INTERNATIONAL UNIVERSITY
VIETNAM NATIONAL UNIVERSITY HOCHIMINH CITY

SCHOOL OF COMPUTER SCIENCE & ENGINEERING

OBJECT – ORIENTED PROGRAMMING

IT069IU

TOPIC :

SPACE IN VADERS GAME

BY GROUP KIN

Course by Mr. Nguyễn Trung Nghĩa

Lê Tuyết Nhung ITITDK22043
Nguyễn Hà An Thạnh ITITWE22051

Table of Contents

Abstract

The Space Invaders project is an implementation of the classic arcade game, developed using Java programming and the principles of Object-Oriented Programming (OOP). This project explores the application of OOP concepts, including encapsulation, inheritance, and polymorphism, to design and implement a fully functional and engaging game.

Benefits:

- **Relaxation and Entertainment**: Playing Space Invaders offers a fun way to unwind and relax, serving as a mental break from academic or professional tasks.
- **Skill Development**: Designing and implementing the game helps enhance coding skills, problem-solving abilities, and familiarity with OOP principles.
- **Creativity and Design**: The project encourages creative thinking in designing game mechanics, graphics, and player interactions.

Challenges:

- **Time Constraints**: Completing the project within a short two-week timeline required effective time management and prioritization.
- **Time Constraints**: Completing the project within a short two-week timeline required effective time management and prioritization.
- **Game Design Complexity**: Balancing the difficulty level of the game while maintaining user engagement required thoughtful design iterations

Project Goals:

- **Application of OOP**: Leverage OOP principles to design a modular, reusable, and maintainable codebase for the game.
- **Game Development**: Create a functional Space Invaders game that includes player controls, enemy movements, scorekeeping, and game-over conditions.
- **Skill Enhancement**: Gain hands-on experience in Java programming, game development, and project management.

I. INTRODUCTION

1.1 Background

Space Invaders is a legendary arcade game first released in 1978, which has since become an icon in gaming history. Created by Tomohiro Nishikado, it was one of the earliest video games to achieve widespread popularity, setting the stage for the global gaming industry. The game's simple yet engaging mechanics—where players control a cannon to shoot descending alien invaders—have captivated audiences for decades. Its cultural impact extends beyond gaming, inspiring numerous adaptations, merchandise, and references in popular culture.

1.2. Objective

The primary objective of this project is to recreate the classic Space Invaders game using Java programming, incorporating Object-Oriented Programming (OOP) principles. Beyond replicating the original game mechanics, this project aims to enhance educational and technical skills through:

- Applying OOP concepts like inheritance, encapsulation, and polymorphism.
- Developing a structured, modular, and reusable codebase.
- Gaining experience in game design and implementation.

Additionally, the project provides an engaging platform for players to relax and experience nostalgia while demonstrating how classic games can be reimagined with modern programming techniques.

1.3. Report Overview

This report outlines the steps taken to develop the Space Invaders project, with the following main sections:

- **Game Design and Implementation**: Detailing the use of OOP principles and Java programming to build the game's backend.
- **Graphics and User Interface**: Exploring the methods used to create an interactive and visually appealing game interface.
- **Gameplay Mechanics**: Describing the implementation of core features, such as player controls, enemy behavior, and score tracking.

- **Challenges and Solutions**: Discussing the technical and design challenges encountered during the project and how they were addressed.
- **Testing and Debugging**: Summarizing the steps taken to ensure a smooth and functional gaming experience.

II. **TASK TIMELINE & DIVISON**

2.1 Contributions

Name	Reposibility	Contribution
Lê Tuyết Nhung	Idea Research, Coding and Implementation, Project Timeline Coordination, Report Review and Revision, Slide Presentation Design	100%
Nguyễn Hà An Thạnh	UML Designer, Slide Presentation Design, PowerPoint Presentation Support	100%

Table 1. Table of Contribution

2.2 Task Timeline & Division

Week 1: Planning and Design

Idea Development (Both):

- Brainstorm and finalize gameplay ideas and enhancements for Space Invaders.
- Collaboratively decide on game mechanics, features, and structure.

Research and Documentation (Le Tuyet Nhung):

- Conduct research on the original game and propose design improvements or modern adaptations.
- Document the findings and ensure alignment with the project objectives.

UML Diagram Creation (Nguyen Ha An Thanh):

- Design UML diagrams to represent the class structure, attributes, methods, and relationships.

Week 2: Implementation

Coding and Development (Le Tuyet Nhung):

- Write and implement the game code, focusing on OOP principles such as inheritance, encapsulation, and polymorphism.

- Develop core features, including player movement, enemy behaviors, collision detection, and scoring system.

Timeline Management (Le Tuyet Nhung):

- Monitor project progress to ensure tasks are completed within the timeline.
- Provide regular updates and adjust plans if necessary.

Week 3: Finalization

Slide Presentation (Nguyen Ha An Thanh):

- Prepare a slide deck summarizing the project, highlighting key achievements, challenges, and results.

Testing and Debugging (Le Tuyet Nhung):

- Work together to test the game thoroughly, identify bugs, and refine the gameplay.

Report Writing (Le Tuyet Nhung):

- Collaboratively create the final report, detailing the project's background, objectives, implementation, challenges, and outcomes.

III. PROJECT ANALYSIS

1. Literature Review

1.1. Analysis of Existing Games

- **Original Space Invaders (1978):** The classic arcade game introduced mechanics like player-controlled shooting and descending alien waves. Its simplicity and progressive difficulty became a template for future fixed-shooter games.
- **Relevance to the Project:** This project retains the core gameplay but incorporates modern programming principles, aiming to replicate the original experience while exploring enhanced design techniques.

1.2. Advancements in Game Development Tools

- **Modern Capabilities:**
  - The original game relied on custom hardware, while this project utilizes Java and OOP for modular and reusable code.

    - Tools like UML diagrams streamline the planning phase, while integrated development environments (IDEs) enable efficient debugging and testing.
- **Project Application:** Using Java and OOP, this project adopts advancements like class inheritance, encapsulation, and polymorphism to ensure a structured codebase suitable for expanding features.

1.3. Justification of the Project

- **Skill Development:** Provides hands-on experience with OOP concepts and game development, enhancing both technical and creative skills.
- **Innovation:** Balances nostalgia with modern programming by introducing enhancements like smoother animations, dynamic enemy behaviors, and scalable difficulty levels.
- **Contribution:** Demonstrates how classic games can be effectively reimagined using contemporary tools, making it a valuable educational exercise for understanding game mechanics and design.

2. Requirements Analysis

2.1. Functional Requirements

- **Player Movement:** Implement smooth and responsive controls for horizontal movement of the player-controlled spaceship.
- **Shooting Mechanics:** Enable the player to shoot bullets at descending alien invaders.
- **Enemy Waves:** Generate dynamic enemy waves with increasing difficulty levels and unique movement behaviors.
- **Scoring System:** Include a mechanism to track player scores and unlock level progression based on performance.

2.2. Non-Functional Requirements:

- **Smooth Gameplay:** Ensure minimal lag, providing a fluid and enjoyable gaming experience.
- **User Interface:** Develop an intuitive, user-friendly interface that aligns with the classic game design.

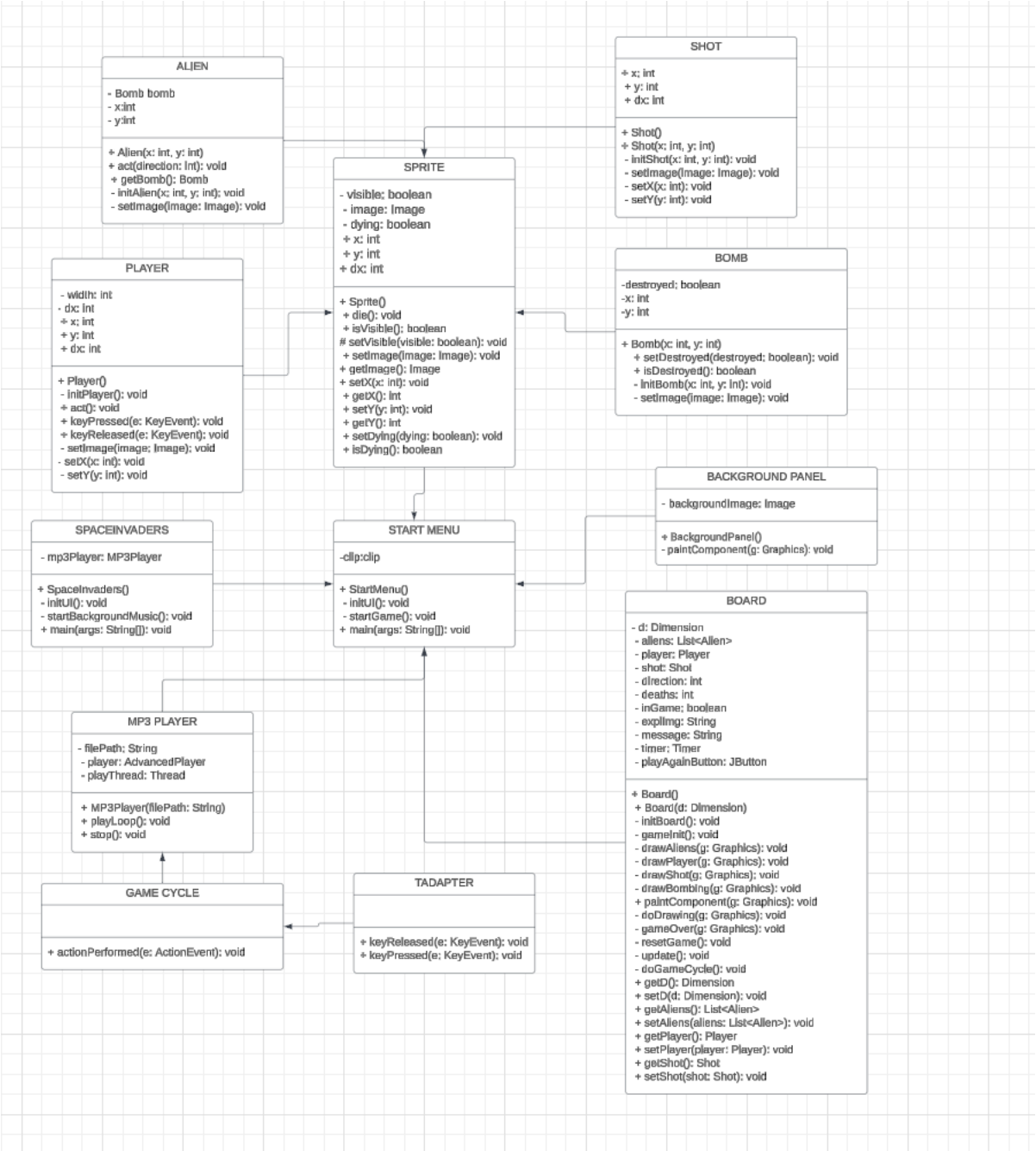
- **Platform Compatibility:** Design the game to run efficiently on the target platform (e.g., PC).

2.3. Technical Requirements

- **Programming Language:** Java, chosen for its robust object-oriented programming capabilities and cross-platform support.
- **Libraries and Tools:** Utilize Java Swing or JavaFX for graphics rendering and event handling.
- **Development Tools:** Use an integrated development environment (e.g., NetBeans, IntelliJ IDEA) for effective project management, debugging, and compilation.

3. Game Design and Architecture

3.1 UML Design



3.2 System Architecture

The game's architecture is designed around several core components:

- **Player**: The player controls the spaceship, which moves left or right and shoots bullets.
- **Alien**: The alien enemies move in patterns and drop bombs.
- **Shot**: Represents the player's bullets fired at aliens.
- **Bomb**: Represents bombs dropped by aliens.

Each component interacts within the game loop, where the player can move, shoot, and interact with the aliens while avoiding their bombs. Components are visually represented with images that are updated on the game screen.

3.3 Gameplay Mechanics

- **Movement Controls**: The player moves the spaceship left or right using the arrow keys. The spaceship's movement is restricted to the screen's width, preventing it from moving off-screen.
- **Shooting and Collision Detection**: The player shoots bullets that move upwards, and the game detects collisions between the bullets and aliens. If a bullet hits an alien, the alien disappears. Additionally, the spaceship can be hit by bombs dropped by aliens, causing the player to lose a life.
- **Enemy Spawning and Attack Patterns**: Aliens spawn at the top of the screen and move in a predetermined pattern. Some aliens drop bombs that move downward, and the aliens' movements and bomb-dropping patterns increase in difficulty over time.

3.4 User Interface Design

The game screen includes:

- **Player**: Positioned at the bottom of the screen, controlled by the user.
- **Aliens**: Positioned at the top in rows, moving horizontally and vertically, with random bomb-dropping behavior.
- **Bullets**: Displayed as small images that move vertically when fired.

- **Score Display**: Positioned at the top-left corner, showing the player's current score based on the number of aliens destroyed.

The layout ensures that the player can clearly see and interact with all elements during the game, with with distinct visual separation between the player, enemies, and projectiles.

4. Implementation

1. Prototyping

Prototyping involves creating a basic version of the game to test core gameplay mechanics without focusing too much on the visual design or final structure. During this stage:

- A simple game prototype was created that focused on the player's ability to move, shoot, and interact with basic game elements like enemies and bullets.
- The goal was to ensure that the basic movement and shooting mechanics worked before adding additional complexity like enemy behavior, collisions, or scoring.

Key points of this stage:

- **Player movement**: Ensuring that the player's spaceship can move left and right based on user input (key presses).
- **Shooting mechanics**: Making sure that the player can shoot bullets that move upward.
- **Basic enemies**: Creating simple enemy sprites and having them appear on the screen.

The prototype's primary aim was to test and refine the mechanics of the game without worrying too much about performance, visuals, or additional features like difficulty scaling.

2. Testing

Testing involved checking individual components to ensure the game worked as intended and refining the code based on feedback or issues that arose. This stage involved:

- **Unit testing** for individual classes (like the Player, Shot, and Alien classes) to verify their functionality, such as image loading, position updating, and input handling.

- **Gameplay testing**: Playing the game to identify bugs or problems in the gameplay experience (e.g., movement bugs, collision detection errors).
- **Bug fixing**: Addressing issues discovered during the testing phase, such as incorrect collision detection or player movement errors.

Feedback from playtesting and internal testing led to several iterations of the game, where mechanics like movement speed, bullet speed, and enemy behavior were tweaked.

3. Iteration

After prototyping and testing, iterative development was employed to refine and expand upon the game. New features were added based on testing feedback, and adjustments were made for balance, difficulty, and gameplay flow.

This included:

- **Improving player controls**: Fine-tuning the speed and responsiveness of the player's spaceship.
- **Enhancing shooting mechanics**: Adding features like the ability to shoot multiple bullets, adjust firing rate, etc.
- **Enemy waves**: Developing more complex enemy behavior and introducing patterns that increase in difficulty as the game progresses.
- **Collision detection and response**: Adding and refining the algorithm to detect when a shot hits an enemy or when the player collides with an enemy.

4.1 Core Features

1. Code Implementation of Player Movement and Shooting Mechanics

- **Player Movement**

```
public void act() {
    x += dx;

    if (x <= 2) {
```

```
            x = 2;
        }

        if (x >= Commons.BOARD_WIDTH - width - 2) {
            x = Commons.BOARD_WIDTH - width - 2;
        }
    }
```

- The act() method controls the player's movement. The player's horizontal position (x) is updated based on the dx value, which represents the direction and speed of movement.
- Boundary checks are implemented to ensure the player doesn't move beyond the left and right edges of the screen.
- The dx value is controlled by keyboard events in keyPressed() and keyReleased(), which allow movement left and right.

- **Player Movement**

```
public class Shot extends Sprite {
    public Shot(int x, int y) {
        initShot(x, y);
    }

    private void initShot(int x, int y) {
        var shotImg = "src/images/shot.png";
        var ii = new ImageIcon(shotImg);
        setImage(ii.getImage());

        int H_SPACE = 6;
        setX(x + H_SPACE);

        int V_SPACE = 1;
        setY(y - V_SPACE);
    }
```

```
}
```

- The Shot class represents a bullet fired by the player. It initializes the shot's position and sets the image for the shot.
- The horizontal (H_SPACE) and vertical (V_SPACE) offsets adjust the bullet's position relative to the player's current position.
- The setX() and setY() methods update the shot's position on the screen.

2. Algorithm for Enemy Wave Behavior and Difficulty Scaling

- **Alien Movement**

```
public void act(int direction) {
    this.x += direction; // Move the alien in the given direction
}
```

- The Alien class represents the enemies, and the act() method moves the aliens in a specified direction, either left or right.
- The direction can be updated to simulate a wave-like movement or random changes as the game progresses.
- Difficulty can be scaled by adjusting the speed of the movement or by introducing new behaviors like dropping bombs.

- **Bomb Dropping (by Aliens)**

```
public class Bomb extends Sprite {
    private boolean destroyed;
    private int x;
    private int y;

    public Bomb(int x, int y) {
        initBomb(x, y);
    }

    private void initBomb(int x, int y) {
        this.x = x;
```

```
            this.y = y;
            this.destroyed = false;
            var bombImg = "src/images/bomb.png";
            var ii = new ImageIcon(bombImg);
            setImage(ii.getImage());
        }
    }
```

- The Bomb class is used by the aliens to drop bombs on the player. Each bomb is initialized with a position, and its state is tracked (whether it's destroyed or not).
- The difficulty can scale by increasing the frequency or speed of bomb drops as the game progresses.

3. Collision Detection and Response System

- **Collision Detection:**

Collision detection is not explicitly shown in code but can be implemented in the Game class or a similar controller that manages interactions between the player, bullets, and enemies. A typical approach for detecting collisions might involve checking if the bounding boxes of the player and enemies overlap.

- The bounding box of each game object (e.g., Player, Shot, Alien) can be calculated using their x, y, width, and height to determine if they intersect.
- When a collision occurs, actions such as destroying the enemy, decreasing health, or removing the bullet can be triggered.

4.2 Tools and Frameworks

- **Pygame or Similar Frameworks**

Java's Swing Library:

While Pygame was mentioned in the section, the project uses Java and Swing (with javax.swing.ImageIcon for image handling). Java's Swing library is being utilized

here for loading images (e.g., ImageIcon), handling user input (via KeyEvent), and displaying graphical components in a window.

- **Image Handling (Player and Alien)**

```
String playerImg = "src/images/player.png";
ImageIcon ii = new ImageIcon(playerImg);
```

The ImageIcon class is used to load images for the player, alien, and shot sprites. Images are displayed by setting the Image to the Sprite object.

- **KeyEvent Handling**

```
public void keyPressed(KeyEvent e) {
    int key = e.getKeyCode();

    if (key == KeyEvent.VK_LEFT) {
        dx = -2;
    }

    if (key == KeyEvent.VK_RIGHT) {
        dx = 2;
    }
}
```

The keyPressed method handles user input for player movement. When the left or right arrow key is pressed, the player's dx value is adjusted to move the player accordingly.

5. Results and Testing

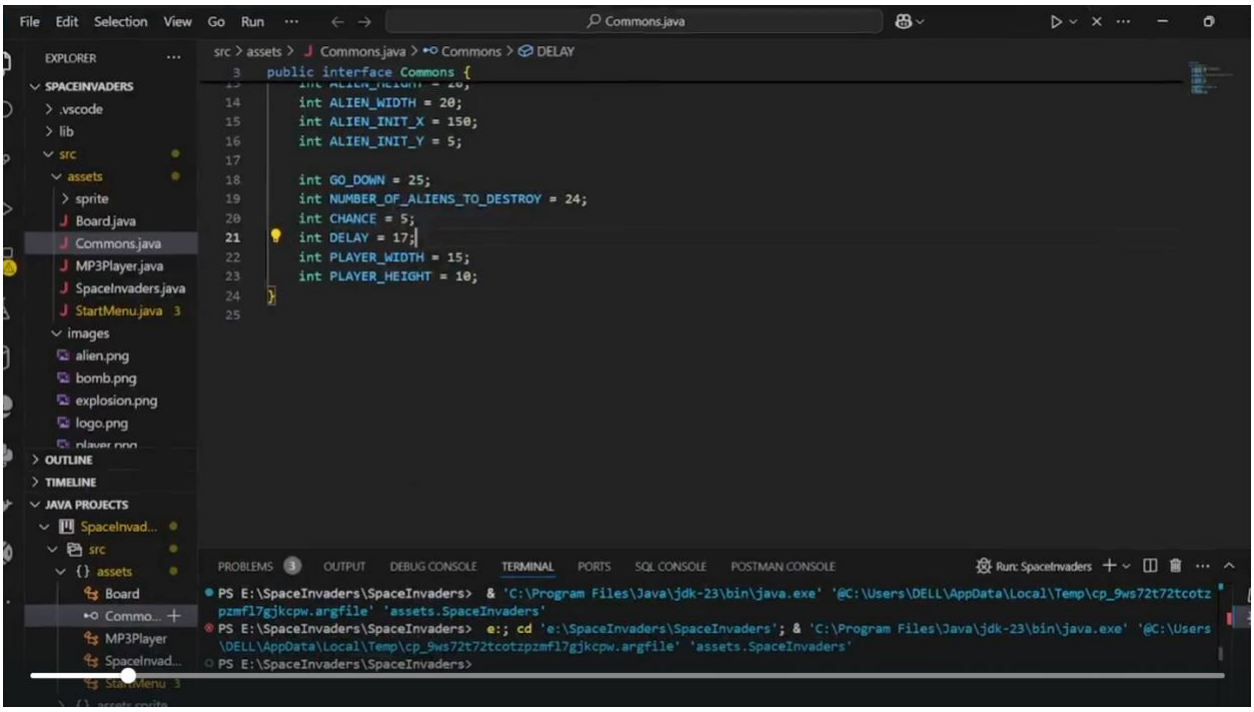


Figure 1. Demo of SpaceInvaders

IV. CONCLUSION

1. Achieved Goals

- **Core Gameplay**: Implemented player movement, shooting mechanics, and enemy behavior.
- **Collision Detection**: Shots hit enemies, causing them to disappear.
- **Enemy Waves**: Progressive difficulty with increasing enemy speed and intensity.
- **Player Controls**: Responsive movement using left and right arrow keys.
- **Basic UI**: Player and enemy sprites are visible with basic functionality.

2. Future Work

- **Enhanced Collision**: Improve collision detection with more precise hitboxes.
- **Difficulty Scaling**: Implement faster or more numerous enemies over time.

- **Power-Ups**: Add items like rapid fire or shields.
- **Sound**: Add sound effects for actions like shooting or enemy destruction.
- **Levels and Bosses**: Create different levels and bosses for added challenge.
- **UI Enhancements**: Add menus, pause button, and better score tracking.
- **Multiplayer**: Introduce a two-player mode.