University of Modena and Reggio Emilia

Enzo Ferrari' Department of Engineering

HeroQuest

Specification of Requirements

I declare that this paper is the result of my personal work, carried out essentially individually and independently.

Paris Luke

Freshman 118307

0. Preface

The following document constitutes the description of the "HeroQuest" software, including Software Requirements Specification (SRS) and appropriate UML diagrams.

The requirements specification was written according to the 'IEEE Recommended Practice for Software Requirements Specification - IEEE Std 830 - 1998', which sets out the guidelines for writing a complete, clear and comprehensive SRS document.

The purpose is the description and specification of software information, such as functionality, constraints and requirements.

INDEX

1. INTRODUCTION

The purpose of this section is to give an overview of the entire Requirements Specification document. The structure of the document is as suggested by the ANSI/IEEE 830 standard known as SRS (Software Requirements Specifications).

1.1 Objective

The purpose of this document is to present, as precisely, consistently, unambiguously and comprehensibly as possible, the general structure of the HeroQuest programme.

It must be borne in mind that the approach followed in developing the software is prototype, so new requirements may be introduced later, in addition to all the constraints that have not yet been identified. This document, like the software, is intended for anyone wishing to use it.

1.2 Scope of Application

The programme developed is HeroQuest, a simple 2D (in two dimensions) horizontally scrolling video game, which offers the player a series of levels with a retro 8-bit style setting.

This product is designed for non-work use. It is intended to analyse only the main functions of the video game.

1.3 Definitions, Acronyms and Abbreviations

Terms	Definitions
HeroQuest, software, video game	Terms used to refer to the subject of this document.
User	Person using the software.
Player, hero	The hero, the character whose movements are controlled by the user. The hero is able to move horizontally, jump, equipped with attacks at short and wide range. It has a certain initial amount of life points and mana points.
Enemy	Set of non-user-controlled characters generated within the various levels. Each inflicts a certain amount of damage to the player if they collide with the player. Some can attack. They can be eliminated by the hero. There are various types of enemies: Nightmare, Skeleton, Skull, etc...
Boss	toughest and most powerful enemy type. If present in the level, the player must eliminate it in order to access the next level of the video game.
Potion	Element generated within the map. If collected by the hero this recovers a certain amount of life points or mana.

Teleportation	Element located at the end of the level, if the hero collides with this accesses the next level of the video game. Sometimes it is only generated in following the elimination of the level boss.
Game	Instance of the game session.
Level	Video game stage. At the end of the level, a screen is shown with scores (enemies defeated, completion time).
HUD	Head-Up Display, during the game it allows you to view your life points and remaining and total mana of the hero.
Map	Place where the hero and enemies move and interact.
GameOver	If the player loses all life points, the level ends.

1.4 Sources

The document was drafted on the basis of the following sources, which were used during the requirements analysis and code implementation:

- Specifications dictated by Paris Luca, software developer.
- IEEE Recommended Practice for Software Requirements Specification IEEE Std 830 - 1998.

1.5 Document Structure

The document herein consists of 5 macro-sections. From this point on, we will find, 4 more chapters:

- **Chapter 2, General Description:** provides a general description of the software, its functionality, special features and the user interface;
- **Chapter 3, SRS:** Contains a presentation of the functional and non-functional requirements of the programme;
- **Chapter 4, Data Model:** shows a general view of how the game works by means of the main UML diagrams;
- **Chapter 5, Design Patterns:** describes the design patterns used during development in order to solve particular problems.

2. GENERAL DESCRIPTION

Let us now describe the main factors affecting the software.

2.1 Software Description

The software presented here is a simple 2D horizontally scrolling video game. The concept behind the game is to impersonate a hero and traverse the various levels by overcoming obstacles, surviving enemies and eliminating the eventual boss at the end of the level.

The hero has an initial amount of life that can decrease by colliding with moving enemies within the level and/or recover with potions. If he loses all life points the level ends automatically and can be recharged. The hero has two types of attack, one short-range (sword attack) and one long-range (fireball) that consumes mana for each use.

At the current stage of development, the video game contains only two levels.

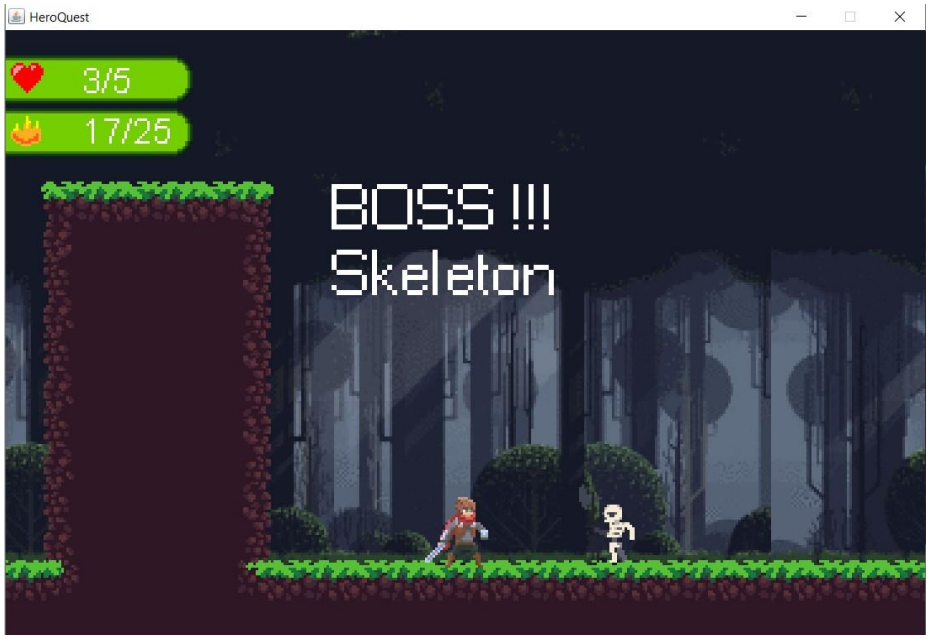


Figure 1: Game screen, skeleton boss.

2.1.1 System/user interface

The user interface should present a very simple main menu, where the user can either start the game, consult the game controls or terminate the execution of the software.

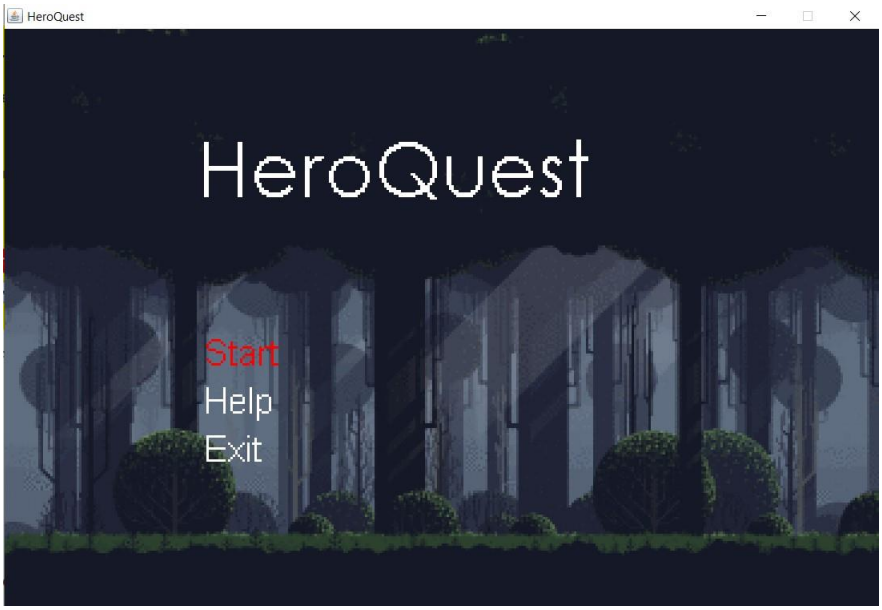


Figure 2: Main menu screen

It should also be possible to pause the video game at any time during the game.

2.1.2 Hardware Interface

The product should only support the keyboard for playing games and navigating menus.

2.1.3 Software Interface

The game will be entirely programmed in JAVA and requires an installed version of JAVA 8.x or higher on your device.

2.1.4 Communication Interface

HeroQuest being a single user application does not require any interface of communication.

2.1.5 Memory occupation constraints (N/A)

2.1.6 Operations

The main operations performed by the software are described in section 2.1.1 of this document.

2.1.7 Installation Constraints

The software requires no special constraints since it is distributed as an executable file and does not require any installation system.

2.2 Macro-Functionality of the System

The main features of HeroQuest are:

- Starting a New Game
- Playing
- Calculating end-of-level scores

2.3 Characteristics of Users

The user does not need any special features to play the game. The commands are described within the software.

2.4 General constraints (N/A)

2.5 Recruitment and Dependencies

Any changes on HeroQuest will affect this document. Any change will require an update of this document.

2.6 Requirements to be analysed in the future

- **Story:** implementation of a story, even a basic one, to make progress between levels more interesting.
- **Music:** implementation of background music and sound effects for the hero's actions.
- **New levels and environments:** implementation of different levels to make the game more varied.
- **Skins:** insertion of different graphic guises for the hero impersonated by the user.

3. FUNCTIONAL AND NON-FUNCTIONAL SYSTEM SPECIFICATIONS

3.1 Functional Requirements

3.1.1 Main Menu Management

RF01	Main menu management.
Introduction	*Actors involved* Player. *General description of the function* The main menu is started automatically when the software is started. The main menu presents the player with the following functions: • Game start-up • Information • Close
Input	*Generic description of data* Press ENTER key (keyboard). Pressing the ARROW UP/ ARROW DOWN keys (keyboard).
Description (Process)	*Sequence of operations* The system must print the game title 'HeroQuest' with the possible interactions to create a new game, display the commands, and exit the software. The key effects are: • UP ARROW/UP ARROW/UP ARROW: Move through the menu. • ENTER: select option in menu.
Output	Depending on the option chosen: • Initialisation of a new game. • Command display. • Termination of software execution.

3.1.2 Initialising a new batch

RF02	Initialise new game, any level.
Introduction	*Actors involved* Player. *General description of the function* A new game is started from the main menu.
Input	None
Description (Process)	*Sequence of operations* The system must: • Load the background and map. • Initialise the player and place it at the start of the map. • Initialise enemies and place them on the map. • Initialise other map elements (potions, teleportation).
Output	A level of the game is started.

3.1.3 Game Management

RF03	Management of any level.
Introduction	*Actors involved* Player. *General description of the function* The player manages the hero's movements on the map. The aim is to overcome the obstacles on the map and defeat the enemies included the eventual boss. Some enemies follow predetermined paths, others will move in the direction of the hero (within a predetermined range). Some enemies attack, others do not.
Input	*Generic description of data* Press keys W, A, D, R, F, ESC (keyboard).
Description (Process)	*Sequence of operations* The key effects are: • W: the hero jumps upwards. • A: The hero moves to the left. • D: The hero moves to the right. • A: The hero performs a sword attack. • F: The hero performs a fireball attack. • ESC: opens the pause menu. The system must: • Manage the movement of enemies. • Manage the life count of both hero and enemy. • Manage the elimination of enemies from the map when they lose their life points. • Manage collisions between enemies and hero. • Manage collisions between the hero's attacks and enemies. • Manage collisions between hero and map, between enemies and map. • Managing the hero's invulnerability condition in the event of a collision with an enemy. • Take into account the number of enemies eliminated. • Time the completion of the level.
Output	The hero and enemies move around the map.

3.1.4 Score screen management

RFO4	Score screen management
Introduction	*Actors involved* Player *General description of the function* The scoring screen is loaded.
Input	*Generic description of data* End of level, when the hero collides with the teleporter. Press ENTER (keyboard).

Description (Process)	*Sequence of operations*
	When the player finishes the level, colliding with the teleporter at the end of the map, the screen with the level scores (number of eliminations, elapsed time) is loaded. When the ENTER button is pressed, the next level is loaded.
Output	The scoring screen is opened. The programme loads the next level.

3.1.5 GameOver Management

RFO5	GameOver Management
Introduction	*Actors involved* Player *General description of the function* The GameOver screen is loaded.
Input	*Generic description of data* When the hero's life points are reduced to 0 (or less). Press ENTER (keyboard). Press ESC key (keyboard).
Description (Process)	*Sequence of operations* When the hero's health points are reduced to 0 (or less), the GameOver screen is loaded, with the possibility of reloading the level from the beginning by pressing the ENTER key, or returning to the main menu with the pressing the ESC key.
Output	The GameOver screen is loaded. The programme reloads the level or the main menu.

3.1.6 Management of the pause menu

RFO6	Management of the pause menu
Introduction	*Actors involved* Player *General description of the function* The pause menu screen is loaded.
Input	*Generic description of data* Press ESC key (keyboard). Press the W (keyboard) key.
Description (Process)	*Sequence of operations* During the level, the player can pause the game by pressing the ESC key. Pressing the ESC key returns the game to the exact point at which it was paused. Pressing the W key returns to the menu main.
Output	The pause screen is loaded. The game is resumed or the main menu is loaded.

3.2 Non-functional requirements

3.2.1 Portability

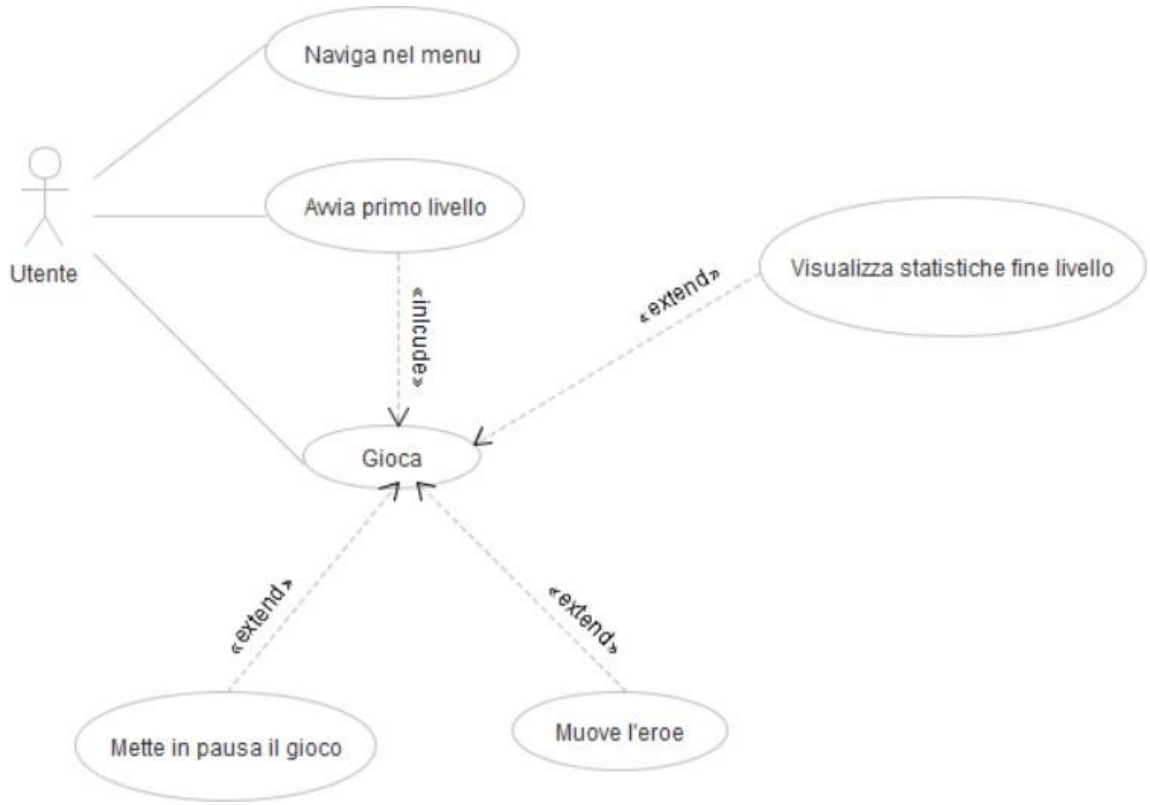
RNF01	Portability
Description	To ensure operation on different operating systems, it will be necessary to adopt JAVA as the programming language for software.

3.2.2 Maintainability

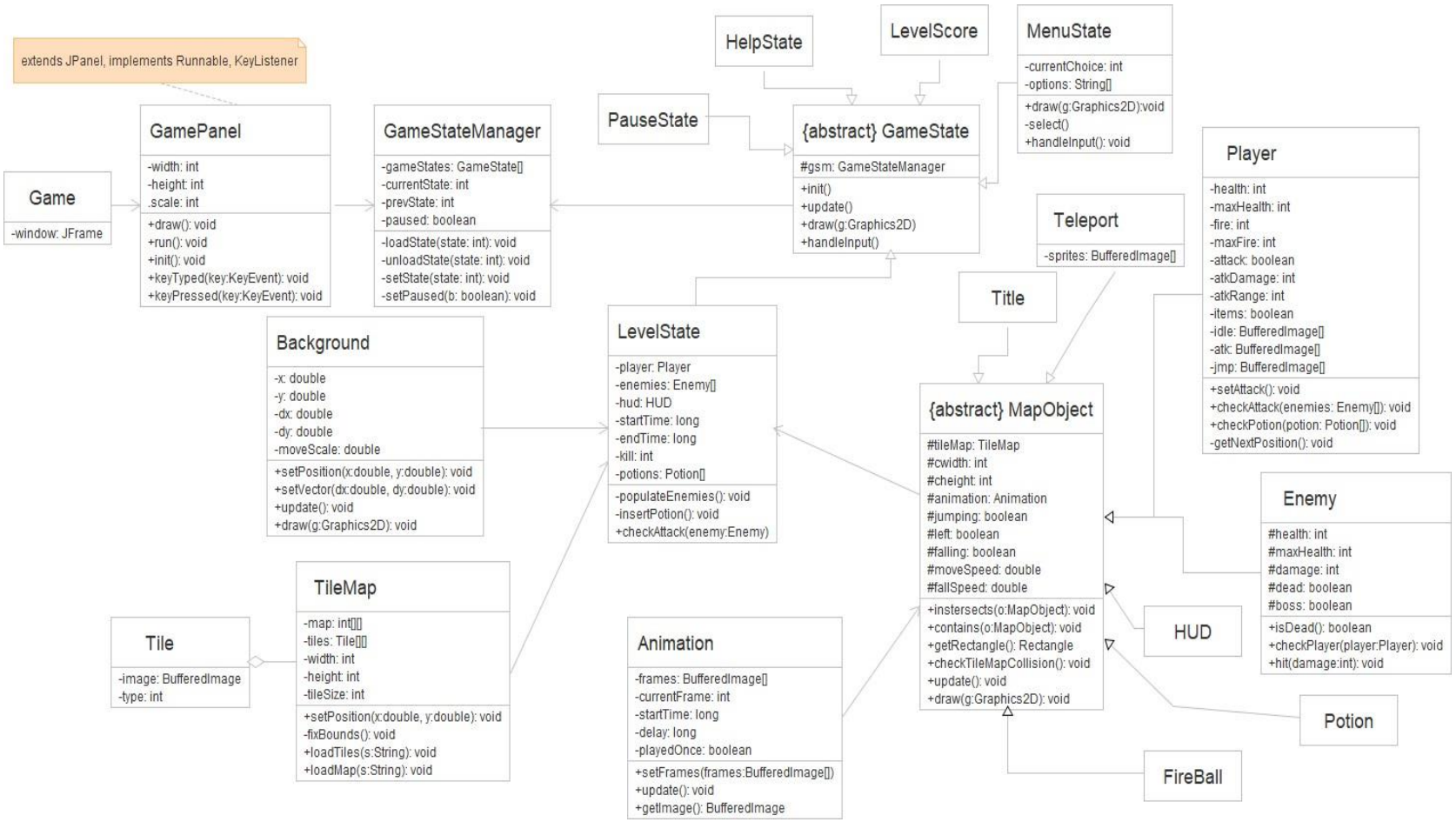
RNF02	Maintainability
Description	The software must be maintainable so that it can be added in future new features.

4. DIAGRAMS

4.1 Use Case Diagram

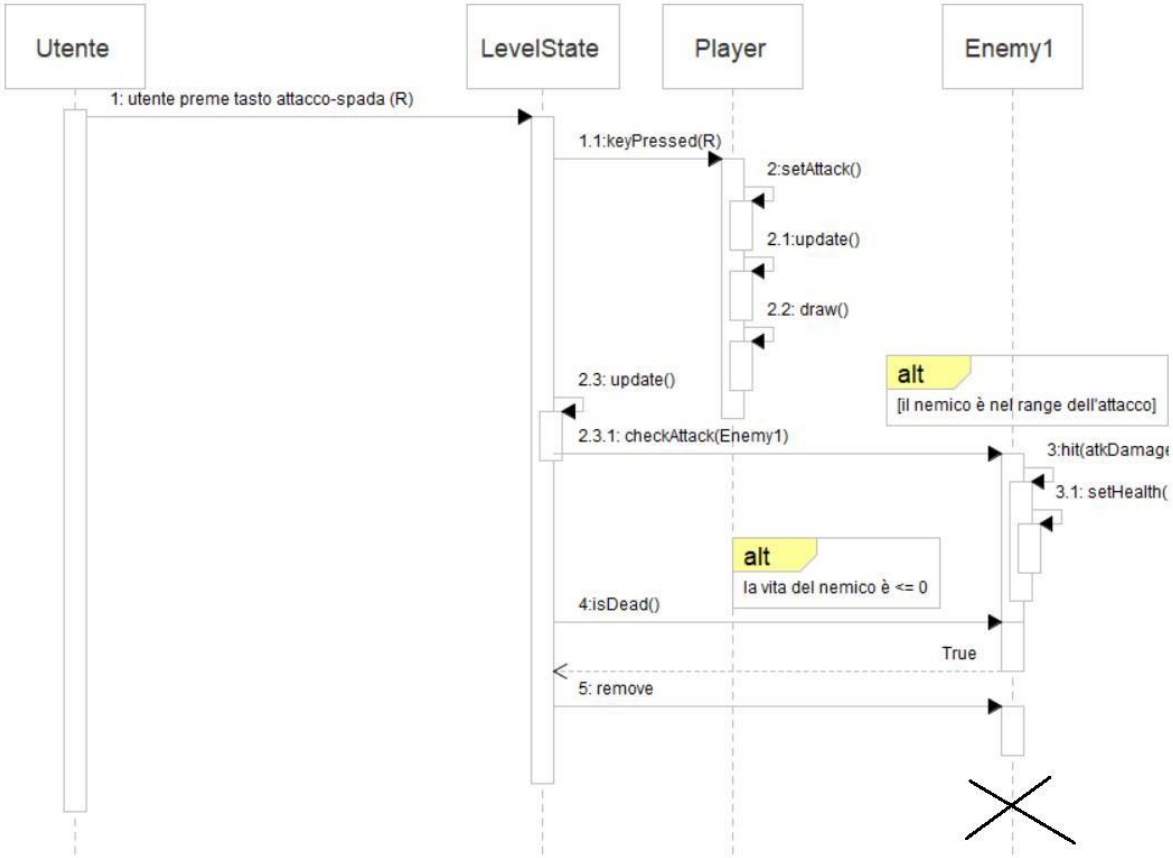


4.2 Class Diagram

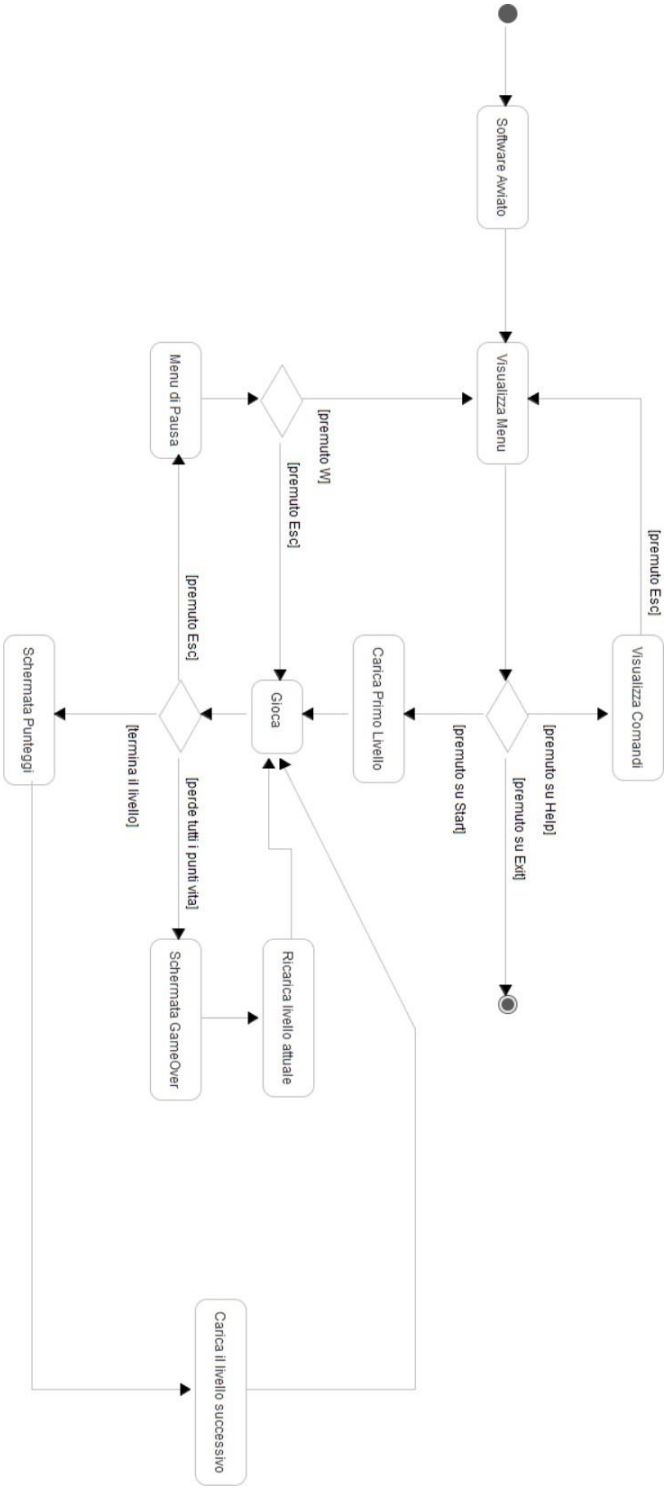

extends JPanel, implements Runnable, KeyListener
Game
-window: JFrame
GamePanel
-width: int
-height: int
.scale: int
+draw(): void
+run(): void
+init(): void
+keyTyped(key:KeyEvent): void
+keyPressed(key:KeyEvent): void
GameStateManager
-gameStates: GameState[]
-currentState: int
-prevState: int
-paused: boolean
-loadState(state: int): void
-unloadState(state: int): void
-setState(state: int): void
-setPaused(b: boolean): void
HelpState
LevelScore
MenuState
-currentChoice: int
-options: String[]
+draw(g:Graphics2D):void
-select()
+handleInput(): void
PauseState
{abstract} GameState
#gsm: GameStateManager
+init()
+update()
+draw(g:Graphics2D)
+handleInput()
Player
-health: int
-maxHealth: int
-fire: int
-maxFire: int
-attack: boolean
-atkDamage: int
-atkRange: int
-items: boolean
-idle: BufferedImage[]
-atk: BufferedImage[]
-jmp: BufferedImage[]
+setAttack(): void
+checkAttack(enemies: Enemy[]): void
+checkPotion(potion: Potion[]): void
-getNextPosition(): void
Teleport
-sprites: BufferedImage[]
Title
Background
-x: double
-y: double
-dx: double
-dy: double
-moveScale: double
+setPosition(x:double, y:double): void
+setVector(dx:double, dy:double): void
+update(): void
+draw(g:Graphics2D): void
LevelState
-player: Player
-enemies: Enemy[]
-hud: HUD
-startTime: long
-endTime: long
-kill: int
-potions: Potion[]
-populateEnemies(): void
-insertPotion(): void
+checkAttack(enemy:Enemy)
{abstract} MapObject
#tileMap: TileMap
#cwidth: int
#cheight: int
#animation: Animation
#jumping: boolean
#left: boolean
#falling: boolean
#moveSpeed: double
#fallSpeed: double
+instersects(o:MapObject): void
+contains(o:MapObject): void
+getRectangle(): Rectangle
+checkTileMapCollision(): void
+update(): void
+draw(g:Graphics2D): void
Enemy
#health: int
#maxHealth: int
#damage: int
#dead: boolean
#boss: boolean
+isDead(): boolean
+checkPlayer(player:Player): void
+hit(damage:int): void
HUD
Potion
FireBall
TileMap
-map: int[][]
-tiles: Tile[][]
-width: int
-height: int
-tileSize: int
+setPosition(x:double, y:double): void
-fixBounds(): void
+loadTiles(s:String): void
+loadMap(s:String): void
Tile
-image: BufferedImage
-type: int
Animation
-frames: BufferedImage[]
-currentFrame: int
-startTime: long
-delay: long
-playedOnce: boolean
+setFrames(frames:BufferedImage[])
+update(): void
+getImage(): BufferedImage

4.3 Sequence Diagram

The player attacks an enemy, eliminating it.



4.4 Activity Diagram



4.5 State Diagram

5. PATTERN DESIGN

Through the following design patterns, the software can be structured in such a way as to improve its modularity and maintainability.

5.1 Strategy Pattern

It was chosen to model the enemy classes with a Strategy Pattern, since enemies can have various fighting styles. For example, some enemy types will be passive and will not attack the hero, while others will be associated with an attack style.

It will then be possible to implement new types of attacks and new types of enemies.

5.1.1 Class Diagram

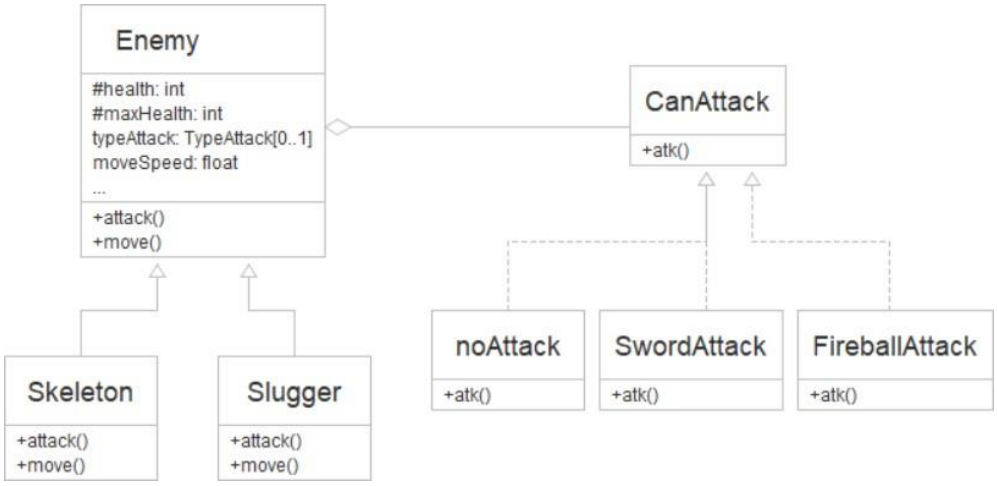


Figure 3: Sprites of the skeleton attack animation.

5.1.2 Code

```
public abstract class Enemy {
    TypeAttack typeAttack;
    //altre variabili

    public void Enemy() {
        //codice
    }

    public void attack() {
        public void typeAttack.attack();
    }
    //altri metodi
}


public class Skeleton extends Enemy {

    public Skeleton() {
        typeAttack = new SwordAttack();
    }
}

public interface TypeAttack {
    public void attack();
}


public class SwordAttack implements TypeAttack {

    public void atk() {
        //codice
    }
}
```

5.2 State Pattern

It was decided to model the classes relating to the various states assumed by the game (such as the various menus and levels) with a State Pattern. All states share and implement in their own way the four functions indicated in the class diagram in the GameState interface. Each class adds functions and variables as necessary.

The GameStateManager manages the transition from one state to another based on the user's actions.

5.2.1 Class Diagram