BILKENT UNIVERSITY

ENGINEERING FACULTY

DEPARTMENT OF COMPUTER ENGINEERING

CS 319

Object Oriented Software Engineering Project

e-Bola

Design Report

Group 2B

Can AVCI

Deniz SİPAHİOĞLU

Ergün Batuhan KAYNAK

Esra Nur AYAZ

Course Instructor: Bora Güngören

December 2, 2017

1 Introduction

1.1 Overview

In this section, we represent the purpose of the system, which is mainly entertaining the player. We listed the design goals in this part, in order to achieve our purpose. Our design goals are providing portability, maintaining an easy use of interface, learning the game easily, smooth gameplay and extendable design. We had to decide between some trade-offs during the project which will be explained in part 1.4.

1.2 Purpose of the System

e-Bola is a basic 2-D game which encourages the player to develop a game strategy in order to fight the enemies and the boss. The player will have multiple choices to choose from while choosing the items and attributes. There is no "correct item" or attribute to win the game, so everything can work. This is what makes the game enjoyable and skill-dependent. The game is very basic compared to today's 3-D games, which are made with thousands of dollars budget and an experienced team of developers. With e-Bola, our aim is to ensure that the player enjoys the game while improving his reflexes and decision making.

1.3 Design Goals

1.3.1 End User Criteria

Easy to learn:
The game takes very basic inputs from the user (4 to 5 keystrokes), so the player will be able to get familiar with the game very quickly. The concepts other than the controls will be explained in the Tutorial level, which can be accessed from the initial menu. The tutorial will consist of several images which explain the basics of the game. We will try to keep the tutorial as short and as informative as possible. We aim to keep it under a minute so that the player can start the game sooner. The tutorial won't cover everything, so the player will have the excitement of exploring other features of the game by himself.

Easy to use:
When the player initializes the game, the menu navigation and the buttons that he is going to use will be crystal clear. The control buttons will be pre-assigned and will be the same with the other games. For example, ESC for pausing and viewing the menu, arrow keys or 'WASD' for movement, just like the other games that the player might've played.

1.3.2 Maintenance Criteria

Extendibility:
We are aware of what keeps the game going is its extendibility. No matter how good the game is, the player will get bored after some time if the developers don't create new content for the game. Since our game is level based, it is very important to make it expendable. We designed the classes in a way that it is very easy to add new levels, monsters, attributes and even new territories to the game. It is only limited to the creativity of the developer. Also, we will add the developer's contact information to the credits page, so that the user can give suggestions of what he wants to see in future versions of the game.

Portability:
There are lots of operating systems to choose from, but in terms of portability and ease of use, we choose to develop our game in Java. This way, we will be able to reach more players, since Java Virtual Machine provides platform independency and our game will be accessible from every computer that runs java.

Reliability:
Every possible scenario will be tested out level by level, so that the user won't witness any crashes or face unexpected events in the game, since these will be handled accordingly.

Modifiability:
Modifiability is very high at this game. Even without the source code, the user can change the text file related to the levels, and create custom levels. However the game may not start if it's done incorrectly.

1.3.3 Performance Criteria

Response Time:
This game requires immediate action from the system when there's any kind of input from the user. That's why the response time will be one of our main focuses. Animations will also respond to the players' input to create smooth gameplay environment.

1.4 Trade-offs

1.4.1 Efficiency and Reusability

Instead of adding more features such as more weapons, attributes and monsters, we kept things as simple as possible to make the game easier to play

and learn. However, the game has so much potential to add functionality, so we can increase the functionality of the game later on if it's necessary.

1.4.2 Memory Use and Efficiency

We store most of the data in text files. Using text files increases our extendibility and modifiability. However, if the user knows how to make the system and hidden files visible, it can easily be used to cheat. Such user can reach any data he has on the game and modify it, such as level, attributes, attack and defence points. Also, the user can damage the file by mistake, and this may cause problems in the initialization of the game.

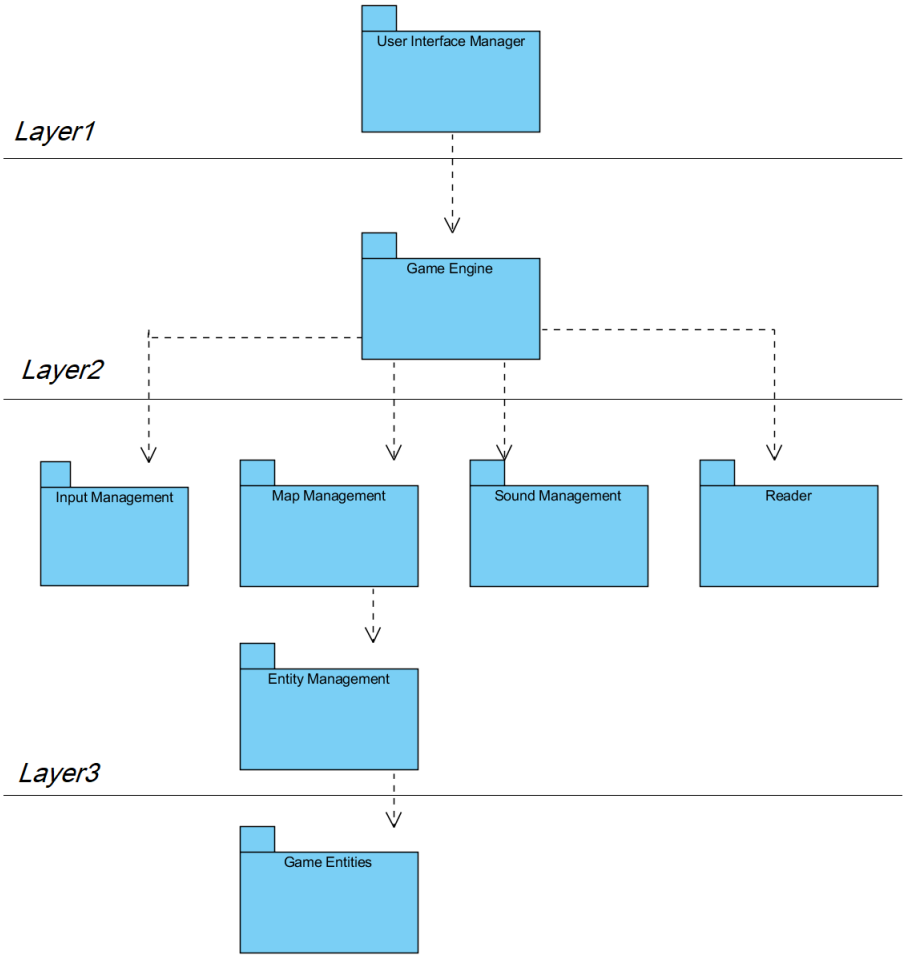
2 Software Architecture

2.1 Overview

e-Bola is designed to be programmer-friendly. For this purpose, the software system is decomposed into subsystems. Decomposition of the project will slightly increase the workload but will also improve the readability, reusability, and ease of maintenance of the project. In addition, decomposition is helpful to implement Model View Controller (MVC) architectural pattern.

2.2 Subsystem Decomposition

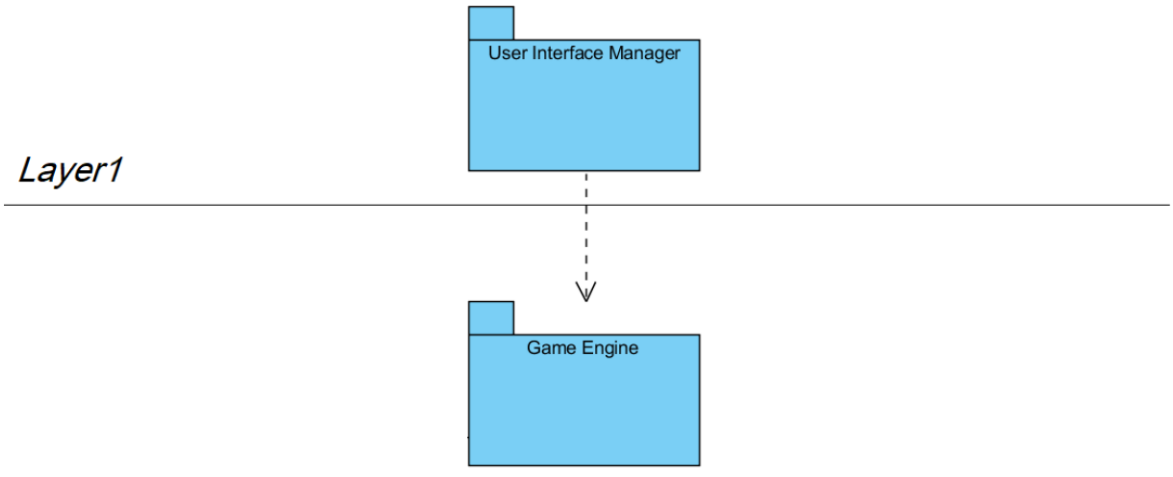
The three-tier architecture is an important design. It suggests having three tiers, one for presentation, logic, and data. The three-tier principle is also suitable for the design of e-Bola project. Presentation tier's only content is "User Interface Manager". Logic tier can be investigated under two parts: one part for "Game Engine" and one part for management units whose controller is "Game Engine". "Game Entities" is in charge of all the data as the only member of data dies. Design of e-bola is divided into three layers as shown:



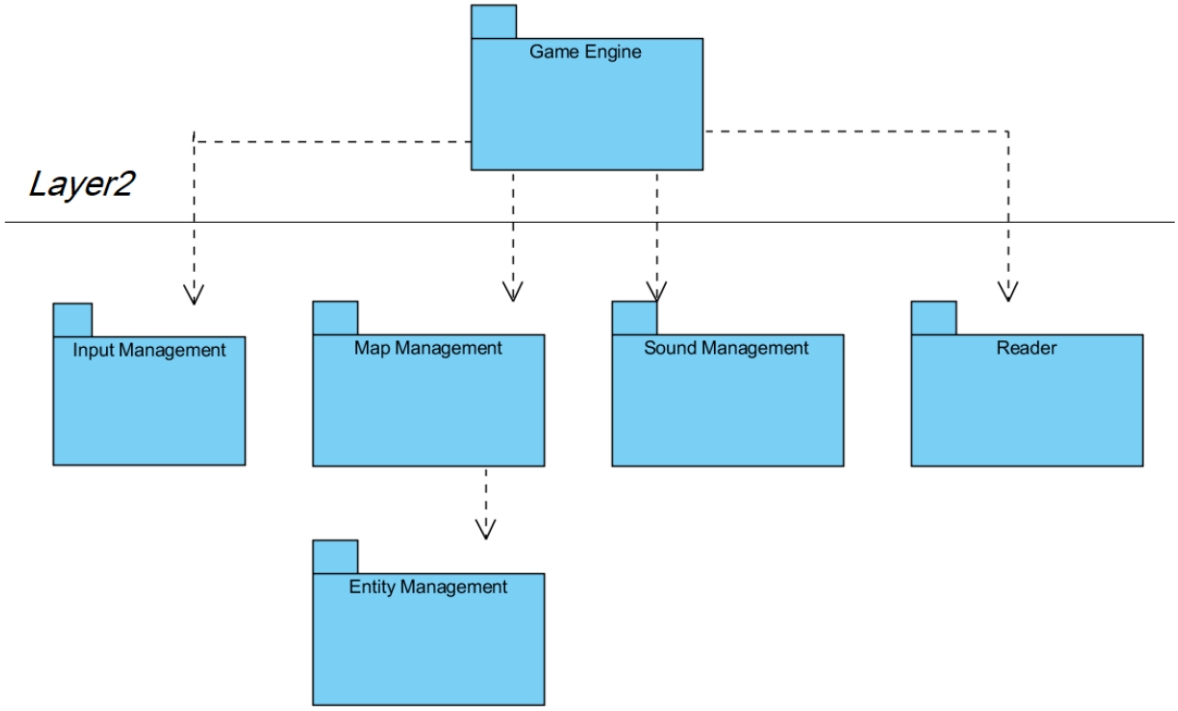
2.3 Architectural Styles

2.3.1 Layers

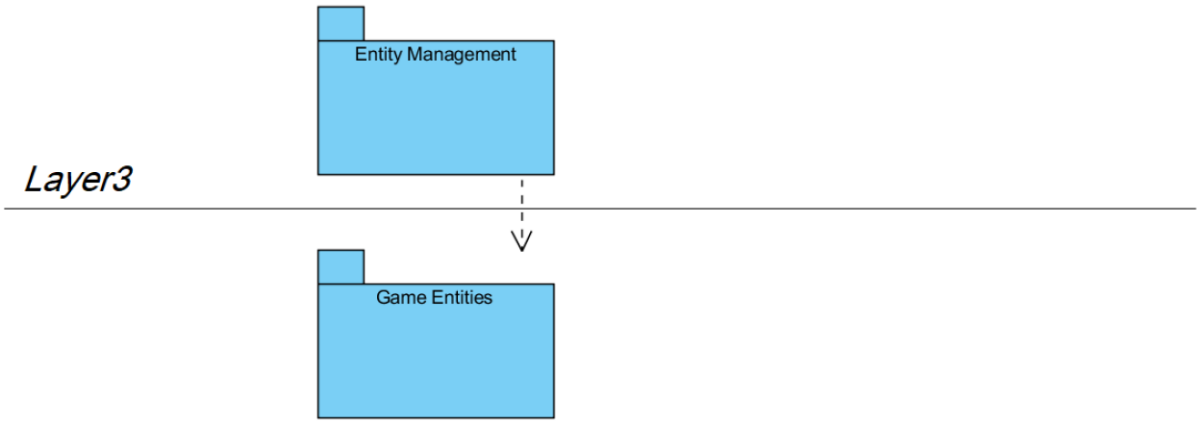
Layer1: The first level of e-Bola is the User Interface Manager and Game Engine. The User Interface Manager is responsible of how the menu is shown and how the user interacts with it. The UI Manager works synchronously with the Game Engine, and updates itself accordingly.



Layer2: The second level of e-Bola is the interaction between the Game Engine and Input Management, Map Management, Sound Management and Reader. This level is responsible of the game logic, and how all entities interact with each other.



Layer3: The third and final level of e-Bola is responsible of the Entities themselves and how they report to the Entity Manager.



2.3.2 Model View Controller

Data tier constitutes the model for our system. *User Interface Management* package represents the view for the system. Logic tier constitutes the controller of the system.

2.4 Hardware/Software Mapping

On hardware side, e-Bola requires a keyboard and a mouse for its input needs. For output, a speaker (optional) and a screen is necessary.

On software side, Java Runtime Environment (JRE) is required to run since the game will be developed using Java programming language. Java uses Java Virtual Machine (JVM). JVM can run Java code on any machine that it is installed. This language choice lets the game be independent from platforms. Therefore, software and hardware needs will be minimal. Also, no network connection is required.

2.5 Persistent Data Management

e-Bola game will store maps, audio files and entity images in the user's disk. It does not require a complex system to store the necessary data. The game instance will be saved in a text file.

2.6 Access Control and Security

e-Bola does not require any off-computer input or output, all data is stored in a local disk. Therefore, it does not have any network-related security or privacy issues. In addition to that, security constraints of the game prevent multiple instances of the game to be opened at the same time. Such restrictions assure that the user's data is safe from being changed by multiple processes.

e-Bola's user data, which is kept in text files, can be edited outside the game, and this could cause the data to be corrupted. Because of this, the file attribute will be "hidden".

2.7 Boundary Conditions

2.7.1 Initialization

The necessary data will be provided by the game, so there is no need for the user to do anything at the initialization step. e-Bola does not need to be installed and it does not need to be registered.

2.7.2 Termination

If Celly dies, the game ends automatically and starts from the last checkpoint. If the user wants to end the game temporarily, he can save the current state of the game and continue playing whenever he wants. If the user wants to start a new game, the user can close the game and restart it, and select the new game option.

3 Subsystem Services

3.1 Overview

The *User interface* package is responsible for the system's interaction with the user. It consists of "Menu Panel", which implements "Menu Button Listener" interface and has "Menu Button"s, and "Game Panel".

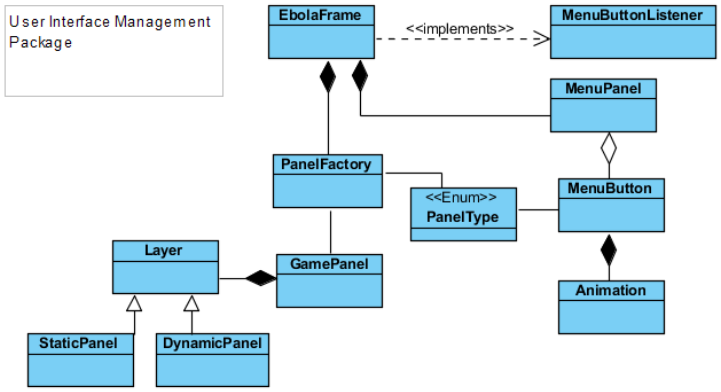


Figure 1

The *Game Engine* controls all the other managers: "Input Manager", "Map Manager", "Sound Manager", and "Reader" packages. "Map Manager" is in charge of "Entity Manager".

The *Input Manager* controls the mouse and keyboard inputs such as commands like moving left or clicking a button.

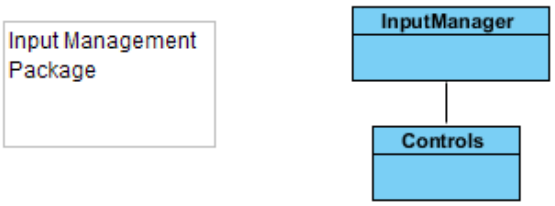


Figure 2

The *Map Manager* manages "Map"s which manages "Room"s.

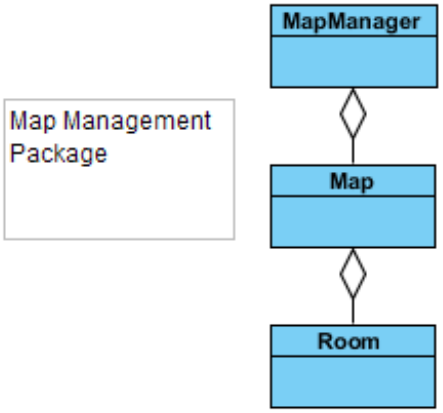


Figure 3

Sound Manager package involves “Sound Manager”. “Sound Manager” is responsible of controlling background sounds via “Background Sound” and interactable sounds such as the sound of an “Alive” entity dying or firing.

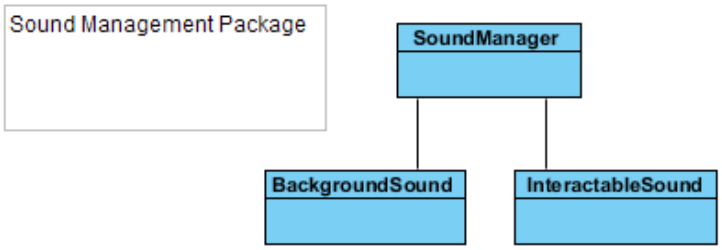


Figure 4

Reader package is responsible of reading from pre-existing files and directing needed data to “Game Engine” to create entities in the game.

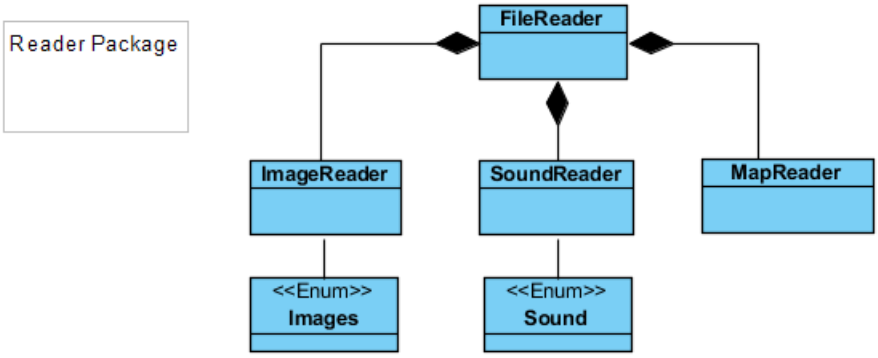


Figure 5

Entity Management has “Celly Manager” to manage Celly and “Enemy AI Manager” to manage the movement and action of Celly’s enemies. “Entity Manager” also manages “Tile Generator”. “Tile Generator” generates tiles using “Tile Factory”.

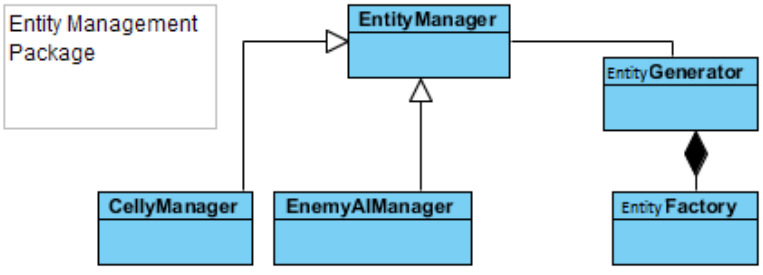


Figure 6

Game Entities package contains all of the other entities. Entities are divided into three categories: “Alive”, “Interactable” and “Tile”. “Alive” entities are

"Celly", "Virus", and the ultimate enemy "EBOLA". "Interactable" entity is "Key". "Celly" will be able to interact with "Key". "Lock"s have "Key"s, and "Portal", and "Chest" implement "Lockable" interface. "Tile" is neither alive nor interactable. "Grass", "Rock", and "Water" is a "Tile".

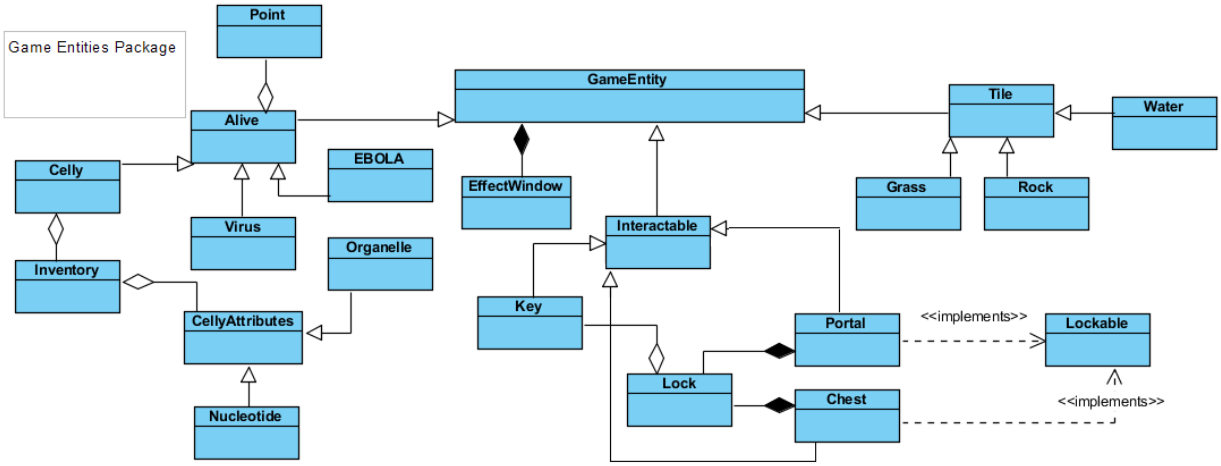


Figure 7

3.2 Design Patterns

e-Bola has a good amount of classes and this makes virtues such as maintainability a prime issue. To this end, e-Bola includes design patterns to improve the understandability, maintainability and efficiency of the code. e-Bola's main game loop runs in the class GameEngine. This class employs the "Mediator Design Pattern". As new features are added, or their addition is considered, it is important that GameEngine class does not "get out of hand". Other design patterns of e-Bola allow us to solve these kinds of problems and ease the job of GameEngine class to prevent it from becoming a "God Object".

3.2.1 Mediator Design Pattern

Mediator objects encapsulate behaviours of other classes and act as a global logic for interaction between classes. This reduces coupling between individual subsystems and makes it easier to make changes. GameEngine class acts as our Mediator. Thanks to this design pattern, we do not have a lot of couplings between subsystems that actually do not need to know about each other. If there are slight interactions, GameEngine takes care of them. For example, when Celly goes through a Portal, we need CellyManager to tell the Map to change the current room itself. Normally, we never need Map and CellyManager to have interactions. Instead of coupling CellyManager and Map for this single case, CellyManager informs GameEngine that something needs to done. GameEngine then communicates with Map to change the room. Since

CellyManager needs to tell a lot about a lot of things, it already has a connection to the GameEngine. So, adding new events is actually not hard because the connections already exist between GameEngine and popular classes. Adding new behaviour is done by adding small lines to GameEngine. A drawback of this design pattern is that as the number of connections increase, GameEngine starts to know a lot about the classes it has to connect. Going on from the same example; GameEngine has to know about Portals to understand there has been a event triggered due to change of Rooms, so it can pass the arguments to the Map object. As the class gets bigger with new interactions, these classes make it harder to read and maintain GameEngine. So, the mediator class has to be refactored frequently and checked to see if some of the functionality can be combined and delegated to some other controller class. Composite Design Patterns for a group of classes can be discovered during these refactors.

3.2.2 Factory Design Pattern

e-Bola has many model classes (classes that do not contain any logic but are created for the sake of polymorphism and inheritance to manage small changes between object more efficiently). This makes it hard to maintain change when new classes or combined behaviour is added. For example, we would have to code for the different necessities of new classes with multiple if-else conditional statements across classes. This causes us to revisit old classes and change them whenever a new model class is added. Instead of this, we encapsulate the change by moving object creation and object behaviours to different classes. Classes like EntityFactory are only responsible of entity object creations. This way, if the new class we are adding does not have unique logic, the only thing we have to do is change a line in EntityFactory class. The rest of classes are written in a way to adapt to new comer classes and work without problems.

3.2.3 Façade Design Pattern

Some controller classes in e-Bola require delegating their responsibilities to other controllers to ease their work and enhance maintainability. As the sub contollers also increase their workload, they become into individual logics rather than staying as mere helpers of the original controller. In situation like these, we discover facades by turning the main controller into a facade for the sub controllers. For example, in e-Bola we have VirusManager and CellyManager as individual logics that evaluate actions of Virus and Celly objects respectively. EntityManager class acts as a facade to the outer world (mainly to the GameEngine). EntityManager combines methods provided by VirusManager and CellyManager to create high level methods for other classes to use. This way, individual classes do not have to be coupled to VirusManager

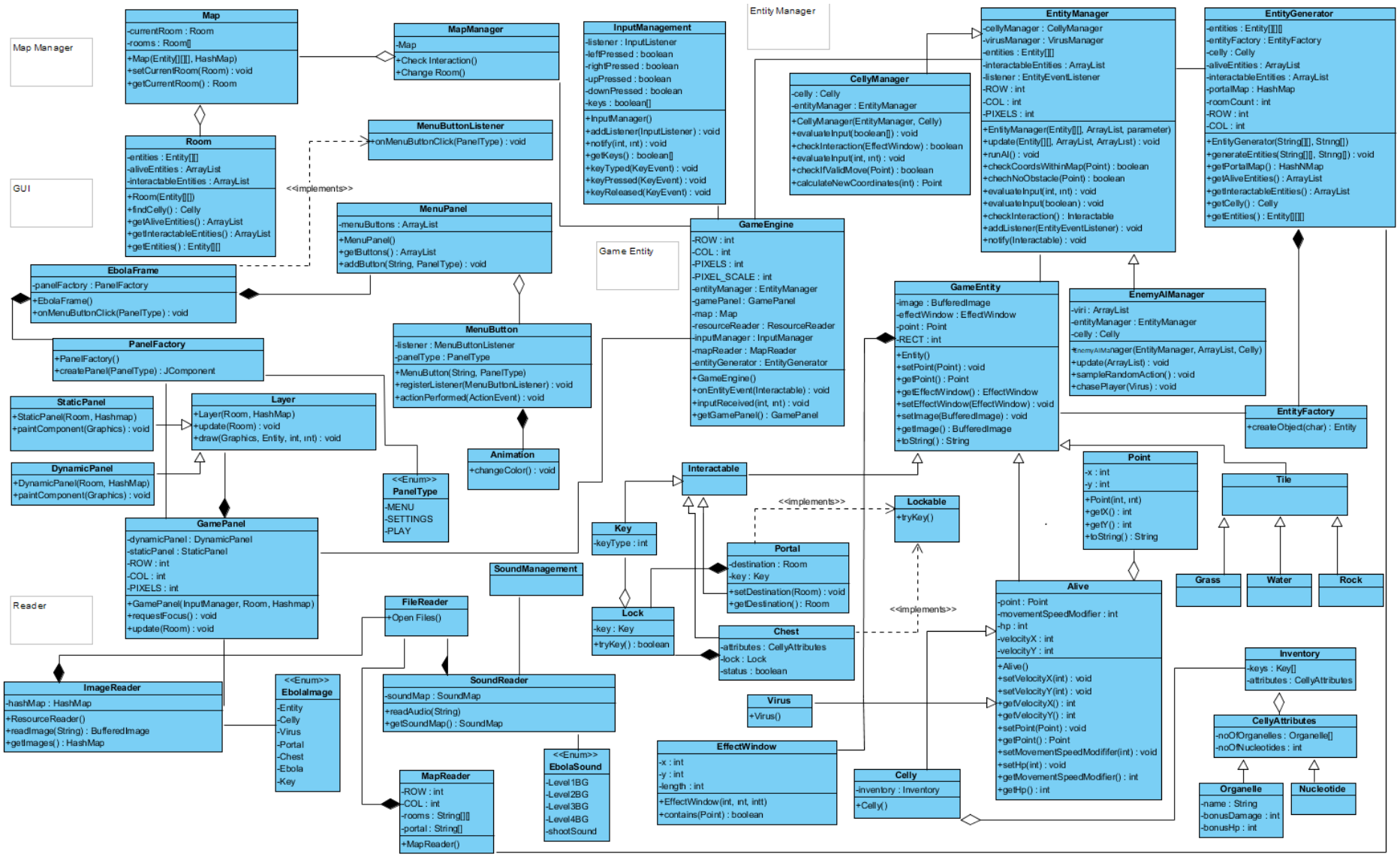
and CellyManager and these managers can change their logics and methods internally. As long as EntityManager is updated from within after the changes, the outer classes that are coupled with EntityManger will not know the difference. Creating facades is also crucial in e-Bola due to the great number of models. Subgroups of models can need individual controllers or managers. Facades allow us to insert these new controllers without changing code in other classes, since they are only coupled at EntityManager.

3.2.4 Observer Design Pattern

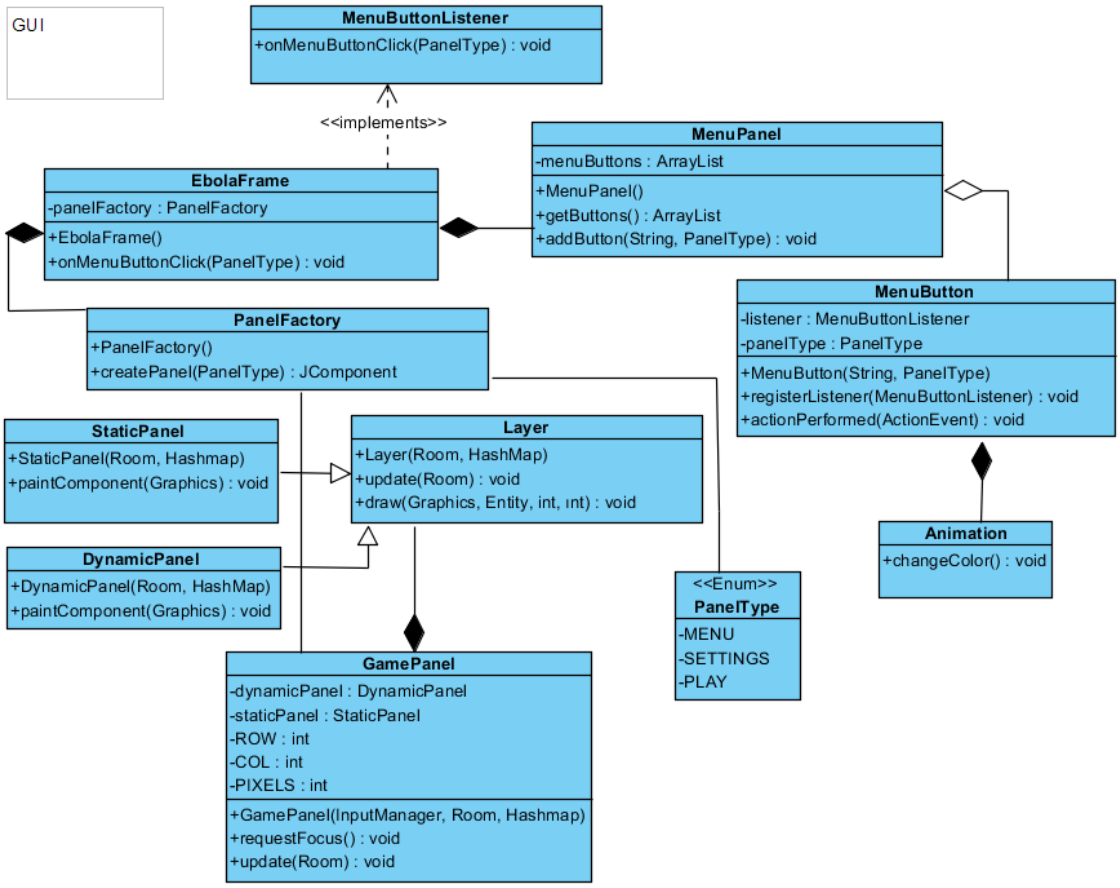
We use observer design pattern mainly to get around high couplings. In GameEngine, each subsystem notifies the GameEngine when a relevant event happens to signal the GameEngine to act. So in our case, GameEngine registers to subsystems to listen to changes caused by the models and the controllers notify their listener(s) (mainly the GameEngine) when such change in models require higher level logic.

3.3 Detailed Object Design

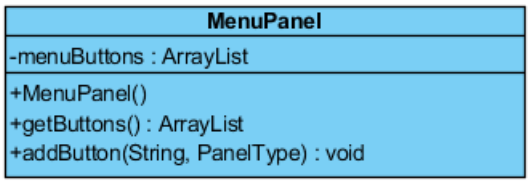
All of the classes of e-Bola, their attributes and their operations are given below. The relationships between classes are also shown in this detailed class diagram. In the rest of section 3, the classes will be explained in-depth, individually.


Map Manager
Map
-currentRoom : Room
-rooms : Room[]
+Map(Entity[][][], HashMap)
+setCurrentRoom(Room) : void
+getCurrentRoom() : Room
MapManager
-Map
+Check Interaction()
+Change Room()
InputManagement
-listener : InputListener
-leftPressed : boolean
-rightPressed : boolean
-upPressed : boolean
-downPressed : boolean
-keys : boolean[]
+InputManager()
+addListener(InputListener) : void
+notify(int, int) : void
+getKeys() : boolean[]
+keyTyped(KeyEvent) : void
+keyPressed(KeyEvent) : void
+keyReleased(KeyEvent) : void
Entity Manager
EntityManager
-cellyManager : CellyManager
-virusManager : VirusManager
-entities : Entity[][]
-interactableEntities : ArrayList
-listener : EntityEventListener
-ROW : int
-COL : int
-PIXELS : int
+EntityManager(Entity[][], ArrayList, parameter)
+update(Entity[][], ArrayList, ArrayList) : void
+runAI() : void
+checkCoordsWithinMap(Point) : boolean
+chechNoObstacle(Point) : boolean
+evaluateInput(int, int) : void
+evaluateInput(boolean) : void
+checkInteraction() : Interactable
+addListener(EntityEventListener) : void
+notify(Interactable) : void
EntityGenerator
-entities : Entity[][][]
-entityFactory : EntityFactory
-celly : Celly
-aliveEntities : ArrayList
-interactableEntities : ArrayList
-portalMap : HashMap
-roomCount : int
-ROW : int
-COL : int
+EntityGenerator(String[][], String[])
+generateEntities(String[][], String[]) : void
+getPortalMap() : HashNMap
+getAliveEntities() : ArrayList
+getInteractableEntities() : ArrayList
+getCelly() : Celly
+getEntities() : Entity[][][]
CellyManager
-celly : Celly
-entityManager : EntityManager
+CellyManager(EntityManager, Celly)
+evaluateInput(boolean[]) : void
+checkInteraction(EffectWindow) : boolean
+evaluateInput(int, int) : void
+checkIfValidMove(Point) : boolean
+calculateNewCoordinates(int) : Point
Room
-entities : Entity[][]
-aliveEntities : ArrayList
-interactableEntities : ArrayList
+Room(Entity[][])
+findCelly() : Celly
+getAliveEntities() : ArrayList
+getInteractableEntities() : ArrayList
+getEntities() : Entity[][]
MenuButtonListener
+onMenuButtonClick(PanelType) : void
<<implements>>
GUI
MenuPanel
-menuButtons : ArrayList
+MenuPanel()
+getButtons() : ArrayList
+addButton(String, PanelType) : void
GameEngine
-ROW : int
-COL : int
-PIXELS : int
-PIXEL_SCALE : int
-entityManager : EntityManager
-gamePanel : GamePanel
-map : Map
-resourceReader : ResourceReader
-inputManager : InputManager
-mapReader : MapReader
-entityGenerator : EntityGenerator
+GameEngine()
+onEntityEvent(Interactable) : void
+inputReceived(int, int) : void
+getGamePanel() : GamePanel
Game Entity
EbolaFrame
-panelFactory : PanelFactory
+EbolaFrame()
+onMenuButtonClick(PanelType) : void
MenuButton
-listener : MenuButtonListener
-panelType : PanelType
+MenuButton(String, PanelType)
+registerListener(MenuButtonListener) : void
+actionPerformed(ActionEvent) : void
GameEntity
-image : BufferedImage
-effectWindow : EffectWindow
-point : Point
-RECT : int
+Entity()
+setPoint(Point) : void
+getPoint() : Point
+getEffectWindow() : EffectWindow
+setEffectWindow(EffectWindow) : void
+setImage(BufferedImage) : void
+getImage() : BufferedImage
+toString() : String
EnemyAIManager
-viri : ArrayList
-entityManager : EntityManager
-celly : Celly
+EnemyAIManager(EntityManager, ArrayList, Celly)
+update(ArrayList) : void
+sampleRandomAction() : void
+chasePlayer(Virus) : void
PanelFactory
+PanelFactory()
+createPanel(PanelType) : JComponent
EntityFactory
+createObject(char) : Entity
StaticPanel
+StaticPanel(Room, Hashmap)
+paintComponent(Graphics) : void
Layer
+Layer(Room, HashMap)
+update(Room) : void
+draw(Graphics, Entity, int, int) : void
Animation
+changeColor() : void
DynamicPanel
+DynamicPanel(Room, HashMap)
+paintComponent(Graphics) : void
<<Enum>>
PanelType
-MENU
-SETTINGS
-PLAY
Interactable
Lockable
+tryKey()
<<implements>>
Point
-x : int
-y : int
+Point(int, int)
+getX() : int
+getY() : int
+toString() : String
Tile
Grass
Water
Rock
GamePanel
-dynamicPanel : DynamicPanel
-staticPanel : StaticPanel
-ROW : int
-COL : int
-PIXELS : int
+GamePanel(InputManager, Room, Hashmap)
+requestFocus() : void
+update(Room) : void
Key
-keyType : int
Portal
-destination : Room
-key : Key
+setDestination(Room) : void
+getDestination() : Room
SoundManagement
Reader
FileReader
+Open Files()
Lock
-key : Key
+tryKey() : boolean
Chest
-attributes : CellyAttributes
-lock : Lock
-status : boolean
Alive
-point : Point
-movementSpeedModifier : int
-hp : int
-velocityX : int
-velocityY : int
+Alive()
+setVelocityX(int) : void
+setVelocityY(int) : void
+getVelocityX() : int
+getVelocityY() : int
+setPoint(Point) : void
+getPoint() : Point
+setMovementSpeedModifier(int) : void
+setHp(int) : void
+getMovementSpeedModifier() : int
+getHp() : int
Inventory
-keys : Key[]
-attributes : CellyAttributes
ImageReader
-hashMap : HashMap
+ResourceReader()
+readImage(String) : BufferedImage
+getImages() : HashMap
<<Enum>>
EbolaImage
-Entity
-Celly
-Virus
-Portal
-Chest
-Ebola
-Key
SoundReader
-soundMap : SoundMap
+readAudio(String)
+getSoundMap() : SoundMap
Virus
+Virus()
CellyAttributes
-noOfOrganelles : Organelle[]
-noOfNucleotides : int
EffectWindow
-x : int
-y : int
-length : int
+EffectWindow(int, int, intt)
+contains(Point) : boolean
Celly
-inventory : Inventory
+Celly()
MapReader
-ROW : int
-COL : int
-rooms : String[][]
-portal : String[]
+MapReader()
<<Enum>>
EbolaSound
-Level1BG
-Level2BG
-Level3BG
-Level4BG
-shootSound
Organelle
-name : String
-bonusDamage : int
-bonusHp : int
Nucleotide

3.4 User Interface Management Subsystem



3.4.1 Menu Panel



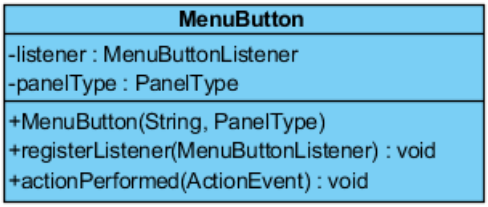
MenuPanel is the panel that welcomes the user when the game launches. It contains buttons, an instantiation of the MenuButton class, which are used to navigate to other panels (e.g. game settings panel). The structure of User Interface package makes adding new buttons easier with the MenuButton class and MenuButtonListener interface.

Constructor(s):
-MenuPanel() : MenuPanel = Sets panel properties of the object such as dimensions, background and layout. Calls addButton method for each button of the menu.

Instance method(s):
-addButton(String, PanelType) : void = Adds a MenuButton object to the MenuPanel. It also adds empty boxes to the view to have enough empty space between buttons.

-getButtons : ArrayList<MenuButton> = Returns all the MenuButtons added to the MenuPanel

3.4.2 Menu Button



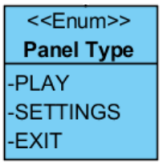
MenuButton class is a child of Java JButton class. It has methods to make the Ebola Frame listen to any instance of MenuButton and also holds values from Panel Type enumeration to help Ebola Frame understand which panel to create when the Menu Button is pressed.

Constructor(s):

-MenuButton(String, PanelType) : MenuButton = Adds an action listener to the button to understand clicks.

Instance method(s):
-registerListener(MenuButtonListener) : void = Registers a MenuButtonListener so it can notify it later.
-actionPerformed(ActionEvent) : void = Delegates the click event to the listener so it can implement its own logic of what to do.

3.4.3 Panel Type

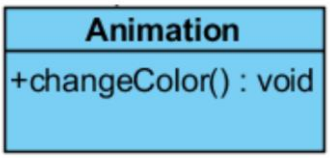


Panel Type enumeration holds the values that correspond to the panels that can be created.

3.4.4 Menu Button Listener

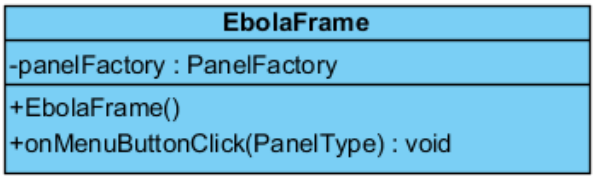
Menu Button Listener interface forms a contract between Ebola Frame class and the Menu Button objects. This way, Ebola Frame knows which buttons is pressed, when it is pressed, and which panel should be created.

3.4.5 Animation



Animation class is used by MenuButton objects to make the interaction with the user more responsive.

3.4.6 Ebola Frame



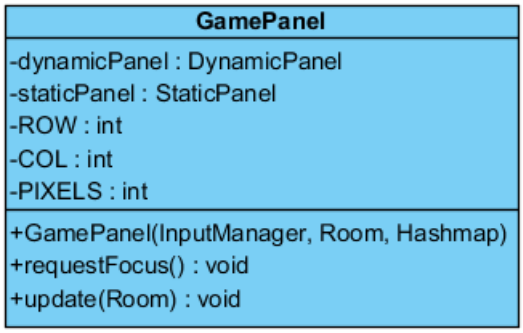
EbolaFrame class holds the main application frame. It implements the MenuButtonListener interface and listens to MenuButton clicks. When there is a MenuButton click, EbolaFrame asks the Panel Factory to create the appropriate panel, using the PanelType enumeration.

Constructor(s):

-EbolaFrame() : EbolaFrame = Creates a PanelFactory and MenuPanel. Collects all the MenuButtons from MenuPanel and registers to them. Sets panel properties of the object such as dimensions and layout.

Instance method(s):
-onMenuButtonClick(PanelType) : void = Creates a panel using a PanelFactory object depending on the PanelType parameter.

3.4.7 Game Panel



GamePanel is the panel where the actual gameplay is shown. When the user clicks the "Play" button, GamePanel is created along with Game Engine. Game Panel uses ImageReader class from the Reader package to obtain a hashmap. Game Panel retrieves information about the game status from the GameEngine via GameEntity objects. Game Entity objects are then painted to the screen using this map.

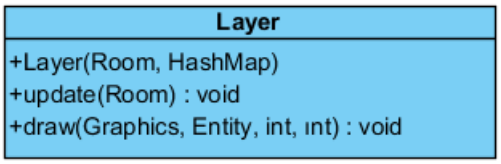
Constructor(s):

-GamePanel() : GamePanel = Sets panel properties of the object such as dimensions and layout. Creates DynamicPanel and StaticPanel instances and puts them on a JLayeredPane accordingly.

Instance method(s):
-update() : void = updates the views of the dynamic and static panels.

3.4.8 Layer

Layer is an abstraction for DynamicPanel and StaticPanel classes. It denotes a layer in a JLayeredPane(our GamePanel).

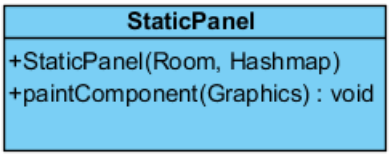


Constructor(s):
-Layer(Room, HashMap) : Layer = Initializes the variables.

Instance method(s):
-update(Room) : void = updates the room variable of type Room. Repaints the layer.
-draw(Graphics, Entity, int, int) : void = Draws the entity object on the screen using the inputs. Gets what image to draw depending on the class name of the entity by using the Images enumeration.

3.4.9 Static Panel

StaticPanel is a Layer on the GamePanel that is responsible with painting the objects that do not move(such as tiles or obstacles).



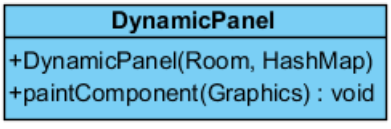
StaticPanel is a Layer on the GamePanel that is responsible with painting the objects that do not move(such as tiles or obstacles).

Constructor(s):
-StaticPanel(Room, HashMap) : StaticPanel= Initializes the variables.

Instance method(s):
-paintComponent(Graphics) : void = Uses entity array obtained by using the getEntities() method on the current room object to paint the objects to the screen.

3.4.10 Dynamic Panel

DynamicPanel is a Layer on the GamePanel that is responsible with painting the objects that are on top of static elements (such as Interactables or Alives)

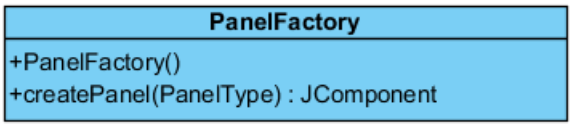


Constructor(s):
-DynamicPanel(Room, HashMap) : DynamicPanel= Initializes the variables.

Instance method(s):
-paintComponent(Graphics) : void = Uses arraylists obtained by using the getAliveEntities() and getInteractableEntities() methods on the current room object to paint the objects to the screen.

3.4.11 Panel Factory

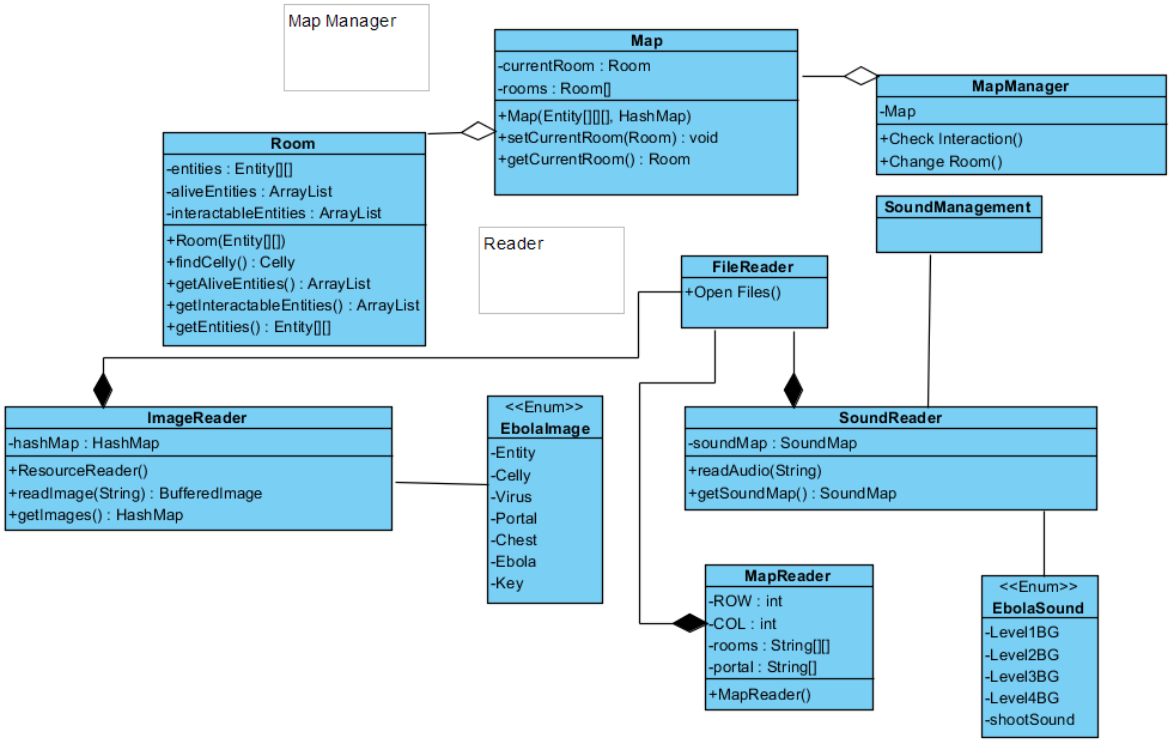
PanelFactory is responsible of creating different panels for the MenuPanel, so that each button goes to a different panel.



Constructor(s):
-Default Constructor

Instance method(s):
-createPanel(PanelType) : JComponent = Creates a panel object depending on the panelType input and returns it.

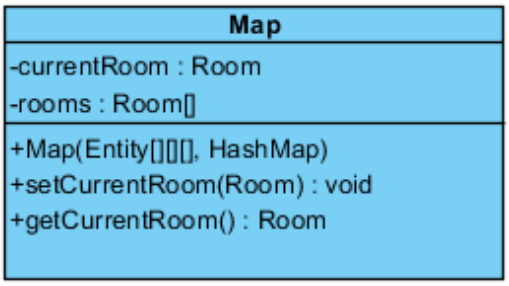
3.5 Game Management Subsystem Interface



Map Management Package

Map Management Package consists of classes that are related to the objects in a game level and their internal statuses.

3.5.1 Map

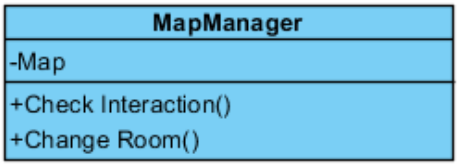


Map class contains the Room objects of the current game level. One of the rooms in the map will be the current room in the game while all others will stay dormant until the player proceeds to another room. The current room will be the room that the player can interact with, so other room objects will not be manipulated.

Constructor(s):
-Map(Entity[][][], HashMap<Portal, Integer>) : Map = Creates Room objects by parsing the input Entity array. Sets the current room variable to the first room. Sets the destination of Portal objects using the input HashMap.

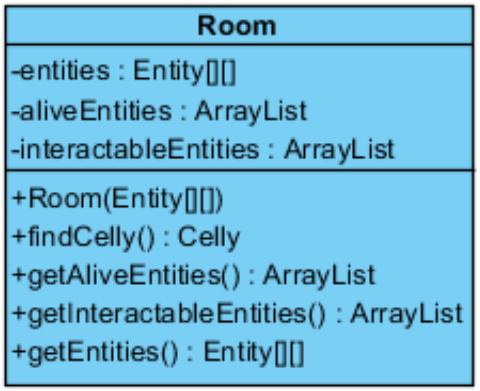
Instance method(s):
-setCurrentRoom(Room) : void = Changes the current room. Also changes Celly object's location by removing it from the current Room's alive entity arraylist and adding it to the alive entity arraylist of the new current room object.
-getCurrentRoom() : Room = Returns the current room.

3.5.2 Map Manager



Map Manager is used to hold all the information about the current Map in the game. Depending on the current state of the game, the Map Manager will check if there are any interactions with any Portal objects. In the presence of an interaction, it will change the currentRoom variable to a different Room object.

3.5.3 Room



Room class contains the Game Entity objects as a two dimensional array. Dimensions of this array correspond to the rows and columns of the Game Panel. Room also contains the Portal objects that link rooms to other rooms.

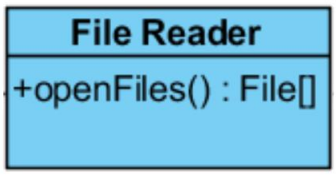
Constructor(s):
-Room(Entity[][]) : Room = Parses the entity array and puts certain abstractions for entities in respective arraylists. e.g an Alive object is put to the arraylist aliveEntities : ArrayList<Alive>.

Instance method(s):
-findCelly() : Celly = Returns the Celly object after searching the aliveEntities arraylist
-getAliveEntities() : ArrayList<Alive> = Returns the aliveEntities arraylist.
-getInteractableEntities() : ArrayList<Interactable> = Returns the interactableEntities arraylist.
-getEntities() : Entities[][] = Returns the entities array.

Reader Package

Reader package is responsible of reading all game related data into memory. Advantages of this is further explained in the trade-offs section (1.4).

3.5.4 File Reader

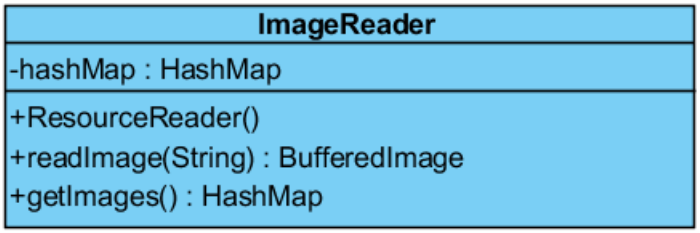


All readers use the File Reader class for their reading processes. The main aim of the File Reader class is to locate the game files and verify their integrity. If there are no problems, the files are opened and sent to the respective readers for the actual reading process.

Constructor(s):
-Default Constructor

Instance method(s):
-openFiles() : File[] = Opens the game data and load all the files into the game system.

3.5.5 Image Reader



The Image Reader reads all image files into the memory. These images are than mapped to Game Entity objects. At runtime, this mapping will be used by the Game Panel to paint images corresponding to the objects that are present at that current screen. The mapping is done between EbolaImage and BufferedImage. EbolaImage is an enumeration whose values

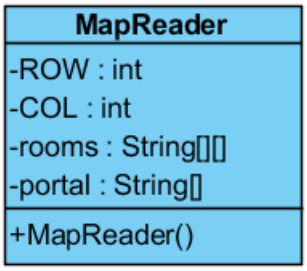
correspond to the image file names and names of Game Entity classes. BufferedImage is a part of Java awt package.

Constructor(s):

-ResourceReader() : ResourceReader = use the method to match the images with the corresponding image name and put them in a hashmap.

Instance method(s):
-readImage(String) : BufferedImage = Load individual images into the system.
-getImages() : HashMap = return the hashmap that contains Images.

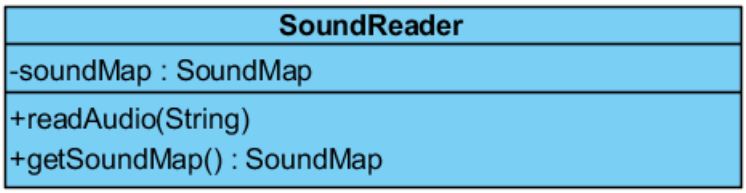
3.5.6 Map Reader



The Map Reader reads the text files that correspond to the Room objects at runtime. Since these text files contain multiple attributes about the room, it is important to differentiate between information and send them to object creation in an orderly manner. The way the files are read will determine the connection between the Portal objects. The Entity Generator will use this information to create Room objects and their connection with the other Portal Objects.

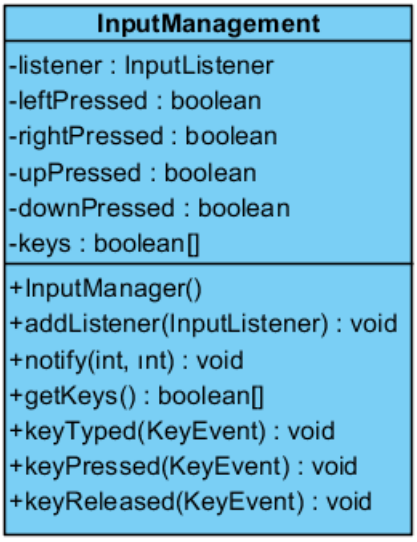
Constructor(s):
-MapReader() : MapReader = Parses the map file contents and turns them into arrays for further processing.

3.5.7 Sound Reader



The Sound Reader class is very similar to the Image Reader class, but is related to the in-game sounds. The Sound Reader reads all of the sound files into the memory. These sounds are than mapped to Room objects, or actions of the Alive objects. At runtime, this mapping will be used by the Sound Manager to play sounds depending on the atmosphere, room or action. The mapping is done between EbolaSound and AudioInputStream. EbolaSound is an enumeration whose values correspond to the sound file names. AudioInputStream is a part of Java sound package.

3.5.8 Input Management



Input Management package is responsible of reading the user input and notifying the listeners with the input keys.

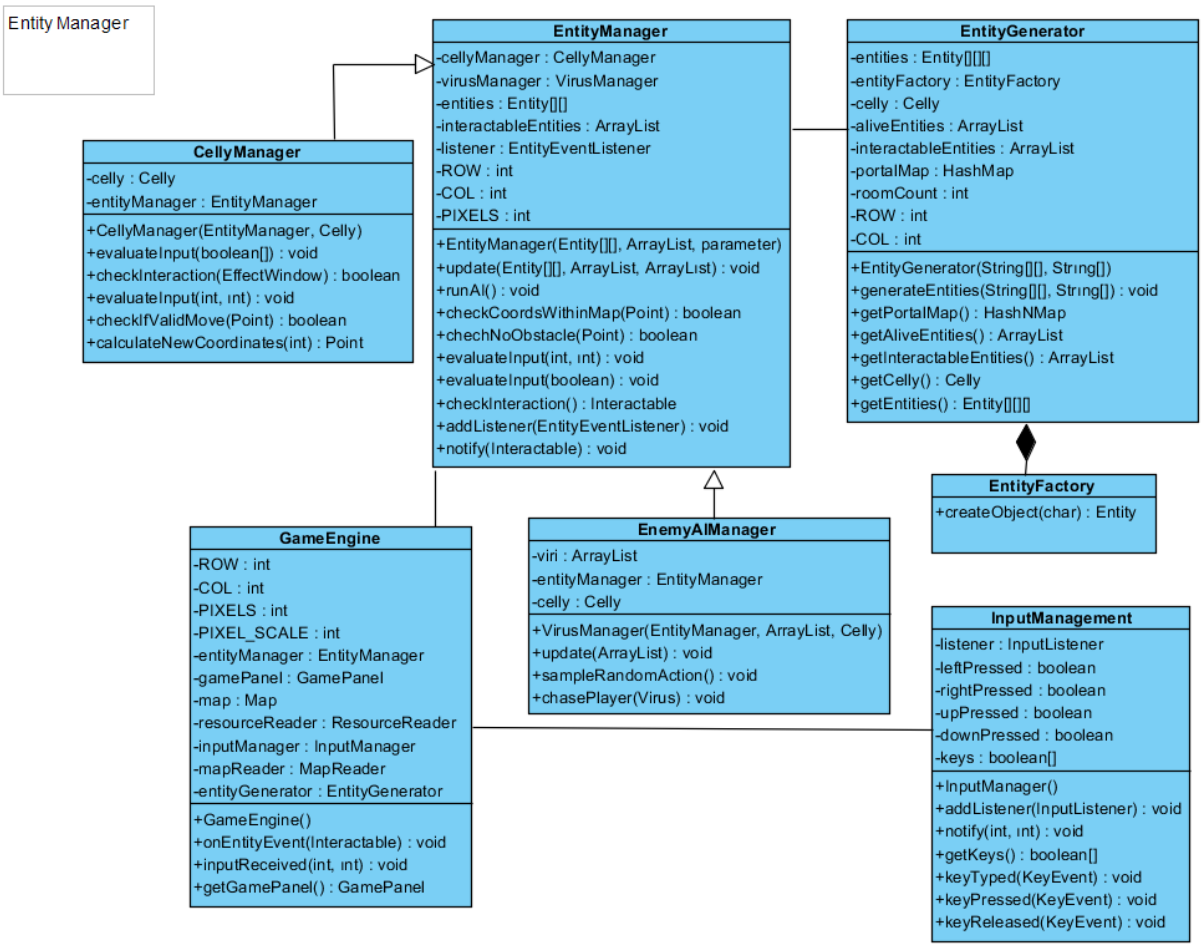
Constructor(s):

-InputManager() : InputManager = Initialize variables

Instance method(s):

-addListener(InputListener) : void = Add listeners.
-notify(int, int) : void = Notify listeners.
-keyTyped(KeyEvent) : void = Required method of Java Keylistener interface. Empty implementation.
-keyPressed(KeyEvent) : void = Required method of Java Keylistener interface. Set one of the direciton keys(up, down, right, left) to true in a boolean array depending on the KeyEvent (user input)
-keyReleased(KeyEvent) : void = Required method of Java Keylistener interface. Empty implementation. Set one of the direciton keys(up, down, right, left) to true in a boolean array depending on the KeyEvent (user input)

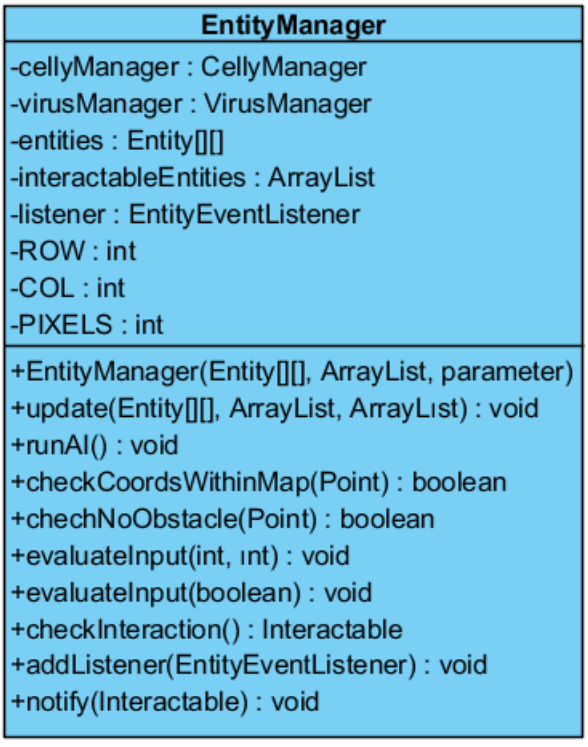
Entity Management Package



Entity management package is used to create and maintain the model objects of the game, namely the classes in the Game Entity Package.

3.5.9 Entity Manager

EntityManager is the core of the GameEntity object lifecycle management. It asks the Entity Generator to create objects whenever GameEngine needs them. It then orders this object data and creates its children, CellyManager and Enemy AI Manager, and passes the relevant data to them. Whenever the



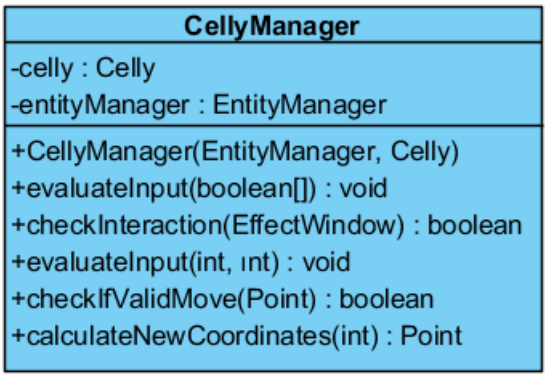
GameEngine updates the game status, EntityManager checks the new status and gives appropriate actions to CellyManager and Enemy AI Manager. Since changes in the status of entities can often conflict with or concern other entity statuses, EntityManager, CellyManager and Enemy AI Manager constantly share data to make sure all actions are legal within the constraints of game logic.

Constructor(s):

-EntityManager(Entity[][], Arraylist<Alive>, Arraylist<Interactable>) : EntityManager = Uses the individual arraylists to create instances of CellyManager and VirusManager

Instance method(s):
-update(Arraylist<Alive>, Arraylist<Interactable>) : void = Update the managers with new arraylists.
-runAI() : void = Ask the VirusManager to sampleRandomAction() for each virus.
-checkCoordsWithinMap() : boolean = Returns true if the newly calculated Point object is within the bounds of the map.
-checkNoObstacle : boolean = Returns true if the newly calculated Point object is not within the range of an obstacle.
-evaluateInput(boolean[]) : void = Ask the CellyManager to evaluate the inputs given via keyboard by the user.
-checkInteraction() : Interactable = Return the interactable object if there is an interaction. Notify the game engine so that it that appropriate action.
-notify(Interactable) : void = Notify listeners.

3.5.10 Celly Manager



Celly Manager is a class that manipulates the Celly object depending on the environmental input (tiles, interactables, and viruses) and user input (movement keys). Certain game statuses can cause Celly to interact with objects. The User input allows Celly to move. AI of viruses can create situations where Celly gets updated without any user input (e.g. losing health points after being attacked by viruses).

Constructor(s):

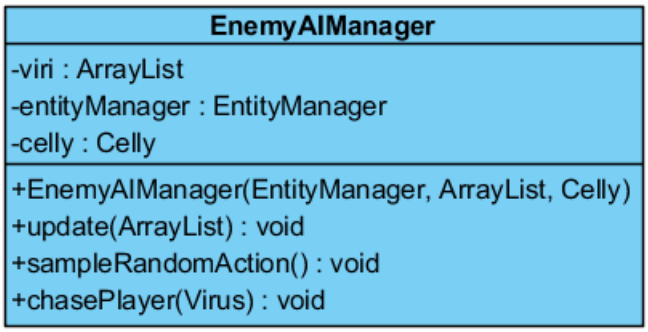
-CellyManager(EntityManager, Celly) : CellyManager = Initialize variables.

Instance method(s):

-evaluateInput(boolean[]) : void = Set velocity of Celly depending on the inputs user gave.
-checkInteraction(EffectWindow) : boolean = Returns true if Celly is within the effect window of an entity.
-checkIfValidMove(Point): boolean = Returns true if the newly calculated Point object is valid(see EntityManager methods on this issue).

3.5.11 Enemy AI Manager

Enemy AI Manager is a class that determines what the computer controlled viruses (Virus objects) will do in the current status of the game and manipulates the Virus objects accordingly.



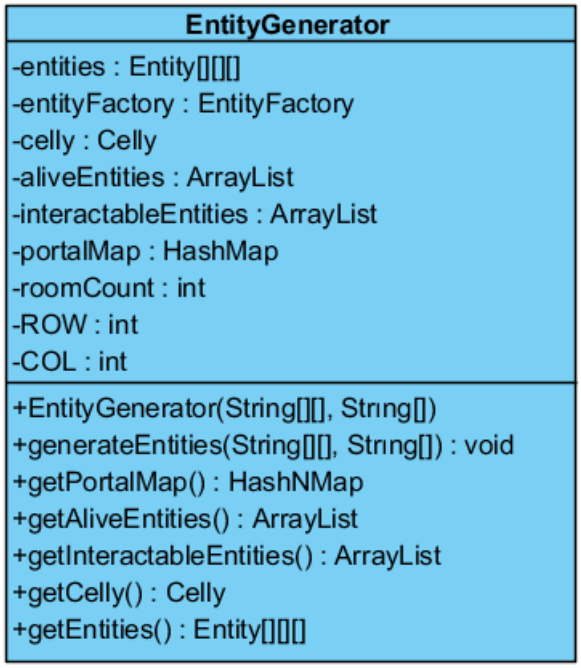
Constructor(s):

-EnemyAIManager(EntityManager, ArrayList<Virus>, Celly) : EnemyAIManager = Initialize variables.

Instance method(s):
-update() : void = Update virus arraylist.
-sampleRandomAction() : void = Do a random action for each virus. Chose one of the other methods in the class.
-chasePlayer(Virus) : void = A random action. Use the coordinates of Celly to move towards that location.

3.5.12 Entity Generator

EntityGenerator creates the objects and sends them to EntityManager for ordering.



Constructor(s):

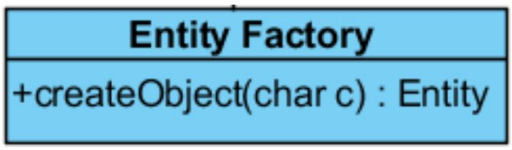
-EntityGenerator(String[][], String[]) : EntityGenerator = Initialize variables. Call generateEntities().

Instance method(s):
-generateEntities(String[][], String[]) : void = Parse the text array and pass each individual char to the

EntityFactory for object creation. Create a hashmap that connects Portals with Rooms.

-Getters and Setters for the instance variables.

3.5.13 Entity Factory

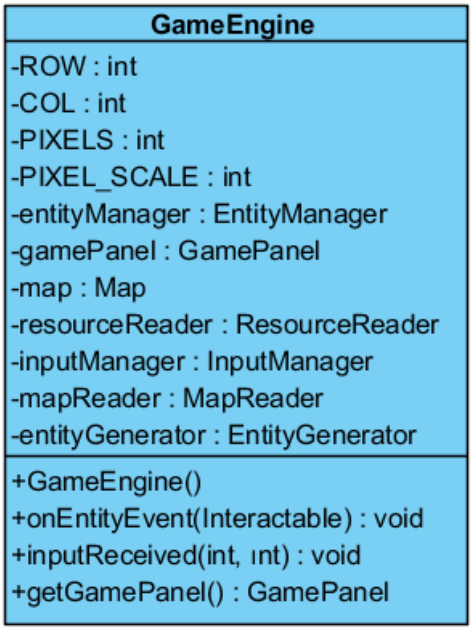


EntityFactory creates objects using the data provided by the MapReader class (from Reader package).

Constructor(s):
-Default Constructor

Instance method(s):
-createObject(char) : Entity= Creates an Entity object depending on the char input and returns it.

3.5.14 Game Engine



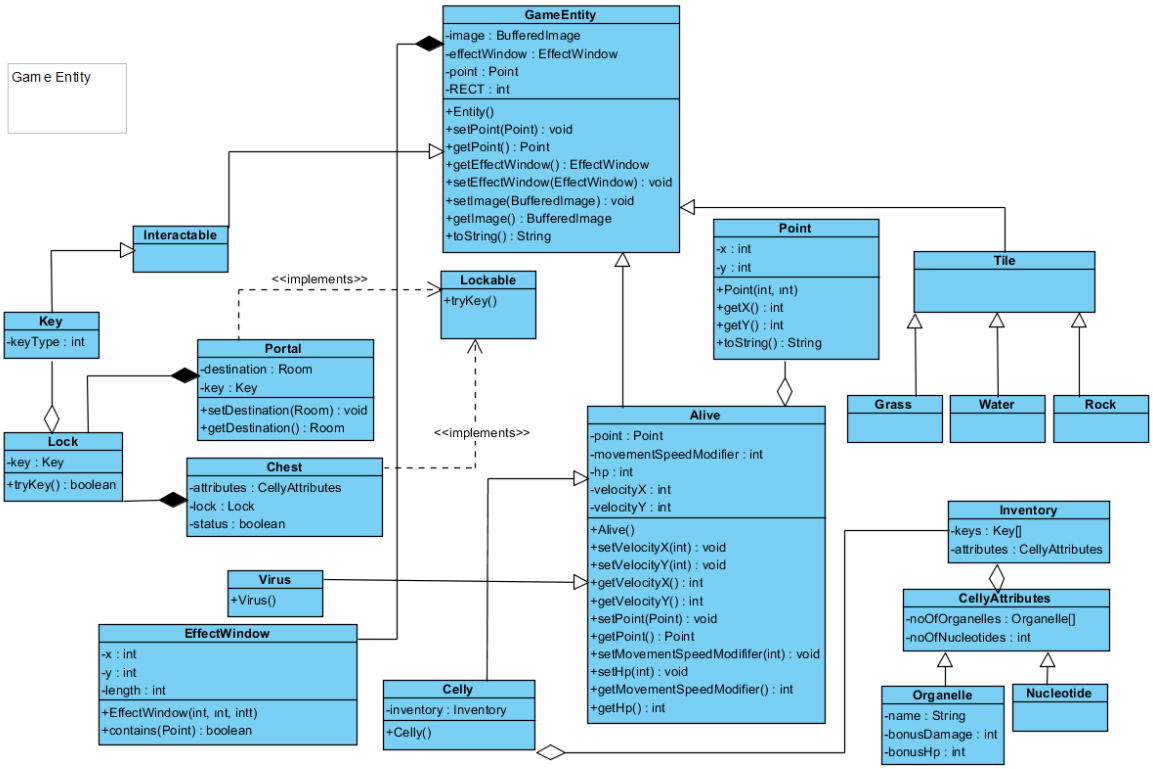
Game Engine is the core of e-Bola game. It connects all the managers across all packages and transmits data among all of them. The main game loop runs within this class and on every iteration, the updated status gets evaluated. The result of this evaluation determines which managers to signal to.

Constructor(s):
-GameEngine() : GameEngine = Create all startup and manager objects. Start up objects consist of Readers and EntityGenerator. Connect outputs of Readers to EntityGenerator. Connect outputs of EntityGenerator to create the Map of the game. Create EntityManager with Entities created by EntityGenerator. Start listening to user input via InputManager and create the GamePanel to show the created entity objects. Start the main game loop that will update the GamePanel and asks the EntityManger to evaulate the statuses of the entities.

Instance method(s):
-onEntityEvent(Interactable) : void = Consists of conditionals that decide what should be done when EntityManger notifies the GameEngine. The conditionals should consist of events that are invoked by entities and should be handled by

other managers. For starters, Interactions with Portals should be delegated to the Map object to change the current room and update the game state.
-getGamePanel() : GamePanel = Return the current GamePanel object.

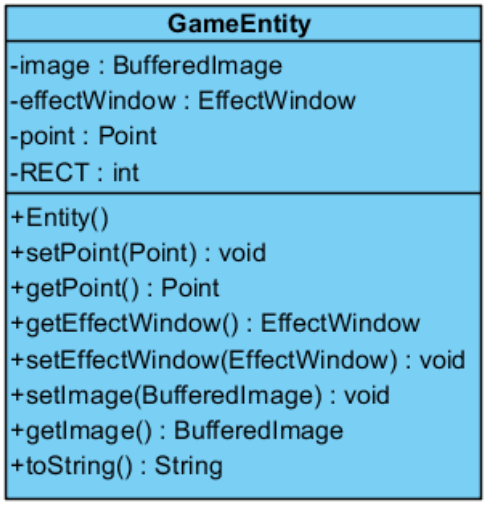
3.6 Game Entities Subsystem Interface



GameEntity Package contains all the entities in the game that are used by other classes. Classes in this package act as models. These models are used to create objects or manipulated depending on the evaluations of manager classes. Most of the model classes that are in the game are located in this package, except some classes that need high coupling with their managers (e.g. Map & Room models with MapManager).

3.6.1 Game Entity

GameEntity class is the main abstraction of all models; it is used to pass all the GameEntity objects in a single polymorphic relationship. It is also conveys the meaning that all GameEntity objects (that are

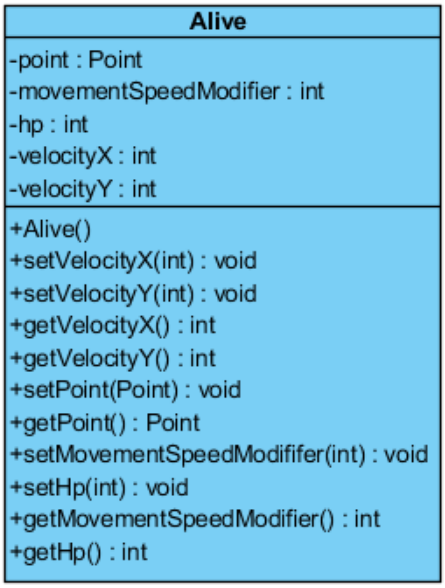


descendants of this class) can be drawn into the GamePanel.

Constructor(s):
-Default Constructor

Instance method(s):
-Getters and Setters for the instance variables.

3.6.2 Alive

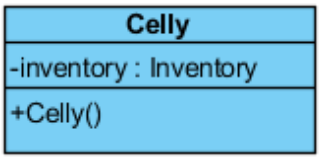


Alive is the abstraction for “moving and mortal” objects of the game. Descendants of Alive have health points that allow them to maintain their presence in the game. They have a certain place in the coordinate system (determined by the Point class) of the GamePanel and they can change their position and the speed at which they move to a new position.

Constructor(s):
-Default Constructor

Instance method(s):
-Getters and Setters for the instance variables.

3.6.3 Celly

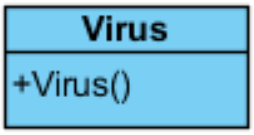


Celly is a concrete entity which represents the cell the user plays as. The distinction between Celly and other concrete Alive entities is that Celly has an Inventory object and can hold a Key object. Also, Celly is a special object that has its own manipulator (CellyManager class) that uses the inputs given by the user.

Constructor(s):
-Default Constructor

Instance method(s):
-Getters and Setters for the instance variables.

3.6.4 Virus



Virus is a concrete entity which represents the enemies of the game. The distinction between Virus and other concrete Alive entities is that it is controlled by an AI in its actions. Virus class has its own manipulator (Enemy AI Manager) to decide on these actions.

Constructor(s):
-Default Constructor

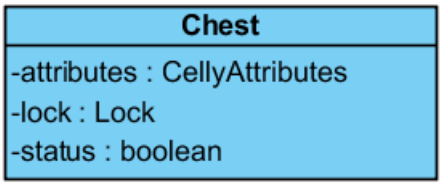
Instance method(s):
-Getters and Setters for the instance variables.

3.6.5 Interactable

Interactable is an abstraction for objects that "do something when triggered". Descendants of Interactable, trigger the GameEngine when Alive entities come too close (in terms of location in the coordinate system [again determined by Point object that Alive entities have]). GameEngine then asks the appropriate managers to manipulate game state.

Constructor(s):
-Default Constructor

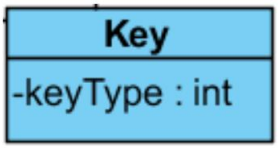
3.6.6 Chest



Chest is a concrete entity which represents chests that can be opened by by the user using a fitting Key object. Chest objects contain CellyAttribute objects that the user can take.

Constructor(s):
-Chest(Lock) : Chest = Returns a chest that is locked by the input Lock object.

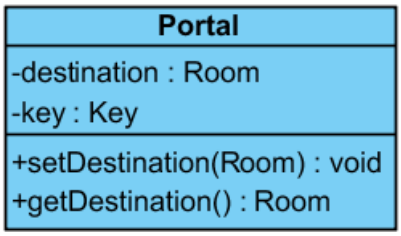
3.6.7 Key



Key is a concrete entity that the user collects to open locks.

Constructor(s):
-Default Constructor

3.6.8 Portal



Portal is a concrete entity that is used to switch between rooms of the map. Each Portal object connects to another Portal object that is contained in another Room object within the same level. Connection between Portal objects are determined by the EntityGenerator, which uses room text files as input.

Constructor(s):
-Portal() : Portal = Creates a portal that has no lock
-Portal(Lock) : Portal = Creates a portal that has the input lock as a lock.

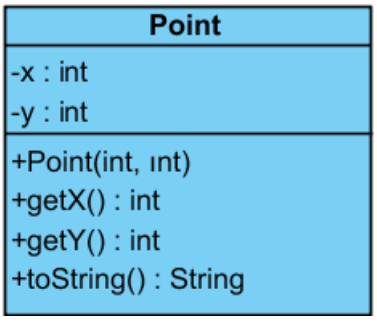
Instance method(s):
-Getters and Setters for the instance variables.

3.6.9 Tile

Tile is an abstraction for objects that represent the floor of the game board, where “Alive entities walk on”. Descendent of Tile class have different values for modifiers and different images that are assigned to them during painting of the GamePanel. Modifiers can change the movement speed of Alive entities depending on the tile they are currently contained within.

Constructor(s):
-Default Constructor

3.6.10 Point

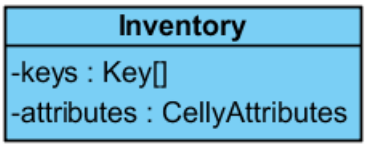


Point class represent the current coordinates of Alive entity objects.

Constructor(s):
-Point(int, int) : Point = Creates a Point object using its two inputs.

Instance method(s):
-Getters and Setters for the instance variables.

3.6.11 Inventory

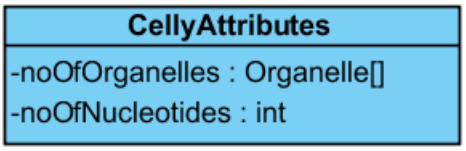


Inventory class holds objects of the CellyAttribute class. It represents a limited amount of CellyAttribute objects Celly can hold.

Constructor(s):
-Default Constructor

Instance method(s):
-Getters and Setters for the instance variables.

3.6.12 Celly Attribute

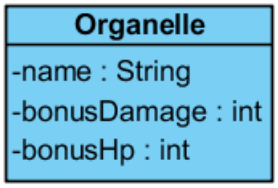


CellyAttribute class is a polymorphic abstraction to objects Celly can have to increase its attributes.

Constructor(s):
-Default Constructor

Instance method(s):
-Getters and Setters for the instance variables.

3.6.13 Organelle

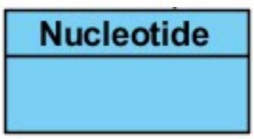


Organelle class represents objects with certain attributes that can be held in the Inventory of Celly.

Constructor(s):
-Default Constructor

Instance method(s):
-Getters and Setters for the instance variables.

3.6.14 Nucleotide

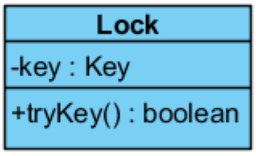


Nucleotide class represents the main currency in game.

Constructor(s):
-Default Constructor

Instance method(s):
-Getters and Setters for the instance variables.

3.6.15 Lock

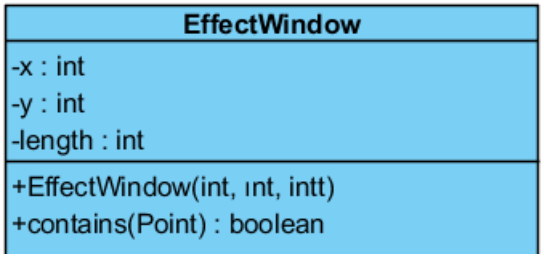


Lock disables to ability to interact with Interactable objects as long as it is present. Certain Interactables always have a lock (like Chests) while other may or may not (Portals). Locks have unique Key objects that can disable them. Celly has to find the appropriate key to get rid of the lock to access the contents of a Chest or go through Portals that have a lock.

Constructor(s):
-Lock(Key) : Lock

Instance method(s):
-tryKey() : boolean = Returns true if the given key is a match and can unlock the Lock.

3.6.16 EffectWindow



EffectWindow denotes the range that the entity performs an action. If Celly is within this range, the actions are performed.

Constructor(s):
-EffectWindow(int, int, int) : Initialize variables.

Instance method(s):
-contains(Point) : boolean = Checks if the given point is within the EffectWindow.

4. Improvement Summary

- Included a section on Design Patterns as section 3.2.
- Included newly discovered classes to sections 3.3 - 3.6
- Added explanations for the constructors and instance methods of classes to sections 3.3 - 3.6
- Updated pictures of classes in sections 3.3 - 3.6 to be more detailed by including method signatures and return types.
- Updated pictures and explanations of section 3.1