**HeapUpBox Game**

**(IP Project)**

Coordinator, Student,

Prof. Cristian Sandu Luca Sergiu

YEAR III CTI

GROUP 22C32 B

YEAR 2022-2023

Content

[Introduction 2](#_Toc125501219)

[The game manual 5](#_Toc125501220)

[Launching the Application 5](#_Toc125501221)

[Level selection 6](#_Toc125501222)

[Level playing 7](#_Toc125501223)

[Game resources 9](#_Toc125501224)

[The logo 10](#_Toc125501225)

[The wallpaper 10](#_Toc125501226)

[Technologies used 11](#_Toc125501227)

[. 11](#_Toc125501228)

[View the source code 12](#_Toc125501229)

[Design Patterns 14](#_Toc125501230)

[Adapter 14](#_Toc125501231)

[Builder 15](#_Toc125501232)

[Chain of Responsability 18](#_Toc125501233)

[Command 20](#_Toc125501234)

[Mediator 21](#_Toc125501235)

[System architecture and structure 22](#_Toc125501236)

[Calculation of the stability index 24](#_Toc125501237)

[The test plan 25](#_Toc125501238)

[List of tested cases 26](#_Toc125501239)

[Test results 27](#_Toc125501240)

[Bibliography resources and libraries 28](#_Toc125501241)

# Introduction

**What is Object Oriented Programming? – "Object oriented" or "object-oriented"?**

Object-oriented programming is one of the most important programming paradigms. It is a system of principles that define what basic logic we use to solve a given task and to structure our programs and how the elements are connected to each other.

A programming language can follow many paradigms, and a piece of code can contain more than one paradigm. There are procedural, object-oriented, functional, and logical programming paradigms.

**Advantages of object oriented programming:**

* It's easier to model complex things as simpler, more reproducible structures
* The code is easy to reuse
* We can create programs that run faster with it
* The code structure is more transparent
* Development is faster because parallel classes can be developed at the same time
* The program can be modified, debugged and maintained more easily
* It is more secure (encapsulation and abstraction make data more secure)

**The most popular object-oriented languages:**

* C++
* C#
* Java (language also used in this project along with the JavaFX library)
* JavaScript
* Python

**Objects vs. Classes**

The main concept of object-oriented programming is the object. Just like objects in real life, objects in programming have properties and methods.

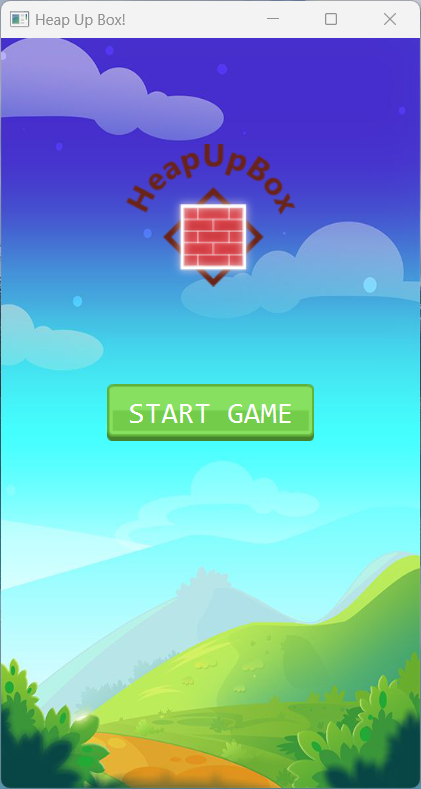
The data structure and functions stored in objects do not need to be re-entered again and again during programming. To make the coding process and code simpler and more economical, we create blueprints called classes. They contain basic information about the structure of objects, and objects contain several pieces of data and methods.

# The game manual

In Heap Up Box, all you have to do is build all the boxes present in each level. You have to build until you reach the dotted line within the time limit you have. Seems simple, doesn't it?

Success!

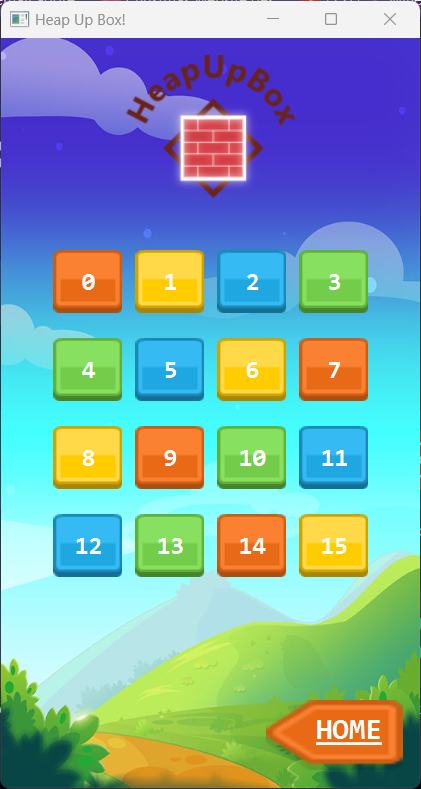
## Launching the Application



Once the game is launched, the player is greeted by the "Home" screen containing the game Logo and the "START GAME" button.

Once the "START GAME" button is pressed, it will send us to the "Level Selection" screen.

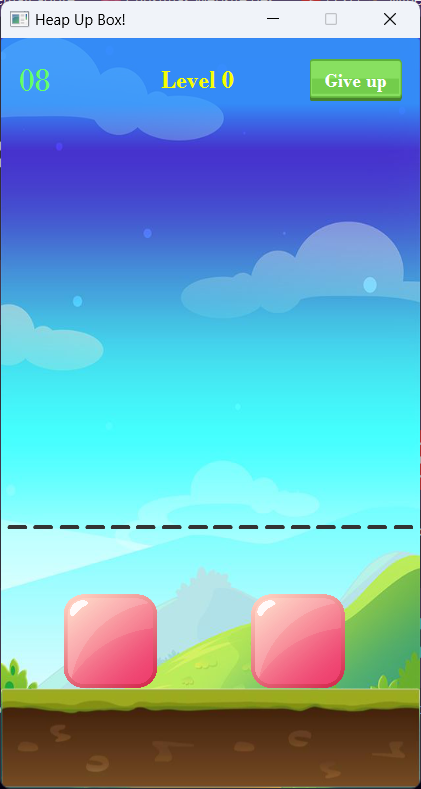
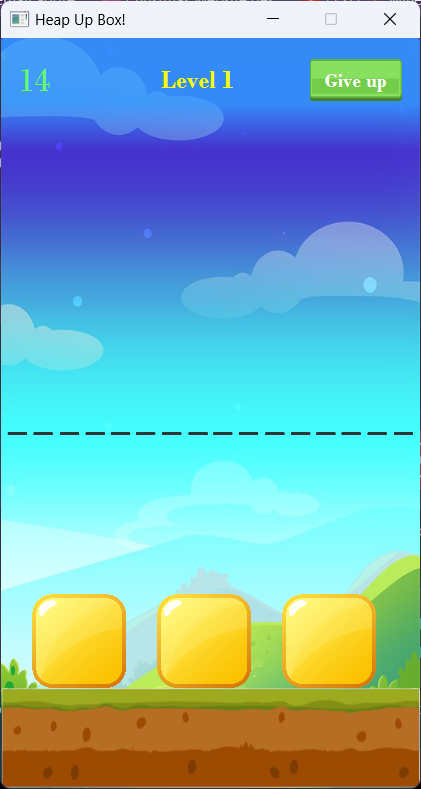
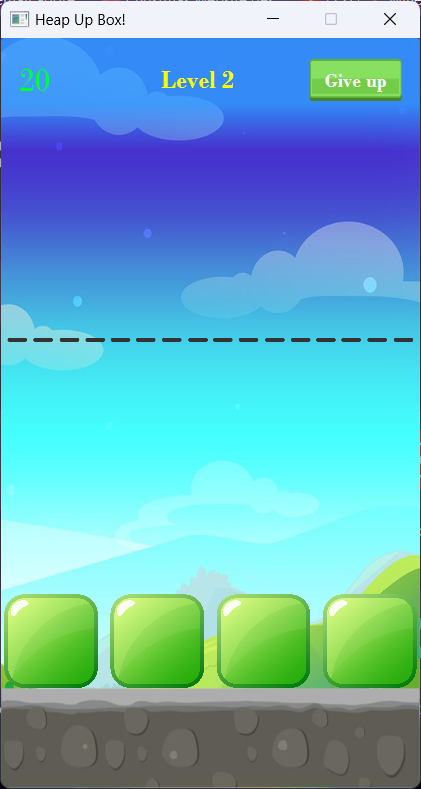
## Level selection



On the "Level selection" screen, the game logo is available once again, along with the specific buttons for selecting the desired level and the "Home" button, which will redirect us to the "Home" screen once pressed.

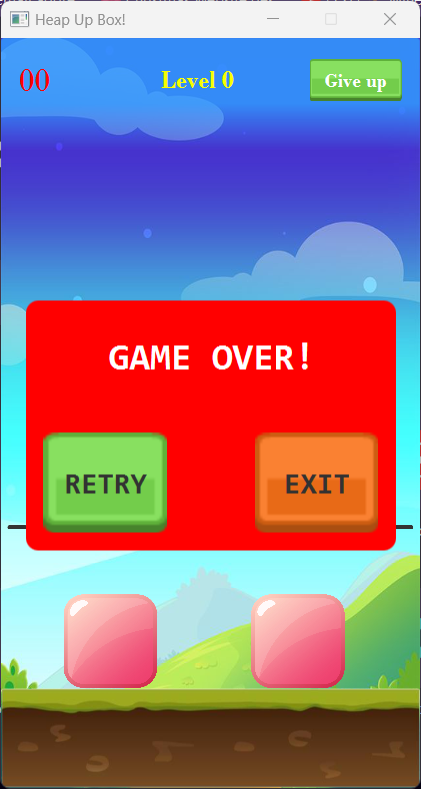
Each level button pressed represents a new set of boxes and platform with a totally new Win line generated depending on the level difficulty.

## Level playing

Each level can have a different type of boxes and a different type of platform.

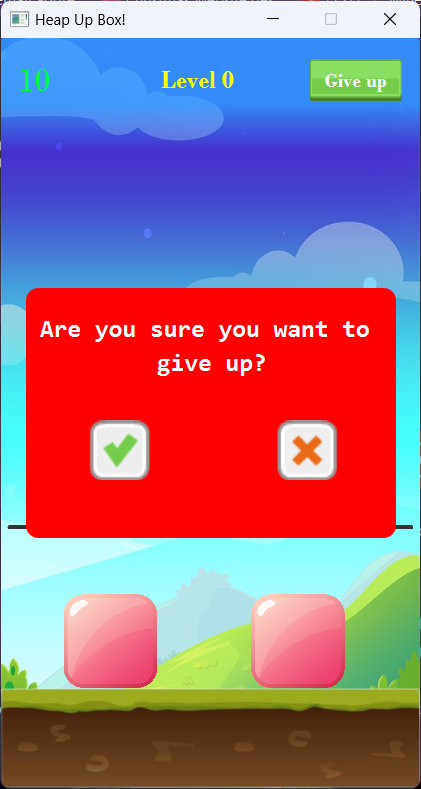
Depending on the difficulty of the level, there will be more or less boxes and the available time will be adjusted according to the existing boxes.



Once the time has reached 0 and the player has not managed to make the tower from the boxes, the "Game over" window is displayed and two options are displayed.

"Retry" Button – This button can be used to reload the level.

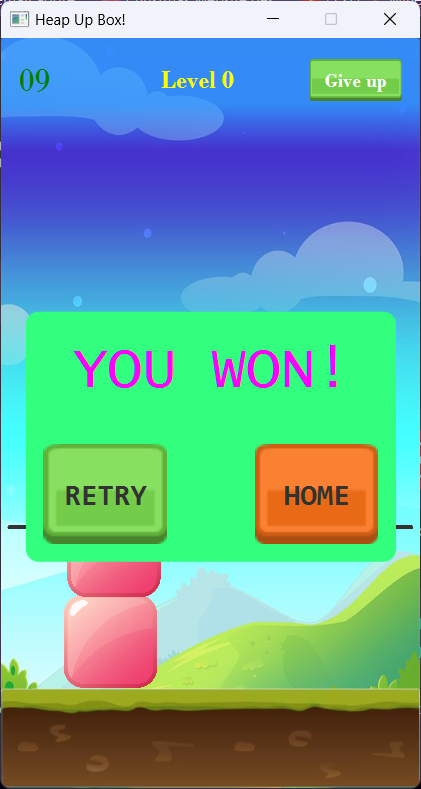
"Exit" Button – This button can be used to return the player to the "Level Select" screen.



If the player considers that he cannot complete the level, he can press the "Give Up" button to give up the current attempt. Once pressed, the player is shown the confirmation window of giving up the game together with the related buttons.

Tick Button – This button can be used to confirm quitting and return to the "Level Selection" screen.

X Button – This button can be used by the player to cancel the attempt to quit the current level and continue where they left off.



If the player managed to make the tower from the boxes in time, the win window will be displayed.

It also features 2 buttons to retry the level and get a better time and to redirect the player to the "Home" screen.

"Retry" Button – This button can be used to reload the level.

Home Button – This button can be used to return the player to the Level Select screen.

# Game resources

For the game resources I used kenney assets.

From this website I downloaded and used:

* Boxes:

* Platforms:

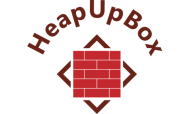




* Buttons:



The logo was created using the Canva.com application using a predefined template:



The wallpaper was downloaded from the Pinterest app:



The game also offers timer, error, game start, level quit, game lost and game won sounds.

# Technologies used

Java is a powerful and cross-platform programming language that allows you to write code that can be executed on any system that has a Java runtime installed. This makes Java a popular choice for desktop and web application development.

JavaFX is a software platform built on Java that enables the creation and delivery of desktop applications as well as rich web applications (RIAs). JavaFX provides a number of tools and libraries for creating graphical user interfaces and handling events, as well as support for animations, graphic effects, and multimedia support.

In this game, JavaFX is used to build the application's GUI, as well as handle events and user interactions. Besides these, some Java libraries were used in this project to manage the logic of the game and its objects.

.fxml files are used in JavaFX applications to define the graphical user interface (GUI) via an XML file format. This format allows developers to create and organize graphical elements in the user interface in a declarative fashion, instead of having to write Java code to build the interface by hand.

fxml files can include elements such as buttons, labels, text fields, tables, images, etc. and can be used to create different pages or windows in the application. These are usually associated with Java classes that handle the logic and behavior of these graphics elements.

In conclusion, this game uses Java and JavaFX technologies to build a fun and interactive desktop game called "HeapUpBoxGame".

# View the source code

The source code can be viewed on the following link:

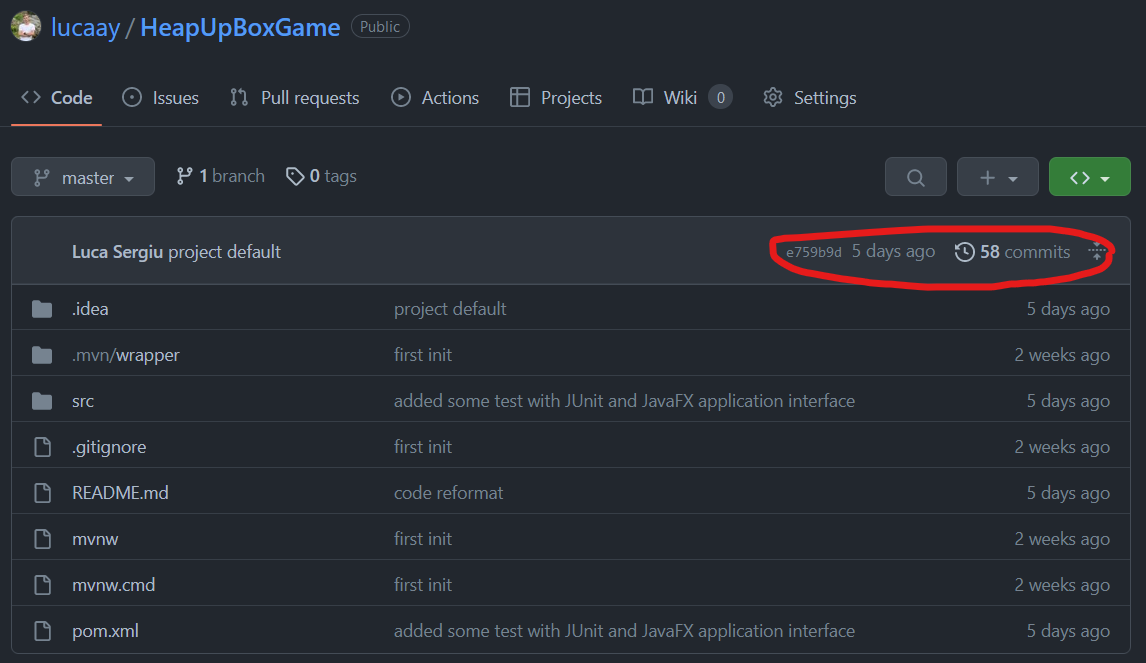
<https://github.com/lucaay/HeapUpBoxGame>

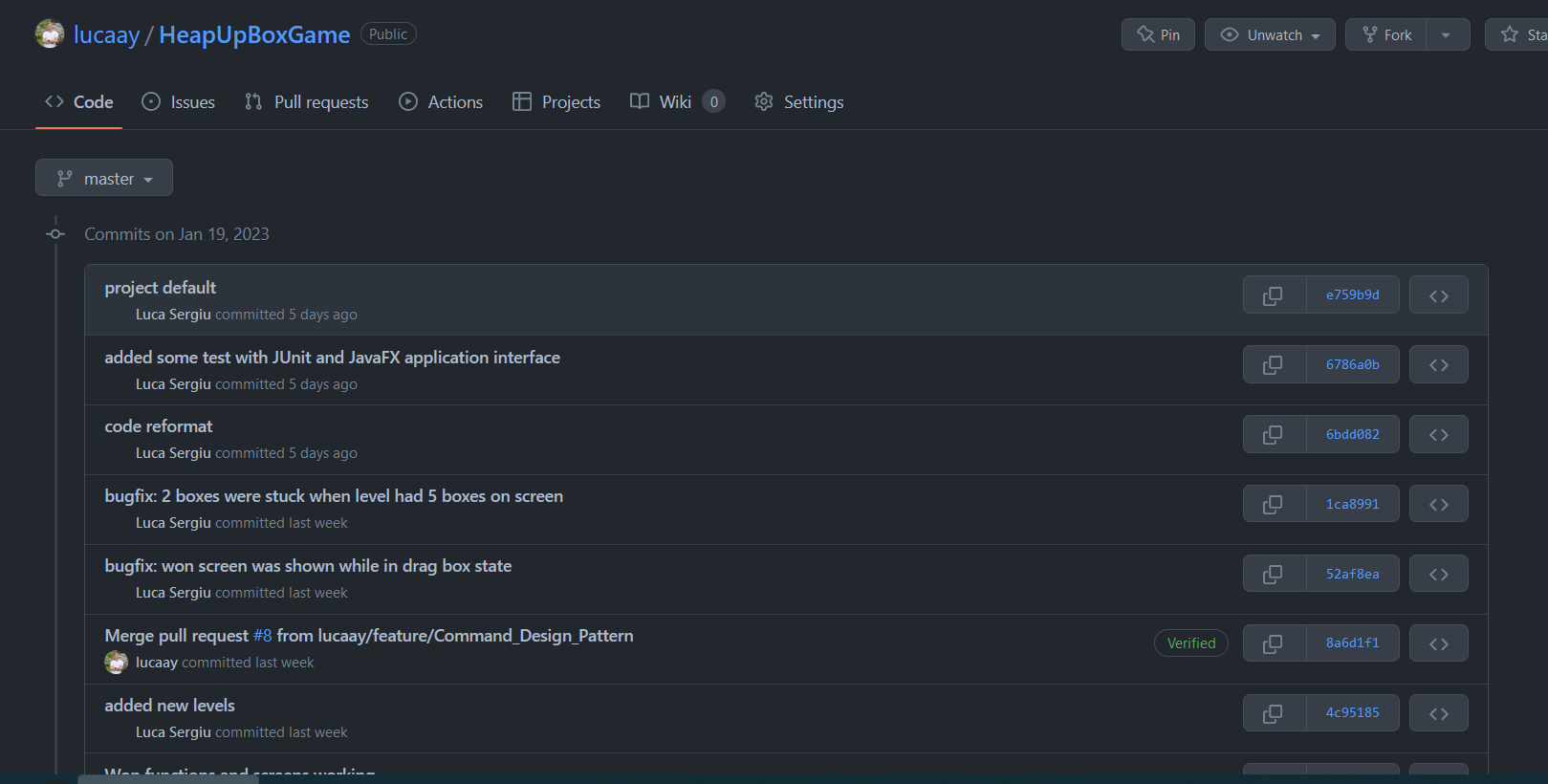
The following can be viewed here:

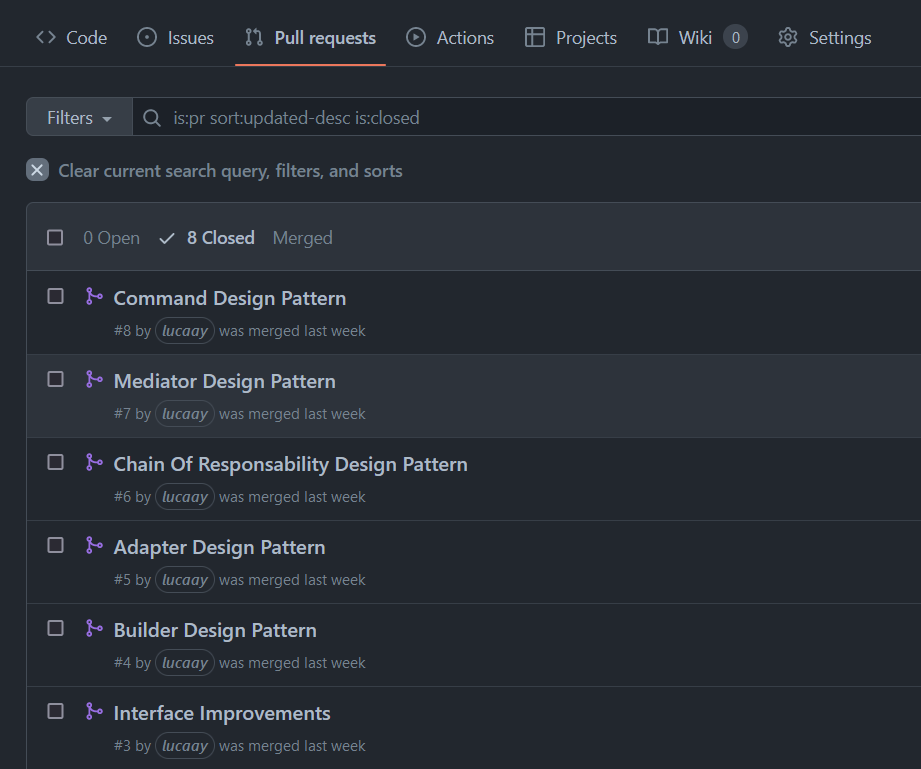
* All game code

• Several commits showing the changes made throughout the development of the game

• The code divided into Pull Requests representing the major Stages in the development of the game.

• A message for each commit explaining the changes made





# Design Patterns

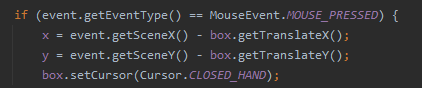
## Adapter

This design pattern allows adapting one class or interface to another so that it can be used together with other components that have a different interface. This pattern is used to allow communication between components that would not normally be able to communicate due to interface differences.

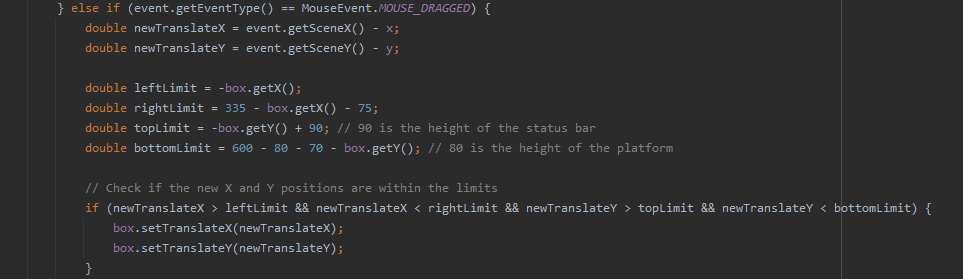
In this game I used the Adapter to make the connection between the player's mouse and the objects in the level.



Once a box is pressed, its x and y coordinates are taken and stored in local private variables:



When the player holds down the click and moves the boxes on the screen, the new coordinates are constantly stored during the move and it is ensured that the player cannot remove the boxes from the screen.



At the end of moving the selected box, it will be updated with its new position.

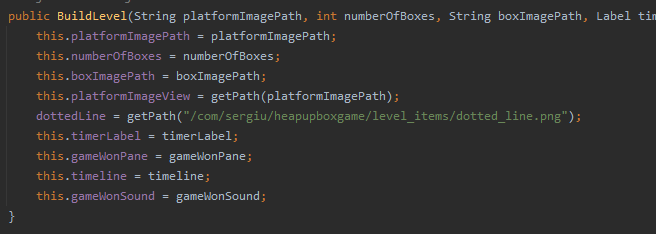
## Builder

This design pattern allows building a complex object through a step-by-step process. This pattern is useful when there are a variety of options for building an object, or when building an object is a complicated process.

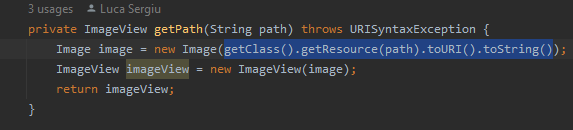
In this game I used Builder to create the game's levels as easily and modularly as possible.



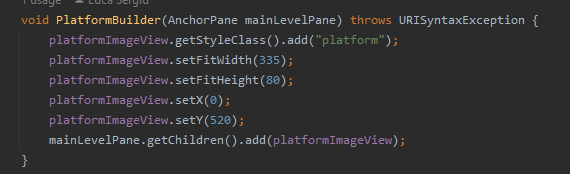
I created the BuildLevel class that contains all the variables needed to initialize a level:



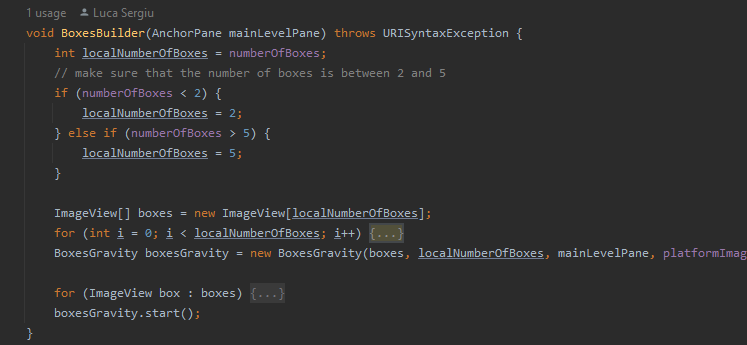
I created the getPath function that changes the image paths in ImageView Objects so that they can be displayed in the game:



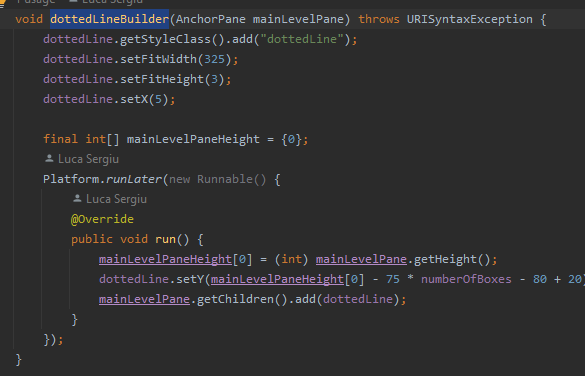
We created the PlatformBuilder function whose role is to initialize, size and position the chosen platform:



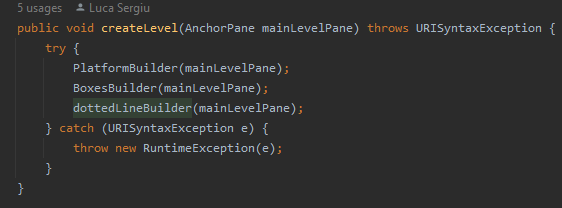
I created the BoxesBuilder function, which has the role of initializing the chosen boxes as well as the gravitational force on them (using the Chain Of Responsibility design template presented below):



I created the dottedLineBuilder function, which has the role of initializing the selected Win line as well as its position depending on the status bar and the existing platform:



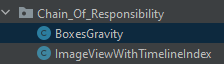
And finally, I created the createLevel function that calls the 3 functions above to initialize a new level:



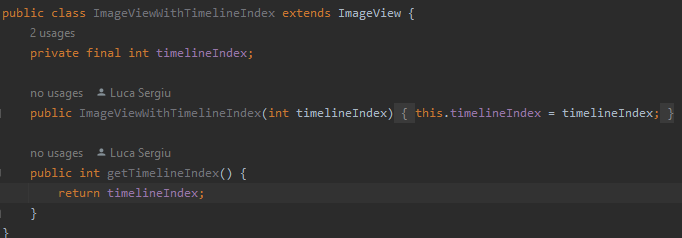
## Chain of Responsability

This design pattern allows a request to be passed through a series of objects until one of the objects is able to handle it. This pattern is useful to avoid binding an object to multiple other objects that might handle the request.

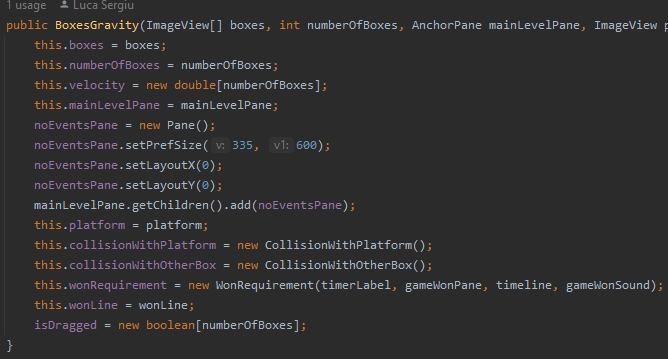
In this game I used Chain of Responsibility to create the gravitational pulse of the boxes:



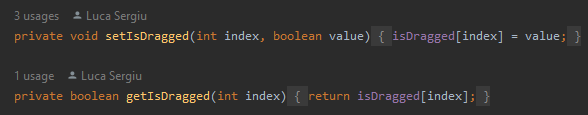
I created a new class that initializes an ImageView with its own index to make it easier to track the falling boxes and the selected one:



I created the BoxesGravity class that contains all the parameters necessary to initialize the gravity pulse and detect the platform and the Win line:



I created an isDragged variable that stores whether or not a box is being moved by the player:



The moveBoxes function performs the gravitational pulse and detects if the box is in collision with the platform, another box or is moved by the player. If the box collides or is moved by the player, its gravitational pulse will stop until the collision disappears or the player decides to release the box:



At the same time, when the boxes are placed correctly one on top of the other, and the tower formed by them reaches and touches the Win line, the You Won window will be displayed and will announce the winning of the level.

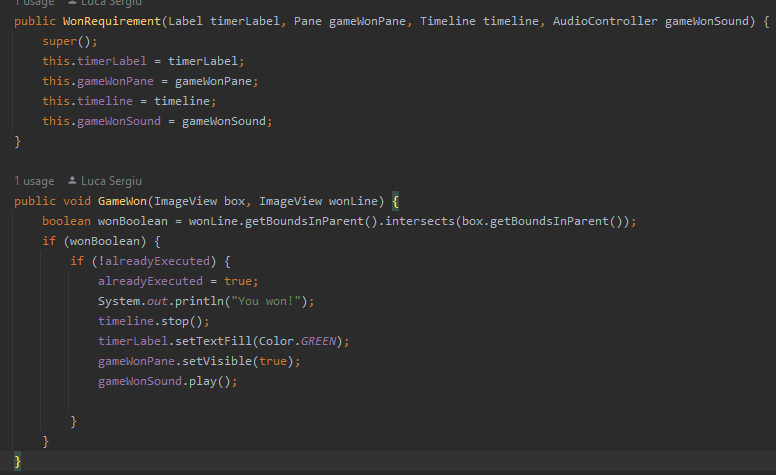
## Command

This design pattern allows encapsulating a request in an object, so that different classes of requests can be parameterized, which can be stored and executed at different times.

In this game I used Command to check the win of the current game.



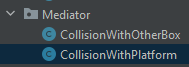
The GameWon function in this class controls if and when the win game window is displayed:



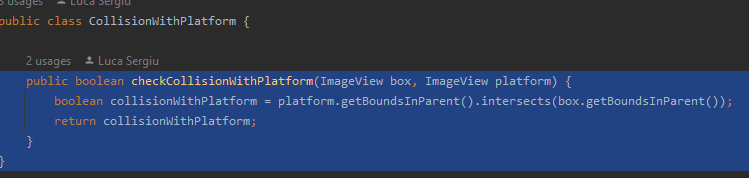
## Mediator

This design pattern allows the centralization of communication between several objects, so that the objects no longer have to communicate directly with each other, but through a mediator object. This pattern is useful to reduce dependencies between objects and make the system more flexible.

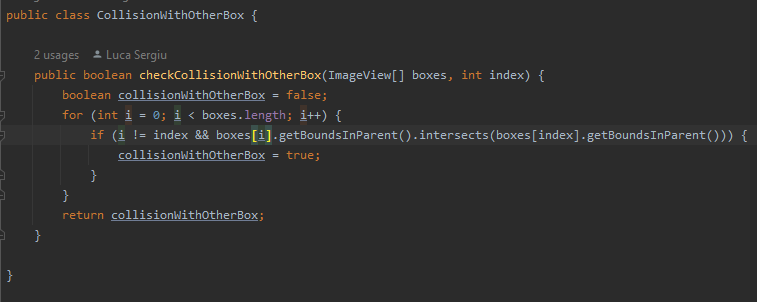
In this game I used Mediator to detect the collision of a box with a platform or with another box.



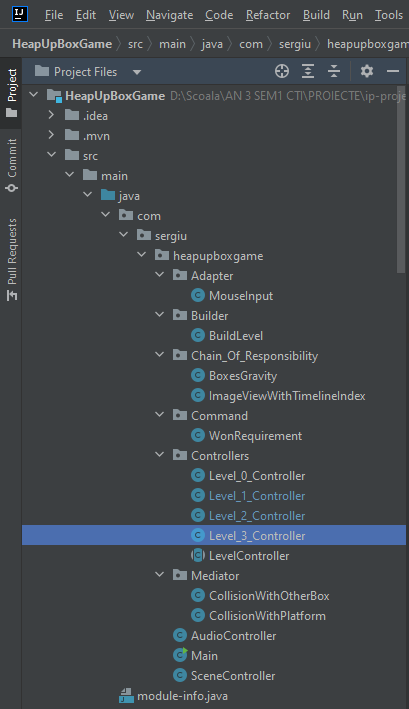
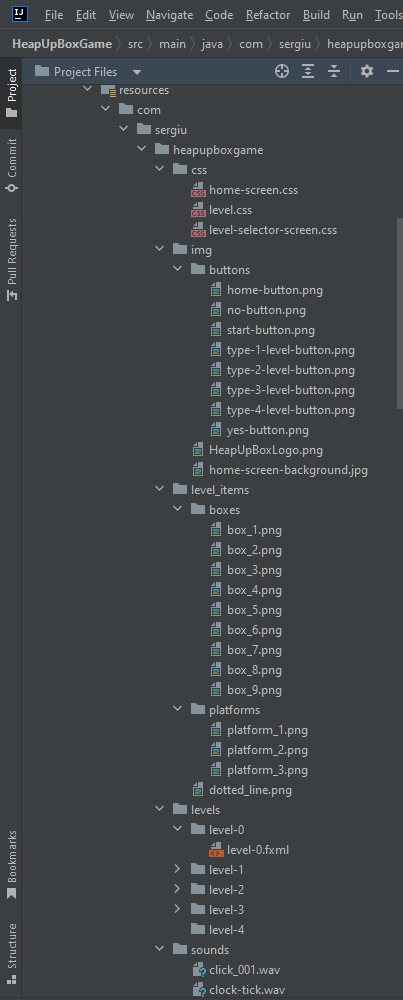
The function that detects the collision with the platform will return a Boolean variable and will notify the rest of the objects that depend on this collision that it has occurred:

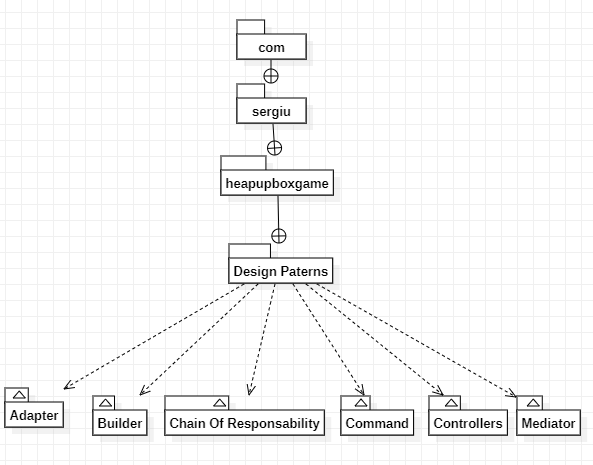


The function that detects the collision with another box will return a boolean variable and will notify the rest of the boxes that depend on this collision that it has occurred:



# System architecture and structure



# Calculation of the stability index

To calculate the stability index for each package in a project:

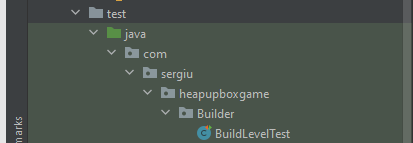
1. Identify all project packages and define the dependencies between them

Dependencies between packages in a project refer to how different packages in that project affect each other. Packages depend on each other when one package requires functionality or data provided by another package to function properly.

1. Use a dependency analysis method, such as a dependency diagram or dependency matrix, to identify the degree of interdependence between packages.
2. Calculate the stability index for each package using an appropriate method.
3. For each package, calculate the McCabe or Halstead methods to measure the complexity of the functions and methods in the package.
4. Use this complexity measure to calculate the stability index for each package.
5. Interpret the results to determine which packages are most stable and which are most vulnerable to change.
6. Use this information to make decisions about project architecture and design and to manage risks associated with future changes.

# The test plan

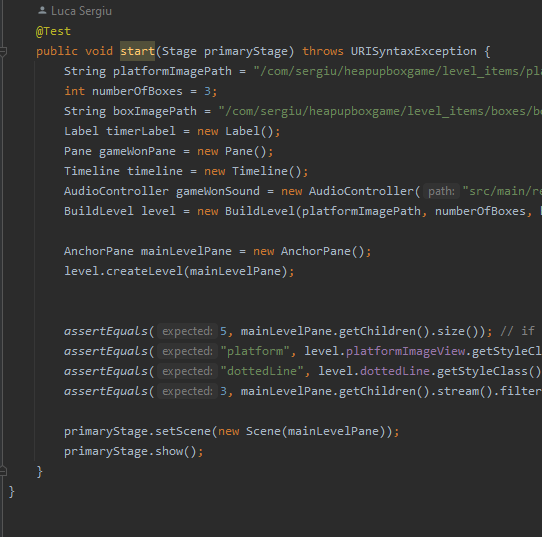
In this example, a test method called BuildLevelTest is created to test if the elements added to the scene during level initialization are the correct ones.



The test method creates a BuildLevel object and assigns predefined parameters to it in the variables platformImagePath, numberOfBoxes, boxImagePath, timerLabel, gameWonPane, timeline, gemWonSound. The test then uses the assertEquals method in JUnit to check if these elements have been created properly and are displayed in the scene.

If the tests pass successfully, no error will be displayed and the interface will be run and can be played.

If the tests do not pass successfully, the errors that were caused will be displayed in the editor's console.

\

# List of tested cases

The tested cases for this application are:

* Case 1:

It is tested and verified if all the elements have been added to the scene.

* Case 2:

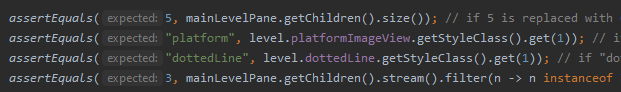
It is tested and verified if the created platform has the correct class for CSS styles.

* Case 3:

It is tested and verified if the created dotted line has the correct class for CSS styles.

* Case 4:

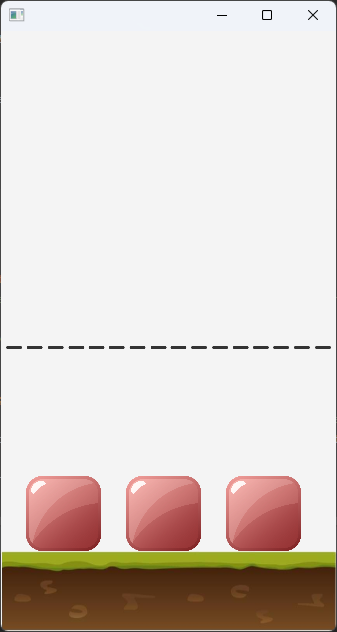
It is tested and verified if the created boxes have the correct class for CSS styles.



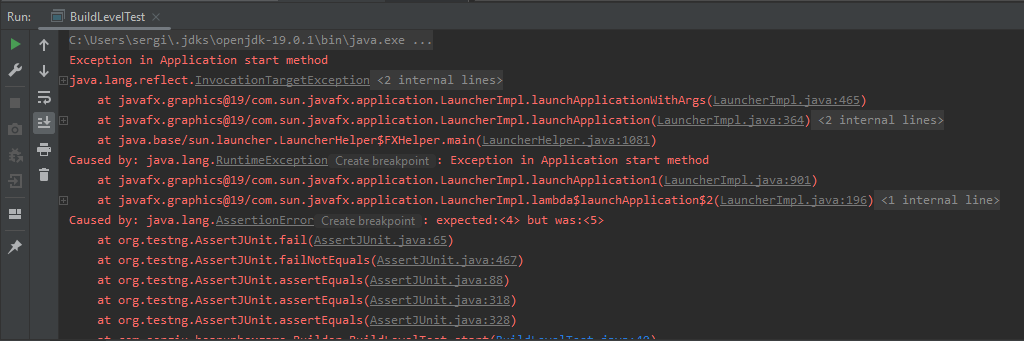


# Test results

If all the tests were completed successfully, no error will be displayed in the console and a test level will be created:



If the tests were NOT completed successfully, an error will be displayed in the console and a test level will NOT be created:



# Bibliography resources and libraries

1. <https://www.java.com/en/>
2. <https://openjfx.io/>
3. <https://gluonhq.com/products/scene-builder/>
4. <https://kenney.nl/assets>
5. <https://opengameart.org/>
6. <https://staruml.io/>
7. IDE: <https://www.jetbrains.com/idea/>
8. GIT Version Control: <https://github.com/lucaay/>
9. GIT Client: <https://www.syntevo.com/smartgit/>