System Design Document
SDD

Project Cosmic Shooter

Team 30
Anton Andersson, Jonatan Arvidsson, Adrian Danielsson,
Linn Lindgren och Simon Bergström

Contents

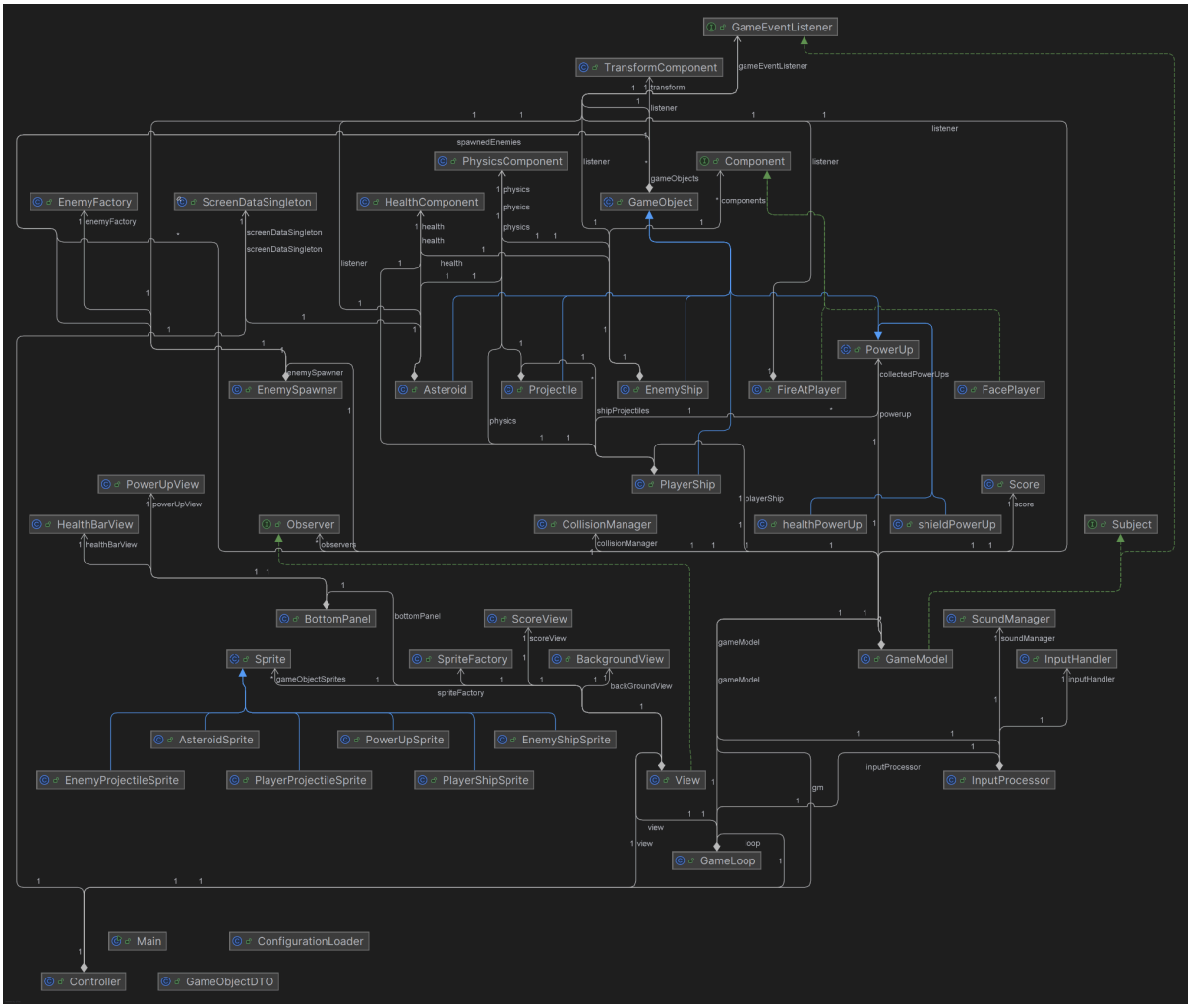
1. Introduction

1.1 Purpose and Scope

The developed system for the retro inspired game Cosmic Shooter, is designed to be as modular and object oriented as possible. A system designed by these mentioned principles, facilitates an easier implementation, modification or removal of new modules in future extensions.

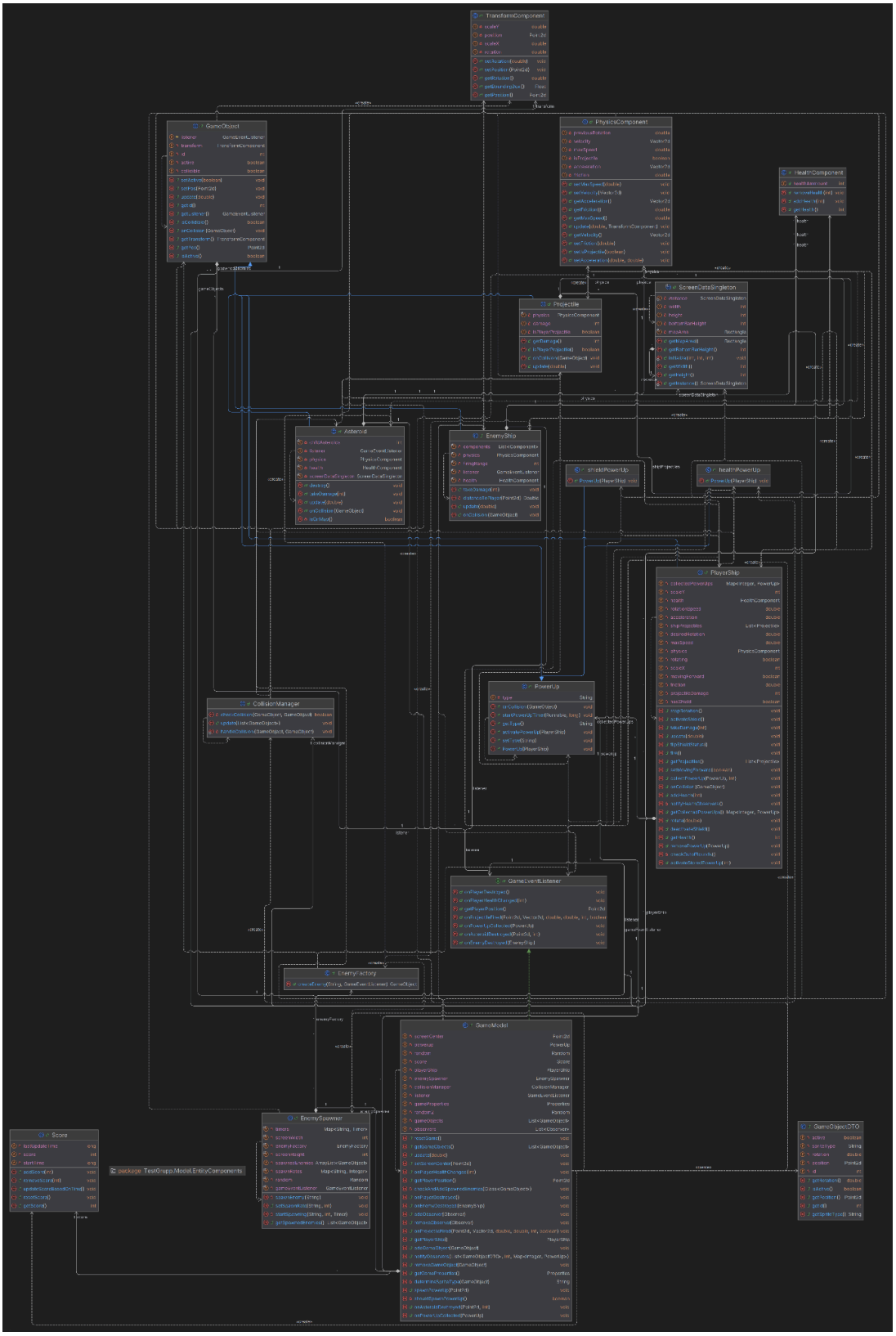
An implementation of the architectural pattern, model-view-controller (MVC), was also appropriate for the project system, because the system is best designed with clear and separate modules for the data and user presentation. This improves the modularity of the application.

2. Overview and UML-diagram



3. Detailed Design

3.1 Model



The model consists of several classes which are responsible for managing the logic of the application.

GameModel

The GameModel class manages the overall logic and state of the game. It initializes the game properties and objects. It also handles updating the game and notifying observers as well as managing spawning of enemies and power-ups.

Observers

The model contains the GameEventListener which acts as an observer for various game events such as when a projectile is fired or an enemy is destroyed. There is also an observer-pattern between the model and the view, where the view is a listener that gets notified when the game model updates and calls its corresponding method in the view.

GameObject

The GameObject class is an abstract class representing the fundamental building blocks for all entities in the game. It encapsulates common attributes by all game objects, such as position, rotation, scale, and activity state. This class can be expanded if more game objects would be implemented.

The subclasses that inherit from the GameObject are Asteroid, EnemyShip, Projectile, PlayerShip and PowerUp.

Components

The PhysicsComponent class in the model is responsible for handling the physical behavior of a game object. It manages movement, such as velocity, acceleration, friction and enforces speed limits. Concisely, this class ensures that the movement of gameobject apply the games physics.

Another component is the TransformComponent. This class encapsulates the position, rotation, and scale of the game objects, but the Components main responsibility is managing geometric transformations. This to define how game objects are oriented on the board.

There is also a HealthComponent that manages the health of a game object.

CollisionManager

This class's main focus is to detect and manage collisions between game objects. It detects overlapping objects based on their bounding boxes and invokes appropriate collision handling methods.

EnemySpawner and EnemyFactory

The EnemySpawner class manages spawning of enemy ships and asteroids. It defines spawning rate and also spawn enemies at random position on the edges of the screen. It also provides a list of spawned enemies that the GameModel utilizes. The EnemySpawner can be considered to be a utility class because it encapsulates the logic for spawning enemies and it also promotes reusability. The EnemySpawner does maintain some state, e.g. spawn rates and a list of spawned enemies, but its primary purpose is to provide utility methods for spawning enemies.

The Factory Method design pattern is being used in the EnemyFactory which is responsible for creating instances of enemy objects. The class consists of a method to create enemies based on a given type. Right now it has cases for Asteroid and EnemyShip. Using a Factory increases reusability in the code. The EnemyFactory is being used in the EnemySpawner class to create instances of different enemies.

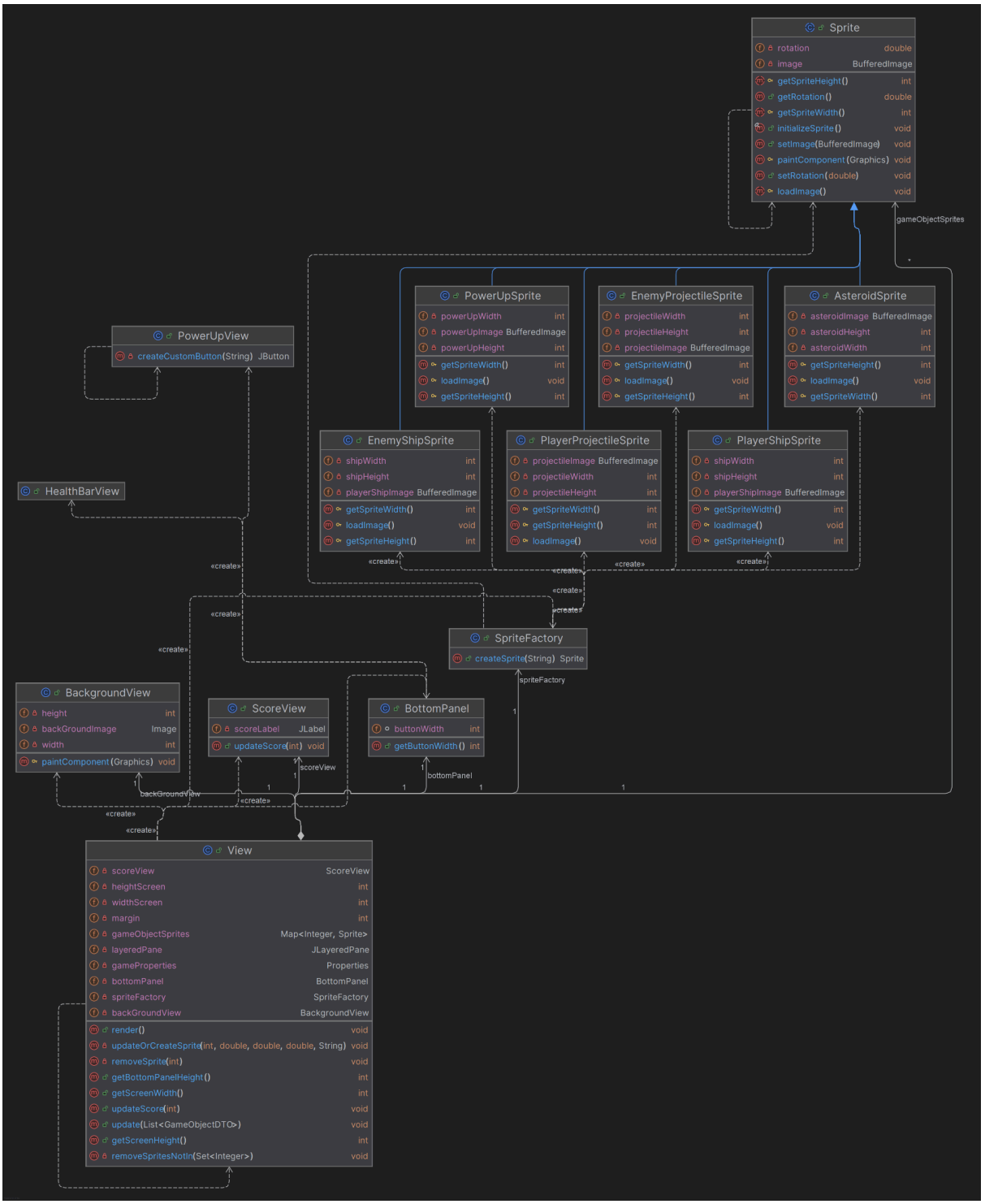
PowerUp

The PowerUp class represents a generic powerup and is an abstract class which is inherited by other power up classes, i.e. ShieldPowerUp and HealthPowerUp. This class extends the GameObject class and serves as an abstract base class for all power-ups entities in the game. It encapsulates properties for all power-ups, such as activation, timer functionality, and collision handling.

ScreenDataSingleton

This class exists to remove the problem that occurs when working with varying screen resolutions. When designed as a singleton it also makes accessing screen data more secure, which means it cannot be modified to represent wrong values. It is also redundant to have more than one instance to represent this feature/class.

3.2 View



The view module uses java swing to display the different game components which are divided into separate classes and updated/rendered based on the state of our model with the use of observers. It also contains a few panels, a background and sprites which aren't dependent on the model.

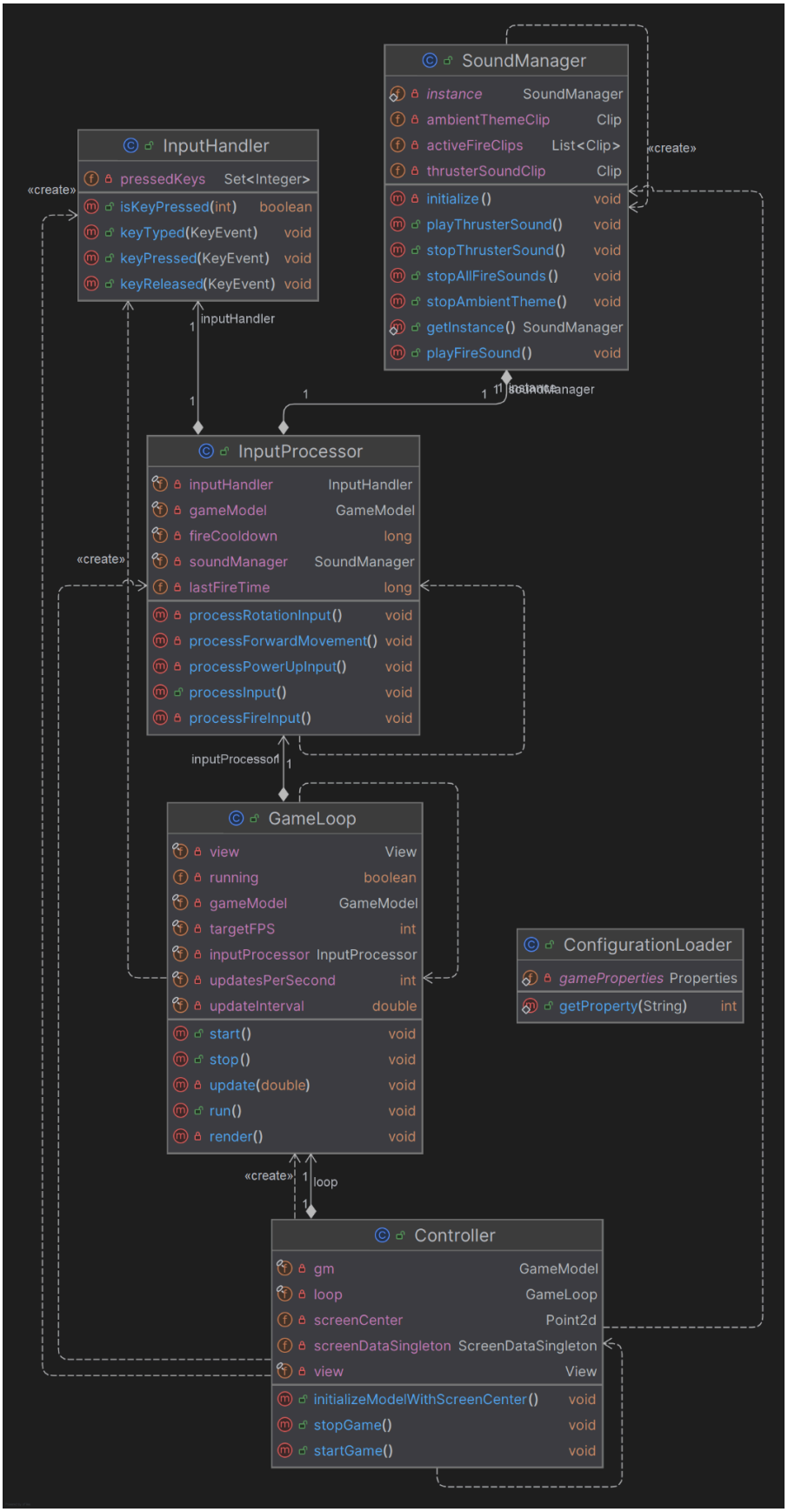
The view consists of a main class, called View, with additional logic divided into separate classes. The View uses a JLayeredPane to paint up different components, i.e. BackgroundView, BottomPanel, ScoreView, as well as different Sprites. The BottomPanel consists of the PowerUpView and the HealthBarView.

Sprites

In the Sprite class and its subclasses Template Method pattern is used to reduce code duplication. Each subclass then overrides the corresponding methods, e.g. loadImage so that each sprite gets the correct image loaded.

A Factory pattern (SpriteFactory) is used to create different sprites. SpriteFactory uses ConfigurationLoader to fetch properties for all the sprites, e.g. height and width for the PlayerShip.

3.3 Controller



The controller includes player input handling, sound handling, the game loop instance as well as the communication of updates to the Model, to the View. The game loop calls on the model to update as well as for the view to re-render sprites based on changes.

The controller consists of: A main class called Controller which instantiates the different parts of the controller, InputHandler a class which wraps Java's KeyListener, listening for key events and keeping track of which keys are pressed in a Set of integers. This class is aggregated by a class called InputProcessor which utilizes it to check if keys relevant to the game are pressed, and if they are firing the appropriate calls to the soundmanager and Model. The InputProcessor also implements a cooldown to the player ship firing by checking that the elapsed time since the player last fired is greater than the cooldown time for the gun.

The SoundManager class implements the singleton pattern, having one instance that is accessible during runtime. It manages all sounds in the game such as ambience sounds as well as sound effects for player thrusters and firing guns.

For running the game the GameLoop class operates in a separate thread and drives the update cycle by calling methods for the game logic in the Model to be updated as well as for the view to be rerendered. The loop checks how much time has elapsed since the last update and fires these calls when enough time, (roughly 8.33ms) has passed, in other words the game logic is updated 120 times per second. Rendering is currently set at a default update-rate of 60 frames per second.

The controller also have the ConfigurationLoader class which is used to help other classes fetch properties from the config.properties file.