Building a Reusable Game Framework in Java

Advanced Object Oriented Programming,
DT4014 7.50HP

Supervisors: Wojciech Mostowski, Nesma Rezk

Halmstad, 26th May 2023

Thomas Lundqvist, Hieu Tran, Rim Abdennour

1. Introduction

The course AOOP, *Advanced Object Oriented Programming* at Halmstad University, is based on developing a software solution for a grid-based game framework project. The goal was to craft a versatile and efficient platform for game development, adhering to the specific project requirements. The project is named the *Breadcrumb Project*.

Breadcrumb comes from the tale of Hansel and Gretel, where breadcrumbs served as markers to find their way home. In this context, Breadcrumb symbolizes a guide through the complexities of grid-based game development. The framework streamlines the coding process, simplifying the creation of complex games, and presents a clear, navigable path for developers.

This report presents the design, implementation, and testing of a software solution developed using a VCS *(Version Control System)* to address the requirements outlined in the project specifications for a framework of grid-based games. Testing is essential to software development as it ensures the correctness of individual units (methods, functions, classes) in isolation.

The developed software aims to provide a framework for grid-based games with audio, Input/Output management, and menu- and entity-handling functionalities through design patterns such as MVC *(Model-view-controller)* and Observer.
The current report discusses the decision-making process, highlights the software's key features, and compares it to similar existing programs.

2. Design

The Breadcrumb project design principles for the framework were a commitment to modularity, reusability, and extensibility, guided by the implementation of Object-Oriented Programming (OOP) principles and design patterns, especially MVC and Observer. This approach served several purposes:

- **Ease of Modification:** With clearly separated components + (model, view, controller, observer, and strategy), the project could easily maintain, improve and enhance the code base to add the ability for the code to act more independently from the rest of the code base, allowing for parts of the system to be replaced and updated without impacting the rest of the systems.
- **Code Reusability:** The framework inherently promotes reusability and reduces code duplications across game implementations[1]. By leveraging the functionality of code reusability, developers can accelerate their development pipeline. This allows the developer to focus on game-specific rules and functionalities.
- **Extendability:** By focusing foremost on the framework and not a singular game, we ensured that our solution could easily be extended to incorporate other games.
- **Testing and Quality Assurance:** The well-organized structure of the framework streamlined the testing process. The design incorporated shared components, allowing easy reuse and tailor-specific testing scenarios. Consequently, this ensures the increased quality and reliability of the framework.

The following diagrams and code snippets provide an overview of our framework and how it was employed to create the Sokoban and 2048 games.

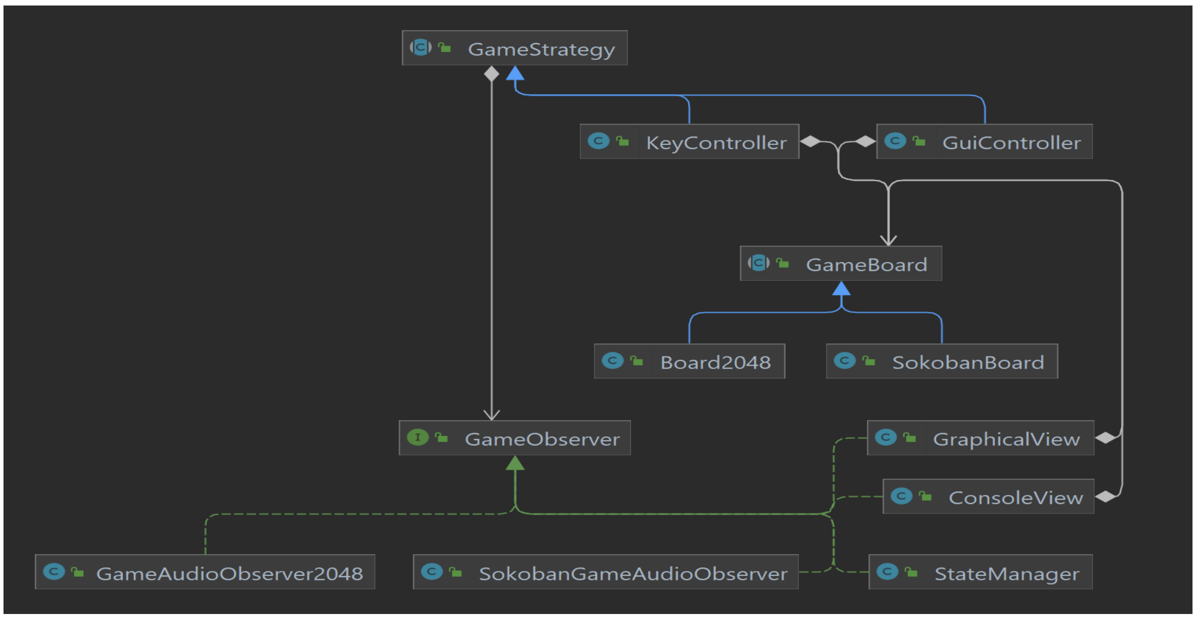


Figure 1: *The UML diagram shows our design clearly. The GameBoard and its parts are the Models. The Controller is part of the Strategy pattern. GameState, Audio, and Graphical observers keep track of game events to enhance gameplay*[2].

```
public      class      StateManager      implements
GameObserver {
   private final Game game;

   public StateManager(Game game){
       this.game = game;
   }

   @Override
   public void update() {
       switch (game.getBoard().getState()) {
           case WIN, GAME_OVER -> game.end();
           case INITIATE -> game.initiate();
           case LEVEL_COMPLETE -> game.start();
           case RUN -> {}
           case RESET -> game.restart();
           case PAUSE -> game.pause();
       }
   }
}
```

```
public  class  ConsoleView  implements
GameObserver {

   private final GameBoard board;

         public   ConsoleView(GameBoard
board){
       this.board = board;
   }

   @Override
   public void update() {
       System.out.println(board);
   }
}
```

Figure 2: *Creation of an Observer class*[3].

```
getController().addStateObserver(
    new StateManager(this)
);
getController().addAudioObserver(
    new SokobanGameAudioObserver(this)
);
getController().addViewObserver(
    frame.getGameView()
);
getController().addViewObserver(
    new ConsoleView(getBoard())
);
```

```
notifyStateObservers();

notifyViewObservers();

notifyAudioObservers();
```

Figure 3: *(Left) Adding observers in Sokoban class and (Right) how to update on an event*[4].

As illustrated in Figure 2 and Figure 3, implementing an observer, registering it with the event listeners, and subsequently generating an event is a straightforward process in the framework. Figure 1 provides a detailed view through a UML diagram, effectively showcasing the interdependencies and interactions among these components, elucidating their cohesive operation within the system.

3. Testing

As mentioned before, the purpose of the testing is to ensure the correctness of individual units, such as methods, functions, and classes, in isolation. While creating the software, several tests were made after hand, and the created method for the software was accomplished. The tests are organized into individual cases where each focuses on a specific aspect or method of the specific class.

Automatic testing techniques were employed using JUnit 5 and Mockito frameworks. JUnit enabled automated test execution for consistent results. Mockito simulated dependencies, allowing isolated testing of specific class methods. By mocking dependencies, tests focused solely on the class behavior, independent of external factors. This approach ensured independent and reproducible tests, enhancing reliability.

```java
class Board2048Test {
   Board2048 underTest;
   Game2048 game;
   @BeforeEach
   void setUp(){
       game = mock(Game2048.class);
       Board2048 board = new Board2048(game, 6, 6);
       underTest = Mockito.spy(board);
       doNothing().when(underTest).addNewTile();
       doNothing().when(underTest).addNewTile();
   }
}
```

Figure 4: *A setUp() method for the board class of the game 2048*[5].

In every test file, a setup method is mandatory to construct the specified arguments adjusted for the class the test is conducted for. This was done with the help of mocking classes and spying on the objects. The spy method is used to partially mock the object, meaning that parts of the methods will be real. It is used when the object is not easily testable[6]. The different statements, such as doNothing(), doReturn(), and more, were used to test the code. This can be seen in Figure 4.

3.1 Unit testing.

For this project, unit testing was prioritized over integration testing. Unit tests are often simpler to write and easier to read, making them a practical choice. Moreover, unit and integration testing effectively detect issues early on. This strategy aimed to boost the reliability and stability of individual components, enhancing the overall integrity of the project's game framework.

```java
public void makeMove(Location direction) {
 if(direction.getX() == 1){
   transpose();
   reverse();
   compress();
   merge();
   compress();
   reverse();
   transpose();
 }
 else if(direction.getX() == -1){
   transpose();
   compress();
   merge();
   compress();
   transpose();
 }
 else if(direction.getY() == 1){
   reverse();
   compress();
   merge();
   compress();
   reverse();
 }
 else if(direction.getY() == -1){
   compress();
   merge();
   compress();
 }
 if(isGameOver())
   game.getBoard().setState(GameState.GAME_OVER);

 game.getController().notifyAudioObservers();
 addNewTile();
}
```

```java
void test_makeMove_right() {
   // Given
   Location right = new Location(1, 0);

 GameStrategy controller = mock(GameStrategy.class);
 doReturn(controller).when(game).getController();
 doNothing().when(controller).notifyAudioObservers();
 doReturn(false).when(underTest).isGameOver();

 doNothing().when(underTest).transpose();
 doNothing().when(underTest).compress();
 doNothng().when(underTest).merge();
 doNothing().when(underTest).reverse();

 // When
 underTest.makeMove(right);

 // Then
 verify(underTest, times(2)).transpose();
 verify(underTest, times(2)).reverse();
 verify(underTest).merge();
 verify(underTest, times(2)).compress();
 verify(underTest).addNewTile();
 verify(controller).notifyAudioObservers();
}
```

Figure 5: (Left) The makeMove() method is for making a move in the specified direction of the gameboard. (Right) The unit test for the makeMove() method to test the move in the right direction[7].

The framework was tested in five main parts: audio, core, entities, IO *(Input/Output),* and models. The few tests that were done in the core were if it was possible to enter the next level, level completion, and clearing entities. The entity testing was tested by setting the entity in different positions, including null. This was done because the entities should be movable. The IO testing was done, with the controllers, working as an input, and the direction was updated to its new value, as seen in Figure 5.

3.2 Integration Testing

By testing the integration and interaction between these different methods, it aims to discover potential defects early. This could be, for example, communication failures that may only happen when the methods are interacting. The primary objective of integration testing is to ensure the integrated system functions properly as a unified entity. This is done after confirming that the different components work individually. By detecting and resolving integration-related early, integration testing improves the software early[8].

```java
public void makeMove(Location direction) {
 if(direction.getX() == 1){
   transpose();
   reverse();
   compress();
   merge();
   compress();
   reverse();
   transpose();
 }
 else if(direction.getX() == -1){
   transpose();
   compress();
   merge();
   compress();
   transpose();
 }
 else if(direction.getY() == 1){
   reverse();
   compress();
   merge();
   compress();
   reverse();
 }
 else if(direction.getY() == -1){
   compress();
   merge();
   compress();
 }
 if(isGameOver())
  game.getBoard().setState(GameState.GAME_OVER);

 game.getController().notifyAudioObservers();
 addNewTile();
}
```

```java
void iTest_makeMove_right() {
 // Given
 Location right = new Location(1, 0);
  Entity  block1  =  new  Block(new  Location(1,  1),
EntityIcon2048.E2);
  Entity  block2  =  new  Block(new  Location(1,  2),
EntityIcon2048.E4);
  Entity  block3  =  new  Block(new  Location(1,  3),
EntityIcon2048.E2);
  Entity  block4  =  new  Block(new  Location(1,  4),
EntityIcon2048.E8);

 ArrayList<Entity> entities = new ArrayList<>();
 entities.add(block1);
 entities.add(block2);
 entities.add(block3);
 entities.add(block4);
 underTest.setEntities(entities);

 GameStrategy controller = mock(GameStrategy.class);
 doReturn(controller).when(game).getController();
 doNothing().when(controller).notifyAudioObservers();
 doReturn(false).when(underTest).isGameOver();

 // When
 underTest.makeMove(right);

 // Then
 assertThat(
  underTest.getEntities().get(0).getLocation()
 ).isEqualTo(new Location(4,1));
 assertThat(
  underTest.getEntities().get(1).getLocation()
 ).isEqualTo(new Location(4,2));
 assertThat(
  underTest.getEntities().get(2).getLocation()
 ).isEqualTo(new Location(4,3));
 assertThat(
  underTest.getEntities().get(3).getLocation()
 ).isEqualTo(new Location(4,4));
}
```

Figure 6: (Left) The makeMove() method is for making a move in the specified direction of the gameboard. (Right) The integration test for the makeMove() method to test the move in the right direction[9].

During the testing process, several integration tests were made to verify the collaboration of multiple methods in the software. An example can be seen in Figure 6, where the integration test is the same as the makeMove() unit testing in Figure 5.

4. Implementation Highlights

Applying the Observer design pattern in the game framework was rewarding. This pattern proved valuable when updating various views following alterations in the game state was needed. Moreover, it offered developers the advantage of efficiently implementing new event triggers. This solution streamlined our development process while maintaining the flexibility of our codebase.

In a standard game, multiple components, such as the graphical user interface *(GUI)*, sound effects, and game states, must be updated when the game changes. Traditionally, the game would need to have direct references to all these components and update each one manually, resulting in tightly-coupled and hard-to-maintain code.

The approach sidestepped the traditional challenge of manually updating various components in the game. It was possible to avoid creating standard hard-to-maintain code by focusing on efficient event-based updates. Instead, obtaining a smoother, more streamlined development process was possible. The effectiveness of this strategy is highlighted in Figure 2 and Figure 3, where it is visualized how the implementation handles these updates.

The MVC design pattern was another tool that was found to be useful during the work process on the project. This pattern helped differentiate aspects of the game, such as game controls, keyboard inputs, mouse clicks, and network-based inputs, but also the outputs, console-based view, and Graphical view. By applying these patterns, we could add, remove, or change these controls and views. This approach provided increased flexibility and ensured code organization for improved management.

One challenge in game development involves ensuring inclusive gameplay experiences that cater to diverse playstyles, no matter how it is wished to play, whether it is console, PC, or browser. With the Strategy pattern, it was possible to support different playing methods without having to rewrite the whole game. Whether a player prefers to use the keyboard or the mouse, the developers could easily accommodate it.

5. Results and evaluation

5.1. Results

The developed software solution, a grid-based game framework implemented in Java, successfully addressed the project requirements and achieved the desired functionality. The framework demonstrates several key features, including reusability and extensibility to the course principles and design patterns such as MVC (Model view controller) and Observer.

The framework enabled the creation of Sokoban and 2048 games, depicted in Figure 1. It separates components (models, views, observers, controllers, strategies) for easy modification, code reusability, and extendability, facilitating the efficient creation of new grid-based games. The code is organized, allowing the developer to seamlessly integrate additional features, minimizing the impact on existing code and ensuring maintainability.

The testing phase of the project involved both unit testing and integration testing. Unit testing was prioritized to focus on individual units, such as methods, functions, and classes, to ensure their correctness in isolation—the integration testing aimed to verify the interaction and collaboration between different methods within the software. The tests were successful and produced the expected results, ensuring the functionality and reliability of the code. The use of Mockito facilitated the mocking of dependency and allowed for isolated testing[10].

5.2. Evaluation

The developed framework demonstrates several strengths and advantages. Adopting design patterns, such as Observer and MVC (Model view controller), proved valuable in achieving modularity, reusability, and extensibility. The MVC(Model view controller) pattern helped differentiate game controls and views, providing flexibility and code organization. In contrast, the Observer pattern facilitated efficient updates of various views based on game state changes.

The program's design encourages abstraction and separation of concerns, making isolating and modifying specific parts easier without affecting the overall system. This approach makes it possible to extend new functionality to the program. Regarding to the program's extensibility, it has been designed flexibly to make it relatively easy to incorporate new functionality.

There are cases where a better implementation of a particular component is possible due to the program's design principles making modifications straightforward. By following modularization and adhering to solid principles, one can easily replace specific classes without disrupting the functionality of other program parts. This flexibility allows for continuous improvement and facilitates the adoption of more efficient algorithms or optimized solutions when necessary.

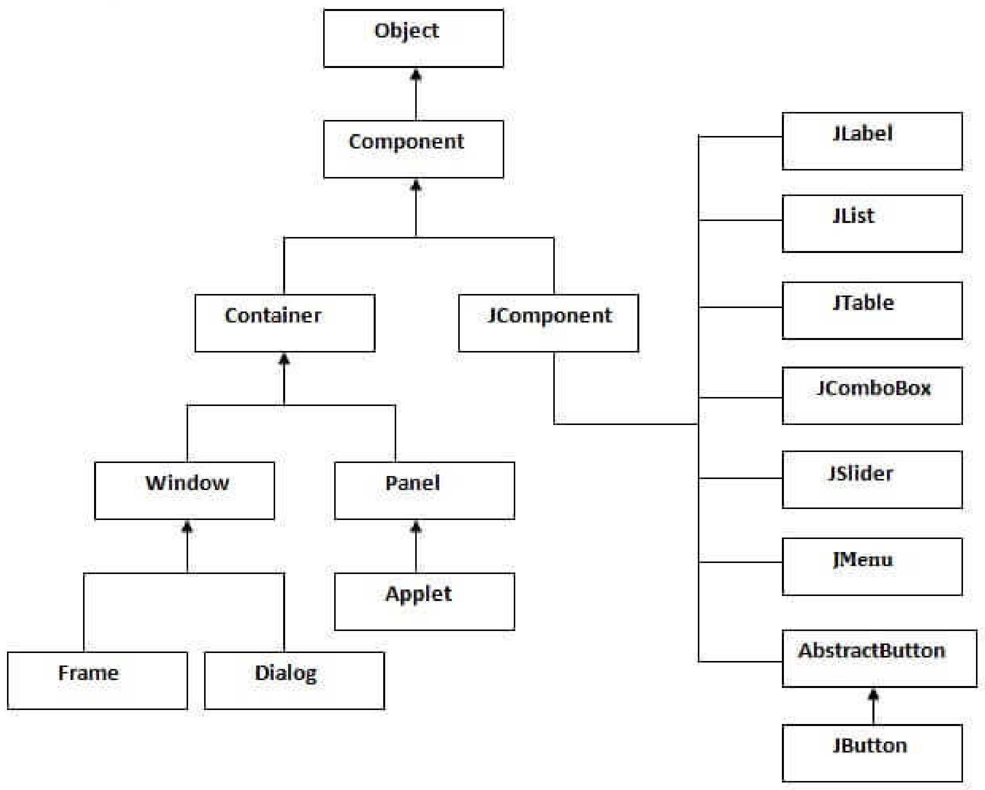
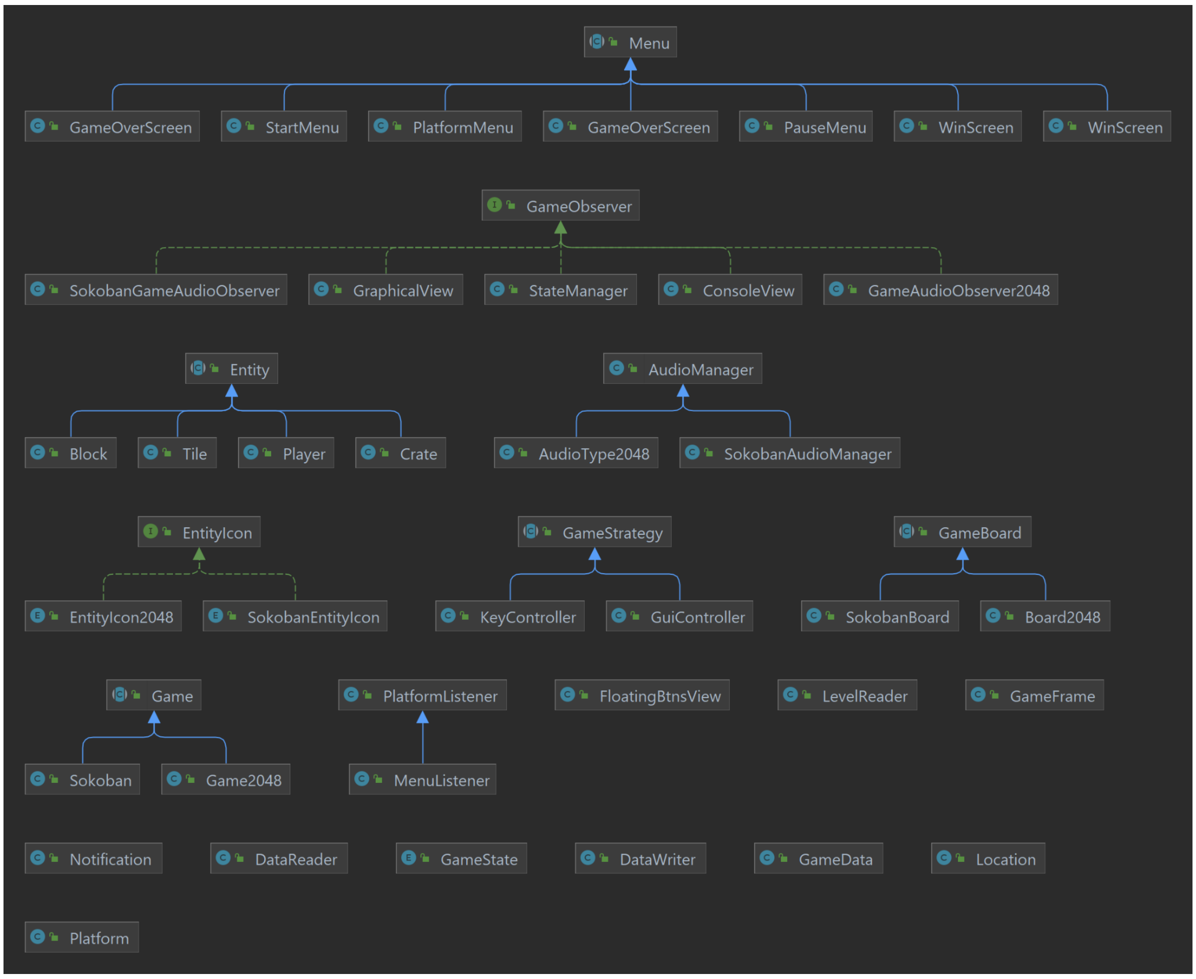


Figure 7: Java Swing framework hierarchy diagram showcases the structure and illustrates the connection between the GUI components and containers for building interactive UI in Java applications[11].



To compare the Breadcrumb project and the Java Swing framework, the Java Swing framework has its components more organized into classes and subclassed based on its functionalities. Java Swing is more of a rich set of GUI components. Meanwhile, the Breadcrumb projects framework suggests a different structure. The structure is followed by specific requirements and design principles and therefore built after that.

For customizability, the Java Swing framework enables developers to extend and modify their different components to suit their requirements. To some level, the Breadcrumb project allows developers to do the same, although some methods could be added to the different framework classes that could be used for future game developments. This could be, for example, finding the entity at a specific location or checking if the player ended up at a dead end.

Overall, the project results indicate the successful development of a reusable game framework in Java and demonstrate the potential of creating grid-based games efficiently and effectively. The code is organized, allowing the developer to seamlessly integrate additional features, minimizing the impact on existing code and ensuring maintainability. The software architecture and design choices make it easy to modify and extend. By adhering to software engineering best practices, the program remains adaptable and open to future enhancements while maintaining stability and existing functionality.

6. Version Control System

As the Breadcrumb project started, the team chose GitHub as the primary Version Control System (VCS) and collaborative development platform. At the outset, the collective knowledge of GitHub was fundamental. The nuances of navigating this platform soon posed several challenges, notably the occasional misstep of committing changes to the wrong branches. This necessitated the need to address and resolve merge conflicts.

To manage the team's GitHub workflow, the project was incorporated through IntelliJ IDEA, an Integrated Development Environment (IDE) with its built-in VCS handler. The integration of the tools made it possible to manipulate version control tasks without leaving the coding environment, streamlining the team's process. IntelliJ's interactive interface helped to resolve conflicts and manage the project's GitHub repositories directly from the IDE, adding a layer of convenience to the workflow.

As the Breadcrumb project progressed, a clear understanding emerged regarding the importance of proper branch management and merge strategies. With the practice of using pull requests for peer reviews, an environment of shared learning and continuous improvement was fostered. This approach was beneficial in elevating the code quality and enriched the collective knowledge of version control systems.

Despite an initially steep learning curve, the journey with GitHub and IntelliJ IDEA has proven incredibly valuable. The team's experience navigating the complexities of GitHub, complemented by the efficiencies provided by IntelliJ's VCS handler, not only facilitated effective collaboration and comprehensive code management and fostered a sense of responsibility and ownership among team members. Consequently, this robust combination of tools has become integral to the project development.

7. References

1. O'Grady, B. (n.d). What is a Framework? Why We Use Software Frameworks. Code Institute. [Online]. Available at: https://codeinstitute.net/se/blog/what-is-a-framework/ (Accessed: 20 May 2023).
2. Lundqvist, T. (2023). UML Diagram. [Image].
3. Lundqvist, T. (2023). Observer Class. [Image].
4. Lundqvist, T. (2023).Observer controllers. [Image].
5. Tran, H. (2023). setUp() for the testing. [Image].
6. Paraschiv, E. (20 May, 2023). Mockito – Using Spies. Baeldung. [Online]. Available at: https://www.baeldung.com/mockito-spy (Accessed: 22 May 2023).
7. Tran, H. (2023). Unit Testing. [Image].
8. Integration Testing. (n.d). JavaTPoint. [Online]. Available at: https://www.javatpoint.com/integration-testing (Accessed: 15 May 2023).
9. Tran, H. (2023). Integration Testing. [Image].
10. Mockito. (n.d). Mockito JavaDocs. [Online]. Available at: https://javadoc.io/doc/org.mockito/mockito-core/latest/org/mockito/Mockito.html (Accessed: 26 May 2023)..
11. JavaTPoint. (2023). Java Swing Framework Hierarchy Diagram. [Image]. Available at: https://www.javatpoint.com/java-swing (Accessed 26 May 2023).