**Bilkent University**

Department of Computer Engineering

**CS 319 Project**

*JCrawl: 2D Top-down Adventure Game*

Design Report

Group Members

* Arda Yücel
* Cheol Woo Park
* Fatih Taş
* Mustafa Fidan

Contents

[1. Introduction 2](#_Toc446854856)

[1.1 Overview 2](#_Toc446854857)

[1.2 Design Goals 3](#_Toc446854858)

[2. Software Architecture 4](#_Toc446854859)

[2.1 Subsystem Decomposition 4](#_Toc446854860)

[2.2 Hardware/Software Mapping 6](#_Toc446854861)

[2.3 Persistent Data Management 6](#_Toc446854862)

[2.4 Access Control/Security 6](#_Toc446854863)

[2.5 Boundary Conditions 7](#_Toc446854864)

[3. Subsystem Services 7](#_Toc446854865)

# 1. Introduction

## 1.1 Overview

JCrawl’s system primary goal is of course, entertainment. In order to achieve this, the system must have well-designed, streamlined gameplay, user friendly interface, along with little to no bugs. One of the key feature of JCrawl is that the levels will be customizable by the user. In order to achieve this, the game will be packaged with simple, easy to learn documentation to help users learn how to do so.

## 1.2 Design Goals

* **Performance**: The nature of the game that is going to be built demands high performance and low latency rates between each keystrokes. Optimizing data structures and misc. code will be the key to decrease the time taken for each update to take place.
* **Robustness**: There is nothing more frustrating than a game-breaking bug from end user’s perspective. The system should be prepared to handle myriad of unexpected/unwanted inputs from the user without causing the game to crash.
* **Extendibility**: Since the system will be designed with an object oriented language, five main principles will be kept in mind while designing the system: S. O. L. I. D
  + **1**. **S**ingle Responsibility Principle – All classes should have single responsibility (A class should not have multiple responsibility)
    - We put all hands on rendering into the platform class, hence achieving SRP.
  + **2**. **O**pen Closed Principle – All classes should be open for extension, but avoid unnecessary edit/revision (extend existing methods instead of writing new ones or editing existing methods)
    - Abstraction at the entities
  + **3**. **L**iskov Substitution Principle – Child Class and Parent Classes should be interchangeable
    - Entities
  + **4**. **I**nterface Segregation Principle – An interface should not have redundant methods that are never used.
    - The only redundant method we see from using interfaces is the keyTyped() which originates from the MouseListener, but since this is built-in interface, it cannot be avoided.
  + **5**. **D**ependency Inversion Principle – Parent Class should not be dependent of Child Class
    - All parent classes do not depend on the child class to function.
* **Usability:** The system will be user-friendly, easy to use. For example, due to the relatively high learning curve of learning custom level design, default level layout will be provided when the system is delivered. However, documentation will be provided to aid in user’s pursuit to learn it as easily as possible.

**Trade Offs**

* **Size vs Performance** – Since our system will be more focused on using the best algorithms to give us the lowest time complexity (big O), memory space usage optimizations will be at lower priority. However, there will be efforts to avoid pointless memory waste.
* **Reusability vs Performance** – In order to squeeze out the best possible performance, sacrifices must be made from reusability portion of the design. Also, if the issue of reusability can be outside of the scope of our design, it will allow for certain methods that highly specializes our system to be used so that performance can be at its best level.
* **Functionality vs Robustness** – Having a lot of gameplay content is appealing for the user, however, more toys usually means more things can break. There will be several restrictions on expanding new game content for the sake of system security and robustness.

# 2. Software Architecture

## 2.1 Subsystem Decomposition

We will use three layer architecture for our game, JCrawl. Our layers will be composed of Render Layer, Game Mechanics Layer, and Gameplay Object Layer. Render Layer will be strictly restricted to rendering the actual game onto the screen and nothing else, this layer will be composed of Platform and Menu class. Game Mechanics Layer is where all the data interaction occurs mainly in the GameManager class. Collision events and their consequences will be calculated by the GameManager with the help of CollisionManager and EntityManager, while the properties of the individual entities will be managed internally from the EntityManager. LevelManager will mostly handle initial level loading and occasional level changes. Gameplay Object Layer will mostly act as data containment layer, made up with classes that holds a lot of different types of data such as a single entity (its location, health, etc.)

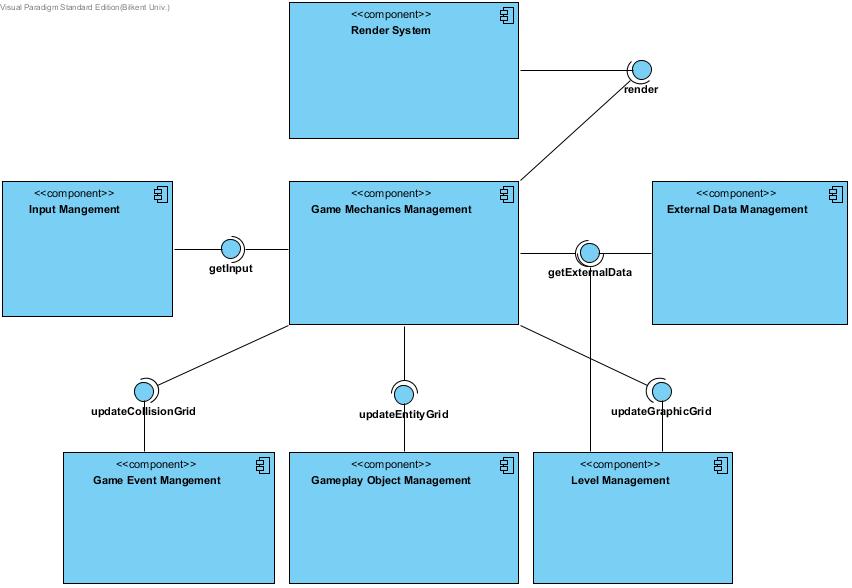


Figure 1 – High Level Representation of Subsystem Decomposition

Render Layer is represented as the Render System package in figure 2, which contains two classes: Platform and Menu. Platform is the primary user interface class which renders all the visual data that the user needs to play the game, and the menu class is there to help the Platform class with its specialized methods (for rendering menus). This package connects to the Game Mechanics Management Package.

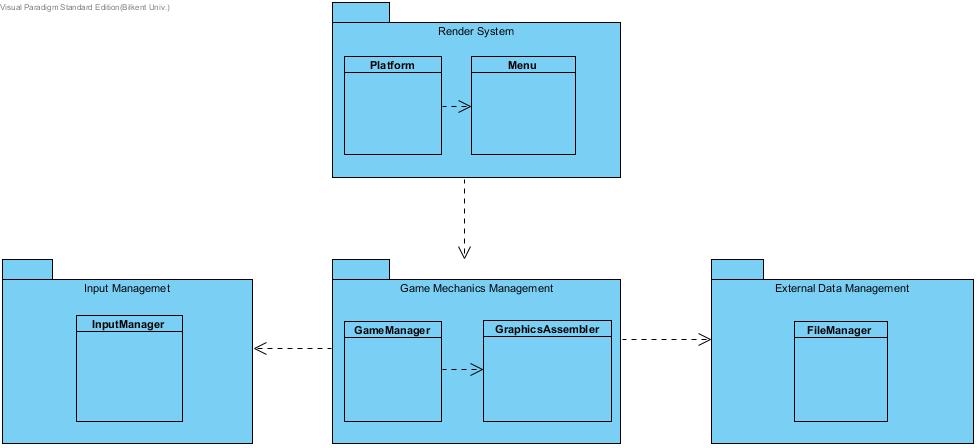


Figure 2 – Interaction between Render Layer (Layer 1) and Game Mechanics Layer (Layer 2)

Game Mechanics Layer consists of three partitions: Game Mechanics Management, External Data Management, and Input Management. Game Mechanics Management acts as primary data bus for all managers, it will translate/manipulate the data it receives from other managers and translate it so that other managers (if they need it) can use it without having to use special methods to do the translation within themselves. Input Management handles all user input and provides data to the Game Mechanics Management can work with. External Data Management receives external data (.txt files or xml files for level for example) and translates it into a format that the Game Mechanics Management can read.

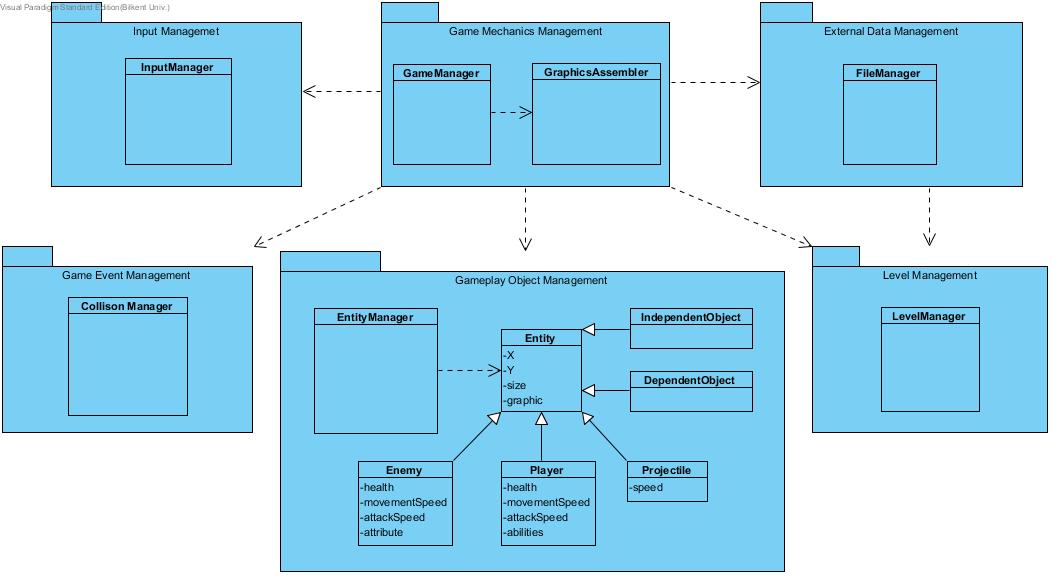


Figure 3 – Interaction between Game Mechanics Layer (Layer 2) and Gameplay Object Layer (Layer 3)

Gameplay Object Layer consists of three partitions like Game Mechanics Layer as show in figure 3: Gameplay Object Management, Level Management, and Game Event Management. Gameplay Object Management holds data for all types of entities and stores currently active entities in a grid. Game Event Management handles collisions and the respective locations where collision can occur and have occurred. Level Management handles placement of background level and their structure by reading from External Data Management.

## 2.2 Hardware/Software Mapping

The game will require JRE, Java Runtime Environment to work as the software will be written in Java. For hardware, it will require keyboard and mouse to use all of the software’s features.

As for the system requirements, the game won’t require such a powerful specifications as we will design our game loop to accommodate both slow hardware and fast hardware. (Fixed Update Time, Varied Frame Update with Interpolation) But, the computer should be able to run Java Runtime Environment and having a GPU will be a plus. The system only handles 7 keys and the mouse real-time.

## 2.3 Persistent Data Management

Both the game data and the level data will be stored in user’s hard drive. All data will be loaded on the user’s memory to be accessed by the system in real time. We will not use any special file format in order to make modding existing graphics easier both for the user and developer in the future. Level files will also be editable by any text editor (notepad, notepad++ or any text editor that the user prefers) and the documentation will be provided with the package.

## 2.4 Access Control/Security

As our game is a single player game without any kind of network related functions, there is literally no need to implement user authentication and all data will be stored in the memory to be accessed real time anyways. For the general security, the file system access will be given only to the External Data Management subsystem. All of our subsystem follows the principles of SOLID, which means catastrophic failure is less likely to occur in case one system fails.

## 2.5 Boundary Conditions

The program will use default level designs in case of: 1. No custom level files detected, 2. there is a syntax/semantics error in one of the custom level files. The game will return to main menu if player exits the game through pause menu or dies. If the user finishes the game, the game will exit to the main menu after displaying respective “congratulation” screen. If the user opens the same program twice, the 2nd instance of the program will not execute while displaying “there is already another instance of this program”.

# 3. Subsystem Services

Render System

Render system is responsible for rendering game data, displaying user interface and handling user actions on menus.

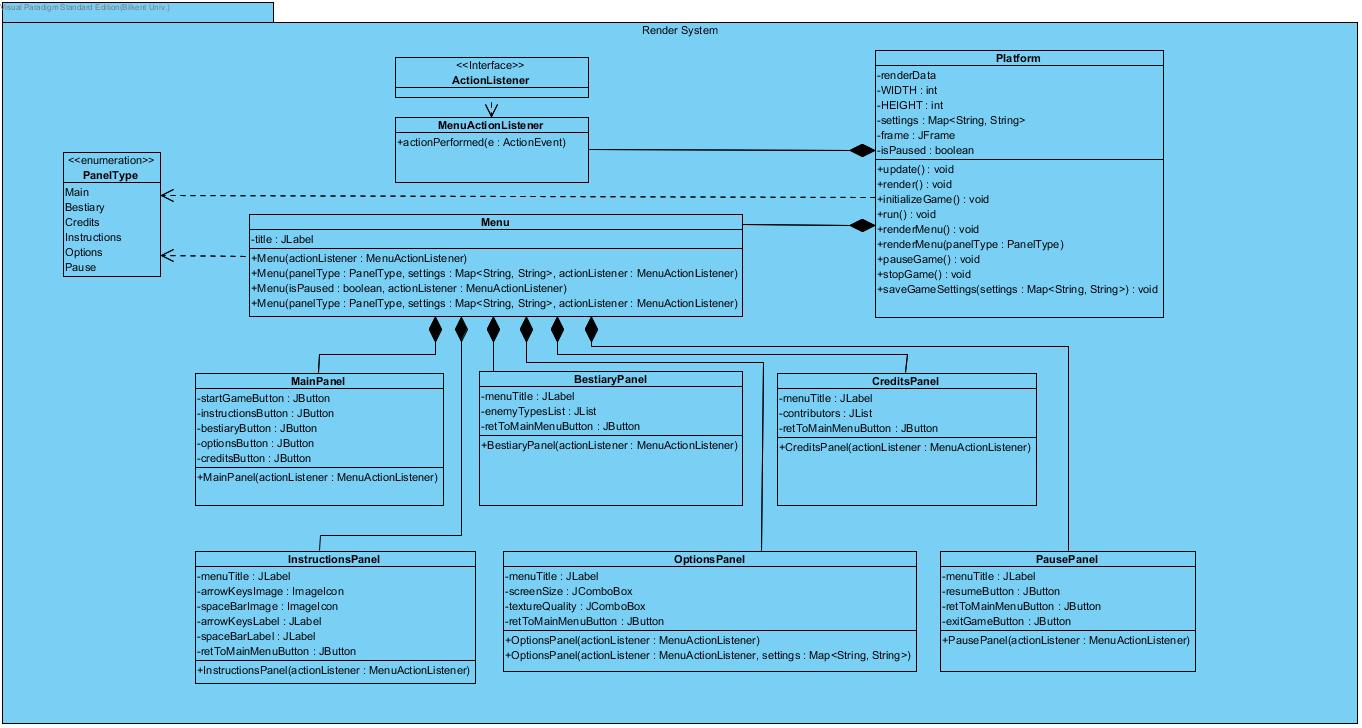


Figure 4 - Render System

Platform Class

Platform class is the first class instantiated when game first runs. Then it instantiates Menu Class with Main PanelType to display Main Menu to user. It also handles the user’s interaction with menu items like buttons, combo boxes. When user presses start game button, it instantiates GameManager class and starts game loop which calls update and render methods repeatedly.

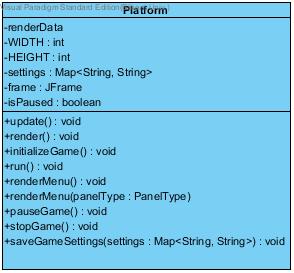


Figure 5 - Platform Class

MenuActionListener Class

Platform class instantiates MenuActionListener class and passes its reference to Menu Class then Menu Class passes that reference to all panel classes for handling user interactions with menu items. It implements ActionListener interface which forces to implement actionPerformed() method. This class decides what action needs to be taken when user interacts with a specific menu item.

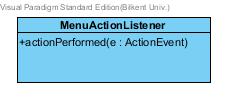


Figure 6 - MenuActionListener class

Menu Class

Menu class extends JPanel class from swing library. It is initialized by Platform class. It includes a label which contains game title. It uses the parameter that is passed from platform class to instantiate one of the panel classes accordingly.

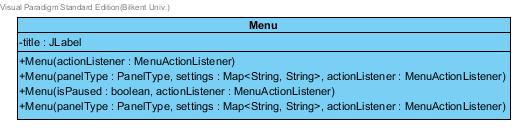


Figure 7 - Menu class

If it is passes Main Panel type it instantiates MainPanel.

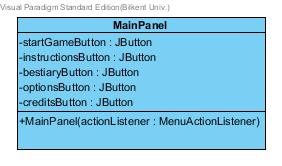


Figure 8 - MainPanel class

If it passes Bestiary Panel type it instantiates BestiaryPanel

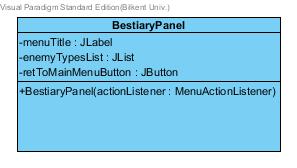


Figure 9 - BestiaryPanel class

If it passes Credits Panel type it instantiates CreditsPanel

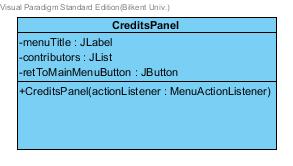


Figure 10 - CreditsPanel class

If it passes Instructions Panel type it instantiates InstructionsPanel.

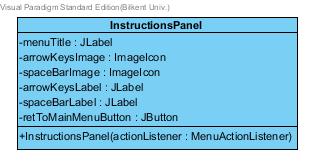


Figure 11 - InstructionsPanel class

If it passes Options Panel type with settings variable it instantiates OptionsPanel with the given settings data.

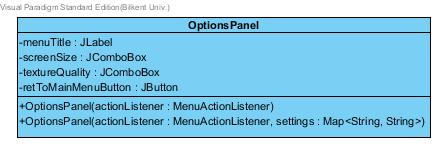


Figure 12 - OptionsPanel class

If it passes PausePanel type and isPaused variable is true, it instantiates PausePanel.

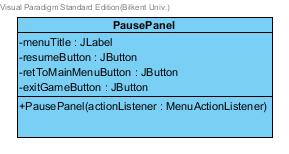


Figure 13 - PausePanel Class

Game Mechanics Management

Game Mechanics Management Subsystem is in charge of controlling the data flow between every other system which makes up the game. It basically acts like a data bus. It passes required data to other managers and it also assembles RenderData which is requested by the platform every render tick.

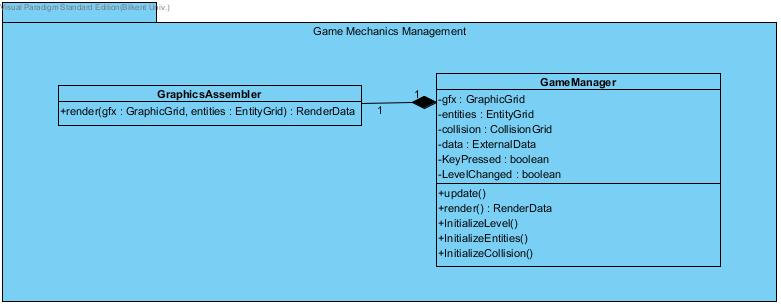


Figure 14 - Game Mechanics Management

GameManager Class

The GameManager class is the primary class of the Game Mechanics Management system, it is in charge of collecting/distributing data it receives from other managers through the update() method. What the update() method does is it calls the update() methods of other managers and update its data while passing the data other managers need in the process. Initialization methods are only called when the object is created (when the game starts basically). Render() method is called by the Platform class and it will call on GraphicsAssembler class’s render() method in the end which will return the data it needs to pass it to the Platofrm.

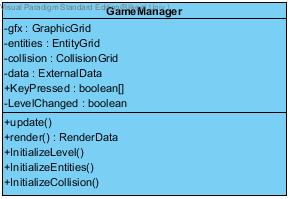


Figure 15 - GameManager Class

GraphicsAssembler Class

The GraphicsAssembler class is basically an extension of GameManager class’s render() method.



Figure 16 - GraphicsAssembler Class

Input Management

Input Management Subsystem handles all user inputs, and parses them into data that the Game Mechanics Management can actually utilize. The system makes use of KeyListener interface (built-in with java)

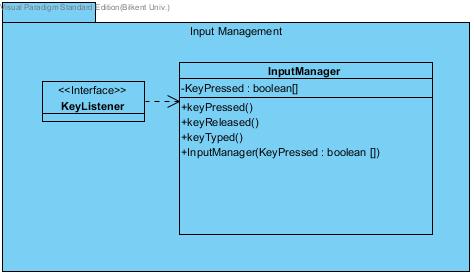


Figure 17 – Input Management

InputManager Class

InputManager Class implements an interface called KeyListener, and the interface requires us to implement three methods: keyPressed(), keyReleased(), keyTyped(). For our case, only the keyPressed() and keyReleased() will have meaningful blocks written in their body (keyTyped() is kinda redundant in this case but the interface requires to have it). The constructor is specified in this case as it does not use the default constructor and it requires a parameter. Whenever a key is pressed or released, the Boolean array KeyPressed will be updated and passed to the GameManager Class.

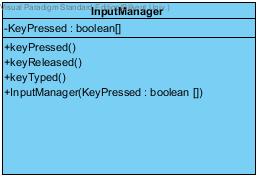


Figure 18 - InputManager Class

External Data Management

External Data Management Subsystem reads external datas like .txt files and then parses them into data format where the managers can read and utilize them.

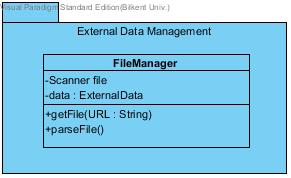


Figure 19 – External Data Management

FileManager Class

FileManager class uses an instance of the Scanner class to read the .txt file given, and then parses it into variety strings which will be stored in ‘data’ and later retrieved by the GameManager class using getter method.

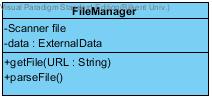


Figure 20 – FileManager Class

Game Event Management

Game Event Management Subsystem takes in required data from the Game Mechanics Management and translates that into collision data and returns it to the Game Mechanics Management Subsystem.

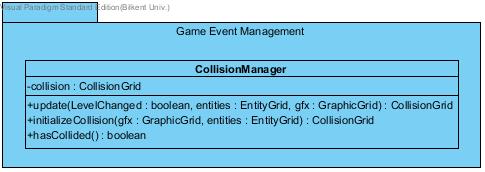


Figure 21 – Game Event Management

CollisionManager Class

CollisionManager Class takes in multiple data from the Game Mechanics Management, and converts into a collision data object and passes it back to the Game Mechanics Management component.

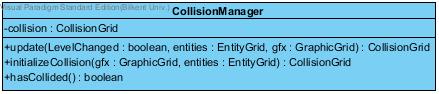


Figure 22 – CollisionManager Class

Gameplay Object Management

Gameplay Object Management holds the data storage classes associated with the entities that will be visible in the gameplay and is in charge of converting them into data where other components can interact with.

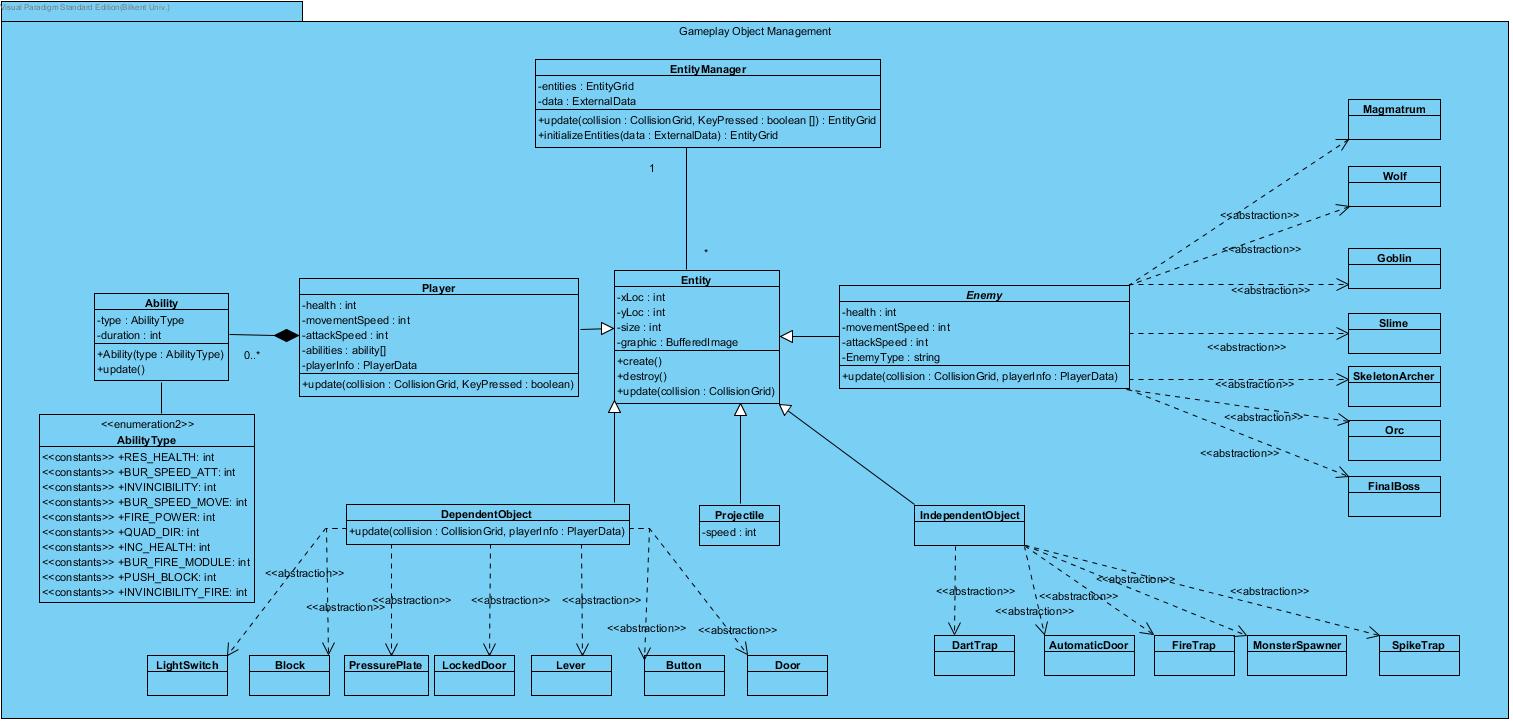


Figure 23 – Gameplay Object Management

EntityManager Class

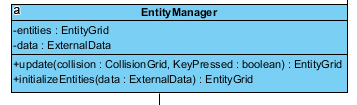


Figure 24 – EntityManager Class

This class is created to manage all entity objects in the game, this class has two variables, data from the external data and “entities” from the entity grid, it has two functions which are “update” and “initializeEntities”, these functions return entity grid which will be needed for running other classes such as “CollisionManager” class

Entity Class

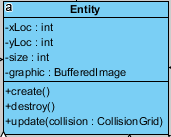
****

Figure 25 – Entity Class

Entity class has main variables which will be used for all objects, “xLoc” and “yLoc” will store location of the object, “size” will have area object occupies, “graphic” will have the image to use specifying object. “create” function will be used for creating this specified object in the level, “destroy” will delete the specified object from the level, and update will send recent data to the “EntityManager” class.

Player Class

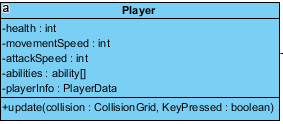
****

Figure 26 – Player Class

“Player” class will inherit the “Entity” Class and will be used to create a character user is going to use in the game. This entity will have health which will determine if game will be continued or not, “movementSpeed” will specify how fast Player will go into the field, “attackSpeed” will specify how frequently user will fire the projectiles, “abilities” will be used in order to boost the player’s stats temporarily or permanently, or give a new feature to the player. “update” method will send the recent data to the “Entity” class.

Ability and AbilityType Class

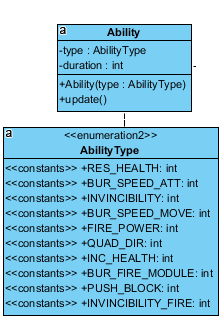


Figure 24 – EntityManager Class

“Ability” class will be used to change stats of player or give player a new ability. “type” variable will have specific ability which will be given to the player, “duration” variable will specify how many seconds this ability will last (10 seconds, permanently, etc.), “Ability(type)” is the constructor of the class, “update” method will send recent data to the “Player” class.

“AbilityType” class will store which abilities Player could get during the game. “RES\_HEALT” will restore the health of the player by 1, “BUR\_SPEED\_ATT” will increase movement speed by certain amount for certain duration or permanently, “INVINCIBILITY” will make player become immune to any source of damage for certain duration, “BUR\_SPEED\_MOVE” will increase attack speed by certain amount for certain duration or permanently, “FIRE\_POWER” will increase damage by certain amount for certain duration, “QUAD\_DIR” will allow player will fire towards all direction at once. “INC\_HEALTH” will increase maximum health by 1, “BUR\_FİRE\_MODULE” will allow player fire more than one projectile at once, “PUSH\_BLOCK” will allow player push certain blocks, “INVINCIBILITY\_FIRE” will protect player from the fire based damage

DependentObject and its abstract Classes

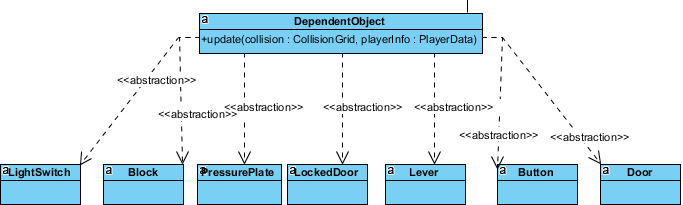


Figure 25 – Dependent Object and its abstract Classes

“DependentObject” class will contain dependent objects who will response according to the player data if the collision occures. “LightSwitch” abstract class will be used for turning off and on the light. “Block” abstract class will be used for block object which could be pushed if player gets related ability. “Button” abstract class will contain objects which will trigger some other objects. “PressurePlate” abstract class will have objects that have same principle with button but will be triggered by applying pressure on it. “Lever” abstract class will contain objects with same principle with button, yet levers could be triggered only once. “LockedDoor” abstract class will be used for passing next level, if player has a key, object will allow player to pass next level. “Door” abstract class will allow player to pass other parts of the current room, unlike the “LockedDoor” object, it will not require key.

Projectile Class



Figure 26 – Projectile Class

“Projectile” class will contain projectiles player and enemies fire, it will have “speed” variable which define how fast it will move from one pixel to another.

IndependentObject and its abstract Classes

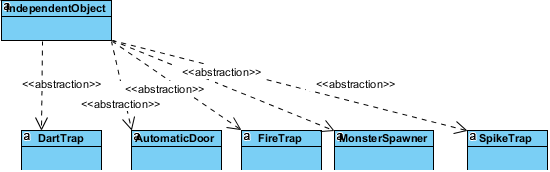


Figure 27 – IndependentObject and its abstract Classes

“IndependentObject” class will consist of object which will stay active by not depending on player data or occurrence of collision. “DartTrap” abstract class will create objects that shoot dart at the fixed direction, “AutomaticDoor” abstract class will create objects which will open and closed in a period, “SpikeTrap” abstract class will create objects with meele version of the dart trap, “FireTrap” abstract class will create objects with same principle with dart trap, but their range is between spike trap and dart trap, “Monster Spawner” abstract class will consist of objects which spawn mosnters in a period.

Enemy and its abstract Classes

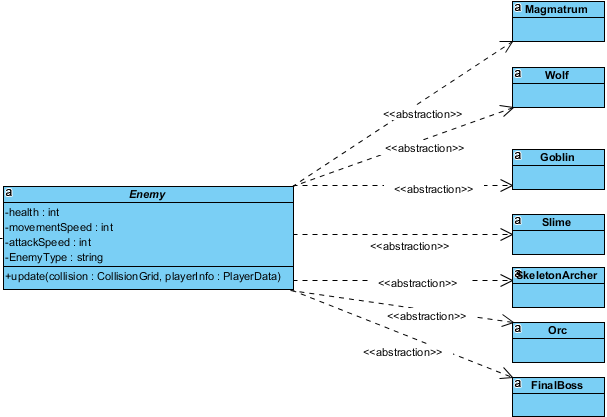
****

Figure 28 – Enemy and its abstract Classes

“Enemy” class will be the main structure of the enemies which player will deal with to access next level. It has “health” variable which will determine if enemy is destroyed or not, “movementSpeed” will determine how fast enemy can move, “attackSpeed” will determine how fast enemy can fire projectile, “EnemyType” variable will determine type of the enemy, so system will use specified graphic for the this enemy. It will send the recent data to the “Entity” class with “update” method. “Slime” abstract class will create weak enemy that moves randomly, “Wolf” abstract class will create slightly stronger and faster enemy than “Slime”, “Orc” abstract class will create slower but stronger enemy than “Wolf”, “Goblin” abstract class will create enemy like Slime, yet moves in fixed route, “SkeletonArcher” abstract class will create stationary enemies which shoot arrow from a distance, “Magmatrum” abstract class will create objects which is a stronger enemy than “Slime” with special effect which changes a terrain to a “Scorched Earth” that damages the player when player touched it, “FinalBoss” abstract class will be used to create the enemy at the last room, so destroying final boss will end the game with a success, it will move fast, track the player and attack him with projectiles from a distance.

Level Management

Level Management Subsystem is in charge of parsing the external data it retrieved into complete grid of level data which will later be used by the GraphicsAssembler class in Game Mechanics Management

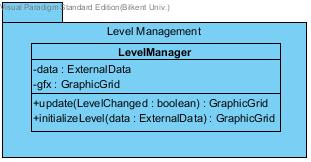


Figure 29 – Level Management

LevelManager Class

LevelManager Class’s functionality is loaded on update() method. Within update() method, the instance of ExternalData, data will be scanned and the method will fetch the Level map section of the data and read through the data, while filling up the GraphicGrid with the corresponding graphic data.

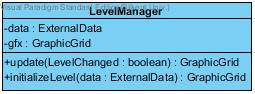


Figure 30 – LevelManager Class

Miscellaneous (Data Storage classes)

As the Java lacks the support of the struct-like feature as the C++ supports it, we had to make some data storage classes in order to avoid cluttering up a method’s parameter list and creating similar methods which all it does is returning single variable. **This is a significant trade off when it comes to readability and reusability since when a method has parameters which requires data storage class instead of primitive data type parameters, it requires the class which wants to reuse that certain method to also import the data storage classes as well and reduces flexibility. However, we are willing to take this tradeoff for the sake of readability/performance.**

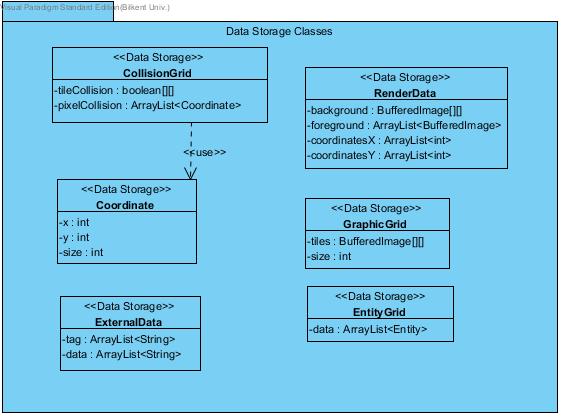


Figure 31 – Data Storage Classes

This specific component was not mentioned in the component diagram as **every component** will use this component (in order to understand the data being passed around).