Bilkent University

Department Of Computer Engineering

Object Oriented Software Engineering Project

Academic Warfare : The Conflict in Bilkent

Design Report

Doğukan Yiğit Polat, Selin Fildiş, Yasin Erdoğdu, Onur Elbirlik

Instructor: Uğur Doğrusöz

Design Report

Nov 28, 2016

Contents

Introduction

Purpose of The System

The purpose of our system for the users to have fun by playing our game, Academic Warfare: The Co-nflict in Bilkent which is a tower defense game. Tower defense games function with waves of enemies coming to a base which the user has to defend. In every wave the enemies become harder to defeat therefore the user has to develop strategies in order to pass through the waves. If the base takes too much damage the game will end.

Design Goals

Our design goal is to develop a smooth, scalable, conveniently-designed and well-engineered computer game that serves fun to its players. We tend to design every part of the system in a complete object oriented manner. Furthermore, we want to keep every subsystem of the project simple, understandable and easy to manage. All of the building blocks and modules should be modified and upgraded without any problems. ￼

Reliability

The game must be reliable as possible as it would need to function smoothly to impress the user and reduce any bug fixes after the release.

Modifiability and Readability

When the game needs to be modified (i.e. updates to the gameplay, new characters, new weapons, new bonuses, new enemies) the code must be understandable, readable and modifiable as it would be hard to re-write the code. To ease modifiability, the design will focus on inheritance and polymorphism.

Maintainability

Although trying to optimize the game so it doesn't have any bugs, we realize that this is not possible. So the code must be maintainable for future bugs and for future adaptability to new environments and upgrades.

Ease of Use

The game will consist of GUI's which every user will understand easily and the users will play the game easily.

Response Time

As Academic Warfare is a game that needs immediate response, the an optimization will be made according to that as well.

Software Architecture

Overview

In this section, the software architecture system of our game project has been decomposed into subsystems to provide maintenance. The essential goal of this decomposition is to reduce the redundancy among subsystems and improve interaction between components of subsystems. The other main goal is to provide convenience and clarity for designing the game

Subsystem Decomposition

The system decomposition is shown on the right. Our decomposed system consists of AcademicWarfare our main packet with three sub packets GameMechanics, UserInterface, Peripherals respectively.

The classes of the UserInterface packet are built to provide an easily understandable, simple des gn. The other packets are independent from the UserInterface class and purposes to enable functionalities of the game.

The UserInterface Package the components of the GUI will be implemented. This and it's subpackage graphics will contain the details of the functionality of the GUI and the screens for the GUI as well. This package will communicate with the GameMechanics package to provide a UI to the game. The GameMechanics package will contain the game mechanics such as the game engine. This package will also be in touch with the peripherals class.

Persistant Data Management

For our game, the only modifiable data is the game saves as it will be done by the user. The save games will be kept as a binary file writen using the "Serializable" interface of Java. In case of corruption in the game data, the user will need to restore the game files again.

Hardware-Software Considerations

We will use JAVA programming language as for implementing the main design of our game project. JDK and J2SE platforms will be used to develop the program and required JRE package will be used to run our Java application as stated. Our Java application will be available for Windows,Linux and Macintosh systems. Since academic warfare has no multiplayer option, it will be executed in one computer at a time. An executable file and some required other files will be enough to execute the our program. Player will use mouse to interact with game and play the game. Keyboard will be used only to enter the name of the highscores.

The game saves will be recorded as a binary file using the "Serializable" interface mentioned above.

Axcess Control and Security

Academic Warfare is fully offline, does not require any online authentication and no exploitations exist in regards of user data therefore there won't be any problem in regards of stealing user data.

Boundary Conditions

Inıtilization

The user must have the JRE installed in their computer to run the game as the executable file is in .jar format.

Gameplay

The mockups in the analysis report represent the GUI of the game as it is a boundary for us.

Termination

The game will exit when the user clicks exit game in the main menu, or the red X in the upper right corner.

Peripheral Modules Management

The game will have peripheral modules used for host system related interactions such as file system accesses, sound or system time. File system accesses are required for storing sounds, textures, sprites, configuration files, game state files and high score files. Sound driver should also be accessed properly in order to play sounds of the game properly. System time is used for synchronization purposes and leaderboard entry registration. High scores will be kept in a formatted text file, game states are stored as java object binaries. Game assets such as required graphics and visuals are loaded from the file system as images. Together with graphics, game sound and music data are also loaded from the file system. They will be encoded/decoded as usual sound files so we can utilize internal Java libraries for handling sound files.

Subsystem Design

Game Logic Design

Overview

In this chapter, with the help of the class diagram, classes of our project will be explained in detail. Trivial methods such as set and get methods will be omitted, since those will be added at implementation stage.

Our game engine initializes with the flow of information from GUI components such as difficulty of the game, current level, name of the player and so on,With this main relations between components starts.

Game engine is the administrative component of the logical structure. Game engine and game map communicates for flow of information about the surroundings of the player with respect to players input. Then the engine decides what to do with those information available. For example, if player tries to place a weapon to somewhere in a map, game engine takes the information of corresponding coordinate from game map. If couple of enemies have already reach to end,engine corresponds to that with decreasing the health points of the base. Also we have a collision mechanic in our game. We solved this issue by designing the game map as a grid map. Every movement of any dynamic object occurs as discrete grid coordinate movements with respect to our grid map design. So we can check any collision by checking coordinate checks,easily. If an enemy's bullets current coordinate overlaps with towers coordinate, towers health point will be decremented. We planned to use grid map design not just for simplifying complex problems, also since our Academic Warfare game is very suitable for this design choice.Most of the tower defence games are using this kind of design too. It makes it possible to increase the smoothness of animation like view of the moving objects. Java is capable of doing these kind of things.

Game Time Procedures

A game program is supposed to update itself regularly at a certain frequency. This frequency should be smaller or equal to the screen refresh rate of the system so that game can respond and be viewed properly. So game engine will repeatedly run an update routine. Once it is provided that game is updated with real time interactions of the player, development process can concentrate on basic game mechanics.

Game mechanics involve player input handling and processing interactions between game objects. Player can interact with the game objects with mouse input and objects will interact with each other depending on the rules that are set by provided game mechanics such as physics, events etc.

Update routine should not block input interrupts. Everything related “with game should be concurrent. So it will run on a separate thread. There will be several concurrent threads. Another thread will handle player input and update input related memory. Update routine involves combining various real time changes such as player input, screen refresh, physics. Once update routine makes sure that all required event handling procedure is completed properly, it will safely combine the changes and wait for a new update call.

Anything related with a game frame should be updated synchronously for avoiding race conditions and false event processing.

Movements of the game objects are provided by the PhysicsManager class. By default, all game objects can have a velocity. PhysicsManager class will look for all moving objects that have a velocity, it will update their positions according to their current velocity vector. Once the coordinates are updated, DisplayManager will draw them to their new coordinates when the new frame refresh event happens.

There will be other more specific interactions between objects such as a weapon firing to an enemy and making the enemy lose health. These interactions are controlled inside the GameEngine class according to various Event classes such as an enemy being inside the range of a weapon. Such Event condition will trigger the functionality section of that particular event and make the weapon fire to that enemy.

Detailed Class Design

- **Game Engine:**

The central executive class of the project is Game Engine class. It manages all game-dynamics and operations between classes. It coordinates interactions between classes. In any games, Game engine is designed to be unique class to controls the whole game.

Properties:

“*gameObjects*” includes any objects (player, enemy, weapon, tower, powerUp, path, tile) of the GameObject class. It means that after GameEngine instance is created, all objects in GameObject class are created and they exists until the game is over.

“physicsManager” controls the motion of enemies and collisions. This means that from the enemies appear in the screen until they are killed, physicsManager object updates the positions of enemies and weapons’ directions. It also checks the collision of enemy and bullet, then, if bullet hits the enemy, it gets collision “displayController” is responsible for drawing objects. After the GameEngine is instantiated, displayController object is created and update the screen according to the render time.

“inputController” is object which is responsible for taking events tom keyboard and mouse. It gets current input from the user.

“gameEngineThread” run always during the game in order to update interaction between game objects.

Methods:

“run” is used for initializing every objects in GameEngine class. It works only the beginning of the game execution.

“update” is used after run method is executed and it updates every object status through game execution until the game is finished.

In addition to above declarations, the size of code for GameEngine class will be much more than our expectation. Therefore, there may be some modification about this class during implementation.

- **Vector2**

2D vector implementation containing 2 public float variables x and y.

- **GameObject**

GameObject is a fundamental generic class in which all game related objects can be extended from.

Properties:

Position is a Vector2 instance. It is to determine the position of a game object on the screen.

Size is a Vector2 instance. It şs to determine the size of the game object to be created and displayed.

Velocity is a Vector2 instance. It also is a Vector2 and it contains information about the game object’s movement which is to be handled by PhysicsManager class.

Texture is an Image instance and it is the image of the game object. For example an enemy’s Texture will be instantiated from something like enemy.gif.

Methods:

Getter and **Setter** methods for above properties.

drawEntity(Graphics g) for defining how that game object will be drawn on the screen.

- **Event <<interface>>**

Event interface is for providing conditional triggers to the GameEngine. For example, weapons will shoot to the nearest enemy if there is an enemy inside their firing range. This conditional action is defined through a class that is implementing Event interface.

- **Player**

Player class is another game object in the design. It is created at the starting game and it exists until game is over. İt is unique object and starts with given budget.

Properties:

“budget” holds the earning money during the by killing enemy. It has default value before starting game and it increases after any enemy is killed during the game.

Methods:

“decreaseBudget” is function which is responsible for increase the current budget by spending money to buy weapons.

“increaseBudget” is a function which is for increasing budget with earning money by killing enemy.

- **Enemy**

This class is a parent class for different enemies types. It includes common features of different enemies such as health, path and maxSpeed.

Properties:

“health” is health tube which is responsible for the number of lives of the enemy. Coming to end of the tube means that enemy is dead and it’s score is added the current score of the player and meantime, it will be immediately cleaned up from game.

“path” is property which is shared commonly by all the enemies because there will be one path for enemies in order to reach tower.

“maxSpeed” is responsible for the speed of motion. This property can differ within enemies. During the game, this feature of the enemy can be decreased by hitting some weapon in specific time during.

- **Weapon**

There will be four different weapons in the game. This class is parent class for all of them. It includes features of weapons which are different for each weapon type.

Properties:

“damage” holds the amount damage which weapon has. When the weapon hits the enemy, its health will be decreased by amount of damage of weapon.

“fireRate” which is responsible for rate of weapons’ fire. It shows that how many fires a weapon can do per second.

“cost” holds the cost amount for buying a weapon during the game. There are four cost amount for different four weapons.

“unlockCost” holds the cost amount to unlock a weapon type for the first time. There are four unlock cost amount for different four weapons.

- **SpecialWeapon**

This class is for weapons which damage enemy in different ways. There will be special weapons which does not just decrease the health. For example, one special weapon will not decrease enemy health, it will slow down enemy motion for a specific time.

Properties:

“effect” is a event properties which determines action which will be performed.

- **Path**

This class is responsible for motion way for the enemies. According the game map, it can differ from each other.

Properties:

“points” is an ArrayList instance holds the coordinates as Vector2 objects for indicating a way which is from starting point to tower.

- **Tile**

This class responsible for checking any grid cell in the game screen. It is controls any cell whether it is appropriate for weaponizing or it includes blocks.

Properties:

“isBlocking” is responsible for determining block cells in game screen. İt is true if the grid cell is used for block.

“isWeaponizing” is responsible for checking the given cell is used for weapon. If the cell is used for weaponizing, it returns true.

- **Tower**

This class is used for fulfilling main aim of the game. Since, the main of the game is defensing tower against from enemies attacks. It is responsible tower game object which player defends.

Properties:

“health” holds the number of lives of the tower. When it is hit by enemy fire, health will be decreased. İf it reaches zero, it means that game is over.

- **PowerUp**

This class is responsible for performing action when some situations are encountered. For example, if the enemy is come to the tile which includes mine in that cell, this class performs a explode action and kill the enemy.

Properties:

“effect” is property which determines the which action will be performed.

- **HighScoresManager**

This class is responsible of managing the highscores of the players. Since the aim of the game is surviving from the enemies and getting higher scores, it is important for players to see their highscores.

Properties:

“highScoresTable” is a property which holds the highscores table itself.

“highScoresFile” is a property which is for the text file that includes highscores and the names of the highscore owners.

Methods:

“loadHighScores()” is a method which loads the high scores from highScoresFile.

“resetHighScores()” is for resetting the high scores in terms of the users demand.It will reset all the scores.

"createTable()" creates the high scores table with the user demand. It can be used to save high scores.

"saveHighScore(score,name)" is a method that gets the users score and name. Later on it saves that high score to high score file.

- **HighScoresScreen**

When user click the high scores button, this class will print out the high scores from highScoreFile in descending order.

Methods:

"highScoresManager" is a property for HighScoresManager to do all the required things to show the high scores table.

- **HighScoresTable**

This class is for getting the names of the players that have high score and their names. These information are going to be used at HighScoresManager

Properties:

"highScoresNames" is a property to hold the names of the players who did high score.

"highScores" is a property for to hold the high score.

- **Screen**

This class is for getting the names of the players that have high score and their names. These information are going to be used at HighScoresManager

Properties:

"gameObjects" is a property that is coming from Game engine. It holds core features of the game, which can be told as "renders the game itself".

"layoutManager" is a property which is for the layout of the screen, borders of the screen and layout of the screen.

"background" is a property to set the background theme of the game.

Methods:

"update()" is a method to refresh the screen and for updating the screen layout.

- **MainMenuScreen**

This class is for to show the main menu screen and available buttons for the user.

Properties:

"menuItems" is a property for the items in the main menu screen such as new game, load game, tutorial , options etc.

- **MapSelectScreen**

This class is for player to choose the map with available options.

Properties:

"mapThumbnails" is a property that includes the options about the maps in the game.

- **TutorialScreen**

This class is for displaying information about how to play game.

Properties:

"tutorialInformation" includes information about the game playing. It shows what the game objects are and what they do and how they will be used.

- **EventManager**

This class is for handling and organizing events which occurs during the gameplay.

Properties:

"events" is an object for event types which occurs during the gameplay

Methods:

"processEvents" is a function that recognizes the event and carries out this event.

- **InputController**

This class is responsible to take inputs from user. User can play the game with using mouse and keyboard. Keyboard will be just used for entering player name for highscore table. Rest of the controls is done by mouse.

Properties:

"mouselistener" is an object for mouse inputs. The possible inputs can be clicking, scrolling and moving.

"keyboardListener" is an object for keyboard inputs. It determines which buttons on the keyboard are pressed.

Methods:

"getCurrentInputs" is a function that takes current inputs which are entered from user by using mouselistener and keyboardlistener.

- **InteractionManager**

This class is responsible to process the possible interaction of the objects. Since during the game there will be interaction this class will be used to handle them.

Properties:

"gameObjects" property comes from the game engine. We need engine to process interactions.

Methods:

"processInteractions(input)" method is for the processing interactions with the given input.

- **PhysicsManager**

This class is responsible of physics management of the game. Uses game objects and motion manager and collision manager which are the physics part of the game.

Properties:

"gameObjects" property comes from the game engine. We need engine to process interactions.

"dTime" is a property which is called delta time and used for updating scene based on elapsed time since the game last updated.

Methods:

"process()" is a method for continuing processes of the game.

- **CollisionManager**

This class for determining whether there is a collision between game object and fires. It is also responsible to handle these collisions

Properties:

“gameObjects” property comes from the game engine. We need engine to process interactions.

“collisions” is property for indicating whether there is a collision or not.

Methods:

“getCollisions” function checks the fire and game object location and if there is overlapping, it returns collision.

- **MotionManager**

This class for processing the motions of the objects during the game.

Properties:

“gameObjects” property comes from the game engine. We need engine to process interactions.

Methods:

“updatePositions()” is a method to update positions of the objects during the game.

- **DisplayController**

This class is to handle displaying of current screen of the game. It determines render time for displaying.

Properties:

“screen” is object of Screen class and is responsible for displaying game screen.

“renderTime” is property for rendering screen in specific time.

Methods:

“drawObjects” is function to draw game objects at their current location in the screen.

Low Level Design

Object Design Trade-offs

Buy vs Build Trade off

Since we divided our game into the subsystems while we are designing, it is possible that we might integrate a module that we found from the market and continue our implementation. However the module or subsystem that we might buy should be almost exactly fit to our system so that it would save us time and human resources. However, we wouldn't get the source code of the module that we might buy and if we can't modify the module properly, it would be a waste.

Durability vs Platform Dependance

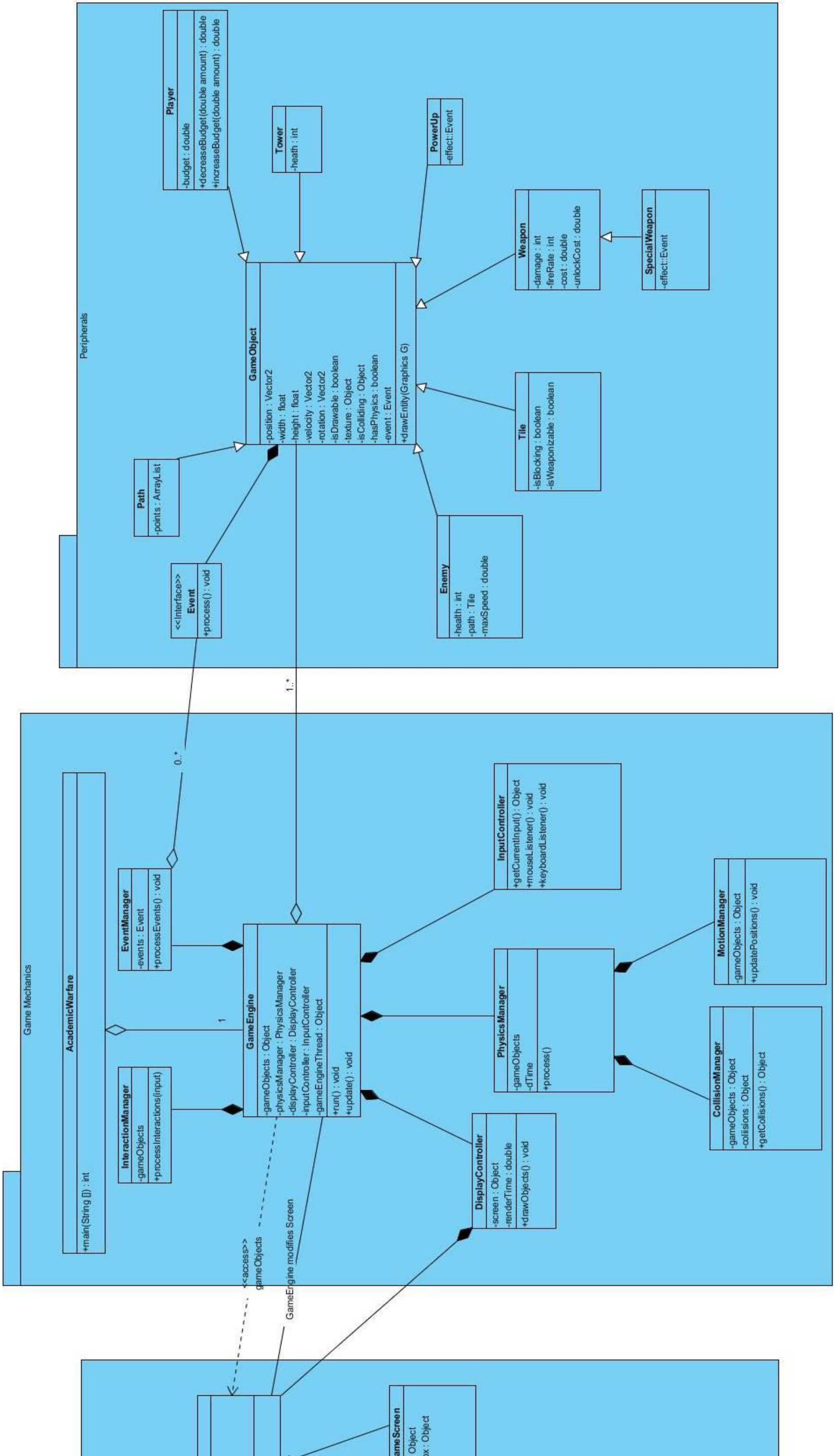
Our game should have as low dependencies as possible on the computer platforms that our game can work on. This will lead our game to work in most computer platforms in the future without needing much modification.

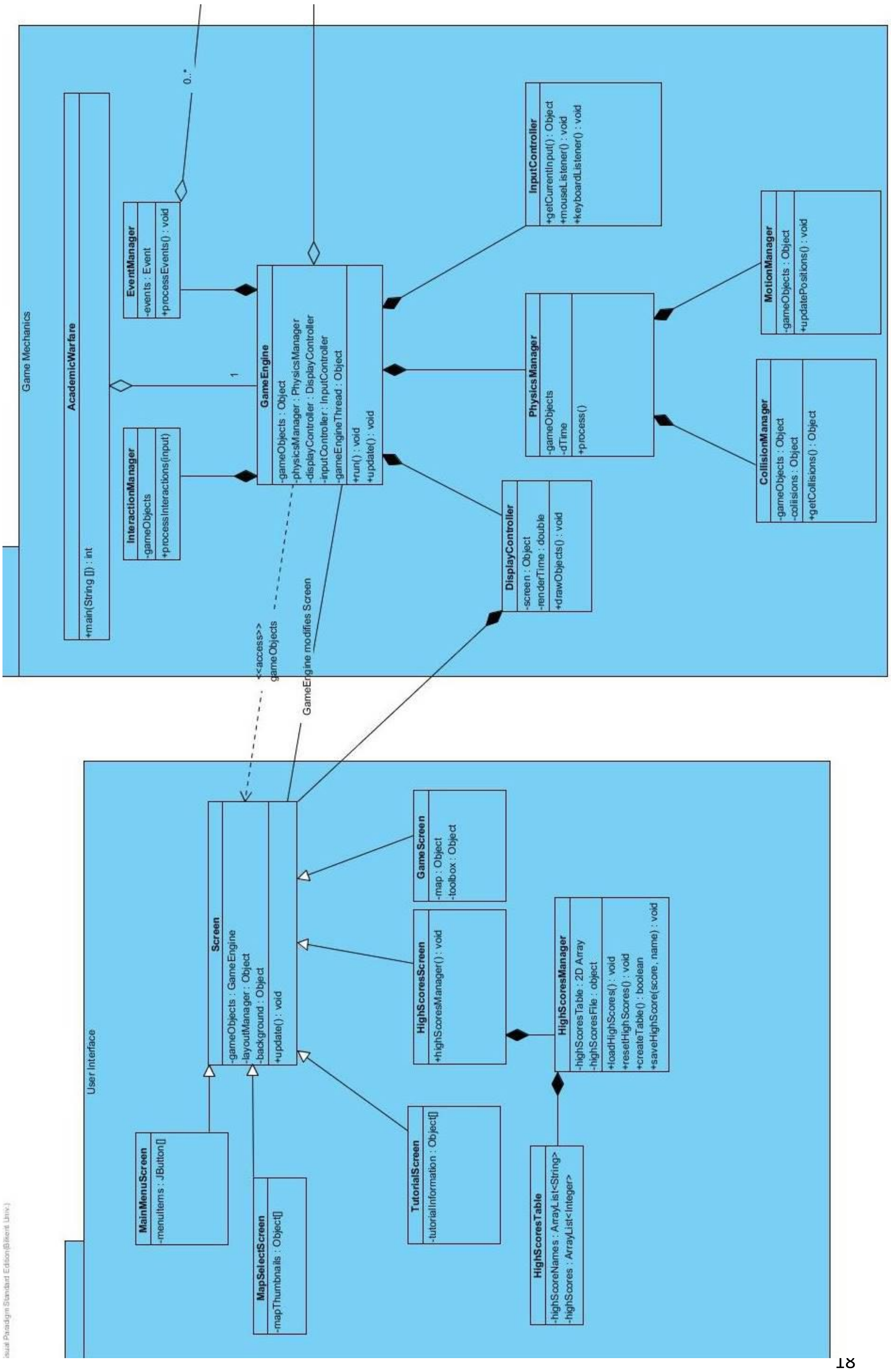
Performance vs Computer Resources

We sacrifice computer resources in order to have a smooth gameplay and graphics. It is important for us to users having a great gameplay without any cut. Since even we sacrifice from computer resources, our game will work on most of the computers without any performance drops.

Final Object Design

See the schemes on the following 2 pages.


Peripherals
Player
-budget : double
+decreaseBudget(double amount) : double
+increaseBudget(double amount) : double
Tower
-heath : int
PowerUp
-effect::Event
GameObject
-position : Vector2
-width : float
-height : float
-velocity : Vector2
-rotation : Vector2
-isDrawable : boolean
-texture : Object
-isColliding : Object
-hasPhysics : boolean
-event : Event
+drawEntity(Graphics G)
Weapon
-damage : int
-fireRate : int
-cost : double
-unlockCost : double
SpecialWeapon
-effect::Event
Tile
-isBlocking : boolean
-isWeaponizable : boolean
Path
-points : ArrayList
<<Interface>>
Event
+process() : void
Enemy
-health : int
-path : Tile
-maxSpeed : double
1..*
0..*
Game Mechanics
AcademicWarfare
+main(String []) : int
EventManager
-events : Event
+processEvents() : void
InteractionManager
-gameObjects
+processInteractions(input)
1
GameEngine
-gameObjects : Object
-physicsManager : PhysicsManager
-displayController : DisplayController
-inputController : InputController
-gameEngineThread : Object
+run() : void
+update() : void
InputController
+getCurrentInput() : Object
+mouseListener() : void
+keyboardListener() : void
PhysicsManager
-gameObjects
-dTime
+process()
MotionManager
-gameObjects : Object
+updatePositions() : void
CollisionManager
-gameObjects : Object
-colisions : Object
+getCollisions() : Object
DisplayController
-screen : Object
-renderTime : double
+drawObjects() : void
<<access>>
gameObjects
GameEngine modifies Screen
ameScreen
Object
x: Object


Game Mechanics
AcademicWarfare
+main(String []) : int
1
0..*
EventManager
-events : Event
+processEvents() : void
InteractionManager
-gameObjects
+processInteractions(input)
GameEngine
-gameObjects : Object
-physicsManager : PhysicsManager
-displayController : DisplayController
-inputController : InputController
-gameEngineThread : Object
+run() : void
+update() : void
InputController
+getCurrentInput() : Object
+mouseListener() : void
+keyboardListener() : void
PhysicsManager
-gameObjects
-dTime
+process()
MotionManager
-gameObjects : Object
+updatePositions() : void
CollisionManager
-gameObjects : Object
-collisions : Object
+getCollisions() : Object
DisplayController
-screen : Object
-renderTime : double
+drawObjects() : void
<<access>>
gameObjects
GameEngine modifies Screen
User Interface
Screen
-gameObjects : GameEngine
-layoutManager : Object
-background : Object
+update() : void
GameScreen
-map : Object
-toolbox : Object
HighScoresScreen
+highScoresManager() : void
HighScoresManager
-highScoresTable : 2D Array
-highScoresFile : object
+loadHighScores() : void
+resetHighScores() : void
+createTable() : boolean
+saveHighScore(score, name) : void
HighScoresTable
-highScoreNames : ArrayList<String>
-highScores : ArrayList<Integer>
MainMenuScreen
-menuItems : JButton[]
MapSelectScreen
-mapThumbnails : Object[]
TutorialScreen
-tutorialInformation : Object[]
isual Paradigm Standard Edition(Bilkent Univ.)

Packages

In Final Design UML Diagram, it is clear to see that we separated the system into three general packages which are related to different cases of our software. The main reason of separating into packages is to clarify how the project is organized and see easily whole system requirements. These packages are Game Mechanics, User Interface and Peripherals. They are focusing totally different aspects of the system and there is connection among them in order to consider any change in future.

If we look inside each packages deeply, İt is realized that classes which performs similar task are gathered. For instance, any class which are responsible for displaying any game entities are gathered into User Interface packages. Since, all of them are related the GUI.

With separating packages and indicating connection between them, it can be also seen that when an error occurs in any part of the system, we can keep track of this error and we can find source of error by looking connection between classes and packages. In addition, if we make any change in any part of the system, we are able to know which parts of system will be affected by this change.

In Game Mechanics package, there are classes which are related to managing game-dynamics. As we sad before, Game Engine is main executive class for the whole game. It coordinates connections between classes. The other classes play a role for fulfilling this purpose.

In Peripherals package, there game object classes. As name of package already suggests, the classes inside of this packages are related to any object in the game screen.

Class Interfaces

Movable:

Movable objects should implement the method move(). If a game object is movable, when move method is called, the object's position will be updated according to the motion dynamics of the object.

Drawable:

Drawable objects should implement the drawEntity() method. drawEntity method takes a Graphics object as an argument. This method is an algorithmic description of how a game object will be visualized. If a game object is drawable then once the drawEntity method is called, it will draw itself accordingly.

Event:

Event classes implement two methods. The first one is eventOccured() which returns true when the conditions for that particular event is satisfied. The other method to be implemented is processEvent() which takes corresponding ArrayList of game objects and modifies them according to the event description.

Weapon:

This interface is implemented by game objects that are weapons. Weapons should implement the shoot() method. Shoot method is generally implemented as shooting to the nearest enemy inside the weapon's range.

Mountable:

Mountable game objects implement mount() method. These objects are generally grid objects. Mount method takes a Weapon instance and mounts that Weapon the cell, itself.