**Software Engineering**

**Projectile Parry**

**Design Document**

**Contents**

**1. Introduction**

**1.1 Purpose of the system**

**1.2 Design Goals**

**1.2.1 Criteria**

**1.2.1.1 End User Criteria**

**1.2.1.2 Maintenance Criteria**

**2. High-Level Software Architecture**

**2.1 Subsystem Decomposition**

**2.1.1 Game Layer**

**2.1.1.1 Projectile Parry Game Engine**

**2.1.1.3 Projectile Parry Components Package**

**2.1.2 User Interface Layer**

**2.1.3 System Interface Layer**

**2.2 Hardware / Software Mapping**

**2.3 Persistent Data Management**

**2.4 Access Control and Security**

**2.5 Boundary Conditions**

**3. Subsystem Services**

**3.1 Interface Layer**

**3.1.1 User Interface Management**

**3.1.1.1 Game Screen Management**

**3.1.1.2 Menu Screen Management**

**3.2 Game Application Layer**

**3.2.1 Game Management Interface**

**3.3 Storage Layer**

**3.3.1 In Game Data Management**

**4. Low-level Design**

**4.1 Object Design Trade-Offs**

**4.2 Final Object Design + Design Patterns**

**4.2.1 Facade Design Pattern**

**4.2.2 Composite Design Pattern**

**4.3 Packages**

**4.3.1 Developed Packages**

**4.3.1.1 Menu Package**

**4.3.1.2 File Manager Package**

**4.3.1.3 Game Object Package**

**4.3.1.4 Game App Package**

**4.3.2 External Library Packages**

**4.3.2.1 java.util**

**4.3.2.2 JavaFX**

**4.3.2.3 LWJGL**

**4.4 Class interfaces**

**4.4.1 Game App class**

**4.4.2 gameMap**

**4.4.3 Game object**

**4.4.3.1 block**

**4.4.3.2 enemy**

**4.4.3.3 projectile**

**4.4.3.4 Paddle**

**4.4.4 collision manager**

**4.4.5 file manager**

**4.4.6 menu**

**4.4.6.1 Pause Menu Class Methods**

**4.4.6.1 Settings Menu Class Methods**

**4.4.6.2 Settings Menu Class Methods**

**4.4.6.3 Credits Menu Class Attributes**

**4.4.6.4 Scores Menu Class Attributes**

# **1. Introduction**

**1.1 Purpose of the system**

The purpose of this system is to provide a fun and engaging game experience for a single player. The system will implement a breakout-style game with enemies that fire at the player and projectiles that can be parried. The game will have simple controls, progressively more challenging levels, and appealing but simple graphics and sound effects. The system will be designed to run on multiple platforms (thanks to Java) and provide an optimized gaming experience for each. The game will also have a data management system to store player data securely and enable customization of gameplay settings. Overall, the purpose of this system is to provide an enjoyable gaming experience for a single player that keeps them engaged and entertained.

**1.2 Design goals**

Design is a crucial stage in creating a system, as it enables us to determine the key design goals that require our attention. As highlighted in our analysis report, there are several non-functional requirements of the system that require further elaboration during the design phase. Therefore, these requirements are the primary focus of this document, and the following sections provide detailed descriptions of the important design goals that need to be addressed.

**1.2.1 Criteria**

**1.2.1.1 End User Criteria**

**Usability**

**Intuitive User Interface**

The game should have an intuitive user interface that is easy to navigate and understand. The interface should be consistent and predictable, with clear labels and instructions.

**Clear Feedback**

The game should provide clear feedback to the user, indicating when they have successfully blocked, parried, hit an enemy, or received damage. The feedback should be immediate, visible, and understandable.

**Customizable Accessibility**

The game should allow users to customize the visuals to their liking, making it easier for them to play and navigate the game. This is primarily useful to users with visual or sensory handicaps.

**Help and Documentation**

The game should include clear and concise help and documentation, with instructions on how to play the game, how to customize settings, and how to troubleshoot common issues.

**Fast Load Times**

The game should have fast load times, allowing users to quickly start playing the game without having to wait for a long time for the game to load.

**Smooth Performance**

The game should have smooth performance, with no lag or delays that could affect gameplay. The game should be optimized to run well on a range of devices and platforms.

**Efficient Resource Usage**

The game should use resources efficiently, including memory and CPU usage. The game should not consume excessive resources that could slow down other applications or affect the user's experience.

**High Frame Rates**

The game should have high frame rates, ensuring that the game appears smooth and responsive to the user. The game should aim for a minimum of 30 frames per second for optimal performance.

**Minimal Crashes and Bugs**

The game should be stable and reliable, with minimal crashes and bugs that could affect the user's experience. Any issues that arise should be quickly addressed and resolved through updates or patches.

**1.2.1.2 Maintenance Criteria**

Modular and Maintainable Code

The game's code should be modular and maintainable, allowing for easy updates and bug fixes. The code should be well-organized and commented, with clear and concise functions and classes.

**Version Control**

The game's code should be stored in a version control system, allowing for easy collaboration and tracking of changes. The version control system should be well-maintained and up-to-date, with regular backups and security measures in place. Typically, a free and simple system such as GitHub will be suitable.

**Scalability**

The game should be designed to be scalable, allowing for future updates and enhancements. The code should be designed to accommodate additional levels, challenges, and features as needed

**Documentation and Knowledge Transfer**

The game's code and development process should be well-documented, allowing for easy knowledge transfer between team members and future developers. The documentation should include clear instructions on how to build, test, and deploy the game.

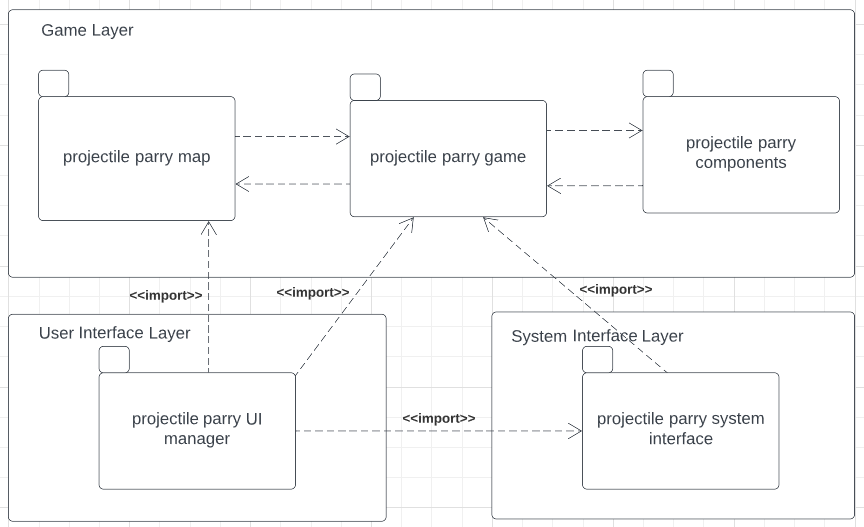
**Reusability**

The game’s code should have aspects such as the UI or the presence of a game loop that can be easily used in other projects with minimal conversion or effort. These aspects will be developed with functional independence in mind.

**Automated Testing**

The game's code should be tested using automated testing tools, to ensure that the code is functioning as expected and to catch any bugs or issues. The testing should be automated as much as possible, with a clear and concise testing plan.

# **2. High-Level Software Architecture**

**2.1 Subsystem Decomposition**

**2.1.1 Game Layer**

The game layer is where the main operations of the game are held. This layers is grouped into projectile parry components, projectile parry map, and projectile parry game.

**2.1.1.1 projectile parry game engine**

The game engine of the program is where the game will play out for the player. The engine communicates with both the components package to pull classes such as the enemies and block types. It also communicates with the map package to both pull level designs and to update those designs as the player defeats enemies and breaks blocks. It also imports UI elements to show to the player during gameplay and imports from the system interface so the game can interact with the system layer of the program.

**2.1.1.2 projectile parry components Package**

The components package is the package where most of the game’s classes will be stored. The player character and its related properties, the multiple types of enemy, and the types of blocks the players will face.

**2.1.1.3 projectile parry map package**

The map package is where the game's level designs will be stored. These designs will be called by the game engine during play. The map package communicates with the GUI so the GUI can show the map on screen for players.

**2.1.2 User Interface Layer**

UI related classes are bundled in the package projectile parry UI manager. These classes provide the visuals for user interactions from the pause menu and settings to the in game UI elements such as health and score.

**2.1.3 System Interface Layer**

System related classes are all in the projectile parry system interface package. These classes power the game’s underlying elements such as saving data, reading I/O from the player, managing computer resources, and running the program.

**2.2 Hardware / Software Mapping**

Projectile Parry will be implemented using Java as our programming library. We will be using LWJGL (Lightweight Java Game Library) for our game development as well. Due to these the game will need the Java Runtime Environment to be played and will need Java Development Kit 8 or newer in order to use the LWJGL for our title. This should make the game playable on Windows, Mac, and Linux devices. Software will be run with a single executable file which will call other needed files when run. These files will all be downloaded together by users.

The main hardware requirements of our game will be a computer, a monitor, and a keyboard. The keyboard will be needed to control the player movement, attacking, and menus. Menus should also be controllable with the mouse but as they will also be controlled with the keyboard the mouse will not be required for I/O of the game.

**2.3 Persistent Data Management**

Projectile Parry will not have much data to store for the users. The game objects like the graphics, enemies, maps and music will be stored in the user’s hard drive along with text files. Most of the text files such as map design and UI designs will be created and implemented by the developer team and will not be changed by the user but some such as game settings will be able to be changed by the users as they wish and some like high scores will be changes and updated overtime depending on how the game is played. Changing these will allow these values to be saved for an individual copy of a program. Files for images will be stored in the .png format and audio will be stored in .ogg file format.

**2.4 Access Control and Security**

Projectile Parry will be entirely stored locally with no online connections. Anyone can play the game once it is loaded from a computer. Due to the simple and small scope of the title there will be no access control and security. The high scores and settings will be stored locally and unique for each user. Meaning a high score on a computer will be the high score of everyone using that computer and not everyone playing the game on every computer.

**2.5 boundary conditions**

The game will be executed via a Jar file and the various classes that the game will need will be contained with it. This should allow owners of the game to transfer the game from one device to another if they so choose.

The game can be exited at any time with the quit feature at the expense of not having data saved. Alternatively, a player can “formally” exit the game by pausing the game, returning to the main menu, then hitting the “quit” option. This will retain some if not all player data obtained in the session.

In the instance that a file (whether it be a png or ogg) has any issues loading, it is recommended that the player validate the game files. The game may become significantly harder if not impossible to play if certain pngs do not load. It is recommended that players keep a backup of high scores as well.

Unfortunately, if the game were to crash during a session. There may be no way to restore data that was obtained in that session and it will be lost.

# 

# **3. Subsystem and Services**

**3.1. Interface Layer**

**3.1.1 User Interface Management**

The user interface management is meant to handle the display of the user interface (menus, game navigation and). There will be 2 subcomponents within the user interface management.

**3.1.1.1 Game Screen Management**

This component will be responsible for displaying the game screen, score, blocks, enemies, projectiles, boundaries, and the paddle.

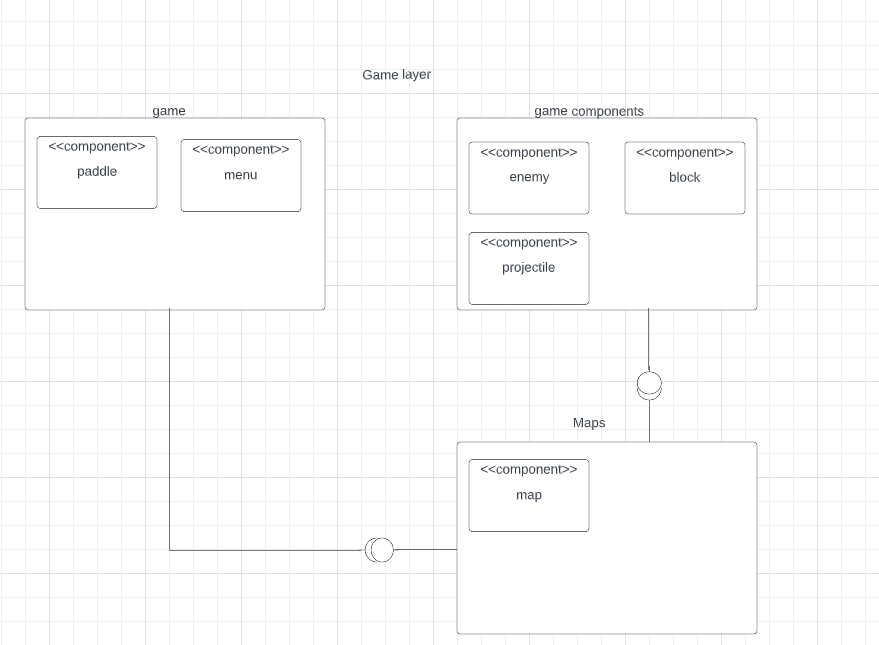
**3.1.1.2 Menu Screen Management**

This component will be responsible for displaying the different options our menu has, such as difficulty and sound. From this component the user will be able to adjust the game difficulty and sound levels to preferred levels.

**3.2 Game Application Layer**

**3.2.1 Game Management Interface**

This component handles the requests of the interface layer and carries out our game logic, described in phase 1 of our project design. After handling the input and applying our game logic, thus changing our game objects like blocks, enemies, and health accordingly. The information of these classes is then updated and stored in our Storage Layer. Our classes can update and send over new information until game logic deems the game over.



**Figure 1.1 Game Application Subsystem**

**3.3 Storage Layer**

**3.3.1 In Game Data Management**

This component is responsible for handling the data retrieved from the Game Application layer, updates the old data with new data, if any, and returns it back to the Game Application layer, updating the score, and game entities on the map.

# **4. Low-level design**

**4.1 Object design trade-offs**

**Memory v. Performance**: Our program will use a lot of objects which will end up using a lot of memory making the program less lightweight and run worse in exchange of the object oriented design.

**Complexity vs. User-Friendliness**:

One potential trade-off is between the complexity of the gameplay and the user-friendliness of the interface. If the gameplay is too complex, it may be difficult for our users to navigate the game. On the other hand, if the game is too simple, it may not be engaging enough for some players. We want to achieve a balance of an acceptably fun and enticing experience without overwhelming any potential players not well skilled or versed in the genre.

**Performance vs. Graphics**:

Another potential trade-off is between the performance of the game and the quality of the graphics. High-quality graphics can enhance the player's experience, but they’ll require more system resources and slow down the game's performance. They will also require significantly more time to create and optimize. On the other hand, lower-quality graphics may provide better performance but may not be as visually appealing.

**Customization vs. Data Management**:

A trade-off between customization and data management is possible. The more customization options provided to the player, the more data needs to be managed by the system. It is important to ensure that the data management system is designed to handle the increased load without any performance issues.

**Time vs. Features**:

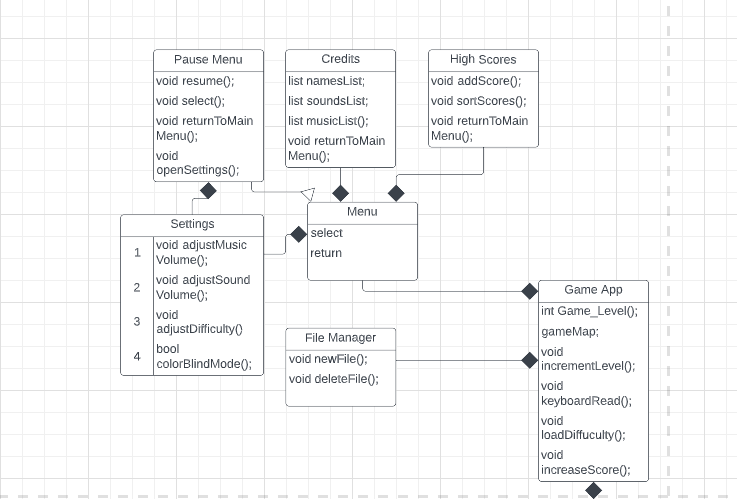
A trade-off between time and features may exist, as implementing more features may require additional development time. It is important to balance the development timeline with the features required to ensure that the game is completed on schedule and meets the desired specifications outlined in the document.

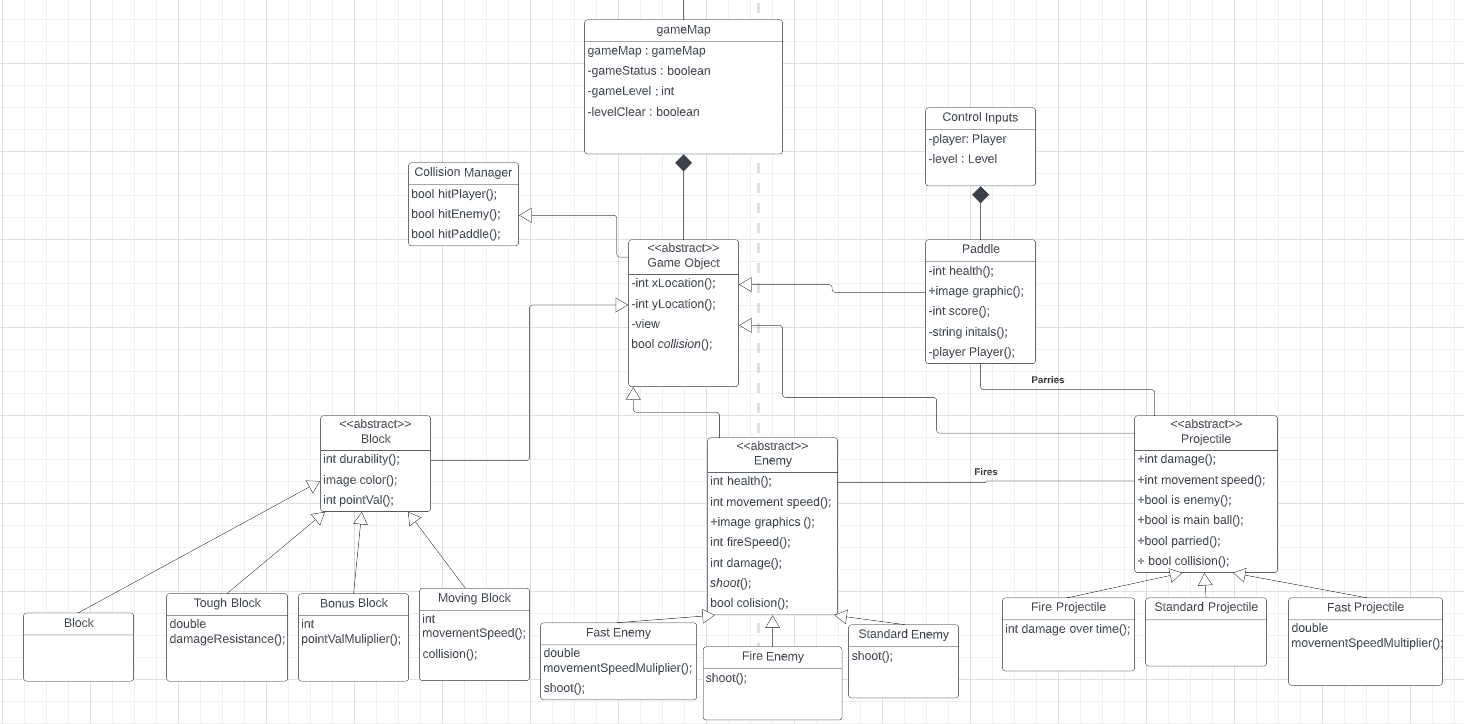
**Code Complexity vs. Code Maintainability**:

Code complexity and maintainability are a trade-off when designing any software system, including a game. Java allows for complex, feature-rich code, but this can make the code difficult to maintain. This is only embellished with the usage of any potential third-party libraries. To ensure that the game's code is maintainable and doesn’t become a pain to expand upon, readability and simplicity become more important.

**Third-Party Libraries vs. Code Flexibility**:

Java has a wealth of third-party libraries available, which can save development time and effort, one of which being LWJGL (Light-Weight Java Game Library) However, using third-party libraries can limit the flexibility of the game's code. To ensure that the game's code is flexible and adaptable over time, it may be necessary to limit the use of third-party libraries

**4.2 Final object design + design patterns.**

**Figure 2.1 Final Object Design Model**

**Figure 2.2 Final Class Design Model**

**4.2.1 facade design pattern**

One of the structural design patterns used in Projectile Parry is the facade design. The UI subsystem and system interface subsystem will interact through the gameApp class/layer. This reduces coupling but there may be an increase to memory usage due to increased steps needed to communicate among layers .

**4.2.2 composite design pattern**

There are plenty of game objects in projectile parry, and various subtypes of those objects, A composite design pattern is utilized to allow one abstract class to form many subclasses. This will allow more sub-types of objects without too many other objects being affected. The gameObject class is the central class that all game objects will be created from.

**4.3 Packages**

In our implementation we are planning on using two types of packages. There will be packages designed by the developers and packages that will be introduced by the addition of external libraries such as LWJGL3.

**4.3.1 Developed Packages**

**4.3.1.1 Menu Package**

The menu package has classes that control the games menus and their associated methods.

**4.3.1.2 File Manager Package**

File manager package provides file management and control for the program

**4.3.1.3 Game Object Package**

Game object package is a package made up of classes that present themselves in the game such as the enemies and blocks. The package also provides methods for these objects.

**4.3.1.4 Game App Package**

The package which is responsible for classes controlling game related functions like the levels, input, and difficulty.

**4.3.2 External Library Packages**

**4.3.2.1 java.util**

This package contains the ArrayList Object which will be used to quickly and efficiently manage several entities such as control schemes, bullets, enemies, and a grid. This package also contains Serializable which will be used to stream bytes from the program to the disk or vice versa for saving and managing data. The IO Subpackage will be used for accessing files on the disk such as level schemes, folders, and player data.

**4.3.2.2 JavaFX**

Rendering the Game UI: JavaFX can be used to create and render the game's UI, including buttons, menus, and graphics. JavaFX has built-in support for 2D and 3D graphics, which could be useful for creating the game's visuals. JavaFX can also handle user input events, such as mouse clicks and keyboard inputs, allowing the user to interact with the game. JavaFX has powerful animation capabilities, which could be used to create smooth and fluid animations for the game's objects, such as the ball, paddle, and enemies.. JavaFX includes support for playing audio, allowing the game to include sound effects and background music.

**4.3.2.3 LWJGL**

While the usage of this library over JavaFX is still in question, LWJGL is a capable framework which can provide direct access to OpenGL, a powerful graphics library that can be used to create high-performance 2D and 3D graphics for games. This could be as useful for creating the game's visuals, including the ball, paddle, and bricks. LWJGL is also designed to use computer resources efficiently, including memory and CPU usage, to maximize performance and minimize lag or delays that could affect gameplay. LWJGL additionally provides direct access to input devices such as keyboards and mice, allowing for precise and responsive input handling that could improve gameplay.

**4.4 Class interfaces**

**4.4.1 Game App class**

attributes:

- Private int game\_level: Displays the current level the player is on.

- gameMap: the map that will correspond to the game\_level.

methods:

- Void Increment level() : if the player is able to complete a level, the method will increment the current level by one and will load the next corresponding game\_map.

- Void Increase score() : method that will increase score depending on various actions that the player does.

- Void keyboardRead() : method that handles the various keyboard inputs which allow the player to move the paddle.

- Void loadDifficulty(int difficulty) : method that will change the difficulty at the player's request. 1 is easy, 2 is normal, and 3 is hard.

**4.4.2 gameMap**

Constructors

- gameMap(int blockCount, int enemyCount): the constructor for the gameMap objects

Attributes:

- Boolean gameStatus: returns true if the game is in progress, returns false if the game is over by any means.

- Int gameLevel: level corresponding to the map.

- Boolean level clear: if all enemies and blocks have been destroyed by the player, returns true and player is permitted access to the next level;

**4.4.3 Game object**

Attributes:

- Int xLocation: an object's location on the x axis of the field

- Int yLocation: an object's location on the y axis of the field

methods:

- View: returns an object's location //i could use some elaboration on what this does

- Boolean collision(object otherObject): detects whether or not an object has collided with another object.

**4.4.3.1 block**

Constructors:

- standardBlock()

- toughBlock(double damageResistance)

- movingBlock(int movingSpeed)

- bonusBlock(int bonus points)

Attributes:

- Int durability: how much damage a block can take

- Color imageColor: what color a block will be;

- Int pointVal: how much points a block gives upon its destructions

- double damageResistance: reduces the damage a block takes by a set amount

- int movingSpeed: how fast a block can move

- int bonus points: how many extra points a block gives upon its destruction

**4.4.3.2 enemy**

Constructors:

- standardEnemy();

- fireEnemy();

- fastEnemy();

Attributes

- Int Health: how much health an enemy has.

- Int movement speed: how fast an enemy can move.

- Int fire speed: how frequently an enemy fires

- Int damage: how much health the player loses when hit by a projectile from this enemy

- Double movementSpeedMultipler: multiplies an enemy’s movement speed by a set amount

Methods

- shoot(): this abstract method will determine what projectile each enemy shoots. If the enemy is a fire enemy, it will shoot a fire projectile, if the enemy is a fast enemy, it will shoot a fast projectile, if the enemy is a standard enemy, it will shoot a standard projectile.

**4.4.3.3 projectile**

Constructors

- standardProjectile();

- fireProjectile();

- fastProjectile();

Attributes

- Int damage: how much damage a projectile will do upon impact with an object;

- Int movement speed: how fast a projectile moves;

- Boolean isEnemy: returns true if created by an enemy object;

- Boolean Parried: returns true if the player parries a projectile shot by an enemy

- Boolean isMainBall: returns true if the projectile is the one the player must keep from falling down.

- Int damageOverTime: how much damage over time a fire projectile will inflict upon a game object

- Double MovementSpeedMultipler: multiplies an enemy’s movement speed by a set amount

**4.4.3.4 Paddle**

Constructors

- paddle()

Attributes:

- Int health: how much health the paddle has.

- Image Graphic: png that will be displayed as the paddle.

- Int score: value that will gradually get increased based on player actions

- String initials: string of letters that will represent the player

**4.4.4 collision manager**

Methods

- Boolean hitPlayer(): returns true if a player has collided with an object

- Boolean hitEnemy(): returns true if an enemy has collided with an object

- Hit paddle(): returns true if a projectile has hit the paddle

**4.4.5 file manager**

Methods

- Void newFile(): creates a new file with the saved data + highscores

- Void deleteFile(): deletes the file with the saved data + highscores

**4.4.6 menu**

Attributes - options → list attribute that contains the list of options the user could select

Methods

- select() → method that runs when an option is selected, leading into another screen

- return() → method that returns the user back to the previous screen

**4.4.6.1 Pause Menu Class Methods**

- resume() → method that destroys the pause menu object and resumes the game

- select() → method that runs when an option is selected, leading into another screen

- return() → method that returns the user back to the main menu screen

**4.4.6.2 Settings Menu Class Methods**

- adjustMusicVolume() → method that changes the volume of the music in the game

- adjustSoundVolume() → method that changes the volume of the sound effects in the game

- adjustDifficulty() → method that adjusts the difficulty of the game

- colorblindMode() → method that turns on/off colorblind mode

**4.4.6.3 Credits Menu Class Attributes**

- nameList → list attribute that represents the names of the developers of the game Methods

- return() → method that returns the user back to the main menu screen

**4.4.6.4 Scores Menu Class Attributes**

- scoresList → list attribute that represents the high scores that have been added Methods

- addScore() → method that allows the user to add their score to the list of scores

- sortScores() → method that sorts the list of scores in various ways

- return() → method that returns the user back to the main menu screen