**, **Docker**, and **SonarCloud**, the project achieves high standards of code quality, reliability, and maintainability. The "**Order Worker Management System**" effectively manages orders and workers, demonstrating practical utility and scalability. The application is rigorously tested through both **white-box** and **black-box** methodologies, encompassing **unit, integration,** and **end-to-end tests.**

Additionally, **GitHub Actions workflows** are utilized to ensure the project is thoroughly tested across multiple operating systems and Java versions, further enhancing its robustness and compatibility. This

project not only meets its objectives but also sets a solid foundation for future enhancements and development, illustrating the importance of modern software development practices.