Object Oriented Software Engineering Project

CS 319 Project: Block Breaker Extreme

Internal Documentation Report

Kaan BİNGÖL, Mevlüt Türker GARİP, Esat Rıdvan KAVLAOĞLU, Bora YALÇINER, Selçuk YILMAZ

Course Instructor: Uğur DOĞRUSÖZ

Progress
December 11, 2011

This report is submitted to the Moodle in partial fulfillment of the requirements of the Object Oriented Software Engineering Project, course CS319.
# Table of Contents

1. Introduction ........................................................................................................4

1.1. Purpose of the System ................................................................................4

1.2. Design Goals .................................................................................................4

1.3. Definitions, acronyms and abbreviations .................................................7

1.4. References .......................................................................................................7

2. Software Architecture .........................................................................................8

2.1. Subsystem Decomposition ........................................................................8

2.3. Hardware/ Software Mapping .................................................................10

2.4. Persistent Data Management .....................................................................11

2.5. Access Control and Security ......................................................................11

2.6. Boundary Conditions ..................................................................................12

3. Subsystem Services ............................................................................................13

 Detailed Object Design .......................................................................................13

 User Interface Management Subsystem ..........................................................12

 Game Logic Subsystem ......................................................................................14

 Game Screen Elements Subsystem ................................................................17

 File Management Subsystem ...........................................................................20

 Input Management Subsystem ..........................................................................21

 Game Map Management Subsystem ................................................................23

 Description of the Interactions between Classes According to the Use Cases ....24

4. Glossary .............................................................................................................27
Figure 1 - High-level Representation of Subsystem Decomposition .................................................. 8
Figure 2 - Interaction between Layer 1 and Layer 2 ................................................................. 9
Figure 3 - Interaction between Layer 2 and Layer 3 ................................................................. 10
Figure 4 - Detailed Class Diagram ............................................................................................. 14
Figure 5 - Package Diagram of User Interface Management ..................................................... 12
Figure 6 - Main Menu Class ....................................................................................................... 12
Figure 7 - ViewPanel Class ....................................................................................................... 13
Figure 8 - SettingsPanel Class .................................................................................................. 13
Figure 9 - Package Diagram of Game Logic ............................................................................. 14
Figure 10 - GameEngine Class ................................................................................................ 15
Figure 11 - GraphicsVisualizer Class ....................................................................................... 16
Figure 12 – Package Diagram of GameScreenElements .............................................................. 17
Figure 13 - GameObject Class .................................................................................................... 17
Figure 14 - Ball Class ................................................................................................................ 18
Figure 15 - Stick Class ............................................................................................................... 18
Figure 16 - BallState & StickState Enumerations .................................................................... 19
Figure 17 - Package Diagram of File Management .................................................................. 20
Figure 18 - FileManager Class .................................................................................................. 20
Figure 19 – Package Diagram of the Input Management ............................................................ 21
Figure 20 – InputManager Class ............................................................................................... 21
Figure 21 – Package Diagram of the Game Map Management .................................................. 23
Figure 22 – BrickMapManager Class ....................................................................................... 23
1. Introduction

1.1. Purpose of the System

Brick Breaker Extreme is a system aiming to entertain users with a well-designed gameplay. The system has a user-friendly interface which enables the user to learn how to play easily. The game starts with an easy level which aims to make the user involved in the gaming experience. Afterwards, it gets harder in order to preserve the user’s attraction. Therefore, our main purpose is to design the game easy to learn hard to master.

1.2. Design Goals

- Adaptability[1]: Java® is one of the few programming languages which provide cross-platform portability. This attribute of Java® enables our system to work in all JRE installed platforms; therefore, user will not have to worry about the operating system requirements. In order to fulfill this adaptability feature, we preferred to program the game in java by sacrificing some of the performance advantages of other programming languages such as C or C++.

- Efficiency: The system is going to be responsive and able to run with high performance. The game will run at least 40 fps in order to provide smoothness in the movements of the game objects. This is the most important design goal because performance of the game has a crucial role for users’ excitement. In order to reach the optimum game performance, rather than trying to minimize the memory usage, we allocated memory for each individual objects so that they will be responsible for their own tasks.
This will boost the performance significantly because it will decrease the workload of the “GameEngine”.

- **Reliability**: System will be bug-free and consistent in the boundary conditions. The system should not crash with unexpected inputs. To achieve this goal, the testing procedures will continue simultaneously with each stage of the development. Besides, boundary conditions will be evaluated very carefully not to miss any unconsidered situation which may crash the system.

- **Usability**: Easiness in the usage is an important design goal in such games in terms of user’s comfort. It makes the game more friendly and attractive. Therefore, the system will be designed such that user can easily interact with the system without any prior knowledge. The game will have a commonplace outline in the main menu and in-game menu. However, user-friendliness of the system does not imply making the gameplay easier, which might make the player bored sooner than expected.

- **Extensibility**: Object oriented architecture of the game enables system customizations without causing any bugs during modifications. For instance, with the abstraction of the brick map manager class from other controller classes, we can easily change or add the level map designs without having to modify anything in other classes. Therefore, this design architecture minimizes the possibility to cause malfunctioning in other classes.
Tradeoffs

- **Efficiency – Reusability:** Reusability is not the main concern for designing our system because we do not plan to integrate any of our classes to a different game or any other similar systems. Therefore, the classes are designed specifically for the tasks of our game so the code is not made more complex than necessary. This design approach fortified the our most important design goal, which is efficiency, since there will not be any fancy "if statements" or type checking to enforce the reusability constraint.

- **Functionality – Usability[2]:** It is very important to have wide range of customers. Therefore, the game will have plain usage. The system should not be too complex to play. It means that the functionality of the system will be basic. Since the purpose of the system is entertaining the users, we focused on the usability of the system rather than making it functional more than necessary. The game has a simple interface and familiar instructions to play instead of complex menus and various features. Thus, the users can spend time enjoying the game rather than struggling to learn it.

- **Space – Speed:** The implementation details planned to make our system more efficient and fast are mentioned in the design goals part above. The creating each game object separately is one of them. Considering there will be more than 40 brick objects to bring efficiency to the game logic, we decided to sacrifice a lot in terms of the memory space used. However, the speed of operations such as the collision detection and brick breaking is increased dramatically; therefore, it compensated the disadvantages caused by extravagant memory usage.
1.3. Definitions, acronyms and abbreviations

**Cross-platform**[3]: Cross-platform refers to ability of software to run in same way on different platforms such as Microsoft Windows, Linux and Mac OS X.

**JRE**[4]: Java execution environment is termed as the Java Runtime Environment (JRE). All systems need Java Runtime Environment (JRE) in order to execute the projects which are developed in Java.

**Fps**[5]: Abbreviation of “frames per second”. Fps represents the number of graphical layouts can be prepared by the system each second.

**GameEngine**: Our system’s core class. It has the most fundamental utility functions such as collision detection and main logic operations.

**Boundary conditions**: Conditions of the system which may generate run-time errors. They are exceptional cases according to the normal flow of the program. These conditions must be handled carefully for robustness of the system.

1.4. References


2. Software Architecture

2.1. Subsystem Decomposition

Among all design architectures, we chose the three-tier architecture to design the system because it is the most suitable design pattern that fits our system structure. Diagram describing the three-tier principle of our design is depicted in Figure 1.

![Diagram of Subsystem Decomposition]

Figure 1- High-level Representation of Subsystem Decomposition

In the presentation layer, we have classes responsible for providing an interface between the user and the system. Therefore, the class “MainMenu”, located in the “User Interface Management” package, is the first point where user interaction starts. Afterwards, it delegates the user’s choice to the “Game Logic” package.
In the logic tier, the user’s choice will be evaluated by the “Game Logic”. If the play game is chosen, the “GameEngine” constructs the game layout with the help of “Game Map Management” package and “GraphicsVisualizer” class. In every stage of the game, “GameEngine” which is the Façade class of the “Game Logic” interacts with “Input Manager” to check any user input.

The “Game Map Management” package is responsible for containing the game layout including the brick map pattern. The information in this package will be updated continually according to the results of game objects’ interactions checked by “GameLogic”.

Figure 2 - Interaction between Layer 1 and Layer 2
Data-layer is composed of the utility classes that are responsible for file operations and data entities. “Data Management Utility” contains the “FileManager” class, which deals with saving and loading the game data by interacting with the file system. The game objects and their attributes are located in the “Game Screen Elements” package and are manipulated by both the “GameEngine” and “BrickMapManager”. The elements in this package store all relevant information such as their current state, position, speed and image.

2.3. Hardware/Software Mapping

Our game will require the Java Runtime Environment to be executed since it is developed by Java. As hardware requirements, the game requires a keyboard or a mouse to be able to interact with the game.
In terms of I/O requirements and graphical requirements of the program, the game requires a decent computer for today’s standards as a requirement. Java’s 2D graphics library (Graphics2D class) uses GPU acceleration when available, so having a GPU is a plus. Input overhead of the program is very small it handles 3 keys and mouse movement in real-time thus, it can be considered nothing on today’s hardware.

2.4. Persistent Data Management

Game data will be stored in the client hard disk drive, we will not use any database since the data we use in the game needs to be accessed in real-time. Therefore, we will load all the necessary files on to the memory and access those files when the game logic or the rendering system requires. The background images, images of the game elements as well as the images of the buttons will be stored un-encrypted to encourage modifying background images, brick images for their personal preference.

2.5. Access Control and Security

Our game will not implement any user authentication system therefore we will not have any database that stores user credentials. For the security of our program, the only access to the file system is given to the "GameEngine" which is the main game logic class. Besides, we made all critical decision variables in the program, which is not supposed to change at any time, constant to assure the security of the information flow. We also decentralized the game logic by delegating basic tasks such as input management, image controlling of the game objects and managing the game map to
the classes that are specialized for these jobs. It did not only reduce the complexity of the "GameEngine" but also increase the robustness of the code.

2.6. Boundary Conditions

The program will give an error if the file or the saved data in it are corrupted and will delete its content. Game will return to the main menu if all the lives of the player are gone. In case of death, high scores will be updated if a new record is broken and the game will return to the main menu. The game has finite number of levels and if the user achieves to complete the game, high scores will be updated and the game will return to the main menu. If the user opens the program again while it is already running, program terminates.
3. Subsystem Services

Detailed Object Design

The detailed class diagram is depicted in the next page in order to provide a better understanding about the interactions and basic fundamentals of our software. With the help of this class diagram, the detailed subsystem descriptions in the next sections will be easier to comprehend.
Figure 4 - Detailed Class Diagram

Accessors and Mutators are not displayed in the class diagram for readability.
User Interface Management Subsystem

UIMS has three classes that are responsible to provide an interface between the user and the system. The class “MainMenu” is façade class of this subsystem.

“ViewPanel” and “SettingsPanel” is used to display other submenus such as “Settings”, “Help” and “High Scores”. The “MainMenu” delegates the control to these classes when the user wants to reach these submenus.

Main Menu Class
“MainMenu” is the first class that will be instantiated when the game is first executed and displays the main menu. In this menu, there are six buttons such as “New Game”, “Load Game”, “Settings”, “High Scores”, “Help” and “Exit”. “savedDataExists()” method will check whether there is any data saved previously and the load button will be disabled if such data doesn’t exists. The error will be given if the saved data is corrupted.

These are the other boundary classes of the UIMS which complements the “MainMenu” to ease its workload.
Game Logic Subsystem

Game Logic Package is responsible for handling game mechanics such as collision and display. Display management is handled by the class “GraphicsVisualizer”, collision detection and the other gameplay mechanics are handled by “GameEngine” class.

Figure 9 - Package Diagram of Game Logic
“GameEngine” is the façade class of the entire system; therefore it handles almost everything about game logic via using its methods. Moreover, “GameEngine” class starts the main game loop and inside the loop, “GameEngine” class repositions the Balls, Bullets and the Stick according to their speed with its updateObjects() method; then it checks the collision between balls, bricks and the stick and changes their states accordingly. Finally, at the end of the loop cycle it draws the objects to the screen using “GameVisualizer” class, checks the
game end conditions using isGameOver() method. If the isGameOver() methods return is true. It terminates the game loop and passes the flow of control back to the MainMenu class.

**GraphicsVisualizer Class**

```
public class GraphicsVisualizer {
    private BufferStrategy frameBufferStrategy;
    private Graphics2D currentGraphicsContent;

    public GraphicsVisualizer(BufferStrategy strategy) {
        this.frameBufferStrategy = strategy;
    }

    public void swapBuffer() {
        // Swap buffer
    }

    public Graphics2D getCurrentGraphics() {
        return this.currentGraphicsContent;
    }
}
```

*Figure 11 - GraphicsVisualizer Class*

“GraphicsVisualizer” class handles the rendering operations of the game logic. It will be used by the “GameEngine” class in order to render the scene. Since the program will use double buffering technique, “GraphicsVisualizer” class designed to handle buffer swaps using the Java® “BufferStrategy” class. While we rendering the current game elements to the buffer, we display the previously rendered buffer.
Game Screen Elements Subsystem

“Game Screen Elements Subsystem” declares the objects to show in the screen while the game is actually running. It has many crucial objects such as “Ball”, “Stick”, “Brick”, “Bonus”, “Bullet” and “GameObject”. Also, it has Enumerations such as “StickState”, “BallState” and “BonusType”.

Figure 12 – Package Diagram of GameScreenElements

GameObject Class

“GameObject” will be instantiated after user decides to play the game. All fundamental objects use the “GameObject” abstract class as a parent class since they all need to have position information such as “xPos” and “yPos”, size information such as “width” and “height”. Therefore “GameObject” has all these attributes. “update()” is defined as a abstract
operation since all objects need to act differently to this operation. We require to draw all of
the objects on to the screen therefore “GameObject” class has a image that will be drawn by
the “GraphicsVisualizer” class.

**Ball Class**

![Ball Class Diagram]

Figure 14 - Ball Class

The “Ball” class has additional attributes in addition to the GameObject class’s
attributes, such as “speedX” and “speedY” because it needs to move through X and Y
directions. It stores the state of ball by using enumeration, which will be overwritten when a
new state is active. It also includes the implementation of “update()” method. The “update()”
method manages the position of “Ball” according to its speed multiplied by “elapsedTime”.

**Stick Class**

![Stick Class Diagram]

Figure 15 - Stick Class

The “Stick” class stores remaining number of lives which will be decreased after
“Ball” dropped. It also has “StickState” and “speedX”.
Enumerations (Stick State, Ball State, Bonus Type Enumerations)

We used three different enumerations in “Game Screen Elements Subsystem” such as “BallState”, “StickState” and “BonusType” since we have a lot of states and bonus types. Instead of using all these types as integers we use these enumerations because they will ease our implementation organization. For example, we don’t need to know which integer value represents long stick state. We can use “E_LONG_STICK” to present the long stick state after we define it in the “StickState” enumeration only once.

![BallState & StickState Enumerations](image_url)
File Management Subsystem

File Management Subsystem is used in all saving and loading operations of the game by interacting with the file system. These operations are handled by the "FileManager" class of this subsystem.

![Diagram of File Management Subsystem](image)

Figure 17 - Package Diagram of File Management

File Manager Class

"FileManager" class is the only class of the FMS. The method "getLevelData()" is used in both "New Game" and "Load Game" operations by using the corresponding episode info. The file system uses text files for save and load operations. The method "saveGame()" takes the data which is generated by the "generateSaveData " method of the "GameEngine" in "Save Game" operation. "getHighScores()" and "getHelpDoc" methods are responsible for sending corresponding information to the "GameEngine".

![Class Diagram of FileManager](image)

Figure 18 - FileManager Class
Input Management Subsystem

This subsystem is responsible for the inputs required to use the system. It perceives user’s commands and makes the other subsystems process accordingly.

![Package Diagram of the Input Management](Image)

**Figure 19 – Package Diagram of the Input Management**

Input Manager Class

![InputManager Class](Image)

**Figure 20 – InputManager Class**
“InputManager” class is the listener class of the system. It allows keyboard and mouse control to the user. “buttonList” keeps the information of activated buttons. “getButtonList()” method is used by the “GameEngine” class to find pressed buttons by the user and changes the stick accordingly.
**Game Map Management Subsystem**

GMMS is the manager of the game’s levels. It stores the current level’s brick layout and brick objects.

![Figure 21 – Package Diagram of the Game Map Management](image)

**BrickMapManager Class**

![Figure 22 – BrickMapManager Class](image)
“BrickMapManager” class keeps the information of the brick map and the updates the map according to the current state of the bricks. The information of the brick map is stored in “brickBitMap” array. It is initialized by the Game Logic according to the level which will be loaded. “bricksToDraw” attribute is used to send the information of the current bricks to the Game Logic so that it can draw only the reaming bricks.

“getBirckIterator()” method returns the iterator of the “bricksToDraw” list. Game Logic uses this iterator to update the reaming bricks. The usage of iterator accelerates the process. “damegeBirck()” method changes the state of the given brick and if the brick is death, it updates the “bickBitMap” and “bricksToDraw” accordingly. “bircksLeft()” method returns the whatever any bricks are left or not. If the all bricks are destroyed, this method returns false and the next level is loaded.

Description of the Interactions between Classes According to the Use Cases

For all of the use cases, "MainMenu" class that has the main method will be instantiated. It constructs all the JButtons and JPanels for the main menu shown in the beginning of the program. The "GameEngine" object will be created even before the game starts and wait for the instructions from the "MainMenu".

**PlayGame:** This is the case that user wants to start a new game with the game data of level 1. Then "MainMenu" calls the startGame() method of the "GameEngine" with the isNew parameter set to true. Afterwards, loadLevel() method is called in the "GameEngine" with the level parameter set to 1. This method calls the getLevelData() function of the "FileManager" with the same level parameter. After the game data, which is int[][], is returned to the "GameEngine", it initializes the "BrickMapManager" object with this data. Consequently, the main game loop starts in the "GameEngine". In this loop, first updateObjects() will be called
to update game objects' positions and stick's according to the keysPressed array in the "InputManager". Then the checkCollision() method is called to handle with any collision between the game objects. If there is any collision between the ball and a brick, damageBrick() method of "BrickMapManager" class is called to modify the brick bitmap. After all the checks and modifications are done, drawObjects() method in the "GameEngine" that draws the ball, stick and bricks in the brick array that is returned by the getBrickIterator() method of the "BrickMapManager". Lastly, isGameOver() is called to decide whether to continue or break the loop. For each seconds passed the remaining time in the game will be updated by the updateTimer() method.

**ViewHelp:** "MainMenu" will switch from the JPanel mainMenuLayout to the helpLayout that will display the help text. Then loadHelpDoc() of the "GameEngine" will be called, which calls getHelpDoc() of the "FileManager" that gets the help document from the file. Consequently, returned value will be shown in the helpLayout.

**ChangeSettings:** "MainMenu" will switch from the JPanel mainMenuLayout to the settingsLayout that will display the currently saved settings and other choices that user can select. Then it calls the loadGameSettings() method of the "GameEngine". Lastly, "GameEngine" calls getSettings() to grab the saved settings from the file. settingsLayout will use these returned values to construct its user interface. After the change button is pressed, "MainMenu" updates the background image, difficulty level of the "GameEngine" and calls setGameKeys() method of "GameEngine" to update the keys in "Input Manager". At the end of the program, when the user clicks the exit game button, all settings will be written back to the file system using saveGameSettings() of "GameEngine".

**ViewHighScores:** "MainMenu" will switch from the JPanel mainMenuLayout to the highScoreLayout that will display the high scores. Then loadHighScores() of the "GameEngine" will be called, which calls getHighScores() of the "FileManager" that gets the
high scores information from the file. Consequently, returned value will be shown in the highScoreLayout.

**LoadGame:** The load game button in the "MainMenu" will only be enabled if saveDataExists() function returns true. Then the same procedure with the use case PlayGame except, this time, "MainMenu" calls the startGame() method of the "GameEngine" with theisNew parameter set to false. Therefore, "GameEngine" will make "FileManager" load from the file where previously saved data is written rather than default game data.

**SaveData:** In the inGameMenu if the user clicks on the save button, "GameEngine" will call generateSaveData() function and it calls the saveGame() method of "FileManager" with the generated game data to be written to the disk.

**Pause Game:** In this use case isPaused attribute of the "GameEngine" will be set to true, which will pause the inner game loop and wait for the user to continue.
4. Glossary

**UIMS**: Abbreviation of “User Interface Management Subsystem”

**FMS**: Abbreviation of “File Management Subsystem”

**GMMS**: Abbreviation of “Game Map Management Subsystem”