Bilkent University

Department of Computer Engineering

Object Oriented Software Engineering Project

CS319 Project Group 1G: Super Mario Bros.

Design Report

Ahmet Çandıroğlu, Unas Sikandar Butt, Umut Balkan, Mert Sezer

Course Instructor: Uğur Doğrusöz
Course Assistant: Hasan Balcı

Progress/ Iteration 2

Dec 02 2017
This report is submitted to the Department of Computer Engineering of Bilkent University in partial fulfilment of the requirements of the Object Oriented Programming course CS319/1

Contents

1 Introduction

1.1 Purpose of the System

Super Mario is a classic game known to almost every person from the '90s. The main purpose of the system is to provide a similar game, with minor changes. The game can be classified as an adventure and level based game. Comparing to current standards of gaming, our system may be lacking in the graphics and complexity departments, nonetheless the game will provide a challenging gameplay which is addictive and really enjoyable. The difficulty level of the game is intentionally going to be kept hard so that the game can become a platform for gamers to compete on.

1.2 Design Goals

<u>Reliability</u>
The game should be reliable in the sense that it should be able to handle any problems. For example if the CPU is too crowded the game should report a warning message or if the keyboard or mouse is not connected there should be an error message.

<u>Extendibility</u>
The best of systems are the ones which are extendable. The game will be implemented in such a way that will permit future additions and development. This way the game can be improved and new features can be added to the current system.

<u>Portability</u>
We want the game to be available to as many users as possible. So portability is a must for the system. The game will be developed in the java environment which is supported by most modern machines today. This will enable us to target greater clients.

Rapid development

This project has a time constraint of one semester. Taking that into account rapid development has to be carried so that the project can be finished in time.

Minimum number of errors

Ideally the game has to be completed without a single error. However we know that we may not be able to accomplish that, so one of the design goals is to have a minimum number of errors so that the system can be as close to perfection as possible.

Ease of learning

The user of any game has to first understand how to play the game. Most modern games provide a tutorial at the start of the game so that the player can get accustomed to the game. Our game however, if compared to most modern games, is very simple. So a simple controls menu should be enough for the player to understand how the game is to be played.

Modifiability

As our system will be implemented using the Object Oriented Programming principles, the system will be kept modifiable. During the design stage we will try to minimize coupling and maximize coherence. This in turn will make it easier to modify any given features of the game.

2 Software Architecture

Before designing the system the most suitable architectural style has to be chosen. For our project the MVC architecture is the most suitable. The MVC stands for Model View Controller. This style of architecture also comes with its own advantages:

1. Faster development

In this architecture the system undergoes system decomposition, this allows for a faster and parallel development process.

2. Modification

Using MVC enables modifications to subsystem parts without affecting the other parts. This makes future updates to the system easier.

3. Low coupling

This architectural style decouples data access (entity objects) and data presentation (boundary objects).

3 Subsystem Decomposition

According to the MVC architectural style the whole system has been decomposed into three main components. The controller package contains all the classes that deal with control, and is further divided into three subsystems which are GraphicsEngine, InputEngine and PhysicsEngine. Similarly we have a model package which represents objects in the game. These classes contain states, locations, types, abilities etc. The view package contains classes that are related to the boundary objects mainly the user interface and design. The composition can be visualized as shown below.

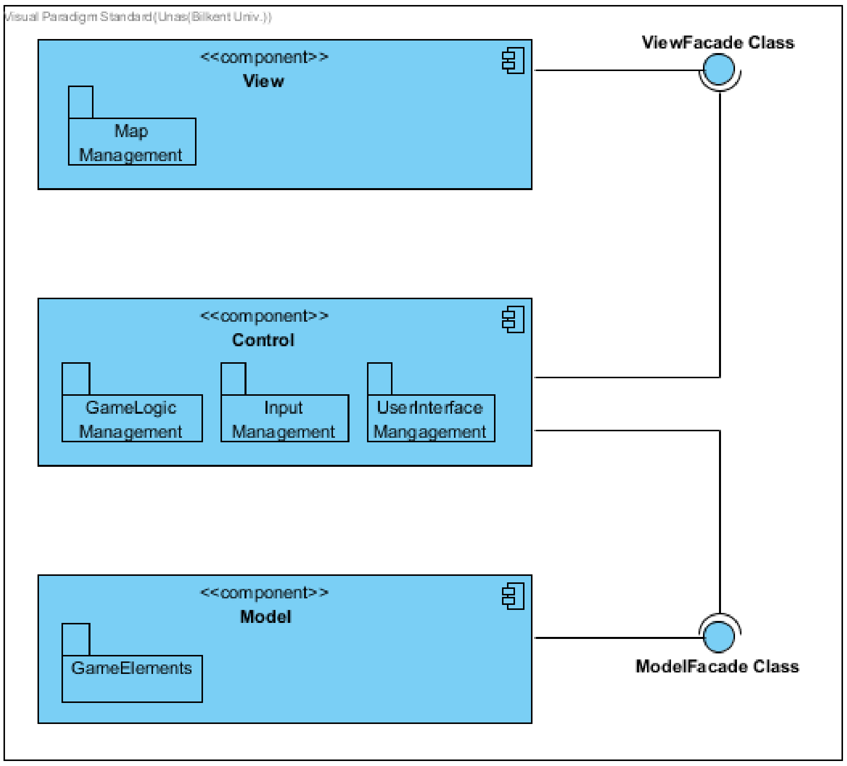

Figure 1: High-level Composition Diagram

Decomposing the system in this way promotes high cohesion and low coupling which is good in itself but it also helps us achieve some of our design goals.

3.1 Hardware/Software Mapping

As the game will be developed in a java environment first and foremost a java compiler will be required which can compile and run the .java file. For hardware specification, a keyboard is required and a mouse (optional). No internet connection is required to play the game. As for the computer any modern computer with average specifications should be able to run the game. A graphics processing unit (GPU) can be optional just to further enhance the performance of the game. In the OS criteria any OS which supports java environment will be compatible. Also for the data storage the computer hard disk will be used to store and load the save game data.

3.2 Persistent Data Management

Most of our in game data consists of the map, Mario, monsters and bricks. To increase performance these objects will not be drawn instead they will be kept on the hard disk as image files and fetched directly. We hope this will make the game much smoother. Other than game image files we will also have some sound files for the in game music and sound effects. These files will have the .wav file format. Apart from these as mentioned earlier the project will also have some saved game files. These files will be kept in the .txt file format. To aid the simplicity of the project no complex database system will be used.

3.3 Access Control and Security

Playing this game the user will have no issues with the security of the system. The access control will be with all the users on the machine, be it administrator or otherwise. Also there should be no chance of game files going corrupt as the user or any third party software should not be able to modify them. Also as the game needs no internet connection there is no chance of any viruses or malicious files being downloaded.

3.4 Boundary Conditions

Initialization

The game will be executed using a .jar file (java archive). The .jar file will contain all the class files and resources (texts, images). The .jar file is a suitable distribution method because it contains all the data needed for the game in one file. Also and there is lesser chance of an individual file getting corrupted. At start upstartup the program will access the user saved games files and user configuration files and load them. As the user starts the program he/she will be shown the main menu page, where the user can load a previously loadsaved game, start a new game, select the level from the map selection menu, view controller options or change game settings.

Termination

The user can terminate the program by using the exit button on the main menu. Alternatively, if the user wishes to exit while playing the game they can use the pause button wish displays the pause menu. From the pause menu the user will have to first go to the main menu and then exit from there. Another method of termination can be directly from the system task manager.

Error

Our program being very simple should have minimum possible errors. Since we are not dealing with a system that does parallel processing or has multiple users working at the same time or something like client/server architecture or service-oriented architecture, our system will not have many complicated errors. The online errors we may face will be run time execution failing or corrupted game files. If the system running the game has a low processing power or low random access memory the game may also face some errors or may get stuck. However, as modern systems have average processors ranging from 2.0-3.0 GHz these problems may never occur.

4 Subsystem Services

4.1 Detailed Object Design

Class diagram for whole system is shown in next page. It is divided into subsystems which makes it easy for people to develop.

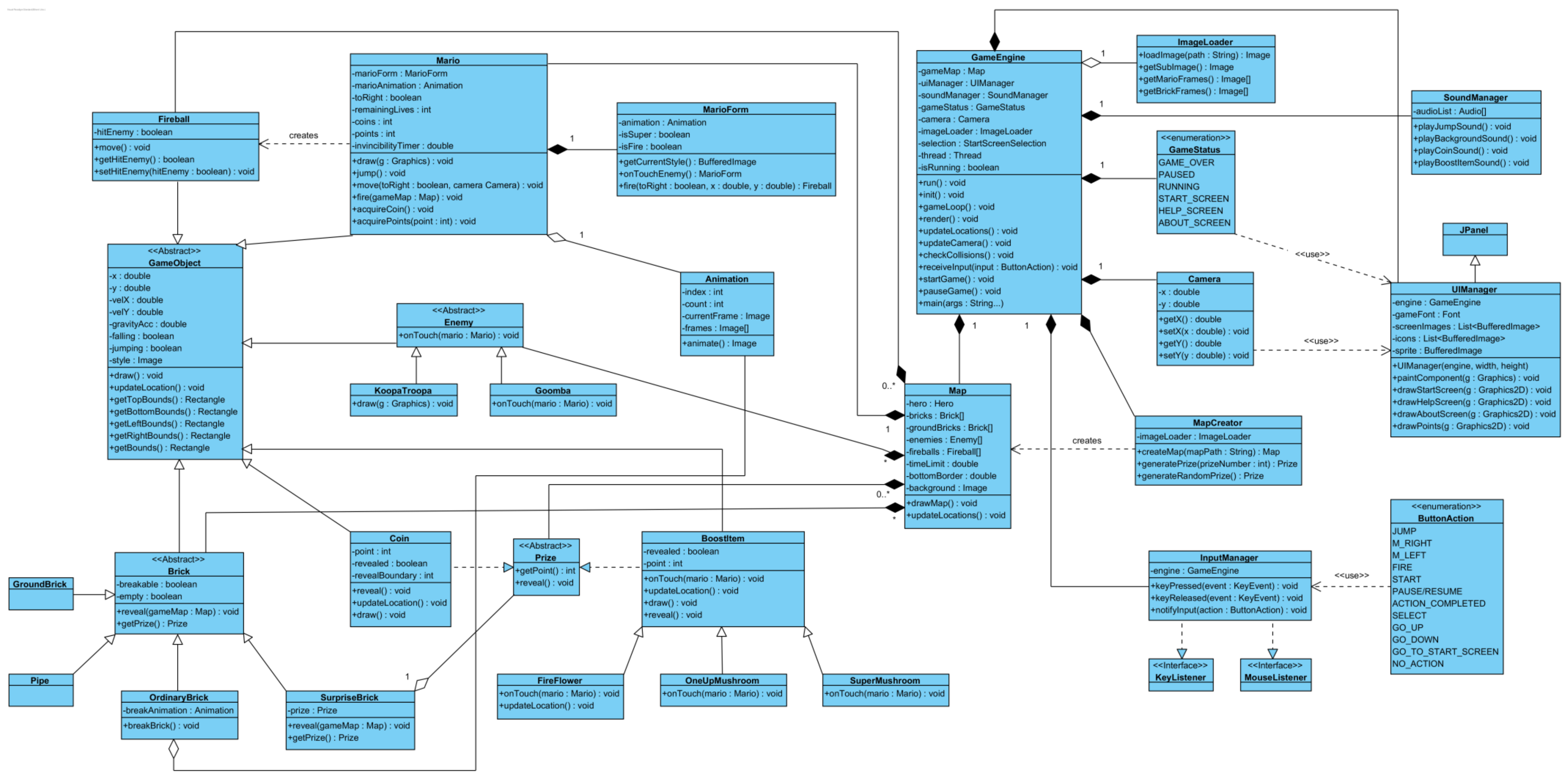

Figure 2: Object Diagram

4.2 User Interface Management Subsystem

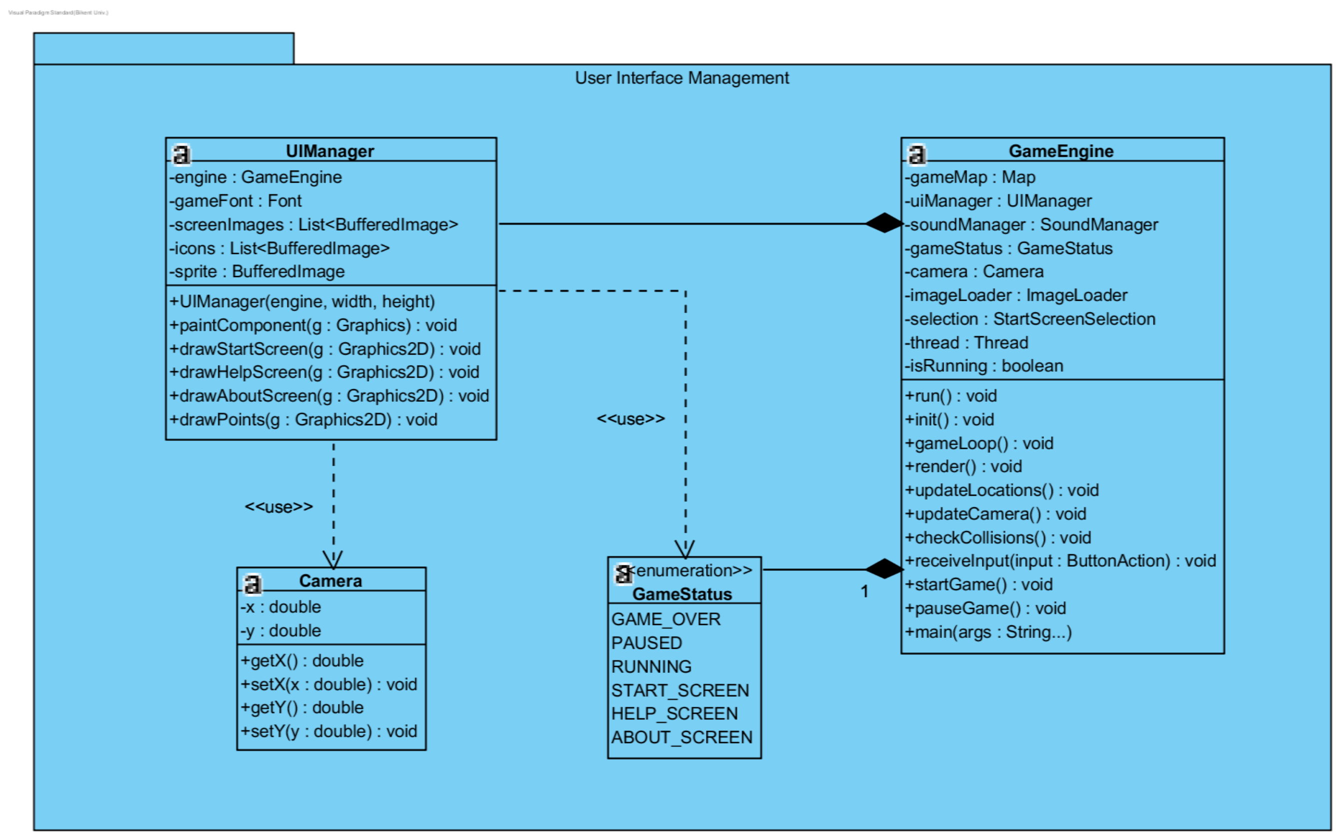

Figure 3: UI Management Subsystem

User Interface Management Subsystem consists of one main class which is UIManager, an enumeration GameStatus and a realized class, GameEngine. UIManager class extends javax.swing.JFrame. It has height and width properties which defines frame size. The game has many statues so a GameStatus enumeration is defined. According to the GameStatus frame will be rendered differently. If game is running then normal game map will be rendered. Else if game is on pause, pause menu panel will be rendered and so. Game status of UIManager can be changed via its set method. This can be done by GameEngine which is the controller of the game.

4.3 Game Logic Subsystem

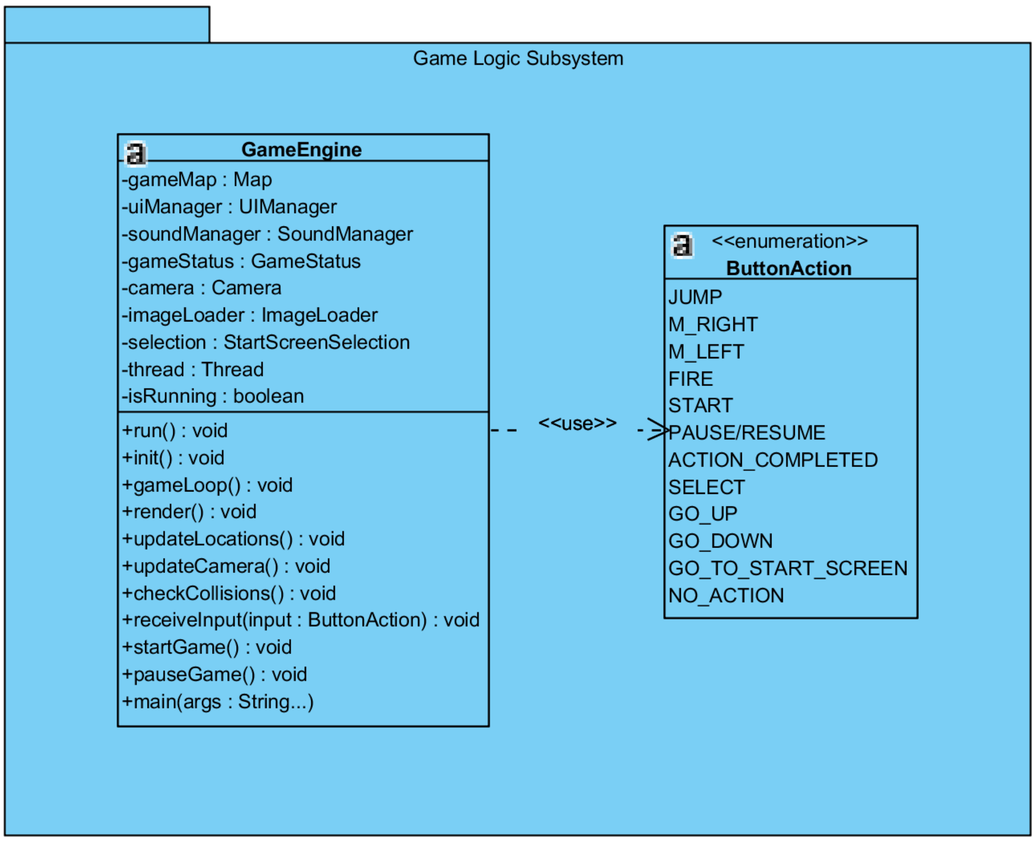

Figure 4: Game Logic Subsystem

Game Logic subsystem consists of only one class which is GameEngine, main class of the game. GameEngine is the controller class of our game which is MVC modelled. So, this class gathers information from outside, makes calculations and let's view class to know everything user should know. To be specific, GameEngine has an instance of InputManager class which gets the input from user via keys. There are several action can be created by user and that is why ButtonAction enumeration is needed. There are actions like JUMP, M_RIGHT, M_LEFT and FIRE which are hero controlling actions. Others like START, PAUSE/RESUME are general actions about game.

GameEngine gets input (if there is any) in each game loop and changes state of hero or game status accordingly.

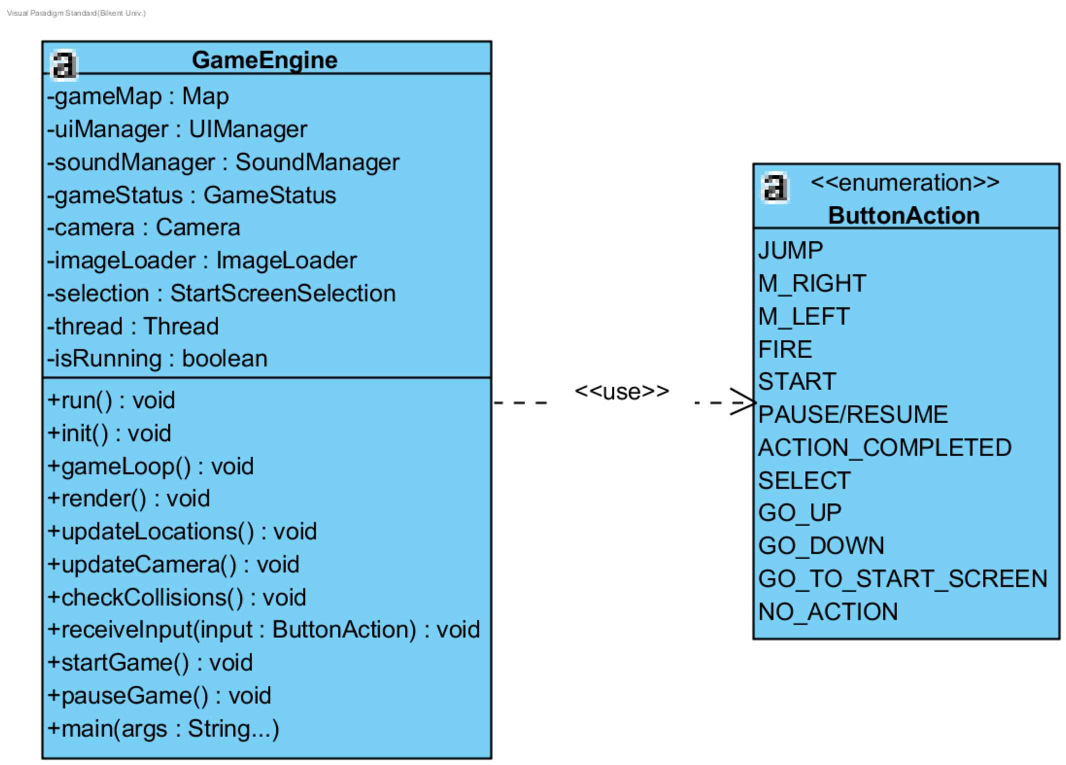

Figure 5: Game Engine Class

GameEngine calls gameLoop() method in each loop which calls several methods to make sure either game is going or not. First of all remaining time and lives are checked. Then, positions of all objects are updated via updateLocation() method. This method calls updateLocations() of Map instance which will call updateLocation() method of each GameObject (Façade design pattern). Finally collisions checked and state of enemies, bricks and Mario changed accordingly.

4.4 Game Screen Elements Subsystem

Game elements subsystem consists of model classes which represents all the objects in the game. This subsystem can be divided into parts which makes it easy to examine whole subsystem.

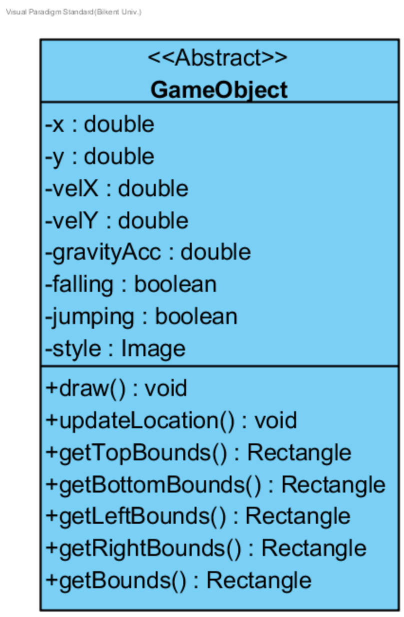

Figure 6: GameObject Class

GameObject class is an abstract class which every object in the game extends to. It has all common information which every object has such as velocity, location and style and methods such as draw(), updateLocation() and getBounds(). Some of the methods are overridden in some of its subclasses according to subclass behaviour (e.g. draw() method is overridden in Prize classes they are invisible when they are not revealed).

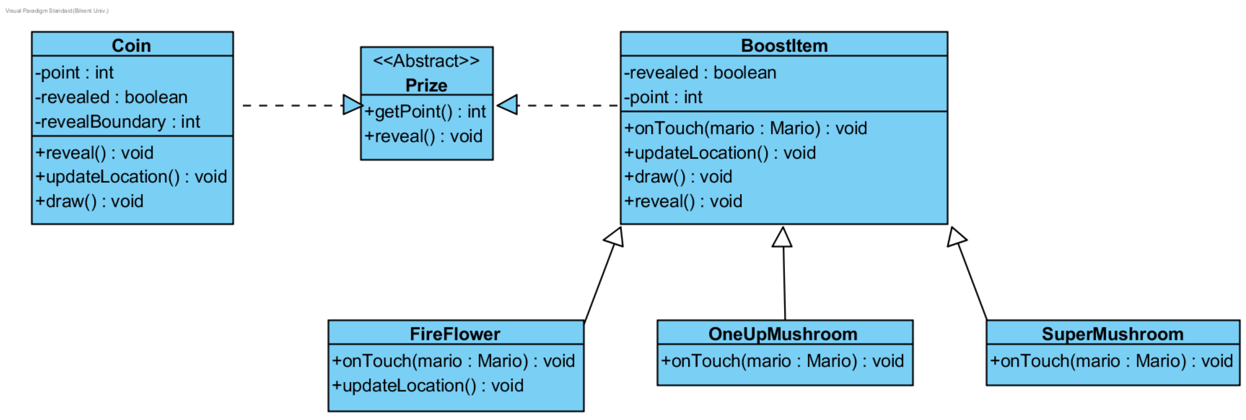

Figure 7: Prize Classes

BoostItem represents mushrooms and other boost items which change form of Mario. There are many type of BoostItem which is represented by subclasses of BoostItem. Each subclass overrides onTouch() method of BoostItem to act differently on Mario. Prize is an abstract class and is parent of Coin and BoostItem. Coin can be acquired by the hero which gives points. BoostItem's can move when they are revealed however Coin's cannot.

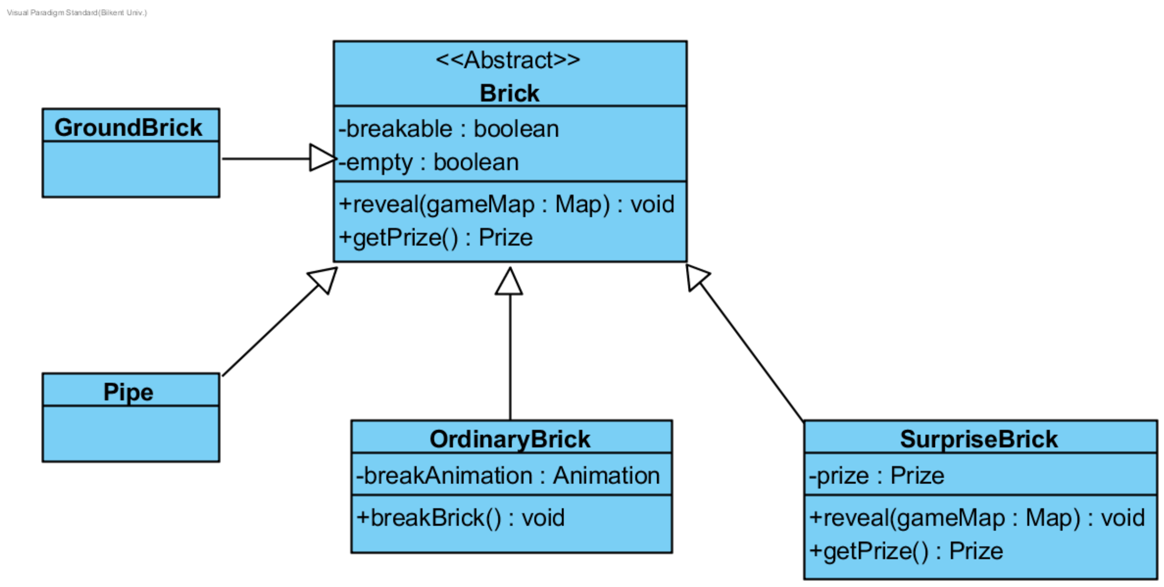

Figure 8: Brick Classes

Brick class represents bricks on the map which has lots of types and that's why subclasses are defined. Brick can be breakable or unbreakable by Mario when he is in super form depending on its flag. Only SurpriseBrick's have prizes in them and they will be revealed when Mario hits from bottom. OrdinaryBrick's have Animation class instance which will be used to animate its breaking animation.

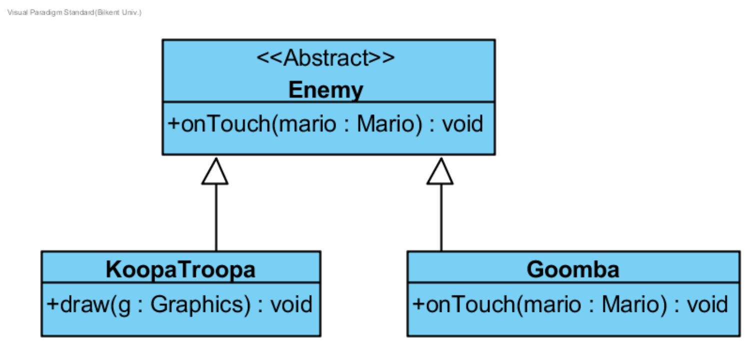
Visual Paradigm Standard(Bilkent Univ.)
<<Abstract>>
Enemy
+onTouch(mario : Mario) : void
KoopaTroopa
+draw(g : Graphics) : void
Goomba
+onTouch(mario : Mario) : void

Figure 9: Enemy Classes

Enemy is as the name implies enemies of the hero which can be killed by smashing. They can kill the hero in case the hero touches them. There are many enemy types and these types define enemies' movability, direction of move etc.

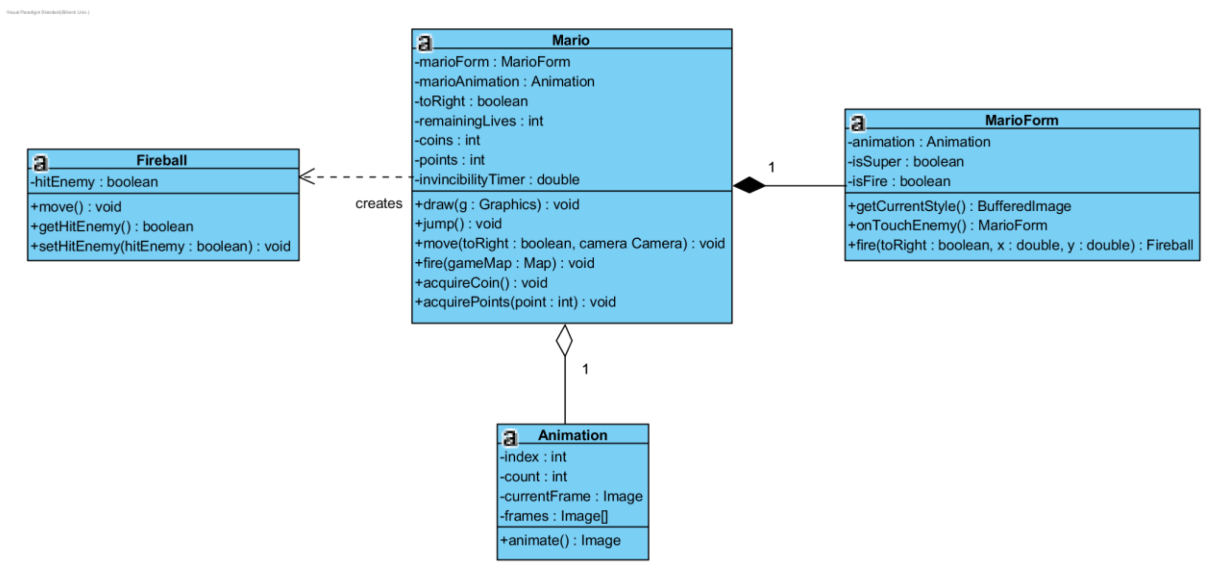
Visual Paradigm Standard(Bilkent Univ.)
Mario
-marioForm : MarioForm
-marioAnimation : Animation
-toRight : boolean
-remainingLives : int
-coins : int
-points : int
-invincibilityTimer : double
+draw(g : Graphics) : void
+jump() : void
+move(toRight : boolean, camera Camera) : void
+fire(gameMap : Map) : void
+acquireCoin() : void
+acquirePoints(point : int) : void
Fireball
-hitEnemy : boolean
+move() : void
+getHitEnemy() : boolean
+setHitEnemy(hitEnemy : boolean) : void
creates
MarioForm
-animation : Animation
-isSuper : boolean
-isFire : boolean
+getCurrentStyle() : BufferedImage
+onTouchEnemy() : MarioForm
+fire(toRight : boolean, x : double, y : double) : Fireball
1
1
Animation
-index : int
-count : int
-currentFrame : Image
-frames : Image[]
+animate() : Image

Figure 10: Mario Classes

Mario is the main hero in the game which will be controlled by user. Mario can move and jump. Mario has different types of forms which defines his strength. Mario can break breakable bricks in SUPER form, can throw fire in FIRE form and is invincible in INVINCIBLE form in a brief time period. His form will change according to enemy contact (if he touches enemy he returns to small form, if he is in small form he loses one life) and obtained BoostItems. MarioForm class represents all these information about him and will be altered when Mario changes form. Fireball can be thrown by Mario only in FIRE form which can kill enemies.

Animation class is a new class which is implemented to animate movement of Mario. It contains sprite sheet (image sheet) for Mario and animate() method will change style (image) of Mario according to its state.

4.5 Input Management Subsystem

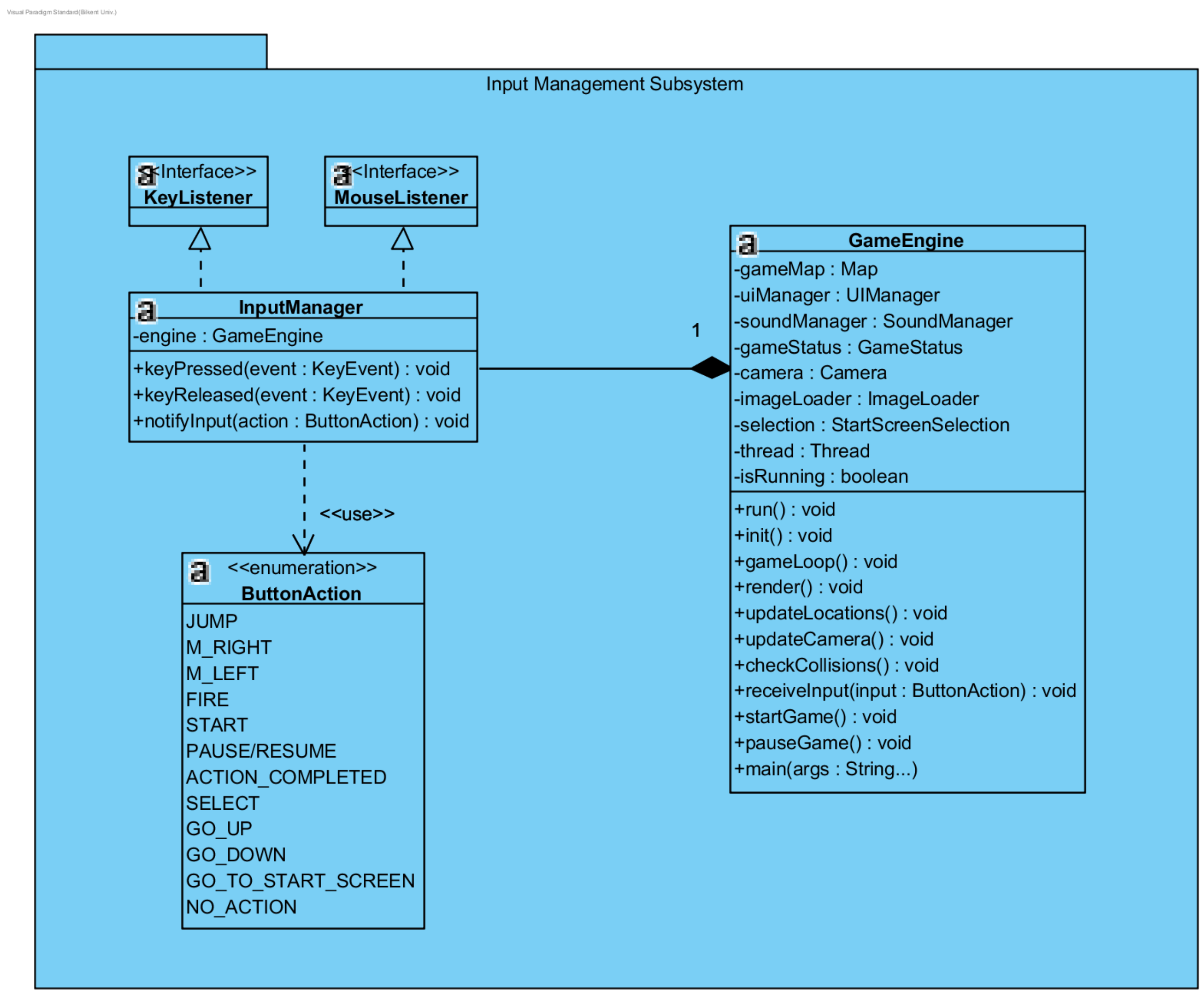

Figure 11: Input Management Subsystem

Input management subsystem consists of InputManager class. InputManager gets input from the user and first checks given input in notifyInput() method. If the given input is needed to be given to GameEngine class, notifyInput() calls GameEngine's receiveInput() method with given input as an argument. This method updates game accordingly. InputManager extends KeyListener and MouseListener of Java and overrides keyPressed(), keyReleased(), mousePressed() methods.

4.6 Game Map Management Subsystem

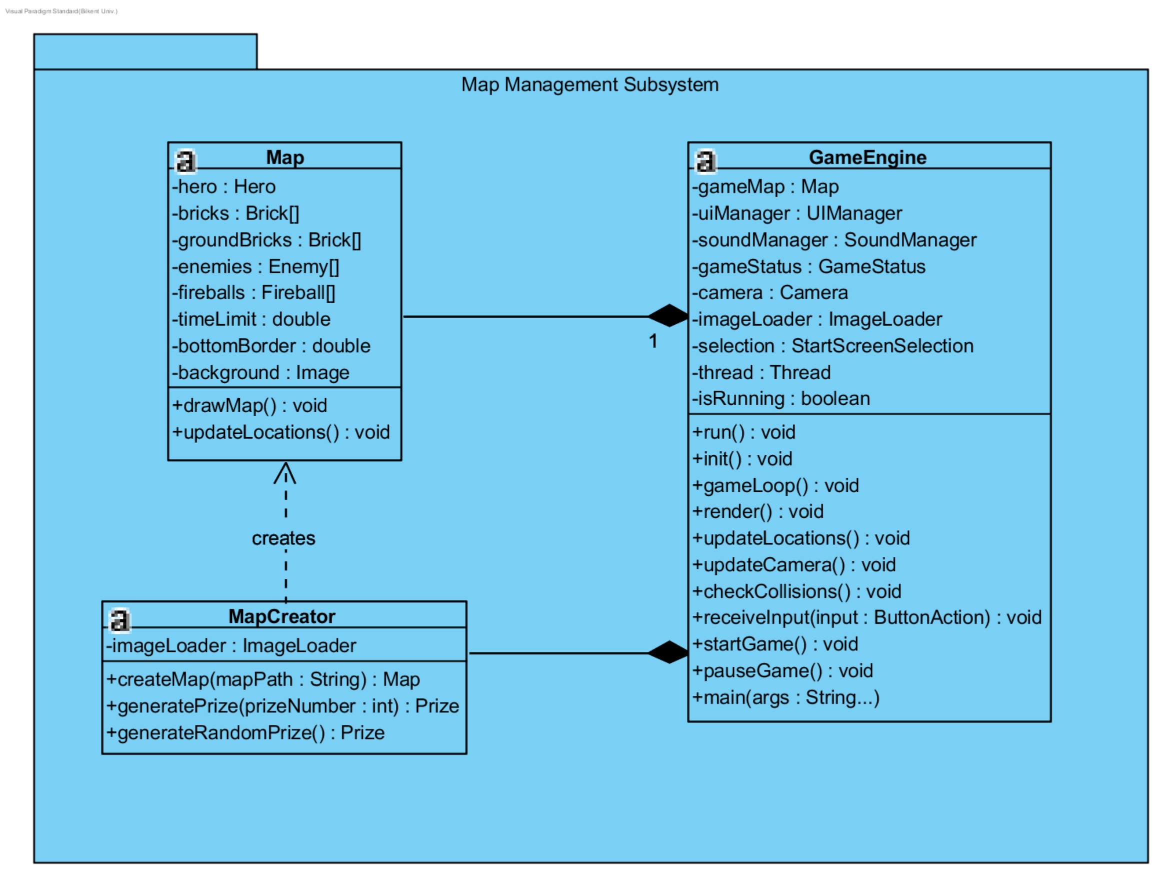

Figure 12: Map Management Subsystem

Map class contains everything that can be seen in a map. Map class is Façade class for game elements and it is contains different objects for different map types. When user selects a map to play (in map selection screen), corresponding Map instance is created by MapCreator class and returns this instance to GameEngine. Map class notifies each GameObject it has according to GameEngine's commands and calls their updateLocation() and draw() methods. This way we reach polymorphism.

4.7 Sound Management Subsystem

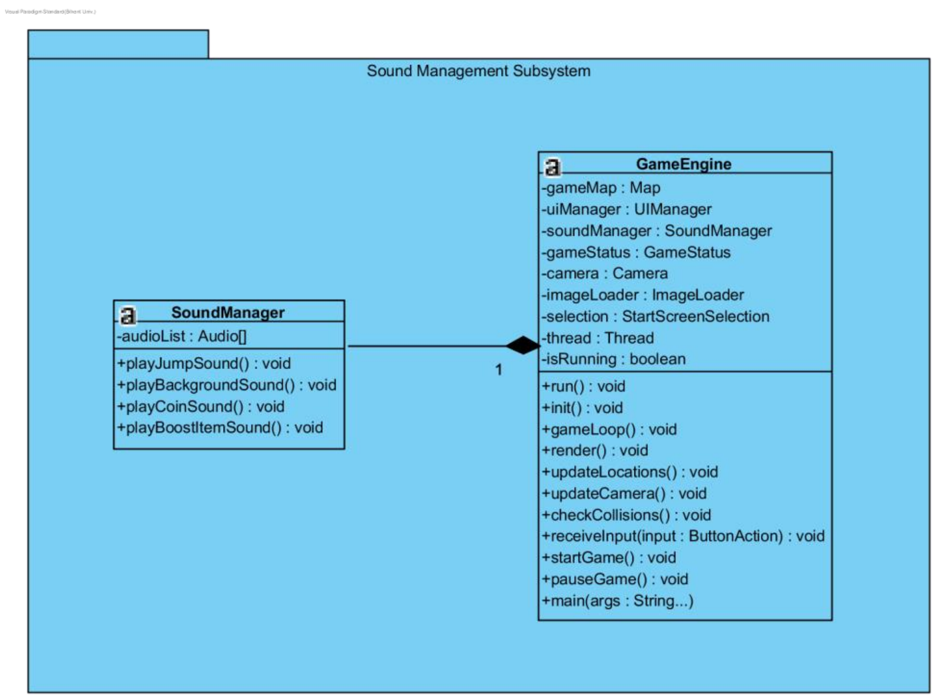

Figure 13: Sound Management Subsystem

SoundManager is a new class we add for second iteration. SoundManager will play audio according to states of GameEngine. For instance when Mario jumps SoundManager will be informed and playJumpSound() will be called.

5 Design Trade-offs

Compatibility: Because the users with access to the game can have different computer architectures, the users should be compatible with each other in the same time frame. We are implementing standard interfaces so that there is less need for data conversion.

Throughput: Because we want the game to be available to as many users as possible; our system must be able to handle parallel requests. In order to do that, our system uses event-based control flow; in which the system provides access to multiple users. Although this increases throughput, latency cost decreases.

Modularity: In order to be able to change the parts of the system without affecting the whole system or without changing the interfaces between different parts of the system, our system must promote modularity. To provide modularity, our system is divided into 5 modules (Game Logic Management, Game Elements, Map Management, Input Management, User Interface Management).

6 Improvement Summary

6.1 Design Patterns

Facade design pattern - Implemented for Map and GameEngine classes which have access to many model classes and packages. For instance GameEngine class cannot directly use Mario, Brick etc. classes; can only use methods of Map classes. Facade pattern makes implementation more reliable since we would not be able to do unintended changes in model classes.

Singleton design pattern - Implemented for Map, UIManager, InputManager and SoundManager classes since we need only one instance of these classes. We made sure that we have only one instance of these classes by using Singleton pattern.

Player-role design pattern - Implemented to handle form changes of Mario. We had subclasses of abstract class Mario which represent Mario forms like SuperMario, FireMario etc. When player gets a bonus or touch an enemy we used to delete current Mario instance and replace it with a new one accordingly. However we saw that these would cause problems due to difficulty of keeping track of these instances. We created a new class, MarioForm, to represent only behaviours of specific Mario forms and Mario class has an instance of this class. So, when we need to change Mario form all we need to do is alter MarioForm instance accordingly.

6.2 Changes in Classes

GameObject - Added features

- `velX, velY, gravityAcc : double` - Every game object will have velocity and gravitational acceleration. Gravitational acceleration may differ from object to object to indicate air resistance of different objects will be different.
- `falling, jumping : boolean` - Indicates whether object is falling or jumping.
- `updateLocation() : void` - This method will calculate position of the object in next frame according to its velocity and falling/jumping flags. This way we don't need to calculate physics in game separately.

MarioForm - New class

- This class represents behaviours of Mario forms and added to implement Player-role design pattern.
- Removed subclasses of Mario (SuperMario, FireMario, SmallMario) when we added this class.

SoundManager - New class

- This class will handle all sound work and added to improve the project in second iteration analysis report.

Animation - New class

- Handles animationS.

Camera - New class

- Handles camera movement according to Mario.

Brick - Changed structure

- BrickType is removed and implemented subclasses instead.
- Added - GroundBrick, Pipe, OrdinaryBrick and SurpriseBrick subclasses.