VIETNAM NATIONAL UNIVERSITY – HCM INTERNATIONAL UNIVERSITY

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

OBJECT - ORIENTED PROGRAMMING

PROJECT REPORT

Semester 1, 2023 - 2024

Instructor: Tran Thanh Tung, Nguyen Quang Phu

Topic: Chicken Invaders

Team: JuniorDevs
Class: ITIT22UN21
Github repository: link
17 pages, 10 figures

Ho Chi Minh City, January 2024

TEAM MEMBERS

No.	Name	Student IDs	Contribution
1	Nguyen Nhat Khiem	ITDSIU21091	25%
2	Nguyen Tri Tin	ITDSIU21123	25%
3	Le Thi Thuy Nga	ITDSIU21024	25%
4	Nguyen Hong Son	ITDSIU21117	25%

Ho Chi Minh City, January 2024

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION **3**
I. Abstract 3
II. Overview 3
III. Project objectives 3
IV. Specifications 4
CHAPTER 2: PROJECT TIMELINE **5**
CHAPTER 3: METHODOLOGY **6**
I. UML Diagram 6
II. Project Structure 7
III. User Interfaces 8
IV. Programming 8
V. OOP concepts implementation 8
A. Encapsulation 8
B. Inheritance 9
C. Abstraction 9
D. Polymorphism 9
E. Design pattern 9
CHAPTER 4: CONCLUSION **9**
I. Accomplishment 9
II. Difficulties 10
III. Future works 10
IV. Final words 10
V. References 10

CHAPTER 1: INTRODUCTION

I. Abstract

This project presents the development of a childhood arcade game inspired by the classic "Chicken Invaders," with a focus on object-oriented programming (OOP) principles. The game adheres to OOP best practices, illustrating a well-structured code base and effective class hierarchy. It features a unique twist on the familiar gameplay, introducing up-to-date mechanics and enhancements designed to create an appealing player experience.

II. Overview

Inspired by the iconic arcade game "Chicken Invaders," this project reimagines the classic experience with fresh mechanics and object-oriented programming (OOP) principles at its core. Let yourself immerse in fast-paced action as you defend against waves of invading enemy chickens, each presenting unique challenges and requiring strategic planning.
Beyond the familiar:

- Evolved enemies: Prepare for surprise attacks, formations, and special abilities unlike anything encountered in the original.
- Power-up arsenal: Unleash a diverse range of offensive and defensive power-ups, strategically adapting your tactics to conquer each wave.

Application of OOP:

- Making use of the power of object-oriented design in action with a well-structured code base, promoting maintainability and future expansion.
- Learn and apply key OOP concepts like inheritance, polymorphism, encapsulation and abstract through game implementation.

III. Project objectives

This project focuses on solidifying your understanding and practical application of Object-Oriented Programming (OOP) principles through the development of a game. By the end of this project, we should be able to:

1. **Concept Mastery:**

- Encapsulation:
    + Design classes with well-defined boundaries between data and methods.
    + Implement access control mechanisms to protect data integrity.
    + Understand the benefits of data hiding and abstraction.
- Inheritance:
    + Create class hierarchies that capture relationships between different entities in the game world.
    + Utilize inheritance to reuse code and promote maintainability.
    + Master the concepts of superclasses, subclasses, and overriding methods.

- Polymorphism:
    + Design methods that can handle different types of objects at runtime.
    + Implement virtual methods and method overriding for dynamic behavior.
    + Leverage polymorphism to create flexible and extensible game mechanics.
- Abstraction:
    + Identify and define interfaces between game components to decouple them.
    + Focus on essential characteristics and functionalities while hiding implementation details.
    + Apply abstraction to increase code reusability and simplify program structure.

2. **Design & Development Skills:**

- Object-Oriented Design:
    + Apply OOP principles to create a well-structured and modular game architecture.
    + Design classes that represent real-world entities in the game world.
    + Define relationships and interactions between different game objects.
- Game Development Techniques:
    + Implement core game mechanics using OOP concepts.
    + Integrate user input and handle events within the game loop.
    + Design and implement a visually appealing and engaging game interface.
- Software Testing & Refactoring:
    + Test our game code thoroughly to identify and fix bugs.
    + Apply refactoring techniques to improve code readability and maintainability.
    + Learn the best practices for writing clean and efficient OOP code.

3. **Overall Understanding:**

- Gain a deeper understanding of the principles and benefits of OOP in game development.
- Appreciate the impact of OOP on software design, organization, and maintainability.
- Build a strong foundation for future projects involving OOP and game development.

4. **Bonus objectives:**

- Implement advanced OOP concepts such as interfaces, abstract classes, and generics.
- Explore additional features and mechanics to enhance our game's depth and complexity.
- Share our finished game with others and receive feedback on our design and implementation.

IV. Specifications

Software	Purpose
JDK 21	for running Java programs
JavaFx	main Java library for creating the game
SceneBuilder	make designing Fxml files easier
Aseprite	create in game entity images
Apache Maven	Java project builder

CHAPTER 2: PROJECT TIMELINE

Stage	Action	Member	Week No.
Planning	Individual’s topic research	All	1
	Set up project timeline	Tin	
	Topic confirmation	All	
Project preparation	Create Github repository	Tin	2
	Build and managing project files structure	Tin	
	Research for tutorial and sample code	Khiem, Nga	
	Research for references and technical documents	All	
	Agree on means of communication, workflow and tools	All	
	Decide scopes, aims, goals and requirements of the project	All	
Demo version 1	Making the first game demo	Khiem, Nga	3 - 4
	Evaluate and testing	Son	
	Research for in-game characters and game mechanisms	Tin	
User Interface	Research for game concept	Tin, Nga	5 - 7
	Agree on means of game concept	All	
	Create sprites for in-game elements	Tin, Nga	
	Design layout for Fxml files	Nga	
	Create sprites for in-game entities	Nga	
	Evaluate and make adjustment	Tin	
Demo version 2	Implement JavaFx Fxml loader	All	8 - 11
	Create SceneController	Son	
	Create Entity class and its children	Khiem	
	Create GameController	Tin	
	Create and Test SfxController	Son	

	Evaluate and test for potential bugs	Son	
	Finalize and clean up code	All	
Presentation	Final report	Khiem, Tin, Nga	12 - 14
	Presentation slide	Son	

CHAPTER 3: METHODOLOGY

I. UML Diagram

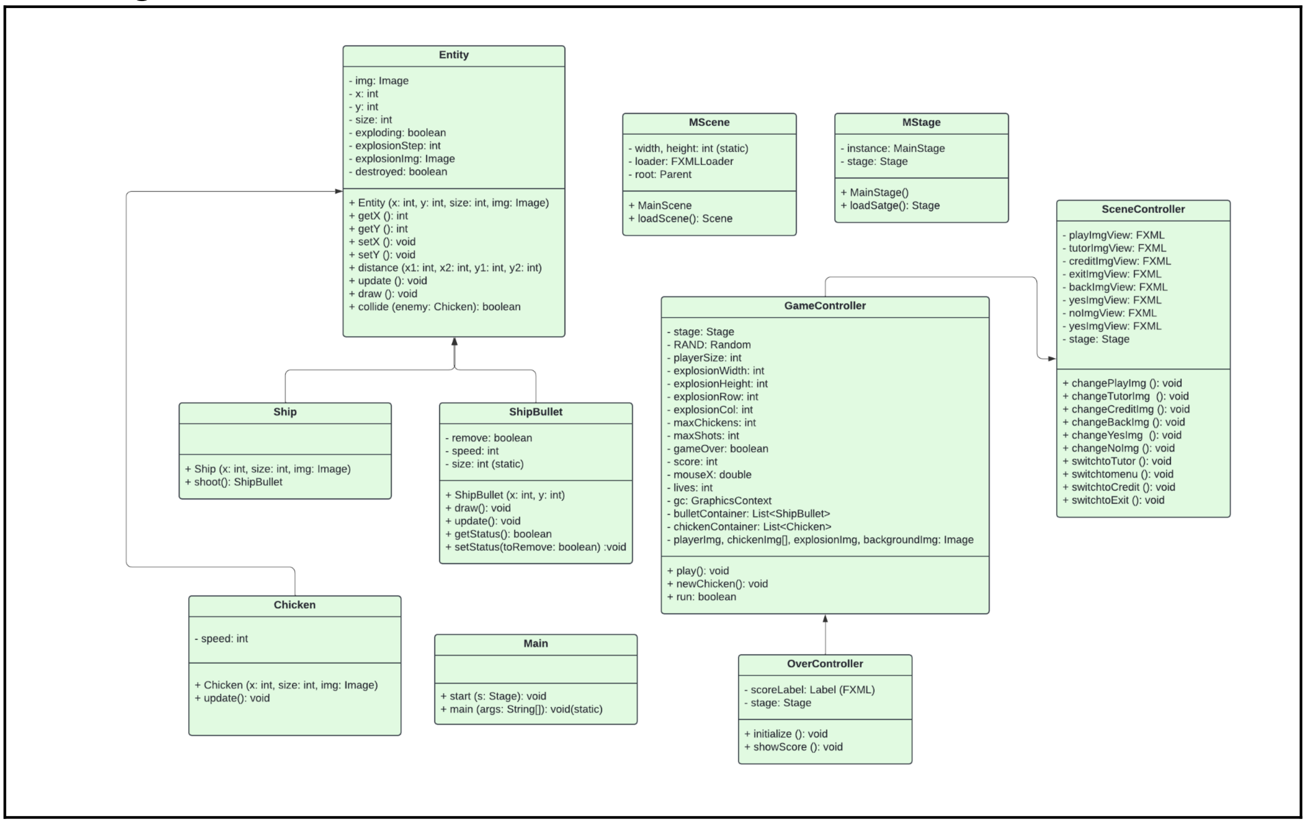


Figure 1. UML diagram

Entity:

- Acts as the superclass for all game objects with shared attributes and behaviors which includes properties like position, velocity, and collision detection, spawning enemy chicken.

Ship:

- Inherits from Entity and represents the player-controlled object.
- Possesses an additional method Shoot for firing bullets.
- Interacting with other entities.

ShipBullet:

- Inherits from Entity and represents projectiles fired by the Ship which has attributes like speed, size, and remove (boolean to check whether the shot is deleted or not), and methods for movement, collision detection, and impact effects.

Chicken:

- Inherits from Entity and represents the enemy units in the game which contains specific attributes like movement speed.

Main:

- Serves as the entry point for the game application, handles initialization, and basic setting of the game stage.

MScene:

- Simplifies the process of scene initialization .
- May handle scene basic settings like width, height and path to Fxml files.

MStage:

- Represents the only stage of the game, provides methods to get and load within other classes.

GameController:

- Extends both StageController and OverController, controlling most of the game mechanisms.
- Handles game state transitions (playing, paused, game over), scorekeeping, and overall game progress.
- Interacts with various classes StageController, MScene, Ship, and Chickens.

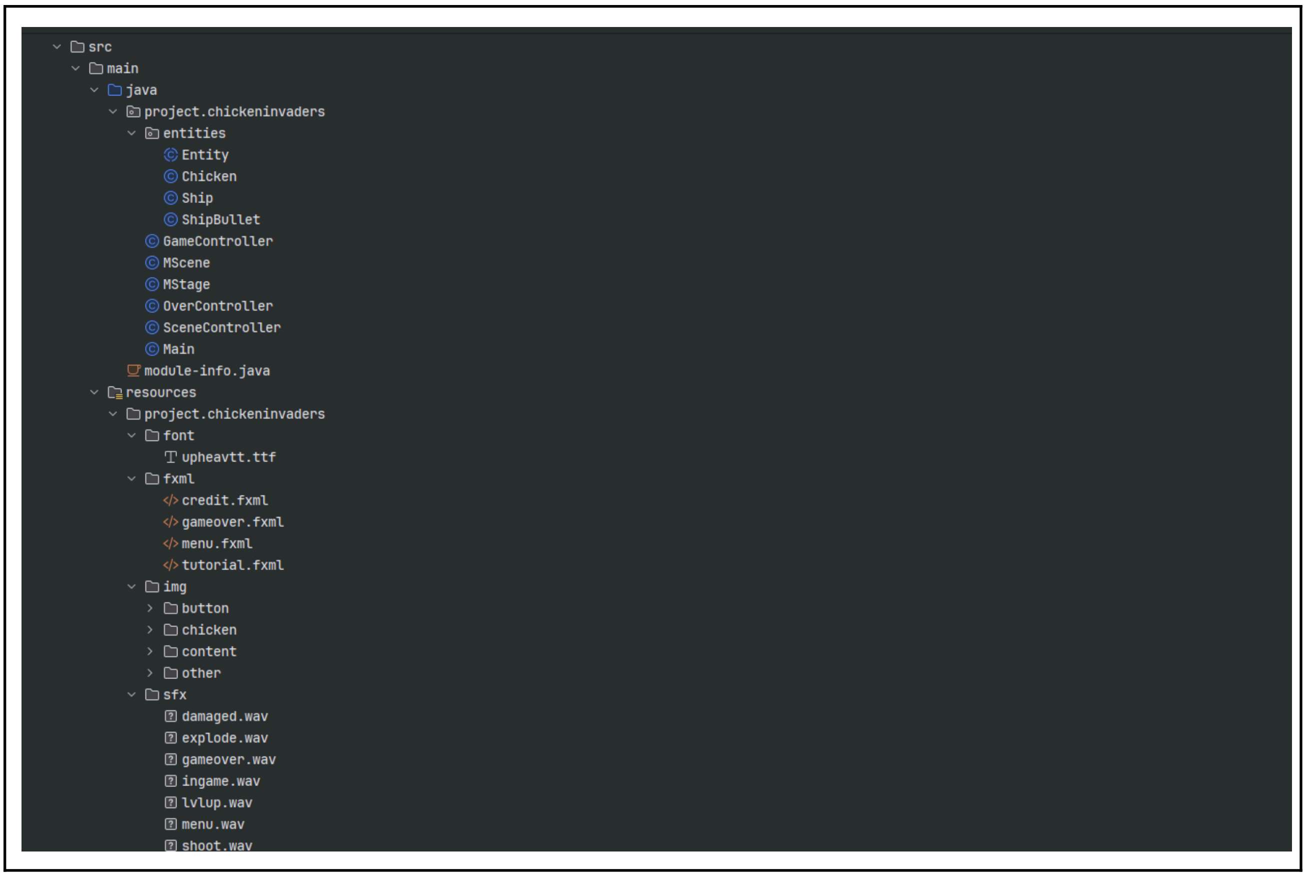
SceneController:

- Responsible for manipulating most of the scene interactivities, handling scene transition
- Interacting with classes like MScene, Chicken, and GameController.

OverController:

- Handles the game over logic, displays scoreboards, and offers options to restart or quit.
- Interacts with classes GameController and Main.

II. Project Structure



```
src
  main
    java
      project.chickeninvaders
        entities
          Entity
          Chicken
          Ship
          ShipBullet
        GameController
        MScene
        MStage
        OverController
        SceneController
        Main
      module-info.java
    resources
      project.chickeninvaders
        font
          upheavtt.ttf
        fxml
          credit.fxml
          gameover.fxml
          menu.fxml
          tutorial.fxml
        img
          button
          chicken
          content
          other
        sfx
          damaged.wav
          explode.wav
          gameover.wav
          ingame.wav
          lvlup.wav
          menu.wav
          shoot.wav
```

Figure 2. Project structure

The default configuration of this project is created with JavaFx generator and Maven build and it consists of two main directories: java and resources. The java directory stores all the classes, while the resources directory holds the other asset files, such as images and Fxml files. We can notice that there is an outer directory named “project.chickeninvaders” that encloses both directories. This will later on enable the java.lang.Class methods to access files across different packages without importing them.

III. User Interfaces

One of the most challenging parts of making this game is coming up with a good enough design to make the game feel unique. Luckily, through an iterative process of experimentation with various artistic styles, we found that the most straightforward approach to tackle these problems was to make everything in pixels. Hence, in the end, we decided to transform our initial concept to a pixel-based version of the game instead of sticking to the original.

This idea was made possible thanks to SceneBuilder and Aseprite. Scene Builder is a visual layout tool specifically designed for building user interfaces (UIs) for JavaFX applications. It allows users to drag and drop UI components with a range of modification options, then transform those components into FXML code. This makes it relatively easy for any beginner JavaFx developers that wants to design UIs themself. However, it wouldn't be complete without the help from Aseprite - a pixel art editor designed for creating animated 2D sprites and graphics for games.

Upon launching the game, the user is first presented with the primary navigation interface, known as the "main menu screen". This screen comprises five discrete interactive elements: a prominent title proclaiming the game's name, followed by four action-oriented buttons labeled "Start," "Tutorial," "Credits," and "Exit." Upon hovering the mouse cursor over any of these buttons, it gets highlighted in order to indirectly ask the players to confirm their choice.

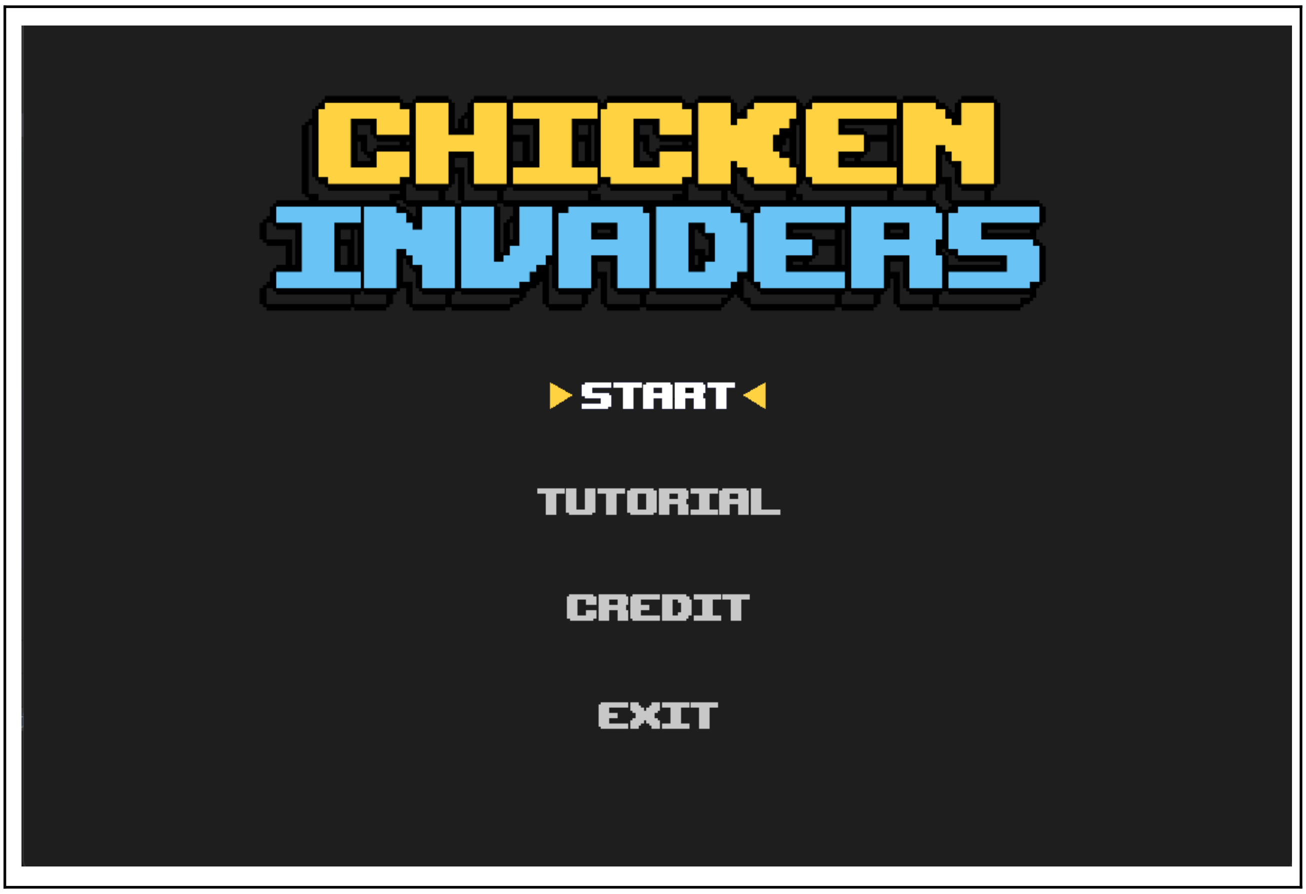


Figure 3. Main menu screen

Pressing the "Start" button initiates a transition to the gameplay screen. Simultaneously, the system spawns ten randomly chosen "chicken" objects on top of the player's ship.Each instance of these entities utilizes a model selected at random from a predefined pool, thereby enhancing the overall gameplay diversity. It is noteworthy that these models were individually hand-crafted, taking inspiration from a popular game - “Angry Birds”.

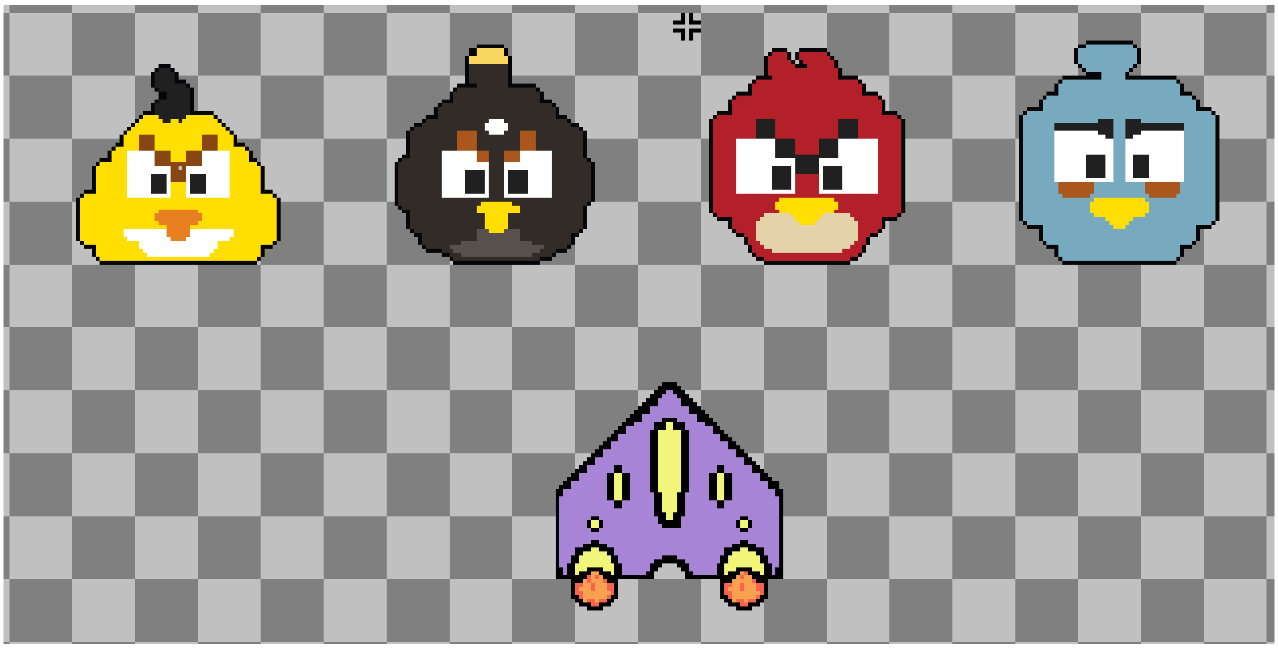


Figure 4. Entity models

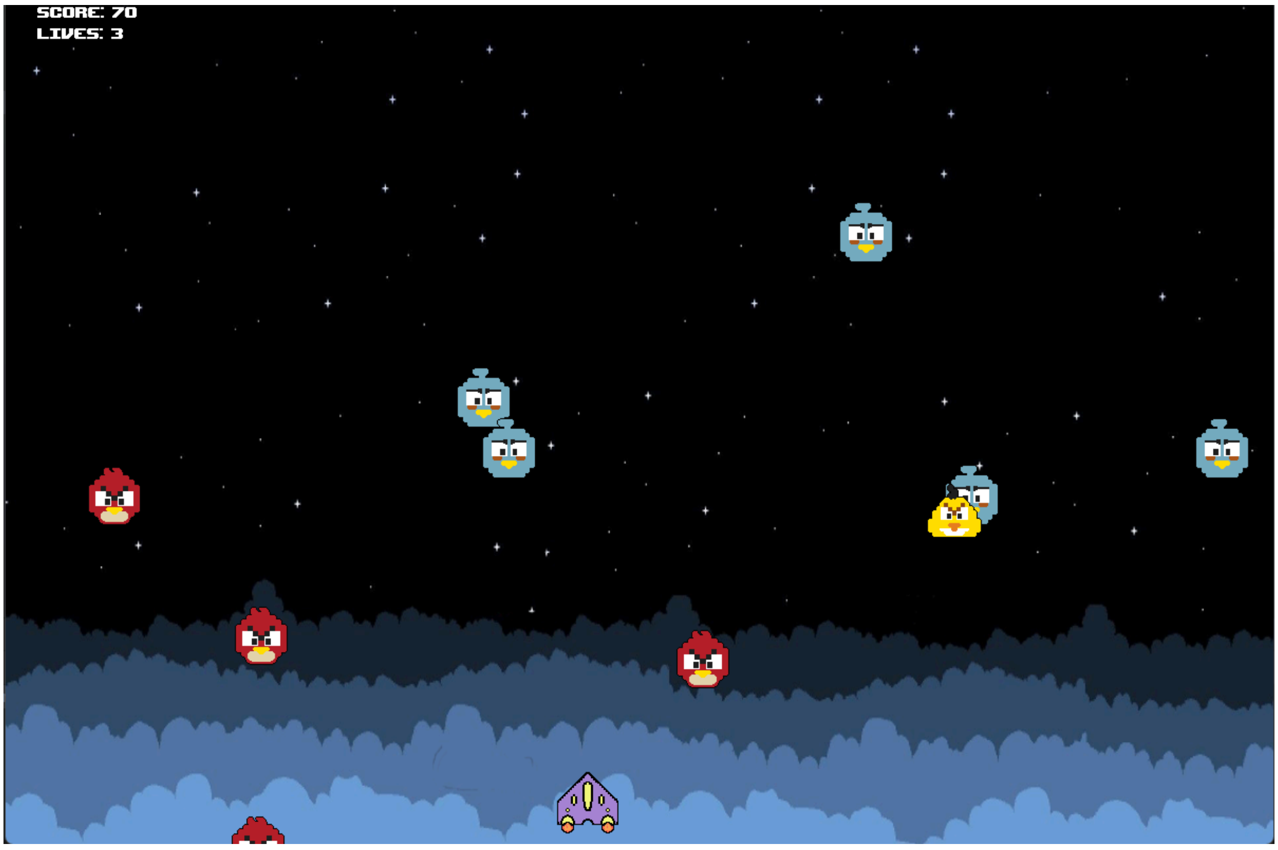


Figure 5. Gameplay

The "game over" menu triggers after the player gets defeated. It then asks the player whether they want to keep playing or go back to the main menu. To continue, they click "YES"; to return to the main menu, they click "NO". These options will also get highlighted when they get hovered.

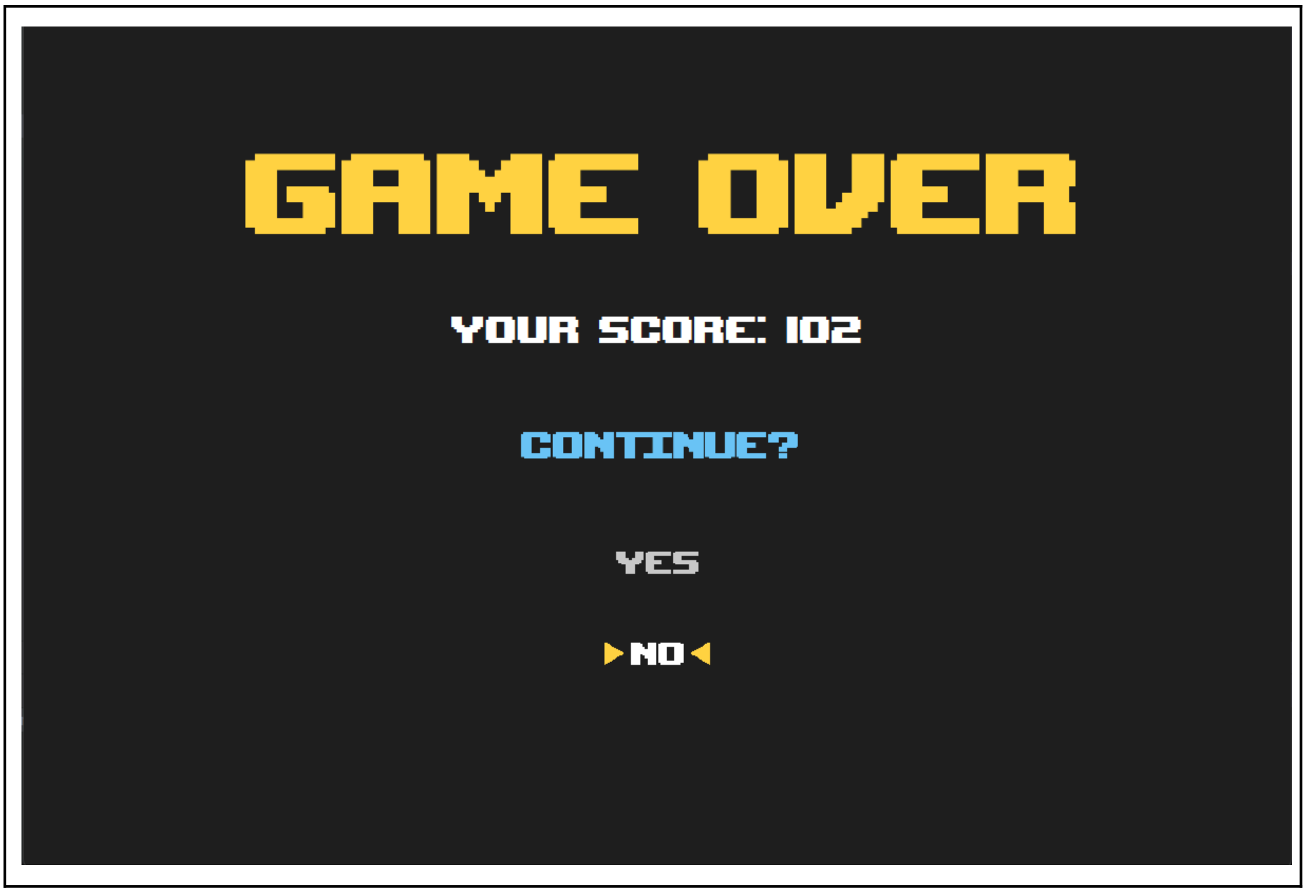


Figure 6. Game over

IV. Programming

A. Version 1

In the initial phase of development, our team ambitiously set out to create a game drawing inspiration from the popular Chicken Invaders, featuring three distinct levels designed to offer players a progressively challenging experience. However, as we moved forward and completed our first demo, we encountered a series of unexpected obstacles that stemmed from our original concept.

One of the primary issues we faced was with the background layer mechanics. Specifically, in the second level, we noticed a critical flaw: the chickens, which are central to the gameplay, were not respawning correctly. This incomplete respawn not only affected the gameplay dynamics of the second level but also had a cascading effect on the subsequent level. The scoring system, which is a pivotal aspect of the game, became inconsistent. The reason for this inconsistency was traced back to the errors in the chicken respawn mechanism in the second stage, which led to irregularities in the number of chicken enemies present and thus, varied the scores unfairly for the player.

Moreover, our initial code architecture for the first demo was not as refined as we had hoped. This lack of refinement in the codebase led to a multitude of bugs that significantly hampered the gameplay experience. These bugs were not mere minor glitches; they profoundly impacted the overall quality of the game, detracting from the player's enjoyment and engagement.

Confronted with these challenges, our team had to make a strategic decision. We realized that maintaining the original structure of three levels, each with its own set of background layers, was contributing to the complexity and the ensuing errors. To mitigate these issues, we opted for a more streamlined approach. We decided to consolidate the game into a single stage, eliminating the three-tier level structure. This decision was not taken lightly, but it was essential to ensure a smoother, more error-free gaming experience.

In executing this revised plan, we employed SceneBuilder, an advanced tool that allowed us to enhance and refine the gameplay within this singular stage. By focusing our efforts on perfecting one well-designed level rather than spreading our resources across three separate stages, we aimed to deliver a game that was not only bug-free but also provided a cohesive and engaging experience for the player. This strategic pivot, though a departure from our initial vision, was a necessary step to uphold the quality and playability of our game.

B. Version 2

After some research we found a different approach from a tutorial on Youtube and decided to use it as a foundation to build upon. However, since it only used one single code file to store all classes, which can lead to practical challenges for debugging and testing. To address this, we adopted an Object-Oriented Programming (OOP) paradigm, segmenting the code into smaller classes with defined functionalities, distributing functionality across multiple classes to enhance code organization and clarity.

The last version lacked Fxml files for loading, which is a serious limitation of the potential for UI design. Therefore, for this version, we implemented Fxml to leverage Scene Builder effectively and create a more comprehensive programming experience. Thus, this also allowed us to extend the initial use of canvas graphics from the source code. Furthermore, we introduced Controller classes to enable interactive elements in the game, enhancing user engagement.

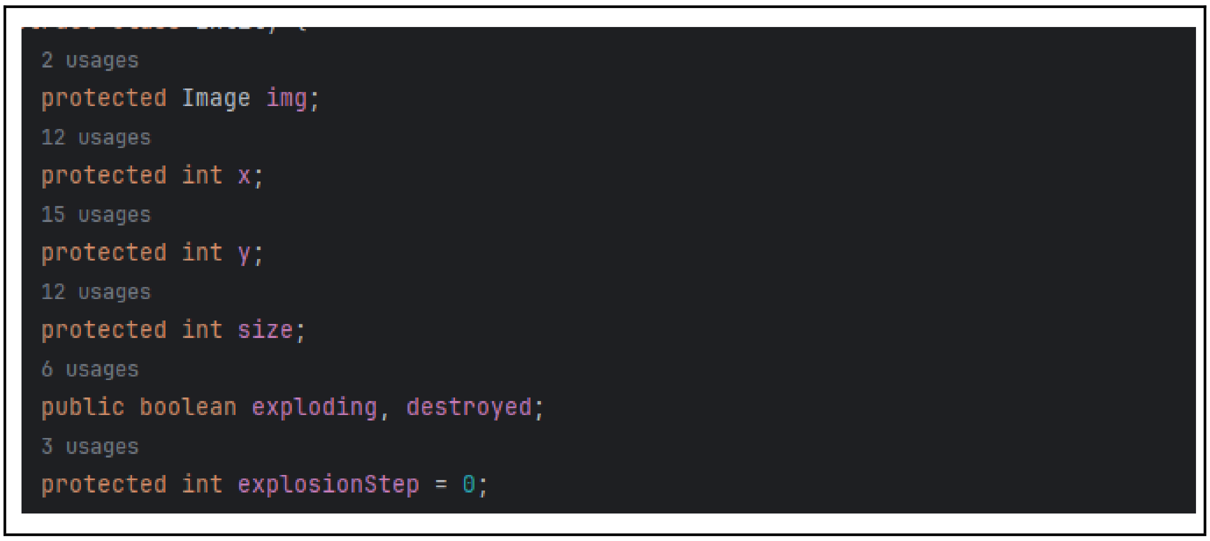
To enhance the game experience, substantial improvements were made to the original features including: encompassing the addition of lives, modifications to bullet firing mechanisms, and the incorporation of menu options, along with game over, replay, and pause functionalities. In addition, we also integrated the “Singleton” design pattern into our game, with a hope of improvement on loading speed and addressing some common design challenges, contributing to a more scalable and maintainable architecture of the project.

In the end, our efforts have not only enriched the game's functionality but also elevated its visual and structural aspects. The combination of FXML integration, custom entity graphics, OOP principles, feature enhancements, and the incorporation of design patterns has resulted in a more sophisticated and user-friendly gaming experience.

V. OOP concepts implementation

This segment dissects the intricate implementation of the four principal concepts of object-oriented programming including: Encapsulation, Inheritance, Abstraction and Polymorphism. Additionally, it examines the implementation of the "Singleton" design pattern with reference to relevant libraries or frameworks if applicable.

A. Encapsulation

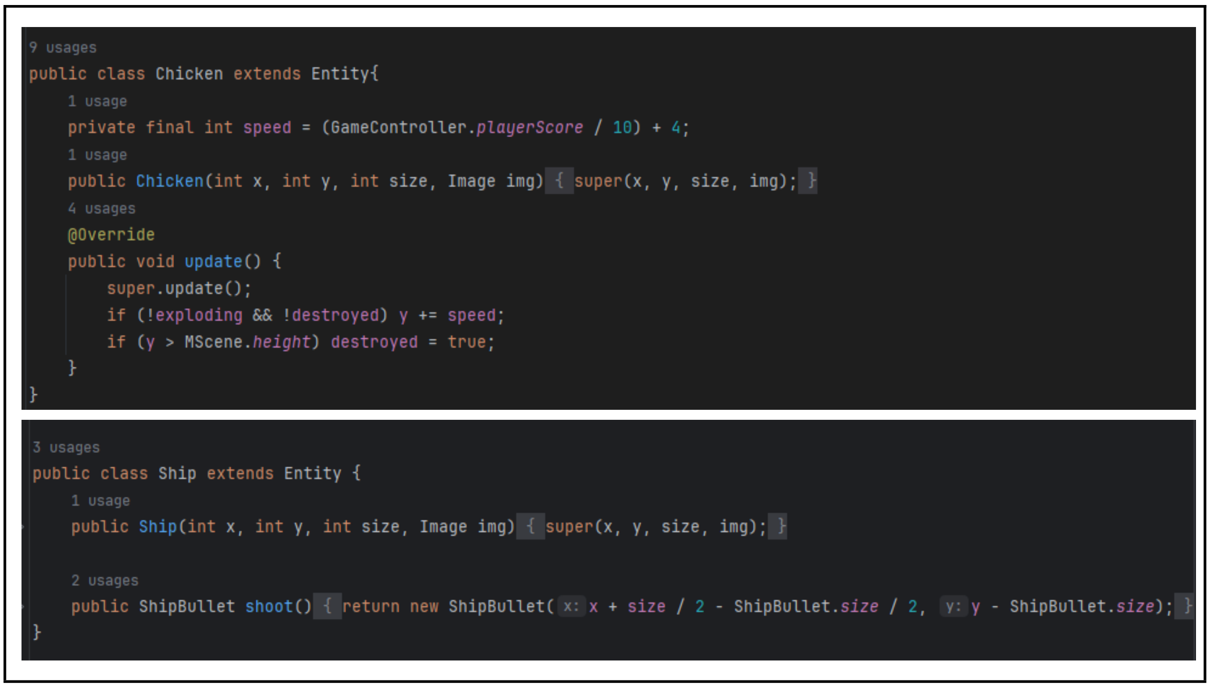


```
2 usages
protected Image img;
12 usages
protected int x;
15 usages
protected int y;
12 usages
protected int size;
6 usages
public boolean exploding, destroyed;
3 usages
protected int explosionStep = 0;
```

Figure 7. Encapsulation usage

By employing access modifiers such as private, protected, or public for class fields, we can ensure encapsulation. This encapsulation prevents unauthorized access from other classes, thereby enhancing the security and integrity of the data within the class. This practice is a fundamental aspect of object-oriented programming and contributes to the robustness and maintainability of the code. In the Figure 7, the data are declared in Entity class and have their accessed levels set to protected and public. By doing this, we ensure that only some data is allowed to access from outside of the class, others are restricted to only classes that are child of the Entity class.

B. Inheritance



```
9 usages
public class Chicken extends Entity{
    1 usage
    private final int speed = (GameController.playerScore / 10) + 4;
    1 usage
    public Chicken(int x, int y, int size, Image img) { super(x, y, size, img); }
    4 usages
    @Override
    public void update() {
        super.update();
        if (!exploding && !destroyed) y += speed;
        if (y > MScene.height) destroyed = true;
    }
}
```

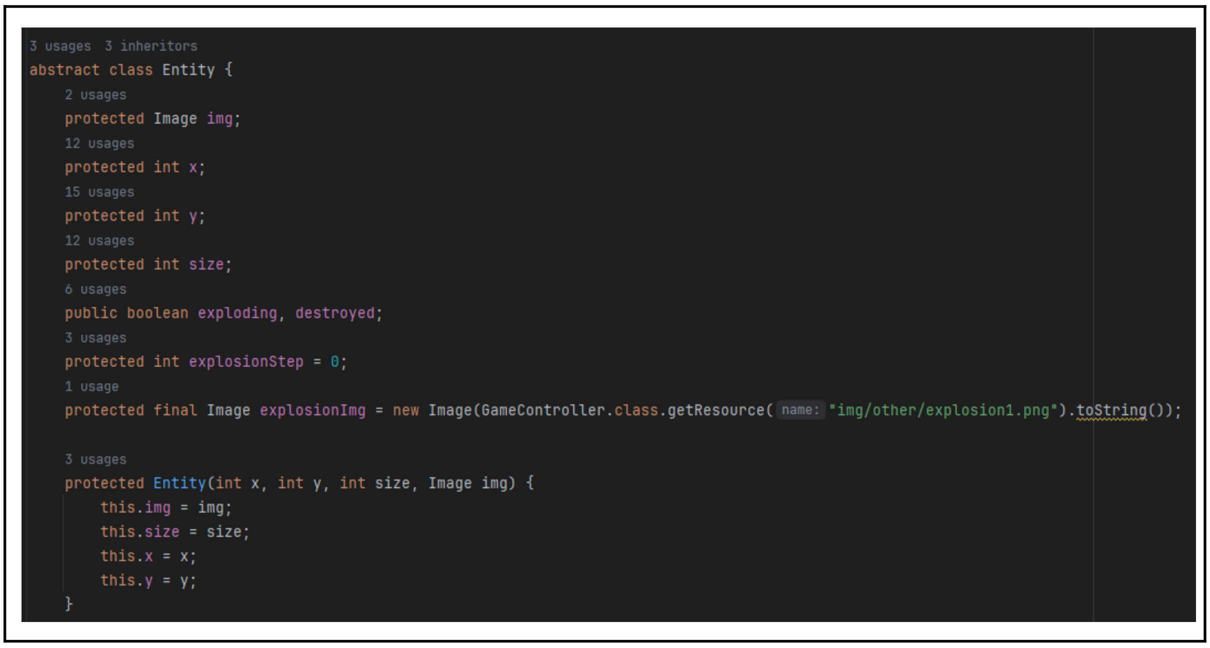
```
3 usages
public class Ship extends Entity {
    1 usage
    public Ship(int x, int y, int size, Image img) { super(x, y, size, img); }

    2 usages
    public ShipBullet shoot() { return new ShipBullet( x: x + size / 2 - ShipBullet.size / 2,  y: y - ShipBullet.size); }
}
```

Figure 8. Inheritance usage

Inheritance is a concept where new classes share the structure and behavior defined in other classes. These classes can also be modified by adding new or overriding methods and fields from their parent. Inheritance represents the IS-A relationship which is also known as a parent-child relationship. Figure 8 is one of the usages from our project, Chicken class extended from class Entity which indicates that Chicken is the subclass and Entity is the super class. Hence, the relationship between the two classes is Chicken IS-A Entity. Furthermore, class Ship also extended from class Entity, therefore the relationship between Ship, Chicken and Entity is a hierarchical inheritance.

C. Abstraction



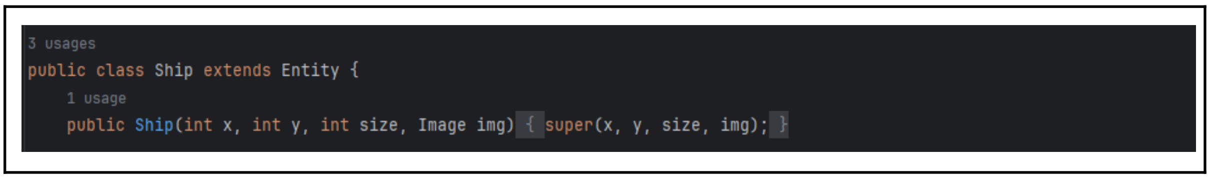
```java
3 usages  3 inheritors
abstract class Entity {
    2 usages
    protected Image img;
    12 usages
    protected int x;
    15 usages
    protected int y;
    12 usages
    protected int size;
    6 usages
    public boolean exploding, destroyed;
    3 usages
    protected int explosionStep = 0;
    1 usage
    protected final Image explosionImg = new Image(GameController.class.getResource( name: "img/other/explosion1.png").toString());

    3 usages
    protected Entity(int x, int y, int size, Image img) {
        this.img = img;
        this.size = size;
        this.x = x;
        this.y = y;
    }
```

Figure 9. Abstraction usage

An abstract class is a class which cannot be instantiated, therefore the creation of an object is not possible with an abstract class, but it can be inherited. It is usually designed to serve as a blueprint for other classes. Java allows an abstract class to have both the regular methods and abstract methods (methods with empty body). This class outlines a common structure and mandates the implementation of certain methods in its subclasses. It ensures that all subclasses maintain a consistent interface while allowing for individualized behavior. Figure 9 shows the abstract Entity which was used as a blueprint for other classes like Chicken and Ship, it contains all of the reusable methods and fields so that others do not have to instantiate again. Understand abstraction, help us to have a cleaner, more precise code structure.

D. Polymorphism

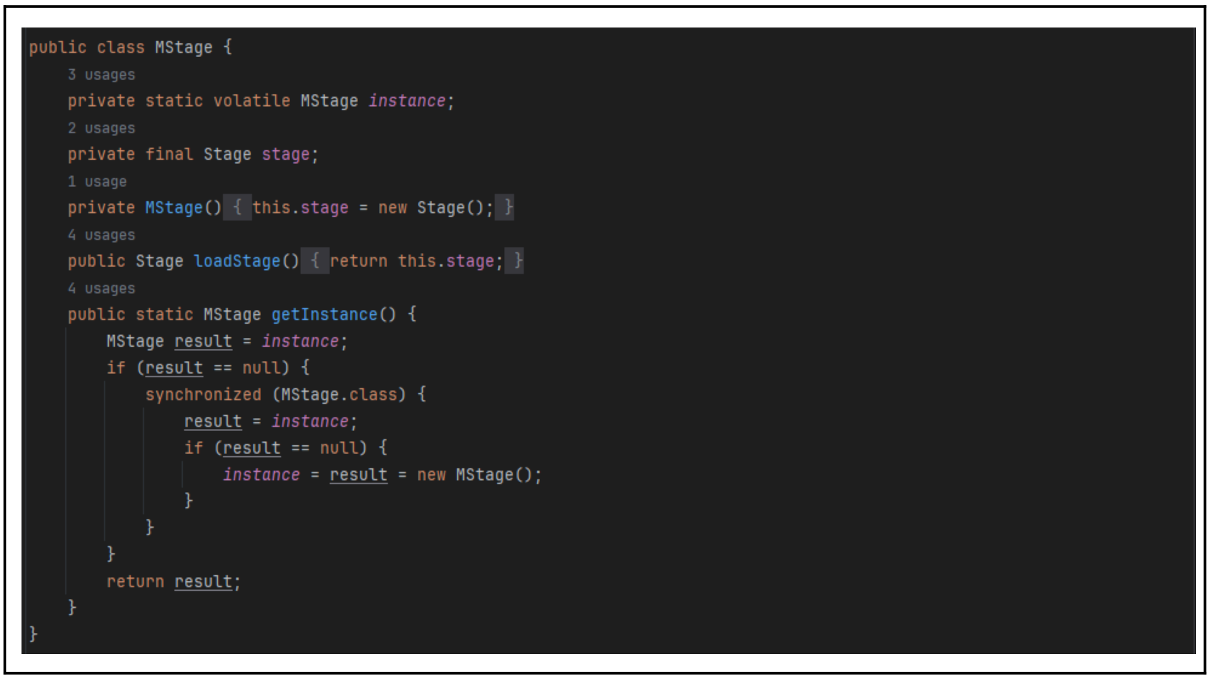


```java
3 usages
public class Ship extends Entity {
    1 usage
    public Ship(int x, int y, int size, Image img) { super(x, y, size, img); }
```

Figure 10. Polymorphism usage

Polymorphism in Java is a concept by which we can perform a single action in different ways. In this sample, the constructor of the "Ship" class in Figure 9 is a constructor polymorphism. It calls the constructor of the superclass "Entity" with its parameters. This allows "Ship" to be initialized with their specific attributes while still being an "Entity".

E. Design pattern



```
public class MStage {
    3 usages
    private static volatile MStage instance;
    2 usages
    private final Stage stage;
    1 usage
    private MStage() { this.stage = new Stage(); }
    4 usages
    public Stage loadStage() { return this.stage; }
    4 usages
    public static MStage getInstance() {
        MStage result = instance;
        if (result == null) {
            synchronized (MStage.class) {
                result = instance;
                if (result == null) {
                    instance = result = new MStage();
                }
            }
        }
        return result;
    }
}
```

Figure 10. Singleton usage

Singleton is one of the 5 design patterns from the Creational Design Pattern group. Singleton will ensure that at any point of time there is only one instance of its kind exits and provide a single point of access to it from any other parts of the application. There are many ways to implement singleton, but in this project we used the Double Check Locking Singleton. Figure 10 shows the implementation of the approach for the "MStage" class. It ensures that only one instance of a particular class exists during the runtime of an application. The "getInstance()" method is provided to check whether an instance of Stage exists or not, if not then it creates a new one, if it does then it returns the instance of that Stage. The "loadStage()" method is then called on the singleton instance of "MStage". This method appears to be responsible for loading the instance of the Stage.

CHAPTER 4: CONCLUSION

I. Accomplishment

- Effective use of OOP: We designed and implemented various classes representing game entities such as Entity, Chicken, and BulletShot. This allowed us to encapsulate related data and behaviors into individual objects, showcasing the core OOP principles of encapsulation, abstraction, polymorphism, and inheritance.

- Game Logic Implementation: Through the use of inheritance and polymorphism, we were able to create a robust game logic. Different types of entities, each a subclass of a general Entity class, could be created and managed, demonstrating the OOP concept of inheritance.

- User Interface Implementation: We developed an interactive and user-friendly interface using Java's GUI libraries with 2d style. This made the game more visually appealing and nostalgic.

- Performance Optimization: By managing game objects effectively and optimizing our code, we ensure that the game runs smoothly, even when the screen is filled with numerous chickens and bullets.

II. Difficulties

- Sound Effect Problems: There are some challenges with the integration of sound effects and background music. There were instances where the sounds did not play as expected or were not properly synchronized with the game events.

- Score Calculation Bug: There was a bug in our scoring system that occasionally led to miscalculations of the player's score. This issue was intermittent and proved difficult to completely resolve within the project timeline.

- Game Simplicity: While the game is functional and entertaining, it is relatively simple in its current state. It lacks some features that could enhance the gameplay experience and make it more engaging.

III. Future works

- Multiplayer Mode: Introducing a two-player mode could add a new dimension to the game, allowing players to compete or cooperate in real-time.

- Screen Splitting: Implementing a split-screen feature would enable multiple players to play simultaneously on the same device, enhancing the multiplayer experience.

- High Score Database: Adding a database to store high scores would allow players to track their progress and compare their scores with others. This could foster a sense of competition and encourage repeated play.

IV. Final words

We would like to express our heartfelt gratitude to our lecturer for their thorough guidance and support throughout our journey. Their expertise and mentorship during our theoria, lab sessions have been essential in equipping us with the necessary skills for success. We deeply appreciate the comprehensive and well-documented resources they provided, which acted as a robust base for our learning and project development. We also extend our commendations to our team members for their collaboration, generous sharing of knowledge, and shared vision. Our efficient teamwork and well-distributed responsibilities have enabled us to work at a swift pace and result in decent results. We are excited to continue on this path, gaining further insights and knowledge.

V. References

1. mostafaHegab. (n.d.). mostafaHegab/Chicken-Invaders: Simple Chicken Invaders Game with JAVA. Retrieved from https://github.com/mostafaHegab/Chicken-Invaders

2. UML class: Lucidchart. (n.d.). Retrieved from https://lucid.app/lucidchart/c6e383ae-e074-4fc5-ae37-65484a9a7a34/edit?invitationId=inv_0a814f23-793c-4559-a23f-30c2dc288661&page=HWEp-vi-RSFO#
3. Best 55+ 8. (n.d.). Retrieved from https://wallpapercave.com/w/wp7872568
4. Programming Space Invaders in Java (fx) Tutorial 1/2. (2019). Retrieved from https://www.youtube.com/watch?v=0szmaHH1hno&list=WL&index=4&t=18s
5. Programming Space Invaders in Java (fx) Tutorial 2/2. (2019). Retrieved from https://www.youtube.com/watch?v=dzcQgv9hqXI
6. (N.d.). Retrieved from https://openjfx.io/#
7. Super Pixel Vintage Gaming Presentation. (n.d.). Retrieved from https://slidesgo.com/theme/super-pixel-vintage-gaming?fbclid=IwAR20XFCnI-9j9m-fXtvFrVnvp8Cf6NYDbKq0FbsinzJ7hjRIAdrVbHhHqkQ