



Bilkent University

Department of Computer Engineering

Object Oriented Software Engineering Project

CS319 Project Group 1G: Super Mario Bros.

Design Report

Ahmet Çandiroğlu, 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.....	3
1.1. Purpose of the System	3
1.2. Design Goals	3
2. Software Architecture	4
3. Subsystem Decomposition.....	5
3.1. Hardware/Software Mapping	6
3.2. Persistent Data Management	6
3.3. Access Control and Security.....	7
3.4. Boundary Conditions	7
4. Subsystem Services	8
4.1. Detailed Object Design	8
4.2. Controller	9
4.3. View	11
4.4. Model Subsystem	12
5. Design Trade-offs	15
6. Improvement Summary	15
6.1. Design Patterns	15
6.2. Changes in Classes	16

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

Reliability

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.

Extendibility

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.

Portability

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

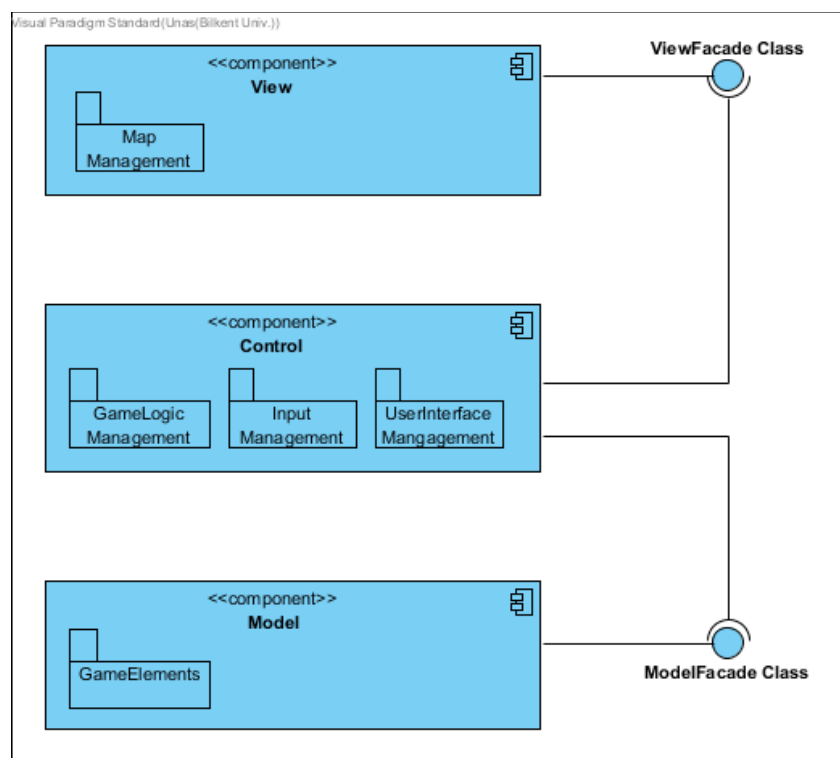
Using MVC enables modifications to subsystem parts without affecting the other parts. This makes future updates to the system easier.

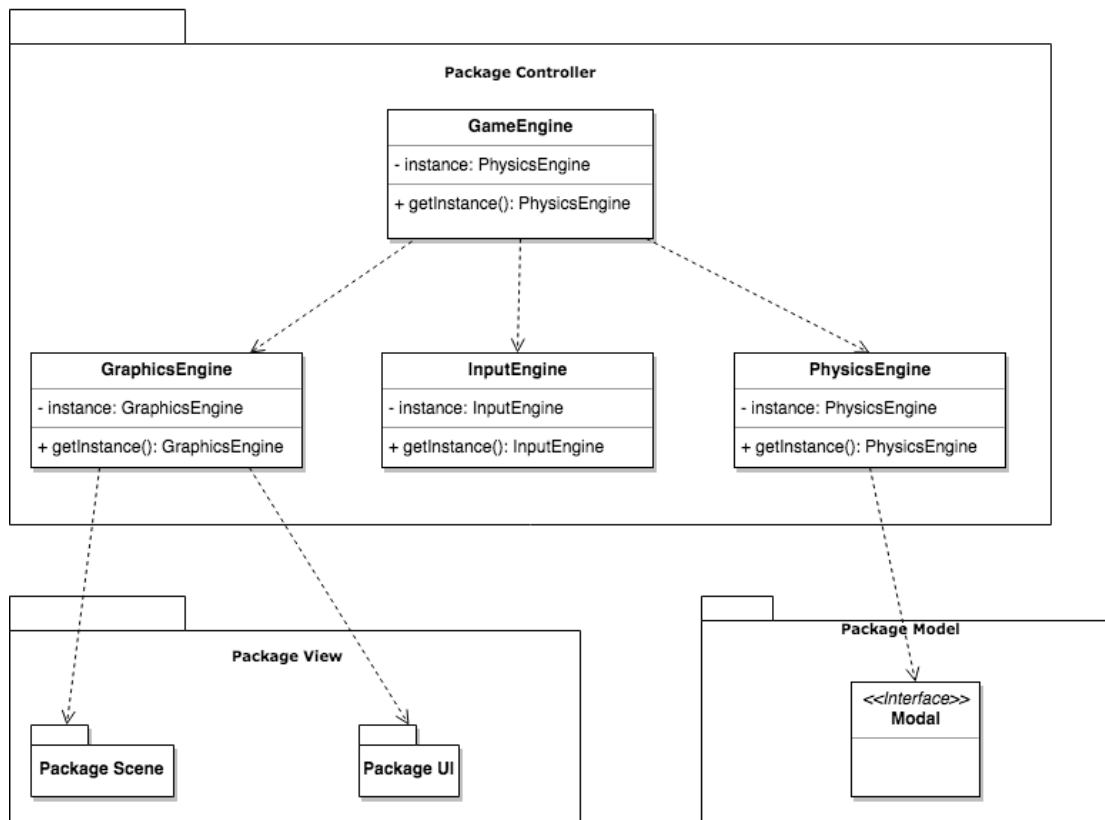
3. Low coupling

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.





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 up startup 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 which 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

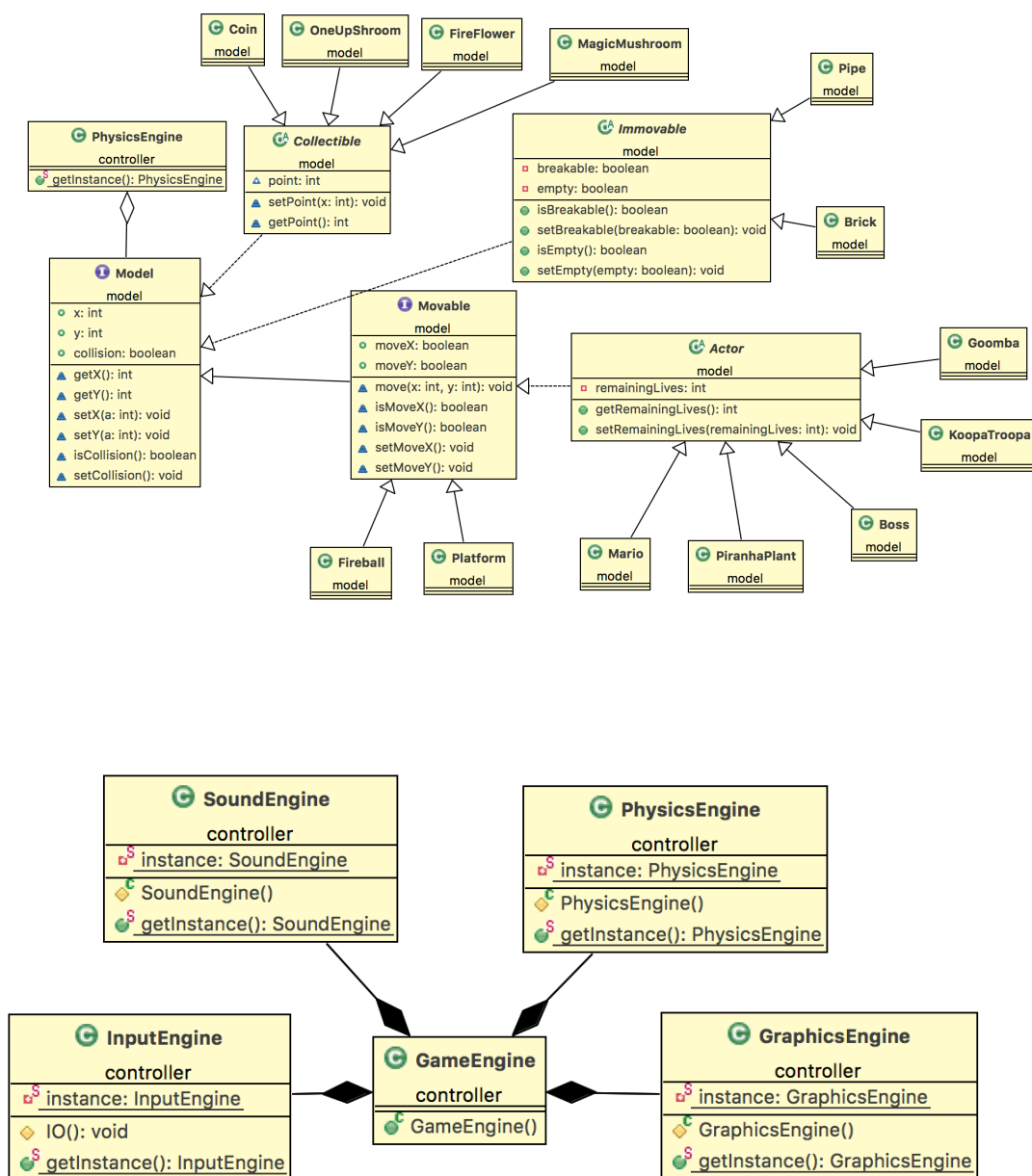
Our program being very simple should have minimum possible errors. Since we are not dealing with a system that does parallel processing or has multiple users working at the same time or something like client/server architecture or service-oriented architecture, our system will not have many complicated 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.



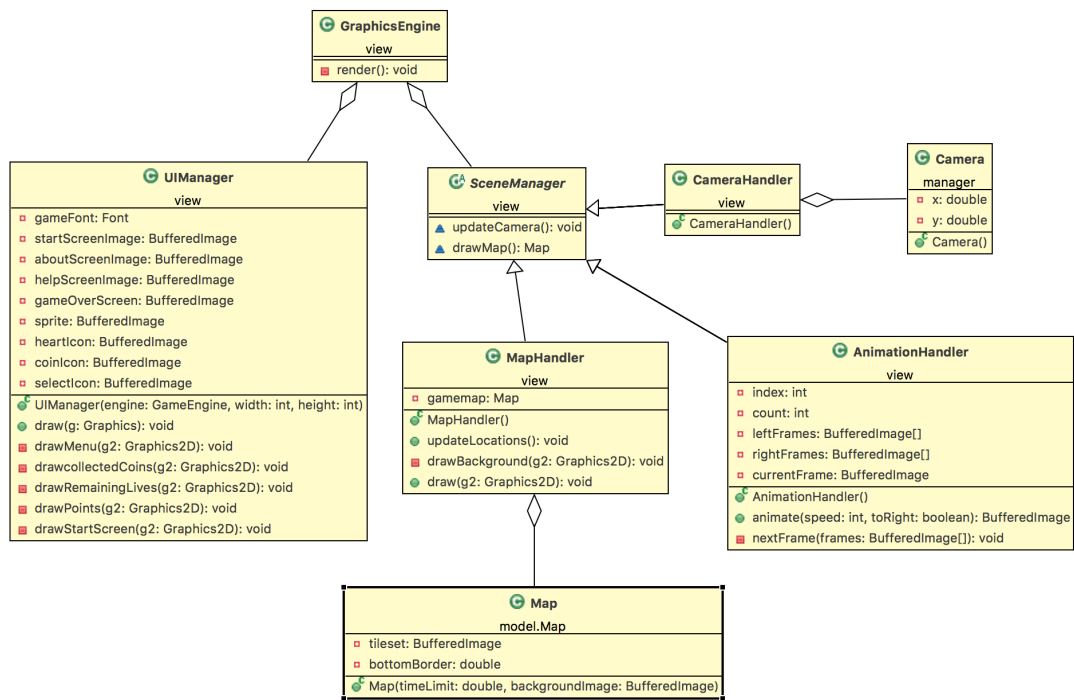


Figure: Detailed Design

4.2. Controller

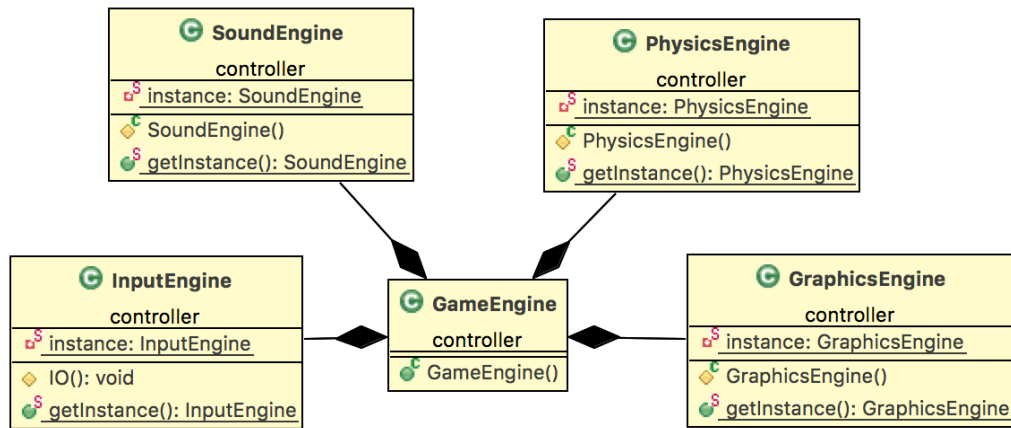


Figure: Package Controller

Controller subsystem consists of 4 classes InputEngine PhysicsEngine and GraphicsEngine and 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 InputEngine class which gets the input from user via keys. 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.

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 GraphicsEngine which will call updateLocation() method of each Model object(Façade design pattern). Finally collisions checked and state of enemies, bricks and Mario changed accordingly.

SoundEngine will play audio according to states of GameEngine. For instance when Mario jumps SoundEngine will be informed and playJumpSound() will be called.

GraphicsEngine handles all the drawing including Animation, Map, User Interface, Camera.

InputEngine 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.

4.3. View

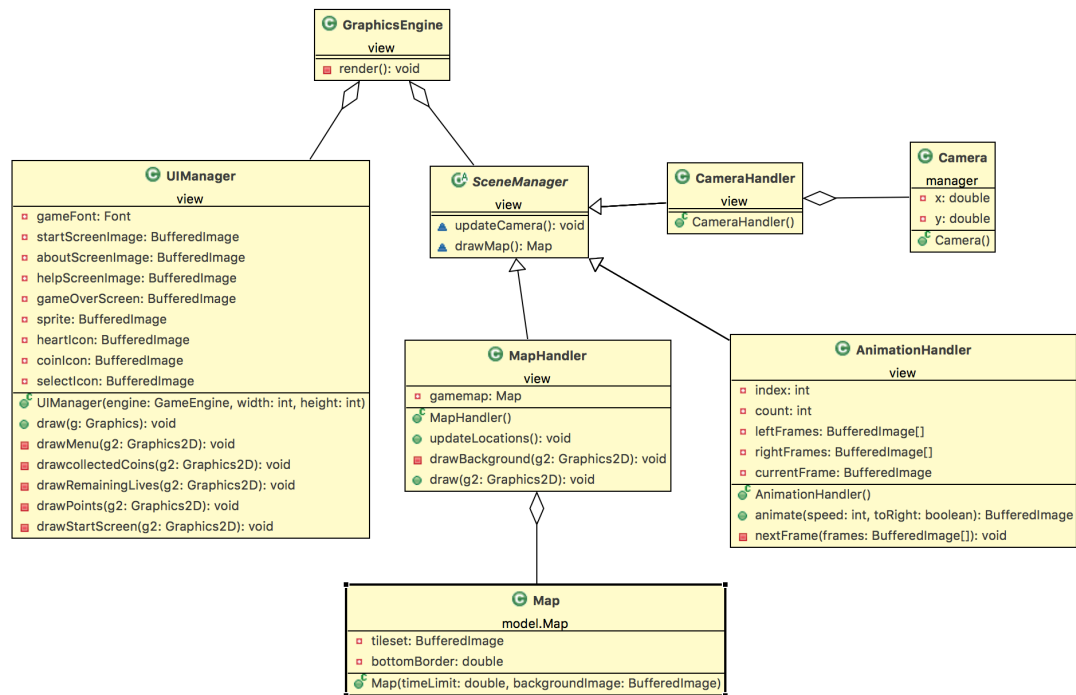


Figure 4: View Subsystem

User Interface Management Subsystem consists of two main class which are UIManager and SceneManager. UIManager class extends javax.swing.JFrame. It has height and width properties which defines frame size. Basically the class responsible for user interface and user interface only (viz. Start Menu, Pause Menu, Settings)

The game has many statues so a GameStatus enumeration is defined in GraphicsEngine. 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 GraphicsEngine can be changed via its set method. This can be done by GameEngine which is the controller of the game.

4.4. Model Subsystem

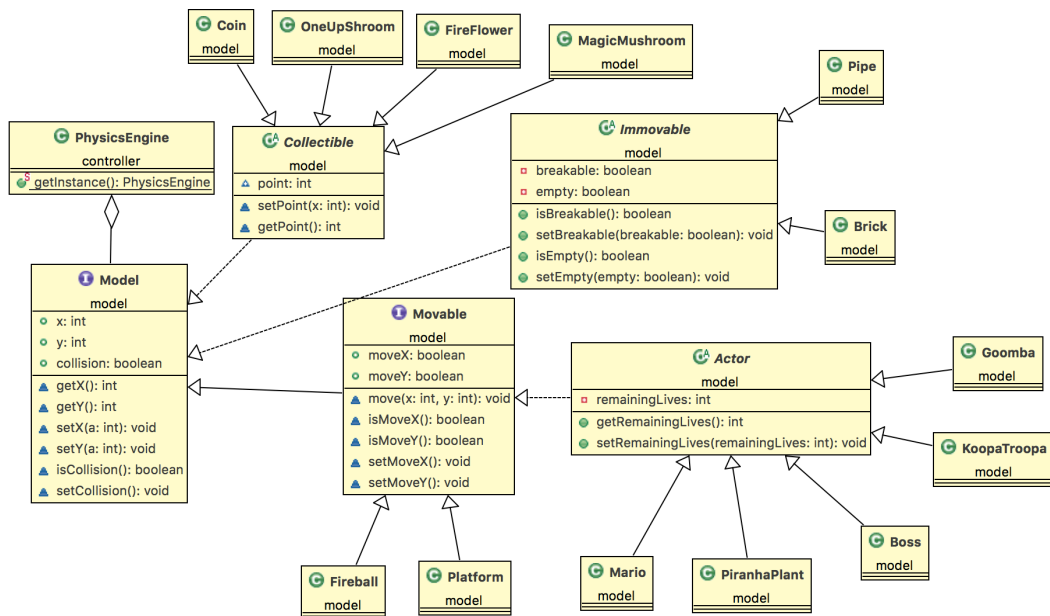


Figure: Model Subsystem

Model 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.

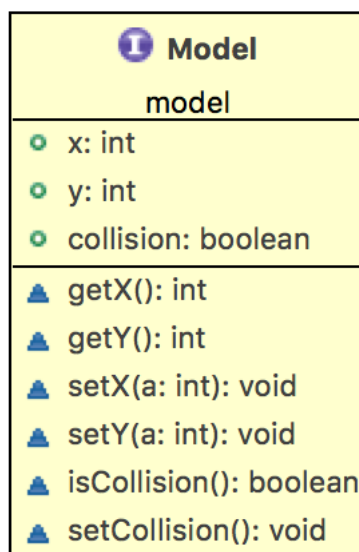


Figure: Model Class

Model 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 and methods such as, setX() and isCollision().

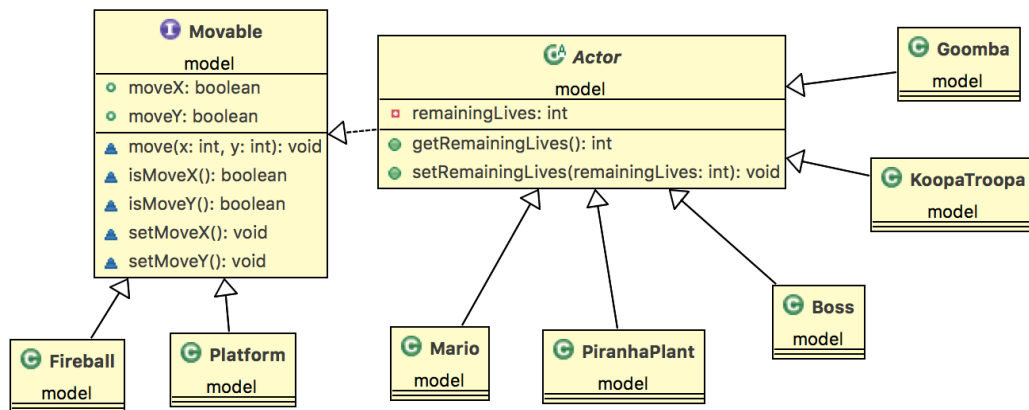


Figure: Movable Classes

Movable represents every movable object in the game system. This division allows us to inherit common methods such as `move()` and their attributes such as `xPos` and `yPos`. Actor class inherits movable even further by adding `remainingLives` attribute and a method for controlling the common behaviour between active objects (Mario, Enemies etc.)

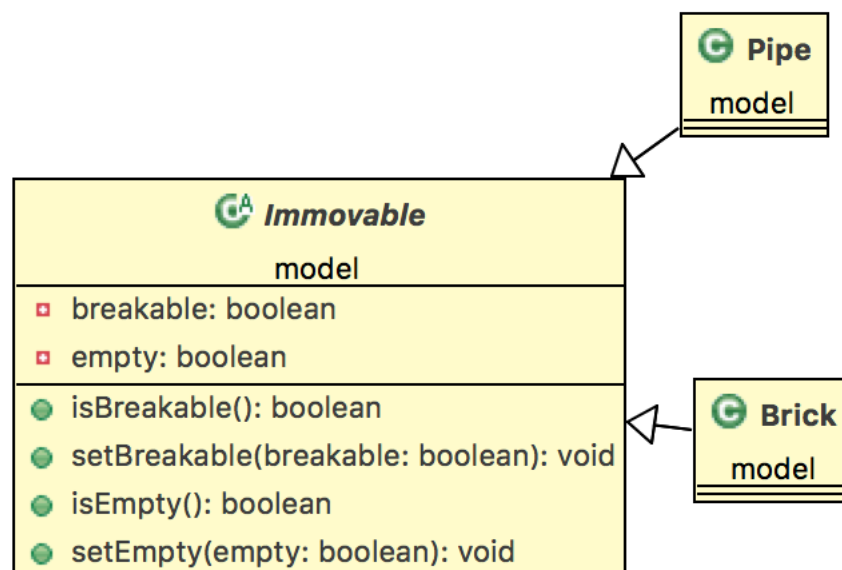


Figure: Immoveable Classes

Classifies common objects such as brick and pipe. They are drawn after the map tiles and basically they are collidable but not movable. This hierarchy allows us to manipulate objects common behavior at abstract level

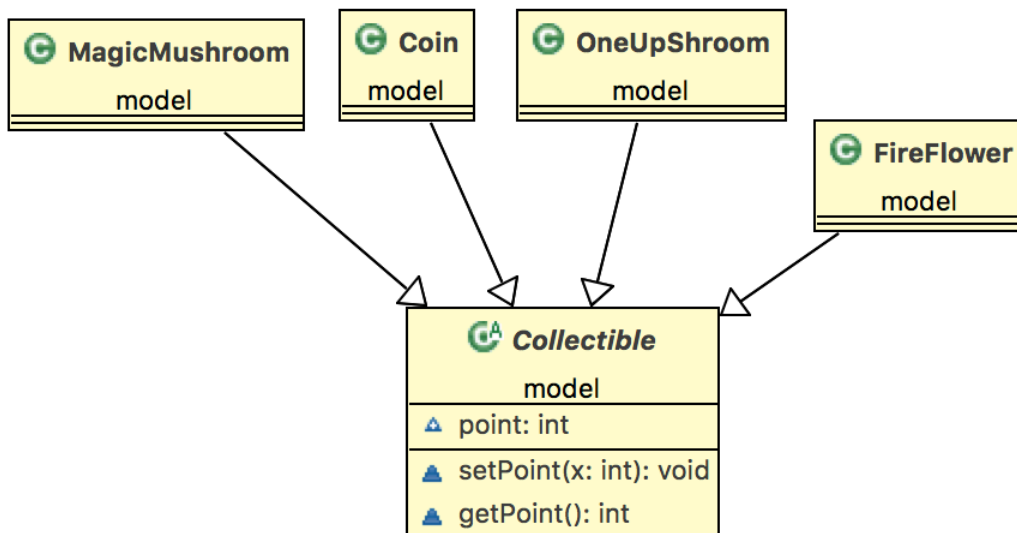


Figure: Collectible Classes

Class contains collectible objects and their effect on the game system (points, bonuses etc.)

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 3 modules (model, view, controller

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, added hierarchy between object types such as Movable, Collectible, Actor etc.

- `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.

SoundManager - New class

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

PhysicsEngine - New class

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

ViewEngine - New class

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

AnimationManager - New class

- Handles animationS.

CameraManager - 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