Bilkent University

Department of Computer Engineering

CS319 PROJECT – GROUP #2

System Design Report

CS 319 Project: Bombalamasyon

Oğuz Demir – 21201712

Anıl Sert – 21201526

Kaya Yıldırım – 21002071

Kaan Kale – 21000912

**Course Instructor: Uğur DOĞRUSÖZ**

Design Report

Mar 27, 2016

This report is submitted to the GitHub in partial fulfillment of the requirements of the Object Oriented Software Engineering Project, course CS319

Table of Contents

[1 Introduction 2](#_Toc447912314)

[1.1 Purpose of the system 2](#_Toc447912315)

[1.2 Design goals 2](#_Toc447912316)

[1.2.1 Ease of Use 2](#_Toc447912317)

[1.2.2 Reliability 2](#_Toc447912318)

[1.2.3 Extendibility 2](#_Toc447912319)

[1.2.4 Responsiveness 3](#_Toc447912320)

[1.2.5 Portability 3](#_Toc447912321)

[2 Software Architecture 3](#_Toc447912322)

[2.1 Subsystem decomposition 3](#_Toc447912323)

[2.2 Hardware/software mapping 4](#_Toc447912324)

[2.3 Persistent data management 4](#_Toc447912325)

[2.4 Access control and security 5](#_Toc447912326)

[2.5 Boundary conditions 5](#_Toc447912327)

[3 Subsystem Services 6](#_Toc447912328)

[3.1 Services of the Controller: 6](#_Toc447912329)

[3.2 Services of the View: 6](#_Toc447912330)

[3.3 Services of the Model: 7](#_Toc447912331)

[4 Low-level design 7](#_Toc447912332)

[4.1 Object design trade-offs 7](#_Toc447912333)

[4.2 Final object design 8](#_Toc447912334)

[4.3 Packages 9](#_Toc447912335)

[4.4 Class Interfaces 11](#_Toc447912336)

[4.4.1 Model Classes 11](#_Toc447912337)

[4.4.2 Controller Classes 11](#_Toc447912338)

[4.4.3 View Classes 14](#_Toc447912339)

[5 Glossary & references 16](#_Toc447912340)

# Introduction

## Purpose of the system

Bombalamasyon is a system that aims to provide users with modified version of bomberman game with multiplayer and singleplayer features. The main purpose is serving a simple but challenging game for small breaks of computer users. The designed game is poor in terms of today’s game standards (such as complex AI, 3D smooth graphics, fluent movements etc.) so the gameplay of the game is changed to maximize the pleasure of achievement. The “Multiplayer” game mode allows two users to play at the same time from the same computer and to challenge each other. The game is designed to be played from most of computer platforms with different standards and without internet connection. The system also aims to provide a plain interface to make users learn the game easily and improve gaming experience.

## Design goals

In order to compose the system we should clarify the design goals we focused on. These design goals provided in analysis stage from non-functional requirements that we did before design. Here are described design goals:

### Ease of Use

Easiness in the usage is one of the most important design goals because it will determine whether the users continue to play the game or not. The game is designed to be played in small time intervals to enjoy and it would not be successful if the users have trouble while playing this game. Similarly, learning the game should not take much time not to waste the limited game time with learning.

### Reliability

The system is aimed to be bug-free in order to prevent from crashes or errors while gaming. The unexpected terminations at the middle of the games would be annoying for users. Also, any error that can cause loss of game statistic information of game statistics cannot be welcomed by users with high scores.

### Extendibility

The game is planned to be completed in limited time, so it is limited in terms of number of different features and levels. It is important to add new features to games in order to make them more attractive. Also, changes in game, will bring back the users who completed the game and abandon it. So, the system is aimed to be designed in a way that it can be easily extended for new features and levels.

### Responsiveness

The game is interactive game, the players use their in-game characters to complete the game objective. So, the users should immediately see their commands’ effect on the screen. In order to satisfy enough responsiveness for the users, the game view is designed to be refreshed every 0.1 second, in other words, the game shows 10 frames per second.

### Portability

In order to serve the game for the users from different platforms, the system should be portable, platform independent. The system aimed to be developed in Java so that it could be run every computer which has Java Virtual Machine.

# Software Architecture

## Subsystem decomposition

For the system Model-View-Controller (MVC) style is chosen to split system into parts for sharing the complexity of the system among the components. Also, since the components are designed to be independent as much as possible, it increases the readability of the code and extendibility of the system.

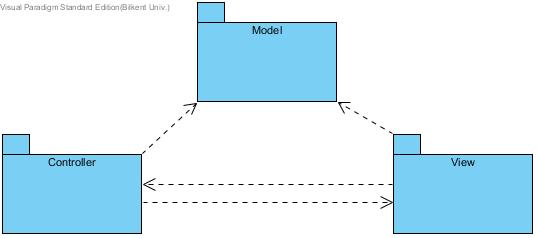


Figure 1 Subsystem Decomposition Diagram

In the Model subsystem we will hold all the game’s model objects which are interacted with each other during the play time and these models will be updated with the advance of the game. The physical events such as movements, collisions, map creations, object creations, ai behaviour of bombers are responsibilites of the Model subsystem.

Controller subsystem is the brain of our system. It includes controlers for Game, File Management, Sound Management. Responsibility of the Controller Subsystem is to manage the flow of the game, take neccessary information from files and pass it to other subsystems and to play appropriate sounds according to game state.

View subsystem has all of the view components of the system and the responsibilty of the View subsystem is to reflect the correct window with needed information on to the screen. The view is updated and/or current window is changed according to changes in the game state.

## Hardware/software mapping

Bombalamasyon requires Java Runtime Environment (JRE) to be played because it is developed by the using Java programming language. Game can be executed with a single executable Java file.

For the I/O requirements computer needs a keyboard, mouse and a monitor to let player interact with the game. For multiplayer gaming, the keyboard should not have ghosting (blocking some of the keys that are pressed at the same time) when 6 keys are pressed (6 keys: 3 for each player; 2 for direction, 1 for dropping bomb). It requires very little system requirements to be played. Graphical Processing Unit (GPU) is not required to play the game.

File system will be used for .png, .wav, and .txt files in order to take the game data and game images and play sound effects and background music.

## Persistent data management

Files are stored in the hard disk drive. The game keeps names and top ten scores in plain text file in order to display to the player in “High Scores” section. To provide better gaming experience to player, some image and sound files are also used at some parts of the game. When they are needed, these files are read from the disk with their specified directions as parameters. In addition, level data is stored in hard disk drive. There are different game maps for each level in hard drive.

## Access control and security

Bombalamasyon does not implement any user authentication system therefore we do not have any database that stores user credentials. Also, as mentioned earlier (in Hardware / Software Mapping), our game does not require network connection. Therefore, player who has no network connection is able to play the game. So that, there is no restriction or control for access the game. In addition, the game has no user profile, only player names and scores. Therefore, there is not security issues in Bombalamasyon.

## Boundary conditions

**Initialization**

When player execute the .jar file, the game initializes. Player does not have to install the game.

Secondly, the game tries to load every file that can be needed during execution. If a critical file such as level map is missing, the initialization will fail.

**Termination**

In order to terminate the game, player can click the “Quit Game” in the main menu. When player is playing the game, he/she wants to exit, firstly the player is need to go to “Pause menu” and then click the “Quit Game”.

Game will return to the main menu if all the levels are done. In case of finishing, high scores are updated if score is higher than 10th best score and the game returns to the main menu.

Game also be closed by the “X” on the top right corner of the window. Unsaved data will be loss.

**Error**

If any file (game resources) could not be loaded such as images or sounds, the game starts without these files. If the game does not respond because of other issues such as problem at hardware, software or operating system, player lose his/her current data.

# Subsystem Services

The system is decomposed into 3 parts as model, view, controller and there are 4 main services between these components. The flow is the following: when user give the input, the View takes the input as the boundary component, and it passes the related input to the Controller with Controllers’ service. Controller change the game status in itself and/or the properties of the Model with Model’s service. After that, the Controller ask for an update on the View via View’s service and before updating the current view, the View component can take the game data from Model with the service of the model. At the end, view is updated and changes with the user’s input is reflected on screen.

## Services of the Controller:

**takeUserInput**: This service of the controller is used by view component in order to pass the related user input to change the program status (main menu, paused game, in game etc.) and to control the game( move bomberman or drop bomb ). For example, if the user pauses the game while playing, the view component who has the action listeners for the keys, pass the corresponding input through takeUserInput service of controller for changing state and controller change the game status which is stored in the controller itself to “pausedGame”.

## Services of the View:

**updateView**: This service of the view is used by the controller to change the program display between menus or reflect the changes in the game map to screen. The status of the program such as main menu, in-game etc. is passed to the viewer and if it is in-game, the game data is taken from model component with the help of getGameMapData service of the model component.

## Services of the Model:

**getGameMapData**: This service of the model is used by the view component in order to get the game map data, in other words, positions of the game objects with their types.

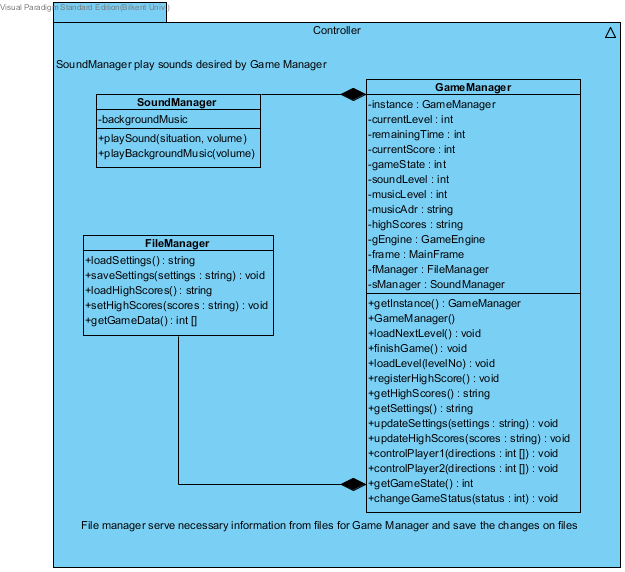
**updateGameObjects**: This service of the model is used by controller to manage in-game data with the desire of the user within a time interval and to process CPU controlled objects in that interval. The score that is earned during that interval is returned.

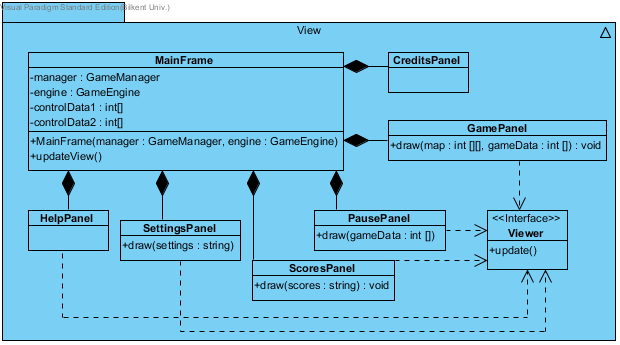
# Low-level design

## **Object design trade-offs**

## C:\Users\od\Desktop\Class Diagram2.jpgFinal object design

## Packages





## Class Interfaces

### Model Classes

GameEngine Class

GameEngine is the top level class of the model subsystem. The communication between other subsystems is done only through that class as explained before. Class is responsible for holding and manipulating all game objects and serve the game map to be drawn to the View subsystem.

Attributes:

**private OverlapEngine oEngine:**  A reference to the overlap engine to call its method when needed.

**private Bomberman[] bombers:** this attribute holds bomber objects in an array.

**private Bomb[] bombs:** this attribute holds bomb objects in an array.

**private Wall[][] walls:** this attribute holds wall objects in a matrix for ease of access and categorization (only walls in specific region are checked for collision with bombs or bombers).

**private PowerUp[] powerUps:** this attribute holds powerup objects in an array.

Constructor:

**public GameEngine(int[][] map):** Constructor takes the map of walls according to the current level and creates the walls according to that map. Also, 4 bombers in each corners are created and added to the collections. Bomb and powerup collections are created as empty arrays.

Methods:

**public boolean elapseTime(int x1, int y1, boolean b1,int timeAmount):** GameManager call this method in single player game and this method takes 3 commands for the player’s bomber and a time amount which is designed as 1second/desiredFps. This method calls elapseTime of all bomb and if a bomb is exploded, the beExploded() methods of nearby objects are called. The bomberman is moved with the commands and/or a new bomb is dropped.

**public boolean elapseTime(int x1, int y1, int x2, int y2, boolean b1, boolean b1, int timeAmount):** this method is multiplayer version of previous one.

**public int[][] serveGameMap():** Game data to be drawn is returned by this method with the request of view component.

**public void??? addGameObject(int type, int x, int y):** this method add new new game object.

**public void??? deleteGameObject(GameObject object):** this method deletes existing game object from game.

GameObject Class

It is abstract class for all game objects with the idea of polymorphism to iterate over bunch of game objects or checking collisions.

Attributes:

**private int xPosition:** this attributeholds current position in coordinate x-axis.

**private int yPosition:** this attribute holds current position in coordinate y-axis.

OverlapEngine Class

This class is used for checking collisions between two objects.

Methods:

**public boolean checkCollide(GameObject gameObject1, GameObject gameObject2):** this method checks if gameObject1 and gameObject2 collides or not and returns the result.

Explodable Interface

It is interface for game objects which are able to exploded by a bomb.

Methods:

**public void beExploded(GameEngine engine)**

PowerUp Class

This is an abstract class for the powerup objects with the idea of polymorphism to access all powerups with a single reference.

Methods:

**public void beTaken(Bomberman bomber):** if bomber moves and found a powerUp this method invokes and user takes a power up of its type.

Shield Class

This class represents the shield powerup.

Methods:

**public void beTaken(Bomberman bomber):** this method works if user founds a power up in shield type and calls the given Bomberman giveShield method.

**public void beExploded(GameEngine engine):** this method showsif a bomb explodes near a shield (power up) than this power up explodes and deleted from game.

MagnitudeUp Class

This class represents the bomb magnitude increase powerup.

Methods:

**public void beTaken(Bomberman bomber):** this method works if user founds a power up in magnitude type and calls the given Bomberman increaseMagnitude method.

**public void beExploded(GameEngine engine):** this method showsif a bomb explodes near a magnitude up (power up) than this power up explodes and deleted from game.

SpeedUp Class

This class represents the bomberman speed increase powerup.

**Methods**:

**public void beTaken(Bomberman bomber):** this method works if user founds a power up in speed up type and and calls the given Bomberman increaseSpeed method.

**public void beExploded(GameEngine engine):** this method showsif a bomb explodes near a speed up (power up) than this power up explodes and deleted from game.

LimitUp Class

This class represents the bomb amount limit increase powerup.

Methods:

**public void beTaken(Bomberman bomber):** this method works if user founds a power up in limit up type and calls the given Bomberman increaseLimit method.

**public void beExploded(GameEngine engine):** this method showsif a bomb explodes near a limit up (power up) than this power up explodes and deleted from game.

Bomb Class

This class is the representation of bomb objects of the game.

Attributes:

**private int timeLeft:** This attribute holds the remaining time to the explosion.

Methods:

**public boolean countdown():** Thiis method decreases the remaining time in each “elapseTime”. It returns true when the remaining time reaches zero.

Bomberman Class

This class is the representation of bombers in the game.

Attributes:

**private int lives:** This attribute holds the remaining lives until the death.

**private int speed:** This attribute holds the speed of the bomber.

**private int bombLimit:** This attribute holds the maximum number of bombs that the bomber can drop on the map at the same time.

**private int bombMagnitude:** This attribute holds the magnitude of the Bomber’s bomb.

**private int shield:** This attribute holds the remaining time for shield of the Bomber. 0 means no shield.

Methods:

**public void beExploded (GameEngine engine):** This method is called whenever the bomber is collided with an explosion and if there is no lives the game engine is called for deletion.

**public void move (int x, int y):** This method is called for moving the bomber.

**public void giveShield():** This method is called whenever a shield powerup is taken and shield property is set to time of the shield.

**public void increaseLimit():**This method is called whenever a shield powerup is taken and shield property is set to time of the shield.

**public void increaseMagnitude():** This method is called whenever a shield powerup is taken and shield property is set to time of the shield.

**public void increaseSpeed():** This method is called whenever a shield powerup is taken and shield property is set to time of the shield.

Wall Class

This class is the representation of walls in the game. The player-role pattern is applied to walls as explained.

Attributes:

**private WallRole role:**  This attribute holds a reference to the WallRole object to hold the wall type.

Methods:

**public void beExploded (GameEngine engine):** This method is called whenever wall is collided with an explosion. It calls the destroy method of its role.

Wall Role Class

This is an abstract class for collecting the wall types together under a class with the idea of polymorpishm and player-role pattern.

Attributes:

**private int scoreValue:** This attribute hold the score amount to be added on player whenever the wall is exploded. (0 for steel walls)

Methods:

**public void destroy (GameEngine engine, Wall wall):**  Abstract method to be implemented in child classes.

BrickWall Class

This class represent a wall type and extends from Wall Role.

Methods:

**public void destroy (GameEngine engine, Wall wall):** This method is delete this wall from collection of gameEngine with the explosion. It also calls the dropPowerUp method and it may drop a powerup.

**private int dropPowerUp():** This method decides whether a powerup is dropped or not with a certain probability. It also randomize the type of the powerup. (0 for no drop, positive numbers for different types of powerups.)

StrongBrickWall Class

This class represent a wall type and extends from Wall Role.

Methods:

**public void destroy (GameEngine engine, Wall wall):** This method is called with the explosion and changes the property of Wall object to change wall type to Brick Wall.

SteelWall Class

This class represent a wall type and extends from Wall Role. It is everlasting in the game and does not change form.

Methods:

**public void destroy (GameEngine engine, Wall wall):** This method is called with polymorphism but it does not do anything.

### Controller Classes

1. GameManager Class

GameManager is the top level class of the controller subsystem and communication between the other subsystems is provided with the help of this class. This controller class is responsible for controlling data transfer between the data files and the game and game datas are holding within the boundaries of this controller object.

**Constructors*:***

**public GameManager():** initializes the GameManager object with default attribute values. When created, it reads the information needed from the files with the help of the FileManager.

**Attributes**

**private GameManager instance: ????????????**

**private int currentLevel:** this attribute holds the current level information that the player is playing.

**private int remainingTime:** this attribute holds the player’s remaining time information to finish the level.

**private int currentScore:** this attribute holds the current score of the player.

**private int gameState: ?????????**

**private int soundLevel:** this attribute holds the loudness information of the game sound.

**private int musicLevel:** this attribute holds the loudness information of the music level.

**private String musicAdr:** this attribute holds the name**?????????????** of the music that are playing in the game.

**private String highScores:** this attribute holds the high scores of the game.

**private GameEngine gEngine:** this attribute holds a reference to GameEngine class to control the physical part of the game with using the engine.

**private MainFrame frame:** this attribute holds a reference to MainFrame class to draws appropriate screens for the game.

**private FileManager fManager:** this attribute holds a reference to FileManager class to use files when needed.

**private SoundManager sManager:** this attribute holds a reference to to SoundManager class to play sounds.

**Methods**

**public GameManager getInstance(): ????????????**

**public void loadNextLevel():** this method gets the information for the next level using the FileManager and load next level of the game.

**public void finishGame():** this method finishes the game and get the appropriate screen usgin the MainFrame.

**public void loadLevel(int levelNo):** this method takes the level number as a parameter that desired to be loaded and get the level information from files using FileManager.

**public void registerHighScores():** this method changes the high scores with the new high score rankings.

**public String getHighScores():** this method gets the high score information from the files using FileManager.

**public String getSettings():** this method gets the settings information from the files using FileManager.

**public void updateSettings(String settings):** this method takes the String of changed settings and update the settings file according to this new String using the FileManager.

**public void updateHighScores(String scores):** this method takes the String of changed high scores and update the hish scores file using the FileManager.

**public void controlPlayer1(int[] directions):** this method uses direction information taken from the MainFrame to control the player1 using GameEngine.

**public void controlPlayer2(int[] directions):** this method uses direction information taken from the MainFrame to control the player2 using GameEngine if there are two players.

**public int getGameState():** this method gets the state of the game.**??????????**

**public void changeGameStatus(int status):** this methods takes an integer value to change the game status according to that value.**??????????????**

1. FileManager Class

It is the class that helps interaction between the game files (i.e. settings, high score files) and the game.

**Methods**

**public String loadSettings():** this method reads the settings file and returns the appropriate information as a String.

**public void saveSettings(String settings):** this method takes changed settings as a parameter coming from GameManager and saves the new settings to the settings file.

**public String loadHighScores():** this method reads the high scores file and returns the high scores information as a String.

**public void setHighScores(String scores):** this method takes changed high scores String as a parameter coming from the GameManager and save the change high scores to the high score file.

**public int[] getGameData():** this method returns the level information of the levels reading the appropriate files.

1. SoundManager Class

This class is responsible for playing sound effects and background music in the game.

**Attributes**

**private ??????? backgroundMusic:** this attribute holds the background music information of the game.

**Methods**

**public void???????? playSound(????? situation, int???? volume):** this method plays the sound with the given volume and **??????????**

**public void playBackgroundMusic(int????? volume):** this method plays the background music of the game with the specified volume level.

### View Classes

1. MainFrame Class

This is a top level class of the view subsystem and data transfers between the model and controller subsystems is done by this class. This class is responsible for drawing the correct screens according to the user interaction and take inputs from the players.

***Attributes***

**private GameManager manager:** this attribute holds a reference to the GameManager class to draws screens with the request of GameManager.

**private GameEngine engine:** this attribute holds a reference to the GameEngine class to take the game data to draw to the screen from GameEngine.

**private int[] controlData1:** this attribute holds the input data for the player1.

**private int[] controlData2:** this attribute holds the input data for the player2 if there are two players playing the game.

**Constructors**

**public MainFrame(GameManager manager, GameEngine engine):** it initializes the object with the given GameManager and GameEngine references.

**Methods**

**public void?????? updateView():** this method updates the screen using the panel classes in accordance with the requests of GameManager.

1. GamePanel Class

This class is the view class that is used for the screen drawn and updated when the player actually playing the game.

**Methods**

**public void draw (int[][] map, int[] gameData):** this method draws the game screen with the map information and game data given from the GameManager.

1. PausePanel Class

This class is used for the screen drawn when user pauses the game and stop the playing action.

**Methods**

**public void draw (int[] gameData):** this method draws the pause screen with the given game data values as a parameter given from GameManager.

1. SettingsPanel Class

This class is used for the screen drawn with the settings information taken from data files when user selects the settings menu option.

**Methods**

**public void draw(String settings):** this method draws the settings screen with using the settings String that are taken from GameManager.

1. ScoresPanel Class

This class is used for the screen drawn with the high scores information taken from data files when user finishes a game or selects the high scores menu option.

**Methods**

**public void draw(String scores):** this method draws the high scores screen with using the scores information taken from the GameManager.

1. CreditsPanel Class

This class is used for the screen drawn when user selects credits menu option.

**Attributes**

**Constructors**

**Methods**

1. HelpPanel Class

This class is used for the screen drawn when user selects the help menu option.

**Attributes**

**Constructors**

**Methods**

# Glossary & references