CS 319 - Object-Oriented Software Engineering

System Design Report

Bombplan

Group 1

Asena Rana Yozgatlı	*21000132*
Berk Yurttaş	*21200581*
Mehmet Furkan Şahin	*21201385*
Saner Turhaner	*21100475*

Deadline: 09/04/2016

Course Instructor: Uğur Doğrusöz

Table of Contents

Table of Figures

1. Introduction

1.1 Purpose of the System

Bombplan is a new version of an existing game called Bomberman. Our aim is making a game such that players from all age groups can enjoy. In this manner, we will design our system to be neither easy nor hard.

In our system, there are two additional bonuses that player can take. The goal of our system is that player eliminates all monsters in the maze in a given time. After player kills all monsters by planting bombs, s/he finds the door which is hidden behind a random wall. Passing the door ends the level successfully. The main purpose is being successful at each level.

1.2 Design Goals

1.2.1 Reliability

We aim to make our system reliable. In order to achieve this, the system should not crash or give an error in runtime. The system is going to be designed such a way that it will not accept any wrong input.

1.2.2 Modifiability

Our system should be modifiable. An update should be implemented by developer with no confliction. We will make our system modifiable by object oriented design. With help of object oriented design, new classes and objects will be added to existing code properly.

1.2.3 Adaptability

Bombplan should be played in different environments. So we will develop our system using Java. The only requirement to play the game is having Java Runtime Environment.

1.2.4 Portability

The system can be deployed to different development environments. We will use Java Archives files to achieve portability so building executable programs from source code will not be a problem for different platforms.

1.2.5 Response Time

We will design our system in order to have low response time meaning that it should not surpass 1 second. Since it is an interactive game, responses to user inputs should not be too high.

1.2.6 Good Documentation

The system has to be well documented for both user and developer. In order to achieve well documentation, all reports need to have minimum error and should be clear to understand. Besides that a user manual will be provided.

1.2.7 Well-defined Interfaces

The user interface should not be complicated. We are going to design our system such a way that all visuals will be self-explanatory.

1.2.8 Readability

We aim to have a readable source code. It will allow that developers understand the source code without any problem. Thus, modifying the existing code will not be a problem.

1.2.9 Ease of Use

Our games will be designed regarding that it can be played by players from different age group. So it will not be difficult to understand and control the game. In order to achieve that,

we will keep the number of input keys as minimum as it can be. Besides, difficulty level of game will be neither too easy nor too hard.

2. Software Architecture

2.1 Subsystem Decomposition

To be able to provide all the classes in our project as a proper, organized and working model, we created the sub-system decomposition. The system structure is created in a way that the classes serving for similar functionalities are working as a one system component. The sub-system components are designed to be working separately as much as possible for modularity and maintainability purposes. In the light of the above principles, we created the sub-system decomposition diagram in Model–View–Controller manner as it can be deduced from Figure 1 below:

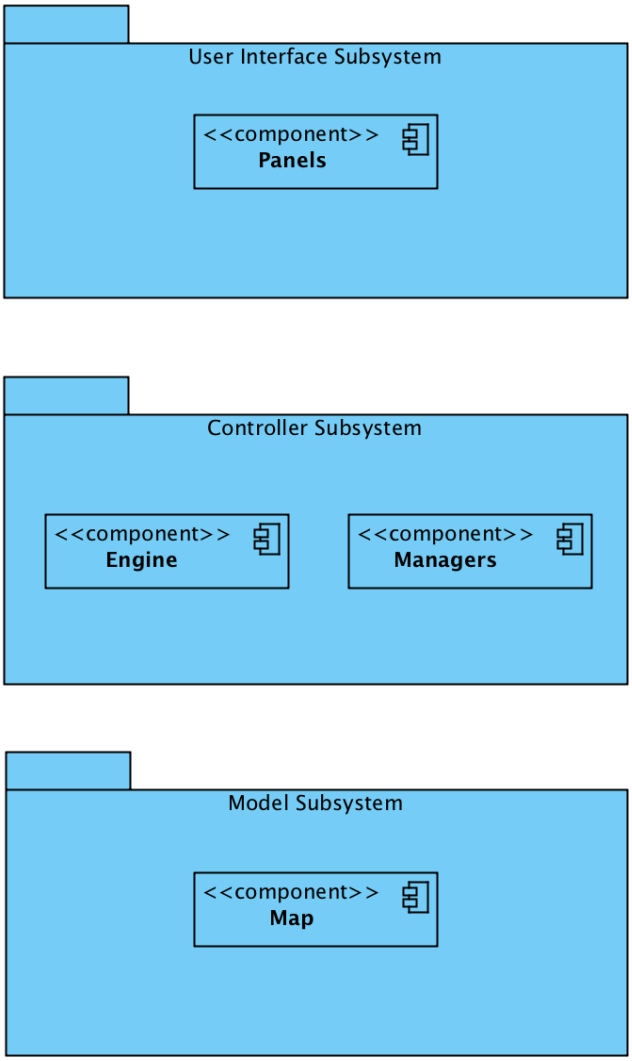

Figure 1: Subsystem Decomposition Diagram: The system is mainly composed of 3 parts; User Interface, Controller, and Model subsystems. They serve for GUI, Management, and the data parts of the project respectively.

2.1.1 User Interface Subsystem

The User Interface Subsystem will be composed of mainly Panels. It will handle the renderings for each specific screen from starting to the termination of the game. Transitions between screens and all of the object renderings for the current screen needs will also be handled in this subsystem.

2.1.2 Controller Subsystem

In the controller subsystem, we grouped together the manager objects to be able to control the game data and its logic.

2.1.3 Model Subsystem

In this subsystem our model objects and the relationship between them are represented. Model subsystem is basically the subsystem that keeps the data of the main game objects.

2.2 Hardware/Software Mapping

Bombplan will be developed using Java programming language; therefore we will use latest JDK (8). As hardware configuration, for player to type names on high score list and give input to the system Bombplan requires a mouse and a keyboard. To be able to run and play Bombplan, the computer needs to have an operating system and a Java compiler installed in advance. Java's platform independency is also needed as well.

Since to form game maps and store high score list and settings data we will use .txt files, and to store images and sounds in the .png and .wav formats respectively, the operating system should support .txt, .png, and .wav file formats. The game will not need internet connection.

Following deployment diagram shows communication between PC, Mouse, Keyboard and Monitor devices as separate nodes. As subsystem decomposition degrades the whole system in four components using Model-View-Controller architecture, these all components are depicted in PC node at below deployment diagram. PC node is in communication with Keyboard since player can play the game via keyboard buttons. And also mouse is using to give inputs to the panels. Monitor is output device which displays data given by view classes, in other words it communicates with Panels component. Inside of the PC, Engine depends on

Managers component. Engine requires Map interface, model data, and it provides interface to the Panels component, View classes.

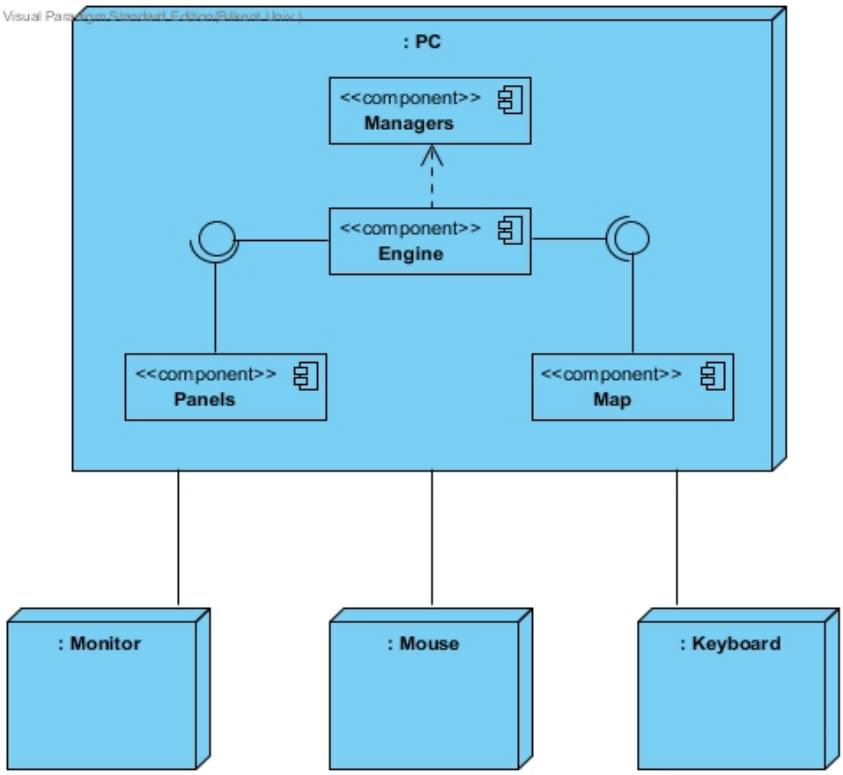

Figure 2: Deployment Diagram for Hardware/Software Mapping

2.3 Persistent Data Management

There is no need to use a complex database system, since Bombplan will store map structure, settings records and high score list in the associated text files. Our game maps as text files will be created in advance, so these data will be persistent. If any of these .txt files are corrupted, it will not affect game objects. On the other hand because of the file corruption the system will not be able to load map or the system will display error on the panel related to corrupted data. We will also store game sounds and object images in hard disk with simple sound and image formats (.png and .waw).

2.4 Access Control and Security

Bombplan will not need internet connection to run. After initialization, player can play the game without any authentication process. There will be no control or restrictions for access to game. Since Bombplan will not include any user profile, there will be no security issues in Bombplan.

2.5 Boundary Conditions

Initialization

- The game starts when player opens executable file. Bombplan is ready to use and main menu screen shows up. JVM loads all classes of the program.
- The game graphics are drawn by Java's Swing library.
- On the main menu screen there are seven buttons: Start Game, Help, Load Game, Options, Highscores, Credits, and Exit.
- When player selects Start Game option from the main menu, game panel will be loaded by the system and level one is going to be created and then game starts.
- When player chooses Help option from the main menu, Help panel will be displayed with its contents.
- If player chooses Load Game option from the main menu, system will load saved games from the associated files and show them as a list.
- If Options is selected, settings are going to be loaded and displayed by the system.
- When Highscores is selected, the system will load high score records and display them as a table.
- If Credits is selected, credits contents are going to be displayed by the system.
- Finally, if Exit is selected, the game will be terminated.

Termination

- On the main menu player can exit from the game by selecting Exit option.
- During the game player can exit using the pause menu by stopping the game. Since game is still continuing, after termination records will not be saved by the system.
- Player can press “x” button on the frame at the top right corner whenever s/he wants. If termination is done by clicking this button then no record will be saved.
- The system will ask player whether s/he really wants to quit when player clicks one of the exit buttons on the game. Termination is done if player chooses yes option.

Failure

- The game cannot be executed unless JRE is installed in the system.
- If high score, saved game or settings records could not be loaded from the associated files, the game will continue to run by displaying error on the panel for the related data.
- In the possible power cut termination, current data will be lost.
- If player forces the game to exit while the system is saving current records or the system is saving score of the player as high score or the system is saving settings at that time, the system could be unable to make these save processes successfully.

3. Subsystem Services

The system is decomposed into three subsystems which are Model, View and Controller subsystems namely. In this part services that are provided by these subsystems are demonstrated.

3.1.1 Services of the Controller

In this subsystem, we grouped together all of the controller objects to manage all of the game contents. Controller subsystem sits in the most upper part of our system, meaning that it have access to all of the other subsystems and manages them. It provides User Interface subsystem with the notifyChange service. Through this service, the user is able to control the whole system. It is simply a one directional pipeline between User Interface Subsystem and Controller Subsystem. When User Interface Subsystem is interrupted by user, according to the input taken from the user by the help of the action listener in GUI, the state of the program changes. After taking user input in User Interface Subsystem, a proper message is sent to the controller through notifyChange service. After the manager is notified through notifyChange service, it does the necessary changes on the system. For example, let's assume user is in the main menu and the program is in idle state, when he clicks on the "New Game" button, the action listener in menu panel sends the appropriate message through notifyChange service to the controller and the controller starts a new game by making the appropriate changes. Thus, screen changes or the control of the hero while user is playing the game is realized by the service "notifyChange" that controller subsystem provides.

3.2 Services of the Model

Model classes all together compose the Model subsystem. This subsystem stores application domain knowledge inside of it. Therefore it provides the necessary information of the entity classes to the Controller subsystem. Controller subsystem uses this data to handle the game operations via getModelData service of the Model subsystem. This service provides the Controller to get locations, types and any other necessary information of entity objects. This

information is crucial for the game engine to perform all business logic of the game. And also view is going to be derived and updated by this data accordingly.

The relation between Controller and Model subsystems is bidirectional. Controller subsystem changes model data as well. Considering player inputs and game-flow operations Controller subsystem stimulates Model subsystem to change, update itself. This is done by updateModelData service of the Model subsystem. This service is stimulated by Controller and it performs all necessary changes on the model data.

3.3 Services of the View

User Interface Subsystem consists of all panels that user interacts with. So the visuals need to be updated according to changing state. To satisfy that need, user interface subsystem provides a service called updateView. This service is used by Controller subsystem and its main functionality is that Controller subsystem updates the User Interface Subsystem by getting information from the service that Model subsystem provides. For example a wall is destroyed by bomb. Controller subsystem gets the information of destroyed bomb. It delegates the information to User Interface subsystem by updateView service. Thus, User Interface subsystem will remove the view of wall from panel.

4. Low-level Design

4.1 Object Design Trade-offs

4.1.1 Rapid Development vs. Functionality

In Bombplan we plan to keep the functionality at minimum by providing only necessary services to users, since we have a limited time. Deciding which services are necessary and

which are not, is an important process since it will affect the abilities and limitations of the game. In order to limit the services provided without limiting the game-play process, we will only provide services essential to the in-game experience of the game. Keeping the services limited will result in rapid development process.

4.1.2 Security vs. Performance

The Bombplan will keep saved game and default map data in txt files. The disadvantage of this method is that these files are open for outside modifications. This may result in corrupted map data that the game is not able to read and load. However, encrypting and decrypting this files will require time for each save and load operations. This situation will lead to a decrease in the performance. In order to provide better performance, the game will not protect the map data from possible manipulations.

4.2 Final Object Design

In this section, design patterns that are used and the reason behind those patterns are described. Pattern applied final class diagram can be seen in the figure 3. Observer, singleton and façade patterns are applied to the system.

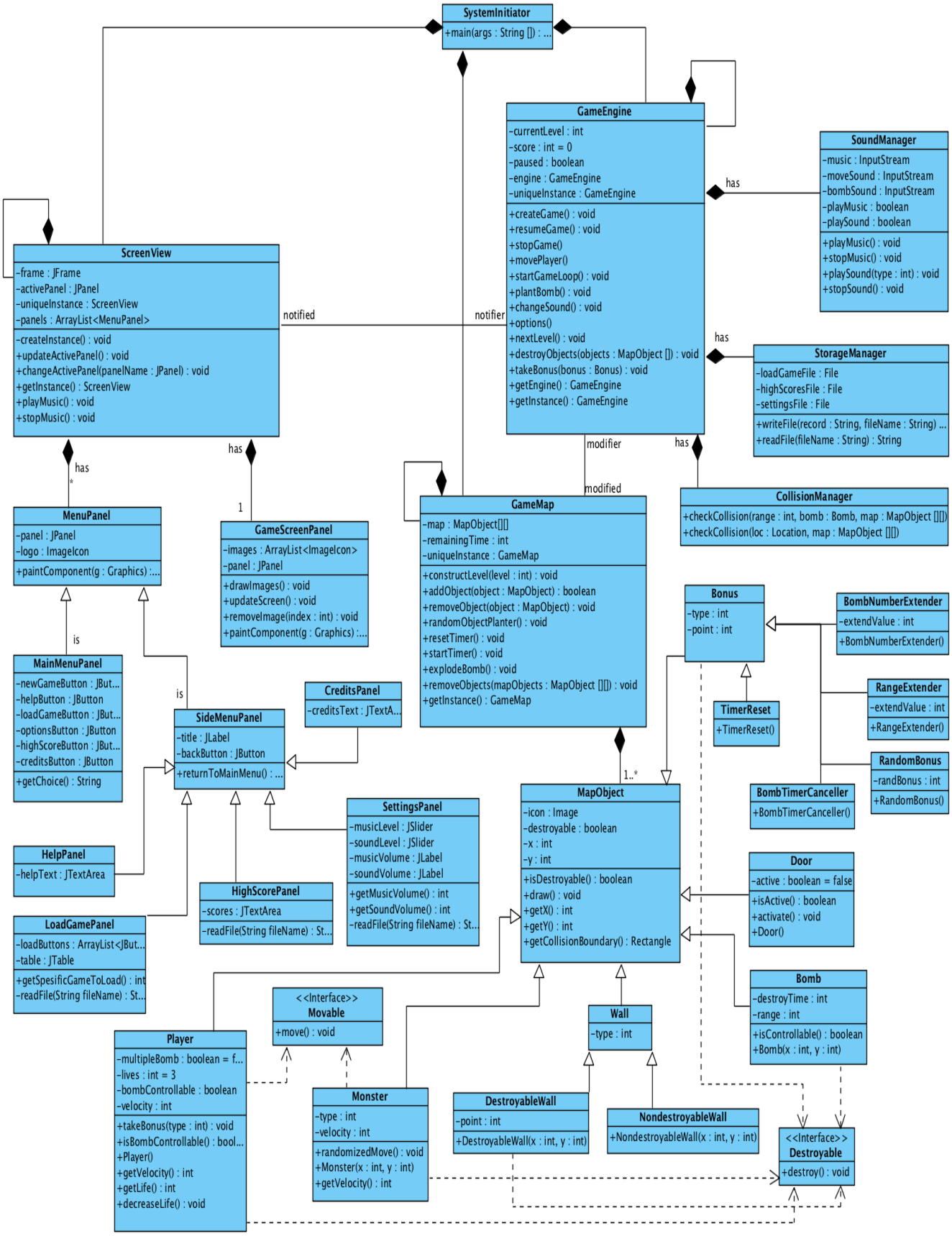

Figure 3: Final Class Diagram.

4.2.1 Façade Pattern

Model view controller pattern was applied as an architectural pattern. Between these subsystems services are explained previously, and communication is needed. Thus between model – controller and controller – view there will be data transactions. Since these communications are done among set of classes, subsystems, which are relatively complex components façade architectural pattern is applied in order to reduce this complexity and handle with the coupling problem within each subsystems. So, in the final class diagram ScreenView class and hierarchically dependent classes represents view façade of the system. GameEngine is the main controller and with its instances (manager classes) it represents controller façade of the system. And finally GameMap class is at the top on hierarchy of persistent model data; with its dependent classes it represents model façade of the whole system. Façade design pattern is applied as follows:

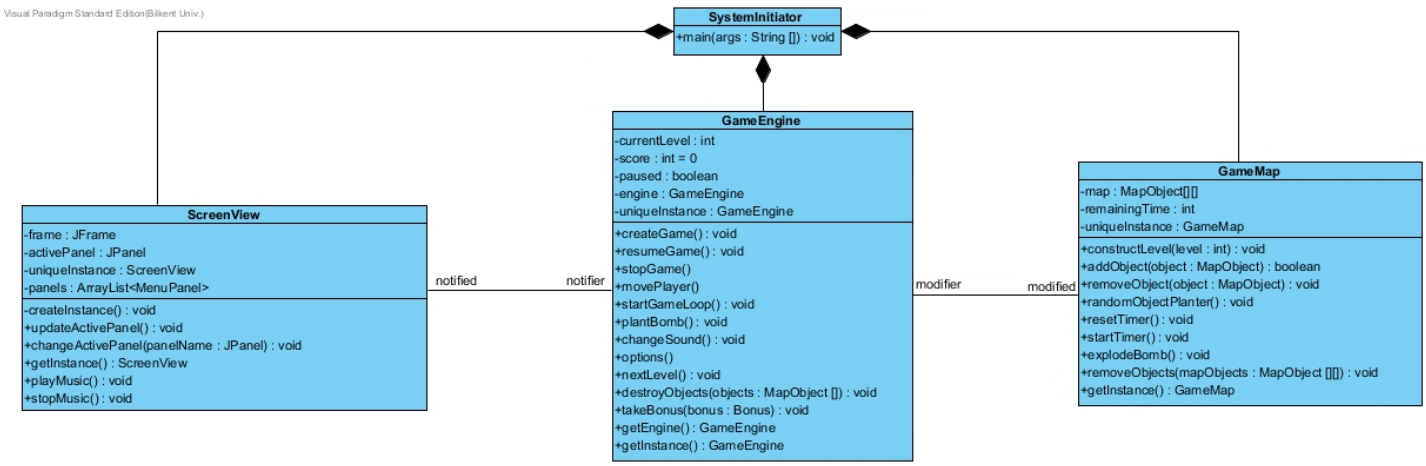

Figure 4: Façade design pattern

4.2.2 Singleton Pattern

In order to ensure façade classes have only one instance at a time, singleton design pattern is applied to these classes; namely GameEngine, ScreenView and GameMap. Since the façade classes are initialized in the main method at the beginning, same instances are need

to be used. Instead of sending these instances to all other classes that communicate with them they are defined as singleton classes. In this way there will be one ScreenView, one GameEngine and one GameMap class and global accesses to them will be done by calling static method getInstance() which is common to all.

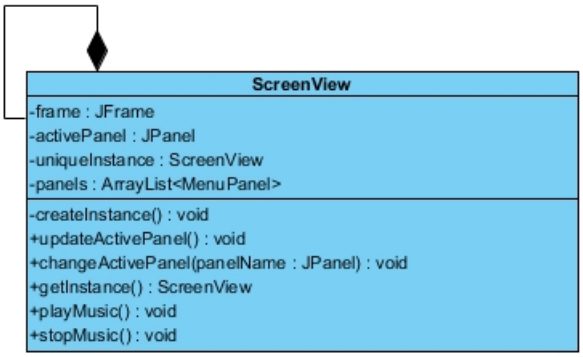

Figure 5: Singleton pattern applied to the ScreenView class

4.3 Packages

In this section detailed package diagrams are going to be supplied according to final class diagram. For each subsystem, class relationships will be explained. Then, how services of the subsystem will work is going to be expressed in the scope of classes and their methods.

4.3.1 User Interface Package

User interface package is view façade of the whole system. This package has frame and all panels including the game screen and menu panels. Thus at the top of the hierarchy there is ScreenView class which has frame and all other panels as instances. Game screen panel and menu panels are separated since their attributes and functionalities are different for most cases. MenuPanel has two types: MainMenuPanel and SideMenuPanel. CreditsPanel, HelpPanel, LoadGamePanel, HighScorePanel and SettingsPanel are types of the SideMenuPanel and all of them have common attributes as they can be accessed via MainMenuPanel. There is unique GameScreenPanel as an instance of ScreenView class and it

is the panel which the game flows, it is not a menu. On the other hand ScreenView has multiple menus.

User interface subsystem has updateView service. The controller façade of the system updates the view façade by getting information from persistent model data. Therefore in the ScreenView class controller façade will be called. Since settings, high scores and saved games are stored in text files and menu panels are going to display the information on the screen by reading these files, MenuPanel must communicate with the StorageManager. And since GameScreenPanel updates view according to current models data by the help of controller, it has to be communicating with GameEngine also. User interface package diagram with its classes is given below:

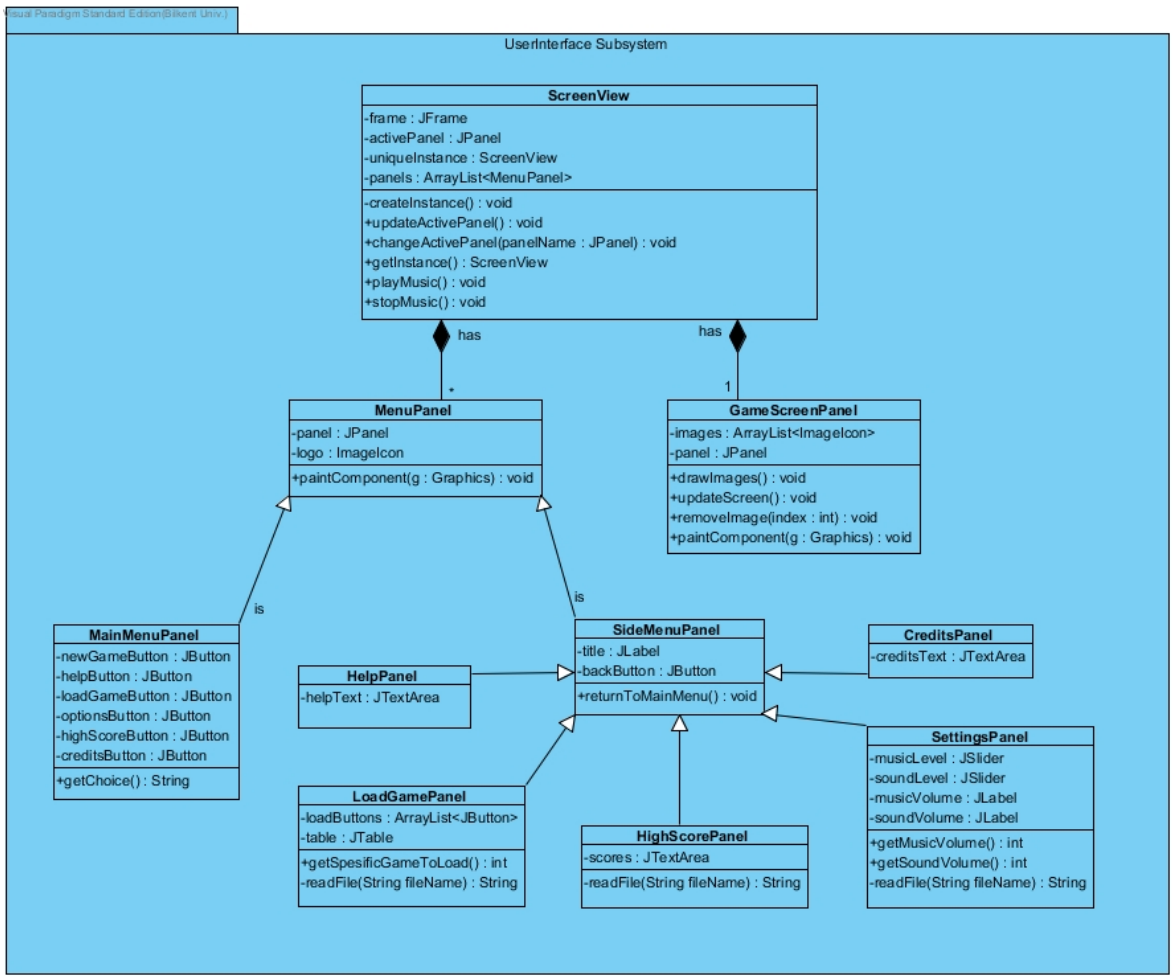

Figure 6: User Interface Package

4.3.2 Controller Package

Controller package is the package that provides the control services for the whole system. This package has manager classes and an engine. Engine is at the top of the class hierarchy in the package and it is the façade interface of the controller package. It has manager classes as an instance and uses them when they are needed. These classes are as following; SoundManager class helps engine to manage the sounds in the game, StorageManagerClass helps engine to manage files and CollisionManager class helps engine to detect the collisions in the game. Controller package also manages model classes by the help of the engine according to the inputs taken from the user. Since there must be only one instance of the controller package, to ensure the order because controller is the one managing models in the system. We ensure this by making the GameEngine class as a singleton because it is the only object that interacts with the other packages.

Controller package provides services to UserInterface to notify changes made by the user, save and load file requests, sound and music setting changes.

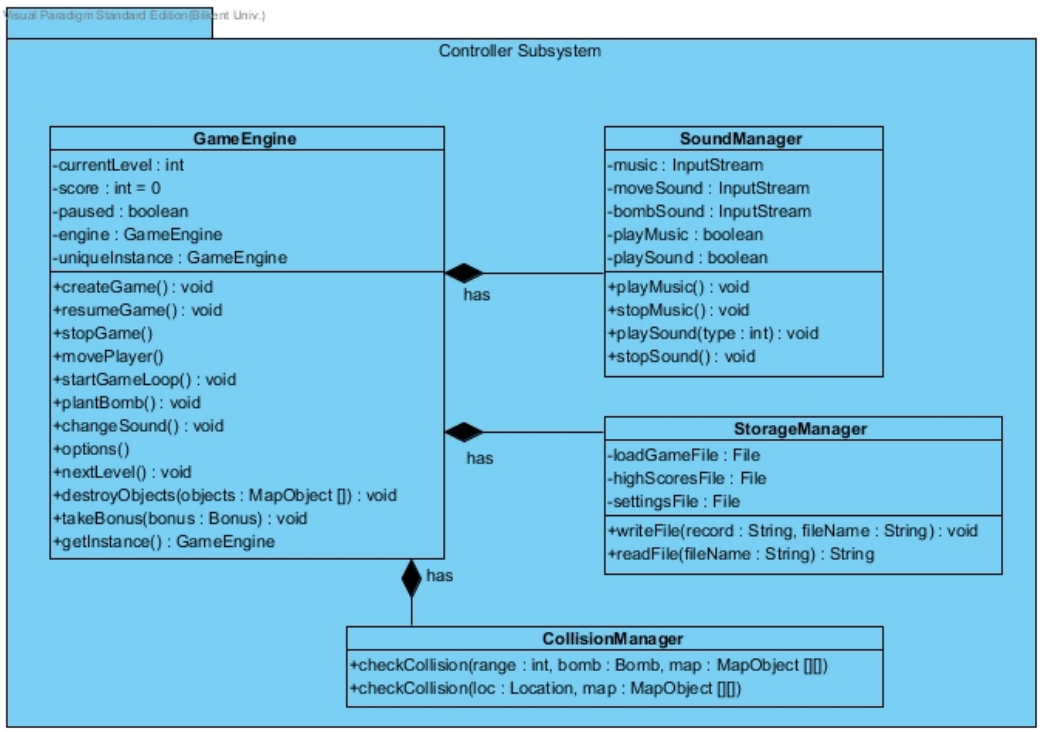

Figure 7: Controller Package

4.3.3 Model Package

This package contains all the entity objects of the system. On the top of the package there is a GameMap class. Controller subsystem reaches entity objects using this class. GameMap holds all MapObjects. There are 6 types of map objects. These are monster, player, wall, bomb, door and bonus.

Model subsystem has two services, getModelData and updateModelData. These services are used by controller subsystem. The aim of the services is communication between controller subsystem and model subsystem. getModelData provides information about entity objects and updateModelData allow controller subsystem to make changes on model objects. Model package diagram with its classes is given in the next page:

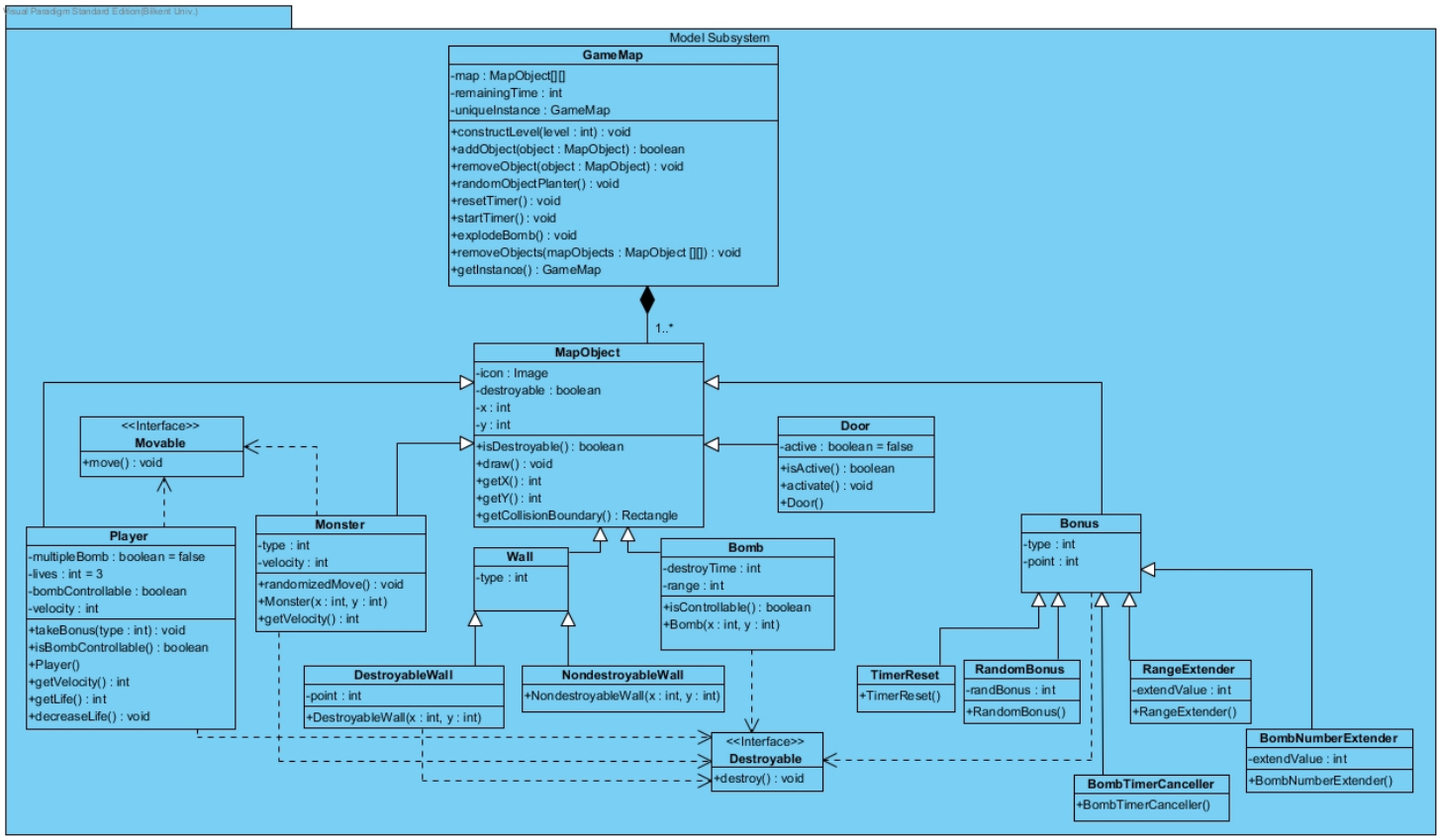

Figure 8: Model Package

4.4 Class Interfaces

4.4.1 User Interface Classes

ScreenView Class

The system has decomposed to model-view-controller parts and façade pattern is used in the system to make encapsulation explicit. So, ScreenView class is the one which holds all view classes together as a view façade of the game. Thus ScreenView is going to communicate with the Controller classes, and it takes information from the model classes.

- ***Attributes:***

private JFrame frame: It is frame of the program; panels are going to located into it.

private JPanel activePanel: This attribute indicates which panel is the active one.

private static ScreenView uniqueInstance: Static unique instance that makes ScreenView reachable from anywhere.

private ArrayList<GamePanel> panels: List of panels that ScreenView instance has.

- ***Constructors:***

private ScreenView(): Private constructor since singleton pattern is applied. It creates each type of panel once and make an array list from them. It initializes activePanel as a MainMenuPanel. Other attributes are also initialized.

- ***Methods:***

private void createInstance(): This method initialize uniqueInstance attribute.

public void updateActivePanel(): It updates view of the active panel of the game.

public void changeActivePanel(JPanel panelName): This method takes an input which type is JPanel and it changes activePanel to the that panel.

public static ScreenView getInstance(): This static method returns to the uniqueInstance attribute of the ScreenView instance from anywhere.

public void playMusic(): It triggers SoundManager to play music on the menu.

public void stopMusic(): It triggers SoundManager to stop music on the menu.

MenuPanel Class

MenuPanel class represents menu of the game and it has two types: one is MainMenuPanel which user encounters first and other is SideMenuPanel which user can go towards using MainMenuPanel. ScreenView class has an array list of MenuPanel classes, so it is responsible for changing active panel from one to other.

- ***Attributes:***

private JPanel panel: It is panel attribute that Java provides as a primitive type. All panel classes must have this attribute.

public ImageIcon logo: The attribute that is associated with image logo at the background.

- ***Constructors:***

public MenuPanel(): Default constructor which takes GameEngine controller instance as a parameter from the ScreenView and initializes all other attributes of the class.

- ***Methods:***

public void paintComponent(Graphics g): This method is default function of every panel: it takes Graphics as parameter and paints it on to the panel.

GameScreenPanel Class

- ***Attributes:***

private ArrayList<ImageIcon> images: Images of the model instances are going to form an arraylist inside of the GameScreenPanel class.

private JPanel panel: Panel attribute of this class.

- ***Constructors:***

public GameScreenPanel(): Default constructor which takes GameEngine controller as a parameter from the ScreenView and initializes all other attributes of the class.

- ***Methods:***

public void drawImages(): It draws all images on to the panel.

public void updateScreen(): It updates the panel with taking information of last locations of the model classes and the game flow attributes such as timer and life of the hero.

public void removeImage(int index): It removes the image that is found at the given index from the arraylist of images.

public void paintComponent(Graphics g): This method takes Graphics as parameter and paints it on to the panel.

MainMenuPanel Class

- ***Attributes:***

private JButton newGameButton: It is a button which enables user to start a new game.

private JButton helpButton: It is a button which enables user to get help on the menu.

private JButton loadGameButton: This is a button which enables user to go towards LoadGamePanel.

private JButton optionsButton: This is a button which enables user to go towards SettingsPanel.

private JButton highScoreButton: This button is used to display HighScorePanel.

private JButton creditsButton: This button is used to display CreditsPanel.

- ***Constructors:***

public MainMenuPanel(): Default constructer that initializes all attributes.

- ***Methods:***

public String getChoice(): When user clicks one of the buttons this attribute holds the preference of the user as a String.

SideMenuPanel Class

- ***Attributes:***

private JLabel title: Title attribute indicates what the panel is about, it is most of the time panel's name. Each class that is inheritor of SideMenuPanel is going to have this attribute.

private JButton backButton: It is a button which enables user to return to the main menu. Each class that is inheritor of SideMenuPanel is going to have this button.

- ***Constructors:***

public SideMenuPanel(): Default constructer that initializes just backButton since inheritor classes are going to have more specific constructors.

- ***Methods:***

public void returnToMainMenu(): This method perceives whether backButton is pressed or not and if this button is clicked the method triggers ScreenView to change the current panel to MainMenuPanel.

HighScorePanel Class

- ***Attributes:***

private JTextArea scores: This attribute is corresponding to high scores text.

- ***Constructors:***

public HighScorePanel(): The constructor that initializes scores attribute, backButton using inherited class and title as JLabel.

- ***Methods:***

private String readFile(String fileName): This method triggers reading of highScoresFile of the StorageManager using controller attribute and it takes high scores as String variables.

CreditsPanel Class

- ***Attributes:***

private JTextArea creditsText: This attribute is corresponding to credits text.

- ***Constructors:***

public CreditsPanel(): The constructor that initializes credits attribute, backButton using superior class and title as JLabel.

HelpPanel Class

- ***Attributes:***

private JTextArea helpText: This attribute is corresponding to instructions and game rules text.

- ***Constructors:***

public CreditsPanel(): The constructor that initializes helpText attribute, backButton using superior class and title as JLabel.

SettingsPanel Class

- ***Attributes:***

private JSlider musicLevel: This is a slider for changing music level.

private JSlider soundLevel: This is a slider for changing sound level.

private JLabel musicVolume: This attribute is label of the current music volume, which displays the value on the SettingsPanel.

private JLabel soundVolume: This attribute is label of the current sound effects volume, which displays the value on the SettingsPanel.

- ***Constructors:***

public SettingsPanel(): The constructor that initializes all attributes of the SettingsPanel, and also title and backButton from the superior class.

- ***Methods:***

public int getMusicVolume(): This method returns music volume as an integer value.

public int getSoundVolume(): This method returns sound effects volume as an integer.

private String readFile(String fileName): This method triggers reading of settingsFile from the StorageManager using controller attribute and it takes stored volume information as String variables.

LoadPanel Class

- ***Attributes:***

private JTabel table: This attribute indicates table of the saved games.

private ArrayList<JButton> loadButtons: Load game buttons, each one is corresponding to a specific saved game, form an arraylist which is named loadButtons.

- ***Constructors:***

public LoadPanel(): The constructor that initializes all attributes; loadButtons and table of the LoadPanel, and also title and backButton from the superior class.

- ***Methods:***

public int getSpesificGameToLoad(): A function which returns the number of the load button that is pressed.

private String readFile(String fileName): This method triggers reading of associated file, namely loadGameFile, from the StorageManager using controller attribute and it takes necessary information of the saved games as String variables.

4.4.2 Controller Classes

GameEngine Class

GameEngine Class is the main controller class of the game. It manages the game objects according to the inputs taken from the User Interface Subsystem and also it provides the continuity of the game loop.

- ***Attributes:***

private int currentLevel: This attribute indicates the current level of the game.

private int score: This attribute is used to store the current score of the user in the game.

private boolean paused: This attribute is used to check if the game is in paused state or not.

private static GameEngine uniqueInstance: Static unique instance that makes GameEngine reachable from anywhere.

- ***Constructors:***

private GameEngine(): Private constructor since singleton pattern is applied. The constructor initializes all of the attributes of the GameEngine.

- ***Methods:***

public void createGame(): This method initiates the game map with objects in the proper locations.

public void resumeGame(): This method resumes the paused game.

public void stopGame(): This method stops the game loop.

public void movePlayer(): This method updates the location of the player.

public void startGameLoop(): This method runs the loop that the system will be updated regularly.

public void plantBomb(): This method will plant a bomb to the location that the player stands.

public void changeSound(): This method will mute/unmute the sound according to the users preferences.

public void options(): This method will pause the game loop if user is playing and start the actions to open the settings screen.

public void nextLevel(): This method is called when game loop is broken by the collision of the player and the door. Then, the method simply initiates the next level if there is any. Otherwise, user is directed to the high score or main menu panel.

public void destroyObjects(objects : MapObject[]): This method is used to destroy the given objects from the map when there is a collision between the fire of the bomb and destroyable objects.

public void takeBonus(bonus : Bonus): This method is used to add the method functionality to the player when there is a collision between the player and one of the bonuses in the map.

public static GameEngine getInstance(): This static method returns to the uniqueInstance attribute of the GameEngine instance from anywhere.

SoundManager Class

SoundManager Class is the controller class of sounds of the game. It manages the sounds according to the inputs taken from the User Interface Subsystem and it is managed by the GameEngine.

- ***Attributes:***

private InputStream music: This attribute is the music file that will be played during the game loop.

private InputStream moveSound: This attribute is the sound file that will be played when the player moves

private InputStream bombSound: This attribute is the sound file that will be played when a bomb is exploded.

private boolean playMusic: This attribute indicates if user prefers to play the music.

private boolean playSound: This attribute indicates if user prefers to play the sound-fx when a bomb is exploded or the player is moving around the map.

- ***Constructors:***

public SoundManager(): The constructor initializes all of the attributes of the SoundManager.

- ***Methods:***

public void playMusic(): This method plays the music when user is playing the game.

public void stopMusic(): This method stops the music that is played when the user is playing the game.

public void playSound(type : int): This method plays the sound effects when user is moving or a bomb is exploded.

public void stopSound(): This method stops playing sound effects on the game.

StorageManager Class

StorageManager class is an instance of the GameEngine class and it is responsible from the reading and writing data procedures. It has all the files to handle these processes as private attributes.

- ***Attributes:***

private File loadGameFile: This attribute is corresponding to the file that stores all saved games as text inside of it.

private File highScoreFile: This is the file which stores high scores text.

private File settingsFile: This attribute indicates the file that stores music and sound information as textual data.

- ***Constructors:***

public StorageManager(): It is default constructer which gives the file names as String type parameters to the files attributes to initialize them.

- ***Methods:***

public void writeFile(String record, String fileName): This method writes the given textual data to the file which name is indicated in the parameter.

public String readFile(String fileName): This method reads the file that is indicated in the parameter by its' name, and returns String variable.

CollisionManager Class

CollisionManager Class is the controller class for possible collisions during game play. It detects the collisions and reports to the GameEngine.

- ***Constructors:***

public CollisionManager(): Default constructor.

- ***Methods:***

public MapObject[] checkCollision(range : int, bomb : Bomb, map : MapObject [][]): This method checks the collision explicitly after destroying a bomb and looks if there is an object that should be destroyed because of the effect of the bomb. It returns the objects that should be destroyed.

public boolean checkCollision(loc : Location, map : MapObject [][]): This method checks the collision explicitly in a specific location and returns a Boolean value.

4.4.3 Model Classes

GameMap Class

Controller subsystem achieves communication with objects by using the GameMap class. The class allows system to control objects on the map.

- ***Attributes:***

private MapObject[][] map: This attribute is the double array of MapObjects which contains all game objects on the map.

private int remainingTime: This attribute indicates the remaining time before the game is ended.

private static GameMap uniqueInstance: Static unique instance that makes GameMap reachable from anywhere.

- ***Constructor:***

public GameMap(): This constructor initializes attributes of GameMap.

- ***Methods:***

public void constructLevel(level: int): This method get the level from parameter and creates the game map based on this level.

public boolean addObject(object: MapObject): This method adds the MapObject to the game map. If it is added without any problem it returns true. If it could not be added then it returns false.

public void removeObject(object: MapObject): This method removes the MapObject from the game map.

public void randomObjectPlanter(): This method randomly adds object to the game map.

public void resetTimer(): This method returns the timer to its initial value.

public void startTimer(): This method starts the timer countdown.

public void explodeBomb(): This methods explodes the bomb and remove it from the map. If the bomb is controllable, user input is waited. Else, it explodes after seconds.

public void removeObjects(mapObjects: MapObject[][]): This method removes map objects from the map.

public static GameMap getInstance(): This static method returns to the uniqueInstance attribute of the GameMap instance from anywhere.

MapObject Class

- ***Attributes:***

private Image icon: This attribute is the visual representation of a MapObject instance.

private boolean destroyable: This attribute indicates whether a MapObject is destroyable or not.

private int x: This attribute is the x position of a MapObject instance.

private int y: This attribute is the y position of a MapObject instance.

- ***Methods:***

public boolean isDestroyable(): This method checks whether a MapObject is destroyable.

public void draw(): This method draws the view of the MapObject instance.

public int getX(): This method returns the x position.

public int getY(): This method returns the y position.

public Rectangle getCollisionBoundary(): This method returns a rectangle which CollisionManager class use to check collision.

Player Class

- ***Attributes:***

private Boolean multipleBomb = false: This attribute indicates whether player can plant multiple bomb or not. It is false by default.

private int lives = 3: This attribute shows number of lives that player has. It is three by default.

private boolean bombControllable: This attribute indicates whether player can control the explosion or not.

private int velocity: This attribute shows the speed of player.

- ***Constructor:***

public Player(): This constructor creates a player object and initialize default attributes of player.

- ***Methods:***

public void takeBonus(type: int): This method change the corresponding attribute according to the type of the taken bonus.

public boolean isBombControllable(): This method returns true if bomb can be controllable by the player.

public int getVelocity(): This method returns the speed of the player.

public int getLife(): This method returns the number of remaining lives of the player.

public void decreaseLife(): This method decreases number of remaining lives of the player by one.

Monster Class

- ***Attributes:***

private int type: This attribute is the difficulty type of the monster. According the type, monster has different attributes.

private int velocity: This attribute shows the speed of the monster.

- ***Constructor:***

public Monster(x: int, y: int): This constructor creates a monster at a given location and initializes its attributes.

- ***Methods:***

public void randomizedMove(): This method allows monsters move without follow a strict route while they are moving.

public int getVelocity(): This method returns the speed of the monster.

Wall Class

- ***Attributes:***

public int type: This attribute indicates whether a wall is a destroyable wall or no destroyable wall.

DestroyableWall Class

- ***Attributes:***

public int point: This attribute indicates the points that player gain after wall is destructed.

- ***Constructor:***

public DestroyableWall(x: int, y: int): This constructor creates a wall at a given location.

NondestroyableWall Class

- ***Constructor:***

public NondestroyableWall(x: int, y: int): This constructor creates a no destroyable wall at a given location.

Bomb Class

- ***Attributes:***

private int destroyTime: This attribute indicates the necessary time to past before bomb is exploded.

private int range: This attribute indicates the range of the bomb.

- ***Constructor:***

public Bomb(x: int, y: int): This constructor creates a bomb at a given location and initializes its attributes.

- ***Methods:***

public boolean isControllable(): This method checks whether the bomb can be controlled by player or not.

Door Class

- ***Attributes:***

private boolean active = false: This attribute indicates whether the door is active or not. The door can be seen only if it is activated.

- ***Constructor:***

public Door(x: int, y: int): This constructor creates a door and initializes its attributes.

- ***Methods:***

public boolean isActive(): This method checks whether the door is active or not.

public void activate(): This method makes the door active. After the door is activated, the door can be seen by player.

Bonus Class

- ***Attributes:***

public int type: This attribute indicates the type of the bonus. There are five type of bonus.

public int point: This attribute indicates the point that is gained after the bonus is taken.

TimerReset Class

- ***Constructor:***

public TimerReset(): This constructor creates a timer reset bonus. It resets the timer after player takes the bonus.

RandomBonus Class

- ***Attributes:***

public int type: This attribute indicates the type of bonus which is created randomly.

- ***Constructor:***

public RandomBonus(): This constructor creates a random bonus.

BombTimerCanceller Class

- ***Constructor:***

public BombTimerCanceller(): This constructor creates a bomb timer canceller bonus. After player takes that bonus, he will be able to control the explosion of the bomb.

RangeExtender Class

- ***Attributes:***

public int extendValue: This attribute indicates the extended range of bomb.

- ***Constructor:***

public RangeExtender(): This constructor creates a range extender bonus. The range of the bomb will be increased after player takes that bonus.

BombNumberExtender Class

- ***Attributes:***

public int extendValue: This attribute indicates the extended number of bomb.

- ***Constructor:***

public BombNumberExtender(): This constructor creates a bomb number extender bonus. The number of the bombs which player can plant increases after this bonus is taken.