**authBilkent 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 3](#_Toc446090630)

[1.1 Overview 3](#_Toc446090631)

[1.2 Design Goals 3](#_Toc446090632)

[2. Software Architecture 4](#_Toc446090633)

[2.1 Subsystem Decomposition 4](#_Toc446090634)

[2.2 Hardware/Software Mapping 6](#_Toc446090635)

[2.3 Persistent Data Management 6](#_Toc446090636)

[2.4 Access Control/Security 6](#_Toc446090637)

[2.5 Boundary Conditions 7](#_Toc446090638)

# 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)
  + **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)
  + **3**. **L**iskov Substitution Principle – Child Class and Parent Classes should be interchangeable
  + **4**. **I**nterface Segregation Principle – An interface should not have redundant methods that are never used.
  + **5**. **D**ependency Inversion Principle – Parent Class should not be dependent of Child Class
* **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

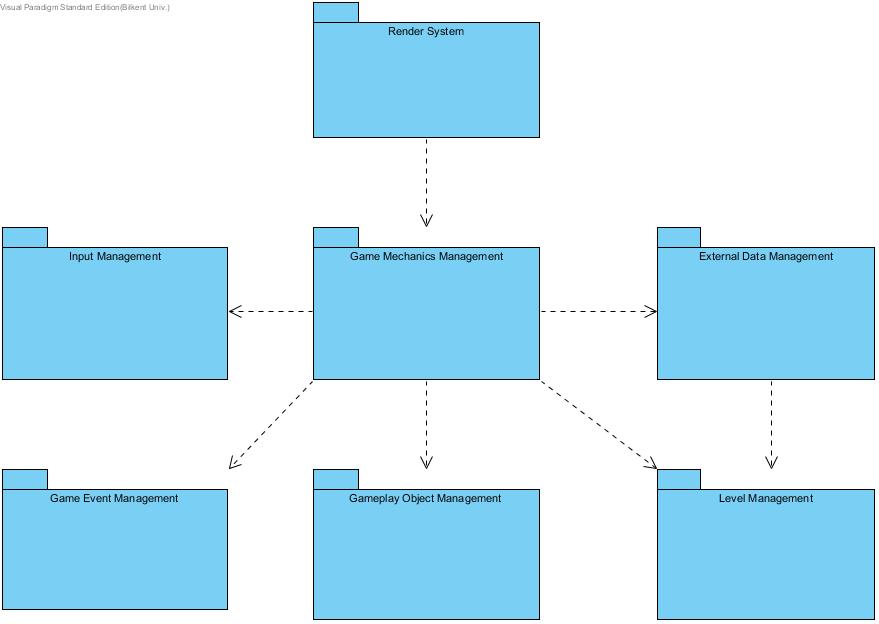
We will use three layer architecture for our game, JCrawl. Our layers will be composed of Render Layer, Game Mechanics Layer, and Gameplay Object Layer. Render Layer will be strictly restricted to rendering the actual game onto the screen and nothing else, this layer will be composed of Platform and 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.

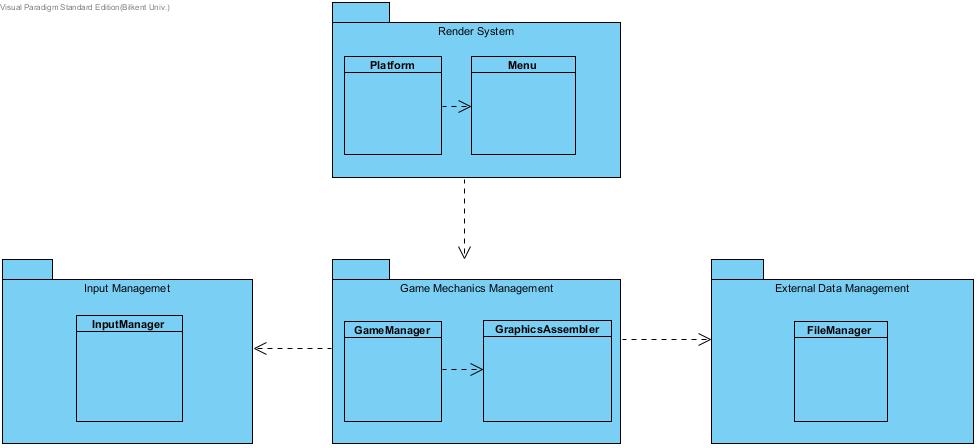


Figure 2 – Interaction between Render Layer (Layer 1) and Game Mechanics Layer (Layer 2)

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.

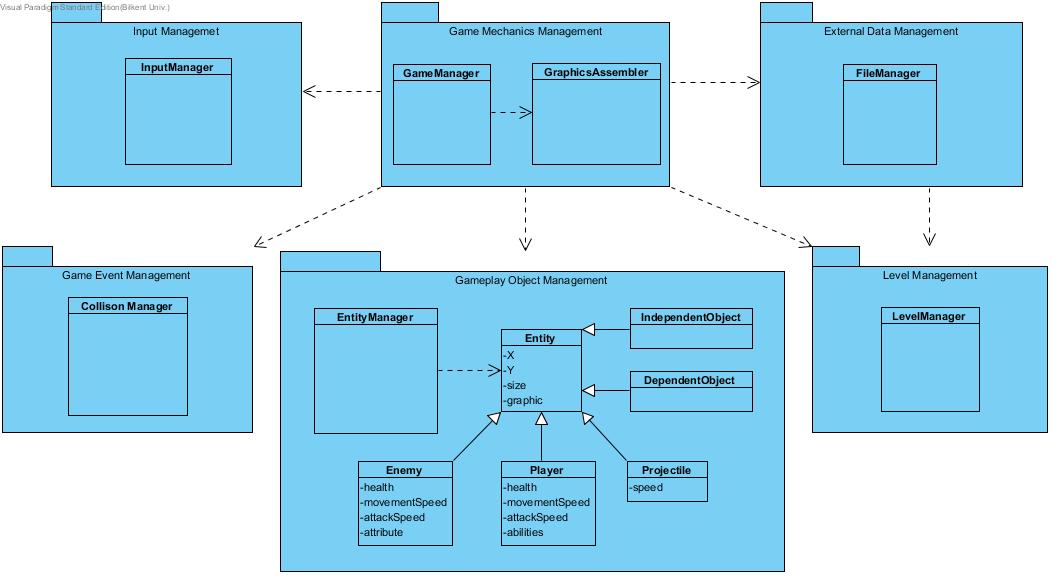


Figure 3 – Interaction between Game Mechanics Layer (Layer 2) and Gameplay Object Layer (Layer 3)

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

## 3.1 Detailed Object Design

Details for the object design is shown in the figure 4 which is a class diagram, in next page.

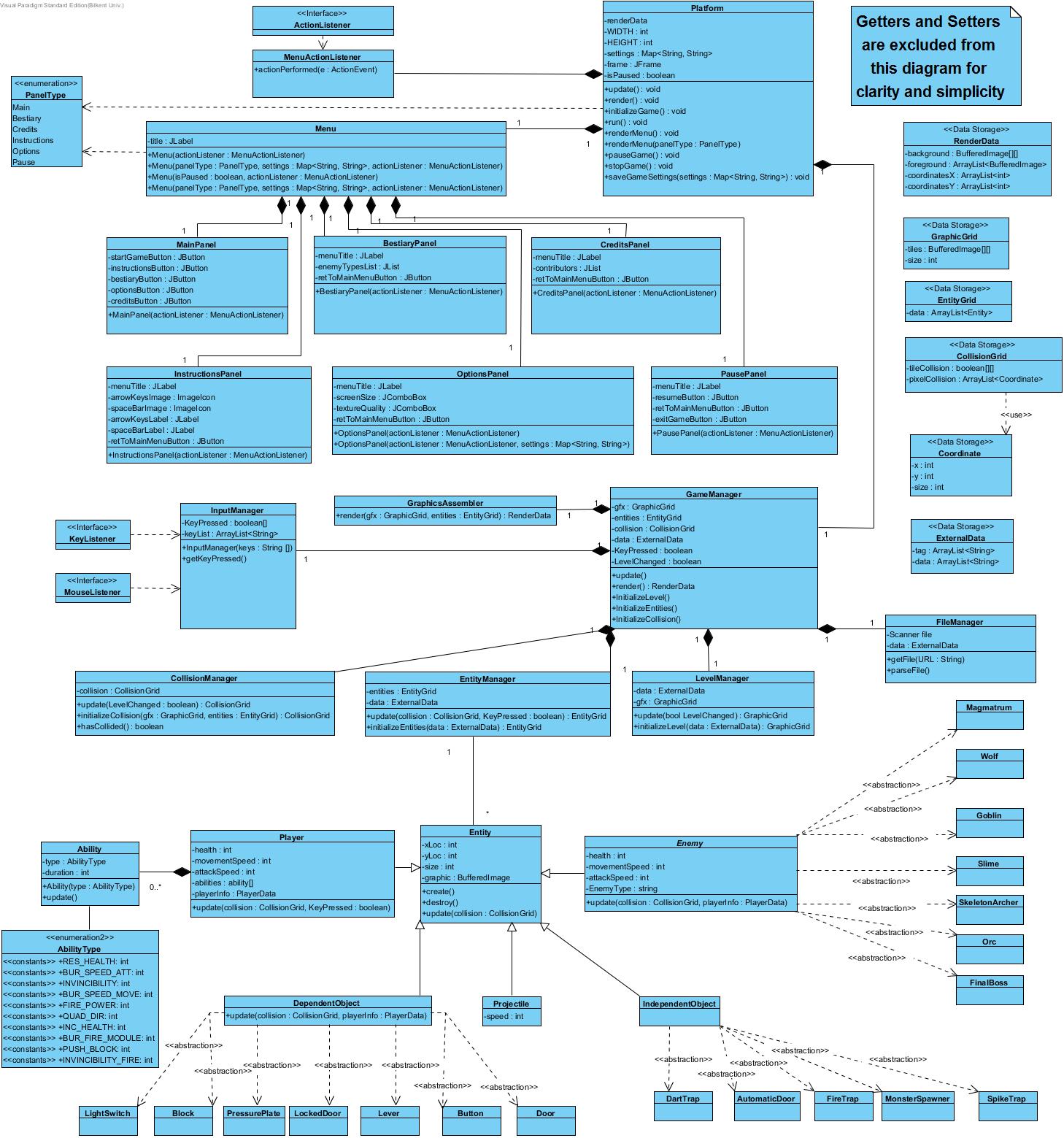


Figure 4 – In-depth Class Diagram

## 3.2 Render System

## 3.3 Game Mechanics Management

## 3.4 Input Management

## 3.5 External Data Management

## 3.6 Game Event Management

## 3.7 Gameplay Object Management

## 3.8 Level Management

## 3.9 Miscellaneous