|  |
| --- |
| Bilkent University |
| Computer Science Department |
| CS 319 - Object-Oriented Software Engineering |



Final Report

Project Name: Protect the Factory

|  |
| --- |
| Group Members: Ahmet Ay, Alper Eroğlu, Tolga İlikli, Kaan Yorgancıoğlu  Instructor: Dr. Hüseyin Özgür TAN |
|  |

**Table of Contents**

1. Introduction ………………………………………………………………………………….. 3
2. Requirements Analysis …………………………………………………………………. 3
   1. Overview …………………………………………………………………………... 3
      1. Evil Machines ………………………………………………………………. 4
      2. Towers ………………………………………………………………………… 5
      3. Workshops ……………………………………………………………….…. 5
      4. Factory ……………………………………………………………………..…. 6
      5. Spare - Parts ……………………………………………………………..…. 7
   2. Functional Requirements …………………………………………………… 7
   3. Non-Functional Requirements …………………………………………… 8
   4. Constraints ………………………………………………………………………… 8
   5. Scenarios …………………………………………………………………………… 8
      1. Play Game ……………………………………………………………………. 8
      2. Put in Item …………………………………………………………………... 9
   6. Use-Case Models …………………………………………………………….. 10
      1. PlayGame ………………………………………………………………….. 10
      2. Mute …………………………………………………………………….…… 11
      3. Unmute …………………………………………………………………...… 12
      4. PauseGame ………………………………………………………………… 12
      5. ViewHelp ……………………………………………………………………. 13
      6. ViewCredits ………………………………………………………………… 13
      7. ExitGame ……………………………………………………………………. 14
   7. User Interface …………………………………………………………………… 14
      1. Navigational Path ……………………………………………………….. 14
      2. User Interface Mockups ……………………………………………... 15
3. Analysis ………………………………………………………………………………………… 20
   1. Object Model ……………………………………………………………………. 20
      1. Domain Lexicon ………………………………………………………….. 20
      2. Class Diagram. ……………………………………………………………. 21
   2. Dynamic Models ………………………………………………………………. 23
      1. State Chart …………………………………………………………………. 23
      2. Activity Diagram …………………………………………………………. 24
      3. Sequence Diagram………………………………………………………. 25
         1. Starting new game with a code ………………………. 25
         2. Muting the game ……………………………………………. 26
         3. Building a new tower ……………………………………… 27
4. Design …………………………………………………………………………………….…… 27
   1. Design Goals ..…………………………………………………………….……. 27
   2. Subsystem Decomposition ………………………………………….……. 31
      1. Overview of Component Diagram….……………………………. 31
      2. Manager Subsystem ..…………………………………………………. 32
      3. Interface Subsystem ..…………………………………………………. 33
      4. Game Entities Subsystem ..…………………………………………. 34
   3. Architectural Patterns ……………………………………………………... 35
      1. Layers ………………………………………………………………………... 35
      2. Model View Controller ……………………………………………….. 36
   4. Hardware Software Mapping ……………………………………………. 37
   5. Addressing Key Concerns …………………………………………………. 38
      1. Persistent Data Management……………………………………... 38
      2. Access Control and Security ….……………………………………. 38
      3. Global Software Control……………………………………………... 39
      4. Boundary Conditions ………………………………………………..… 39
5. Object Design ………………………………………………………………………………. 40
   1. Pattern Applications ………………………………………………………... 40
   2. Class Interfaces ………………………………………………………………… 44
   3. Specifying Contracts ………………………………………………………… 49
6. Conclusion …………………………………………………………………………………… 51
   1. About the Game ………………………………………………………………. 51
   2. The Design Process ………………………………………………………….. 51
   3. Lessons Learned ………………………………………………………………. 52
7. **Introduction**

Protect the Factory is a simple tower defence game. The purpose of the game is to build defensive structures in order to defend against hordes of evil machines marching to the factory. Defensive towers require a resource called “spare parts” to be built. Towers fire projectiles that reduce the health of the evil machines. Workshops produce the “spare parts” that will be used in the building and repairing of the towers. The player must place these structures on the lanes that the evil machines are using to reach the factory.

As the player progresses in the game, the levels become harder to complete with difficult enemies. However, the quality and power of the player’s defensive structures also increases with increasing levels. The game is a desktop application.

Protect the Factory is inspired by the game “Plants vs Zombies” (2009) by PopCap Games.

Plants vs Zombies:

<http://www.popcap.com/games/plants-vs-zombies>

1. **Requirement Analysis**
   1. **Overview**

Protect the Factory is a tower defense game like “Plants vs Zombies” by PopCap Games. In the game, the player must protect the factories from evil machines that are attacking it. Like the games of the same genre, the player must place obstacles to accomplish this task. The level is won when all the evil machines are destroyed by the obstacles before they reach the factory.

The game consists of 10 levels which become harder to complete as the player progresses. Also, the obstacles and towers the player can place in the way of the evil machines become more powerful and harder to destroy. There are also small workshops in the game that are placed as well that produce spare-parts for the towers. These help the player to repair his damaged towers.

There are 5 horizontal lanes that the evil machines attack from and the obstacles are also placed in these lanes. Each tower is assigned to destroy the machines that are attacking from its lane. The workshops are also placed on these lanes and provide spare-parts for the towers that are in the same lane. The towers and the workshops are placed by the player to the desired location by drag-drop. The player chooses the item he wants to place by choosing the item from the small menu above and drags it to its location. As the player progresses in the game new items are applicable for him to choose.

The player can also keep on playing from his latest level. After each completed level, he is given a code that indicates he has completed the previous level. At a later time when he starts playing, he can enter the code and choose his level to continue where he left. The player can replay the levels he had passed after completing that level or by choosing it later on with the appropriate code.

* + 1. **Evil Machines**

Evil Machines, are the antagonists of the game. Their task is to destroy the towers, workshops and the factory. In each level their type changes and they become stronger. Moreover, with each level the amount of evil machines that are attacking increases. They have a value associated with them that indicates how much life they are left with. They also have a damage value that shows the impact of their attacks on the towers, workshops and the factory. The types of the machines are as follows:

* Code 1: This is the least powerful type of machine that is seen in each of the levels. It has 20 life points and 20 damage points.
* Code 2: This is seen after level 3. It has 50 life points and 50 damage points.
* Code 3: This is seen after level 6. It has 100 life points and 100 damage points.
  + 1. **Towers**

Towers are one of the items that can be placed on the lanes by the player. They are assigned to protect the factory from the attackers. They fire projectiles and try to decrease the life points of the evil machines. The towers also have life and damage points. If the evil machines attack, their life points decreases. As the player progresses in the game, new tower types are opened to him which are more powerful. In order to place a tower, the user must have enough spare-parts that are produced by the workshops and also energy that is produced by the factory that is being protected. The spare-parts also allow the player to repair his towers when they are damaged. The types of towers are as follows:

* Turret 1: This is the least powerful type of tower. It is open from the beginning. No energy is required for it. It only requires 1 spare-part to be built. It has 10 life points and 10 damage points.
* Turret 2: This is seen after level 2. 20 units of energy and 5 spare-parts are required to build it. It has 50 life points and 50 damage points.
* Turret 3: This is seen after level 5. 50 units of energy and 10 spare-parts are required to build it. It has 200 life points and 200 damage points.
  + 1. **Workshops**

Workshops are another type of items that can be placed on the lanes by the player. The player uses them to produce spare-parts. The spare-parts are later used to build and repair the towers. They cannot damage the evil machines, however the evil machines must destroy them to continue in the lane. They have life points like towers that indicate how long they can keep up with the attacks. They also have spare-part points that show how much spare-parts they can produce in 30 seconds of time. As the levels increase more durable and efficient workshops are opened for the players use. The player must have enough energy units to build these workshops. The types of workshops are as follows:

* Local Shop: This is the least efficient and durable workshop. It is open from the beginning. No energy is required for it. It has 10 life points and 1 spare-part points.
* National Shop: This is seen after level 3. 10 units of energy are required to build it. It has 50 life points and 10 spare-part points.
* Global Shop: This is seen after level 7. 20 units of energy are required to build it. It has 100 life points and 20 spare-part points.
  + 1. **Factory**

The factory is the main item of the game. All of the lanes end up at the factory and the objective of the player is to protect it against the attackers. It produces a certain amount of energy that allows the player to build items. It has 200 life points that decrease when the evil machines come near it. In start of each level, the life points is reset to 200. If the factory is destroyed the game ends and the player loses. However, if he is able to stop all attackers he progresses to the next level. The amount of energy produced by the factory increases in each level. The factory begins with 10 energy points in the first level of the game. The energy producing points are increased to the double of the old value in each level.

* + 1. **Spare-Part**

The spare-parts are produced by the workshops that allow towers to be built and repaired. The amount of spare parts are located in the above menu. When a spare-part is produced the amount on the menu changes.

* 1. **Functional Requirements**
* Player can start a new game from the 1st level.
* Player can continue the game from the last level he completed with the required code.
* Player can replay an old level he completed by choosing it from the levels menu and providing the appropriate code of that level.
* Player can get a code after completing a level that indicates he can play the next level.
* Player can mute the game’s sounds by choosing the mute option from the 1st menu.
* The game will have 5 horizontal lanes that the items can be placed and the evil machines can move.
* The towers will hit the closest evil machine in the same lane.
* The workshops will provide spare-parts for building and repairing the game.
* Player cannot play the level if he cannot provide the appropriate code.
* Player can replay a level if he has lost in that level.
* The player can choose to quit the game after he has lost or completed a level.
* The player can choose the pause option in the game menu and restart the level, quit the game or resume playing the game.
* The game will give the option of continuing into the next level if the player has completed the current level.
* The player can choose Help option from the main menu and documentation about the game will be shown.
* The player can choose Credits option from the main menu and the developers’ name will be seen.
  1. **Non-Functional Requirements**
* The game will have a user-friendly interface. The menus will be as simple as possible and the in-game menu will be easy to use so that the player is not distracted from the game by the complexity of the menu.
* The game will give 30 seconds before the first enemy appears in each level.
* The game will have smooth graphics. The actions of the projectiles fired and also the movement of the evil machines will be similar to real life movements.
* The game will be extendable to have different difficulty levels in the future. The evil machines will come faster or more powerful machines will come if the difficulty is increased.
* The game will be extendable to be reused as a mobile game.
  1. **Constraints**
* The game will be implemented in Java.
* The game will use jdk 7.0.
* The game will be written in the eclipse luna IDE.
* The game will not be developed with frameworks.
  1. **Scenarios**
     1. **Play Game**

Scenario 1: Player chooses the “Play Game” option from the Main Menu. Player chooses “Start New Game” from the next menu in order to start from the 1st level. System initializes the screen for the game to begin. System starts the game with the appearance of the first evil machine. System continues to update the screen for the game to proceed.

Scenario 2: Player chooses the “Play Game” option from the Main Menu. Player chooses “Choose Level” from the next menu. Player enters the appropriate code to the system, the code will allow him to choose from the levels he passed. System provides the player with a list of levels he can choose from. Player chooses the level he wants to play. System initializes the screen for the game to begin. System starts the game with the appearance of the first evil machine. System continues to update the screen for the game to proceed.

* + 1. **Put in Item**

Scenario 1: Player continues playing the game. Player decides to put a Tower to one of the Lanes. Player chooses a Tower type from the menu above and drags and drops it to the location on the Lane. System updates the screen. System updates the spare-part and energy values on the menu. Tower starts firing projectiles to the nearest Evil Machine.

Scenario 2: Player continues playing the game. Player decides to put a Workshop to one of the Lanes. Player chooses a Workshop type from the menu above and drags and drops it to the location on the Lane. System updates the screen. System updates the energy value on the screen. Workshop starts producing spare-parts.

* 1. **Use-Case Models**

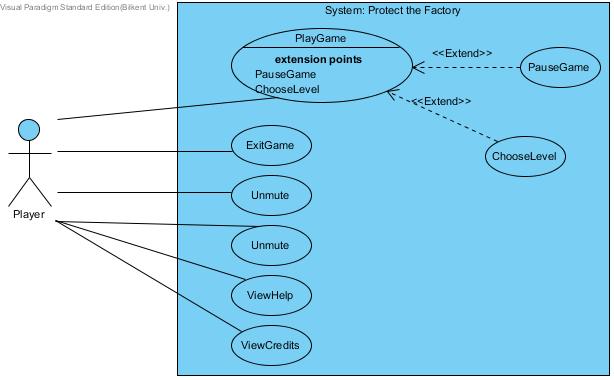


Figure 2.1. Use Case Diagram of Protect the Factory

* + 1. **PlayGame**

Use Case Name: Play Game

Participating Actor: Player

Entry Condition: Player selects “Play Game” from Main Menu.

Exit Condition: Player selects “Quit” from Pause Menu or End of Level Menu to exit to Main Menu.

Flow of Events:

1. Player chooses “Start New Game” option from Menu
2. System initializes the level
3. Player destroys all the Evil Machines in the level
4. System asks if the user wants to continue and gives level code to player
5. Player chooses to continue to next level
6. *Steps 2-5 are repeated for 10 levels unless the Factory is destroyed*
7. System displays End of Game Menu
8. Player chooses “Quit”
9. System goes to Main Menu

Alternative Flows:

* 1. Player chooses “Choose Level” option from Menu
     + - 1. Player enters the code last given to him by the system
         2. System displays levels
         3. Player chooses level
         4. Steps from 2 are repeated

3.1. Player decides to put Tower onto one of the lanes

(a) Player chooses one of the Towers

(b) Player drag and drops the Tower to the desired location on the Lane

(c) System updates energy and spare-parts values

3.2. Player decides to put Workshop onto one of the lanes

(a) Player chooses one of the Workshop

(b) Player drag and drops the Workshop to the desired location on the Lane

(c) System updates energy value

* + 1. **Mute**

Use Case Name: Mute

Participating Actor: Player

Entry Condition: Player is in Main Menu or Pause Menu.

Exit Condition: Player select Mute icon on screen.

Flow of Events:

Player chooses the Mute icon in the screen

System turns of the sounds

* + 1. **Unmute**

Use Case Name: Unmute

Participating Actor: Player

Entry Condition: Player is in Main Menu or Pause Menu and sounds are muted.

Exit Condition: Player select Unmute icon on screen.

Flow of Events:

Player chooses the Unmute icon in the screen

System turns on the sounds

* + 1. **PauseGame**

Use Case Name: Pause Game

Participating Actor: Player

Entry Condition: Player is playing the game.

Exit Condition: Player can continue game or quit to Main Menu or restart the level.

Flow of Events:

Player chooses the Pause icon in the game

System shows Pause menu

Player chooses “Continue” option

Player continues playing

Alternative Flows:

3.1. Player chooses “Restart” option

(a) System initializes to start of the level

(b) Player starts playing the game

3.2. Player chooses “Quit” option

(a) System exits to the Main Menu

* + 1. **ViewHelp**

Use Case Name: View Help

Participating Actor: Player

Entry Condition: Player is in the Main Menu.

Exit Condition: Player chooses “Back” option on the screen.

Flow of Events:

Player chooses the “Help” option in the Main Menu

System displays the guide of the game

Player chooses “Back” option

System returns to Main Menu

* + 1. **ViewCredits**

Use Case Name: View Credits

Participating Actor: Player

Entry Condition: Player is in the Main Menu.

Exit Condition: Player chooses “Back” option on the screen.

Flow of Events:

Player chooses the “Credits” option in the Main Menu

System displays the names of the game’s developers

Player chooses “Back” option

System returns to Main Menu

* + 1. **ExitGame**

Use Case Name: Exit Game

Participating Actor: Player

Entry Condition: Player is in the game.

Exit Condition: Player clicks the close screen icon from the frame.

Flow of Events:

Player clicks the close screen icon from the frame

The system exits the game and game frame closes

* 1. **User Interface**
     1. **Navigational Path**

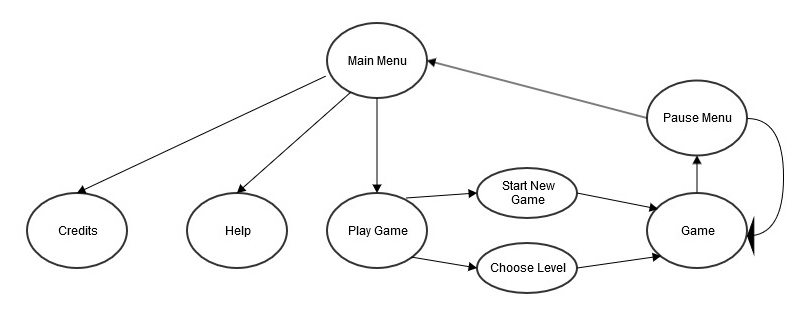


Figure 2.2. The navigational path of the user interface

* + 1. **User Interface Mockups**

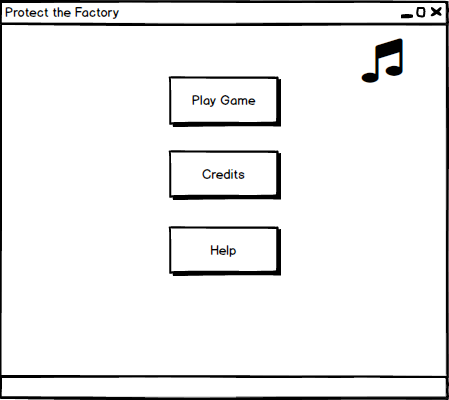


Figure 2.3. The Main Menu of the game

When the player starts the game, the first frame he will see is the Main Menu (Figure 2.3). In this Menu the player has 3 options to proceed. If he chooses the “Play Game” option he will proceed to the Play Game Menu (Figure 2.4). If he chooses “Credits” option he will go to the credits page that contains information about the developers of the game. If he chooses the “Help” option he will see a page that contains information about the game. These information will be about the gameplay, rules and other necessary documentation. In this Menu, the player also has the ability to mute the sounds of the game by pressing the sound symbol.

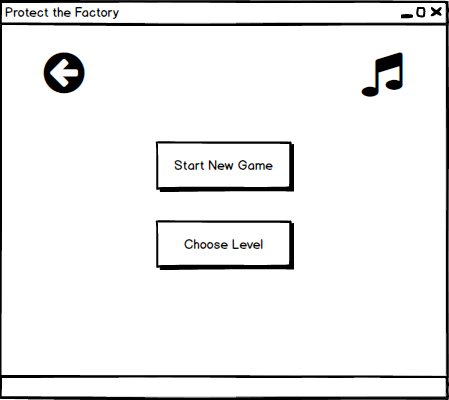


Figure 2.4. The Play Game Menu

In the Play Game Menu, the player again has 3 options and the mute button. If the player chooses the “Start New Game” he will directly start the game (Figure2.7). This option will start the game from the 1st level. If he chooses the “Choose Level” option he will proceed to the 1st Choose Level frame (Figure 2.5). The mute button has the same functionality as the one in the Main Menu. The back button can also be used to return to the Main Menu.

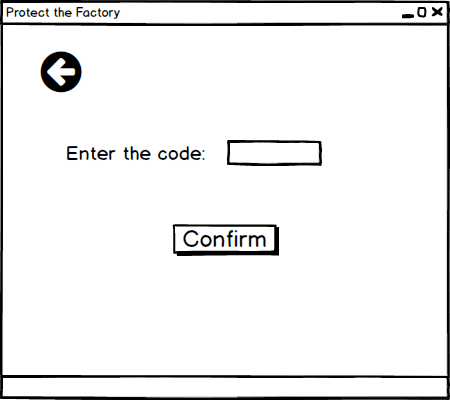


Figure 2.5. The 1st Choose Level frame

In the 1st Choose Level frame the system asks the user for a code. This code is given to the player at the end of a level if he has completed it successfully. He will enter the code to the given blank space. If the code is accepted by the system, the 2nd Choose Level frame (Figure 2.6) will be opened. The player can use the back button to return to the Play Game Menu.

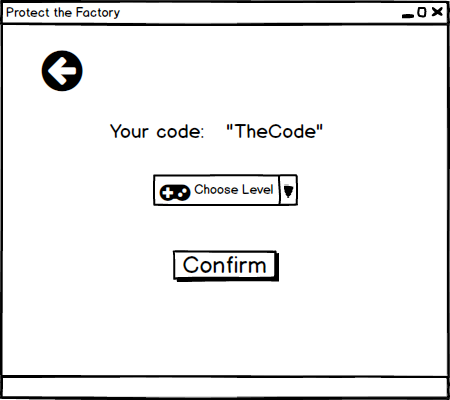


Figure 2.6. The 2nd Choose Level frame

In the 2nd Choose Level frame, the player is given the option of replaying a level that he completed before. The system will open the levels that the player can play in a combobox according to the code entered by the player. The player will choose the level he wants to play from the combobox menu. The system will proceed to the game screen (Figure 2.7). The game will start from the level the player had chosen. The back button in this frame can be used to return to the 1st Choose Level frame, so that the player can enter another code.

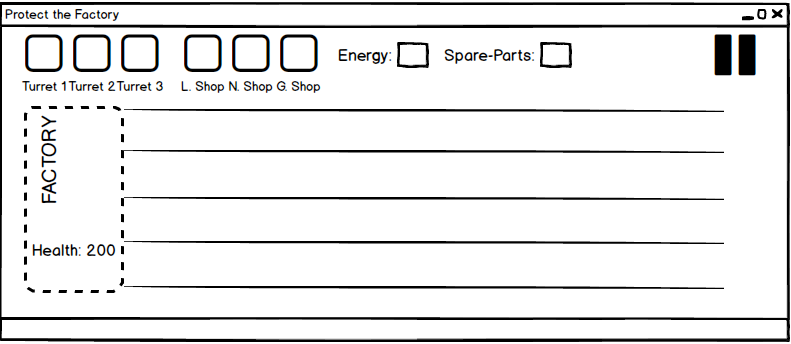


Figure 2.7. The Game screen

The game screen consists of the main game frame and the upper menu. The main game frame contains the Factory, towers, workshops, lanes and the evil machines. The main events of the game are depicted in this frame. The upper menu contains the items that the player can place on the lanes, the status of the energy and the amount of the spare-parts. It also contains the pause game button. When the player presses the pause button the Pause Game Menu (Figure 2.8) will be opened. If the player is able to complete the game with success the End Game Menu (Figure 2.9) will be opened by the system.

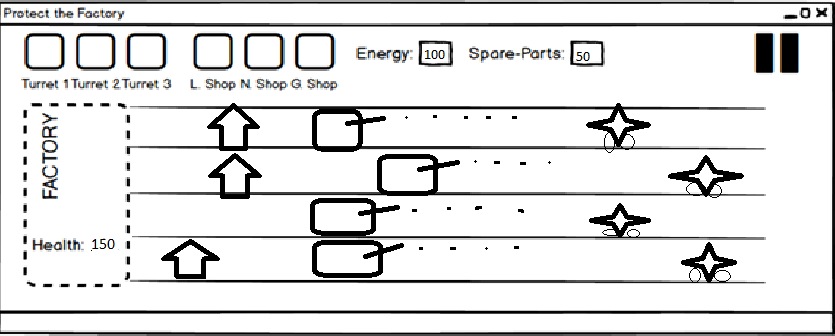


Figure 2.8. The Game Screen with enemies

During the play of the game, the screen is seen as in Figure 2.8. The player chooses turrets from the upper menu and places them on the lanes as seen. The player can also place workshops in the lanes. The status variables like energy and spare-parts are also visible and are updated in the upper menu depending on the flow of the game. The enemies also approach from the same lanes as the towers and workshops. The projectiles fired by the turrets are also visible on the lanes. These are small dots that are directed to the approaching evil machines.

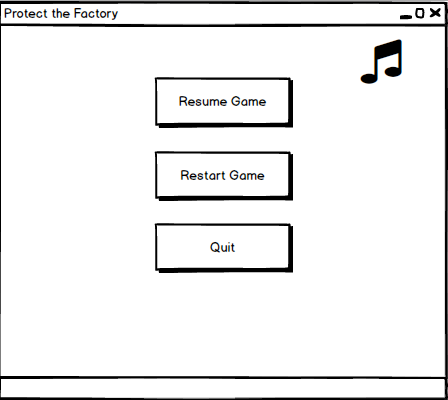


Figure 2.9. The Pause Game Menu

In this screen, the player can continue playing the game from the point he paused by pressing the “Resume Game” option. If he chooses the “Restart Game” option, the system will start the level from scratch. If the player chooses the “Quit” option the game will exit the game and open the Main Menu. The mute button can be used to silence of the sounds of the game.

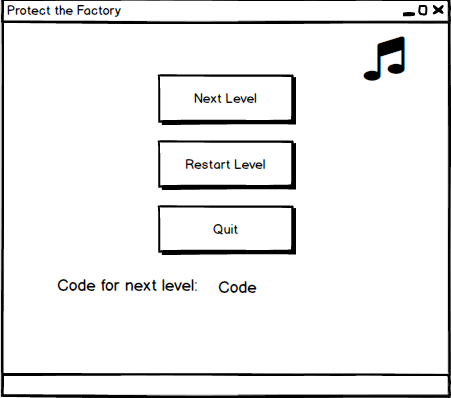


Figure 2.10. The End of Game Menu

In the End of Game Menu, player can choose to move to the next level by choosing the “Next Level” option. This choice will make the system start the game screen for the next level. If the “Restart Level” is chosen, the system will start the completed level from scratch. If the “Quit” option is chosen, the system will open the Main Menu. The next level code will be given in this frame, which will allow the player to play the next one and all previous levels when he enters it in the Choose Level frame.

1. **Analysis**
   1. **Object Model** 
      1. **Domain Lexicon**

* **Spare Parts:** Spare Parts are the main currency in Protect The Factory. All buildings require spare parts to be built. By default there is a slow and steady supply of spare parts but it will not be enough to establish a sufficient defence therefore the player has to build workshops that provide larger quantities of spare parts.
* **Energy**: The factory provides a different amount of energy output for each level depending on the difficulty. All buildings use up some of the energy output when they are up on the field but the energy is refunded when the building is destroyed. The reason that this mechanic exists is to limit the number of buildings that can be built in a level and therefore encourage the player to make more strategic decisions instead of amassing towers.
* **Tiles**: Tiles are squares that make up the field. Only one building can be built on a tile but multiple enemies can be on the same tile.
* **Lanes:** Lanes are the horizontal lines that are made up of tiles. Enemies march along the lanes in the direction of the factory. If an enemy reaches a tile that is occupied by a building it will attack the building until it is destroyed, then it will continue to march. If there is an enemy on a lane that a tower exists, the tower will start to attack the enemy. Towers always attack the enemy that is closest to them.
* **Factory:** The factory is on the leftmost side of the game screen. Each enemy that reaches the factory reduces its hit points for a certain amount. When the hit points of the factory reach zero the player loses the game.
* **Workshop**: Workshops provide the player a steady flow of spare parts.
* **Tower:** Towers attack the enemies on their lane.
  + 1. **Class Diagram**

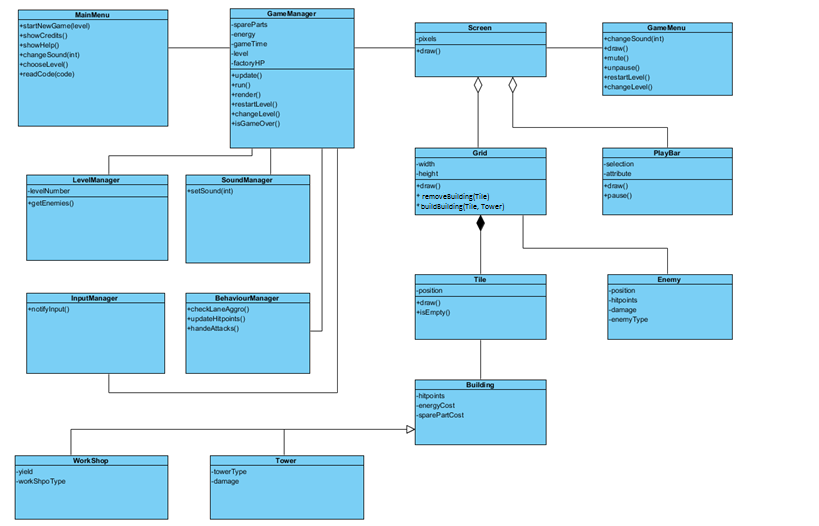


Figure3.1. The class diagram of the Game

**Explanations of Important Classes:**

* **MainMenu:** It is the class that handles the decisions that you make before starting the game such as selecting which level to play
* **GameManager:** GameManager is the main class that handles the game. It has many functions such as rendering of the game screen, updating the game objects and keeping track of the resources. It also contains other important manager classes.
* **SoundManager:** This class is responsible of generating all sound effects in the game.
* **LevelManager:** This class handles all mechanics that depend on the level that the user is playing such as when and which lane the enemies are generated.
* **InputManager**: The InputManager class listens to user inputs and notifies the game when the user provides an appropriate input.
* **BehaviorManager**: This class basically handles the behaviour of the towers and the enemies. It checks if there are any towers and enemies on the same lane that might be attacking each other. Then it refreshes the hitpoints of the entities.
* **Screen**: This class contains the elements of the game and it is responsible for helping the GameManager generate the graphics
* **GameMenu**: Also known as Pause Menu. This is the menu that appears when the pause button is pressed. It provides the user with options such as to restart the level. It then notifies the GameManager if the user makes a selection.
* **PlayBar:** This class handles all the actions of the player. It is the main way that the player interacts with the game. It provides options to build buildings, to destroy them and to pause the game. If the user pauses the game it notifies the GameManager and brings up the GameMenu.
* **Grid:** Grid is made up of tiles. It is where the enemies and towers are.
* **Tile:** This class is a container that might be empty or might be occupied by a building.
* **Building:** This class represents the actual buildings in the game. It contains valuable information about the building such as its cost and hitpoints.
* **Workshop:** Extends Building. It contains Information about the type of the workshop and its yield.
* **Tower:** Extends Building. It contains information about the Tower type and its damage.
* **Enemy**: Represents the enemies in the game. It contains Info about its position, type, hitpoints and its damage.
  1. **Dynamic Models**
     1. **State Chart**

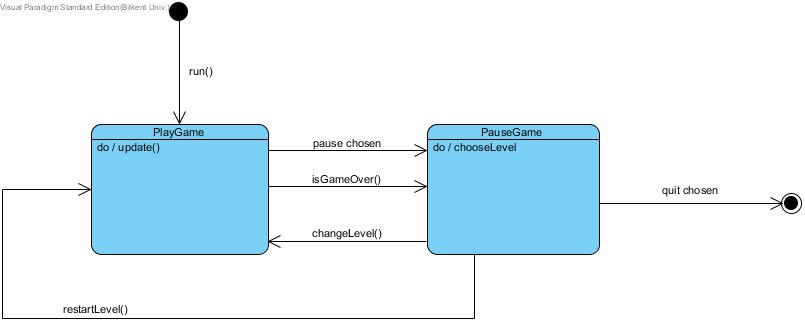


Figure 3.1. The state chart diagram of the GameManager object

GameManager is the most important object in the game. It manages the gameplay and helps the system function. The object has 2 states. When the player wants the game to start, the GameManager goes to the PlayGame state. In this state, the game is played by the player and the graphics are updated. If the game is paused by input of the player, by winning the game or if the game is over, the GameManager goes to the PauseGame state.

In the PauseGame state, the player can choose to restart the level, this will make the GameManager go to PlayGame state. The system will react in a similar way when the player wants to proceed in the game and change the level. The level is chosen in this state. If the player wishes to quit the game the GameManager object will end its states and the system will not require it until further notice.

* + 1. **Activity Diagram**

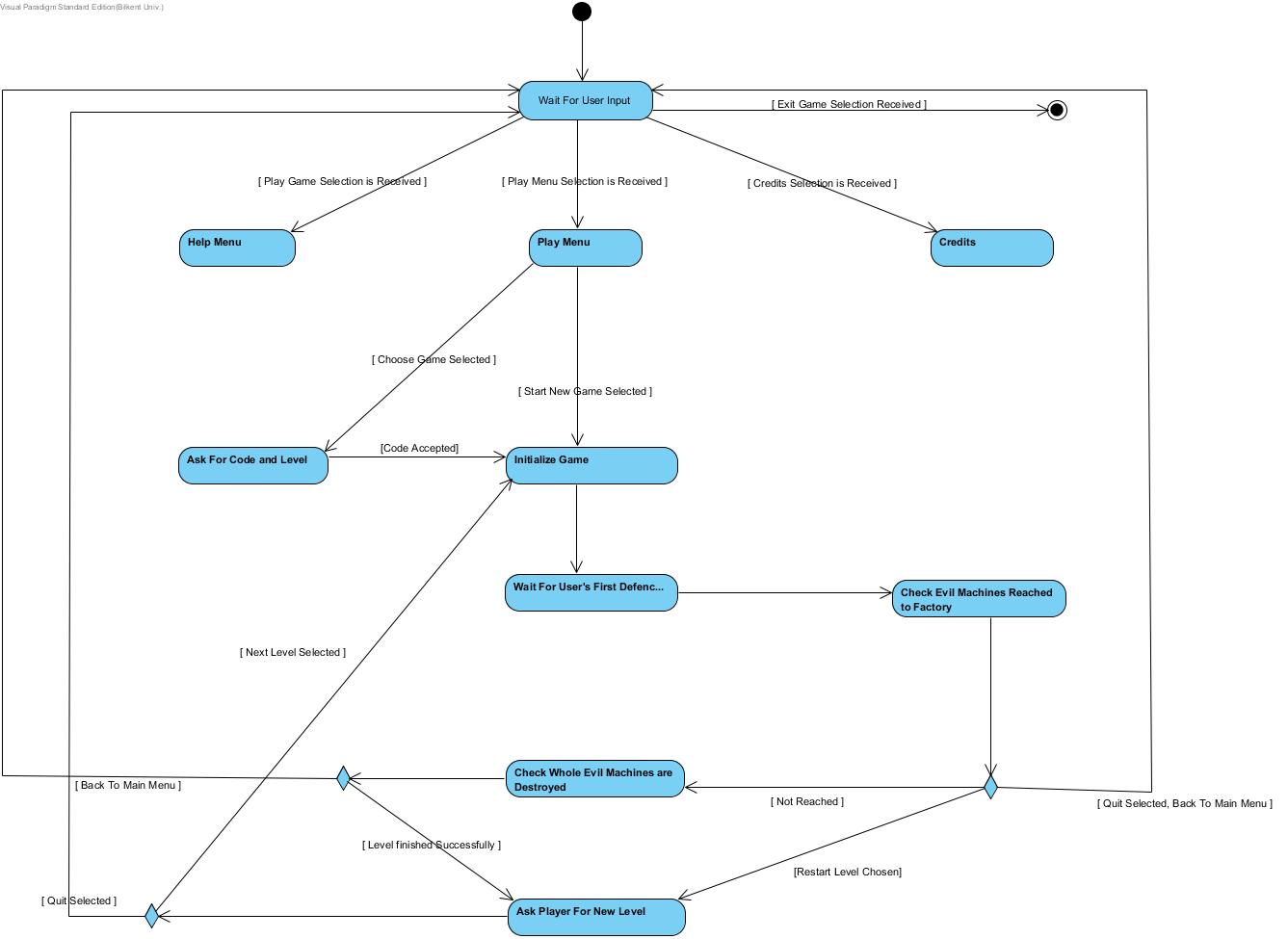


Figure 3.2. The activity diagram of Protect the Factory

When user selected Play Game selection, system redirects user to play game menu. In this menu, system presents 2 different options to user. When user selected Start New Game option, system automatically creates basic game objects and wait for user to place first defensive towers. After that process system starts to create random evil machines to attack towards factory. When towers destroy evil machines, user gets chance to build new towers against evil machines. System consistently checks whether evil machines reached to factory or towers destroyed the whole evil machines in the given time. If evil machines are completely destroyed, system finishes the level and ask user for next level. If any evil machine reaches and destroy the factory, system finishes the game and asks the user if he wishes to restart the game or quit to the main menu.

If the user selects Choose Level option in the Play Game menu, system asks the user for a code. This code is given to the player at the end of a level if he has completed it successfully. User enters the code to the given blank space. If the code is accepted by the system, the system gives the levels he can play to the user. User selects a level. The game is initialised by the system and starts from the level that the player wants to play.

* + 1. **Sequence Diagrams** 
       1. **Starting new game with a code**

In this scenario the player decides to play a specific level. Player presses “choose Level” button. The main Menu asks for the player’s code. Player enters the code. MainMenu displays which levels are available. Player chooses Level 3. LevelManager sets the level to 3. GameManager starts the game.

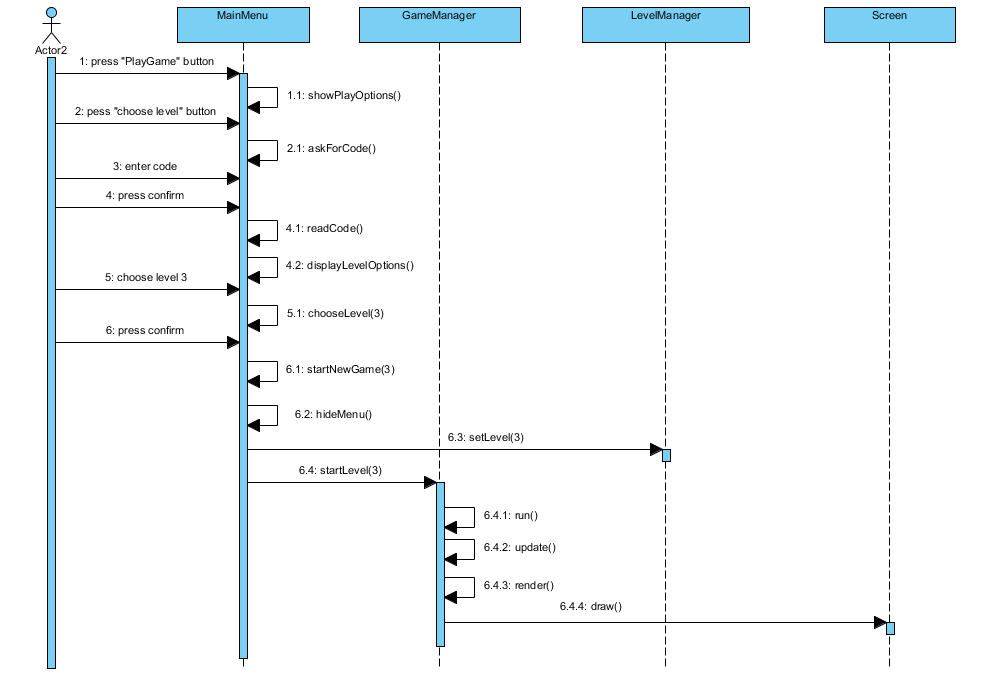
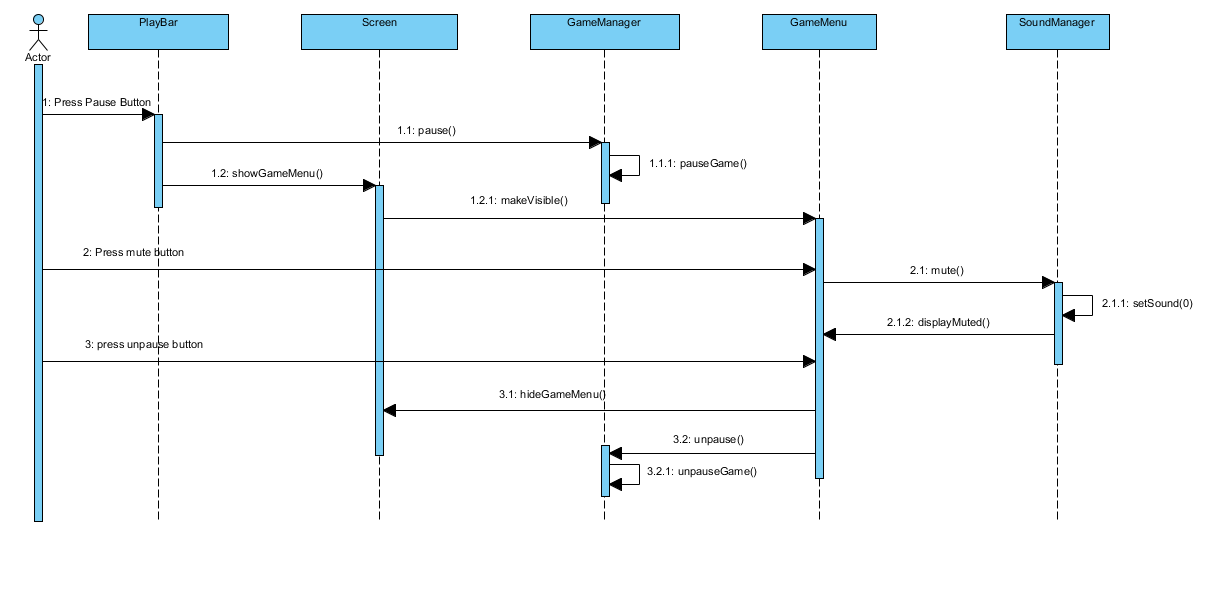
****

Figure 3.3: Sequence Diagram showing the player choosing to play level three with code

* + - 1. **Muting the game**

While playing the game, the player pauses the game and the pause menu pops up. Player presses the mute button. SoundManager is notified. The player unpauses the game.

Figure3.4: The sequence diagram that explains the process of muting the sound of the game.

* + - 1. **Building a new tower**

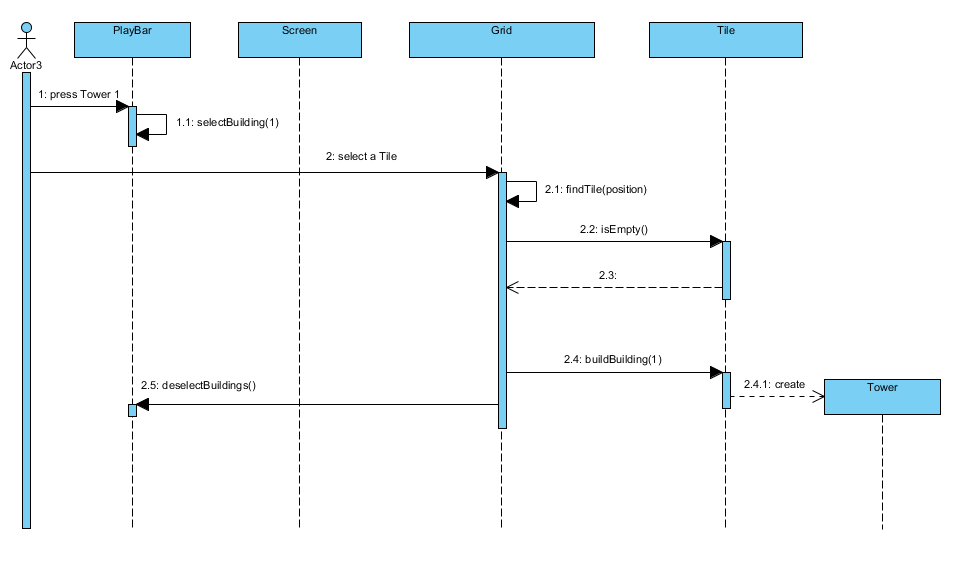
In this scenario the player clicks the tower icon on the play bar. Then he/she clicks a tile on the grid. The Grid Checks if the specific tile is empty then a new tower is created in that tile.

Figure 3.5: Sequence diagram showing the player building a new tower.

1. **Design**
   1. **Design Goals**

The design goals of a system are one of the most important steps in designing a system. The goals established in this step will allow easier decision making in the later steps of the system design. The non-functional requirements that were established in the requirement elicitation part will be used to address and choose the design goals.

* **Usability:**

One of the most important features for a game is its usability. An increased usability will allow the users to play the game comfortably and as a result be entertained from the game. One step for achieving this will be through a user-friendly game design. An easier user interface will help the user to play the game easily without confusion. For this the menus provided in the game will be simple and provide through but useful options. The menus in the game will have distinct features so that the player can identify whether he is at the start game menu or the pause menu. The in game menu is the basis of the game where the users choose turrets or workshops or see the latest status of their energy and spare part values. This menu will be have large and clear icons for the user to choose the right item. The status bars will be clear and visible so that the user can see them and act accordingly.

The user inputs that the game requires will be easy and simple. The main inputs will be through the mouse actions. The player will drag and drop the items with the mouse. The choose level part of the game will require keyboard control which will be a simple use consisting of entering letters for the code.

The game will also provide a Help option in the menu. This screen will provide knowledge about the gameplay and how to proceed in the game. This will help the player to learn about the game easily without any prior knowledge.

* **Robustness**

An important feature for any system is its ability to maintain errors and exceptions. The game will be tolerable to errors or exceptions. Routine checks will be made by the system in order to see if it is having exceptional errors. The system will also help the user against any possible error that they might implement during the game.

* **Response Time**

An important feature for any game is its ability to react to the user’s input as quickly as possible. The game will be designed so that it can react to the actions of the user in minimum amount of time. The game will have swiftly moving objects on the screen so that the user maintains his interest and is not confused. However, the game will wait before start of each game without any response to user input so that they can become ready for the game to start.

* **Portability**

The systems used in computer are becoming more portable and it has become an important issue to reach different users. This is made for the system to be useful in multiple environments. The game will be written in Java and thus, will use the Java Virtual Machine in order to run. The Java Virtual Machine provides a portability as it is functional in different environments. This functionality will be used by the game, so that the game can be played by users in different environments.

* **Extendibility**

Games must be updated frequently by the developers so that they can still be played. For this to happen the game will be designed to be extendable. This will allow the developers of the game to add new features to the game. New and harder levels can be added at future implementations to keep the user’s interest in the game alive. Also new turrets, workshops and enemies can be added in the future to offer the user diversity and change.

The game will also be extendable to be implemented as a mobile app in the future. As mobile devices are becoming more and more popular these days, having the game extendable to be a mobile app will have benefits. However, this will not be implemented now and will be left as a future goal.

* **Modifiability**

To have an extendable game and to maintain the game in the future, the game will be modifiable. The developers will not have trouble in extending the game with new features and modifying the game to maintain issues. This will be allowed by our Model-View-Controller (MVC) system architecture. This MVC approach will be useful in modifying as the main contents of change will be required in the controller part of the system.

**Trade Offs**

* **Functionality vs Usability**

The game is designed to be easy to use. This means that the usability of the system is a more important concern for us in this design. The game will not provide the user with complex functionalities that might make the game harder to play and make the user uncomfortable. Thus, the game will provide simple options to the user in the gameplay to not bore them with unnecessary details and functionalities.

* **Efficiency vs Portability**

As the game is written in Java and uses the Java Virtual Machine, it is very portable. However, this reveals an efficiency problem. The portability of the Java Virtual Machine takes longer time while running when compared with other languages. This causes problems in performance. On the other hand, our main objective in this game is to make it portable so that it can be played in different environments. Thus, portability of the Java language is chosen over efficiency.

* 1. **Sub-System Decomposition**
     1. **Overview of Component Diagram**

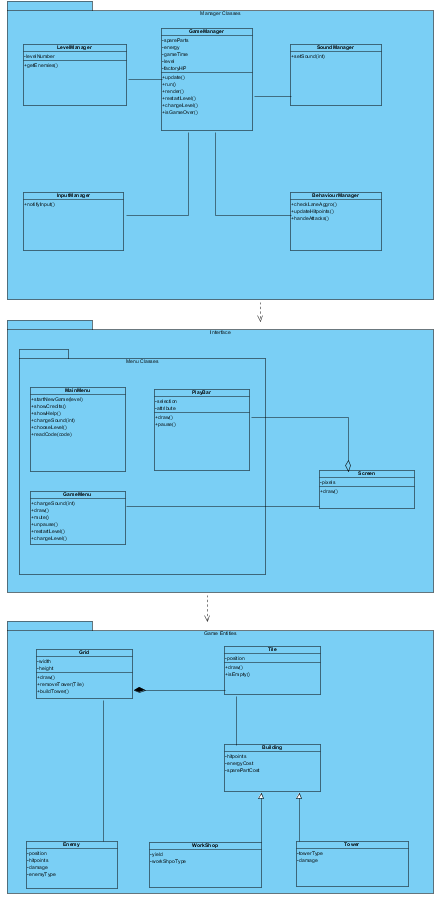
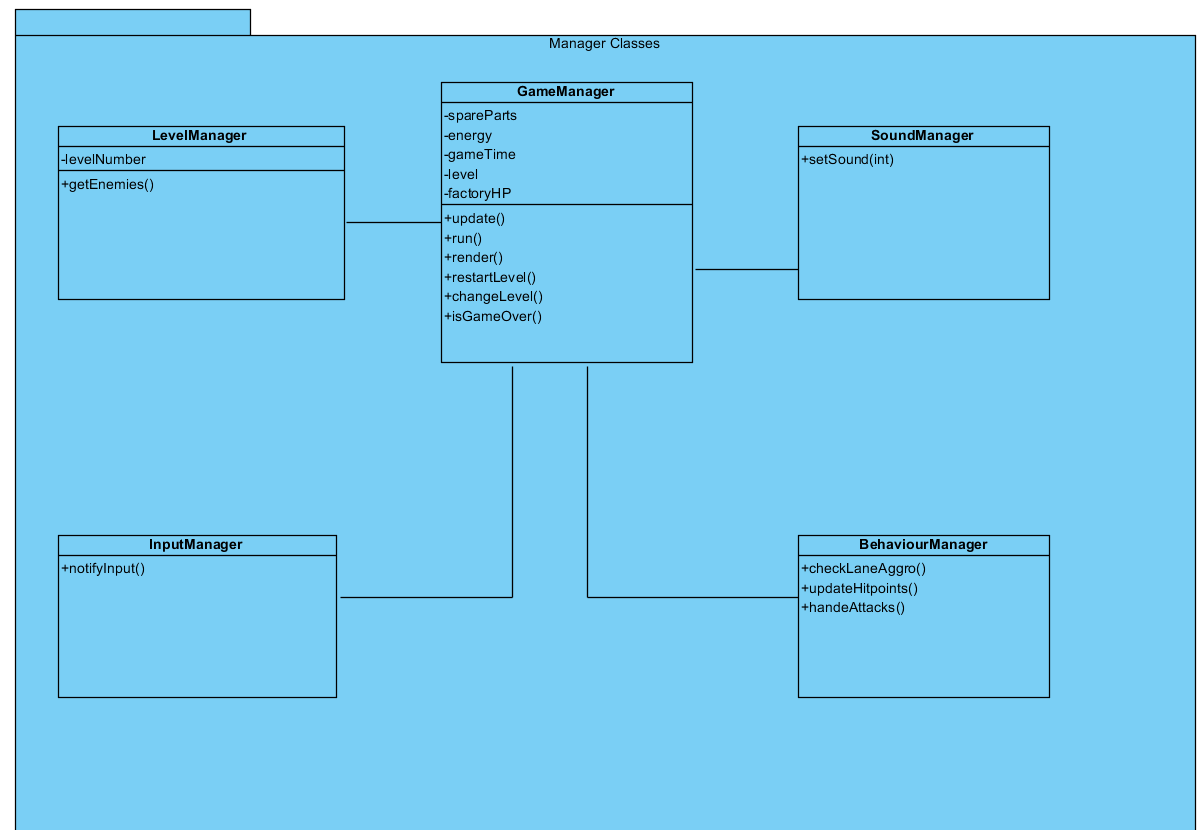
****

Figure 4.1. The general structure of the subsystems

The System is composed of three Layers. The topmost layer is Manager Subsystem which contains all manager classes. The middle layer is Interface Subsystem. It contains all classes that function as a user interface. Finally, the bottom layer is Game Entities subsystem. All layers are loosely coupled with the subsystem below them. The subsystems are arranged in a way that maximizes coherence and minimizes coupling.

* + 1. **Manager Subsystem**

****Figure 4.2. Detailed View of Manager Subsystem*.*

Manager Subsystem contains five classes. They are; GameManager, LevelManager, SoundManager, InputManager and BehaviourManager. GameManager class calls the draw method of Screen class which is inside the Interface Subsystem. Therefore subsystem has runtime dependency to Interface Subsystem.

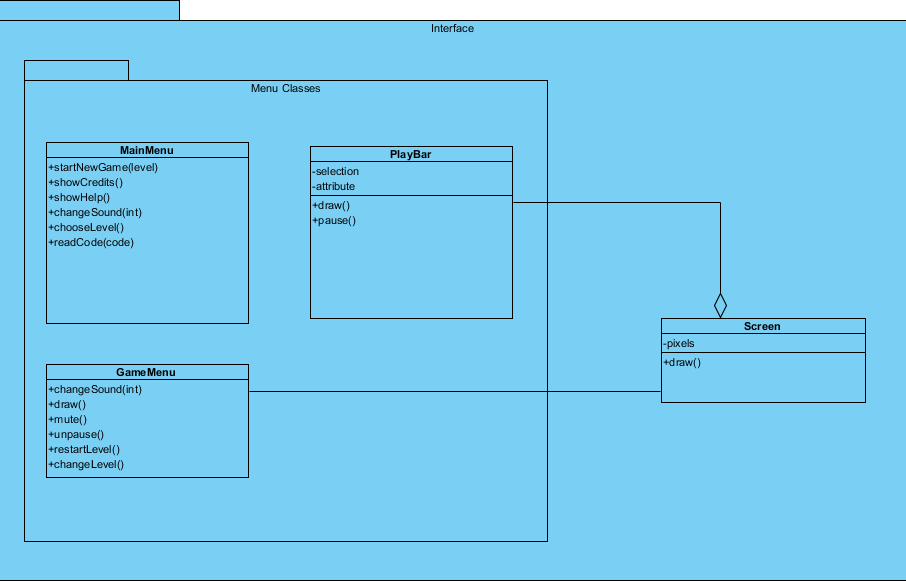
* + 1. **Interface Subsystem**

Figure 4.3. A detailed view of Interface Subsystem

The Interface subsystem contains four classes. MainMenu, PlayBar and GameMenu are inside another subsystem named Menu Classes which is inside the Interface Subsystem. The Screen class has compile time dependency to Menu Subsystem. The Interface Subsystem has runtime dependency to Game Entities subsystem.

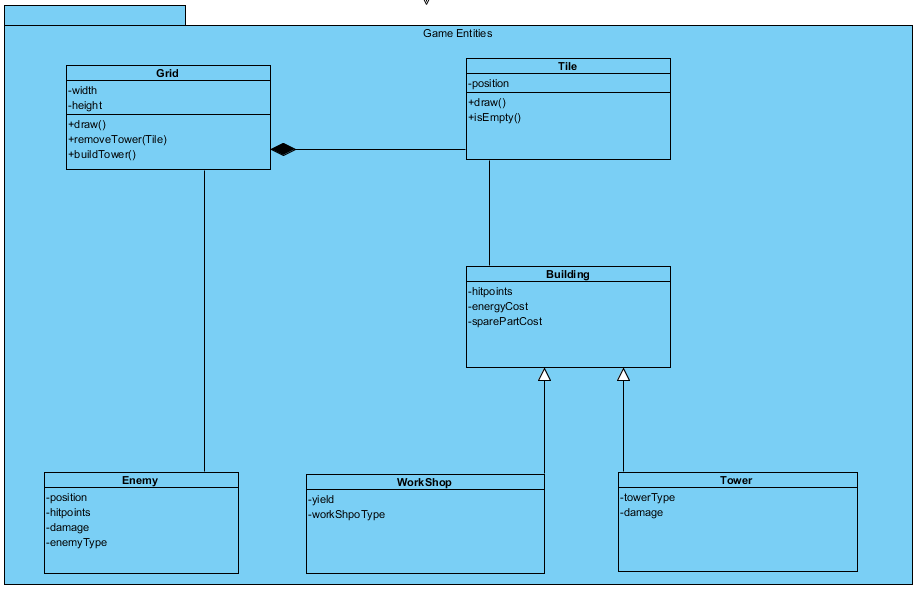
* + 1. **Game Entities Subsystem**

Figure 4.4. Detailed view of Game Entities subsystem

The Game Entities subsystem contains all the entity classes. These include Grid, Tile, Enemy, Building, Workshop and Tower classes. The subsystem is not dependent on any other subsystem.

* 1. **Architectural Patterns**

### **4.3.1 Layers**

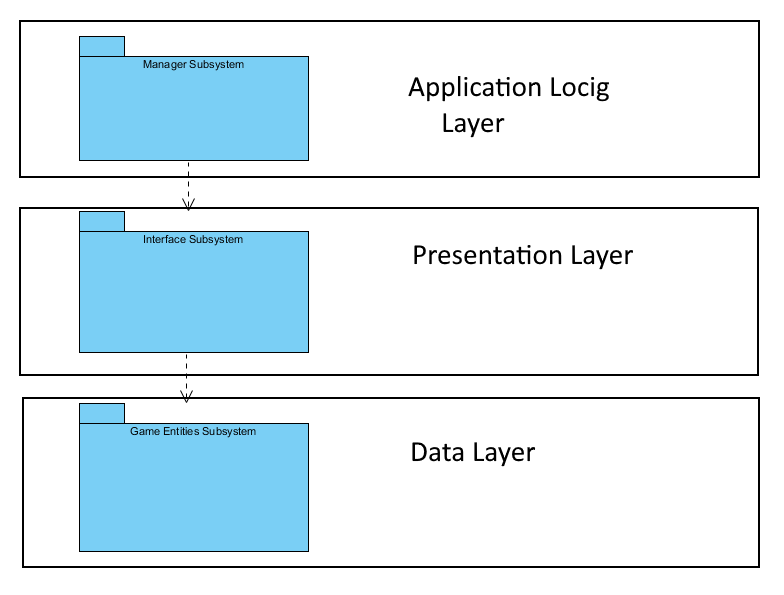
In our system we used three different layers. These layers show the relations and data flows between packages in the system. The first layer is Application Logic Layer. This layer consist of “Manager Subsystem” and can access the services of “Interface Subsystem”. Second layer is Presentation Logic Layer which includes “Interface Subsystem”. In this layer, “Interface Subsystem” is allowed to use “Game Entities Subsystem”. Last and the lowest layer is Data Layer which consists of “Game Entities Subsystem”. This layer can be accessed from the layer above. Following diagram shows the relationship between layers.

Figure 4.5. Layered structure of the program

### **4.3.2 Model View Controller**

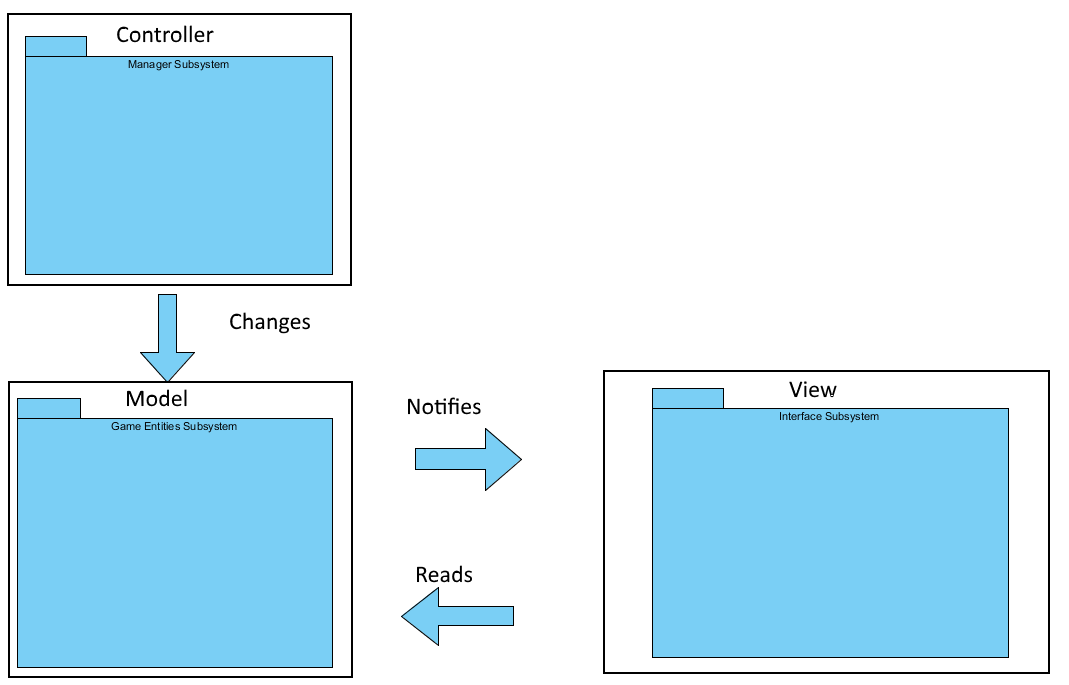
In system design of ProtectTheFactory, we use ModelViewController pattern to connect the subsystems to each other appropriately. When some operations (such as login, select level, add tower etc.) are used, controller part of the system which is “Manager Subsystem” is updated. After that, controller updates the model parts of the system which is “Game Entities Subsystem”. Then, model notifies the view part of the system which is “Interface Subsystem”. The view parts get data from the model parts and views are updated. The following diagram shows the relations between model-view-controller parts.

Figure 4.6. Model- View – Controller structure of the program

* 1. **Hardware/Software Mapping**

The game requires simple input devices which are a mouse and keyboard for the user to control the game. The user will control the mouse to play the game and choose from the menu. The keyboard will be used to enter the code to choose a level. The deployment diagram for the Personal Computer (PC) can be seen below:

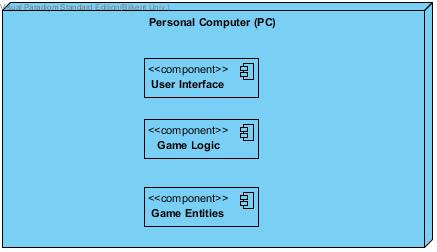


Figure 4.7. Hardware Software mapping deployment diagram

A computer with decent capabilities and a functioning operating system will be enough to run the game. The graphics of this computer will also be enough to screen the visuals of the game. The computer will need to have a Java Environment pre-installed in order to run the game as it only works on a Java Virtual Machine. This Java Runtime Environment must be at least version 7.

No internet connection is required for the game as it does not have any functionalities needing an internet connection.

* 1. **Addressing Key Concerns**
     1. **Persistent Data Management**

The game does not require any database systems as the game data and progress are not stored for future use. Moreover, the database system is not required as the game needs a real-time access to the data in memory. Some features in the game like the attributes of the in game objects will be stored as file systems in the computer’s hard disk in order to make them persistent. Their persistence is important as the game will require them in each execution with the same data. The game logic will access these files if necessary to acquire the data required. These will be file instead of databases as they are written once but read multiple times.

* + 1. **Access Control and Security**

The game does not require any authentication from the user to run and start the game. As the only user type is a player, no user info or data will be needed. Moreover, there will be no access limitations as the player can use every functionality offered by the game.

However, our game has a choosing level feature which asks the user for a code. This code allows the player to choose a level. This code is the only authentication scheme asked by the system. This allows the users with the right authentication code to enter the choose level menu. If the code entered by the user is invalid they will not be allowed to choose levels in the game. These codes will be stored with the files mentioned in the Persistent Data Management Part and the controlling system will enter this file and check if the code entered by the user is valid or not.

* + 1. **Global Software Control**

The game has a decentralized software control system. Although the GameManger object exists in the game some simpler tasks are implemented by other control objects. The inputs to the system are addressed and controlled by a separate object and the user does not directly communicate with the GameManager object. The visuals of the game are also controlled by a separate object. The GameManager is not the only control object and this helps the game to become more efficient as all the workload is not implemented by a single class. However, as the control is spread out in the system, the interactions between these control objects must be implemented carefully and function properly during run time for a good system.

* + 1. **Boundary Conditions**

**Initialization:** At initialization, all classes at interface and Manager Subsystems are created an initialized. Also the Grid and Tile classes are initialized. The sprites are loaded from a sprite sheet file. Then the threads are started. Main Menu is shown at the start. It has buttons that serve to access the main functionalities of the game. See figure 3.3 of analysis report showing the sequence diagram showing the player choosing to play level three with code.

**Termination:**  During termination only the GameManager Class is terminated. It contains the references to all other classes therefore garbage collection of JVM terminates all other classes. The game does not store data externally (other than sprite sheet) therefore no extra measures are taken to preserve data.

**Failure:** In case of a failure the game is terminated. All data except the sprite sheet is stored in sourcecode and sprite sheet is not modified in any way therefore no data recovery function is required. If the game cannot find the spritesheet file during initialization, an exception is thrown. This causes an error message to appear stating that the file is missing. The error message also contains information on how to resolve this issue by relocating the spritesheet file to the right directory.

1. **Object Design**
   1. **Pattern Applications**

Design patterns are used to solve common problems found in application designs and implementation. They are templates on how to solve these problems through the interactions of classes. In our “Protect the Factory” game we have chosen to use the Adapter, Singleton, Façade and Bridge design patterns.

**Adapter Design Pattern**

Adapter design is used to allow different classes with similar interests to work together. This allows the users to reuse some of the classes that were implemented beforehand. This is useful in this application as we want to save from time and reuse some of the classes that we have created.

This design pattern will be used to adept the different menu classes to each other. Instead of writing a different classes for each other, these menu classes will be adapted by reusing a single menu class for all them. It is not difficult to introduce this to our implantation as most of the menus in our game share similar traits and functions. This adapter pattern will be implemented as seen in the class diagram below:

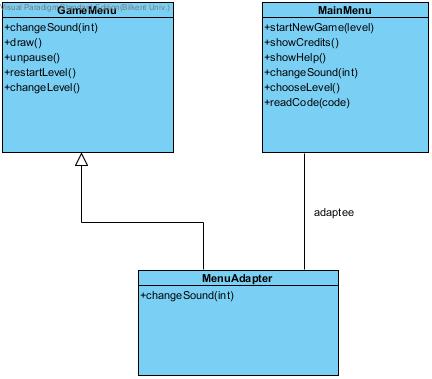


Figure 5.1. The changed adapter design pattern relationship of the menus

**Singleton Design Pattern**

The singleton design pattern is used to guarantee that only one instance of a class will be created in all of the program. This is important as some classes are managing the whole of the game and there must be a single of them in the program. If there are more than one of these classes in the program, there can be ambiguity in the execution of the program that might result in unintended problems.

This pattern will be used for the manager classes found in the game as there must be a single of them in the system. This pattern will be achieved by making the following code modifications to the implementations of the manager classes for the GameManager class:

public class GameManager {

private static GameManager gameManager = new GameManager();

private GameManager(){

//the implementation of the constructor

}

public static GameManager getInstance(){

return instance;

}

//the implemenation of the GameManager class

}

**Façade Design Pattern**

Façade design pattern is commonly used to implement a common interface for a group of classes or a subsystem. This allows these classes to be more cohered and less coupled. This pattern is commonly used to implement the relationship between the unique subsystems of a program.

In this application, the façade design pattern will be used to form the communication between each individual subsystem which are the Manager Classes, the Interface and Game Entities. Through this communication the Model-View-Controller architecture will also be implemented as each of these subsystems will communicate through one channel and will be grouped separately. Instead of implementing separate façade classes for each of these subsystems, the most used classes of each subsystem will be implemented as façade classes. The GameManager class will be the façade class of the Manger Classes, whereas the Screen class will be the façade class of the Interface subsystem. These classes were chosen as they were the main classes that had interactions with the other subsystems.

**Bridge Design Pattern**

The bridge pattern is used to differentiate the implementation and the interface of a class. This is useful when the implementation of a class can change in time or depends on varying conditions.

In this application the bridge pattern can be used for the behaviour manager which controls the actions of the characters in the game. The functions like hit and move can have different implementations. This pattern will not be directly implemented in this application as there are not varying implementation algorithms for these functions. However, the bridge implementation will be used in the future. This pattern will be useful for extending and changing the implementation of these functionalities.

**Other design pattern**

The observer design pattern is also visible in the application as there is a frame that is showing the energy and spare-part level of the player. For this frame the actions in the game are observed, however as there are not multiple frames that observe these actions, there is not a need for a complex observer design pattern.

* 1. **Class Interfaces**

**Enemy.java :**

****

Figure 5.2. The enemy interface

Enemy class models the properties of our evil machines. It has x,y coordinates, hitpoints, damage and EnemyType. Its constructor takes one parameter (int level). This level goes to enemy type object. And this object returns which enemy type is. According to enemy types, Enemy’s constructor determines enemy’s hitpoints and damage.

Also this class has public methods which are void move(),getDamaged(int dmg), boolean isAlive(),int attack(boolean collision), void destroy() and getter, setter methods.

**EnemyType.java :**

It is child of Enemy class. There are 3 enemy types and this class determines random enemy type according to level.

**Tower.java :**

****

Figure 5.3. The tower interface

This class extends Building.java class. Tower class is a building that damages evil creatures in its path. It has damage, tower type and mutual building properties such as hitpoints, sparepartCost, EnergyCost. Its constructor get one parameter (int towerType).This parameter determines tower’s damage, hitpoints and costs. Public methods are setter and getter methods.

**WorkShop.java :**

****

Figure 5.4. The workshop interface

It extends Building class. WorkShop is a building that has yield, type and building properties. As a constructor, public WorkShop(int shopType) it gets one parameter to determine which workshop has chosen.

Public methods are void setYield(int yield), int getYield().

**Building.java :**

****

Figure 5.5. The building interface

It is a parent class. It has some properties of Tower and WorkShop classes such as hitpoints, energyCost, sparePartCost. Public methods are setter and getter methods to reach its properties.

**Tile.java :**

****

Figure 5.6. The tile interface

Tiles are squares that make up the field. Only one building can be built on a tile but multiple enemies can be on the same tile. To set tiles in Grid class there is position properties. It has public method isEmpty() to check whether a tile has a building or not. Also it has draw() method to draw invisible squares in specified location.

**Grid.java :**

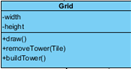
****

Figure 5.7. The grid interface

It is made of tiles. Each grid has 8 tiles and the game made of 5 grids. There are ArrayList<Tile>, width, height as its properties. It has public methods such as removeBuilding(Tile), buildBuilding(Tile,Building) and draw().

**PlayBar.java :**

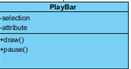
****

Figure 5.8. The playbar interface

This class handles all the actions of the player. It is the main way that the player interacts with the game. It provides options to build buildings, to destroy them and to pause the game. It has public pause() method that pause and open GameMenu panel to the user.

**GameMenu.java :**

****

Figure 5.9. The gamemenu interface

It extends JPanel. It is activated while the user press pause button. It offers the user some public methods such as changeSound(int), mute(), changeLevel(), restartLevel(), unpause().

**Screen.java :**

****

Figure 5.10. The screen interface

It extends JPanel. It draws Grid, PlayBar and GameMenu on a screen. All necessary Towers, Enemy and WorkShops be showed in Screen Class. And this Screen class is updated by GameManager class while game continues. It has the paintComponent(g: Graphics).

**LevelManager.java :**

****

Figure 5.11. The levelmanager interface

It manages level. Its property is levelNumber. It uses getEnemies() public method that calls enemy class according to level.

**SoundManager.java :**

****

Figure 5.12. The soundmanager interface

It plays music while the user plays the game. Its methods are setSound(int level), playMp3().

**InputManager.java :**

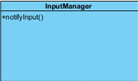
****

Figure 5.13. The inputmanager interface

Takes level code from the user and notify this code. Public method is notifyInput().

**BehaviourManager.java :**

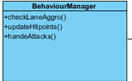
****

Figure 5.14. The behaviourmanager interface

This class controls and updates enemy and tower reactions. It is used in Game Manager class. Public methods are checkLaneAggro(), updateHitPoints(), handleAttacks().

**GameManager.Java:**

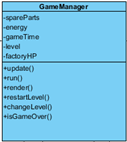
****

Figure 5.15. The gamemanager interface

It containsnecessary properties to make game playable are spareParts, energy, gameTime, level, factoryHP. It contains public methods such as update(), run(), render(), restartLevel(), changeLevel(), isGameOver().

**MainMenu.java :**

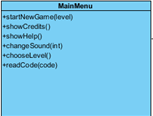
****

Figure 5.16. The mainmenu interface

It extends JPanel. This class is the bridge between other panels while the user clicks desired buttons. Its public methods are startNewGame(), showHelp(), changeSound(int level), chooseLevel(), readCode().

* 1. **Specifying Contracts**

1. context SoundManager::setSound(i: int) post: sound = i
2. context MainMenu::startNewGame(lvl: int) pre: GameManager.isGameOver() = true
3. context MainMenu::startNewGame(lvl: int) post: GameManager.isGameOver() = false
4. context MainMenu::startNewGame(lvl: int) post: GameManager.level = lvl
5. context GameManager::run() pre: gameTime = 0
6. context GameManager::restartLevel() pre: isGameOver() = false
7. context GameManager::restartLevel() pre: gameTime > 0
8. context GameManager::restartLevel() post: gameTime = 0;
9. context GameManager::restartLevel() post: isGameOver() = false
10. context GameManager::restartLevel() post: @pre.level = level
11. context GameManager inv: level > 0
12. context GameManager inv: sound >= 0
13. context Grid::buildBuilding(t :Tile, b: Building) pre: t.isEmpty = true
14. context Grid::buildBuilding(t :Tile, b: Building) pre: GameManager.energy - b.energyCost >= 0
15. context Grid::buildBuilding(t :Tile, b: Building) pre: GameManager.spareParts - b.sparePartCost >= 0
16. context Grid::buildBuilding(t :Tile, b: Building) post: t.isEmpty = false
17. context Grid::buildBuilding(t :Tile, b: Building) post: GameManager.energy = GameManager@pre.energy - b.energyCost
18. context Grid::buildBuilding(t :Tile, b: Building) post: GameManager.spareParts = GameManager@pre.spareParts - b.sparePartCost
19. context Grid::removeBuilding(t :Tile) pre: t.isEmpty = false
20. context Grid::removeBuilding(t :Tile) post: t.isEmpty = true
21. context GameMenu::changeSound(i: int) post: SoundManager.sound = i
22. context GameMenu::mute() post: SoundManager.sound = 0
23. context Building inv: hitpoints > 0
24. context BUilding inv: energyCost > 0
25. context Building inv: sparePartCost > 0
26. context GameManager inv: factoryHP >= 0
27. context GameManager inv: energy >= 0
28. context GameManager inv: spareParts >= 0
29. context GameManager::restartLevel() post: energy = 0
30. context GameManager::restartLevel() post: spareParts = 0
31. **Conclusion**
    1. **About the Game**

Protect the Factory is a desktop based tower defence game created for the Windows operating system. It is inspired by the game “Plants vs Zombies”. The games objective is to protect a factory from evil enemies that are attacking it. The player can build towers and workshops that help him or her to succeed in protecting the factory from these attacks. The game has different levels with increasing difficulty that can be played by completing the previous levels and receiving a code.

* 1. **The Design Process**

The game’s design process was made of three main stages that produced three reports. These steps were the analysis of the system, system design and the object design. These steps were required to make a successful and easy implementation of the application.

In the initial stage of analysis, we started by defining the requirements of the system. These were made of the functional, non-functional requirements and constraints. We later stated use-cases and scenarios that allowed the system to perform these functional requirements. In this part we also created mock-ups for the user-interface of our game to give us an idea about how it might look. In this stage we also created the object and dynamic models of the system. The object models and its consisting classes were created by the use of the scenarios and the actors inside it. The dynamic models consisted of the activity, state and sequence diagrams of the system. These were created by using the actions in the scenarios stated.

In the next stage we worked on the system design of our system. This stage was started by the acknowledgement of the design goals of the system. These goals were chosen by the help of the non-functional requirements stated in the analysis stage. The system was also decomposed into simpler subsystems that would allow it to be easier to implement and understand by dividing it. The architectural pattern was also chosen in this stage as the Model-View-Controller. We also addressed other key concerns like the Hardware/Software Mapping, Persistent Data Management, Access Control and Security, Global Software Control, and Boundary Conditions.

In the final stage that can be seen in this report we worked on the object design of the system. We first chose design patterns to have an easier implementation, the patterns we chose were Adapter, Singleton, Façade and Bridge. We later derived the interfaces of the system by defining them and explaining their functionalities. In the final stage of this level we worked on the contracts of the system by defining them with the OCL language.

* 1. **Lessons Learned**

One of the first lessons we learned in this project was the importance of the design process of a system. The implementation of a system becomes far simpler by addressing these steps. We were also able to understand our system more thoroughly with these steps as they allowed us to go deeper in its structure. These all helped us to understand design’s importance.

The second lesson we learned that design is an extremely unstable process and varies frequently. We made many changes in our reports in each step and we believe there will be even more changes in the implementation process. There were many classes introduced and removed from the system in these steps that showed how frequent the change was.

We were also able to learn about object oriented programming. This will be very useful for us as the most frequently used type of programming is this. We were able to put on our knowledge from the previous semesters by becoming more interactive with this concept.

We also learned how to derive a problem into solution. We can use this in our work life as this helped us to gain experience on this matter. This projects also helped us to learn how to work as a group and how each member is an important part of the work. We believe that we will not have a hard time in adapting to a work environment through this process.

Finally, we learned about new tools that are used in the design process. We learned about Visual Paradigm and drawing class diagrams with this tool. We also extended our knowledge on Java by using it immensely. We also learned about design patterns that we can use in our future work life and projects. OCL language became familiar to us as well in this process.