Bilkent University

Department of Computer Engineering

Object-Oriented Software Engineering Project

CS 319 Project: Sky Wars

Final Report

Ayşe Berceste Dinçer, Beril Başak Tukaç, Dilara Ercan, Sertaç Onur Hakbilen

Course Instructor: Hüseyin Özgür Tan
TA: İstemi Bahçeci

Table of Contents	
1. Introduction	8
2. Requirement Analysis	9
2.1. Overview	9
2.1.1. Gameplay and Control	9
2.1.2. Leveling	9
2.1.3. UserPlane	10
2.1.4. Targets	12
2.1.4.1. Target Plane	12
2.1.4.2. Rocket	14
2.1.4.3. Carriage	14
2.1.4.4. Ally	14
2.1.4.5. Bonus Mission Target	16
2.1.4.6. Ship	16
2.1.5. Bonus Packages	16
2.1.5.1. Present Bonus Packages	17
2.1.5.2. Trap Bonus Packages	18
2.1.5.3. Present Bonus Package Locks	19
2.1.6. Obstacles	19
2.1.7. Pilot	19
2.1.8. Bonus Missions	22
2.1.9. Weapon	22
2.1.9.1. Shoots	23
2.1.9.2. Explosives	24
2.1.10. Store	24
2.1.11. Collection	24
2.1.12. Scoring	24
2.2. Functional Requirements	25
2.2.1. Playing Level	25
2.2.2. Playing Bonus Mission	26
2.2.3. Pausing The Game	26
2.2.4. Viewing Store and Purchasing Items	26
2.2.5. Viewing Collection and Changing Preferences	27
2.2.6. Viewing Level Map	27
2.2.7. Changing Settings	27
2.2.8. Viewing Help	27
2.2.9. Viewing Credits	28
2.2.10. Playing Sound	28
2.3. Non-Functional Requirements	28
2.3.1. Usability	28
2.3.2. Performance	28
2.3.3. Reliability	28
2.3.4. Supportability	28
2.4. Constraints	29
2.5. Scenarios	29

2.6. Use Case Models	35
2.6.1. View Main Menu	37
2.6.2. View Level Map	37
2.6.3. Change Settings	37
2.6.4. View Help	38
2.6.5. View Store	38
2.6.6. Purchase Item	39
2.6.7.View Collection	39
2.6.8. Change Preferences	40
2.6.9. View Credits	40
2.6.10. Play Game	40
2.6.11. Play Bonus Mission	41
2.6.12. Pause Game	42
2.6.13. Quit Game	42
2.7. User Interface	43
2.7.1. Navigational Path	43
2.7.2. Main Menu Screen	43
2.7.3. Level Map Screen	44
2.7.4. Level Play Screen	45
2.7.5. Store Screen	45
2.7.6. Collection Screen	46
2.7.7. Settings Screen	47
2.7.8. Help Screen	48
2.7.9. Credits Screen	49
2.7.10. Pause Menu Screen	49
3.Analysis	51
3.1. Object Model	51
3.1.1. Domain Lexicon	51
3.1.2. Class Diagram	52
3.2. Dynamic Models	54
3.2.1. State Chart Diagrams	54
3.2.1.1. State Chart Diagram of UserPlane	54
3.2.1.2. Activity Diagram for Overall Game Flow	55
3.2.1.3. Activity Diagram for Game Play	56
3.2.2. Sequence Diagrams	56
3.2.2.1. Start Game	56
3.2.2.2. Creation of GameObjects During Level	57
3.2.2.3. Level Play	58
3.2.2.4. Collecting Present Bonus Package	59
3.2.2.5. Collecting Trap Bonus Package	60
3.2.2.6. Obstacle Hit	60
3.2.2.7. Using an Explosive as Weapon	61
3.2.2.8. Shooting A Rocket	62
3.2.2.9. Playing Bonus Mission	63
3.2.2.10. Purchasing Plane from Store	65
3.2.2.11. Changing UserPlane Preference from Collection	65
3.2.2.12. Changing Settings	66
4. Design	67

4.1. Design Goals	67
4.2. Subsystem Decomposition	69
4.3. Architectural Patterns	74
4.4. Hardware/Software Mapping	75
4.5. Addressing Key Concerns	76
4.5.1. Persistent Data Management	76
4.5.2. Access Control and Security	76
4.5.3. Global Software Control	76
4.5.4. Boundary Conditions	76
5. Object Design	78
5.1. Pattern Applications	78
5.1.1. Facade Design Pattern	78
5.1.2. Observer Design Pattern	79
5.1.3. Strategy Design Pattern	81
5.2. Class Interfaces	82
5.2.1. User Interface Subsystem Class Interfaces	82
5.2.2. Game Control Subsystem Class Interfaces	93
5.2.3. Game Model Subsystem Class Interfaces	105
5.3. Specifying Contracts	130
6. Conclusions and Lessons Learned	136

Table of Figures

Figure 1 Example Level Background Level 4 Glacial Area	9
Figure 2 Alderaan Cruiser UserPlane	10
Figure 3 Tomcat UserPlane	11
Figure 4 F22-Raptor UserPlane	11
Figure 5 Saab-Gripen UserPlane	11
Figure 6 Wunderwaffe UserPlane	12
Figure 7 F-16 TargetPlane	12
Figure 8 Republican Attack TargetPlane	13
Figure 9 Imperial Shuttle TargetPlane	13
Figure 10 Havoc Marauder TargetPlane	13
Figure 11 BOSS TargetPlane	14
Figure 12 Rocket	14
Figure 13 Alderaan Cruiser Ally	15
Figure 14 Tomcat Ally	15
Figure 15 F22-Raptor Ally	15
Figure 16 Saab-Gripen Ally	16
Figure 17 Wanderwaffe Ally	16
Figure 18 Present Bonus Package	17
Figure 19 Trap Bonus Package	18
Figure 20 Obstacle Example Iceberg	19
Figure 21 Pilot Nick	20
Figure 22 Pilot Penny	20
Figure 23 Pilot Mike	21
Figure 24 Pilot Eva	21
Figure 25 Pilot Neo	22
Figure 26 Bullet	23
Figure 27 Metal Ball	23
Figure 28 Flame Gun	23
Figure 29 Frost Laser	23
Figure 30 Laser	23
Figure 31 Torpedo	23
Figure 32 Bomb	24
Figure 33 Missile	24
Figure 34 Use Case Diagram of Sky Wars	36
Figure 35 Navigational Path of Sky Wars	43
Figure 36 Main Menu Screen	43
Figure 37 Level Map Screen	44
Figure 38 Level Play Screen	44
Figure 39 Store Screen	46
Figure 40 Collection Screen	47
Figure 41 Settings Screen	48
Figure 42 Help Screen	48
Figure 43 Credits Screen	49
Figure 44 Pause Menu Screen	50
Figure 45 Class Diagram of Sky Wars	52
Figure 46 State Chart Diagram of UserPlane	54

Figure 47 Activity Diagram of Overall Game Flow	55
Figure 48 Activity Diagram of Game Play	56
Figure 49 Start Game Sequence Diagram	57
Figure 50 Create GameObjects During Level Sequence Diagram	58
Figure 51 Level Play Sequence Diagram	59
Figure 52 Colliding with Present Bonus Package Sequence Diagram	60
Figure 53 Colliding with Trap Bonus Package Sequence Diagram	60
Figure 54 Obstacle Hit Sequence Diagram	61
Figure 55 Using an Explosive as Weapon Sequence Diagram	62
Figure 56 Shooting A Rocket Sequence Diagram	62
Figure 57 Playing Bonus Mission Sequence Diagram	64
Figure 58 Purchasing a UserPlane from Store Sequence Diagram	65
Figure 59 Changing UserPlane Preference from Collection Sequence Diagram	66
Figure 60 Changing Settings Sequence Diagram	66
Figure 61 Basic Layers of Sky Wars	70
Figure 62 Sky Wars Subsystem Decomposition with Subsystem Details	71
Figure 63 Subsystem Decomposition with Connection Details	72
Figure 64 User Interface Subsystem Details	73
Figure 65 Game Control Subsystem Details	73
Figure 66 Game Model Subsystem Details	74
Figure 67 Component Diagram of Sky Wars	75
Figure 68 Deployment Diagram of Sky Wars	75
Figure 69 Detailed Deployment Diagram of Sky Wars	76
Figure 70 Facade Pattern Application to Game Control Subsystem	78
Figure 71 Facade Pattern Application to Game Model Subsystem	79
Figure 72 Observer Pattern Application to Sky Wars	80
Figure 73 Strategy Pattern Application to Sky Wars	81
Figure 74 GameFrame Class Diagram	82
Figure 75 PauseMenuPanel Class Diagram	83
Figure 76 LevelPanel Class Diagram	85
Figure 77 CollectionPanel Class Diagram	87
Figure 78 StorePanel Class Diagram	88
Figure 79 MainMenuPanel Class Diagram	89
Figure 80 LevelMapPanel Class Diagram	90
Figure 81 CreditsPanel Class Diagram	91
Figure 82 HelpPanel Class Diagram	91
Figure 83 SettingsPanel Class Diagram	92
Figure 84 GameObjectInterface Class Diagram	92
Figure 85 GameManagerObserver Class Diagram	93
Figure 86 GameManager Class Diagram	94
Figure 87 SoundManager Class Diagram	96
Figure 88 CollectionManager Class Diagram	97
Figure 89 LevelManager Class Diagram	99
Figure 90 FileManager Class Diagram	102
Figure 91 CollisionManager Class Diagram	102
Figure 92 UserPlaneDestroyedCollisionManager Class Diagram	103
Figure 93 UserPlaneHitCollisionManager Class Diagram	103
Figure 94 TargetHitCollisionManager Class Diagram	104

Figure 95 BonusPackageCollisionManager	104
Figure 96 CollisionPolicy Class Diagram	105
Figure 97 Level Class Diagram	106
Figure 98 GameObject Class Diagram	108
Figure 99 Movement Class Diagram	109
Figure 100 UserPlane Class Diagram	110
Figure 101 UserPlaneEnum Class Diagram	111
Figure 102 Pilot Class Diagram	112
Figure 103 PilotEnum Class Diagram	112
Figure 104 Obstacle Class Diagram	113
Figure 105 Weapon Class Diagram	114
Figure 106 Shoot Class Diagram	114
Figure 107 ShootEnum Class Diagram	115
Figure 108 Explosive Class Diagram	115
Figure 109 ExplosiveEnum Class Diagram	116
Figure 110 Target Class Diagram	117
Figure 111 Ally Class Diagram	117
Figure 112 AllyEnum Class Diagram	118
Figure 113 BonusMissionTarget Class Diagram	118
Figure 114 BonusMissionTargetEnum Class Diagram	119
Figure 115 Rocket Class Diagram	119
Figure 116 RocketEnum Class Diagram	120
Figure 117 Carriage Class Diagram	121
Figure 118 CarriageEnum Class Diagram	121
Figure 119 Ship Class Diagram	122
Figure 120 ShipEnum Class Diagram	122
Figure 121 TargetPlane Class Diagram	123
Figure 122 TargetPlaneEnum Class Diagram	123
Figure 123 BonusPackage Class Diagram	124
Figure 124 PresentBonusPackage Class Diagram	124
Figure 125 SpeedBonusPackage Class Diagram	125
Figure 126 DamageBonusPackage Class Diagram	125
Figure 127 ObstacleInvisibilityPackage Class Diagram	126
Figure 128 InvincibilityPackage	126
Figure 129 CoinPackage Class Diagram	127
Figure 130 TimeBonusPackage Class Diagram	127
Figure 131 HealthBonusPackage Class Diagram	128
Figure 132 TrapBonusPackage Class Diagram	128
Figure 133 DamageTrapPackage Class Diagram	129
Figure 134 SpeedTrapPackage Class Diagram	129
Figure 135 HealthTrapPackage Class Diagram	130
Figure 136 PlaneEnlargePackage Class Diagram	130

1. Introduction

Sky Wars is a game of air battle where the Player is controlling a war plane which can shoot and be hit. Sky Wars is a level-based game. The Player gains access to next level after completing the current level. Along with regular levels, the User can also play Bonus Missions. The aim of the Player is to pass the level by reaching point threshold without depleting his health within limited level time.

In Sky Wars, various dangers and Targets exist. Levels have different themes and various Obstacles appear according to these themes. The game also contains Bonus Packages, some of them helpful for the User and some of them act as traps. Present Bonus Packages include speed or health boosts and time-limited powers such as Obstacle Invisibility and Invincibility. On the other hand, Trap Packages make the game harder for Player by decreasing health or speed. Along with Obstacles and Bonus Packages, various Targets exist. Any object with a certain health which can be shot by the Player is considered as a Target. Different Targets exist with various shapes, powers and movements. Target Planes shoot to Player with various Weapons. During the game, Obstacles, various Targets and Bonus Packages appear on the screen. Moreover, enemy planes fly and shoot to plane of the Player.

The Player tries to avoid colliding with Obstacles, Targets and Trap Packages. Collision with Trap Packages makes the game harder while colliding with Targets and Obstacles depletes the health of User and results in failing the level. Player collects Bonus Packages to use various boosts. The Player shall escape from Target Plane shots as well in order not to lose points. Furthermore, Player shoots Targets with Weapons to earn points. Each level has a certain time period and a point threshold. If the Player reaches the end of the level, his points are transformed to coins. These coins are used for the purchase of items.

Sky Wars has a Store within where the Player can purchase new items. The game offers options for UserPlane, each of them with various properties such as speed or health. There are also different Pilots which the Player can select. Bonus Packages Locks are also sold in the store for users to purchase in order to collect Bonus Packages during levels. Moreover, different Weapons with different damage amounts and Explosives exist. The Player can purchase various weapons and can use these weapons in the game for shooting Targets.

Sky Wars is a desktop application and it is controlled with keyboard.

2. Requirements Analysis

2.1. Overview

2.1.1. Gameplay & Control

In Sky Wars, the Player is controlling a plane, UserPlane. The aim of the game is to escape from colliding with Targets, Obstacles, Trap Packages and Weapons and also shoot to Targets and collect Bonus Packages. In the game, the User will be using up and down keys to get away from the Obstacles, Targets and also to collect Bonus Packages. For shooting, space key; for weapon change 'C' key will be used. In a Bonus Mission the User will use right and left keys instead of regular up and down keys or all four arrow keys according to Bonus Mission type.

2.1.2. Leveling

Sky Wars is a level-based game. Each level has a theme, an allocated time period, a point threshold, a coin coefficient and a list of GameObjects in which their coordinates, movements and appearance times are specified. Point threshold is for evaluating Player success. When the User reaches point threshold at the end of the level the level is successful. Coin coefficient on the other hand, is the amount of points necessary to obtain one Coin. For instance, if the coin coefficient is 4, at the end of the level for each 4 points the User will earn a coin. GameObjects within a Level are UserPlane, Pilot and Weapons of UserPlane, various Targets and Bonus Packages. The User purchases a UserPlane and a Pilot from Store and selects one of those as preference from the Collection before level. User also purchases Bonus Package keys from Store. If a Bonus Package Lock is purchased, it will appear in the level. However the game decides how many Bonus Packages will appear and when they will appear. Moreover, the game decides which Target, TrapBonusPackage or Obstacle types will appear during the game how many will appear.

There will be a Level Map, showing all the levels completed and uncompleted all through the path. The User will be able to play previous levels. When the game is started, the Player has access to Level 1 only. The User is allowed to access next level only when he successfully completes the current level. While the Player continues passing levels, levels become harder. The point threshold and coin coefficient increase while the levels become harder. Number of Targets, Obstacles and attacks on UserPlane increase and intensify.

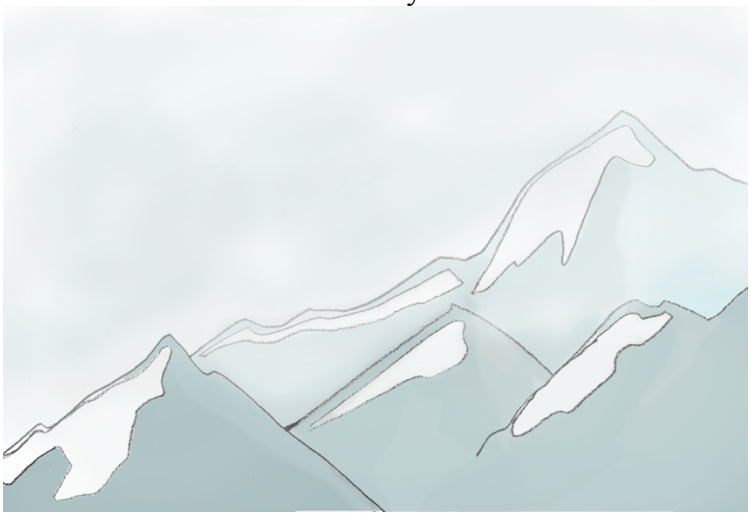

Figure 1 Example Level Background Level 4 Glacial Area

The game will contain 5 different levels for now with various themes and increasing difficulties:

- **Level 1** Field Area
- **Level 2** City Area
- **Level 3** Desert Area
- **Level 4** Glacial Area
- **Level 5** Forest Area

2.1.3. UserPlane

UserPlane is the plane the Player is controlling. UserPlane is controlled by keyboard and it can fire Weapons. The UserPlane has below listed properties:

- **Speed** which corresponds to vertical speed in regular levels and horizontal or/and vertical speed in Bonus Missions. Basically speed corresponds to how fast the UserPlane reacts to keyboard input.
- **Health** represents the living power of the UserPlane. When health of the UserPlane is depleted, the game is over and the Player fails.
- **Shoot Damage** is the amount of damage the UserPlane can give to Target. When a Target is shot the total damage given to Target is the sum of the damage value of the Weapon and the Shoot Damage of the UserPlane.
- **Shooting Type** indicates whether the UserPlane has singular or double shoot which means that the UserPlane can send one or two Weapon at one time, when 'Space' key is pressed.

Sky Wars will include several UserPlanes. The UserPlanes differ in terms of above listed properties. The User will be able to use different UserPlanes in Levels as long as he purchased it previously from the Store and specify it as current selection from the Collection. Furthermore, the UserPlanes can be upgraded or weakened in terms of its speed, health, damage etc. when a BonusPackage is collected. UserPlanes do not have a specific Weapon type, they can shoot any Weapon the Player has purchased and selected. The UserPlane has a certain health and its health is decreased when UserPlane collides with a Weapon that is sent by TargetPlanes. The health of the UserPlane is depleted when it collides with an Obstacle or Target. There will be 5 different types of planes which are;

- **Alderaan Cruiser** which is the standard plane, having the standard health, speed and damage.

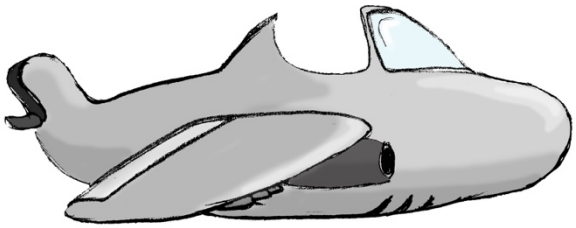

Figure 2 Alderaan Cruiser UserPlane

- **Tomcat** is faster than Alderaan Cruiser and its health is much more.

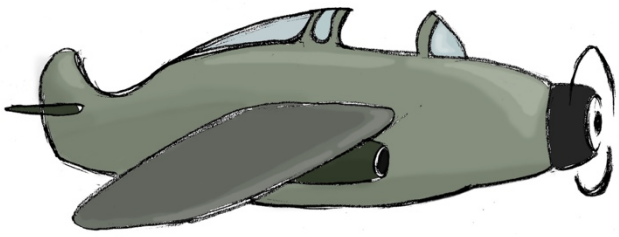

Figure 3 Tomcat UserPlane

- **F-22 Raptor** has the standard speed but it has 2 locations for shooting, has double shoot as shoot type, and the plane's default damage is higher than Alderaan Cruiser and Tomcat.

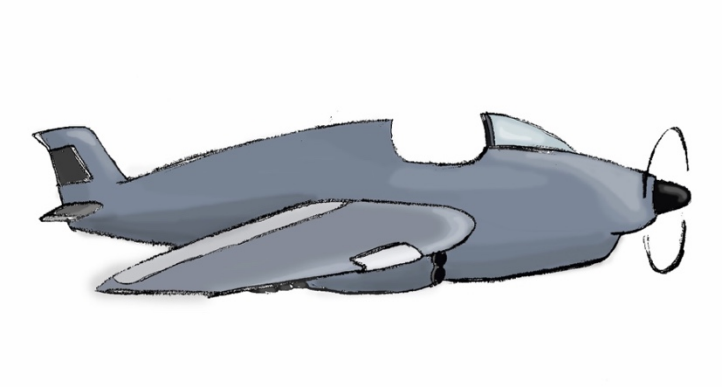

Figure 4 F22-Raptor UserPlane

- **Saab Gripen** is faster than F-22 Raptor, has a higher level of health and default damage compared to the previous planes and also possesses 2 shooting locations for shooting.

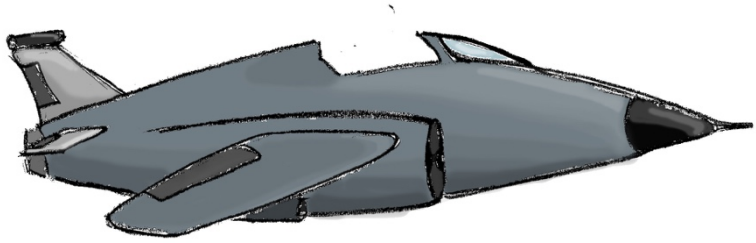

Figure 5 Saab Gripen UserPlane

- **Wunderwaffe** has the maximum capacity in all of the domains, rendering itself the most powerful UserPlane in the game.

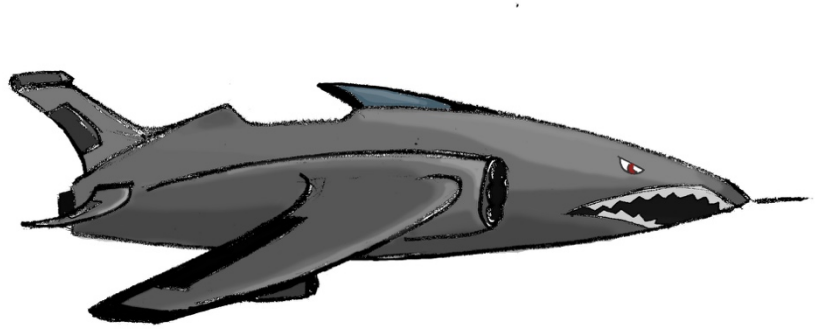

Figure 6 Wunderwaffe UserPlane

2.4.1. Target

The Targets are the destructible objects in the game. They have a certain health value and can be shot by Player Weapons. When shot, Targets earn the User points. However, some kind of Targets is not supposed to be damaged and if they are shot, the User will lose points. Besides, Bonus Mission has also a corresponding Target which when shot, earns the Player the access to Bonus Mission. If UserPlane collides with any Target, the game is over and the Player fails.

2.1.4.1. *TargetPlane*

TargetPlane is a Target type which can shoot to UserPlane and represented as enemy planes. TargetPlane has below listed properties:

- **Health** represents the living power of the TargetPlane. The health of the TargetPlane decreases when the Player shoots it with Weapons.
- **Shooting Time Period** represents the shooting speed or intensity of the TargetPlane. This property can be explained as the amount of seconds between consecutive shoots of the TargetPlane.
- **Weapon Type** is the Weapon that is associated with the TargetPlane. Contrary to UserPlane, TargetPlanes have a constant Weapon type which cannot be changed by the User and determined by the game itself.

TargetPlanes differ by these properties. Levels contain many TargetPlanes with various properties. Their paths (movement route), coordinates and appearance in the level will be specified beforehand for a level. If they get shot their health will be decreased and additional points will be given to the User. Sky Wars contains 5 types of TargetPlanes:

- **F-16** has low health and its shooting period is long. Its Weapon type is Bullet.

Figure 7 F-16 TargetPlane

- **Republic Attack Cruiser** has medium health and long shooting period. Its Weapon type is Metal Ball.

Figure 8 Republic Attack TargetPlane

- **Imperial Shuttle** has medium health and medium shooting period time. Its Weapon type is Flame Gun.

Figure 9 Imperial Shuttle TargetPlane

- **Havoc Marauder** has the maximum health with low shooting period time. Its weapon is Frost Laser.

Figure 10 Havoc Marauder TargetPlane

- **BOSS** has the maximum health and minimum shooting period. Its Weapon type is Laser.

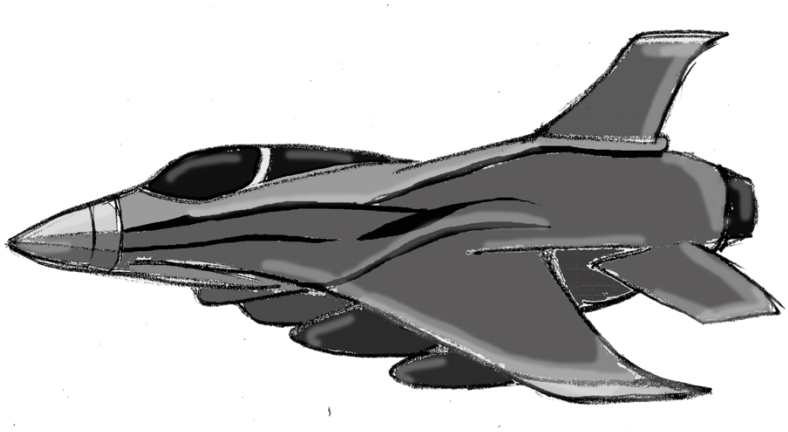

Figure 11 BOSS TargetPlane

2.1.4.2. *Rocket*

Rocket is a different type of Target which explodes when its health is depleted. Rockets create an explosion within a specified area and gives damage to all GameObjects within the area. However, if the UserPlane in that area its health is also decreased or even depleted sometimes. Hence, Rocket should be destroyed by the Player when it is far from the UserPlane to earn maximum possible points. Rocket has two properties which are health and damage area that specifies the explosion area.

Figure 12 Rocket

2.1.4.3. *Carriage*

Carriage is a Target which does not cause any damage to UserPlane, meaning it cannot shoot. When User plane destroys carriage it gains the Player points. The only property of carriage is Health.

2.1.4.4. *Ally*

Ally is a harmless plane in Sky Wars. It only passes from aerospace of UserPlane. The only thing that User should do is not to shoot the Ally Plane. If User hits an Ally, User's points decrease according to the Ally type. The Ally Planes are the same as UserPlanes and they are driven by Pilots other than the current Pilot of UserPlane. There are 5 different Ally types. The health of Ally planes increase gradually in below list:

- **Alderaan Cruiser Ally**

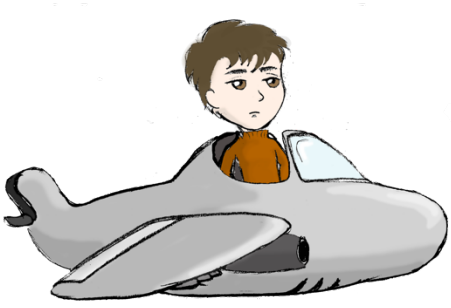

Figure 13 Alderaan Cruiser Ally

- **Tomcat Ally**

Figure 14 Tomcat Ally

- **F-22 Raptor Ally**

Figure 15 F22-Raptor Ally

- **Saab Gripen Ally**

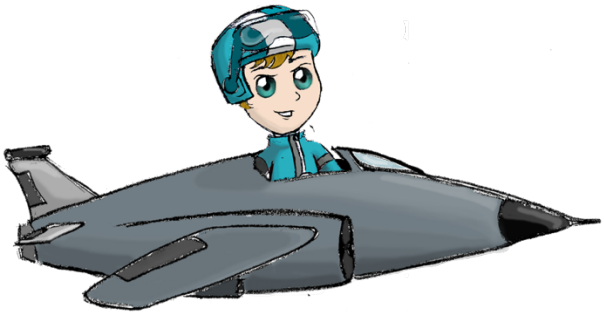

Figure 16 Saab Gripen Ally

- **Wunderwaffe Ally**

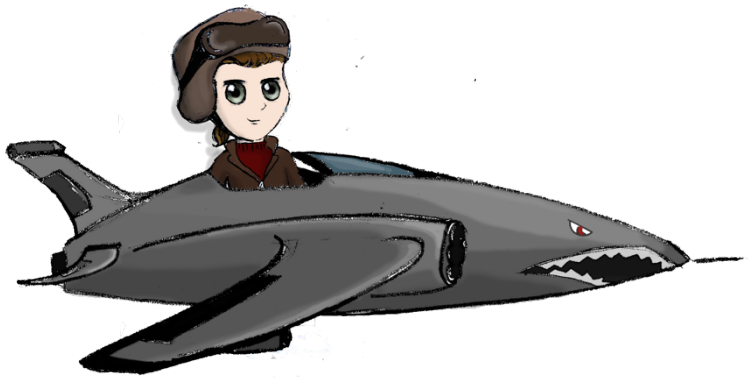

Figure 17 Wunderwaffe Ally

2.1.4.5. *Bonus Mission Target*

Bonus Mission Target is used for unlocking the Bonus Mission in the Level Map. In every chapter there is only one Bonus Mission Target and User should destroy this Target to open access to Bonus Mission. The only property of Bonus Mission Target is health. However relative to its rareness it has a high health value.

2.1.4.6. *Ship*

Ship has a no fire power and it only appears in Bonus Missions. User plane can damage Ship by throwing torpedoes from above. Ships are naturally in water, ground level.

2.1.5. *Bonus Package*

Bonus Packages are exceptions to regular game flow. When UserPlane collides with a Bonus Package, the collided bonus is applied. There are two types of Bonus Packages. Present Bonus Packages enhance the game while Trap Bonus Packages make the game harder. The Player has to purchase keys to Present Bonus Packages in order for them to appear during level. There are three levels of Present Bonus Package Locks in Store. The higher level keys create Packages with higher power during level. Trap Bonus Packages are not sold in Store and they can appear in any Level. The appearance time, number and coordinates of the Bonus Packages are determined by the game itself. However, whether a

PresentBonusPackage will occur during levels and how powerful it will be depend on whether the Package Lock is purchased by the user and the level of the Bonus Package Lock.

2.1.5.1. *Present Bonus Packages*

Figure 18 Present Bonus Package

- **Health Bonus Package**

When UserPlane collects these Packages, the health of the UserPlane is increased by the amount specified in the Bonus Package. There are three type of health Packages, each of them contribute to health of the UserPlane with different values.

- **Speed Bonus Package**

Speed Bonus Packages increase UserPlane speed by the amount specified in the Bonus Package. However, these Bonus Packages have a time limit. When the time period ends UserPlane's speed get back to its original level. There are three types of speed Packages and these Bonus Packages add different amounts of speed to UserPlane.

- **Shoot Damage Bonus Package**

When UserPlane collects Shoot Damage Packages, these Packages boost UserPlane's damage points according to corresponding Package value. These Packages have a time period and when it finishes the User plane's damage returns to its default value. There are three types of Packages and each of them increases the Shoot Damage by different amounts.

- **Time Bonus Package**

Time Bonus Package increases remaining time allocated for the level or Bonus Mission. It also has 3 types and these Packages add different amounts of time to level time. These Bonus Packages can help User collect more points until the end of the mission.

- **Obstacle Invisibility Package**

Obstacle Invisibility Package helps User to avoid from Obstacles in a time period. Obstacles become invisible and User plane can move more easily. This bonus also has a certain time period and until the end of this period the health of the UserPlane is not changed when it hits an Obstacle. There are three types of Obstacle Invisibility Package with varying effect periods.

- **Invincibility Package**

UserPlane does not get any damage when it collects Invincibility Package. This Package also has a certain time period and during this period the health of the UserPlane stays the same. When the time period finishes the UserPlane returns to its default state. There are three types of Invisibility Package with varying effect periods.

- **Coin Package**

UserPlane can collect this Package to increase the amount of Player coins. There are three types of Coin Packages, each of them earning User different amount of coins. Coin Package Locks are not solved in Store. They only appear in Bonus Missions.

2.1.5.2. *Trap Bonus Packages*

Figure 19 Trap Bonus Package

- **Plane Enlarge Package**

This Package increases the area of the UserPlane. If the UserPlane collides with this Package it can get damage easily because of its surface area. UserPlane returns its original state after the time period of this Package finishes. Plane Enlarge Package has only one type.

- **Health Trap Package**

These Packages decrease the health of UserPlane when it is collected. It also has three types like Health Bonus Packages.

- **Speed Trap Package**

These Packages decrease the speed of UserPlane when it is collected. UserPlane returns to its original speed when the time period of Speed Trap Package finishes. These Packages have also three types each of them decreasing speed by a certain amount just like Speed Bonus Packages.

- **Damage Trap Package**

These Packages decrease UserPlane's damage according to its type. There are three types of Damage Trap Packages with various levels of damage decrease. Similar to Damage Bonus Packages, they have a certain effect period. At the end of this period, the damage of the UserPlane returns to default value.

2.1.5.2. *Present Bonus Package Locks*

In order for a Present Bonus Package to appear during level, the Player has to purchase Bonus Package Lock from Store. Bonus Package Locks, when purchased, allow Present Bonus Package to appear during level. There are corresponding Bonus Package Locks for Health, Speed, Shoot Damage, Time Bonus, Obstacle Invisibility and Invincibility Bonus Packages. Package Locks have three levels. When the Player purchases a Lock for the first time, the level of the Lock becomes one and the level increases up to three with each purchase. According to the level of the lock, the power of the Bonus Package in level increase. There are three variations to all Present Bonus Packages and three levels of Locks corresponding to these variations. For instance, if the Player has purchased Health Bonus Package Lock Level 2, then Health Bonus Packages with medium health value appear during levels.

2.1.6. *Obstacle*

Obstacles are bumps that appear in the Level View. They do not move. When UserPlane collides with an Obstacle its health is depleted and the game is over. The appearance times and coordinates of Obstacles are determined by the System.

Figure 20 Obstacle Example Iceberg

There are six different Obstacles:

- **Cloud**
- **Tree**
- **Iceberg**
- **Sand Hill**
- **Pyramid**
- **Building**

These Obstacles are adapted to Level themes. For instance, in City themed Level, Buildings appear as Obstacles.

2.1.7. *Pilot*

Pilots are the characters that represent the Player. Pilot drives the UserPlane. It has speed property. Different characters have different speed capabilities. The total speed amount of UserPlane is calculated as the sum of the speed of UserPlane and speed of Pilot. Pilots can also be purchased from Store and selected as the current choice from Collection before the level. There are 5 pilots in Sky Wars, 2 girls and 3 boys.

- **Nick** is a boy. He is the default Pilot and has 0 speed

Figure 21 Pilot Nick

- **Penny** is a girl and has low speed

Figure 22 Pilot Penny

- **Mike** is a boy and has low speed

Figure 23 Pilot Mike

- **Eva** is a girl and has high speed

Figure 24 Pilot Eva

- **Neo** is a boy and has high speed

Figure 25 Pilot Neo

2.1.8. *Bonus Mission*

If the Player shoots Bonus Mission Target in a Level, Bonus Mission is opened in Level Map. In Bonus Missions, there are not any Obstacles and the UserPlane is not attacked. The Player is allowed to play a Bonus Mission only once and then the Bonus Mission button is removed from the Level Map until the Player shoots and destroys another Bonus Mission Target. The health of UserPlane cannot be depleted during Bonus Mission since the Player is not attacked. Hence, there is no concept of success or fail for a Bonus Mission. The Player only aims to earn as many points as he can. There are 2 types of Bonus Missions:

- **Shooting Warships**

In this Bonus Mission Player tries to destroy Ship objects by shooting Torpedos from above. In this Bonus Mission UserPlane can only move to left or right. The Player can shoot only Torpedos and cannot change Weapon type.

- **Coin Heaven**

In Coin Heaven Player tries to collect as many Coin Packages as possible until the end of the mission. UserPlane can be directed to all four directions within this mission. The Player cannot shoot hence cannot change Weapon.

2.1.9. *Weapon*

Planes, both UserPlane and TargetPlane use Weapons to give damage to the GameObjects. Weapons cannot exist on their own, they are either sent by the TargetPlanes or the UserPlane. User must purchase Weapons from store in order to use it within levels. Weapon purchase is done with amounts; Player specifies the amount of a Weapon type when he tries to purchase it. The Player has infinitely many standard type Weapon, Bullets and he does not need to purchase it. During Level, Player can change Weapon with ‘C’ key and he can use different Weapon types as long as the Weapon is left. There are basically two types of Weapons, Shoot and Explosive.

2.1.9.1. *Shoot*

Shoot is a type of Weapon which gives damage to only the GameObject it collides with. Shoot has damage property which represents the amount of health it deducts from GameObjects. There are 5 different Shoot types with different damage values. They are listed from lowest to highest damage amount:

- **Bullet**

Figure 26 Bullet

- **Metal Ball**

Figure 27 Metal Ball

- **Flame-gun**

Figure 28 Flame-gun

- **Frost-laser**

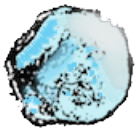

Figure 29 Frost-laser

- **Laser**

Figure 30 Laser

- **Torpedo** is an additional type of Shoot which is not sold in the Store and can only be used in Bonus Missions.

Figure 31 Torpedo

2.1.9.2. *Explosive*

Explosives create explosion when they collide with a Target. They have damage and damage area properties. They decrease the health of all GameObjects within their damage area. Explosives have 2 types:

- **Bomb** has small damage value and small damage area

Figure 32 Bomb

- **Missile** has larger damage value and larger damage area

Figure 33 Missile

2.1.10. *Store*

User can use coins, which are collected in levels and Bonus Missions in Sky Wars' Store. In Store, User can purchase UserPlanes, Weapons, Present Bonus Package Locks and Pilots if he has sufficient amount of coins. The Store locks the access to GameObjects the Player cannot afford. Moreover, when the User tries to purchase a Weapon he is required to specify the amount of the Weapon. When the User cannot afford specified amount of Weapons he is notified. After purchase is completed, the User can view all purchased items in Collection. Purchased Bonus Packages can be seen in levels after purchase. All purchased Weapons can be used within levels as well.

2.1.11. Collection

In Collection, the User will be able to see all purchased items UserPlane, Pilot, Weapons and Bonus Package Locks. The Player is also able to select a UserPlane and Pilot from Collection to use during level. When the Player has not purchased any items, default UserPlane and Pilot is used in levels.

2.1.12. Scoring

The scoring system is used to measure the performance of the Player. When a Weapon hits UserPlane, the health of UserPlane is decreased by the amount of Weapon damage. The same amount of points is deduced from total amount of points within level. When the Player sends a Weapon and this Weapon collides with an object (or objects if the Weapon is an Explosive) the health of the Target is decreased by the sum of Weapon damage and UserPlane shoot damage. The same amount of points is added to total amount of points. The point for a level starts from 0 and cannot drop below 0. The level may end with 3 different situations:

- **Level Failed with Depleted Health:** When the UserPlane collides with a Target or Obstacle or its health is depleted because of multiple shots, the game is over. The Level is unsuccessful. However, the amount of points collected so far is turned into Coins according to Level coefficient.

- **Level Failed with Threshold Unreached:** When the Player completes the level but cannot reach threshold points for the level, the level is unsuccessful. Still, the amount of points collected so far is turned into Coins according to Level coefficient.

- **Level Passed:** When the Player completes the level and reaches threshold points for the level, the level is passed. The amount of points collected so far is turned into Coins according to Level coefficient. The Player gains access to next level.

2.2. Functional Requirements

2.2.1.Playing Level

- The Player should be able to move UserPlane up or down
- The Player must be able to shoot, fire Weapons
- The Player should be able to change Weapon type
- The System should notify the Player when a certain Weapon type is depleted and UserPlane cannot shoot
- The System should decide the background and map of game objects according to the selected level
- The System should place Obstacles to certain locations in the screen
- The System should place Trap Bonus Packages to certain locations in the screen
- The System should place Present Bonus Packages that the Player has purchased to certain locations in the screen
- The System should display and move various Targets according to the level the Player is playing
- The System must send Weapons to UserPlane via Target Planes
- The System must display the time left on the screen
- The System must display the health left on the screen
- The System should display the weapon types and the number of the left weapons relatively on the screen
- The system should end the game and notify the Player when a collision occurs between the UserPlane and an Obstacle or a Target.
- The System should create an explosion animation when an Explosive Weapon is used or a rocket is destroyed
- The System should update the health of a Target when it is hit

- The System must update the Player health when the UserPlane is hit
- The System should create a destruction animation when health of a Target is depleted
- The System should update the time each second
- The System must update the points after each collision
- The System should notify the User about the change in points or health after each collision
- The System should apply the respective bonus when a Bonus Package is collected
- The System must notify the Player about the current Bonus Package
- The System should notify the Player if the level is failed
- The System should notify the Player if he has completed the level
- The System should check the total amount of points collected during level and decide whether the level is successful or not
- The System should notify the Player whether the level is successful or not
- The System should turn the points into coins according to the level coefficient at the end of the level
- The System must notify the Player about the total points and coins earned at the end of the level

2.2.2. *Playing Bonus Mission*

- The Player should be able to move the UserPlane left-right and/or up-down
- The Player should be able to collect Coin Packages or shoot Ships with Torpedoes, according to the Bonus Mission definition
- The System should load the background image and game objects according to the Bonus Mission theme
- The System should not allow the Player to play the same Bonus Mission more than once
- The System should update the time each second
- The System should update points after each collision
- The System must not consider or update health of the Player during Bonus Mission
- The System should turn the total points earned during Bonus Mission into coins
- The System should upload coins earned during Bonus Mission directly to Player account
- The System should notify the Player about point updates after each collision
- The System should notify the Player about total points and coins earned after the Bonus Mission completion
- The System should open access to Bonus Mission only if the Bonus Mission Target is destroyed during levels
- The System should lock access to Bonus Mission after it is played once until the access is earned again

2.2.3. *Pausing The Game*

- The Player should be able to pause the game while playing any level or Bonus Mission
- The System should pause the game, stop all moving game objects and time when relative button is clicked
- When the game is paused the System must display pause menu
- The Player should be able to continue to play the game by pressing relative button in the Pause Menu

2.2.4. *Viewing Store and Purchasing Items*

- The Player must be able to view Store
- The System should display all User Planes, Pilots, Bonus Package Locks and Weapons present in the game
- The System should lock the items the Player cannot afford
- The System should highlight the items the Player can afford
- The System should indicate an already purchased item and should not allow it to be purchased again
- The Player must be able to view details of an available item
- The Player should be able to select an item to purchase
- If the Player selects a Weapon to purchase the System should obtain the amount from the Player
- The Player should be able to specify Weapon amount if he wants to purchase a Weapon
- The System should calculate the total price of selected items
- The System must notify the Player if the total amount exceeds the number of Player coins
- The System should notify the Player when purchase is completed
- The System should update Player Collection after a purchase
- The System should update the Store view after a purchase

2.2.5. *Viewing Collection and Changing Preferences*

- The Player must be able to display Collection
- The System should display all purchased items in the Collection
- The System should specify the amount of existing Weapons
- The Player should be able to change UserPlane or Pilot selections
- The System should update the game if the Player changes item selections

2.2.6. *Viewing Level Map*

- The System should display all levels the User has completed
- The System must highlight the current level, the level the User is required to pass in order to open access to next level
- The System should display Bonus Mission if the Player has opened access to it
- The System has to disable access to Bonus Mission after the Player plays it once
- The Player should be able to select any level on Level Map to play
- The System should start the selected level or Bonus Mission
- The System should update the Level Map after completion of a level or Bonus Mission
- The System should update the Level Map after the User destroys a Bonus Mission Target in a level

2.2.7. *Changing Settings*

- The System should display the current settings to User
- The Player must be able to turn the music on or off
- The Player must be able to alter volume level
- The System should record and update changed settings

2.2.8. *Viewing Help*

- The System should display Help page and provide tutorials to the Player

- The System should specify basic concepts of the game
- The System must demonstrate User controls
- The System should explain Store and Collection operations

2.2.9. *Viewing Credits*

- The System should display credits

2.2.10. *Playing Sound*

- The System should play music during levels and Bonus Missions
- The System should play music while the Player is on Menus and other pages
- The System should play animation sounds during gameplay when collision occurs

2.3. Non-Functional Requirements

2.3.1. *Usability*

User-friendliness: The System should be simple and aim user satisfaction
- The System should use meaningful names for buttons and icons in order to ease navigation

Ease of Use: The system should be easy to understand and use
Understandability: The game itself and its documentation should be understandable

2.3.2. *Performance*

High Performance: The system should respond to user input immediately
Rapid Development: The system should be quickly developed
Efficiency: The system should be efficient, operations should be implemented to maximize usage and minimize time cost
Functionality: The system should include many functionalities to satisfy user

2.3.3. *Reliability*

- The System should automatically save the progress of the Player after the end of each level
- The System should not lose the Player progress in a power-loss situation
- The System should not lose the Player progress if the System crushes

Traceability of Requirements: The System development should be coherent with the requirements

2.3.4. *Supportability*

Modifiability: The System should be open to development, it should be possible to add new features to the game
Flexibility: The System should be flexible to adaptations
Maintainability: The System can be changed to adapt to new technology
Adaptability: The System should be able to deal with additional application domain concepts

2.4. Constraints

- The System shall be implemented in Java
- Adobe Photoshop shall be used in the design of game graphics
- Bohemian Coding Sketch shall be used in the design of game screens

2.5. Scenarios

Scenario 1

Use Case Name: OpenLevelMapToPlayLevel1
Actors: Player Ali
Entry Conditions:
- Player Ali is on Main Menu

Exit Conditions:
- Player Ali is playing Level 1

Main Flow of Events:
1. Player Ali presses Level Map button
2. Sky Wars displays Level Map page
3. Player Ali presses Level 1 button
4. Sky Wars starts Level 1

Scenario 2

Use Case Name: CompleteLevel1
Actors: Player Ali
Entry Conditions:
- Player Ali is on Level Map page

Exit Conditions:
- Player Ali completes Level 1 AND
- Player Ali is on Level Map page

Flow of Events:
1. Player selects Level 1 to play from Level Map
2. Sky Wars initializes Level 1
3. Player Ali plays the game
    - **3a.** Player moves the UserPlane up or down
    - **3b.** Sky Wars handles collisions
    - **3c.** Player shoots
    - **3d.** Sky Wars handles shoot
    - **3e.** Sky Wars controls and updates time, points and health of the Player
4. Player Ali completes the level with 1000 points
5. Sky Wars checks whether Player has reached Level 1 threshold, 800 points
6. Sky Wars declares Level 1 as successful since Player Ali has reached threshold
7. Sky Wars turns points into coins
8. Sky Wars directs Player Ali to Level Map
9. Sky Wars displays Level 2 button as well in Level Map

Scenario 3

Use Case Name: FailLevel2
Actors: Player Ali
Entry Conditions:

- Player Ali is on Level Map page

Exit Conditions:

- Player Ali fails Level 2 AND
- Player Ali is on Level Map

Flow of Events:

1. Player selects Level 2 to play
2. Sky Wars initializes Level 2
3. Player Ali plays the game
    - **3a.** Player moves the UserPlane up or down
    - **3b.** Sky Wars handles collisions
    - **3c.** Player shoots
    - **3d.** Sky Wars handles shoot
    - **3e.** Sky Wars controls and updates time, points and health of the Player
4. Player Ali collides with an Obstacle
5. Sky Wars ends the game
6. Sky Wars declares Level 2 as unsuccessful
7. Sky Wars turns points into coins
8. Sky Wars directs Player Ali to Level Map

Scenario 4

Use Case Name: FailLevel2CannotReachTreshold
Actors: Player Ali
Entry Conditions:

- Player Ali is on Level Map page

Exit Conditions:

- Player Ali fails Level 2 AND
- Player Ali is on Level Map

Flow of Events:

1. Player selects Level 2 to play
2. Sky Wars initializes Level 2
3. Player Ali plays the game
    - **3a.** Player moves the UserPlane up or down
    - **3b.** Sky Wars handles collisions
    - **3c.** Player shoots
    - **3d.** Sky Wars handles shoot
    - **3e.** Sky Wars controls and updates time, points and health of the Player
4. Player Ali completes the level with 1500 points
5. Sky Wars checks whether Player has reached Level 2 threshold, 2000 points
6. Sky Wars declares Level 2 as unsuccessful since Player Ali has not reached threshold
7. Sky Wars turns points into coins
8. Sky Wars directs Player Ali to Level Map

Scenario 5

Use Case Name: PlayShipBonusMission
Actors: Player Ali
Entry Conditions:

- Player Ali has opened Bonus Mission in previous levels AND
- Player Ali in on Level Map

Exit Conditions:

- Player Ali is on Level Map

Flow of Events:

1. Player Ali selects Bonus Mission to play
2. Sky Wars initializes Bonus Level which has a theme of destroying below ships with torpedos
3. Player Ali plays the Bonus Mission
    - **3a.** Player Ali moves UserPlane left and right
    - **3b.** Player Ali shoots Torpedoes
    - **3c.** Sky Wars updates points
4. Player Ali completes the Bonus Mission
5. Sky Wars declares Bonus Mission as completed
6. Sky Wars turns points into coins
7. Sky Wars removes Bonus Mission button from Level Map
8. Sky Wars directs User to level map

Scenario 6

Use Case Name: PauseGameFor5MinutesAndContinue
Actors: Player Ali
Entry Conditions:

- Player Ali is playing Level 2

Exit Conditions:

- Player Ali is playing Level 2

Flow of Events:

1. Player Ali presses Pause button
2. Sky Wars pauses the game screen
3. Sky Wars displays Pause Menu
4. Sky Wars stays in pause mode for 5 minutes
5. Player Ali selects to continue level
6. Sky Wars continues the game, Level 2

Scenario 7

Use Case Name: PauseGameLearnAboutBonusPackagesAndContinue
Actors: Player Ali
Entry Conditions:

- Player Ali is playing Level 2

Exit Conditions:

- Player Ali is playing Level 2

Flow of Events:

1. Player Ali presses Pause button
2. Sky Wars pauses the game screen
3. Sky Wars displays Pause Menu
4. Player Ali presses Help button from Pause Menu
5. Player Ali views tutorials about Bonus Packages
6. Player Ali selects to return to Pause Menu
7. Player Ali selects to continue level
8. Sky Wars continues the game, Level 2

Scenario 8

Use Case Name: TurnOffMusic
Actors: Player Ali
Entry Conditions:

- Player Ali is on Main Menu

Exit Conditions:

- Player Ali is on Main Menu

Main Flow of Events:

1. Player Ali presses Settings button
2. Sky Wars displays Settings page
3. Player Ali views relative buttons for turning the music on or off
4. Player Ali turns music off
5. Player presses Main Menu button
6. Sky Wars updates the settings
7. Sky Wars displays Main Menu or Pause Menu

Scenario 9

Use Case Name: SetVolumeLevelToMaximum
Actors: Player Ali
Entry Conditions:

- Player Ali is on Main Menu

Exit Conditions:

- Player Ali is on Main Menu

Main Flow of Events:

1. Player Ali presses Settings button
2. Sky Wars displays Settings page
3. Player Ali views relative buttons for altering volume level
4. Player Ali sets volume to maximum
5. Player presses Main Menu button
6. Sky Wars updates the settings
7. Sky Wars displays Main Menu or Pause Menu

Scenario 10

Use Case Name: LearnHowToHandleStoreOperations
Actors: Player Ali
Entry Conditions:

- Player Ali is on Main Menu

Exit Conditions:

- Player Ali is on Main Menu

Main Flow of Events:

1. Player Ali presses Help button
2. Sky Wars displays Help page
3. Player Ali examines tutorials about Store operations
4. Player presses Main Menu button
5. Sky Wars displays Main Menu

Scenario 11

Use Case Name: CheckAffordableWeapons
Actors: Player Ali

Entry Conditions:

- Player Ali is on Main Menu

Exit Conditions:

- Player Ali is on Store page

Main Flow of Events:

1. Player Ali presses Store button
2. Sky Wars locks the items on Store that Player cannot afford
3. Sky Wars displays Store page
4. Player Ali views unlocked Weapons and their details

Scenario 12

Use Case Name: PurchaseUserPlaneF22Raptor
Actors: Player Ali
Entry Conditions:

- Player Ali is on Store page

Exit Conditions:

- Player Ali is on Main Menu

Flow of Events:

1. Sky Wars displays Store
2. Player Ali views unlocked UserPlanes
3. Player selects F22 Raptor UserPlane
4. Player presses Purchase button to complete purchase
5. Sky Wars handles purchase and updates Player Ali's Collection
6. Player presses Main Menu button
7. Sky Wars displays Main Menu

Scenario 13

Use Case Name: Purchase10FlameGuns
Actors: Player Ali
Entry Conditions:

- Player Ali is on Store page

Exit Conditions:

- Player Ali is on Main Menu

Flow of Events:

1. Sky Wars displays Store
2. Player Ali views unlocked Weapons
3. Player selects FlameGun
4. Sky Wars requests from Player Ali to enter number of FlameGuns
5. Player Ali enters '20'
6. Player presses Purchase button to complete purchase
7. Sky Wars cancel purchase and notify User that there not enough coins
8. Player Ali reenters FlameGun amount, '10' this time
9. Sky Wars handles purchase and updates Player Ali's Collection
10. Player presses Main Menu button
11. Sky Wars displays Main Menu

Scenario 14

Use Case Name: ViewPurchasedWeapons
Actors: Player Ali

Entry Conditions:

- Player Ali is on Main Menu

Exit Conditions:

- Player Ali is on Main Menu

Flow of Events:

1. Player Ali presses Collection button
2. Sky Wars displays Collection page with the purchased items
3. Player Ali views purchased Weapons
4. Player Ali presses Main Menu button
5. Sky Wars displays Main Menu

Scenario 15

Use Case Name: ChangeSelectedPlaneToF22
Actors: Player Ali
Entry Conditions:

- Player Ali is on Collection page

Exit Conditions:

- Player is Ali on Main Menu

Flow of Events:

1. Sky Wars displays Collection page with the purchased items
2. Sky Wars highlights current UserPlane and Pilot selections
3. Player Ali changes current UserPlane to F22
4. Sky Wars updates Player selection
5. Player presses Main Menu button
6. Sky Wars displays Main Menu

Scenario 16

Use Case Name: DisplayNamesOfDevelopers
Actors: Player Ali
Entry Conditions:

- Player Ali is on Main Menu

Exit Conditions:

- Player Ali is on Main Menu

Flow of Events:

1. Player Ali presses Credits button
2. Sky Wars displays Credits page
3. Player Ali views names of developers
4. Player Ali presses Main Menu button
5. Sky Wars displays Main Menu

Scenario 17

Use Case Name: QuitGameFromMainMenu
Actors: Player Ali
Entry Conditions:

- Player Ali is on Main Menu

Exit Conditions:

- Player Ali exited the game

Flow of Events:

1. Player Ali presses Quit button on Main Menu
2. System displays a dialog box to make sure Ali wants to quit
3. Player Ali presses Yes button
4. System is exited

Scenario 18

Use Case Name: QuitGameDuringLevel
Actors: Player Ali
Entry Conditions:

- Player Ali is playing Level 2

Exit Conditions:

- Player Ali exited the game

Flow of Events:

1. Player Ali presses Pause button
2. System displays Pause Menu
3. Player Ali presses Quit button
4. System displays a dialog box to make sure Ali wants to quit
5. Player Ali presses Yes button
6. System is exited

2.6. *Use Case Models*

Use cases of Sky Wars are represented with the use case diagram. Verbal descriptions of use cases are included below.

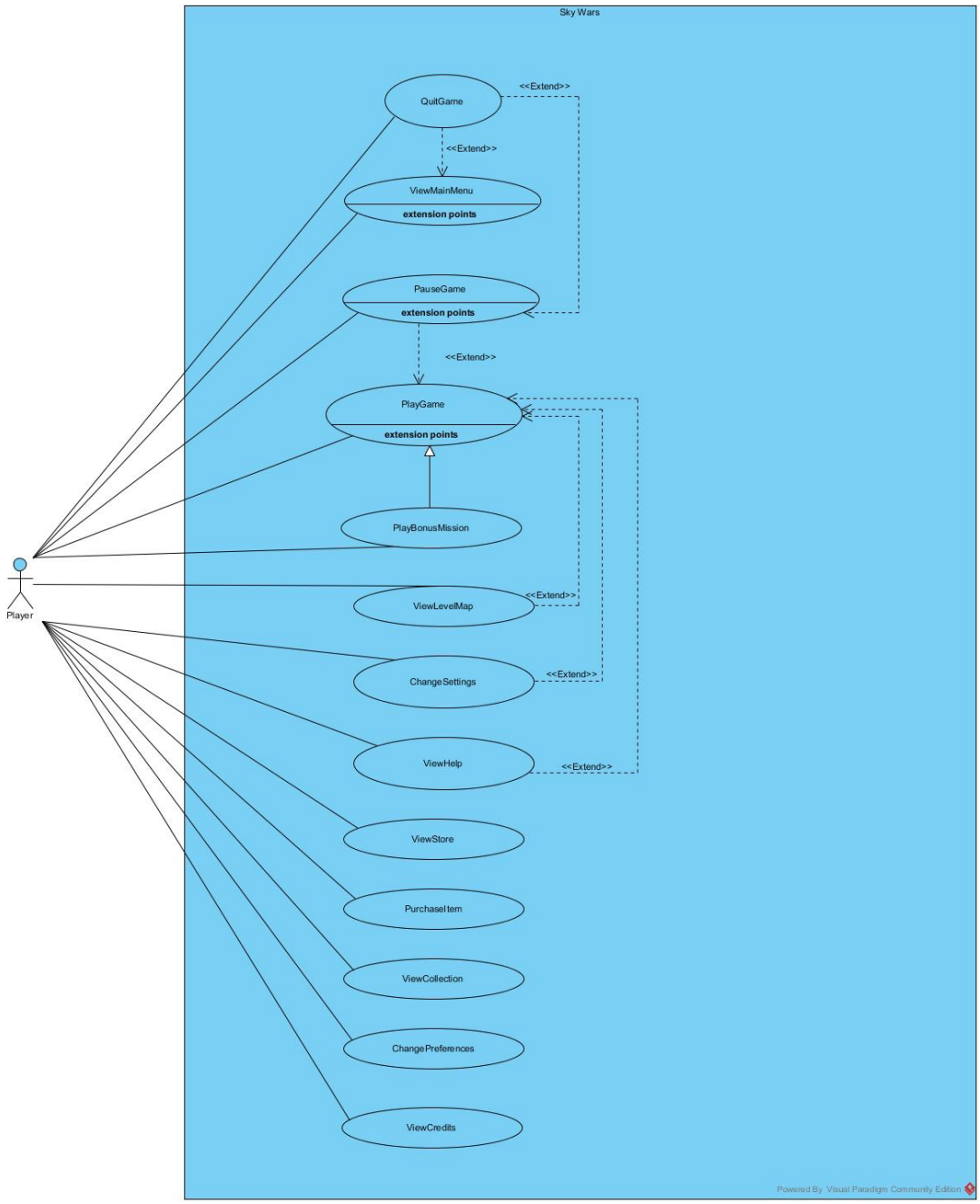

Figure34 Use Case Diagram of Sky Wars

ViewMainMenu: Player can request to view Main Menu from all screens of Sky Wars
ViewLevelMap: Player can request to view Level Map to select a level to play
ChangeSettings: Player can view Settings page and change game Settings
ViewHelp: Player can view Help page to learn how to play the game
ViewStore: Player can view Store page to examine items that are open to purchase
PurchaseItem: Player can select an item from Store and purchase it
ViewCollection: Player can request to view Collection page and examine list of all purchased items
ChangePreferences: Player can change Pilot or UserPlane selections from Collection View.
ViewCredits: Player can request to view Credits page to see developer details
PlayGame: Player can request to play the game levels by selecting a level from Level Map page
PauseGame: Player can request to pause the game while playing the game
PlayBonusMission: As a part of the gameplay, Player can play Bonus Mission which appears on Level

Map page.
QuitGame: Player can request to quit the game from Main Menu or Pause Menu

2.6.1. *View Main Menu*

Use Case Name: ViewMainMenu
Actors: Player
Entry Conditions:
- Player is on Store Screen OR
- Player is on Level Map Screen OR
- Player is on Collection Screen OR
- Player is on Settings Screen OR
- Player is on Credits Screen OR
- Player is on Help Screen OR
- Player is on Pause Menu

Exit Conditions:
- Player is on Main Menu

Main Flow of Events:
1. Player presses Main Menu button
2. Sky Wars displays Main Menu

2.6.2. *View Level Map*

Use Case Name: ViewLevelMap
Actors: Player
Entry Conditions:
- Player is on Main Menu OR
- Player is on Pause Menu

Exit Conditions:
- Player is on Level Map page OR
- Player is playing Level

Main Flow of Events:
1. Player presses Level Map button
2. Sky Wars displays Level Map page
3. Player views relative buttons for completed levels and current level
4. Player views Bonus Mission button
5. Player presses Main Menu button
6. Sky Wars displays Main Menu

Alternative Flow of Events:
5A. Player selects a level or Bonus Mission to play (Skip Step 5 and 6)
6A. Sky Wars starts the game

2.6.3. *Change Settings*

Use Case Name: ChangeSettings
Actors: Player
Entry Conditions:

- ChangeSettings use case extends PlayGame use case
- Player is on Main Menu OR
- Player is on Pause Menu

Exit Conditions:

- Player is on Main Menu OR
- Player is on Pause Menu

Main Flow of Events:

1. Player presses Settings button
2. Sky Wars displays Settings page
3. Player views relative buttons for setting volume level
4. Player views relative buttons for turning the music on or off
5. Player changes settings by turning music on or off
6. Player alters volume level
7. Player presses Main Menu or Pause Menu button
8. Sky Wars updates the settings
9. Sky Wars displays Main Menu or Pause Menu

Alternative Flow of Events:

5A. Player does not turn music on or off
6A. Player does not alter volume level

2.6.4. *View Help*

Use Case Name: ViewHelp
Actors: Player
Entry Conditions:

- ViewHelp use case extends PlayGame use case
- Player is on Main Menu OR
- Player is on Pause Menu

Exit Conditions:

- Player is on Main Menu OR
- Player is on Pause Menu

Main Flow of Events:

1. Player presses Help button
2. Sky Wars displays Help page
3. Player views tutorials to learn how to play Sky Wars
4. Player presses Main Menu button
5. Sky Wars displays Main Menu or Pause Menu

2.6.5. *View Store*

Use Case Name: ViewStore
Actors: Player
Entry Conditions:

- Player is on Main Menu

Exit Conditions:

- Player is on Main Menu OR
- Player is in Store Page

Main Flow of Events:

1. Player presses Store button
2. Sky Wars locks the items on Store that Player cannot afford
3. Sky Wars displays Store page
4. Player views UserPlane, Pilot, Weapon and BonusPackage items, some of them ready to be purchased and some of them locked
5. Player presses Main Menu button
6. Sky Wars displays Main Menu

Alternative Flow of Events:

5A. Player initiates PurchaseItem use case
6A. Sky Wars starts PurchaseItem operations

2.6.6. *Purchase Item*

Use Case Name: PurchaseItem
Actors: Player
Entry Conditions:

- Player is on Store page
- PurchaseItem use case extends ViewStore

Exit Conditions:

- Player is on Main Menu

Main Flow of Events:

1. Sky Wars displays Store
2. Player views UserPlane, Pilot, Weapon and Bonus Package Lock items, some of them ready to be purchased and some of them locked
3. Player selects Weapon item to purchase
4. Sky Wars asks multiplicity of Weapon
5. Player presses Purchase button to complete purchase
6. Sky Wars checks Player coins whether they are enough or not
7. Sky Wars completes purchase and updates Player collection
8. Player presses Main Menu button
9. Sky Wars displays Main Menu

Alternative Flow of Events:

3A. Player selects UserPlane, Pilot or Bonus Package to purchase (Skip 4,5 and 6)
6A. Sky Wars detect that Player coins are not enough
7A. Sky Wars indicate that purchase cannot be completed

2.6.7. *View Collection*

Use Case Name: ViewCollection
Actors: Player
Entry Conditions:

- Player is on Main Menu

Exit Conditions:

- Player is on Main Menu

Main Flow of Events:

1. Player presses Collection button
2. Sky Wars displays Collection page with the purchased items
3. Sky Wars highlights current UserWeapon and Pilot selections

4. Player views UserPlane, Pilot, Weapon and BonusPackage items
5. Player views current selections
6. Player presses Main Menu button
7. Sky Wars displays Main Menu

Alternative Flow of Events:

6A. Player initiates ChangePreferences use case
7A. Sky Wars starts ChangePreferences operations

2.6.8. *Change Preferences*

Use Case Name: ChangePreferences
Actors: Player
Entry Conditions:

- Player is on Collection page
- ChangePreferences use case extends ViewCollection use case

Exit Conditions:

- Player is on Main Menu

Main Flow of Events:

1. Sky Wars displays Collection page with the purchased items
2. Sky Wars highlights current UserWeapon and Pilot selections
3. Player views UserPlane, Pilot, Weapon and BonusPackage items
4. Player views current selections
5. Player changes current selection
6. Sky Wars updates Player selections
7. Player presses Main Menu button
8. Sky Wars displays Main Menu

Alternative Flow of Events:

5A. Player changes UserPlane selection
5A. Player changes Pilot selection
5A. Player does not chance preferences(Skip 5 and 6)

2.6.9. *View Credits*

Use Case Name: ViewCredits
Actors: Player
Entry Conditions:

- Player is on Main Menu

Exit Conditions:

- Player is on Main Menu

Main Flow of Events:

1. Player presses Credits button
2. Sky Wars displays Credits page
3. Player views credits to learn developer details
4. Player presses Main Menu button
5. Sky Wars displays Main Menu

2.6.10. *Play Game*

Use Case Name: PlayGame
Actors: Player
Entry Conditions:
- Player has selected a level from Level Map

Exit Conditions:
- Player completed level and returned to Level Map OR
- Player failed the level and returned to Level Map OR
- Player failed the level and requested to play again

Main Flow of Events:
1. Player selects a level to play
2. Sky Wars initializes level
3. Player plays the game
    - **3a.** Player moves the UserPlane up or down
    - **3b.** Sky Wars handles collisions
    - **3c.** Player shoots
    - **3d.** Sky Wars handles shoot
    - **3e.** Sky Wars controls and updates time, points and health of the Player
4. Player completes the level
5. Sky Wars checks whether Player has reached level threshold
6. Sky Wars declares level successful
7. Sky Wars turns points into coins
8. Sky Wars directs User to level map
9. Sky Wars opens access to next level

Alternative Flow of Events:
4A. Player depletes his health and fails the level (Skip 5)
6A. Sky Wars declares level unsuccessful
9A. Sky Wars does not open access to next level
6A. Sky Wars declares level unsuccessful since Player could not reach point threshold
9A. Sky Wars does not open access to next level

2.6.11. *Play Bonus Mission*

Use Case Name: PlayBonusMission
Actors: Player
Entry Conditions:
- PlayBonusMission use case inherits PlayGame use case
- Player has opened Bonus Mission in previous levels
- Player has selected Bonus Mission from Level Map

Exit Conditions:
- Player is on Level Map

Main Flow of Events:
1. Player selects Bonus Mission to play
2. Sky Wars initializes level
3. Player plays the Bonus Mission
4. Player completes the Bonus Mission
5. Sky Wars turns points into coins
6. Sky Wars removes Bonus Mission button from Level Map
7. Sky Wars directs User to Level Map

2.6.12. *Pause Game*

Use Case Name: PauseGame
Actors: Player
Entry Conditions:
- PauseGame use case extends PlayGame use case
- Player is playing a Level OR
- Player is playing a Bonus Mission

Exit Conditions:
- Player is on Level Map OR
- Player is playing the game
- Player changes Settings
- Player views Help page

Main Flow of Events:
1. Player presses pause button
2. Sky Wars pauses the game screen
3. Sky Wars displays Pause Menu
4. Player selects to continue level
5. Sky Wars continues the game

Alternative Flow of Events:
4A. Player presses Settings button
5A. Sky Wars displays Settings page
4A. Player presses Help button
5A. Sky Wars displays Help page

2.6.13. *Quit Game*

Use Case Name: QuitGame
Actors: Player
Entry Conditions:
- QuitGame use case extends PauseGame and ViewMainMenu use cases
- Player is on Pause Menu OR
- Player is on Main Menu

Exit Conditions:
- Player has exited the game

Main Flow of Events:
1. Player presses quit button
2. Sky Wars is exited
3. Player is directed to desktop

2.7. *User Interface*

2.7.1. *Navigational Path*

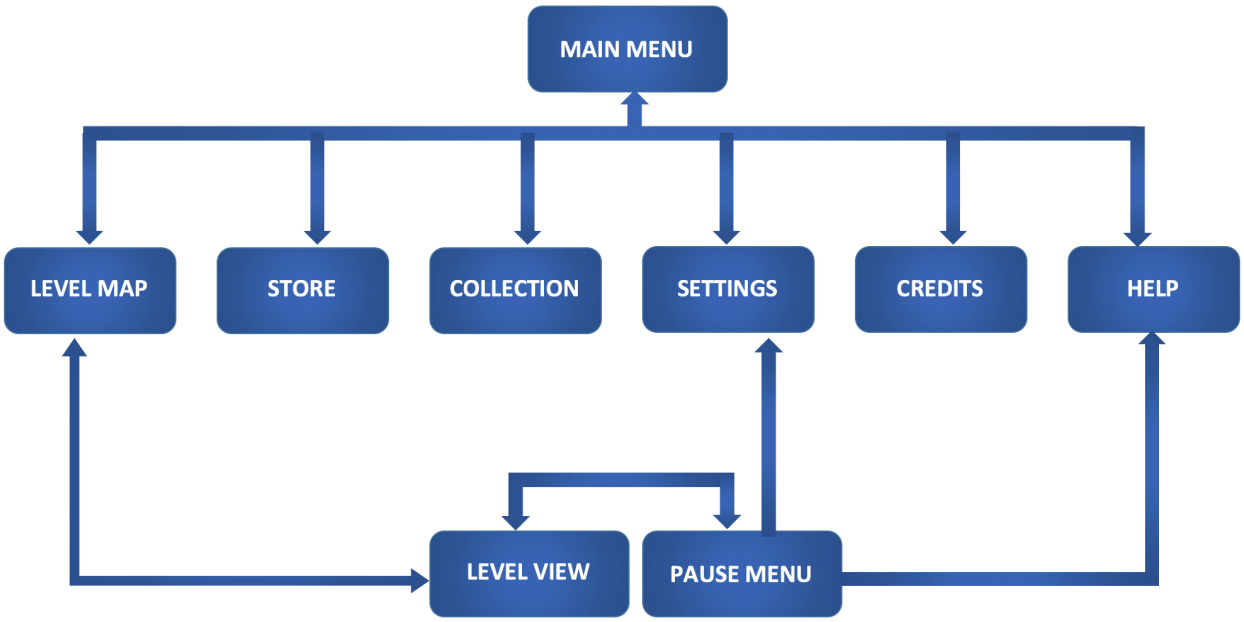

Figure 35 Navigational Path of Sky Wars

2.7.2. *Main Menu Screen*

Main Menu screen is the first screen that is displayed when the game is started. Main Menu leads User to 'Level Map', 'Collection' and 'Store' screens when relative buttons represented with clouds are clicked. Moreover, clicking the items in bubbles direct User to 'Help', 'Settings' and 'Credits' pages from top to bottom. The icon on right top of the page is for quitting the game.

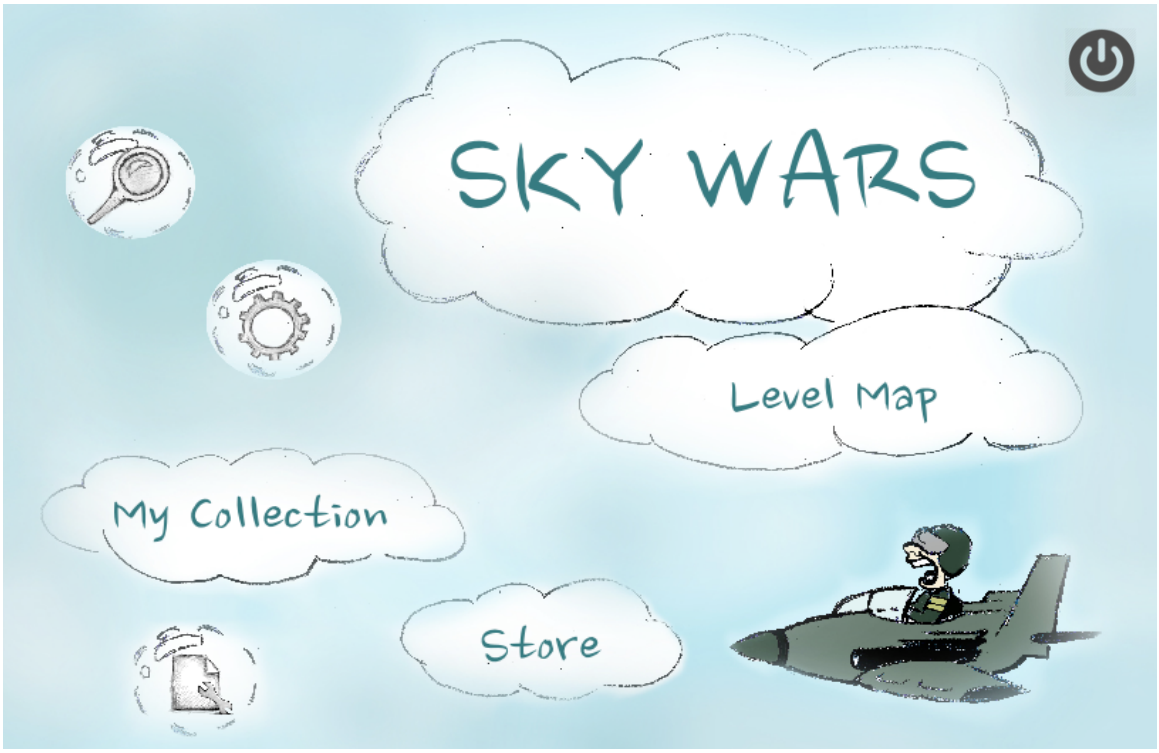

Figure 36 Main Menu Screen

2.7.3. *Level Map Screen*

Level Map screen displays all levels in Sky Wars. Current level, Level 5 is highlighted. Additionally, present icon can be clicked to play Bonus Mission. Home icon directs user to Main Menu.

Figure 37 Level Map Screen

2.7.4. *Level Play Screen*

Level Play Screen is an example of how the actual game looks like. A sample screen is demonstrated with Pilot and UserPlane on the left and other GameObjects around the screen. Pause button directs User to Pause Menu.

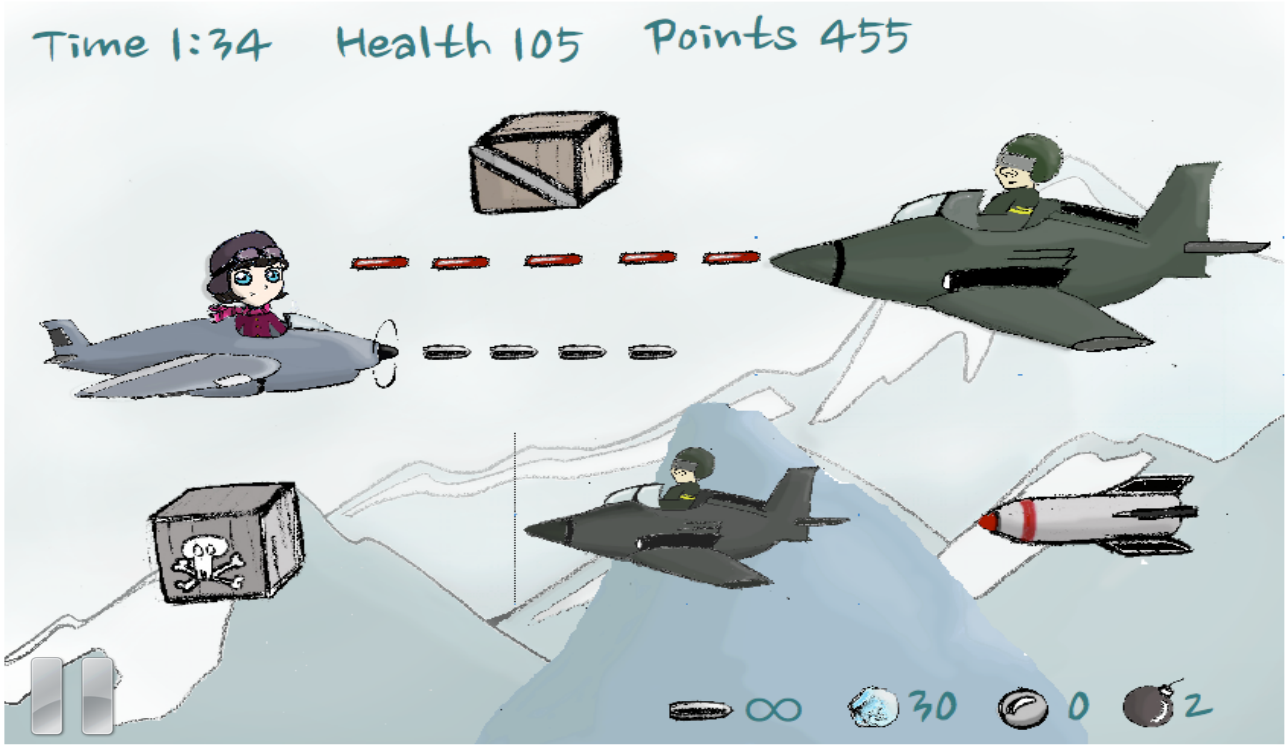

Figure 38 Level Play Screen

2.7.5. *Store Screen*

Store Screen lists all purchasable GameObjects hence it is a scrollable page. Below scene is the representation of the complete page. The detailed information and price of items are provided. The items Player cannot afford are locked. Moreover, the items that are already purchased are also unavailable. User coins are shown on the upper left corner of the screen. User can return to Main Menu by clicking on Home icon.

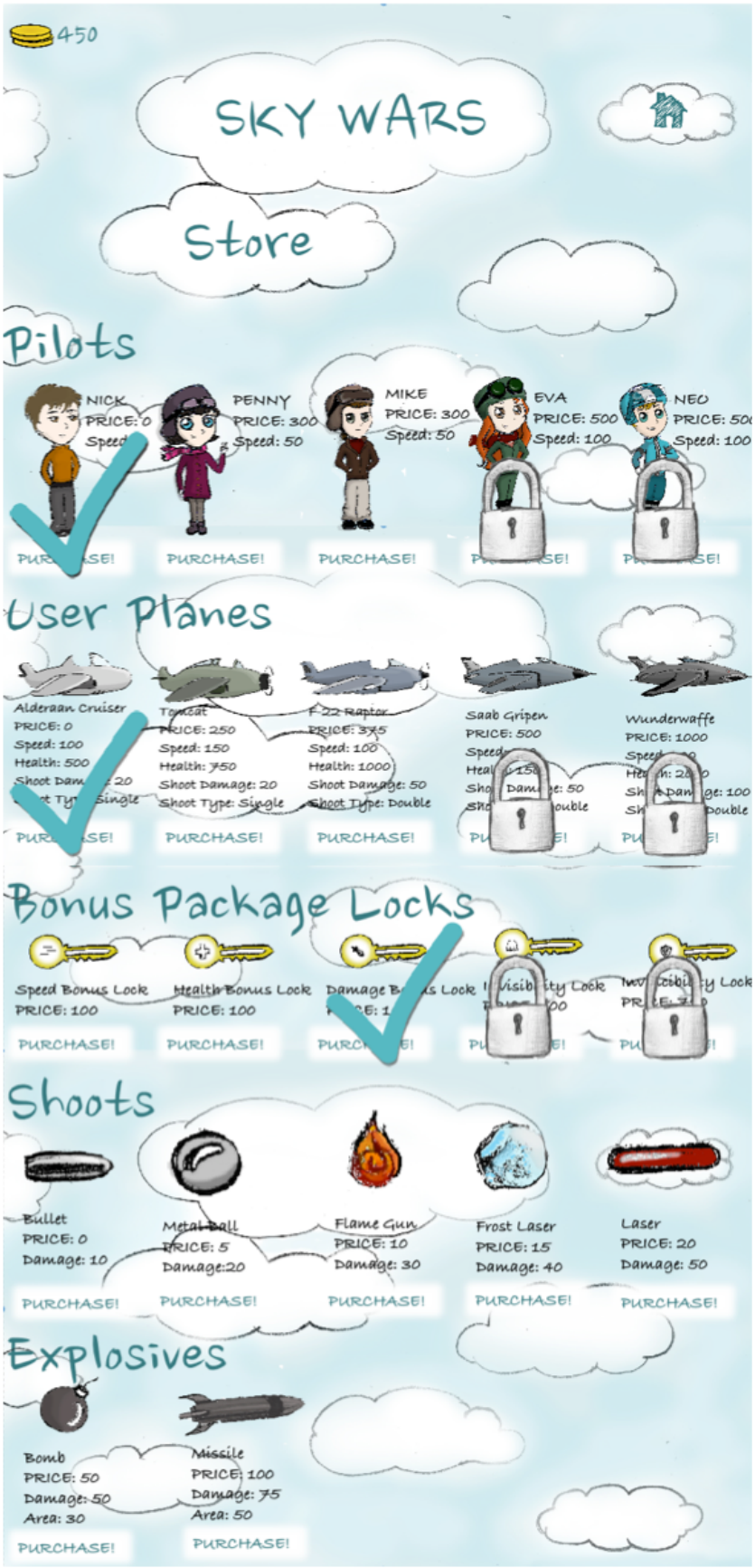

Figure 39 Store Screen

2.7.6. *Collection Screen*

Collection Screen lists all purchased items hence it is also a scrollable page. Below scene shows entire Collection page. The detailed information of purchased items is provided. The current selection of UserPlane and Pilot is indicated in the left side of the screen. The user can change selection by clicking on any Pilot. User coins are shown on the upper left corner of the screen. User can return to Main Menu by clicking on Home icon.

Figure 40 Collection Screen

2.7.7. Settings *Screen*

Setting Screen allows user to change volume level by clicking on ‘plus’ and ‘minus’ signs near volume signal and turn music on or off by switching Note icon on or off. User can return to Main Menu by clicking on Home icon.

Figure 41 Settings Screen

2.7.8. *Help Screen*

Help Screen provides User a video that explains how to play Sky Wars. The User can control video with Video Player icons. User can return to Main Menu by clicking on Home icon.

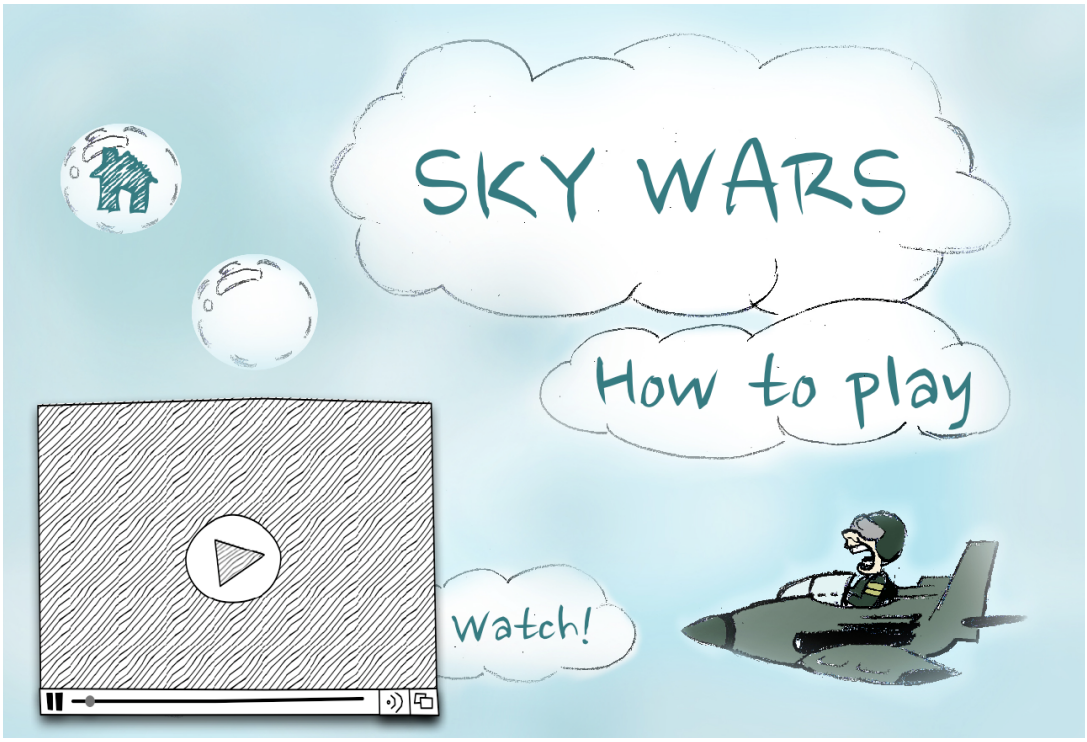

Figure 42 Help Screen

2.7.9. *Credits Screen*

Credits Screen display User the developer names, publication date and place. User can return to Main Menu by clicking on Home icon.

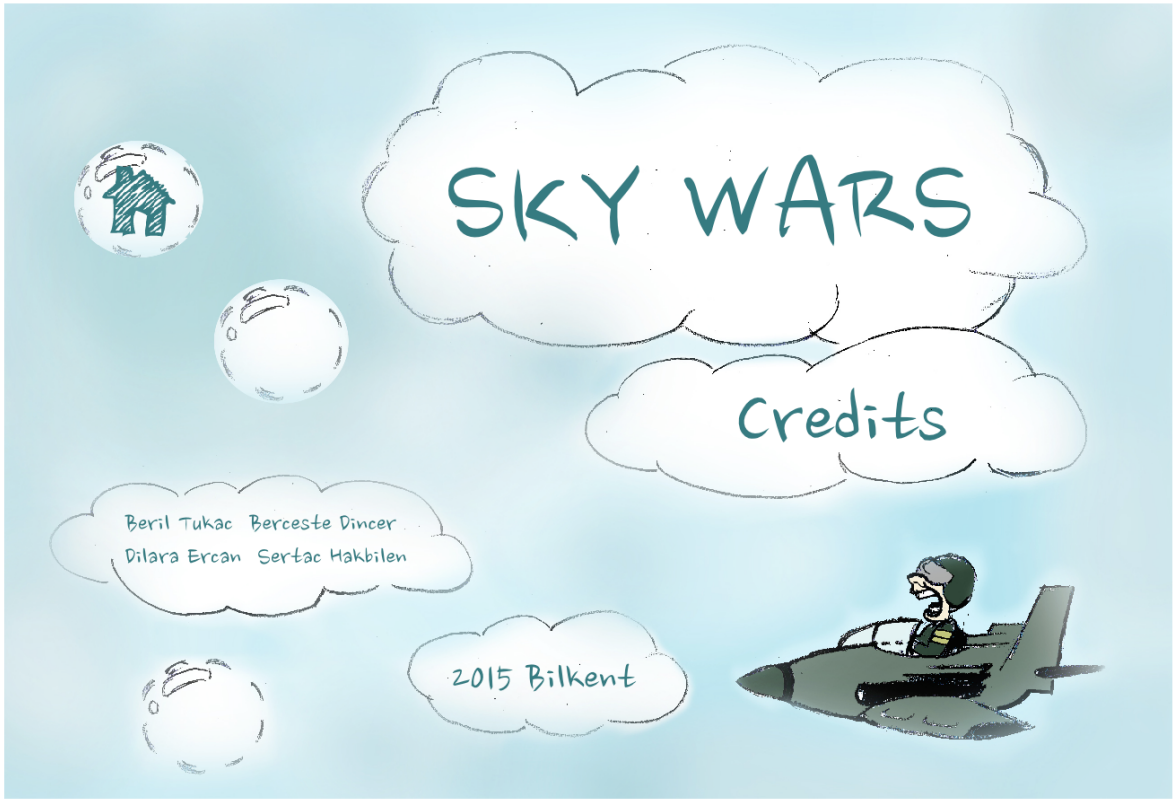

Figure 43 Credits Screen

2.7.10. *Pause Menu Screen*

Pause Menu Screen is the Menu displayed when the game is paused. The User can click on Continue button to continue playing game or can press Quit button to exit Level and return to Level Map. The icons in clouds direct User to ‘Help’ and ‘Settings’ screens from top to bottom. The icon on right top of the page is for quitting the game.

Figure 44 Pause Menu Screen

3. Analysis
3.1.Object Model
3.1.1.Domain Lexicon

User/Player: Person who plays and controls Sky Wars

Game: Concept of overall system, Sky Wars

Level: Small parts of the game which has a time limit and point threshold. A level is opened after previous level is completed.

Bonus Mission: A different kind of Level independent from the level flow which Player earns access during regular level and can play once

GameObject: Any item visible on the gameplay screen.

Weapon: A GameObject which can be shot by Planes, UserPlane or TargetPlane and gives damage to other GameObjects

Shoot: A Weapon that can effect only the GameObject it collided with

Explosive: A Weapon that creates an explosion and damages nearby objects

Pilot: A character figure which represents Player within the game

UserPlane: The plane controlled by the Player

BonusPackage: A GameObject which creates different bonuses, unexpected variations in the basic game flow. BonusPackages named PresentBonusPackages can help Player, make game easier and help earning point while TrapBonusPackages make the game harder and lead to loss of points.

Point: Success unit of the game. Any damage given to enemies, increase Player points while any damage of UserPlane decrease Player points. Points are calculated level based.

Coin: The representation of total amount of points gained in all level plays that are kept in Player Account.

Time: The amount of minutes specified for each level

Target: Any GameObject that has a certain health and can be shot by the User

Health: A property Targets and UserPlane has which represents the left damage resistance of a GameObject. Whenever a Target or UserPlane is shot, its health is decreased.

Collision: The touch between any two GameObjects

Level Map: Map of all Levels and Bonus Missions

Store: The place where GameObjects are sold

Collection: The list of all items Player has purchased

3.1.2.Class Diagrams

Figure 45 Class Diagram of Sky Wars

Boundary Classes

GameFrame is the basic Boundary class that contains all panels and allows User to interact with the system

PauseMenuPanel is the JPanel of Pause Menu

MainMenuPanel is the JPanel of Main Menu

StorePanel is the JPanel of Store Screen
SettingsPanel is the JPanel of Settings Screen
CollectionPanel is the JPanel of Collection Screen
HelpPanel is the JPanel of Help Screen
CreditsPanel is the JPanel of Credits Screen
LevelMapPanel is the JPanel of Level Map Screen
LevelPanel is the JPanel of Level Screen
GameManagerObserver is the interface of the observers of GameManager
GameObjectInterface is the interface of the GameManager subject

Controller Classes

GameManager is the fundamental Controller class which handles all game operations
FileManager handles files that specify level details
SoundManager plays game music and sounds
CollectionManager handles Collection related operations
LevelManager controls the GameObjects, their creations, movements and operations within the Level play
CollisionManager is the abstract class for CollisionManagers
UserPlaneDestroyedCollisionManager is the CollisionManager that handles user plane and target and obstacle collisions
UserPlaneHitCollisionManager is the CollisionManager that handles user plane and weapon collisions
BonusPackageCollisionManager is the CollisionManager that handles user plane and bonus package collisions
TargetHitCollisionManager is the CollisionManager that handles target and weapon collisions
CollisionPolicy is the policy class for determining collision strategy

Entity Classes

Level holds all GameObjects within level
Target is a GameObject which can be shot by the Player
TargetPlane, Rocket, Ship, Carriage, Ally and **BonusMissionTarget** are different types of Target with various properties. They all have corresponding enumeration classes specifying pre-defined values for different instances of the class
UserPlane is the Plane controlled by the Player which can move and shoot. It has a corresponding enumeration class
Pilot is the character which controls the UserPlane. It has a corresponding enumeration class
Obstacle is the objects that kill user when collided. It has a corresponding enumeration class
Weapon class represents the object that can be shot by planes and damages GameObjects by decreasing their health
Shoot is a type of Weapon which only decreases the health of the collided object. It has a corresponding enumeration class
Explosive is a type of Weapon that causes an explosion and damages all objects within a certain area. It has a corresponding enumeration class
BonusPackage is a GameObject which creates different bonuses
PresentBonusPackage is a BonusPackage that helps the Player and boosts the game
SpeedBonusPackage, DamageBonusPackage, CoinPackage, TimeBonusPackage, HealthBonusPackage, InvincibilityPackage, ObstacleInvisibilityPackage are classes that represent different PresentBonusPackages
TrapBonusPackage is a BonusPackage that makes the game difficult for the User
PlaneEnlargePackage, SpeedTrapPackage, DamageTrapPackage and **HealthTrapPackage** are

classes that represent various TrapBonusPackages

GameObject is the basic class which represents objects within Game Level, UserPlane, Pilot, Weapon, BonusPackage and Target classes inherit GameObject class

Movement is the class that is responsible from the movement of GameObject

3.2. Dynamic Models

3.2.1.State Chart and Activity Diagrams

3.2.1.1.State Chart Diagram of UserPlane

The below State Chart diagrams demonstrates the dynamic behavior of UserPlane class. State Chart diagram for UserPlane shall be examined in 4 subgroups, damage, speed, size and health. All these activities occur concurrently.

When the game is started, UserPlane is in 'Full Health', 'Default Size', 'Default Speed' and 'Default Damage' states. When the UserPlane collides with a DamageBonusPackage, the UserPlane goes to 'Increased Damage' state, its damage value is increased. UserPlane stays in 'Increased Damage' state until the time period for BonusPackage ends. At the end of the period UserPlane returns to 'Default Damage' state. Similarly, when the Player collides with a DamageTrapPackage, the UserPlane goes to 'Decreased Damage' state. Again at the end of the BonusPackage time, UserPlane goes back to 'Default Damage' state.

Just like damage state flow, when the UserPlane collides with SpeedBonusPackage, the UserPlane goes to 'Increased Speed' state, its speed value is increased. UserPlane stays in 'Increased Speed' state until the time period for BonusPackage ends. At the end of the period UserPlane returns to 'Default Speed' state. Similarly, when the Player collides with a SpeedTrapPackage, the UserPlane goes to 'Decreased Speed' state. Again at the end of the BonusPackage time, UserPlane goes back to 'Default Speed' state.

When the UserPlane collides with a PlaneEnlargePackage, the object passes to 'Enlarged' state. Until the time period for BonusPackage ends the UserPlane states in the same state and then goes back to 'Standard Size' state.

When a Weapon collides with the UserPlane, its health is decreased hence it passes to 'Decreased Health' state. Moreover, when the UserPlane collides with a HealthBonusPackage it goes to 'Increased Health' state, its health value is increased. Similarly, when the Player collects a HealthTrapPackage the UserPlane passes to 'Decreased Health' state. From each health related state, the UserPlane passes to 'Health Depleted' state when the UserPlane collides with an Obstacle or Target. 'Health Depleted' state is the final state since the game is over when the health is depleted.

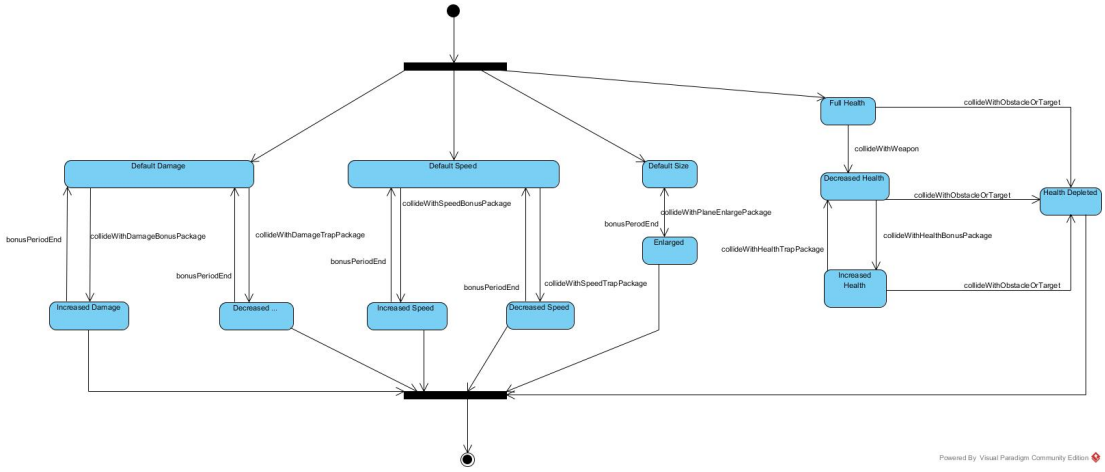

Figure 46 State Chart Diagram of UserPlane

3.2.1.2. Activity Diagram for Overall Game Flow

The below diagram represents the overall dynamic behavior of Sky Wars, game navigations and operations are explained with activity flow.

When the Sky Wars is opened, Main Menu page is displayed. Various activities can be invoked from Main Menu according to Player input. If the Player presses Level Map button Level Map View activity is started. If the Player selects a level or Bonus Mission to play from Level Map Level Play activity is invoked. During Level Play activity, the Player can select to pause the game, which directs game flow to Pause Menu View activity. The Level Play activity continues when the Player selects to continue to play.

When Store button is clicked in the Main Menu View activity, the system moves to Store View activity. Purchase Item activity can be invoked from Store page when the User presses Purchase button.

The Player can press Collection button to move to Collection View activity. From the Collection page, the Player can pass to Change Preferences activity by changing Player item selections.

After Main Menu View activity Settings View activity can be invoked as well if the Player clicks Settings button. When the Player changes Settings game flow moves to Change Settings activity.

When the Player presses Help and Credits buttons, Help View and Credits View activities are invoked respectively. Activity flow passes to Main Menu from all activities when the Player clicks Main Menu button.

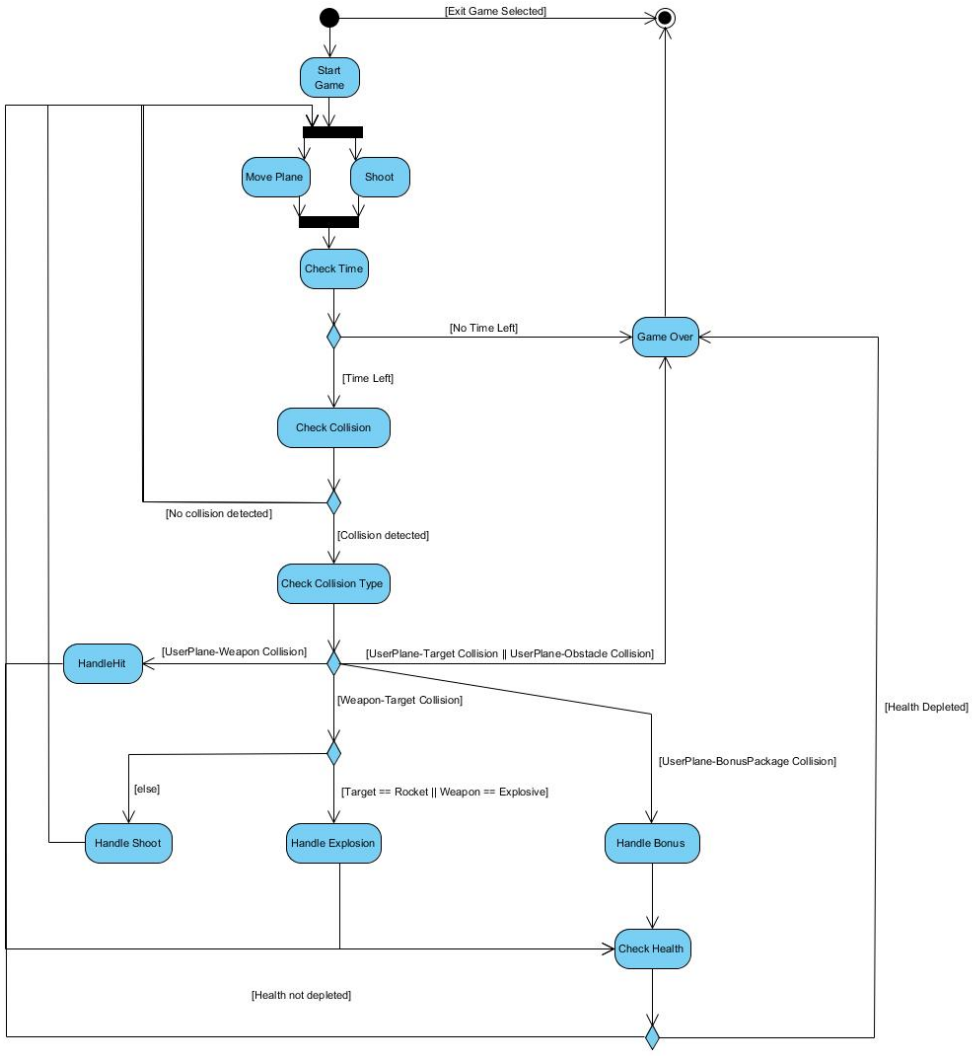

Figure 47 Activity Diagram of Overall Game Flow

3.2.1.3. Activity Diagram for Game Play

The above diagram represents the overall dynamic behavior of Sky Wars game play. The diagram demonstrates activity flow within game play, User actions and system responses.

The level starts with 'Start Game' activity. After the level is constructed the Player is given the option to Move Plane or Shoot. These activities occur concurrently. Then the system moves to 'Check Time' activity. If the time is over, the flow passes to 'Game Over' activity and the flow stops afterwards. If there is time 'Check Collision' activity is initialized. If no collision exists then the flow returns to Player activities, 'Move Plane' and 'Shoot'. Otherwise 'Check Collision Type' activity is initialized. If the collision has occurred between UserPlane and Obstacle or Target, game is over and the game flow ends. If the collision type is UserPlane-Weapon then the game flow passes to 'Handle Hit' activity and necessary operations are performed by the system. Afterwards 'Check Health' activity is initiated. If the heath is depleted the game flow ends. Otherwise, the flow is directed back to Player activities. When the collision occurs between UserPlane and BonusPackage the System initializes 'Handle Bonus' activity. Afterwards the game flow connects to 'Check Health' activity. The fourth possibility is that the Weapon sent by the Player has collided with Target. Then the game flow makes another decision. If the sent Weapon is an Explosive or the Target is Rocket then the system is expected to 'Handle Explosion'. After this activity the game flow connects to 'Check Health' activity. If the Target was not an Explosive 'Handle Shoot' activity is started. This activity directs the game flow back to 'Move Plane' and 'Shoot' activities.

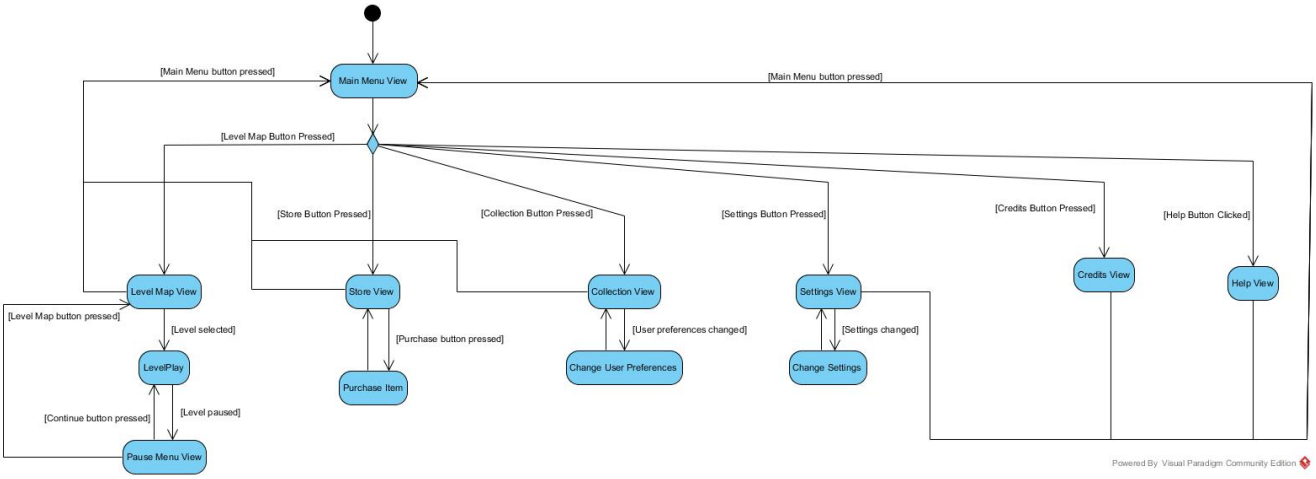

Figure 48 Activity Diagram of Game Play

3.2.2.Sequence Diagrams

3.2.2.1. Start Game

Scenario: Player Ali requests to view Level Map by pressing relative button in the GameFrame. GameFrame loads Level Map View. Ali chooses a level, Level 1, from the Level Map by pressing the relative level icon. System requests GameManager to initialize the level. GameManager conducts the message to LevelManager. LevelManager creates the corresponding Level. Level requests all purchased items from CollectionManager. Level creates UserPlane and Pilot. Then the created Level is loaded to LevelManager. LevelManager asks FileManager to createLevelFromFile. FileManager creates all objects of the level, a TargetPlane, a Target, an Obstacle and a BonusPackage in this scenario. Then FileManager sends these created objects to LevelManager and LevelManager sends them to Level. Consequently, all objects are created and added to Level and LevelManager. Finally, LevelManager requests all object lists, targets, target planes, obstacles and bonus packages from Level in order to initialize its own lists.

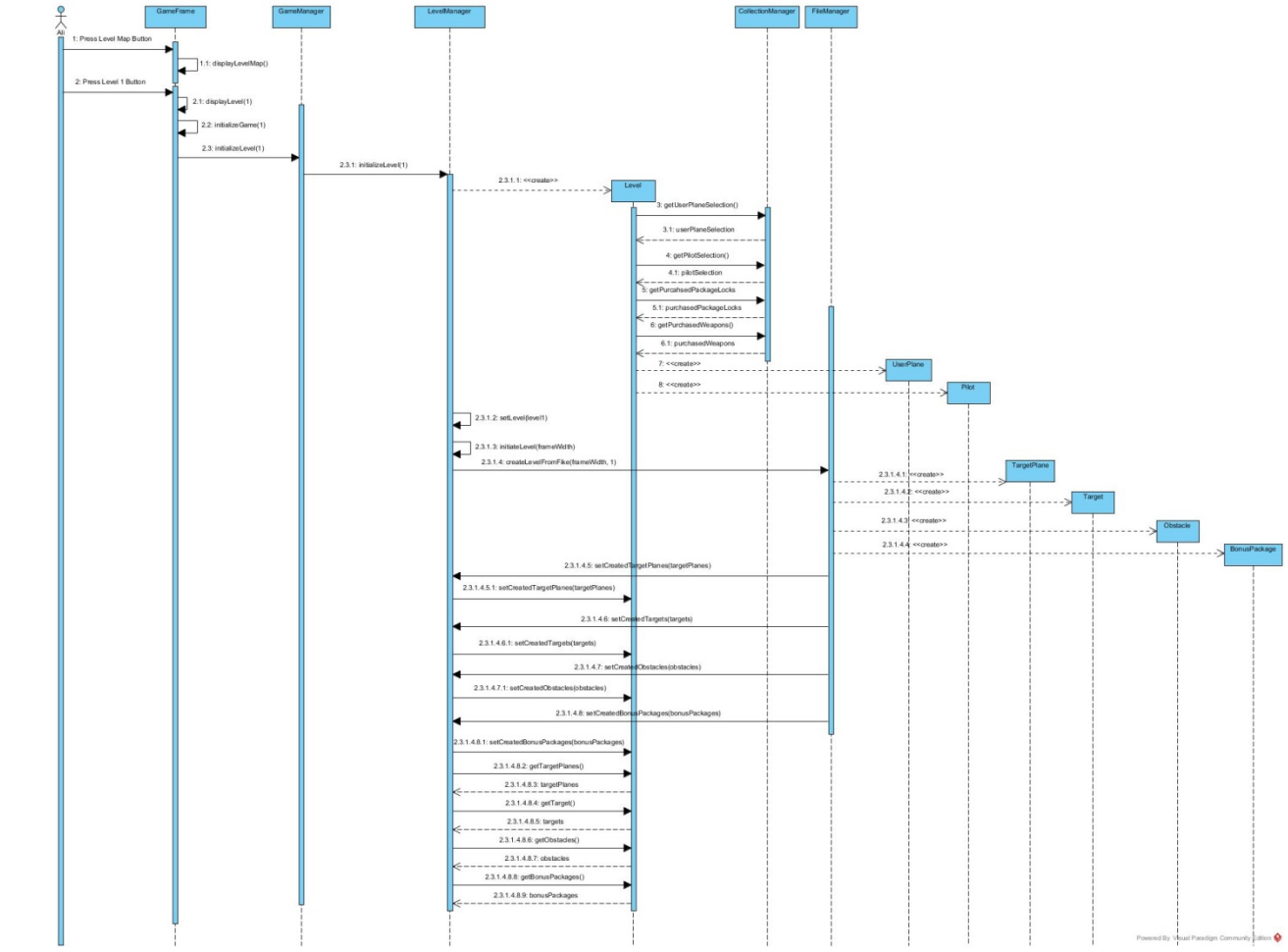

Figure 49 Start Game Sequence Diagram

3.2.2.2. Creation of GameObjects During Level

Scenario: (Continued after Start Game) The system starts the time allocated for the selected level, Level 1. GameFrame paints the component and requests GameManager to createObjects by passing the time. GameManager conducts the call to LevelManager and LevelManager conducts to Level. Level checks the list of created targets and request the appearance time of targets. If the appearance time of target matches the current time, then the Level moves the object from created items list to actual list. Then Level updates the targets in LevelManager. After the creation of new object, the GameFrame requests to moveObjects and then Level draws all objects on to the screen. This process of creating and loading GameObjects continue until the game is over, time can be over or Player can die.

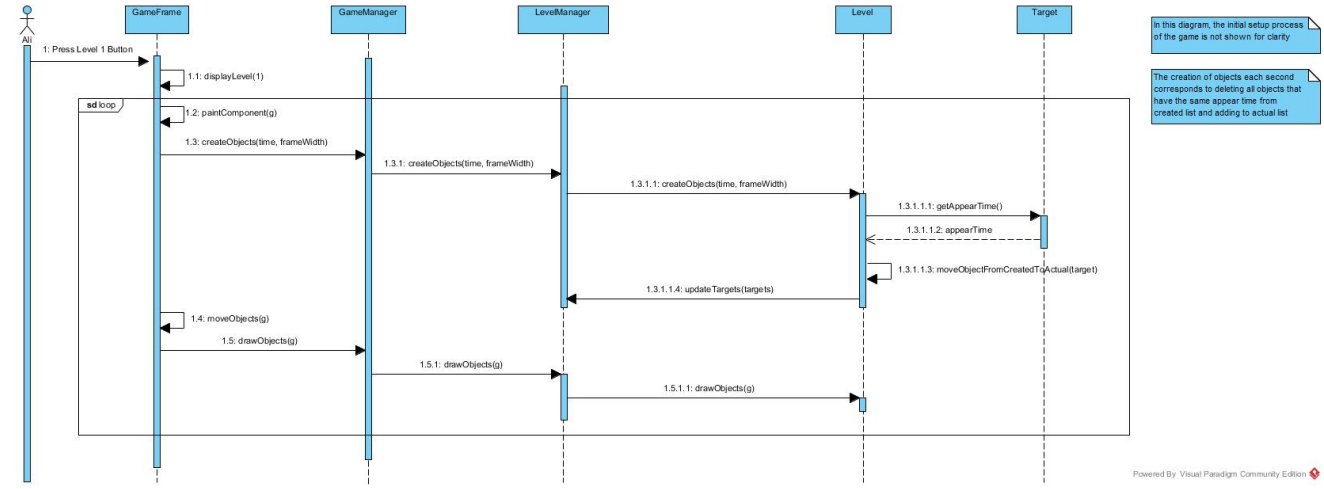

Figure 50 Create Game Objects During Level Sequence Diagram

3.2.2.3. Level Play

Scenario: (In this scenario the loop of Creation of GameObjects During Level is not represented for clarity) Player Ali requests to start Level 1 by pressing on the corresponding button. The system initializes the game and starts the time. Player Ali presses 'down' key to move the UserPlane down. The message is conducted to UserPlane via GameManager, LevelManager and then Level . UserPlane moves down. The system checks for collisions. LevelManager specifies that there is no collision. Player Ali presses 'up' key. The system responds by moving the plane up. Again, system checks for collisions and this time LevelManager detects a collision between UserPlane and Weapon. LevelManager requests CollisionManager to handle the collision and CollisionManager handles it by setting the health of UserPlane. Then the system checks whether the health of the UserPlane is depleted. Since this is not the situation the game continues. Player Ali presses space button. Shoot message is conducted to LevelManager. LevelManager first checks whether User has the weapon. Since user doesn't have the weapon, shoot operation fails. Then Player Ali presses 'C' key to change the weapon type. The system updates the selected weapon by conducting message to LevelManager. Then Player Ali presses space key. The system again checks if the User has the weapon. User has the weapon this time and LevelManager creates a Weapon object. Then the system checks whether collision has occurred. LevelManager indicates that collision has occurred and it is between newly created weapon and Carriage. LevelManager calls CollisionManager to handle the collision. CollisionManager handles the collision by updating health of Carriage. Then the system checks whether the life of Carriage has depleted. Carriage indicates that its health has depleted. Then the CollisionManager removes Carriage and Weapon from screen. Finally, it updates user points.

This game loop continues until the system detects that the time is over. When the time is over the system requests the total points earned and requests Level to turn points into coins. GameFrame also requests the game over text and whether the game is passed or not. Since the player has collected enough points, Level returns true through manager classes and GameFrame requests GameManager to add level 1 to list of passed levels. Finally, Player Ali is directed back to the Level Map Page.

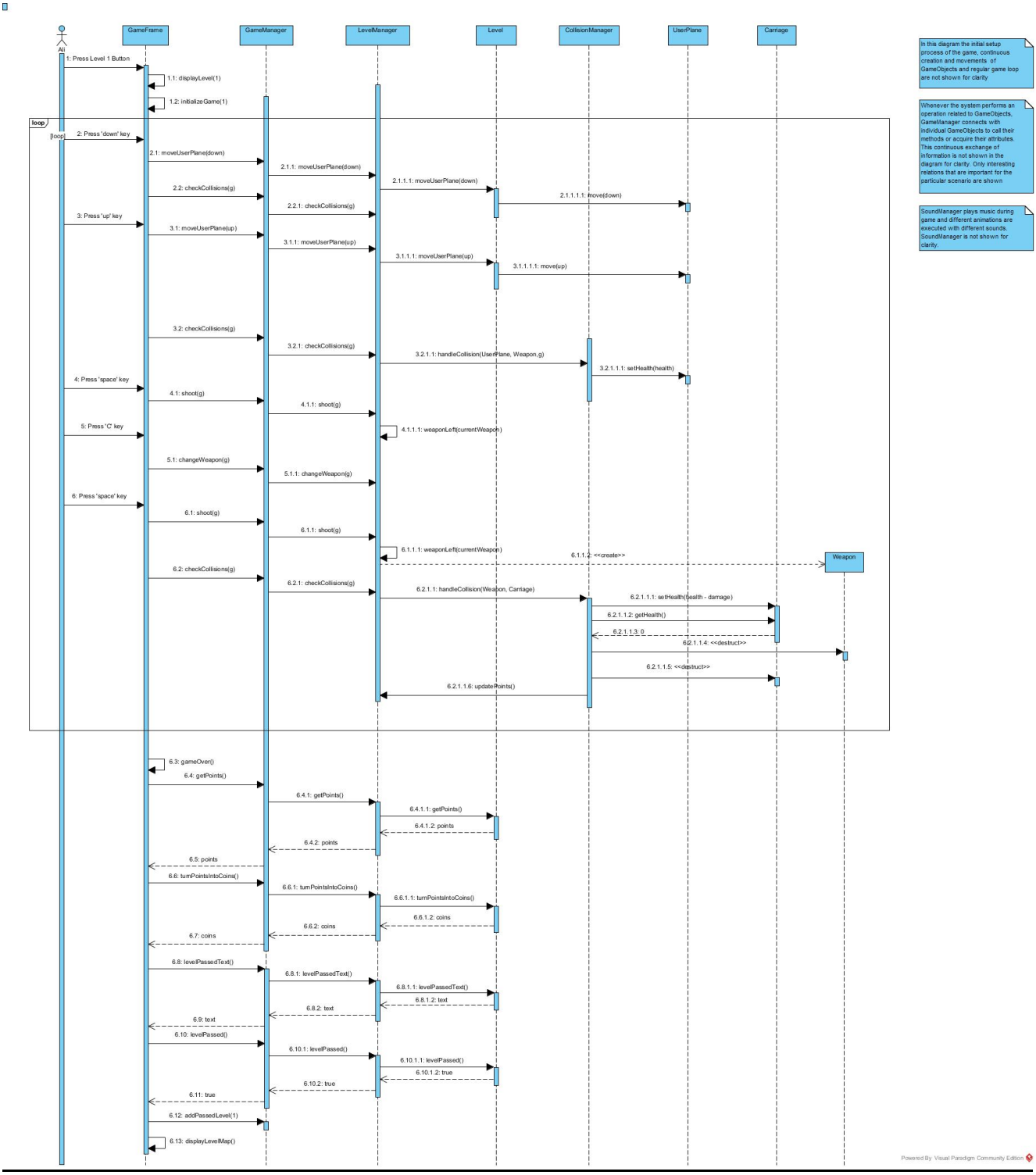

Figure 51 Level Play Sequence Diagram

3.2.2.4. Collecting Present Bonus Package

Scenario: (In this scenario the loop of Creation of GameObjects During Level and the actual game loop details are not shown for clarity) While playing Level 2, Player Ali presses 'up' key. The system requests UserPlane to move by conducting the message through GameManager, LevelManager and Level and the UserPlane moves down. The system checks for collisions. LevelManager specifies that collision occurred and indicates that the collision has occurred between UserPlane and HeathBonusPackage. LevelManager calls CollisionManager to handle the collision. CollisionManager

gets the health amount of HealthBonusPackage and updates the UserPlane health accordingly. Then the game loop continues as usual.

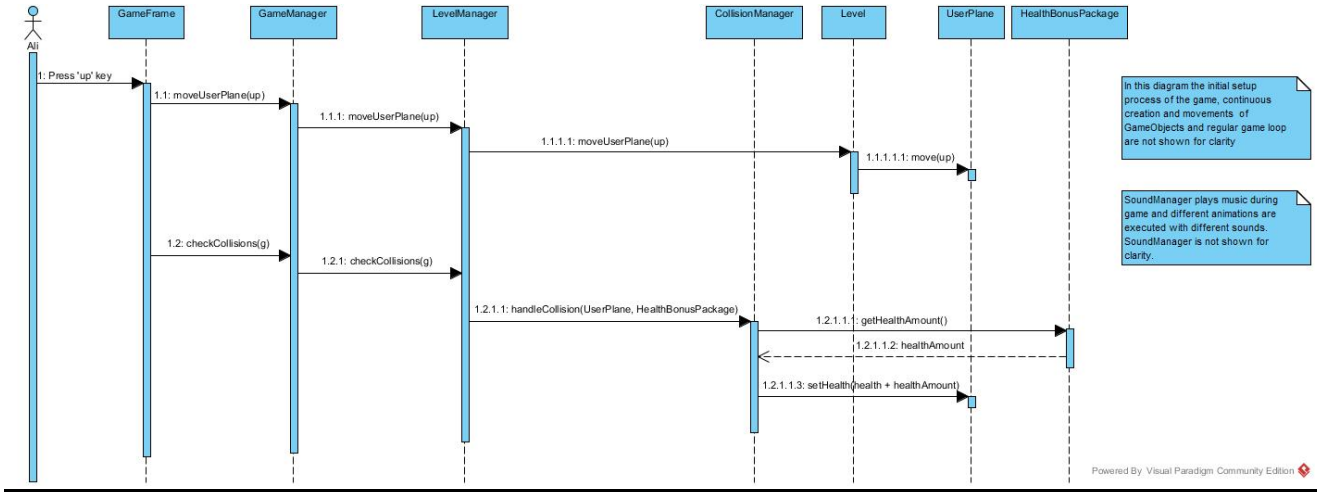

Figure 52 Collecting Present Bonus Package Sequence Diagram

3.2.2.5. Collecting Trap Package

Scenario: (In this scenario the loop of Creation of GameObjects During Level and the actual game loop details are not shown for clarity) While playing Level 2, Player Ali presses 'up' key. LevelManager requests UserPlane to move and the UserPlane moves up. The system checks for collisions. LevelManager specifies that collision has occurred and indicates that the collision has occurred between UserPlane and PlaneEnlargeTrapPackage. LevelManager calls CollisionManager to handle the collision. CollionManager tells user UserPlane to enlarge and UserPlane is drawn on screen with enlarged size. Then GameFrame checks the bonus time. Since the effect time of PlaneEnlargePackage has ended, the GameFrame requests to draw the user plane again. This time the UserPlane is drawn on screen with its default size. Afterwards, the game loop continues as usual.

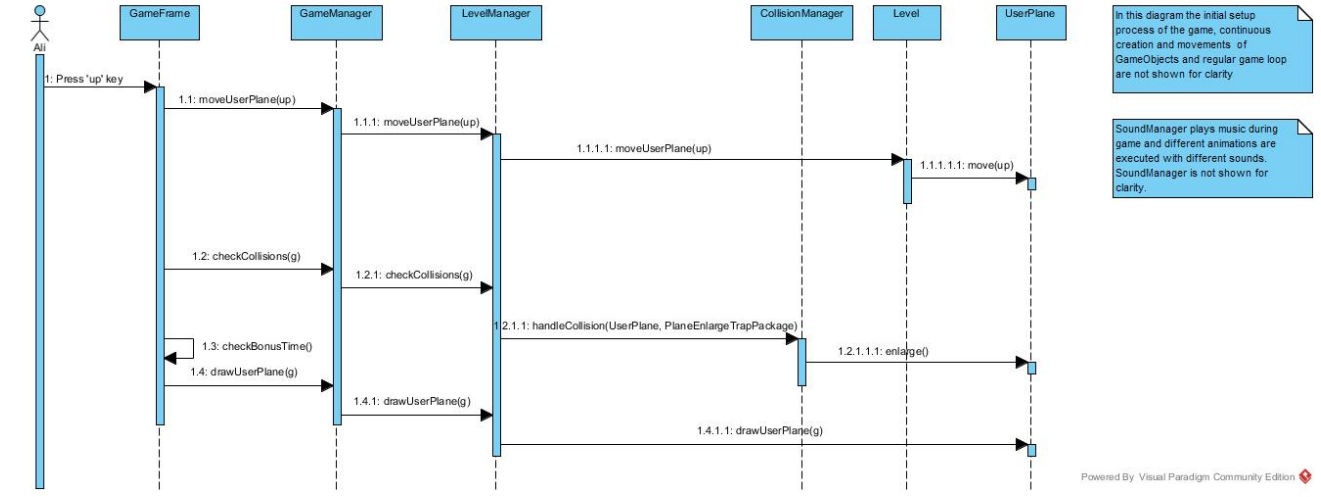

Figure 53 Collecting Trap Bonus Package Sequence Diagram

3.2.2.6. Obstacle Hit

Scenario: (In this scenario the loop of Creation of GameObjects During Level and the actual game loop details are not shown for clarity) While playing Level 2, Player Ali presses 'up' key. The system

requests UserPlane to move and the UserPlane moves up. The system checks for collisions. LevelManager specifies that collision occurred and indicates that the collision has occurred between UserPlane and Obstacle. LevelManager requests CollisionManager to handle the collision. CollisionManager sets the UserPlane health to 0 and removes UserPlane from screen. It also conducts the game over message to GameFrame. GameFrame ends the game and directs Ali to Level Map.

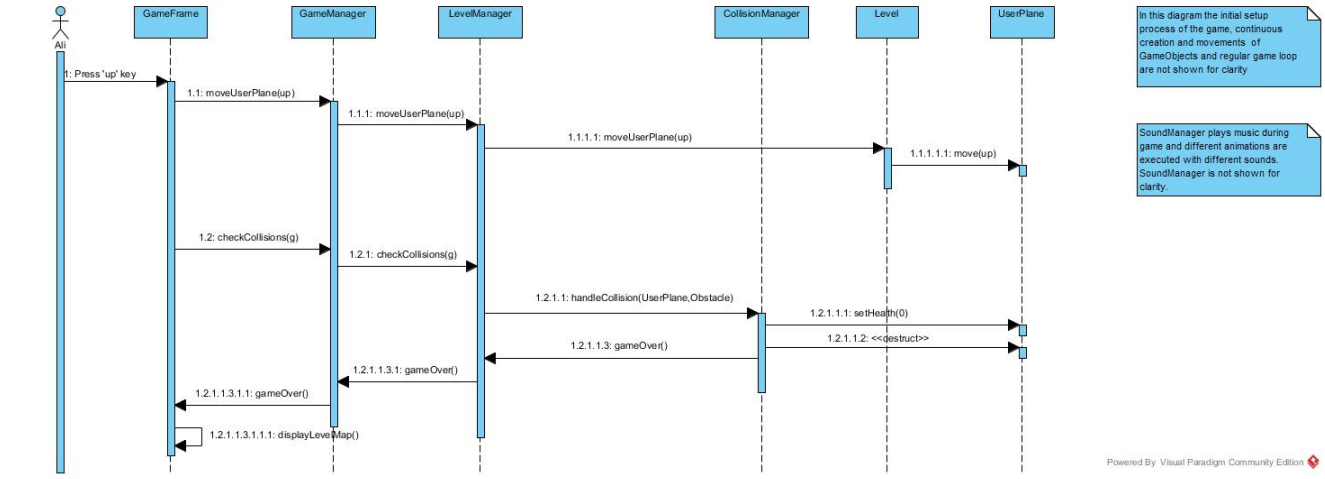

Figure 54 Obstacle Hit Sequence Diagram

3.2.2.7. Using an Explosive as Weapon

Scenario: (In this scenario the loop of Creation of GameObjects During Level is not represented for clarity) While playing level 2, Player Ali presses 'C' key iteratively until he reaches to Bomb which is an Explosive Weapon. Player Ali presses space button. The system first checks whether the User has the weapon. The user has the weapon hence, LevelManager creates an Explosive. And moves the weapon together with all game objects. Then the system checks whether collision has occurred. LevelManager indicates that collision has occurred and it is between newly created Explosive and TargetPlane. Then the CollisionManager is called to handle the collision and it updates the health of game objects in damage area. It also updates the health of UserPlane and removes Explosive from screen. The points of user is also updated. The game continues afterwards.

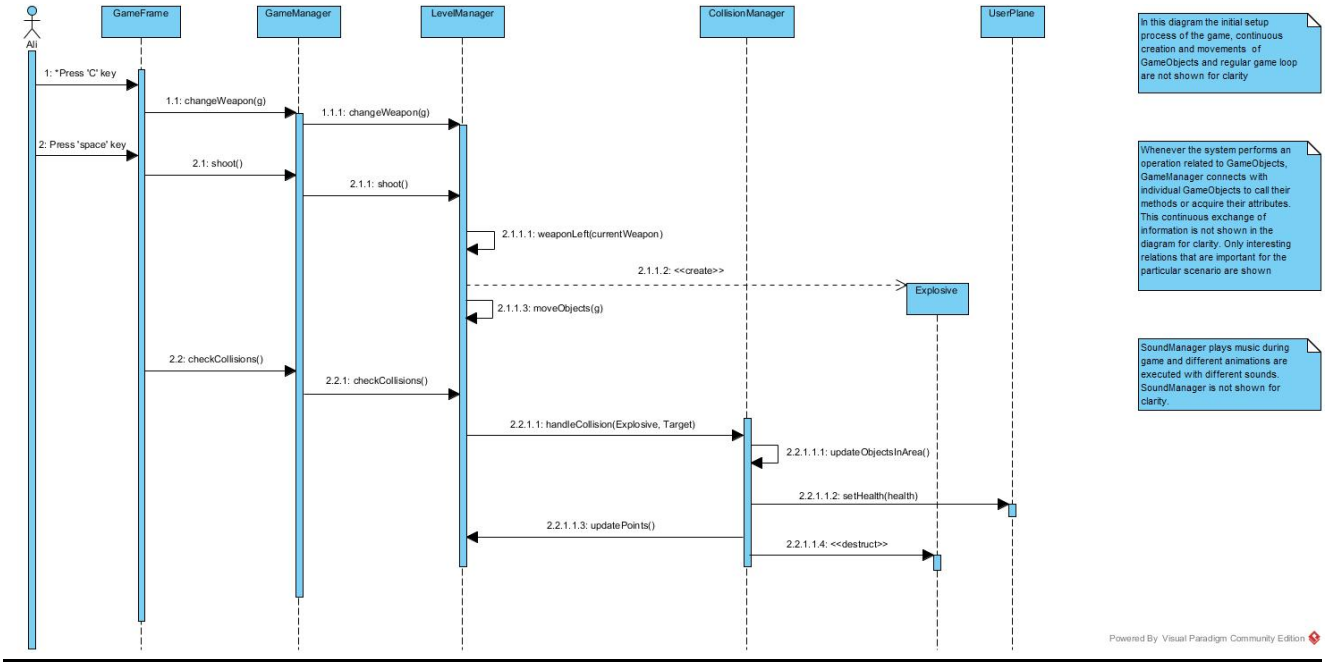

Figure 55 Using an Explosive as Weapon Sequence Diagram

3.2.2.8. Shooting A Rocket

Scenario: (In this scenario the loop of Creation of GameObjects During Level is not represented for clarity) While playing level 2, Player Ali presses space button. The system first checks whether the User has the weapon. User has the weapon therefore LevelManager creates a Weapon object and moves it with other game objects. Then the system checks whether collision has occurred. LevelManager indicates that collision has occurred and it is between newly created Weapon and Rocket. The system first updates the health of Rocket and checks whether the health of the Rocket has depleted. Rocket indicates that this is the situation. The CollisionManager updates health of game objects in area, including UserPlane and removes Weapon and Rocket from screen. Finally, CollisionManager updates user points and the game continues.

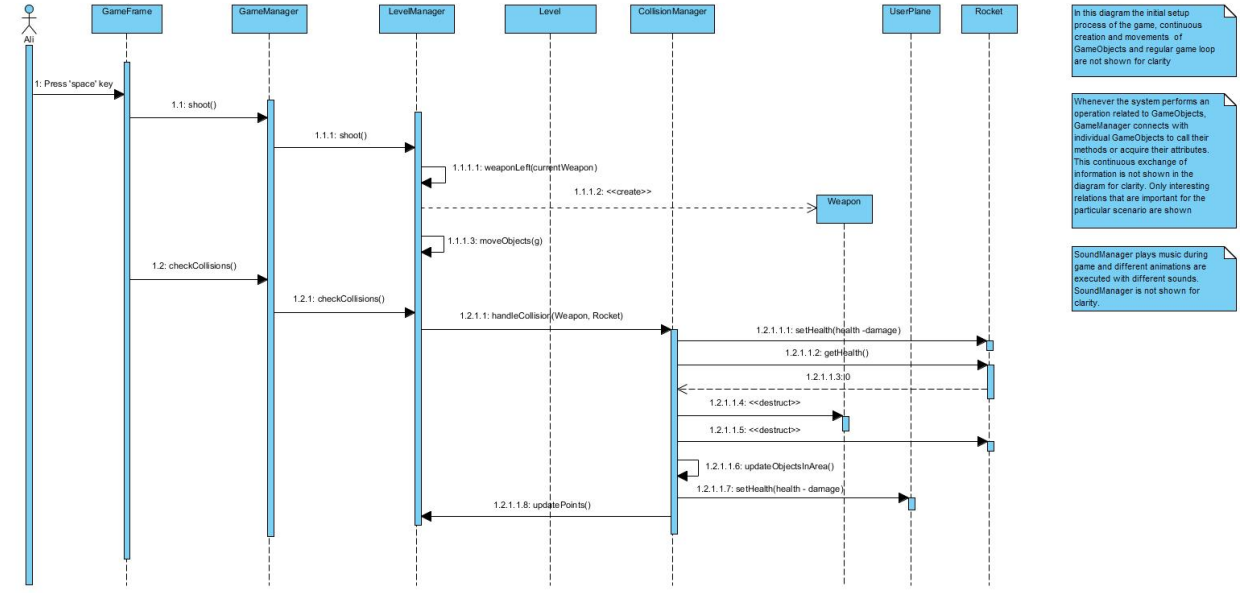

Figure 56 Shooting a Rocket Sequence Diagram

3.2.2.9. Playing Bonus Mission

Scenario: (It is assumed that the Player has shot the BonusMissionTarget in the level and Bonus Mission option is visible in the Level Map.) Player Ali requests to view Level Map by pressing relative button in the GameFrame. GameFrame displays Level Map screen. Ali chooses the Bonus Mission, Ship Shooting in this scenario by pressing the proper icon.

System requests GameManager to initialize the game. GameManager conducts the message to LevelManager. LevelManager creates the corresponding Level. Level requests all purchased items from CollectionManager. Level creates UserPlane and Pilot. Then the created Level is loaded to LevelManager. LevelManager asks FileManager to createLevelFromFile. FileManager creates all objects of the level, a Ship in this scenario. Then FileManager sends these created objects to LevelManager and LevelManager sends them to Level. Consequently, all objects are created and added to Level and LevelManager. Finally, LevelManager requests targets from Level in order to initialize its own list.

Then the system starts the time allocated for the Bonus Mission. GameFrame requests to createObjects according to time. The system detects that the appear time of ship matches the current time. Therefore, the ship is moved from created targets to actual targets and targets of LevelManager is also updated. This process of creating and loading GameObjects continue until the Bonus Mission is over.

Player Ali presses ‘left’ key to move the UserPlane left. System requests UserPlane to move and the UserPlane moves left. LevelManager checks for collisions and does not detect a collision.

Then Player Ali presses space button. LevelManager creates a Weapon object, Torpedo specifically. Then the system checks whether collision has occurred. LevelManager indicates that collision has occurred and it is between newly created Torpedo and Ship. CollisionManager handles the collision by updating health of Ship and removing Torpedo from the screen. Then this game loop continues until the system detects that the time is over. When the time is over GameFrame requests total points collected and requests to turn points into coins. Then the Bonus Mission is completed and Player Ali is directed back to the Level Map Page. Bonus Mission button is removed from Level Map until Player Ali shoots another BonusMission Target.

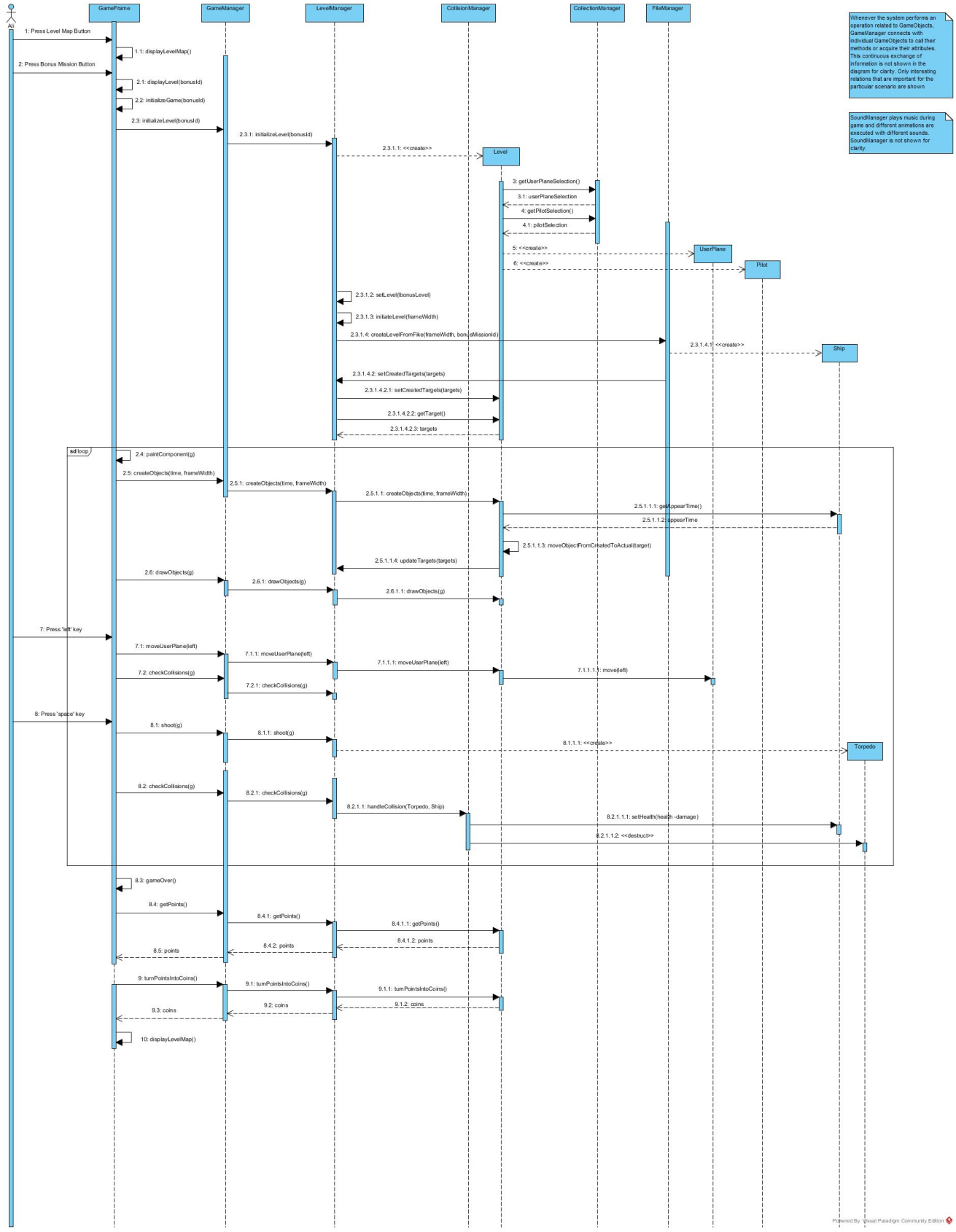

Figure 57 Playing Bonus Mission Sequence Diagram

3.2.2.10. Purchasing UserPlane from Store

Scenario: Player Ali requests to view Store by pressing relative button in GameFrame. GameFrame updates store view by requesting user coins from CollectionManager and disabling items user cannot afford. Then GameFrame displays the Store screen. Player Ali selects a Weapon by pressing the item view, Flame Gun in this scenario. GameFrame then tells GameManager to purchase the item and GameManager tells to CollectionManager. CollectionManager purchases the item and updates the amount of coins. Then Player Ali presses Main Menu button and the System displays the main menu.

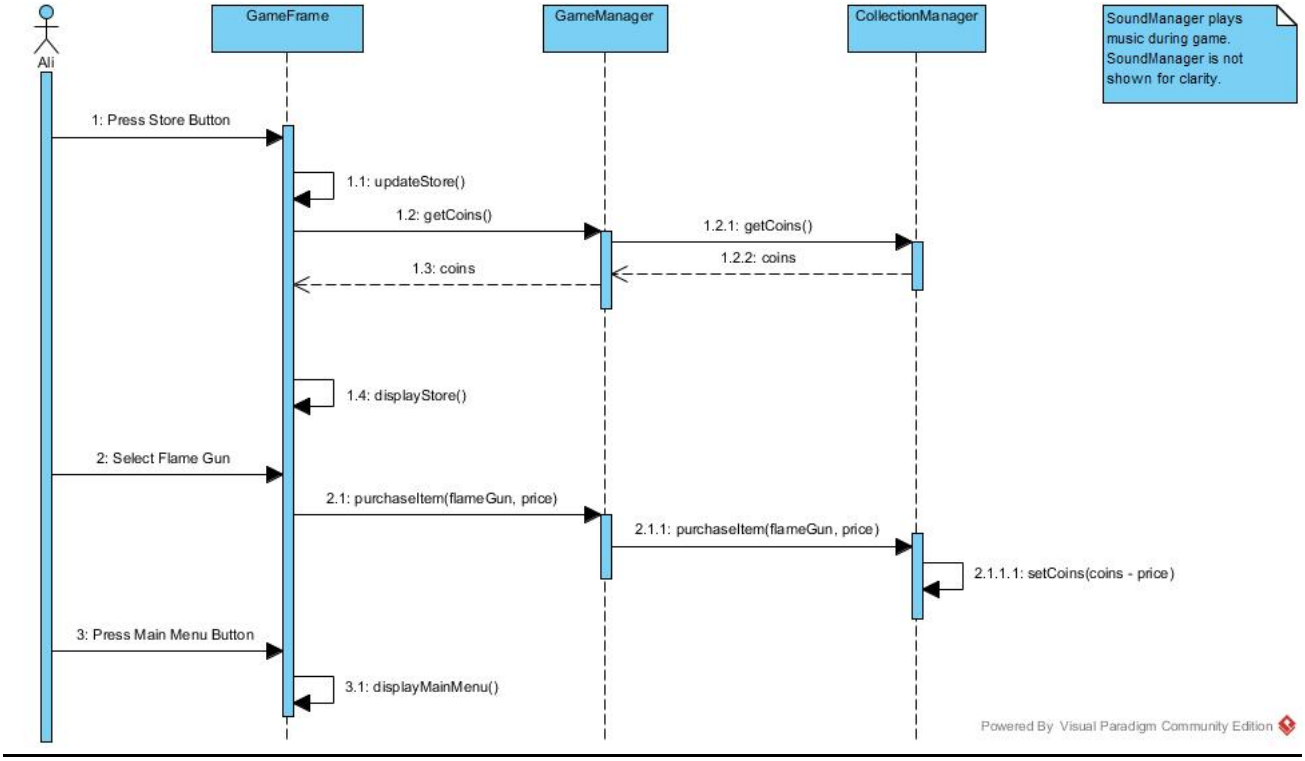

Figure 58 Purchasing a UserPlane from Store Sequence Diagram

3.2.2.11. Changing UserPlane Preference from Collection

Scenario: Player Ali requests to view Collection by pressing relative button in GameFrame. GameFrame loads the Collection View. GameFrame updates the collection by requests all purchased planes and pilots from CollectionManager. Then GameFrame displays Collection screen. Player Ali changes plane choice by selecting F-22 plane. GameFrame requests from CollectionManager via GameManager to update the UserPlane selection. Then Player Ali presses Main Menu button hence the system directs him to Main Menu.

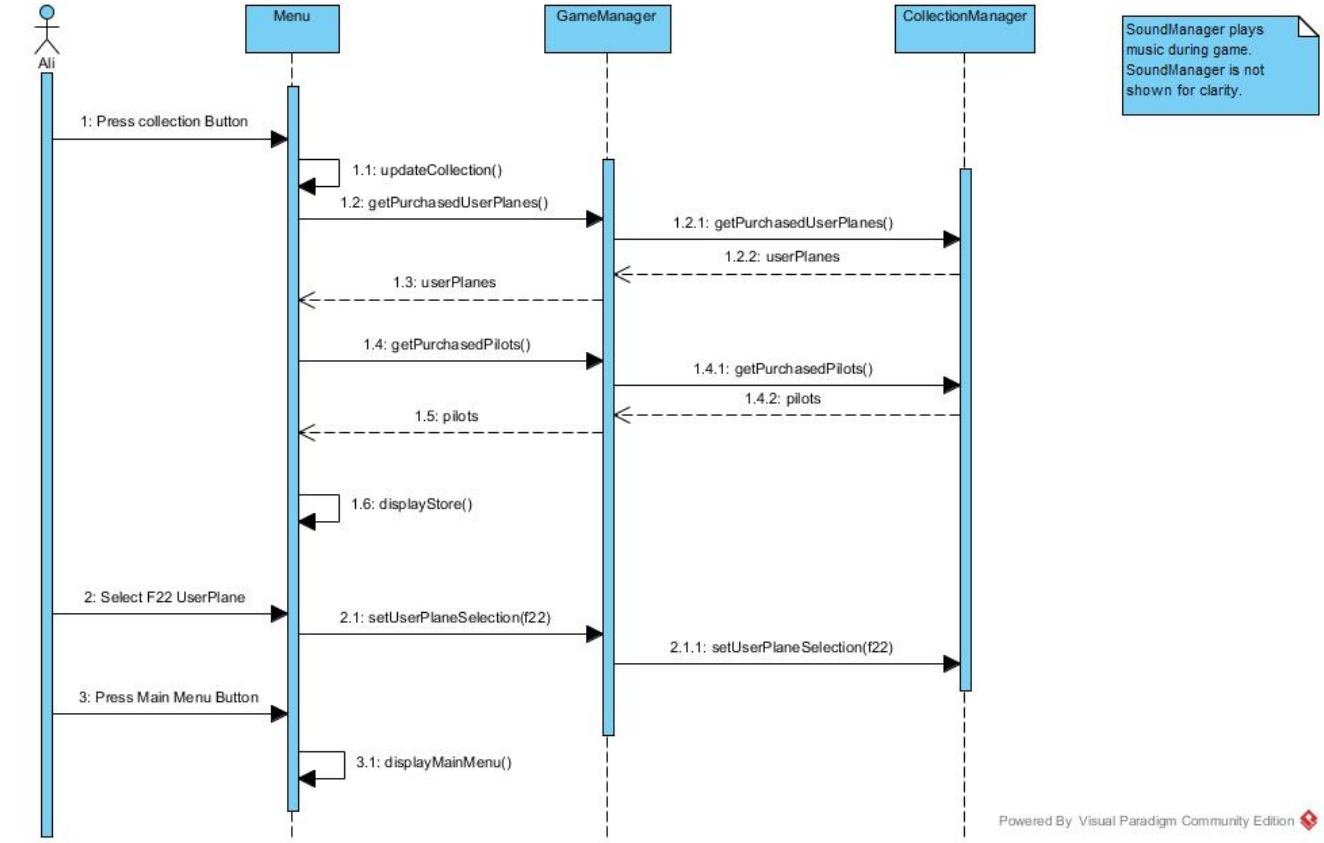

Figure 59 Changing UserPlane Preference from Collection Sequence Diagram

3.2.2.12. Changing Settings

Scenario: Player Ali requests to view Settings by pressing relative button in GameFrame. GameFrame loads the Settings View. Player Ali turns the music off. The System updates the settings by telling GameManager to turn music off. Player Ali presses Main Menu button and System directs him to Main Menu.

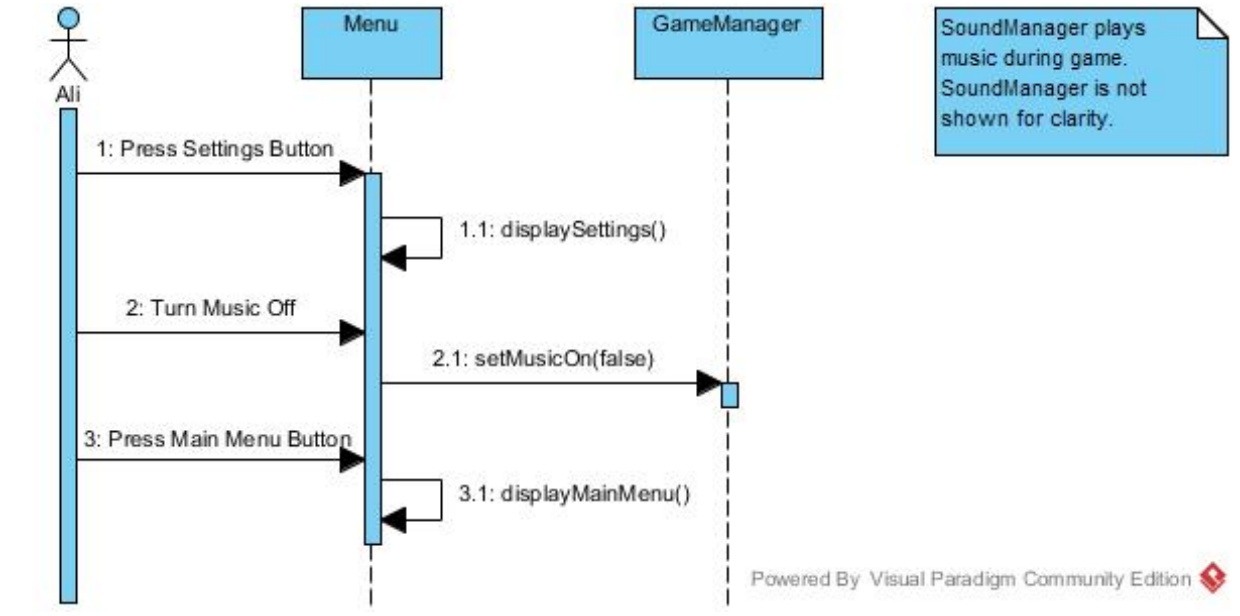

Figure 60 Changing Settings Sequence Diagram

4. Design
4.1.Design Goals

Reliability

Reliability is fundamentally the ability of the system to perform required operations. We aim our system to be reliable, that is the system should be able to perform regular game operations until the user stops playing. Hence, the system should be resistant to external and internal failures and prevent crashes. In order to achieve this goal, boundary conditions should be determined carefully. During implementation, necessary precautions should be taken to prevent system from crashing. Moreover, if a power-loss or any other problem occurs, the system should not result in data loss. The levels the player has completed, his selections and shopping should be restored if the system fails.

Traceability of Requirements

Traceability is important for design of "Sky Wars" in terms of refining requirements into lower-level design components, in understanding the consequences of a possible change and in justification of the existing requirements. In order to achieve this goal, linking each system requirement to real-world dimensions is necessary.

Usability

User-Friendliness

In our software, the user friendliness will be achieved by considering efficiency, effectiveness and satisfaction. The program will be developed in a user-centered way and it will be tested to see the user experience so that it can be evaluated and improved. A pleasant and an explanatory graphical user interface will help us to achieve this design goal.

Ease of Use

The game will be easy to use because it will be easy to learn. The interface of the game will be easy to navigate and there will be an efficient error handling. Also the menu and the instructions at the beginning of the game will help the user to play game without trouble.

Understandability

Sky Wars, its design model, architecture, implementation and documentation should be understandable. We aim to design the system in a way that users, developers and reviewers can understand the system boundaries, functions and structure. In order to achieve this goal, we need to focus on consistency and completeness during development process.

Performance

Rapid-Development

Since the software will be developed by aiming to make it an object-oriented program, the components will be reused. The models and the UML diagrams will greatly reduce the implementation process. On the other hand, our programming language will be Java. It provides us visual aid tools which also maintain a rapid-development.

High Performance

In non-functional requirements of the system, it is declared that the system should respond to the Player input immediately. This is crucial for "Sky Wars" in order to keep player entertained and interested in the game. Besides, the system must also move the game objects smoothly and handle animations without any pause. In order to reach a good game performance and smooth animations, images can be loaded to memory before the game starts.

Efficiency

In order to make Sky Wars as efficient as possible, performance goals must be reconsidered in terms of complexity. Thus, possible bottlenecks of the system design can be found and reorganized before the implementation phase. Writing maintainable and readable code would also be helpful through the implementation in terms of efficiency. So that, the code segments would be eligible for reconsideration.

Supportability

Flexibility

A flexible software should be able to adapt itself for future changes. In order to achieve that, we will keep the coupling of the components at minimum so that a change will not the affect the whole system. We will limit the locations of the necessary modifications. Also, the boundaries between the objects will be kept simple.

Modifiability

We aim Sky Wars to be Modifiable, changes should be easy to implement. When the functionality of the system needs to be changed, this change should not alter the whole structure. In order to achieve this goal, the coupling between subsystems should be minimized. Components of Sky Wars should be independent enough that a change in a component does not alter other components much.

Maintainability

One of the design goals is Maintainability, the ability to change the system to fix bugs and introduce new technologies. This goal can be achieved by minimizing the complexity of the system. Hence the system should be decomposed to subsystems in a way that complexity is reduced.

Adaptability

Since an adaptable design allows developers to easily customize the system, through adapting the existing design, and also gives the privilege of quickly developing new and upgraded models, it is important for the system. Since Java is one of the few programming languages, which provides cross-platform portability, it will be helpful through the development process for the system adaptability. Operating system constraints will not be a problem. Another key point for the adaptability of Sky Wars would be writing the code as readable and maintainable as possible.

Trade-offs

Functionality vs. Ease of Use

In Sky Wars, to allow user to easily adapt to game, we use only arrow keys, space button and C button to play the game. We thought that having too many buttons can create distraction while user is playing the game. Thus ease of use is more important than functionality in our game.

Rapid Development vs. Functionality

Since we have limited time, we have to analyze and design our game faster than a normal procedure. Because of that we cannot add many things. Thus we reduce functionality. However, we will develop Sky Wars with a sufficient functionality which user can demand.

High Performance vs. Reliability

Sky Wars is an arcade game because of that user has to see all moving objects without any skipping. Thus we believe that performance of Sky Wars is really important. We also want our game to be reliable, consistent even in failure. We also aim to prevent data-loss in a system crash. Hence, we aim higher performance as long as the system is reliable.

High Performance vs. Memory

In order to maximize performance memory is effectively used in Sky Wars. For instance, game images and sounds are loaded to memory before the game is started. However, when memory usage is

tried to restricted, the performance decreases.

High Performance vs. Adaptability

Sky Wars could be written in another programming language such as C/C++ for performance advantages. However, easily customizing the system through adapting the existing design is one of the important design goals of Sky Wars. Thus, a trade-off between performance and adaptability must be done.

4.2.Subsystem Decomposition

Sky Wars system is decomposed to three parts: User Interface, Game Control and Game Model. These layers are ordered hierarchically and they have run-time dependency. User Interface subsystem contains Boundary classes and View components. This subsystem is responsible from interacting with user and handling view operations. It is the top layer and calls Game Control subsystem to handle game-related operations. Game Control subsystem is the middle layer and it contains classes responsible from the fundamental functions and operations of the game. The Game Control subsystem provides services related to all operations of the game including level operations. This subsystem calls Game Model subsystem for retrieving necessary data for game operations. Game Model is the lowest layer. It provides data-related services such as data retrievals and updates. It consists of all classes containing game data.

Closed Architecture

Sky Wars architecture is a closed architecture in which layers can access to services of only layer below them. Hence, User Interface subsystem can only call services from Game Control subsystem and not from Game Model subsystem. Even though this architecture decreases Efficiency, it gains Flexibility to system since layers become more independent of each other. Moreover, a closed architecture is also more Maintainable since the run-time dependencies decrease.

Coupling and Coherence of Subsystems

In our subsystem decomposition, we aimed and achieved high coherency. Classes within subsystems share a common concept and they perform similar and related operations. They interact with each other frequently for implementing related functionalities. For instance, Game Control subsystem consists of Controller classes such as GameManager, LevelManager, and SoundManager. They are all responsible from handling general game functions related to store, collection, settings and level-specific operations. They interact with each other for all game operations. For instance, GameManager calls LevelManager for level control, LevelManager calls GameManager for obtaining related information. SoundManager is called by GameManager for sound operations. This strong relation and unity among subsystem classes indicate that we have high coherence within our subsystems.

We also aimed and achieved low coupling. Hence, subsystems mostly do not depend on each other. There are weak relations between subsystems therefore, they can act more independently. For instance, only connection between User Interface subsystem and Game Control subsystem is between GameManager and Menu and ScreenManager. Game Control subsystem is connected to Game Model only via relation between LevelManager and Level classes. Hence classes within subsystems mostly do not know about interfaces of other layers. Hence changes in a subsystem generally do not affect other subsystems. Our subsystem architecture demonstrates that we have low coupling.

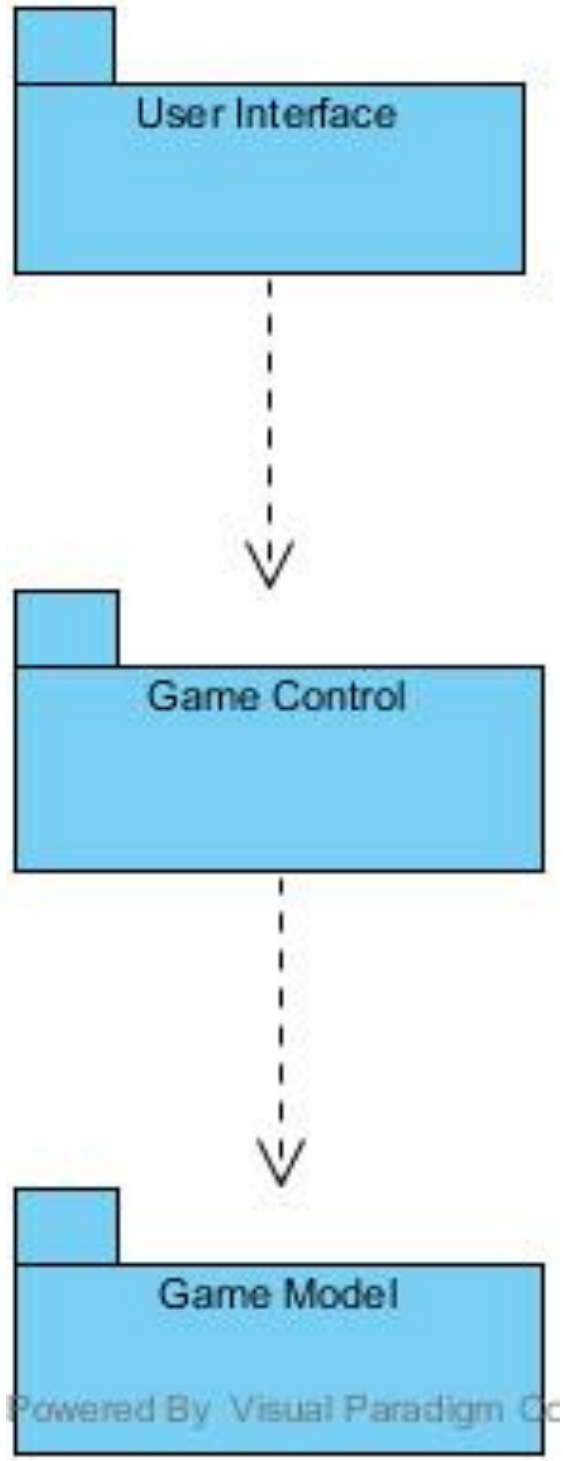

Figure 61 Basic Layers of Sky Wars

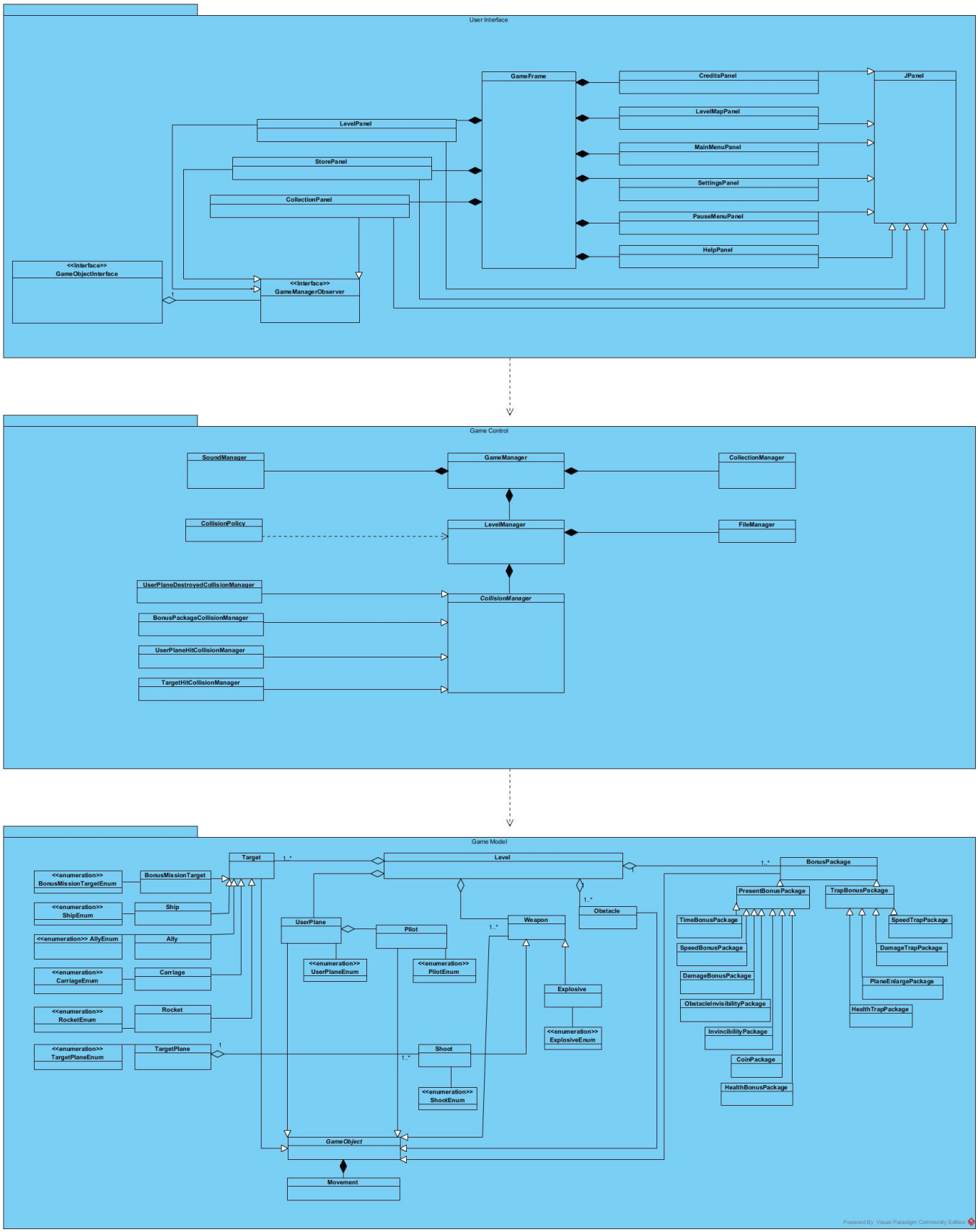

Figure 62 Sky Wars Subsystem Decomposition with Subsystem Details

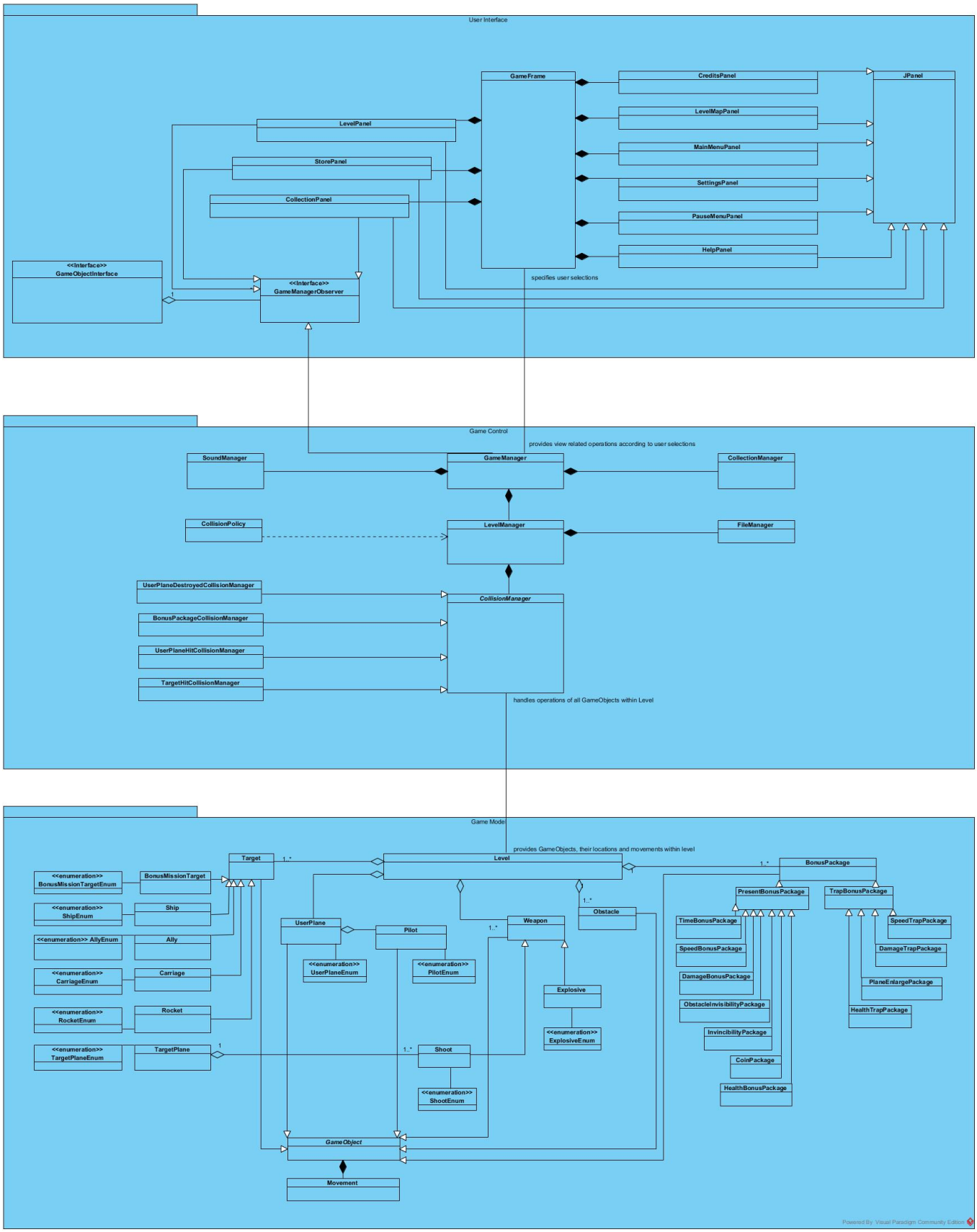

Figure 63 Subsystem Decomposition with Connection Details

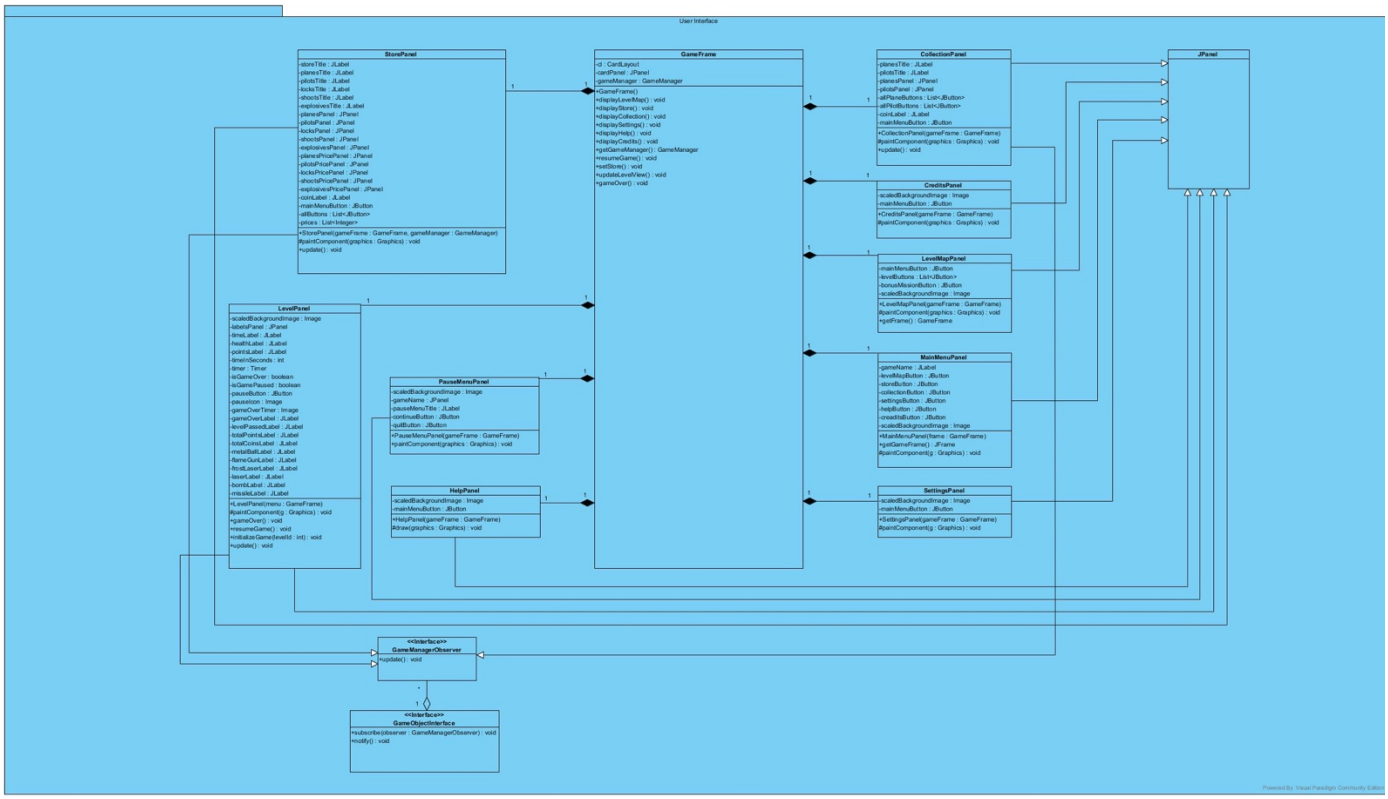

Figure 64 User Interface Subsystem Details

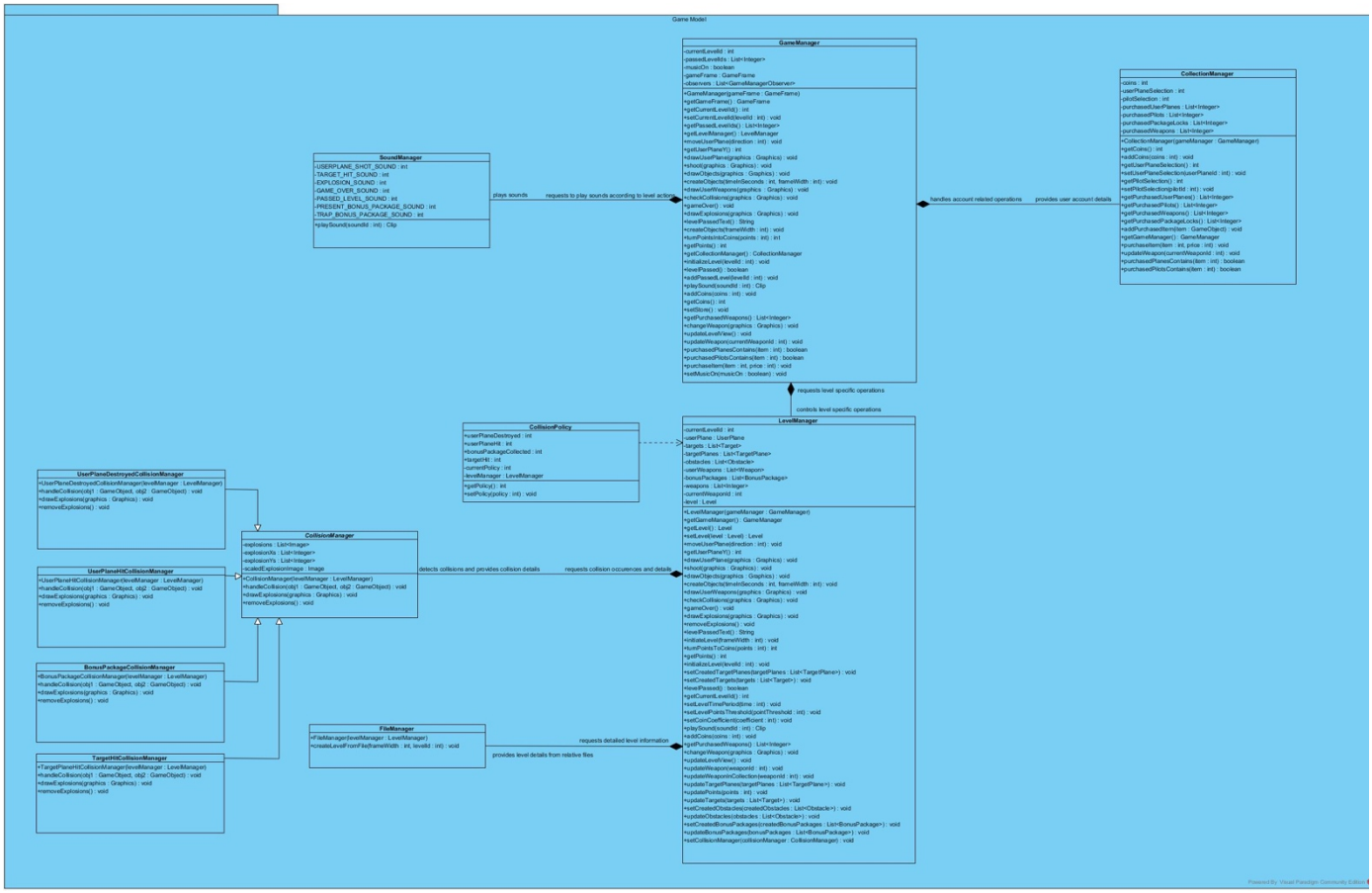

Figure 65 Game Control Subsystem Details

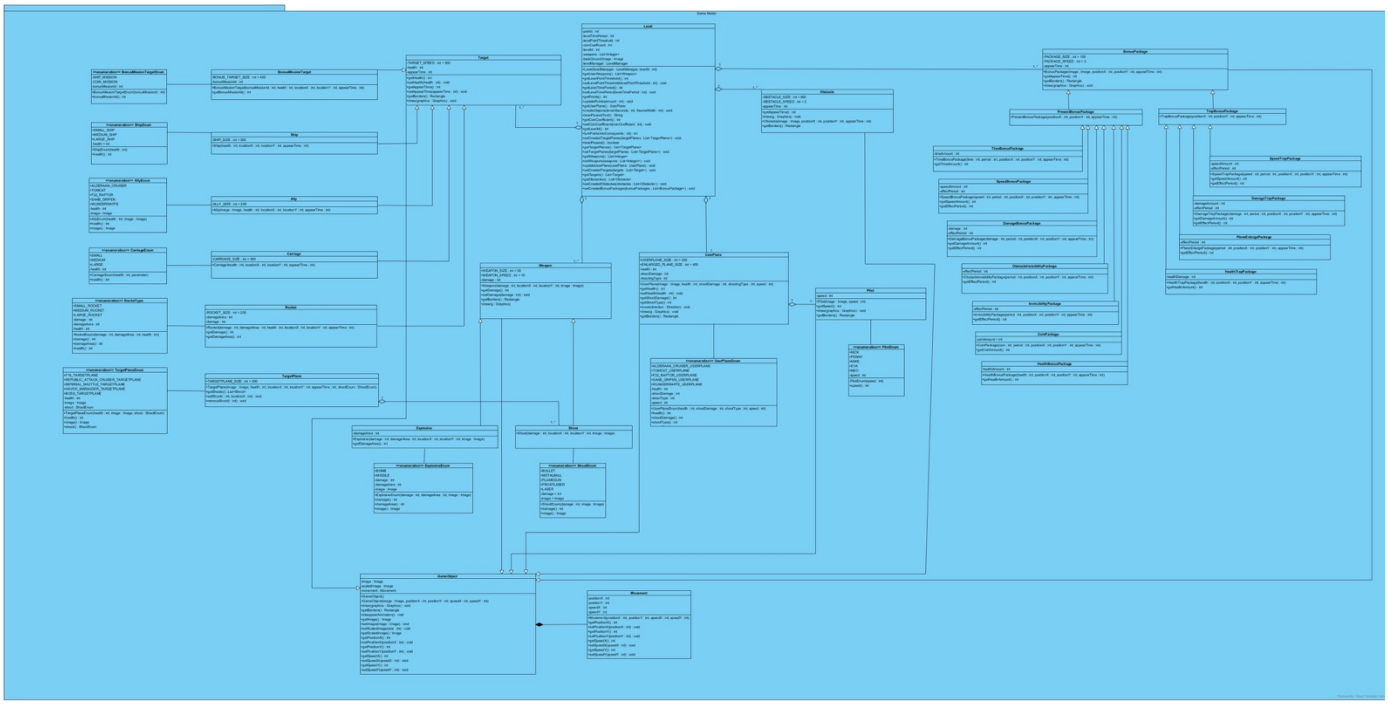

Figure 66 Game Model Subsystem Details

4.3.Architectural Patterns

3-Tier Architectural Style

We applied 3-Tier Architectural Style for Sky Wars system. Hence the game consists of three hierarchically ordered virtual machines. The first layer is the Presentation Layer. This layer is the layer that communicates with the user. In our application User Interface subsystem is the Presentation Layer. It is the top layer which represents application boundary. The Application layer is the second layer which is the logic part of the system. Game Control subsystem represents the Application level since it is the part which controls the game. The last layer in a 3-layer architecture is Data Layer which provides persistent data of the game. In Sky Wars, Architecture Game Model subsystem is the Data Layer since it holds the game data.
3-tier architecture is appropriate for Sky Wars since the game can be divided to client, logic and data parts. This architectural style makes the game more Manageable and Maintainable.

Model View Controller Architectural Style

Even though Sky Wars system is decomposed as layers and 3-Tier architecture is the fundamental pattern, we also have a structure which is similar to Model View Controller pattern. The User Interface subsystem contains all View objects of the system. Hence User Interface subsystem acts as the View subsystem which is responsible from displaying game entities to user. The Game Control subsystem corresponds to Controller subsystem naturally, since all Controller classes are in Game Control subsystem. Hence, it acts like Controller and allows communication between Model and View by handling user input, updating models and views. Model subsystem is the Game Model subsystem because it contains all entity objects. It holds the application domain data.

Since Sky Wars can be decomposed to subsystems in a way that entity, boundary and control objects are grouped together, MVC pattern can be followed in system design. Using MVC architecture increases Flexibility, hence it becomes possible to make changes in a subsystem without affecting others.

4.4.Hardware/ Software Mapping

Sky Wars does not have a complex operation system. During level, objects are created, moved, updated and removed. Hence, the computation rate is not demanding for a single processor. One processor will be necessary to maintain the game without flaws. Similarly, there is no need for additional memory space. The game needs keyboard and mouse as external I/O devices for the game control.

Sky Wars is a single user game. Hence, it does not require connection to a server or internet or any other device. Therefore, the only node in the system is UserComputer. The UserComputer should have java compiler installed. Keyboard and Mouse I/O devices are also represented as external nodes.

The components of Sky Wars are associated with subsystems; User Interface, Game Control and Game Model. Game Control component provides game operations interface and User Interface uses these services. Game Model component provides data operations and Game Control uses this interface. This architecture is represented with below diagrams.

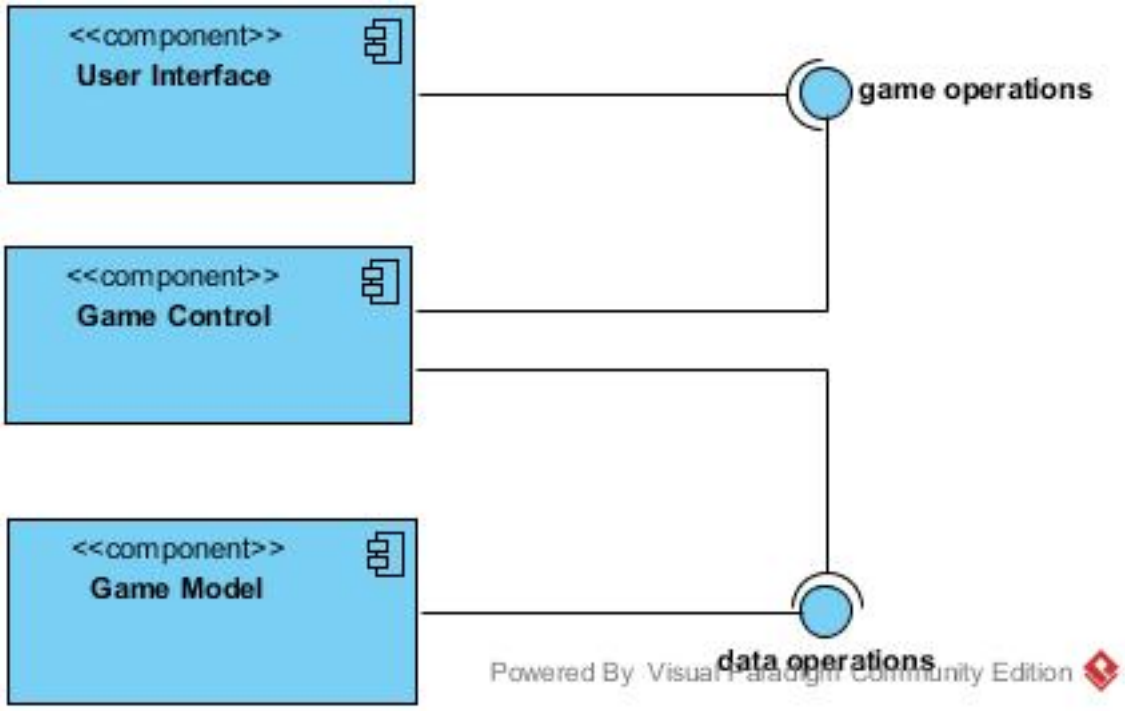

Figure 67 Component Diagram of Sky Wars

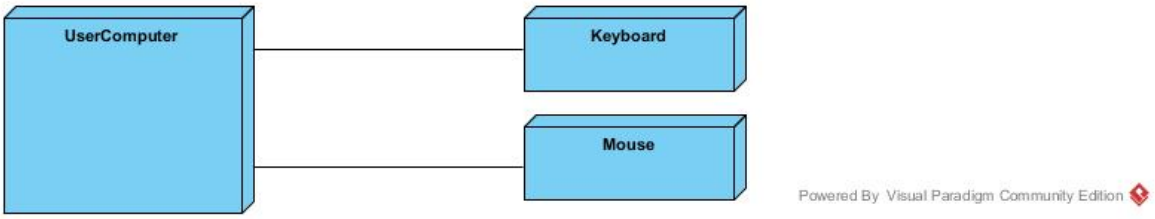

Figure 68 Deployment Diagram of Sky Wars

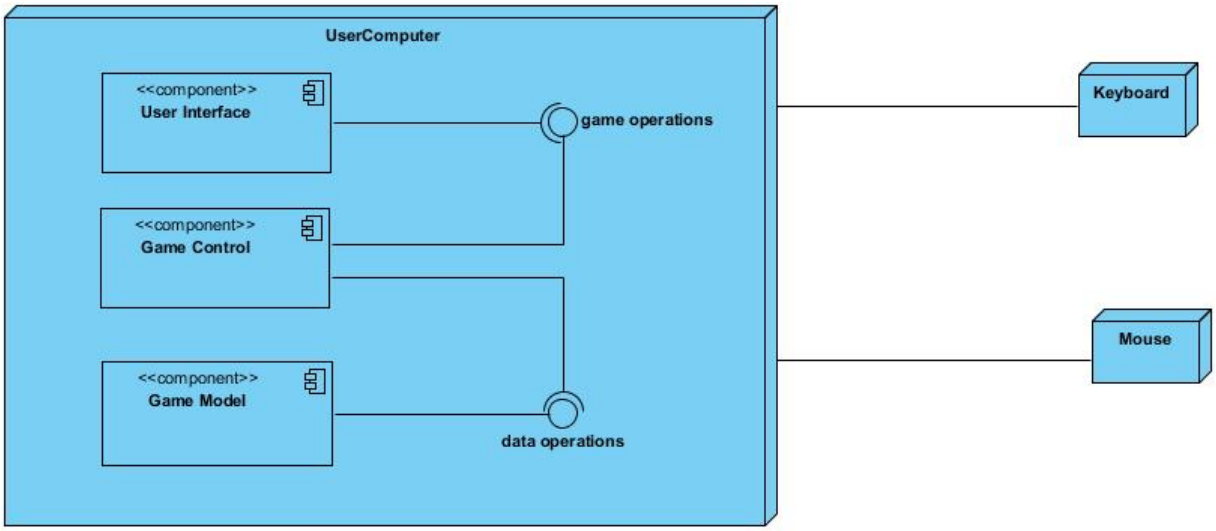

Figure 69 Detailed Deployment Diagram of Sky Wars

4.5. Addressing Key Concerns

4.5.1. Persistent Data Management

In Sky Wars, most of the objects are not persistent. Only entities that remain after the game is exited are current level, collection details, bonus level access and settings. The GameObjects and their status within level play will not be saved. Hence for saving, we decided to use file system where objects are saved as text. This data management system will increase efficiency and it will be easy to use since few objects will be recorded. Moreover, we have single writer of the file. Since we do not have multiple users accessing to data storage, it is appropriate to use file system instead of a database. Using a database will increase cost, make development and maintainability harder and it is unnecessary since we do not have a large set of persistent data. Moreover, image and sound files will be saved to drive.

4.5.2. Access Control and Security

As stated earlier, Sky Wars is a single player game. It does not need authentication so anyone with the executable file can start the application and play the game. Since there is only one player and there is no authentication, we did not create an access matrix. All use cases that is all operations defined externally by the system are open to access of any user. Since there is no access control, there cannot be a security issue about the system.

4.5.3. Global Software Control

In Sky Wars we have a fundamental controller class GameManager. Even though there are other controllers they all work under the control of Game Manager. Hence we adopted a Centralized Design for software control. We decided to have an event-driven control for Sky Wars. Since the game is managed according to external events, user control, the software control is based on these events. Therefore, the software system will have a simpler structure and centralized design will be handled within main loop of the game.

4.5.4. Boundary Conditions

Initialization

Sky Wars is provided as an executable .jar file. Hence, it does not require installation. It can be ran directly on any computer which has a Java compiler.

Termination

Player will be able to quit the game at any point of the game. Close button of the JFrame can be used for system termination. However, if the game is played in full screen mode, Quit buttons on game on Main Menu and Pause Menu screens can be used to exit game. During level, the player should pause the level first and click on Quit from Pause Menu. When the user quits the game, the levels he has passed, the bonus mission level access, the items he has purchased so far and his preferences in Settings and Collection will be saved to file document used for persistent data management. If the user quits in the middle of a level, his progress in level will not be saved and he will be required to start the level over.

Failure

Sky Wars contains plenty of image and sound files within. Hence, if initialization of the process is affected from this files and a problem occurs, the game will be started without sounds.

If a system error occurs during level play, the level will be restarted. The progress of the user in level will be lost if such an error occurs, however the game will not need restart.

If a certain screen does not respond to user, the panel will be closed by the system and main screen will be displayed.

If the system crushes, it will be restarted. The game will be automatically saved in certain time intervals to prevent data-loss in a system crash.

5. Object Design
5.1. Pattern Applications
5.1.1. Facade Design Pattern

As we demonstrated earlier in the report, our game consists of three layers. The User Interface subsystem calls operations of Game Control subsystem and Game Control layer calls operations of Game Model subsystem. However, both Game Control and Game Model subsystems contain many classes and include complex relations. Hence, all layers have access to any class it needs in the lower layer which results in Ravioli design. This design is destroying the clarity and modularity of the system. The upper layer needs to know all details of the lower layer in order to call functions, which increases coupling. Hence, we decided to apply Facade Pattern to our system and provide unified interfaces for callable subsystems.

Facade pattern is applicable to our system. Selecting the most general class in the subsystem and extending it to provide a unified interface is enough to adopt the design. With Facade pattern, one class of a subsystem takes the role of 'Facade class' and provides a simple interface for the whole subsystem. Hence, it becomes very easy to use the subsystem. Any call to subsystem only uses the methods specified in the unified interface and the Facade class then communicates with other subsystem elements to execute the call. In our system, GameManager is the Facade class of Game Control subsystem. GameManager provides all operations that the Game Control layer can handle and hence, User Interface subsystem only calls operations of GameManager. GameManager then leads the command to necessary subsystem components. Facade class of Game Model subsystem, on the other hand, is Level class. Level class contains all game entities hence Game Control layer calls Level to obtain any game object or its operations.

The below diagram denotes how we applied Facade Pattern to Game Control subsystem:

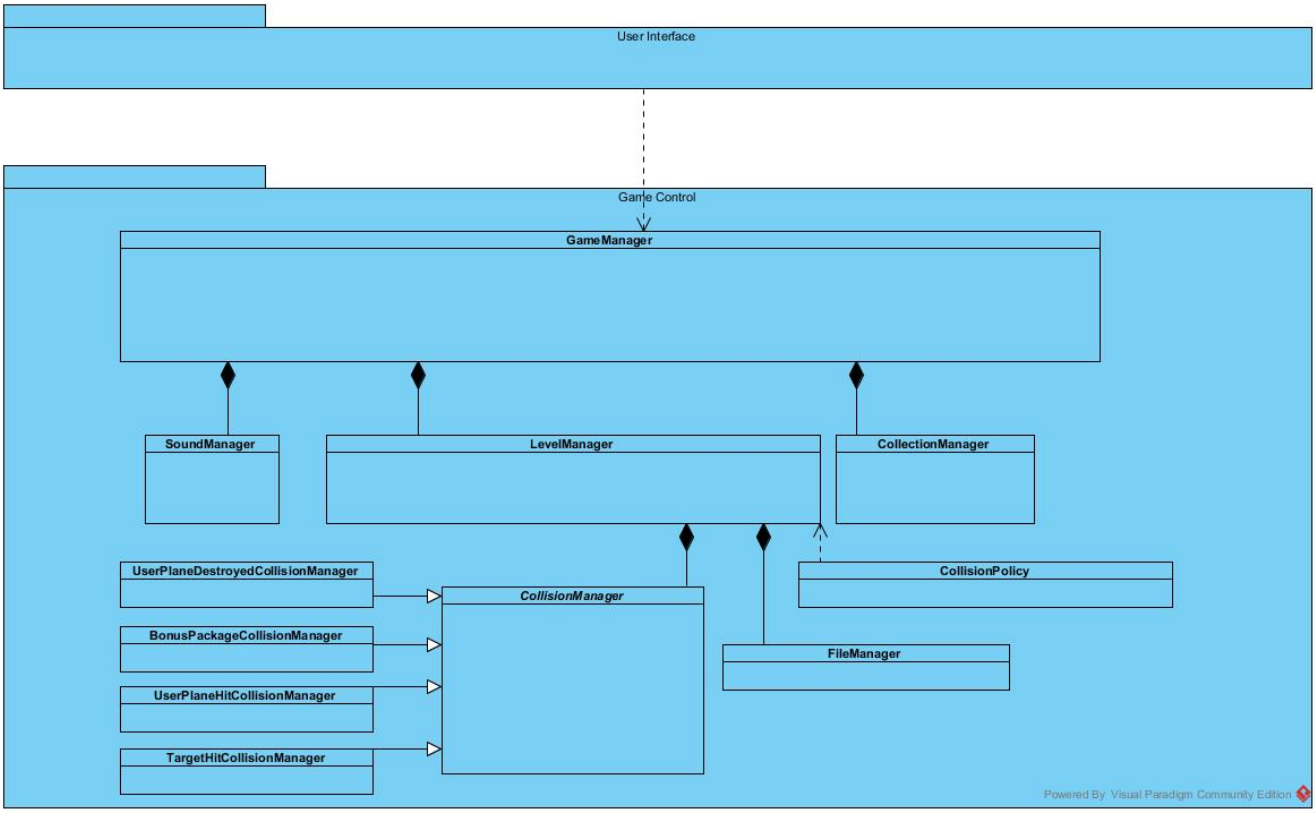

Figure 70 Facade Pattern Application to Game Control Subsystem

In this diagram it is shown that the GameManager is the Facade class, it contains all Game Control classes within and User Interface layer communicates only with GameManager class for all control operations. Hence GameManager presents a unified interface to Game Control subsystem.

The below diagram denotes how we applied Facade Pattern to Game Model subsystem:

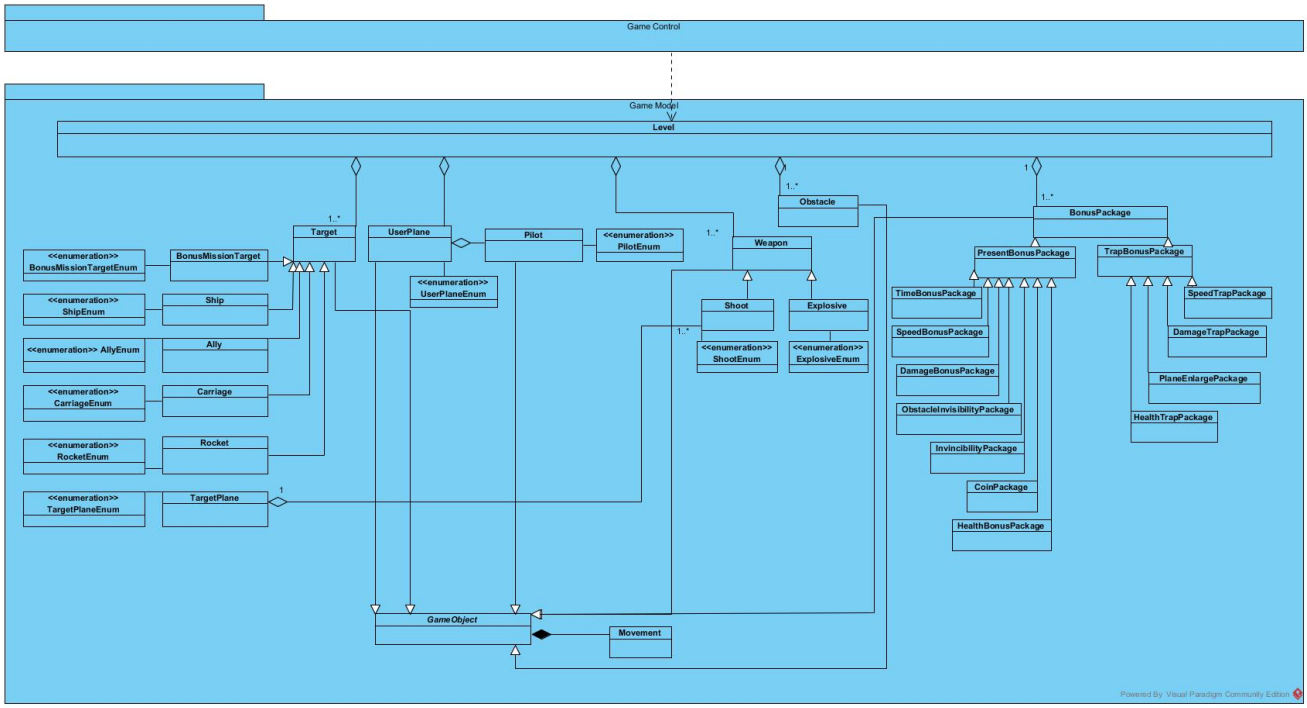

Figure 71 Facade Pattern Application to Game Model Subsystem

In this diagram it is shown that the Level is the Facade class, it contains all game entities and Game Control subsystem communicates only with Level for all data operations. Hence, Level presents a unified interface to Game Model subsystem.

With Facade pattern, we lowered the coupling between subsystems and we managed to increase clarity of the design. We managed to hide details of a subsystem from the caller subsystem.

5.1.2. Observer Design Pattern

In Sky Wars, we have a GameManager class which provides all game related operations. Therefore, this class contains all states of the game within. User Interface subsystem classes interact with the GameManager class for all information of the game itself such as collection status, level status, states of all game objects. Multiple panels are requesting GameManager data to decide view details. For instance, LevelPanel is requesting coordinates of the game object, StorePanel and CollectionPanel are requesting collection details from GameManager. This current system is disadvantageous for some reasons. We designed our game using the three-tier architecture, hence we want to minimize the dependencies between layers. With this architecture, the views and GameManager are highly coupled. Furthermore, it is very hard to add new views without changing the GameManager and hard to change GameManager without affecting the view components. In order to handle these issues, we followed Observer Design Pattern.

With this pattern, the system becomes highly extensible. It is very easy to add a new view element without changing GameManager class or any other component. The system also maintains consistency between game state and the views. Observer Pattern is applicable to our project. We already have update methods in the views. Instead of requesting game details from panels, we need to

add panels to GameManager as observers and when states change, GameManager notifies all its observers.

The below diagram denotes how we applied Observer Pattern to our design:

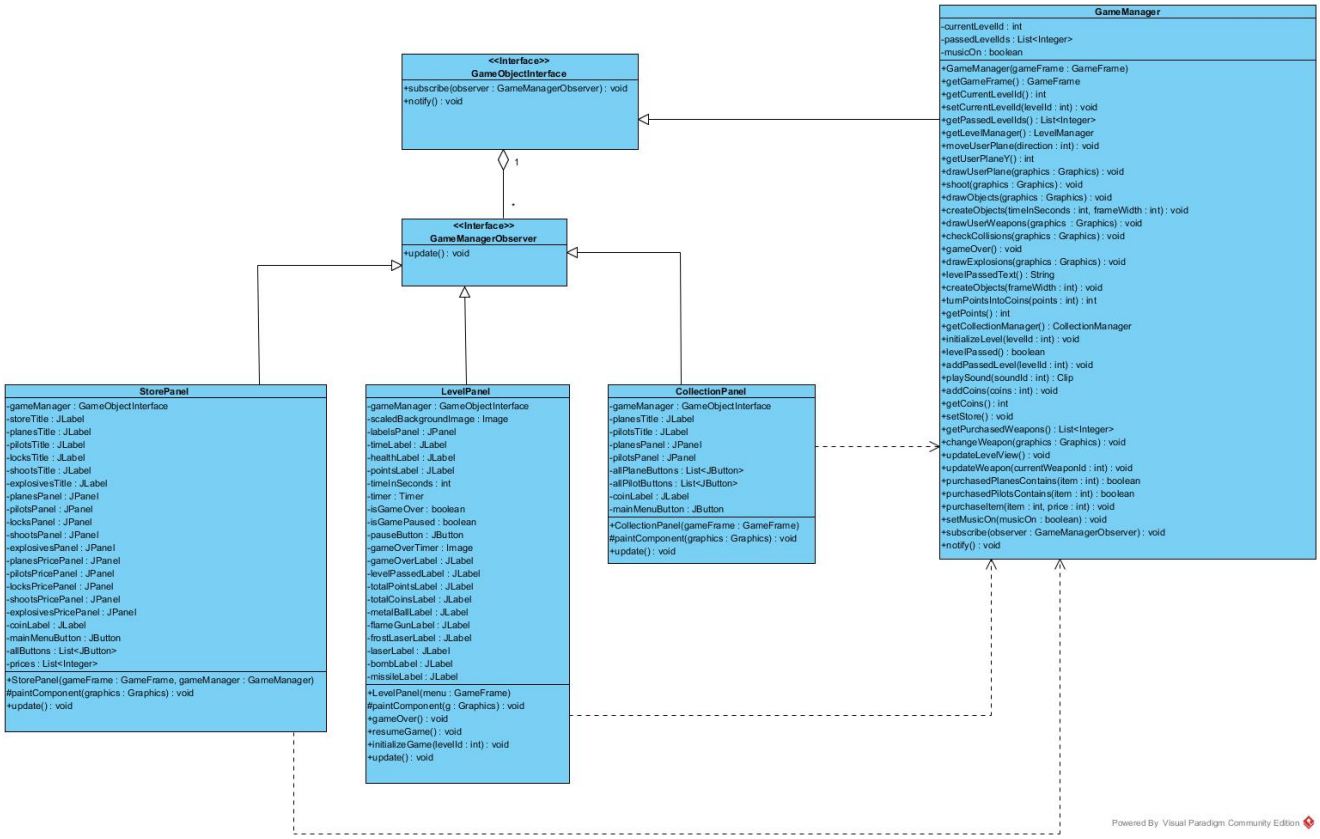

Figure 72 Observer Pattern Application to Sky Wars

In this diagram GameObjectInterface denotes the subject class in the Observer Pattern. It is an interface that generalizes the subject, GameManager. GameManager, therefore corresponds to the concrete subject. GameManager contains the actual states of the game. GameObjectInterface forces its descendants to implement subscribe(GameManagerObserver observer) and notify() methods. Hence, GameManager implements subscribe and notify methods. We created another interface class named GameManagerObserver which corresponds to Observer class of Observer pattern. GameManagerObserver contains the method update(). Hence all of its descendant implement update method. StorePanel, LevelPanel and CollectionPanel classes implement GameManagerObserver. They are concrete observers that request game states in order to determine panel details.

Basically, the LevelPanel, CollectionPanel and StorePanel classes implement GameManagerObserver interface. GameObjectInterface has multiple GameManagerObservers. GameManager implements the GameObjectInterface. Finally; LevelPanel, CollectionPanel and StorePanel call GameManager for their operations.

The Observer Pattern functions as the following: First the views that need game states implement GameManagerObserver interface and its update method. Then they subscribe to GameManager class and these observers are added to GameManager's list of observers. Then, whenever GameManager changes state, the notify method of the class is called. In notify() method update methods of all observers in the observer list are called. Hence the consistency between GameManager and panels is maintained.

With this pattern, it is now very easy to add a new view or alter subject classes. Hence, we

managed to make our system extensible.

5.1.3. Strategy Design Pattern

In Sky Wars, game objects collide and different algorithms are executed according to types of the collided objects. A class CollisionManager contained within LevelManager is handling all collision operations. With this design, if statements are determining the collision strategies and it is very hard to add a new algorithm to this structure without affecting multiple classes. Furthermore, the algorithms are closed to change and they are closed to other clients. Therefore, we applied Strategy Design Pattern to our game in order to apply open-closed principle.

With Strategy Pattern, we manage to separate algorithms from application and gain the ability to change algorithms during run-time. Different collision algorithms are implemented independently and the program changes current algorithm according to the collision type. Strategy Pattern is applicable to our system since we already have a CollisionManager that contains different algorithms within. It is enough to separate these algorithms from GameManager and create a Policy for changing the algorithm according to collision type.

The below diagram denotes how we applied Strategy Pattern to our system:

Figure 73 Strategy Pattern Application to Sky Wars

LevelManager class represents the client or context class which detects collisions and sends them to CollisionManager. CollisionManager corresponds to abstract class representing abstract strategy. This class provides an interface for concrete strategies, algorithms. UserPlaneDestroyedCollisionManager, UserPlaneHitCollisionManager, TargetHitCollisionManager and BonusPackageCollisionManager classes all extend CollisionManager. They all implement handleCollision(obj1, obj2) abstract method of CollisionManager and they use specific algorithms to handle the collision. CollisionPolicy class corresponds to Policy class of Strategy pattern that determines which algorithm will be used in different collision situations.

When a collision occurs, LevelManager detects collision and identifies the objects involved. Then it sets the policy of CollisionPolicy according to collision objects. Then the PolicyManager sets the CollisionManager of LevelManager according to the selected strategy. LevelManager calls the

handleCollision method of the current instance of CollisionManager. Then the active concrete algorithm class handles the collision with its specific algorithm. For instance, if user plane and an enemy weapon has collided, LevelManager sets the policy to userPlaneHit. Then CollisionPolicy sets the CollisionManager of LevelManager to UserPlaneHitCollisionManager instance. Then LevelManager calls UserPlaneHitCollisionManager to handle the operation. UserPlaneHitCollisionManager handles it by updating object states.

With Strategy Pattern, it is possible to extend the list of collisions without affecting other classes. If another strategy for collisions is developed, it is enough to add the new strategy and extend the policy only. With this pattern, our design became more extensible and flexible.

5.2. Class Interfaces

5.2.1. User Interface Subsystem Class Interfaces

GameFrame Class

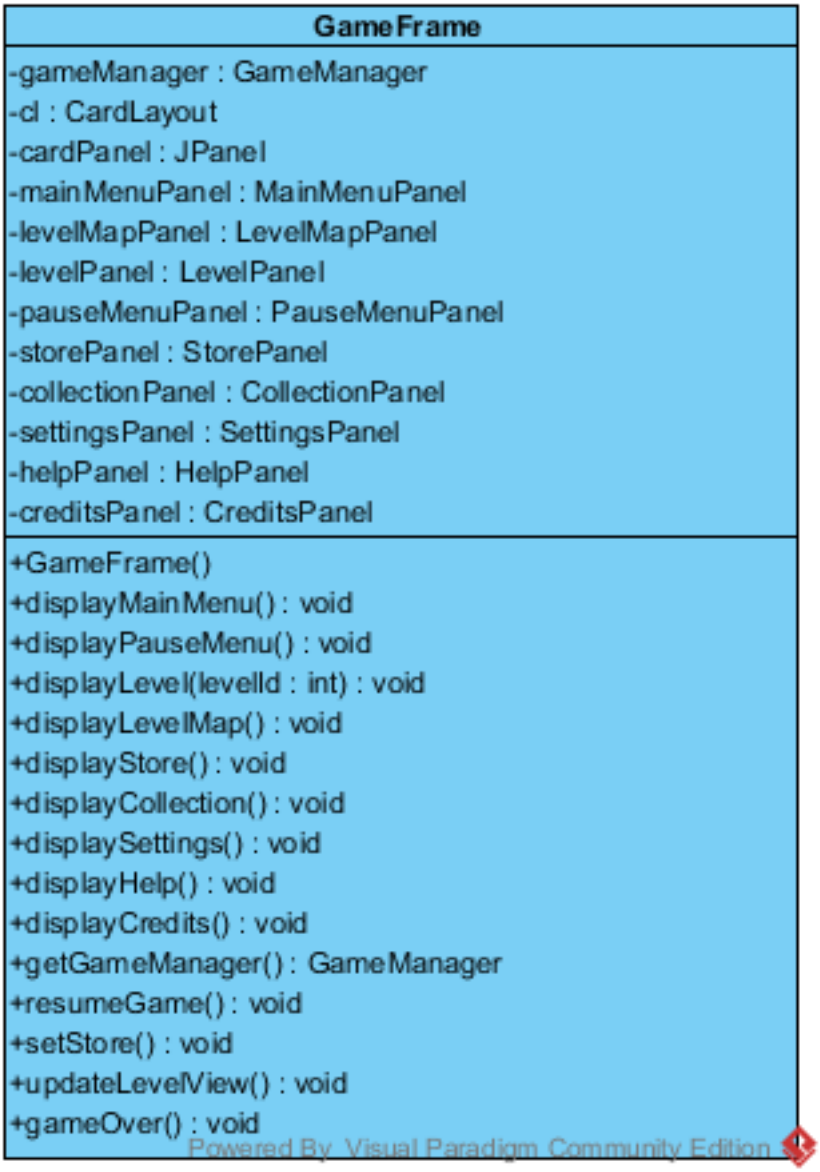

Figure 74 GameFrame Class Diagram

GameFrame class extends JFrame and it contains all game screens. GameFrame is responsible from

displaying screens and changing them.

Attributes
private GameManager gameManager holds a reference to GameManager
private CardLayout cl is the card layout of the frame which handles changing of screens
private JPanel cardPanel is the JPanel which keeps and arranges all panels
private MainMenuPanel mainMenuPanel is Main Menu Screen panel
private LevelMapPanel levelMapPanel is Level Map Screen panel
private LevelPanel levelPanel is Level Screen panel
private PauseMenuPanel pauseMenuPanel is Pause Menu Screen panel
private StorePanel storePanel is Store Screen panel
private CollectionPanel collectionPanel is Collection Screen panel
private SettingsPanel settingsPanel is Settings Screen panel
private HelpPanel helpPanel is Help Screen panel
private CreditsPanel creditsPanel is Credits Screen panel

Constructor
public GameFrame() is the constructor GameFrame class. All other controllers are created by GameFrame.

Methods
public void displayMainMenu() displays Main Menu Screen
public void displayPauseMenu() displays Pause Menu Screen
public void displayLevel(int levelId) displays Level Screen with the given level id
public void displayLevelMap() displays Level Map Screen
public void displayStore() displays Store Screen
public void displayCollection() displays Collection Screen
public void displaySettings() displays Settings Screen
public void displayHelp() displays Help screen
public void displayCredits() displays Credits screen
public GameManager getGameManager() returns reference to game manager
public void resumeGame() resumes Level screen after paused and resumed
public void setStore() sets store view by locking/unlocking items
public void updateLevelView() updates Level screen
public void gameOver() ends the game by telling it to level panel

PauseMenuPanel Class

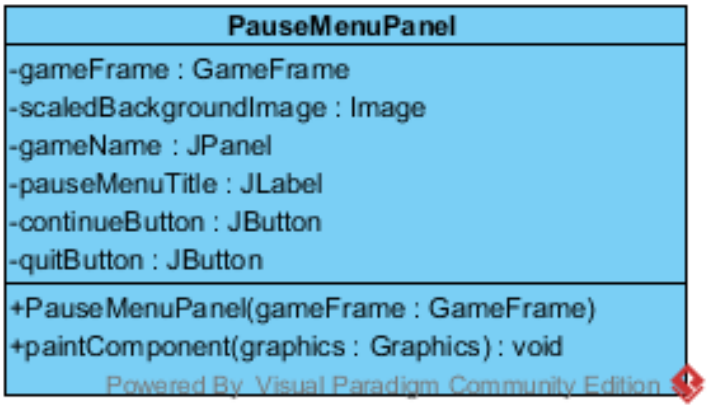

Figure 75 PauseMenuPanel Class Diagram

PauseMenuPanel is a JPanel and it is contained within GameFrame class. This panel represents the Pause Menu screen.

Attributes
private GameFrame gameFrame is a reference to game frame which contains pause menu panel
private Image scaledBackgroundImage is the background image of pause menu screen
private JLabel gameName holds the title of game
private JLabel pauseMenuTitle holds the pause menu title
private JButton continueButton is the JButton that returns user to level screen
private JButton quitButton is the JButton that returns user to level map

Constructor
public PauseMenuPanel(GameFrame gameFrame) is the constructor of pause menu panel. It takes a reference to game frame which creates pause menu panel.

Methods
protected void paintComponent(Graphics g) overrides the parent method to draw the panel on screen.

LevelPanel Class

Figure 76 LevelPanel Class Diagram

LevelPanel is a JPanel and it is contained within GameFrame class. This panel represents the Level screen. Furthermore, LevelPanel observes GameManager, hence it implements GameManagerObserver interface.

Attributes

private GameFrame gameFrame is a reference to game frame which contains collection panel
private Image scaledBackgroundImage is the background image of level screen
private GameObjectInterface gameManager is a reference to game manager the panel observes
private JPanel labelsPanel is the panel that contains all panels of level view
private JLabel timeLabel is the label that denotes the level time
private JLabel healthLabel is the label that shows the user plane health
private JLabel pointsLabel is the label that denotes total points user has collected
private int timeInSeconds keeps the seconds passed after level starts
private Timer timer is the timer for keeping time for the level
private boolean isGameOver represents whether the game is over or not

private boolean isGamePaused represents whether game is paused or not
private JButton pauseButton is the JButton for pausing the game
private Timer gameOverTimer is the timer for counting the time for displaying end of level messages
private JLabel gameOverLabel is the label that holds "Game Over" title
private JLabel levelPassedLabel is the label for indicating end of level status
private JLabel totalPointsLabel is the label used for displaying total points collected at the end of the level
private JLabel totalCoinsLabel is the label that displays the total coins earned in level
private JLabel metalBallLabel is the label that denotes the number of metal balls user currently has
private **JLabel flameGunLabel** is the label that denotes the number flame guns user currently has
private **JLabel frostLaserLabel** is the label that denotes the number of frost lasers user currently has
private **JLabel laserLabel** is the label that denotes the number of lasers user currently has
private **JLabel bombLabel** is the label that denotes the number of bombs user currently has
private **JLabel missileLabel** is the label that denotes the number of missiles user currently has

Constructor
public LevelPanel(GameFrame gameFrame) is the constructor of level panel. It takes a reference to game frame which creates level panel.

Methods
protected void paintComponent(Graphics g) overrides the parent method to draw the panel on screen.
public void gameOver() ends the game by stopping the time, displaying game over labels and returning to main menu
public void resumeGame() resumes the game after it is paused and then continued
public void initializeGame(int levelId) initializes the screen by setting the view according to levelId and starting time
public void update() is the method specified in GameManagerObserver interface, which updates the Level screen when gameManager changes state

CollectionPanel Class

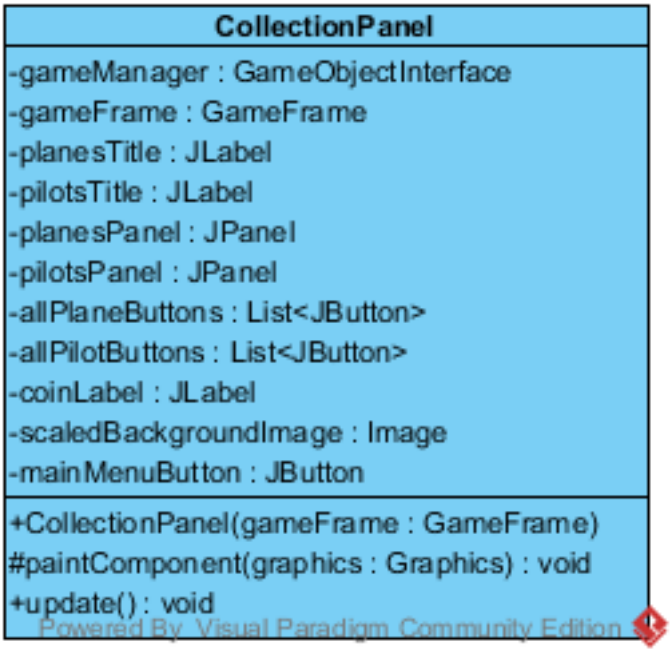

Figure 77 CollectionPanel Class Diagram

CollectionPanel is a JPanel and it is contained within GameFrame class. This panel represents the Collection screen. Furthermore, CollectionPanel observes GameManager, hence it implements GameManagerObserver interface.

Attributes
private GameObjectInterface gameManager is a reference to game manager the panel observes
private GameFrame gameFrame is a reference to game frame which contains collection panel
private JLabel planesTitle is the JLabel that holds the "Planes" title
private JLabel pilotsTitle is the JLabel that holds the "Pilots" title
private JPanel planesPanel is the JPanel that holds user planes in collection
private JPanel pilotsPanel is the JPanel that holds pilots in collection
private List<JButton> allPlaneButtons is the list of user plane buttons which allows user to select a plane
private List<JButton> allPilotButtons is the list of pilot buttons which allows user to select a pilot
private JLabel coinLabel is the JLabel that shows user coins
private Image scaledBackgroundImage is the background image of collection screen
private JButton mainMenuButton is the JButton used to return to main menu

Constructor
public CollectionPanel(GameFrame gameFrame) is the constructor of collection panel. It takes a reference to game frame which creates collection panel.

Methods
protected void paintComponent(Graphics g) overrides the parent method to draw the panel on screen
public void update() method is the method specified in GameManagerObserver interface, which updates the Collection screen when GameManager changes state

StorePanel Class

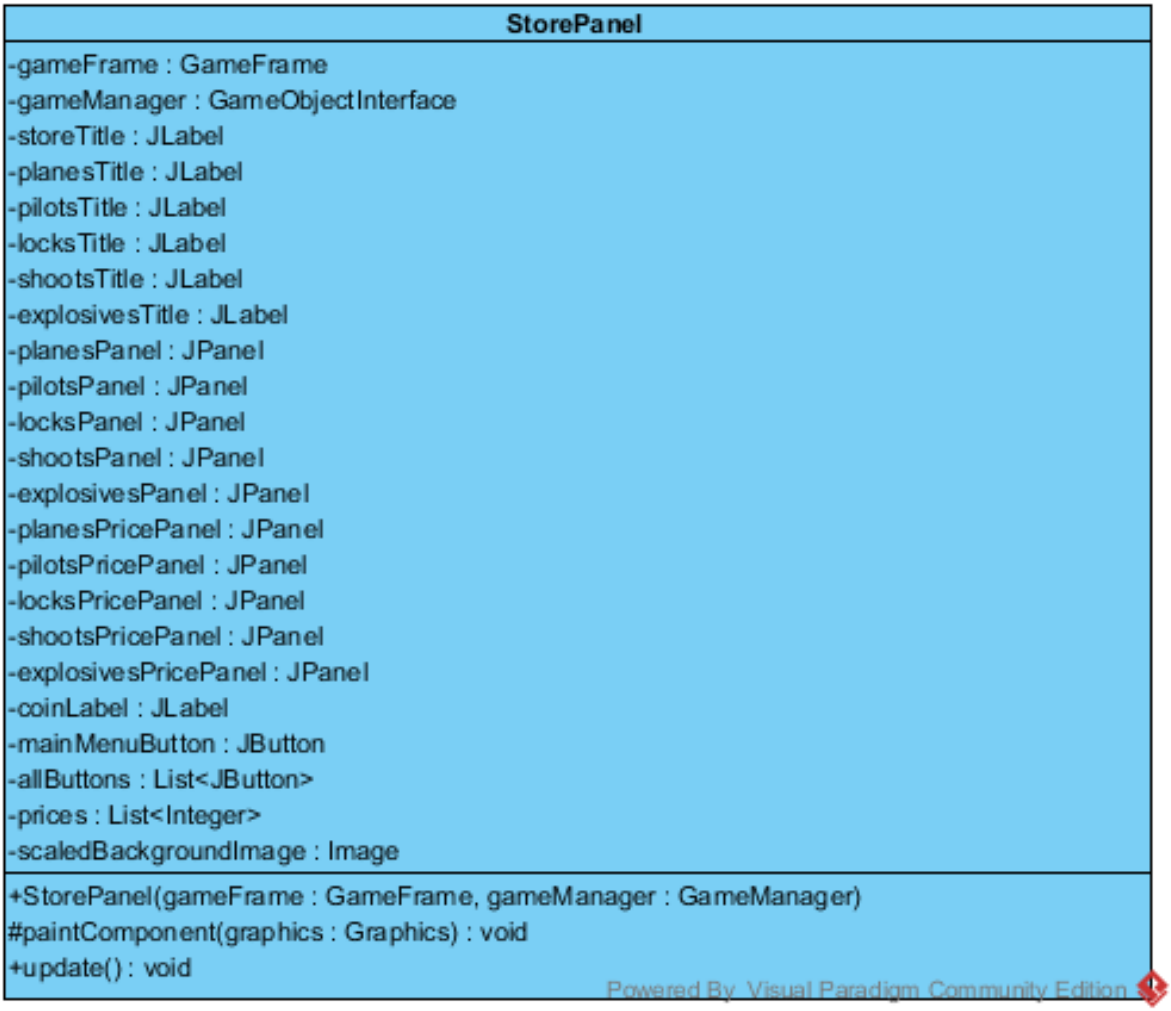

Figure 78 StorePanel Class Diagram

StorePanel is a JPanel and it is contained within GameFrame class. This panel represents the Store screen. Furthermore, StorePanel observes GameManager, hence it implements GameManagerObserver interface.

Attributes
private GameFrame gameFrame is a reference to game frame which contains store panel
private GameObjectInterface gameManager is a reference to game manager the panel observes
private JLabel storeTitle holds the “Store” title
private JLabel planesTitle holds the “Planes” title
private JLabel pilotsTitle holds the “Pilots” title
private JLabel locksTitle holds the “Bonus Package Locks” title
private JLabel shootsTitle holds the “Shoots” title
private JLabel explosivesTitle holds the “Explosives” title
private JPanel planesPanel is the panel that holds all plane buttons
private JPanel pilotsPanel is the panel that holds all pilot buttons
private JPanel locksPanel is the panel that holds all package lock buttons
private JPanel shootsPanel is the panel that holds all shoot buttons
private JPanel explosivesPanel is the panel that holds all explosive buttons

private JPanel planesPricePanel is the panel that holds price labels of all planes
private JPanel pilotsPricePanel is the panel that holds price labels of all pilots
private JPanel locksPricePanel is the panel that holds price labels of all bonus package locks
private JPanel shootsPricePanel is the panel that holds price labels of all shoots
private JPanel explosivesPricePanel is the panel that holds price labels of all explosives
private JLabel coinLabel is the label that shows the amount of user coins
private JButton mainMenuButton is a JButton used to go back to main menu
private List<JButton> allButtons is the list of all JButtons corresponding to all items sold in the store
private List<Integer> prices holds the list of prices of all items
private Image scaledBackgroundImage is the background image of store screen

Constructor
public StorePanel(GameFrame gameFrame, GameManager gameManager) is the constructor of store panel. It takes a reference to game frame which creates store panel and a reference to game manager which is observed.

Methods
protected void paintComponent(Graphics g) overrides the parent method to draw the panel on screen.
public void update() method is the method specified in GameManagerObserver interface, which updates the Store screen when GameManager changes state

MainMenuPanel Class

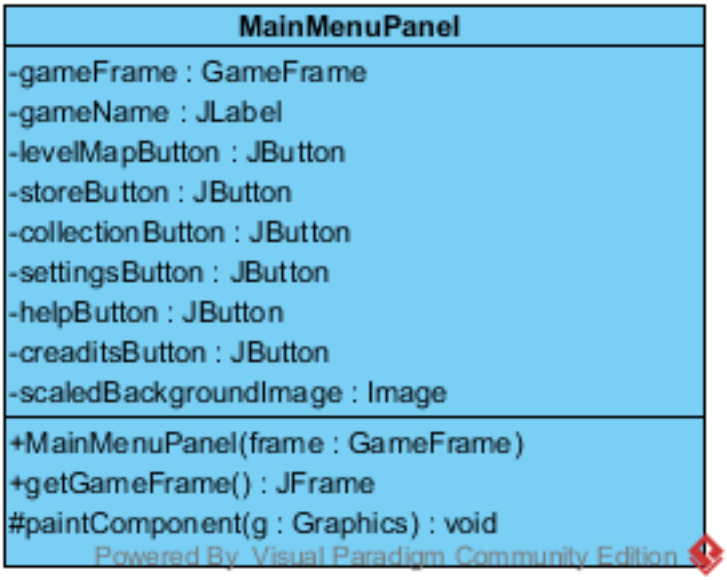

Figure 79 MainMenuPanel Class Diagram

MainMenuPanel is a JPanel and it is contained within GameFrame class. This panel represents the Main Menu screen.

Attributes
private GameFrame gameFrame is a reference to game frame which contains main menu panel
private JLabel gameName holds the title of game
private JButton levelMapButton is the JButton used to go to Level Map screen

private JButton storeButton is the JButton used to go to Store screen
private JButton collectionButton is the JButton used to go to Collection screen
private JButton settingsButton is the JButton used to go to Settings screen
private JButton helpButton is the JButton used to go to Help screen
private JButton creditsButton is the JButton used to go to Credits screen
private Image scaledBackgroundImage is the background image of main menu screen

Constructor
public MainMenuPanel(GameFrame gameFrame) is the constructor of main menu panel. It takes a reference to game frame which creates main menu panel.

Methods
protected void paintComponent(Graphics g) overrides the parent method to draw the panel on screen
public GameFrame getGameFrame() returns the reference to game frame

LevelMapPanel Class

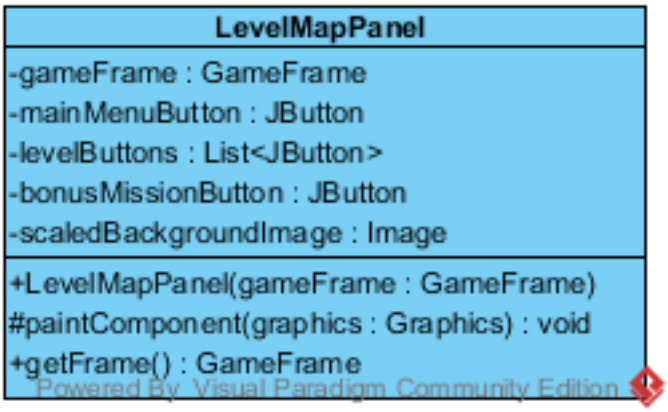

Figure 80 LevelMapPanel Class Diagram

LevelMapPanel is a JPanel and it is contained within GameFrame class. This panel represents the Level Map screen.

Attributes
private GameFrame gameFrame is a reference to game frame which contains level map panel
private JButton mainMenuButton is the JButton used to return to main menu
private List<JButton> levelButtons is the list of JButtons used to access to levels
private JButton bonusMissionButton is the JButton used to play bonus mission
private Image scaledBackgroundImage is the background image of level map screen

Constructor
public LevelMapPanel(GameFrame gameFrame) is the constructor of level map panel. It takes a reference to game frame which creates the level map panel.

Methods
protected void paintComponent(Graphics grahics) overrides the parent method to draw the panel on screen
public GameFrame getFrame() returns the reference to game frame

CreditsPanel Class

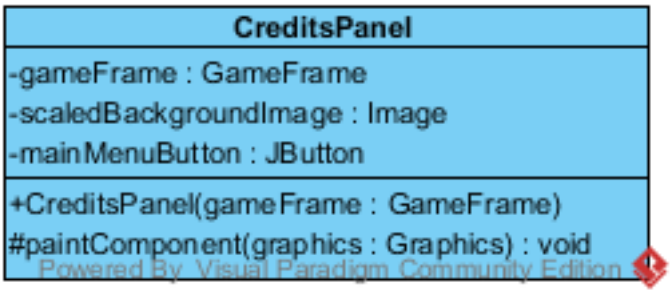

Figure 81 CreditsPanel Class Diagram

CreditsPanel is a JPanel and it is contained within GameFrame class. This panel represents the Credits screen.

Attributes
private GameFrame gameFrame is a reference to game frame which contains credits panel
private Image scaledBackgroundImage is the background image of credits screen
private JButton mainMenuButton is the JButton that returns user to main menu screen

Constructor
public CreditsPanel(GameFrame gameFrame) is the constructor of credits panel. It takes a reference to game frame which creates credits panel.

Methods
protected void paintComponent(Graphics graphics) overrides the parent method to draw the panel on screen

HelpPanel Class

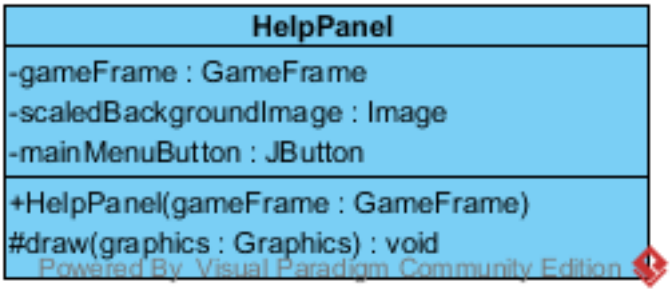

Figure 82 HelpPanel Class Diagram

HelpPanel is a JPanel and it is contained within GameFrame class. This panel represents the Help screen.

Attributes
private GameFrame gameFrame is a reference to game frame which contains help panel
private Image scaledBackgroundImage is the background image of help screen
private JButton mainMenuButton is the JButton to return to main menu

Constructor
public HelpPanel(GameFrame gameFrame) is the constructor of help panel. It takes a reference to game frame which creates help panel.

Methods
protected void paintComponent(Graphics g) overrides the parent method to draw the panel on screen.

SettingsPanel Class

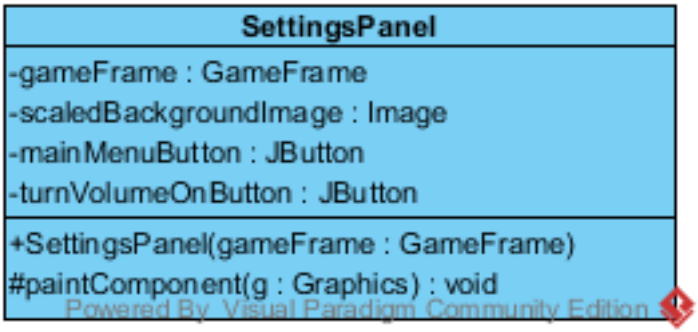

Figure 83 SettingsPanel Class Diagram

SettingsPanel is a JPanel and it is contained within GameFrame class. This panel represents the Settings screen.

Attributes
private GameFrame gameFrame is a reference to game frame which contains pause menu panel
private Image scaledBackgroundImage is the background image of settings screen
private JButton mainMenuButton is the JButton that returns user to main menu
private JButton turnVolumeOnButton is the JButton to turn the music on or off

Constructor
public SettingsPanel(GameFrame gameFrame) is the constructor of settings panel. It takes a reference to game frame which creates settings panel.

Methods
protected void paintComponent(Graphics g) overrides the parent method to draw the panel on screen

GameObjectInterface Interface

Figure 84 GameObjectInterface Class Diagram

GameObjectInterface is an interface which provides an abstraction for GameManager. This class is created for adopting Observer pattern to system.

Methods
public void subscribe(GameManagerObserver observer) is the method interface implementers are supposed to implement. It adds an observer to the observer list of subject.
public void notify() is the method interface implementers are supposed to implement. It allows subject to notify observers.

GameManagerObserver Interface

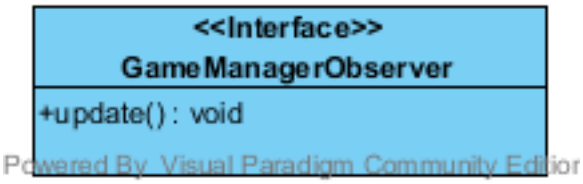

Figure 85 GameManagerObserver Class Diagram

GameManagerObserver is an interface which provides an abstraction for observers of GameManager. This class is created for adopting Observer pattern to system.

Methods
public void update() is the method interface implementers are supposed to implement. It allows observer to update itself.

5.2.2. Game Control Subsystem Class Interfaces

GameManager Class

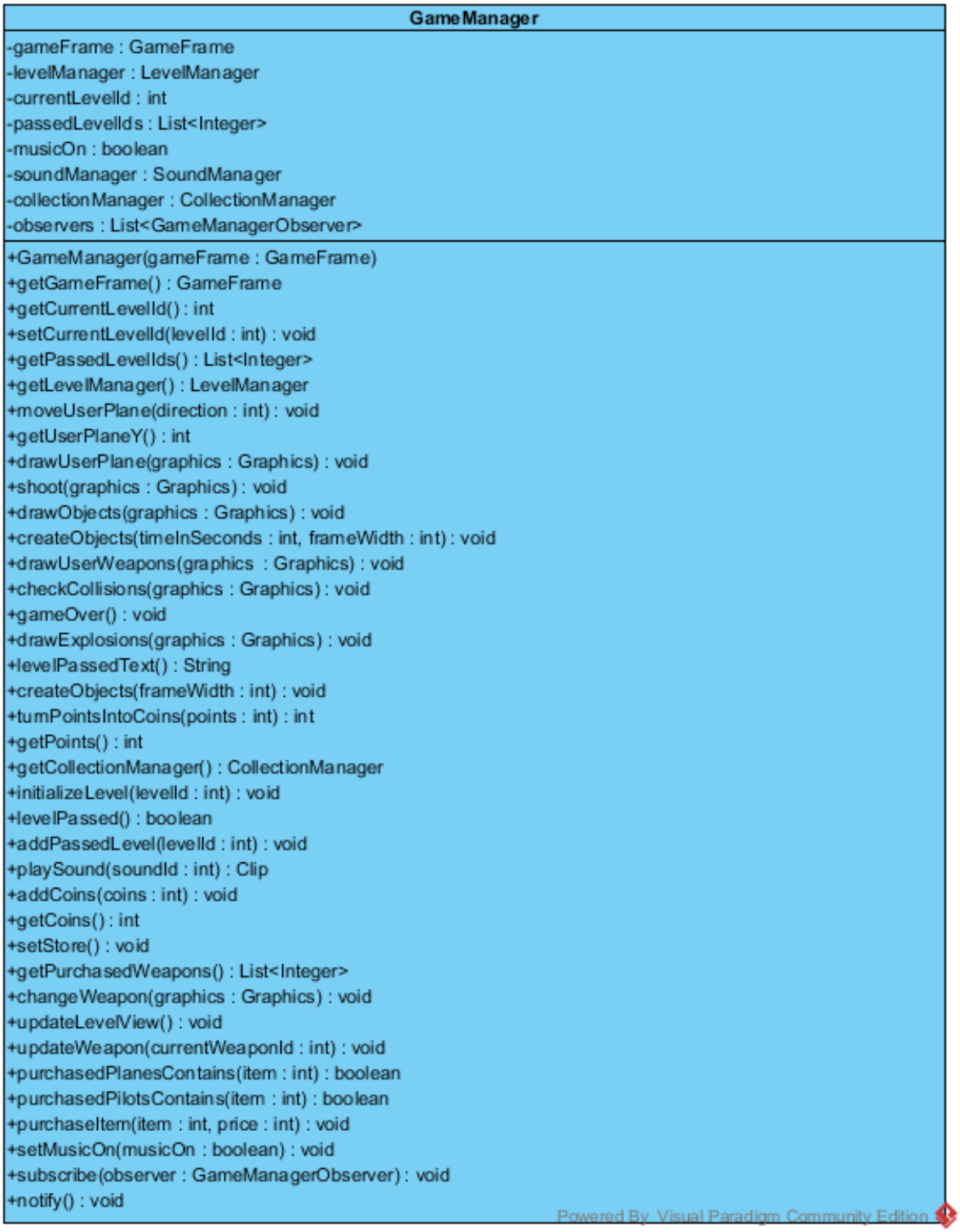

Figure 86 GameManager Class Diagram

GameManager class is the fundamental controller class that contains all other controllers and manages all game operations. Furthermore GameManager is the Facade class of Game Control subsystem and it implements GameObjectInterface for the application of Observer pattern.

Attributes

private GameFrame gameFrame is a reference to game frame which contains all panels of the game

private LevelManager levelManager relates LevelManager class and GameManager class and

handles the level operations
private int currentLevelId holds the id of current level that player is playing
private List<Integer> passedLevelIds has the passed level ids as a list of integers
private boolean musicOn is a boolean attribute to check if the sound effects and music must be played or not
private SoundManager soundManager manages all the sounds during the game play.
private CollectionManager collectionManager holds this CollectionManager type attribute in order to manage the collection, purchased items and selections of user
private List<GameManagerObserver> observers holds the list of observer panels of GameManager

Constructor
public GameManager(GameFrame gameFrame) takes a GameFrame type parameter and creates a new GameManager object that holds all the current status of the game.

Methods
public GameFrame getGameFrame() returns GameFrame attribute of the class
public int getCurrentLevelId() returns the current level id
public void setCurrentLevelId(int currentLevelId) updates the currentLevelId by setting the attribute to the levelId parameter
public List<Integer> getPassedLevelIds() returns all the ids of passed level by user, as a list
public LevelManager getLevelManager() returns the level manager
public void moveUserPlane(int direction) informs the level manager about which direction the user wants the plane to move, by taking the direction as a parameter
public int getUserPlaneY() returns y coordinate of the user plane by getting the coordinate from LevelManager
public void drawUserPlane(Graphics graphics) uses levelManager to have the user plane drawn on the screen
public void shoot(Graphics graphics) uses levelManager to have user plane shoot
public void drawObjects(Graphics graphics) uses levelManager to have objects, which specified within a level, drawn on the screen
public void createObjects(int timeInSeconds, int frameWidth) uses levelManager to have all the objects, which should appear in the specified time, created and placed to screen according to frameWidth
public void drawUserWeapons(Graphics graphics) uses levelManager to have all user weapons drawn to screen
public void checkCollisions(Graphics graphics) uses levelManager to check collisions in the game play
public void gameOver() ends the played game by using gameFrame object
public void drawExplosions(Graphics graphics) uses levelManager to draw all explosions to screen
public String levelPassedText() gets a text about whether the played level is passed or failed from levelManager and returns it as a string
public void createObjects(int frameWidth) uses levelManager to create initial game objects at the beginning of levels
public int turnPointsIntoCoins(int points) uses levelManager to turn level points to coins at the end of the played level
public int getPoints() returns level points by using levelManager
public CollectionManager getCollectionManager() returns collection manager
public void initializeLevel(int levelId) initializes corresponding level

public boolean levelPassed() returns whether level is passed or not by using levelManager
public void addPassedLevel(int currentLevelId) uses levelManager to add current level to passed levels, if the current level is passed, by telling it to level manager
public Clip playSound(int soundId) requests sound manager to return the sound clip
public void addCoins(int coins) increases user coins by telling it to collection manager
public int getCoins() returns the amount of total coins by using CollectionManager
public void setStore() updates the store
public List<Integer> getPurchasedWeapons() returns list of purchased weapons by using collectionManager
public void changeWeapon(Graphics graphics) uses collectionManager to change currently selected weapon for shooting
public void updateLevelView() updates level view by using gameFrame
public void updateWeapon(int currentWeaponId) updates weapon number in screen when weapons are used
public boolean purchasedPlanesContains(int item) uses CollectionManager to return whether purchased planes contain a particular item or not
public boolean purchasedPilotsContains(int item) uses collectionManager to return whether purchased planes contain a particular item or not
public void purchaseItem(int id, int price) uses collectionManager to purchase an item
public void setMusicOn(boolean musicOn) turns the music on or off
public void subscribe(GameManagerObserver observer) allows an observer to add itself to observer list of the GameManager
public void notify() updates all observers

SoundManager Class

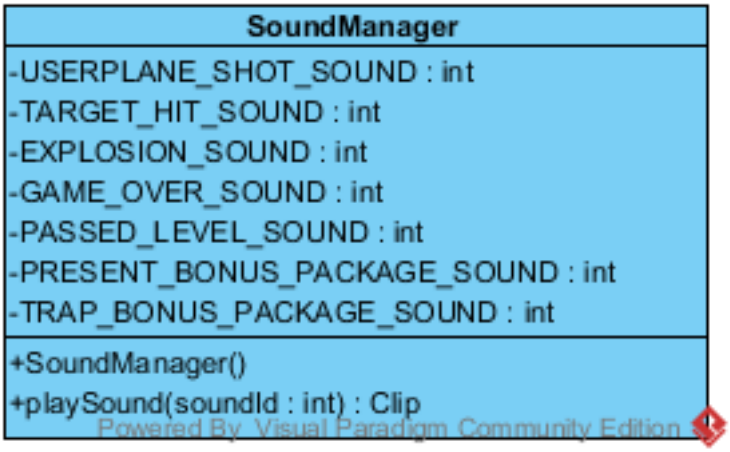

Figure 87 SoundManager Class Diagram

SoundManager class controls playing sounds and music.

Constants
public static int USER_PLANE_SHOT_SOUND is a constant that holds the id of the sound that occurs when the user plane is shot
public static int TARGET_HIT_SOUND has the id of the sound occurs when shooting of the user hits a target
public static int EXPLOSION_SOUND is the id of the explosion sound constant
public static int GAME_OVER_SOUND holds the id of the sound plays when game is over
public static int PASSED_LEVEL_SOUND holds the id of the sound plays when a level is passed

public static int PRESENT_BONUS_PACKAGE_SOUND has the id of the sound when user plane gets a present bonus package
public static int TRAP_BONUS_PACKAGE_SOUND has the id of the sound when user plane collides with a trap bonus package

Constructor
public SoundManager() is the constructor of SoundManager

Methods
public Clip playSound(int soundId) plays the sound with specified id

CollectionManager Class

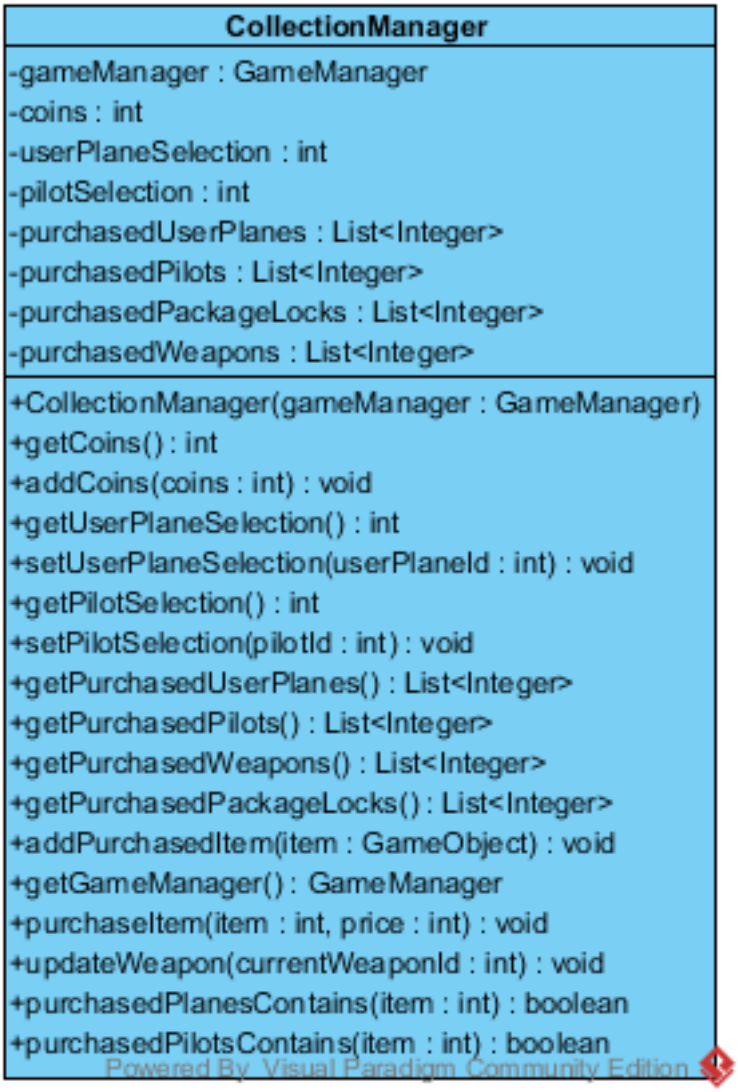

Figure 88 CollectionManager Class Diagram

CollectionManager handles all collection related operations.

Attributes
private GameManager gameManager provides a connection between GameManager and the collection

private int coins holds the total amount of coins
private int userPlaneSelection saves the current plane selection of the user by holding id of the plane
private int pilotSelection holds the currently selected character id
private List<Integer> purchasedUserPlanes has all the plane ids which has been purchased by the user as a list
private List<Integer> purchasedPilots has all the character ids, which has been purchased by the user, as a list
private List<Integer> purchasedPackageLocks holds levels of purchased package locks as integers according to "indices of the list – locks" mapping
private List<Integer> purchasedWeapons holds a list of total amounts of purchased weapons

Constructor
public CollectionManager(GameManager gameManager) creates the object that holds all the purchased items and relates them with the game play by the help of the gameManager parameter

Methods
public int getCoins() returns the total number of coins that user has
public void addCoins(int coins) increments coins by the given amount as parameter
public int getUserPlaneSelection() returns the id of the plane that user chose for the game play
public void setUserPlaneSelection(int userPlaneSelection) changes plane selection of user for the game by taking the id of the plane, that user lately chose, as an integer parameter and seting it to the userPlaneSelection attribute
public int getPilotSelection() returns the id of the latest character that user chose for the game play
public void setPilotSelection(int pilotSelection) changes plane selection of user for the game by taking the id of the character, that user lately chose, as an integer parameter and sets it to the pilotSelection attribute
public List<Integer> getPurchasedUserPlanes() returns the list of purchased user plane ids
public List<Integer> getPurchasedPilots() returns the list of purchased pilot character ids
public List<Integer> getPurchasedWeapons() returns the list of purchased weapon ids
public List<Integer> getPurchasedPackageLocks() returns levels of purchased package locks as integers according to "indices of the list – locks" mapping
public void addPurchasedItem(GameObject item) adds a new purchased item to the collection according to the item's type
public GameManager getGameManager() returns the gameManager attribute of the class
public void purchaseItem(int id, int price) purchases a new item by taking item's id and price as parameters and updates the collection according to the change
public void updateWeapon(int currentWeaponId) updates number of corresponding weapons.
public boolean purchasedPlanesContains(int item) returns whether the list of purchased planes contains a plane with id item
public boolean purchasedPilotsContains(int item) returns whether the list of purchased pilots contains a pilot with id item

LevelManager Class

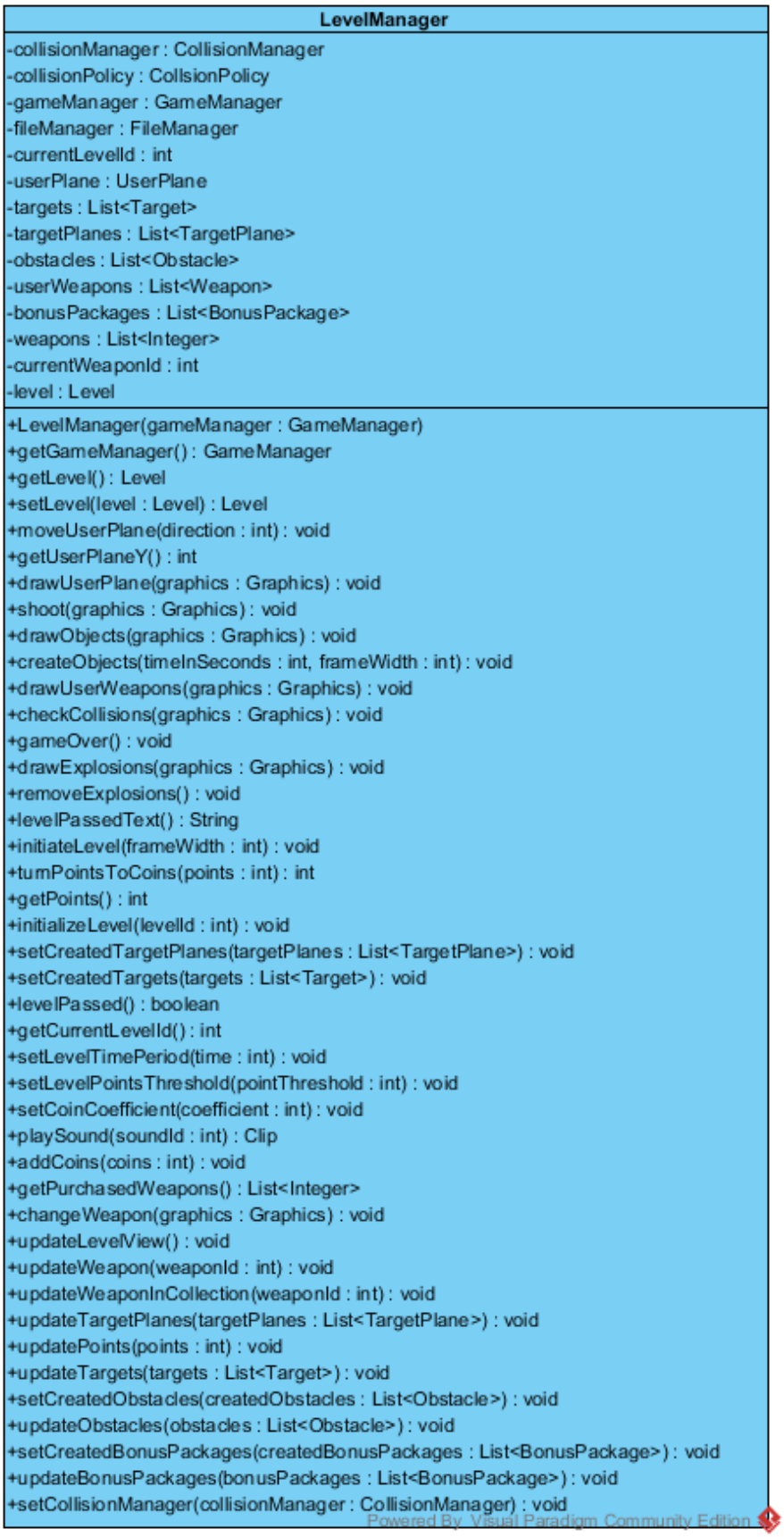

Figure 89 LevelManager Class Diagram

LevelManager class controls all level related operations.

Attributes
private CollisionManager collisionManager handles all the collision in the specific level
private CollisionPolicy collisionPolicy specifies the policy for current collision
private GameManager gameManager relates LevelManager to GameManager class and handles all game related operations
private FileManager fileManager holds a reference to FileManager that creates all game object of a specific level
private int currentLevelId holds the id of the current level in order to save which level the user last played
private UserPlane userPlane is reference for the user plane
private List<Target> targets holds the targets that come up in a specific level
private List<TargetPlane> targetPlanes holds the target planes that come up in a specific level
private List<Obstacle> obstacles holds the all obstacles that come up in a specific level
private List<Weapon> userWeapons holds the list of weapons user has fired
private List<BonusPackage>bonusPackages holds the bonus packages that come up in a specific level
private List<Integer> weapons holds the remaining amounts of the weapons user has
private int currentWeaponId holds the id of selected weapon for shooting during the game.
private Level level is a reference for current level

Constructor
public LevelManager(GameManager gameManager) is the constructor of LevelManager. It takes a reference to GameManager which creates the LevelManager.

Methods
public GameManager getGameManager() returns gameManager object
public Level getLevel() returns the current level
public void setLevel(Level level) updates current level object
public void moveUserPlane(int direction) uses the userPlane object to move the user plane in the specified direction within screen
public int getUserPlaneY() returns y coordinate of user plane by using userPlane object
public void drawUserPlane(Graphics graphics) draws user plane on screen by using userPlane object
public void shoot(Graphics graphics) implements shoot operation by creating a new user weapon
public void drawObjects(Graphics graphics) draws objects that are specified within a level, on the screen
public void createObjects(int timeInSeconds, int frameWidth) calls the level object to create game play objects on screen if the seconds match
public void drawUserWeapons(Graphics graphics) draws all user weapons to screen
public void checkCollisions(Graphics graphics) detects whether a collision has occurred and sends a collision to CollectionManager and ask it to handle it
public void gameOver() calls gameManager object to end the game
public void drawExplosions(Graphics graphics) calls gameManager object to draw all explosions.
public void removeExplosions() calls gameManager object to remove all explosions.
public String levelPassedText() gets a text about whether the played level is passed or failed from levelManager and returns it as a string
public void initiateLevel(int frameWidth) uses fileManager to create initial objects, according to the related level file

public int turnPointsIntoCoins(int points) turns points to coins at the end of a played level
public int getPoints() returns points collected during a level of game play
public void initializeLevel(int levelId) initializes a new level object
public void setCreatedTargetPlanes(List<TargetPlane> targetPlanes) initializes target planes from fileManager
public void setCreatedTargets(List<Target> targets)initializes target objects from fileManager
public boolean levelPassed() returns whether level is passed or not as a Boolean
public int getCurrentLevelId() returns the value of currentLevelId attribute
public void setLevelTimePeriod(int time) changes time period of the level attribute
public void setLevelPointThreshold(int pointTreshold) changes point threshold of the level
public void setCoinCoefficient(int coefficient) changes coin coefficient, which represents points per coin of a level
public Clip playSound(int soundId) returns the sound clip, which has the same id with the soundId parameter, by using gameManager
public void addCoins(int coins) calls gameManager to increase the amount of coins at the end of the played level by amount given as parameter
public List<Integer> getPurchasedWeapons() returns list of remaining purchased weapons at the end of the level with the help of "indices of the list –weapons" mapping
public void changeWeapon(Graphics graphics) changes the currently selected weapon during the game play
public void updateLevelView() updates level screen according to view changes by using gameManager
private void updateWeapon(int weaponId) updates remaining amount of weapons after shooting
public void updateWeaponInCollection(int weaponId) updates the weapons in collection of user when the user plane shoots
public void updateTargetPlanes(List<TargetPlane> targetPlanes) gets a list of target planes as parameter and updates the targetPlanes attribute according to it
public void updatePoints(int points) gets an integer as parameter and increments the total points user has collected in the last played level, according to the value of the parameter
public void updateTargets(List<Target> targets) gets a list of targets as parameter and updates the targets attribute according to it
public void setCreatedObstacles(List<Obstacle> obstacles) gets a list of bonus packages as parameter and updates the obstacles of a specific level, according to it
public void updateObstacles(List<Obstacle> obstacles) gets a list of obstacles as parameter and updates the obstacles attribute according to it
public void setCreatedBonusPackages(List<BonusPackage> bonusPackages) gets a list of bonus packages as parameter and updates the bonusPackages of a specific level, according to it
public void updateBonusPackages(List<BonusPackage> bonusPackages) gets a list of bonus packages as parameter and updates the bonusPackages attribute according to it
public void setCollisionManager (CollisionManager collisionManager) changes the current CollisionManager instance

FileManager Class

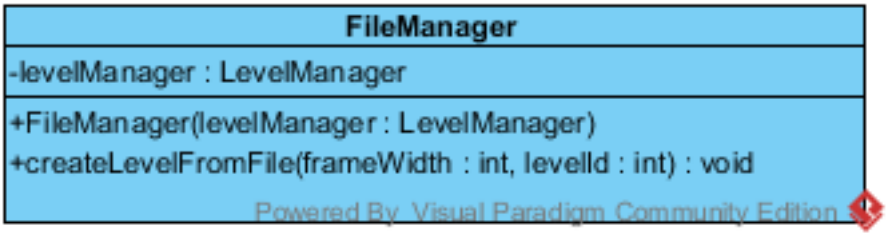

Figure 90 FileManager Class Diagram

FileManager class creates level objects by reading them from file.

Attributes
LevelManager levelManager holds a reference to LevelManager which creates file manager

Constructor
public FileManager(LevelManager levelManager) takes a reference of LevelManager and creates the object, which handles files by reading and creating levels according to them, with the help of LevelManager

Methods
void createLevelFromFile(int frameWidth, int levelId) takes the frame width and the level id which will be played, in order to read from the file and create the level that will be played, by using LevelManager class

CollisionManager Class

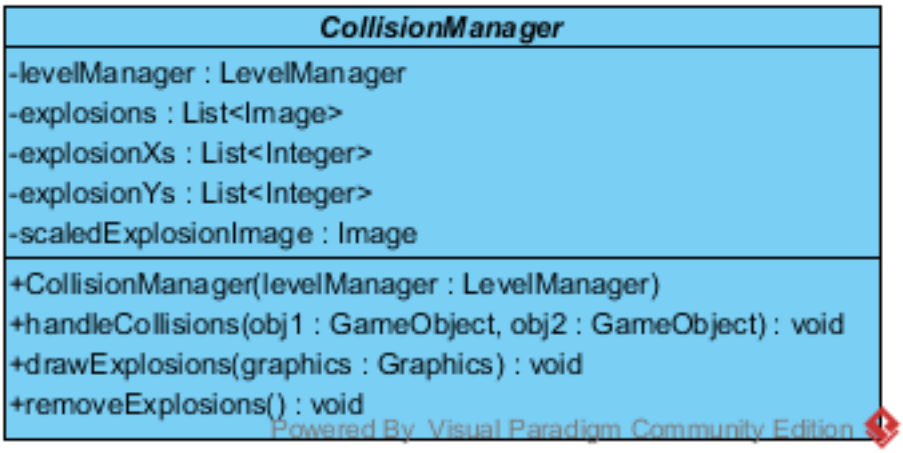

Figure 91 CollisionManager Class Diagram

CollisionManager is the abstract class for handling collisions.

Attributes
private LevelManager levelManager holds a reference to LevelManager which creates the CollisionManager
private List<Image> explosions holds the list of all explosion images
private List<Integer> explosionXs holds the x coordinates of explosionImages
private List<Integer> explosionYs holds the y coordinates of explosionImages
private Image scaledExplosionImage is the scaled image of the explosion

Constructor
public CollisionManager(LevelManager levelManager) is the constructor of CollisionManager. It takes a reference to LevelManager which creates the CollisionManager.

Methods
public void handleCollision(GameObject obj1, GameObject obj2) is the abstract method for handling a collision between two game objects
public void drawExplosions(Graphics graphics) draws all explosion images on screen
public void removeExplosions() removes all explosion images from screen

UserPlaneDestroyedCollisionManager Class

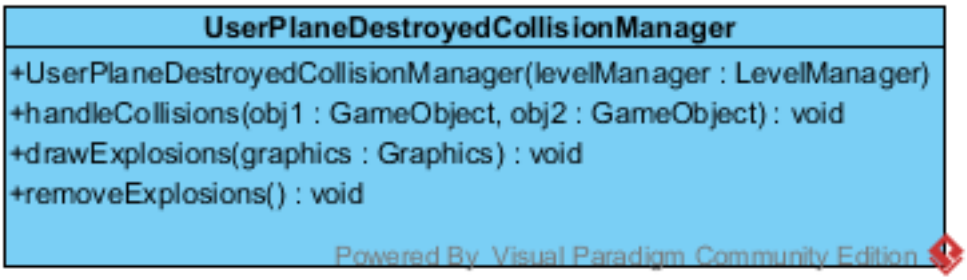

Figure 92 UserPlaneDestroyedCollisionManager Class Diagram

UserPlaneDestroyedCollisionManager is one of the concrete strategy classes. It handles collisions between user plane and targets-obstacles. The class extends CollisionManager class.

Constructor
public UserPlaneDestroyedCollisionManager(LevelManager levelManager) is the constructor of UserPlaneDestroyedCollisionManager. It takes a reference to LevelManager which creates the CollisionManager.

Methods
public void handleCollision(GameObject obj1, GameObject obj2) overrides the super method to handle the specific collision
public void drawExplosions(Graphics graphics) draws all explosion images on screen
public void removeExplosions() removes all explosion images from screen

UserPlaneHitCollisionManager Class

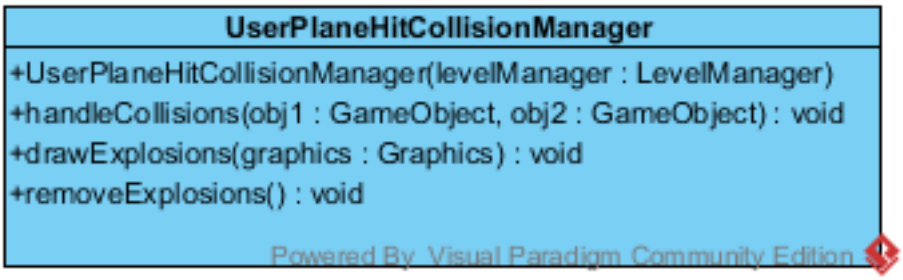

Figure 93 UserPlaneHitCollisionManager Class Diagram

UserPlaneHitCollisionManager is one of the concrete strategy classes. It handles collisions between user plane and enemy weapons. The class extends CollisionManager class.

Constructor
public UserPlaneHitCollisionManager(LevelManager levelManager) is the constructor of UserPlaneHitCollisionManager. It takes a reference to LevelManager which creates the CollisionManager.

Methods
public void handleCollision(GameObject obj1, GameObject obj2) overrides the super method to handle the specific collision
public void drawExplosions(Graphics graphics) draws all explosion images on screen
public void removeExplosions() removes all explosion images from screen

TargetHitCollisionManager Class

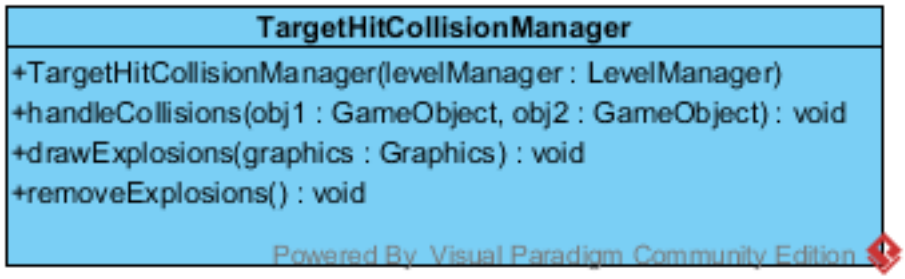

Figure 94 TargetHitCollisionManager Class Diagram

TargetHitCollisionManager is one of the concrete strategy classes. It handles collisions between user plane weapons and targets. The class extends CollisionManager class.

Constructor
public TargetHitCollisionManager(LevelManager levelManager) is the constructor of TargetHitCollisionManager. It takes a reference to LevelManager which creates the CollisionManager.

Methods
public void handleCollision(GameObject obj1, GameObject obj2) overrides the super method to handle the specific collision
public void drawExplosions(Graphics graphics) draws all explosion images on screen
public void removeExplosions() removes all explosion images from screen

BonusPackageCollisionManager

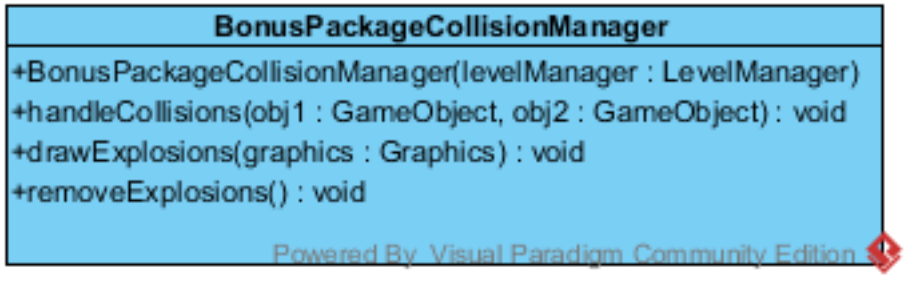

Figure 95 BonusPackageCollisionManager Class Diagram

BonusPackageCollisionManager is one of the concrete strategy classes. It handles collisions between user plane and bonus packages. The class extends CollisionManager class.

Constructor
public BonusPackageCollisionManager(LevelManager levelManager) is the constructor of BonusPackageCollisionManager. It takes a reference to LevelManager which creates the CollisionManager.

Methods
public void handleCollision(GameObject obj1, GameObject obj2) overrides the super method to handle the specific collision
public void drawExplosions(Graphics graphics) draws all explosion images on screen
public void removeExplosions() removes all explosion images from screen

CollisionPolicy Class

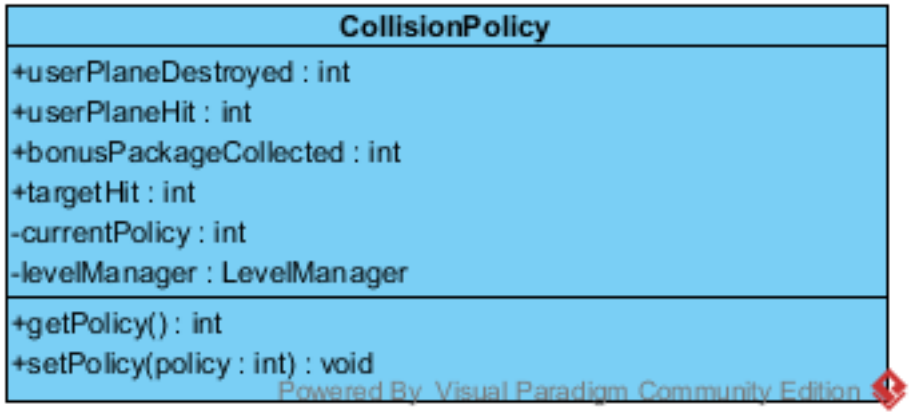

Figure 96 CollisionPolicy Class Diagram

CollisionPolicy class is the policy class of strategy pattern which decides on appropriate collision manager for a specific collision.

Enums
public int userPlaneDestroyed is the policy id for lethal user plane collisions
public int userPlaneHit is the policy id for user plane and weapon collisions
public int bonusPackageCollected is the policy id for user plane and bonus package collisions
public int targetHit is the policy id for user weapon and target collisions

Attributes
private int currentPolicy keeps the id of policy of the current collision
private LevelManager levelManager holds a reference to LevelManager in order to handle collisions

Methods
public int getPolicy() returns the current collision policy
public void setPolicy(int policy) updates the current policy

5.2.3. Game Model Subsystem Class Interfaces

Level Class

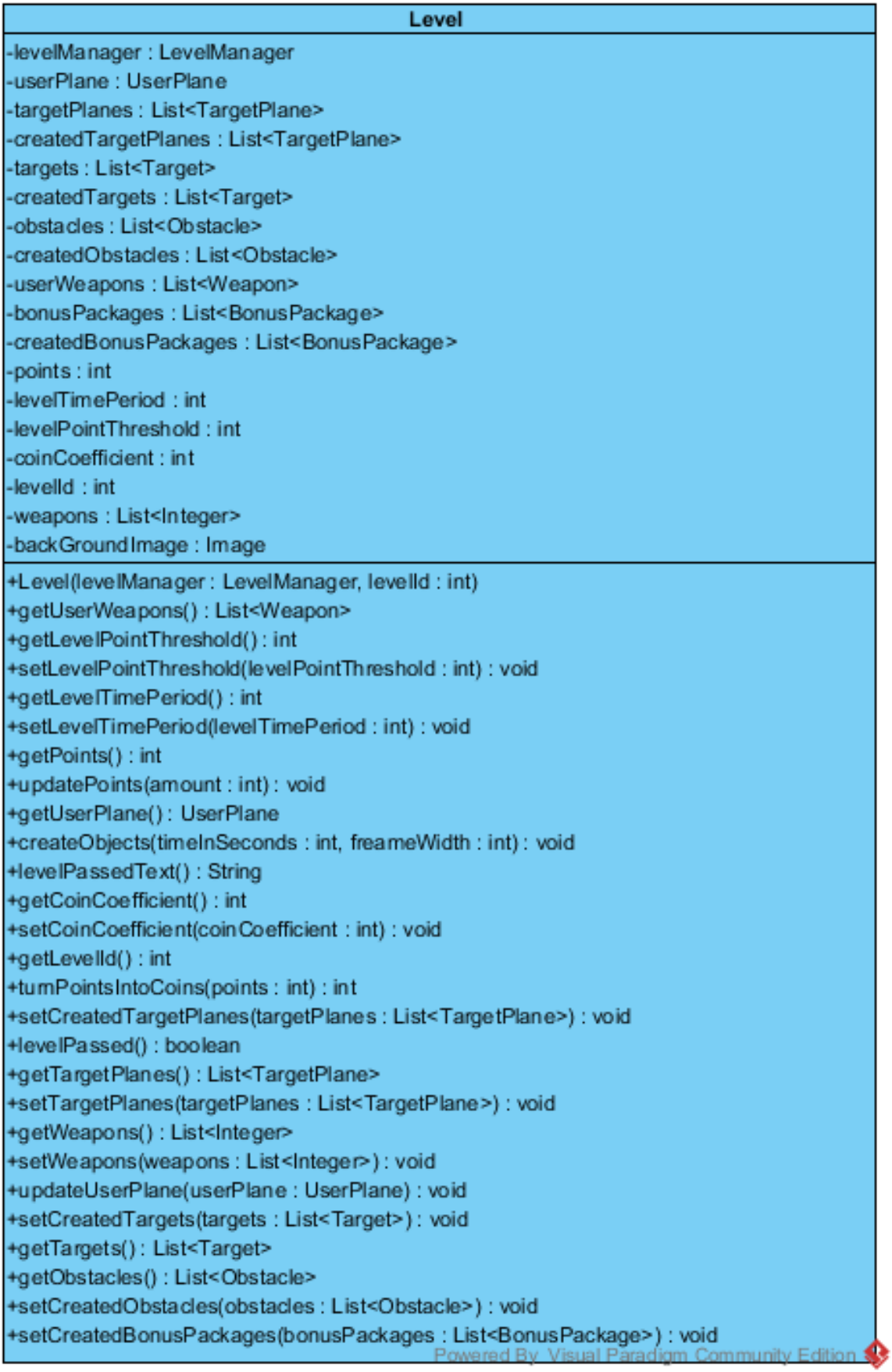
Level
-levelManager : LevelManager
-userPlane : UserPlane
-targetPlanes : List<TargetPlane>
-createdTargetPlanes : List<TargetPlane>
-targets : List<Target>
-createdTargets : List<Target>
-obstacles : List<Obstacle>
-createdObstacles : List<Obstacle>
-userWeapons : List<Weapon>
-bonusPackages : List<BonusPackage>
-createdBonusPackages : List<BonusPackage>
-points : int
-levelTimePeriod : int
-levelPointThreshold : int
-coinCoefficient : int
-levelId : int
-weapons : List<Integer>
-backGroundImage : Image
+Level(levelManager : LevelManager, levelId : int)
+getUserWeapons() : List<Weapon>
+getLevelPointThreshold() : int
+setLevelPointThreshold(levelPointThreshold : int) : void
+getLevelTimePeriod() : int
+setLevelTimePeriod(levelTimePeriod : int) : void
+getPoints() : int
+updatePoints(amount : int) : void
+getUserPlane() : UserPlane
+createObjects(timeInSeconds : int, freameWidth : int) : void
+levelPassedText() : String
+getCoinCoefficient() : int
+setCoinCoefficient(coinCoefficient : int) : void
+getLevelId() : int
+turnPointsIntoCoins(points : int) : int
+setCreatedTargetPlanes(targetPlanes : List<TargetPlane>) : void
+levelPassed() : boolean
+getTargetPlanes() : List<TargetPlane>
+setTargetPlanes(targetPlanes : List<TargetPlane>) : void
+getWeapons() : List<Integer>
+setWeapons(weapons : List<Integer>) : void
+updateUserPlane(userPlane : UserPlane) : void
+setCreatedTargets(targets : List<Target>) : void
+getTargets() : List<Target>
+getObstacles() : List<Obstacle>
+setCreatedObstacles(obstacles : List<Obstacle>) : void
+setCreatedBonusPackages(bonusPackages : List<BonusPackage>) : void
Powered By Visual Paradigm Community Edition

Figure 97 Level Class Diagram

Level class is the Facade class of Game Model subsystem. Game Control subsystem accesses Game Model entities from level class, that is, Level class provides an interface for Game Model subsystem. Level class corresponds to levels in the game and it contains all game objects.

Attributes
private LevelManager levelManager holds a reference to LevelManager class. LevelManager contacts with Level to take game objects from level and handle level operations.
private UserPlane UserPlane is the UserPlane user is controlling in level
private List<TargetPlane> targetPlanes is the list of target planes that appear in level
private List<TargetPlane> createdTargetPlanes is the list of target planes that are read from the file via FileManager
private List<Target> targets is the list of targets that appear in level
private List<Target> createdTargets is the list of targets that are read from the file via FileManager
private List<Obstacle> obstacles is the list of obstacles that appear in level
private List<Obstacle> createdObstacles is the list of obstacles that are read from the file via FileManager
private List<Weapon> userWeapons holds the list of weapons user has fired
private List<BonusPackage> bonusPackages is the list of bonus packages that appear in level
private List<BonusPackage> createdBonusPackages is the list of bonus packages that are read from file via FileManager
private int points is the total points collected in level
private int levelTimePeriod is the total seconds allocated for a specific level
private int levelPointThreshold is the amount of points user has to collect in order to pass a specific level
private int coinCoefficient specifies how many points correspond to 1 coin in a specific level
private int levelId specifies the id of the level
private List<Integer> weapons specifies the number of each Weapon user has as a list
private Image baskgroundImage is the background image of a specific level

Constructor
Level(LevelManager levelManager, int levelId) is the constructor of Level object. It takes a reference to LevelManager which creates the level and it takes the id of level.

Methods
public List<Weapon> getUserWeapons() returns the list of user weapons in level
public int getLevelPointThreshold() returns level point threshold
public void setLevelPointThreshold(int levelPointThreshold) sets the level point threshold to a new value
public int getLevelTimePeriod() returns amount of seconds allocated to level
public void setLevelTimePeriod(int levelTimePeriod) sets the time allocated for level to a new value
public int getPoints() returns amount of collected points in total
public void updatePoints(int amount) increments or decrements the total points of level according to new point amount
public UserPlane getUserPlane() returns user plane in the game
public void createObjects(int timeInSeconds, int frameWidth) creates all targets, obstacles and bonus packages if the current level time matches the appear time of objects. The frameWidth is used to place objects to end of the screen.
public String levelPassedText() returns a String which specifies whether level is passed or not and if

failed, why the player has failed
public int getCoinCoefficient() returns the coin coefficient of level
public void setCoinCoefficient(int coinCoefficient) sets the coin coefficient to new value
public int getLevelId() returns the id of the level
public int turnPointsIntoCoins(int points) takes the points collected and turns these points to coins according to coin coefficient and returns the coin amount
public void setCreatedTargetPlanes(List<TargetPlane> targetPlanes) sets created target planes to new list of target planes read from file
public boolean levelPassed() returns whether level is passed or not
public List<TargetPlane> getTargetPlanes() returns the list of target planes in level
public void setTargetPlanes(List<TargetPlane> targetPlanes) updates the list of target planes with new list taken from level manager
public List<Integer> getWeapons() returns amounts of left user weapons
public void setWeapons(List<Integer> weapons) updates the amounts of weapons
public void updateUserPlane(UserPlane userPlane) updates user plane object
public void setCreatedTargets(List<Target> targets) sets created target to new list of targets read from file
public List<Target> getTargets() returns the list of targets in level
public List<Obstacle> getObstacles() returns the list of obstacles in level
public void setCreatedObstacles(List<Obstacle> obstacles) sets created obstacles to new list of obstacles read from file
public void setCreatedBonusPackages(List<BonusPackage> bonusPackages) sets created bonus packages to new list of bonus packages read from file.

GameObject Class

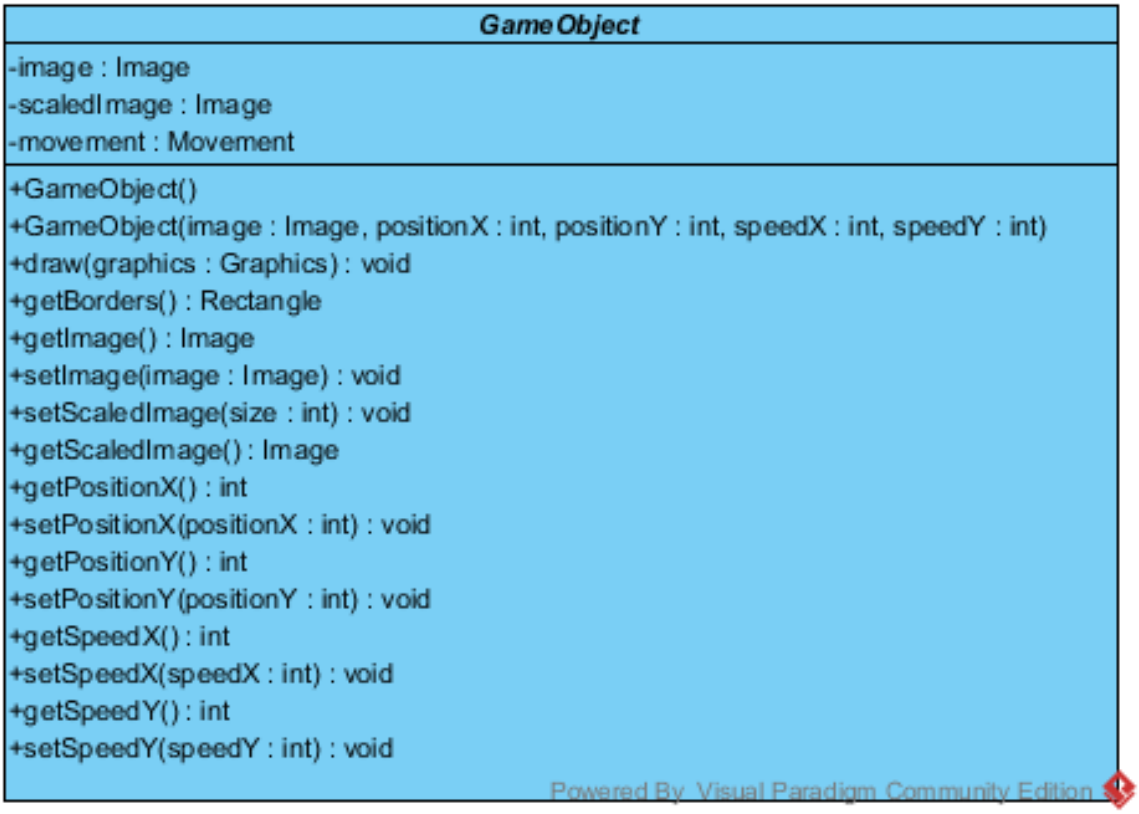

Figure 98 GameObject Class Diagram

GameObject class is an abstract class which provides an abstraction for all objects in game.

Attributes
private Image image specifies the image of the object
private Image scaledImage specifies scaled image of the object, the scaled form of image
private Movement movement holds a reference to movement contained within GameObject class

Constructors
public GameObject() is an empty constructor.
public GameObject(Image imageName, int positionX, int positionY, int speedX, int speedY) is the constructor of game object. It initializes the image, positions in x and y coordinates and speeds in x and y coordinates of the game object.

Methods
public abstract void draw(Graphics graphics) is the abstract method for drawing the object on panel
public abstract Rectangle getBorders() is the abstract method for getting the borders of scaled images of game objects
public Image getImage() returns image of the game object
public void setImage(Image image) changes the image of the game object
public Image getScaledImage() returns the scaled image of the game object
public void setScaledImage(int size) sets the scaled image by scaling the image to given size
public int getPositionX() returns the coordinate of the game object in x axis
public void setPositionX(int position) changes the coordinate of the game object in x axis
public int getPositionY() returns the coordinate of the game object in y axis
public void setPositionY(int position) changes the coordinate of the game object in y axis
public int getSpeedX() returns the speed of the game object in x axis
public void setSpeedX(int speedX) changes the speed of the game object in x axis
public int getSpeedY() returns the speed of the game object in y axis
public void setSpeedY(int speedY) changes the speed of the game object in y axis

Movement Class

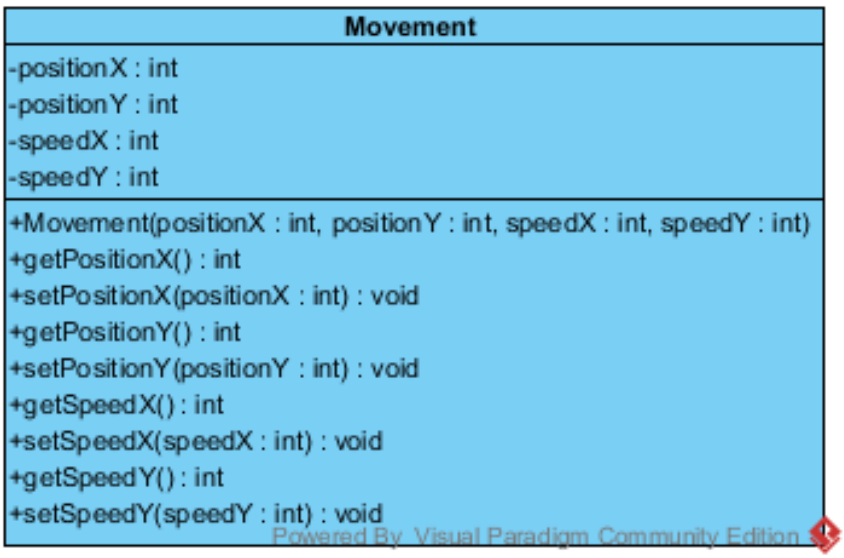

Figure 99 Movement Class Diagram

Movement class specifies the motional properties of objects, their locations and speeds.

Attributes
private int positionX specifies the location of the object on x axis
private int positionY specifies the location of the object on y axis
private int speedX specifies the speed of the object on x axis
private int speedY specifies the speed of the object on y axis

Constructor
public Movement(int positionX, int positionY, int speedX, int speedY) is the constructor of movement. It initializes the positions in x and y coordinates and speeds in x and y coordinates of the movement object.

Methods
public int getPositionX() returns the coordinate of the object in x axis
public void setPositionX(int positionX) changes the coordinate of the object in x axis
public int getPositionY() returns the coordinate of the object in y axis
public void setPositionY(int positionY) changes the coordinate of the object in y axis
public int getSpeedX() returns the speed of the object in x axis
public void setSpeedX(int speedX) changes the speed of the object in x axis
public int getSpeedY() returns the speed of the object in y axis
public void setSpeedY(int speedY) changes the speed of the object in y axis

UserPlane Class

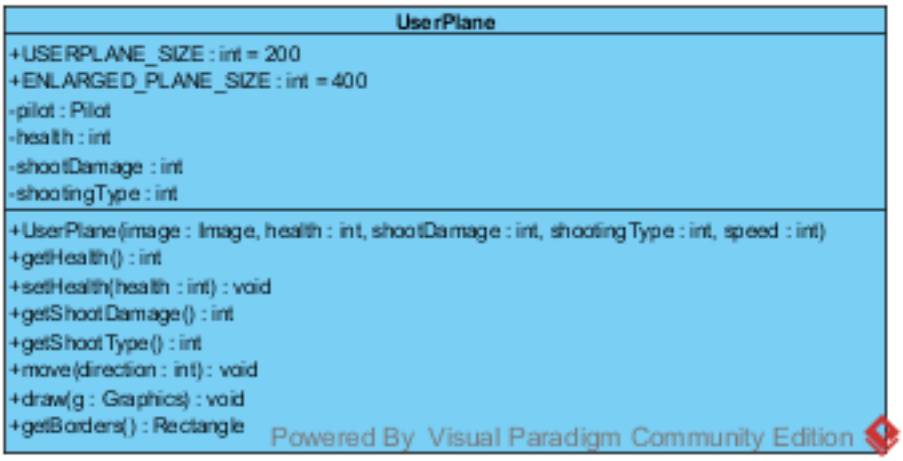

Figure 100 UserPlane Class Diagram

UserPlane class represents the plane user is controlling. It extends GameObject abstract class.

Constants
public static final int USERPLANE_SIZE is the size of the user plane
public static final int ENLARGED_PLANE_SIZE is the size of the user plane when it is enlarged

Attributes
private Pilot pilot is the pilot of the user plane
private int health is the health amount of the plane
private int shootDamage is the damage the plane gives to targets

private int shootingType is the shooting type of the plane, whether it shoots one weapon at a time or two

Constructor
public UserPlane(Image image, int health, int shootDamage, int shootType, int speed) is the constructor of UserPlane. It initializes the position of the plane, the image, the shooting type, the health and also the speed.

Methods
public int getHealth() returns the health
public void setHealth(int health) updates the health
public int getShootDamage() returns the shootDamage
public int getShootType() returns the shoot type
public void move(int direction): moves the user plane in the specifies direction
public void draw(Graphics g) overwrites the parent method and draws the user plane image on screen
public Rectangle getBorders() returns borders of the user plane image

UserPlaneEnum Enumeration

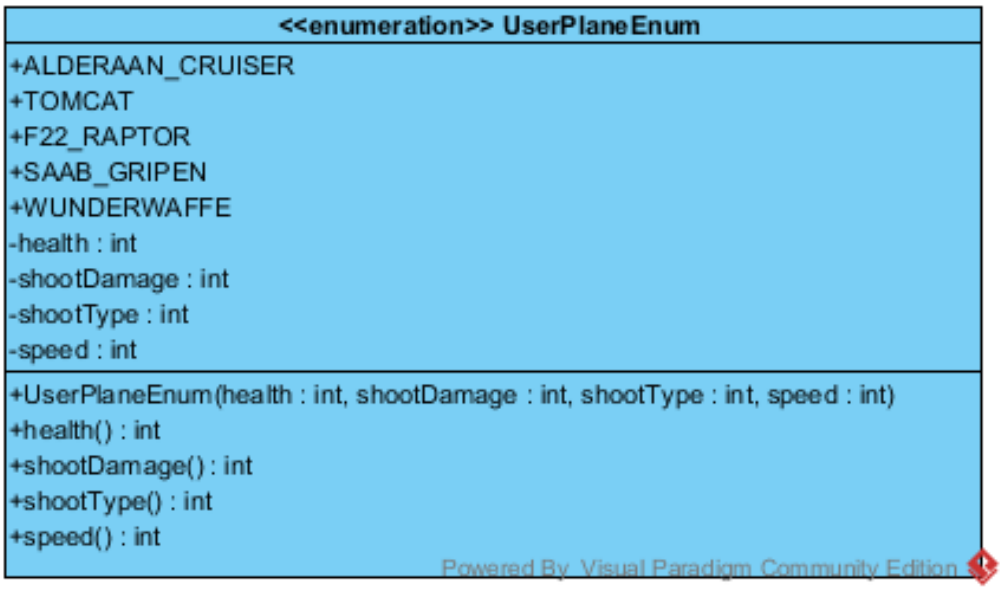

Figure 101 UserPlaneEnum Class Diagram

UserPlaneEnum class represents the enumeration of UserPlane. It specifies the values of specific instances of the class.

Enums
ALDERAAN_CRUISER represents values for Alderaan Cruiser user planes
TOMCAT represents values for Tomcat user planes
F22_RAPTOR represents values for F22Raptor user planes
SSAB_GRIPEN represents values for Saab Gripen user planes
WUNDERWAFFE represents values for Wunderwaffe user planes

Attributes
private int health is the health amount of the plane

private int shootDamage is the damage the plane gives to targets
private int shootType is the shooting type of the plane, whether it shoots one weapon at a time or two
private int speed is the speed amount of the plane

Constructor
public UserPlaneEnum(int health, int shootDamage, int shootType, int speed) is the constructor of UserPlaneEnum. It initializes the shooting type, shooting damage, health and the speed of enumeration.

Methods
public int health() returns the health
public int shootDamage() returns the shoot damage
public int shootType() returns the shoot type
public int speed() returns the speed

Pilot Class

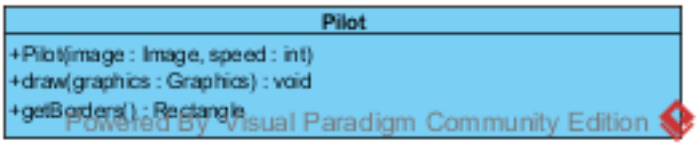

Figure 102 Pilot Class Diagram

Pilot class represents the character figure that controls the user plane. Pilot class extends abstract GameObject class.

Constructor
public Pilot(Image image, int speed) is the constructor of pilot. It takes and initializes the image and speed of pilot.

Methods
public void draw(Graphics graphics) overwrites the parent method and draws the pilot image on screen
public Rectangle getBorders() returns borders of the pilot image

PilotEnum Enumeration

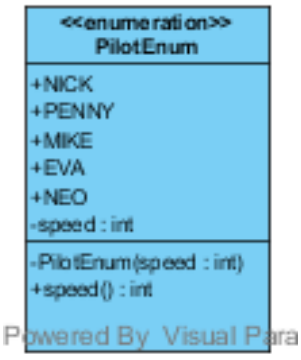

Figure 103 PilotEnum Class Diagram

PilotEnum class represents the enumeration of Pilot. It specifies the values of specific instances of the class.

Enums

NICK represents values for Nick pilots
PENNY represents values for Penny pilots
MIKE represents values for Mike pilots
EVA represents values for Eva pilots
NEO represents values for Neo pilots

Attributes

private int speed is the speed amount of the pilot, his capability of increasing the speed of the plane it drives

Constructor

public PilotEnum(int speed) is the constructor of PilotEnum. It initializes the speed of enumeration.

Methods

public int speed() returns the speed

Obstacle Class

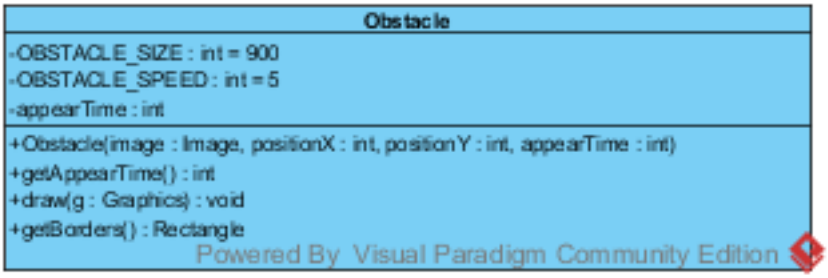

Figure 104 Obstacle Class Diagram

Obstacle class represents obstacles appear in level that kills user when collided

Constants

private final int OBSTACLE_SIZE holds default size of obstacle
private static final int OBSTACLE_SPEED holds default speed of obstacle

Attributes

private int appearTime holds appear time of obstacle

Constructor

public Obstacle(Image image, int positionX, int position, int appearTime) is the constructor for obstacle. It initializes the image, appear time and the positions of obstacle.

Methods

public int getAppearTime() returns appear time
public void draw(Graphics g) draws the scaled image on screen
public Rectangle getBorders() returns borders of the scaled image

Weapon Class

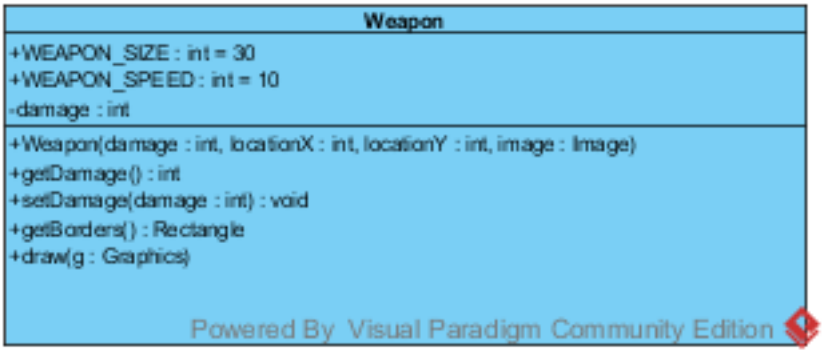

Figure 105 Weapon Class Diagram

Weapon class represents shoots and explosives that can be fired from planes. Weapon class extends GameObject class.

Constants
public static final int WEAPON_SIZE holds default size of weapon
public static final int WEAPON_SPEED holds default speed of weapon

Attributes
private int damage is the damage amount, how much weapon decreases health of an object

Constructor
public Weapon(int damage, int locationX, int locationY) is the constructor of Weapon. It initializes damage and coordinates of weapon.

Methods
public int getDamage() returns damage of weapon
public void setDamage(int damage) changes damage of weapon
public Rectangle getBorders() returns borders of the weapon image
public void draw(Graphics g) draws weapon image on screen

Shoot Class

Figure 106 Shoot Class Diagram

Shoot class represents weapons that can only damage the collided object. Shoot extends Weapon class.

Constructor
public Shoot(int damage, int locationX, int locationY, Image image) is the constructor of Shoot. It initializes the damage, image and coordinates of shoot.

ShootEnum Enumeration

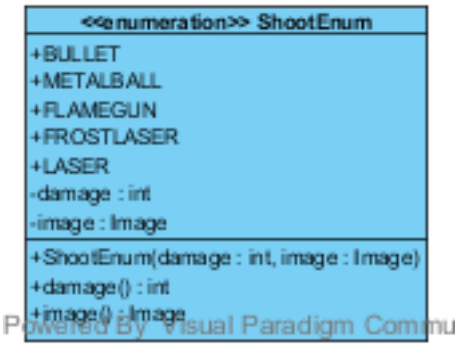

Figure 107 ShootEnum Class Diagram

ShootEnum class represents the enumeration of Shoot. It specifies the values of specific instances of the class.

Enums
BULLET represents values for Bullet shoots
METALBALL represents values for Metal Ball shoots
FLAMEGUN represents values for Flame Gun shoots
FROSTLASER represents values for Frost Laser shoots
LASER represents values for Laser shoots

Attributes
private int damage is the damage amount of the shoot
private Image image is the image of the shoot

Constructor
public ShootEnum(int damage, Image image) is the constructor of ShootEnum. It initializes the damage and image of enumeration.

Methods
public int damage() returns the damage
public Image image() returns the image

Explosive Class

Figure 108 Explosive Class Diagram

Explosive class represents weapons that give damage to object in its damage area. Explosive extends Weapon class.

Attributes
private int damageArea is the area of explosive damage

Constructor
public Explosive(int damage, int damageArea, int locationX, int locationY, Image image) is the

constructor of Explosive. It initializes the damage, damage area, image and coordinates of explosive.

Methods
public int getDamageArea() returns the damage area

ExplosiveEnum Enumeration

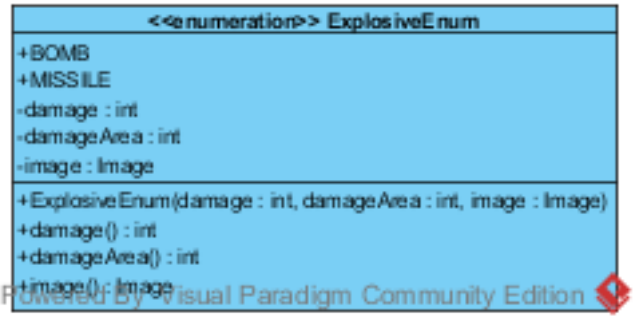

Figure 109 ExplosiveEnum Class Diagram

ExplosiveEnum class represents the enumeration of Explosive. It specifies the values of specific instances of the class.

Enums
BOMB represents values for Bomb explosives
MISSILE represents values for Missile explosives

Attributes
private int damage is the damage amount of the explosive
private int damageArea is the damage area of the explosive
private Image image is the image of the explosive

Constructor
public ExplosiveEnum(int damage, int damageArea, Image image) is the constructor of ExplosiveEnum. It initializes the damage, damage area and image of enumeration.

Methods
public int damage() returns the damage
public int damageArea() returns the damage area
public Image image() returns the image

Target Class

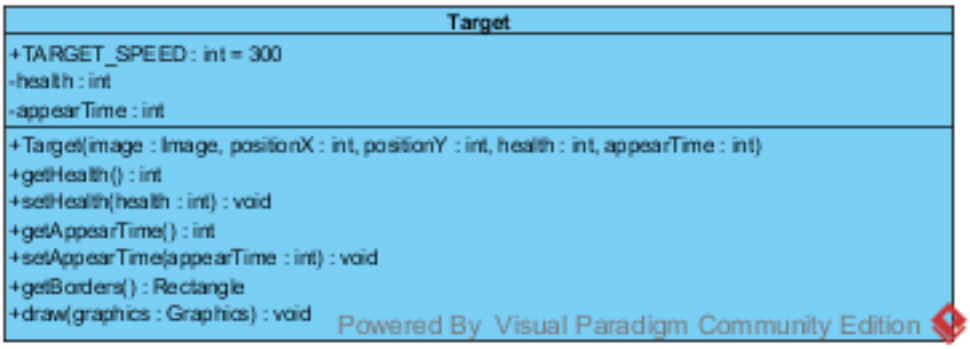

Figure 110 Target Class Diagram

Target class represents all shootable items appear in the game that has a health and appear time in level.

Constants
public static final int TARGET_SPEED is the pre-set speed of all targets

Attributes
private int health is the health of target.
private int appearTime is the time the target appears on game screen

Constructor
public Target(Image image, int positionX, int positionY, int health, int apppearTime) is the constructor of target. It initializes the image, coordinates, health and appear time of target.

Methods
public int getHealth() returns the health
public void setHealth(int health) changes health
public int getAppearTime() returns appear time
public void setAppearTime(int appearTime) changes appear time
public Rectangle getBorders() returns borders of the scaled image
public void draw(Graphics graphics) draws the scaled image on screen

Ally Class

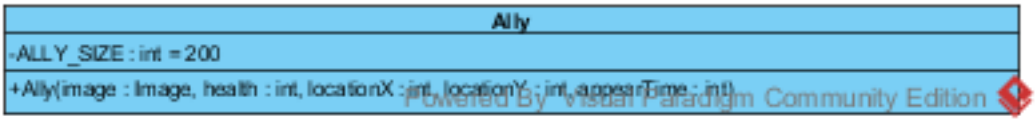

Figure 111 Ally Class Diagram

Ally class represents targets that are ally planes of the user, when shot they deduce user points. Ally class extends Target.

Constants
private static final int ALLY_SIZE is the pre-set size of all allies

Constructor
public Ally(Image image, int health, int locationX, int locationY, int apppearTime) is the constructor of ally. It initializes the image, coordinates, health and appear time of ally.

AllyEnum Enumeration

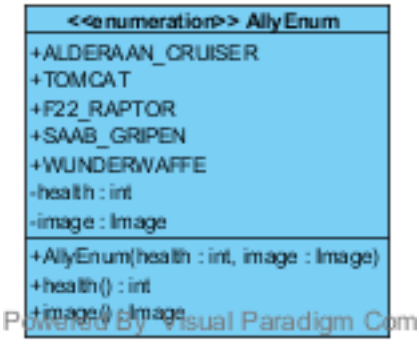

Figure 112 AllyEnum Class Diagram

AllyEnum class represents the enumeration of Ally. It specifies the values of specific instances of the class.

Enums
ALDERAAN_CRUISER represents values for Alderaan Cruiser allies
TOMCAT represents values for Tomcat allies
F22_RAPTOR represents values for F22 Raptor allies
SSAB_GRIPEN represents values for Saab Gripen allies
WUNDERWAFFE represents values for Wunderwaffe allies

Attributes
private int health is the health amount of the plane
privateImage image is the image of the ally plane

Constructor
public AllyEnum(int health, Image image) is the constructor of AllyEnum. It initializes the health and the image of enumeration.

Methods
public int health() returns the health
public Image image() returns the image

BonusMissionTarget Class

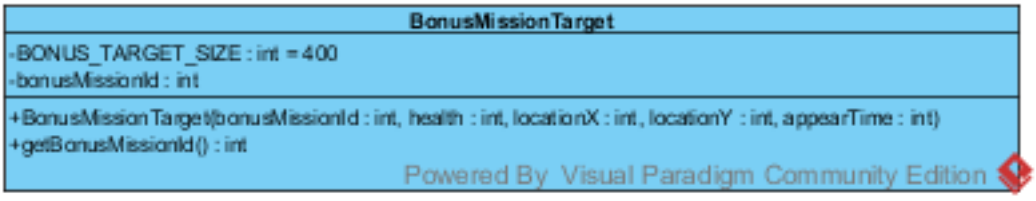

Figure 113 BonusMissionTarget Class Diagram

BonusMissionTarget class represents targets that when destroyed, gains user access to corresponding bonus mission. BonusMissionTarget extends Target class.

Constants
private static final int BONUS_TARGET_SIZE is the pre-set size of all bonus mission targets

Attributes
private int bonusMissionId holds the bonus mission id of the target, which bonus mission the target opens access to.

Constructor
public BonusMissionTarget(int bonusMissionId, int health, int locationX, int locationY, int apppearTime) is the constructor of bonus mission target. It initializes the bonus mission id, coordinates, health and appear time of bonus mission target.

Methods
public int getBonusMissionId() returns bonus mission id

BonusMissionTargetEnum Enumeration

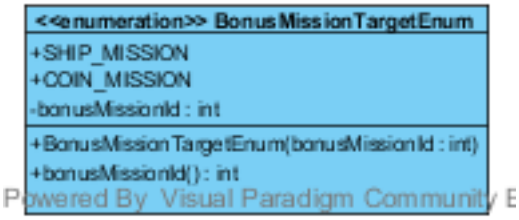

Figure 114 BonusMissionTargetEnum Class Diagram

BonusMissionTargetEnum class represents the enumeration of BonusMissionTarget. It specifies the values of specific instances of the class.

Enums
SHIP_MISSION represents values for Ship bonus mission targets
COIN_MISSION represents values for Coin bonus mission targets

Attributes
private int bonusMissionId is the id of the bonus mission

Constructor
public BonusMissionTargetEnum(int bonusMissionId) is the constructor of BonusMissionTargetEnum. It initializes bonus mission id of enumeration.

Methods
public int bonusMissionId() returns the bonus mission id

Rocket Class

Figure 115 Rocket Class Diagram

Rocket class represents targets that explodes and gives area damage when destroyed. Rocket extends Target class.

Constants
private static final int ROCKET_SIZE is the pre-set size of all rockets

Attributes
private int damage holds the damage of rocket
private int damageArea holds the damage area of rocket

Constructor
public Rocket(int damage, int damageArea, int health, int locationX, int locationY, int apppearTime) is the constructor of rocket. It initializes the damage and damage area, coordinates, health and appear time of rocket.

RocketEnum Enumeration

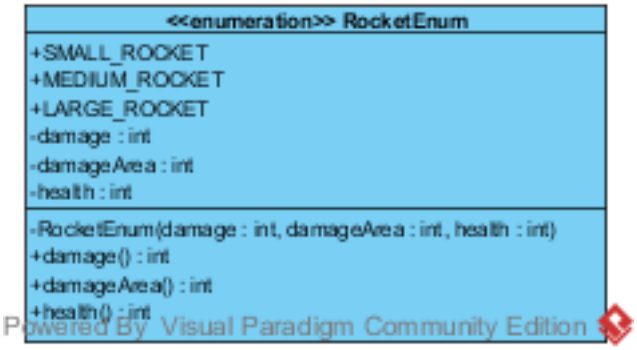

Figure 116 RocketEnum Class Diagram

RocketEnum class represents the enumeration of Rocket. It specifies the values of specific instances of the class.

Enums
SMALL_ROCKET represents values for rockets with small damage
MEDIUM_ROCKET represents values for rockets with medium damage
LARGE_ROCKET represents values for rockets with big damage

Attributes
private int damage holds the damage of rocket
private int damageArea holds the damage area of rocket

Constructor
public RocketEnum(int damage, int damageArea, int health) is the constructor of RocketEnum. It initializes the damage, damage area and health of the enumeration.

Methods
public int damage() returns the damage
public int damageArea() returns the damage area
public int health() returns the health

Carriage Class

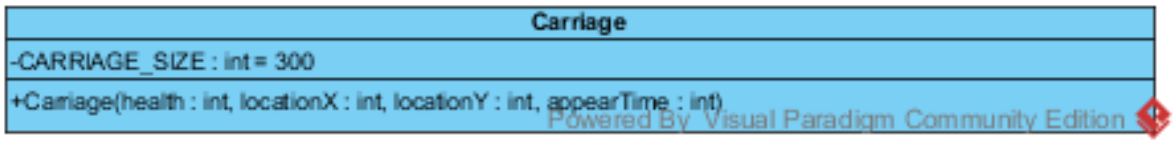

Figure 117 Carriage Class Diagram

Carriage class represents target planes that cannot shoot. Carriage extends Target class.

Constants
private static final int CARRIAGE_SIZE is the pre-set size of all carriages

Constructor
public Carriage(int health, int locationX, int locationY, int apppearTime) is the constructor of carriage. It initializes the coordinates, health and appear time of carriage.

CarriageEnum Enumeration

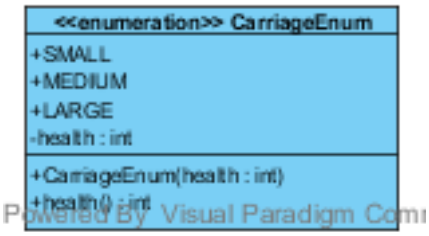

Figure 118 CarriageEnum Class Diagram

CarriageEnum class represents the enumeration of Carriage. It specifies the values of specific instances of the class.

Enums
SMALL represents values for carriages with small health
MEDIUM represents values for carriages with medium health
LARGE represents values for carriages with big health

Attributes
private int health holds the damage of carriage

Constructor
public CarriageEnum(int health) is the constructor of CarriageEnum. It initializes the health of the enumeration.

Methods
public int health() returns the health

Ship Class

Figure 119 Ship Class Diagram

Ship class represents ship targets appear in ship bonus mission. Ship extends Target class.

Constants
private static final int SHIP_SIZE is the pre-set size of all ships

Constructor
public Ship(int health, int locationX, int locationY, int apppearTime) is the constructor of ship. It initializes the coordinates, health and appear time of ship.

ShipEnum Enumeration

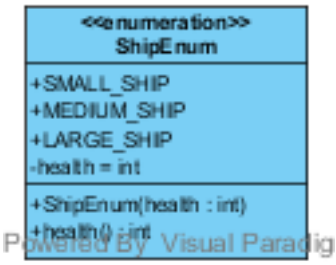

Figure 120 ShipEnum Class Diagram

ShipEnum class represents the enumeration of Ship. It specifies the values of specific instances of the class.

Enums
SMALL_SHIP represents values for ships with small health
MEDIUM_SHIP represents values for ships with medium health
LARGE_SHIP represents values for ships with big health

Attributes
private int health holds the damage of carriage

Constructor
public ShipEnum(int health) is the constructor of ShipEnum. It initializes the health of the enumeration.

Methods
public int health() returns the health

TargetPlane Class

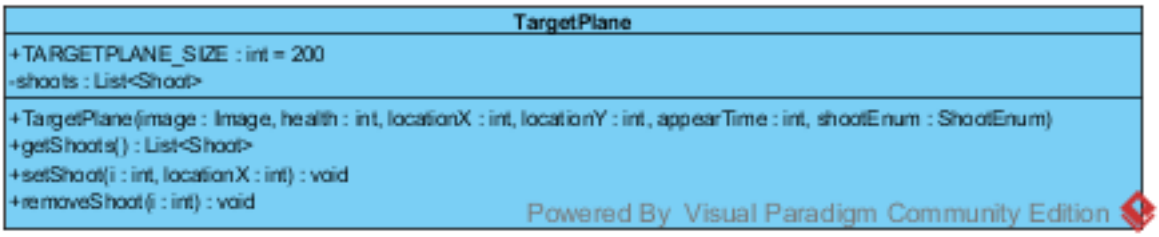

Figure 121 TargetPlane Class Diagram

TargetPlane class represents enemy planes in the level. TargetPlane extends Target class.

Constants
public static final int TARGET_PLANE_SIZE is the pre-set size of all target planes

Attributes
private List<Shoot> shoots is the list of target plane weapons

Constructor
public TargetPlane(Image image, int health, int positionX, int positionY, int apppearTime, ShootEnum shootEnum) is the constructor of target plane. It initializes the image, coordinates, health, appear time and specific Shoot enum of target plane.

Methods
public List<Shoot> getShoots() returns list of shoots
public void setShoot(int i, int location) sets the coordinates of a specific shoot in the shoot list
public void removeShoot(int i) removes a shoot from the shoot list

TargetPlaneEnum Enumeration

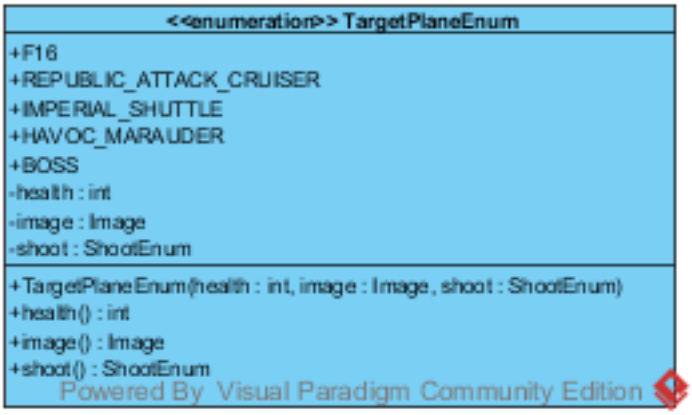

Figure 122 TargetPlaneEnum Class Diagram

TargetPlaneEnum class represents the enumeration of TargetPlane. It specifies the values of specific instances of the class.

Enums
public F16 represents values of F16 target planes
public REPUBLİC_ATTACK_CRUISER represents values of Republic Attack Cruiser target planes

public IMPERİAL_SHUTTLE represents values of Imperial Shuttle target planes
public HAVOC_MARAUDER represents values of Havoc Marauder target planes
public BOSS represents values of Boss target planes

Attributes
private int health is the health of target plane
private Image image is the image of target plane
private ShootEnum shoot keeps the specific instance of shoot of target plane

Constructor

public TargetPlaneEnum(int health, Image image, ShootEnum shootEnum) is the constructor of TargtePlaneEnum. It initializes the health, image and shoot enum of the enumeration.

Methods
public int health() returns health
public Image image() returns image
public ShootEnum shoot() returns shoot enumeration

BonusPackage Class

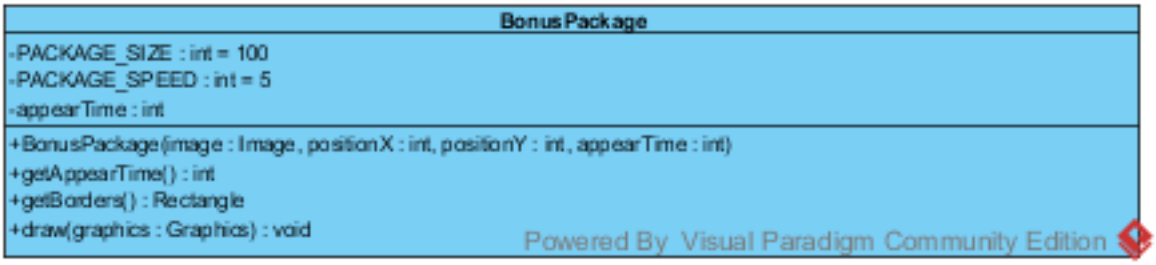

Figure 123 BonusPackage Class Diagram

BonusPackage class represents bonus packages appear in level, both trap packages and present packages. BonusPackage class extends GameObject class.

Constants
private static final int PACKAGE_SIZE the pre-set size of all bonus packages
private static final int PACKAGE_SPEED is the pre-set speed of all bonus package

Attributes
private int appearTime is the time the bonus package appears on game screen

Constructor
public BonusPackage(Image image, int positionX, int positionY, int apppearTime) is the constructor of bonus package. It initializes the image, coordinates and appear time of bonus package.

Methods
public int getAppearTime() returns appear time
public Rectangle getBorders() returns borders of the scaled image
public void draw(Graphics graphics) draws the scaled image on screen

PresentBonusPackage Class

Figure 124 PresentBonusPackage Class Diagram

PresentBonusPackage class represents bonus packages that boost game. PresentBonusPackage class extends BonusPackage class.

Constructor
public PresentBonusPackage(int positionX, int positionY, int apppearTime) is the constructor of present bonus package. It initializes the coordinates and appear time of bonus package.

SpeedBonusPackage Class

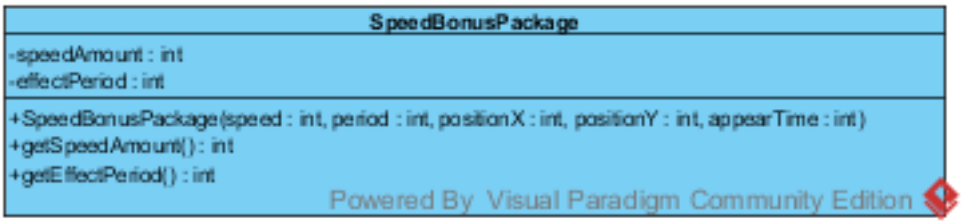

Figure 125 SpeedBonusPackage Class Diagram

SpeedBonusPackage class represents present bonus packages that increase user plane speed for a certain time period. SpeedBonusPackage class extends PresentBonusPackage class.

Attributes
private int speedAmount is the speed amount package adds to user plane
private int effectPeriod is the time amount package is active

Constructor
public SpeedBonusPackage(int speed, int period, int positionX, int positionY, int apppearTime) is the constructor of speed bonus package. It initializes the speed, effect period, coordinates and appear time of speed bonus package.

Methods
public int getSpeedAmount() returns speed amount of package
public int getEffectPeriod() returns effect period of package

DamageBonusPackage Class

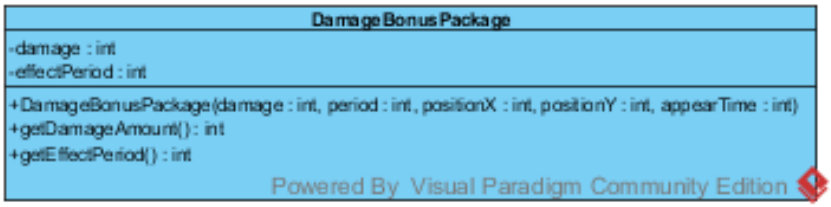

Figure 126 DamageBonusPackage Class Diagram

DamageBonusPackage class represents present bonus packages that increase user plane shoot damage for a certain time period. DamageBonusPackage class extends PresentBonusPackage class.

Attributes
private int damage is the damage amount package adds to user plane
private int effectPeriod is the time amount package is active

Constructor
public DamageBonusPackage(int damage int period, int positionX, int positionY, int apppearTime) is the constructor of damage bonus package. It initializes the damage, effect period, coordinates and appear time of speed bonus package.

Methods
public int getDamageAmount() returns damage amount of package
public int getEffectPeriod() returns effect period of package

ObstacleInvisibilityPackage Class

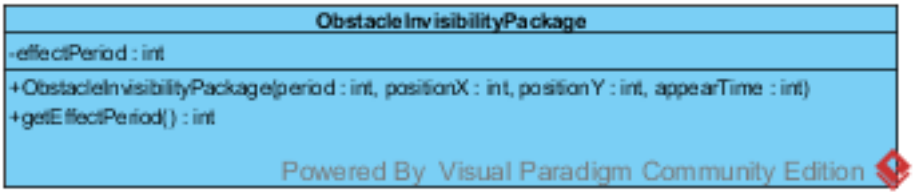

Figure 127 ObstacleInvisibilityPackage Class Diagram

ObstacleInvisibilityBonusPackage class represents present bonus packages that protects user from obstacles for a certain time period. ObstacleInvisibilityBonusPackage class extends PresentBonusPackage class.

Attributes
private int effectPeriod is the time amount package is active

Constructor
public ObstacleInvisibilityBonusPackage(int period, int positionX, int positionY, int apppearTime) is the constructor of obstacle invisibility bonus package. It initializes the effect period, coordinates and appear time of bonus package.

Methods
public int getEffectPeriod() returns effect period of package

InvincibilityPackage Class

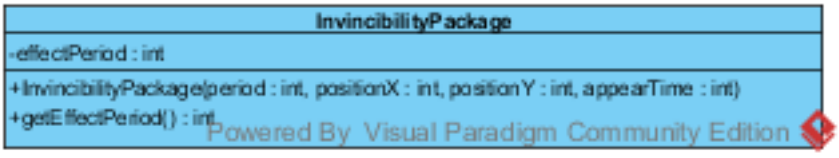

Figure 128 InvincibilityPackage Class Diagram

InvincibilityBonusPackage class represents present bonus packages that protects user from all dangers for a certain time period. ObstacleInvisibilityBonusPackage class extends PresentBonusPackage class.

Attributes
private int effectPeriod is the time amount package is active

Constructor
InvincibilityBonusPackage(int period, int positionX, int positionY, int apppearTime) is the constructor of invincibility bonus package. It initializes the effect period, coordinates and appear time of bonus package.

Methods
public int getEffectPeriod() returns effect period of package

CoinPackage Class

Figure 129 CoinPackage Class Diagram

CoinPackage class represents present bonus packages that increase user coins. CoinPackage class extends PresentBonusPackage class.

Attributes
private int coinAmount is the coin amount package adds to user collection

Constructor
public CoinPackage(int coin, int period, int positionX, int position, int appearTime) is the constructor of coin package. It initializes the coin amount, coordinates and appear time of bonus package.

Methods
public int getCoinAmount() returns coin amount of package

TimeBonusPackage Class

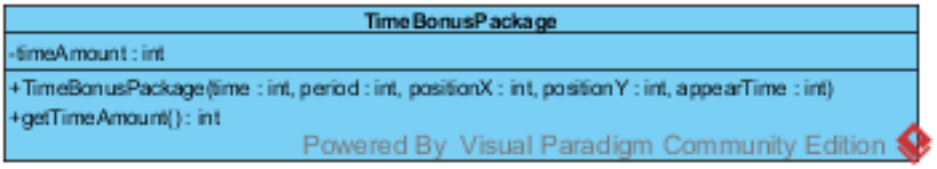

Figure 130 TimeBonusPackage Class Diagram

TimeBonusPackage class represents present bonus packages that increase time in level. TimeBonusPackage class extends PresentBonusPackage class.

Attributes
private int timeAmount is the time amount package adds to total level time

Constructor
public TimeBonusPackage(int time, int positionX, int positionY, int apppearTime) is the constructor of time package. It initializes the time amount, coordinates and appear time of bonus package.

Methods
public int getTimeAmount() returns time amount of package

HealthBonusPackage Class

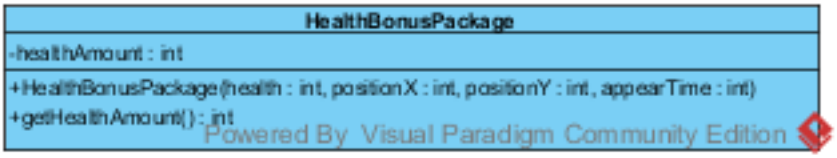

Figure 131 HealthBonusPackage Class Diagram

HealthBonusPackage class represents present bonus packages that increase health amount of user plane. HealthBonusPackage class extends PresentBonusPackage class.

Attributes
private int healthAmount is the health amount package adds to user plane

Constructor
public HealthBonusPackage(int health, int positionX, int positionY, int apppearTime) is the constructor of health bonus package. It initializes the health amount, coordinates and appear time of bonus package.

Methods
public int getHealthAmount() returns health amount of package

TrapBonusPackage Class

Figure 132 TrapBonusPackage Class Diagram

TrapBonusPackage class represents bonus packages that hardens the game. TrapBonusPackage class extends BonusPackage class.

Constructor
public TrapBonusPackage(int positionX, int positionY, int apppearTime) is the constructor of trap bonus package. It initializes the coordinates and appear time of bonus package.

DamageTrapPackage Class

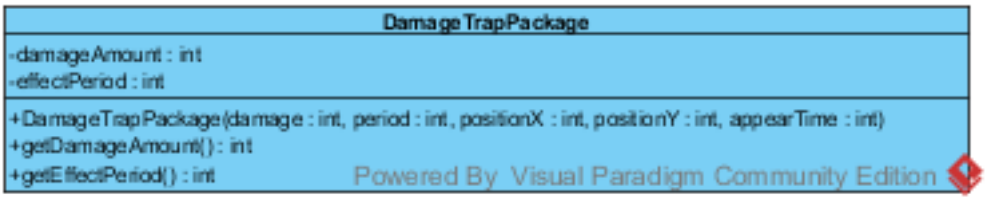

Figure 133 DamageTrapPackage Class Diagram

DamageTrapPackage class represents trap bonus packages that decrease user plane shoot damage for a certain time period. DamageTrapPackage class extends TrapBonusPackage class.

Attributes
private int damageAmount is the damage amount package subtracts form user plane
private int effectPeriod is the time amount package is active

Constructor
public DamageTrapPackage(int damage, int period, int positionX, int positionY, int apppearTime) is the constructor of damage trap package. It initializes the damage, effect period, coordinates and appear time of speed trap package.

Methods
public int getDamageAmount() returns damage amount of package
public int getEffectPeriod() returns effect period of package

SpeedTrapPackage Class

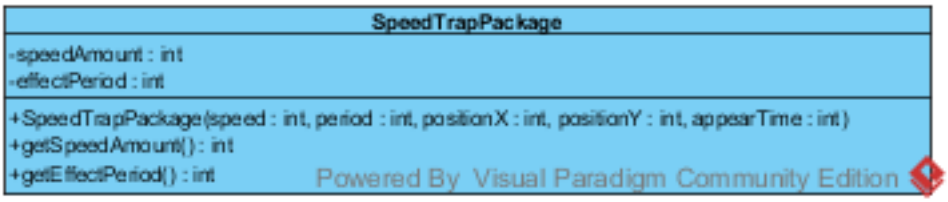

Figure 134 SpeedTrapPackage Class Diagram

SpeedTrapPackage class represents trap bonus packages that decrease user plane speed for a certain time period. SpeedTrapPackage class extends TrapBonusPackage class.

Attributes
private int speedAmount is the speed amount package subtracts from user plane
private int effectPeriod is the time amount package is active

Constructor
public SpeedTargetPackage(int speed, int period, int positionX, int positionY, int apppearTime) is the constructor of speed trap package. It initializes the speed, effect period, coordinates and appear time of speed trap package.

Methods
public int getSpeedAmount() returns speed amount of package
public int getEffectPeriod() returns effect period of package

HealthTrapPackage Class

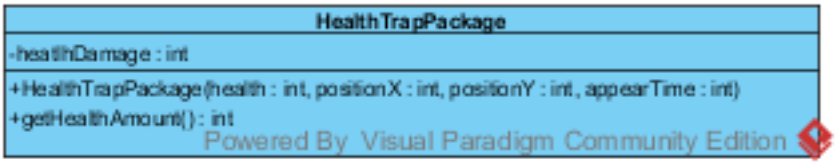

Figure 135 HealthTrapPackage Class Diagram

HealthTrapPackage class represents trap bonus packages that decreases health amount of user plane. HealthTrapPackage class extends TrapBonusPackage class.

Attributes
private int healthAmount is the health amount package subtracts from user plane

Constructor
public HealthTrapPackage(int health, int positionX, int positionY, int apppearTime) is the constructor of health trap package. It initializes the health amount, coordinates and appear time of bonus package.

Methods
public int getHealthAmount() returns health amount of package

PlaneEnlargePackage Class

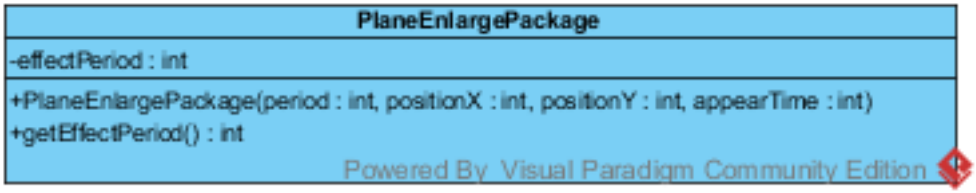

Figure 136 PlaneEnlargePackage Class Diagram

PlaneEnlargePackage class represents trap bonus packages that increases user plane size for a certain period of time. PlaneEnlargePackage class extends TrapBonusPackage class.

Attributes
private int effectPeriod is the time amount package is active

Constructor
public PlaneEnlargePackage(int period, int positionX, int positionY, int apppearTime) is the constructor of plane enlarge package. It initializes the time period, coordinates and appear time of bonus package.

Methods
public int getEffectPeriodt() returns effect period of package

5.3. Specifying Contracts

- **context LevelPanel inv:**
  timeInSeconds <= gameManager.levelManager.level.levelTimePeriod

The time count in LevelPanel should never exceed the total time allocated for the specific level.

- **context LevelPanel::resumeGame pre:**
  isGamePaused = true

In order to call resumeGame() method of LevelPanel the game should be paused currently. It only makes sense to resume game if it is paused.

- **context LevelPanel::initializeGame(i) post:**
  gameManager.levelManager.currentLevelId = i

When the game is initialized the currentLevelId in LevelManager should be set to the id of current level. According to level number player is requesting to play, the currentLevelId should be updated.

- **context LevelPanel::initializeGame(i) post:**
  isGamePaused = false
  isGameOver = false

When the game is initialized, isGameOver and isGamePaused parameters should be set to false.

- **context GameManager inv:**
  not passedLevelIds->contains(currentLevelId)

The list of all passedLevelIds should not contain the currentLevelId. A level should only be put to passedLevelIds after it is passed and the currentLevelId is then incremented.

- **context GameManager inv:**
  musicOn init: true

When the GameManager is created the value of musicOn attribute should be set to true. Hence when we start the game, the sound should be on by default.

- **context GameManager::purchaseItem(i, p) post:**
  collectionManager.coins = @pre.collectionManager.coins – p

After the user purchases an item, the user coins should be deduced by the price amount of the purchased item.

- **context GameManager::gameOver() post:**
  observers.levelPanel.isGameOver = true

After the game is over, time has ended or health is depleted, the isGameOver property of LevelPanel should indicate that the game is over. Therefore, it should update the level for game over scenario.

- **context GameObjectInterface:: subscribe(o) post:**
  observers->contains(o)

After a GameManagerObserver subscribes to a GameObjectInterface, the observer should be added to list of observers of the subject. For instance, when a panel subscribes to GameManager,

GameManager should add that panel to its list of observers.

- **context SoundManager::playSound(i) pre:**
  gameManager.musicOn = true

In order to play a sound, the music should be on. If the user has turned off the music, SoundManager should not play any sounds.

- **context LevelManager::shoot(g) post:**
  weapons->select(w:Weapon | w.index = currentLevelId) = @pre.weapons->select(w:Weapon | w.index = currentLevelId) – 1

After the user shoots, the amount of weapon with currentWeapon id should be decreased by one. Thus, the number of sent weapon should be one less.

- **context LevelManager::moveUserPlane(d) post:**
  userPlane.getPositionY = @pre.userPlane.getPositionY + (userPlane.speedY + userPlane.pilot.speed)

After the user moves the plane, the y position of user plane should be updated by the sum of y axis speed of user plane and the speed of the pilot of the user plane

- **context LevelManager::turnPointsIntoCoins(p) post:**
  collectionManager.coins = @pre. collectionManager.coins + levelManager.level.points / levelManager.level.coinCoefficient

After the level is over, to turn points collected in level to coins turnPointsIntoCoins(p) method of LevelManager is called. After the points are turned into coins, the coin amount in user collection should be increased by the points collected in level divided by the level coin coefficient.

- **context LevelManager::shoot(g) post:**
  if (weapons->select(w:Weapon | w.index = currentWeaponId) > 0)
    userWeapons->size = @pre.userWeapons->size + 1

If the user has enough amount of the current weapon which is specified as the element of weapons list at index currentWeaponId, then a new weapon is added to the user weapons list. Hence if his weapon is left, the player can shoot and increase number of his fired weapons.

- **context LevelManager::changeWeapon(g) post:**
  if(@pre.currentWeaponId == 6)
    currentWeaponId = 0
  else
    currentWeaponId = @pre.currentWeaponId + 1

When changeWeapon(g) method of LevelManager is called the currentWeaponId is increased by one, to pass to next weapon. However, if the currentWeapon is the last weapon, it had id 6, then the system should go back first weapon which has id 0. This means if user currently has Missile, when he changes weapon he goes back to Bullet. Otherwise he passes to next weapon.

- **context CollisionManager::removeExplosions() post:**

explosions->size = 0

After removeExplosions() method of CollisionManager is called, all explosion Images should be removed from explosions list. This means, the explosions should not be drawn on the screen anymore.

- **context UserPlaneDestroyedCollisionManager::handleCollision(o1, o2) post:**
  levelManager.userPlane.health = 0

If user plane collides with a target or obstacle, and UserPlaneDestoyedCollisionManager is called for handling the collision, the health of the user plane should be 0. Hence, if user plane collides with a target or obstacle, its health depletes.

- **context UserPlaneDestroyedCollisionManager::handleCollision(o1, o2) post:**
  levelManager.gameManager.observers->select(o: GameManagerObserver | o.type = LevelPanel).isGameOver = true

If user plane collides with a target or obstacle, and UserPlaneDestoyedCollisionManager is called for handling the collision, the levelPanel should indicate that the game is over. Hence, if user plane collides with a target or obstacle, the game is over and the level panel is updated accordingly.

- **context UserPlaneHitCollisionManager::handleCollision(o1, o2) post:**
  levelManager.userPlane.health = @pre.levelManager.userPlane.health – o2.damage

If one of enemy weapons collides with user plane, and UserPlaneHitCollisionManager is called for handling the collision, the health of user plane should be decreased by the damage of the enemy wepon. Hence, if user plane collides with an enemy weapon, its health decreases be the damage amount of weapon.

- **context TargetHitCollisionManager::handleCollision(o1, o2) post:**
  o2.health = @o2.health – (o1.damage + userPlane.shootDamage)

If one of user weapons collides with a target, and TargetHitCollisionManager is called for handling the collision, the health of target should be decreased by the the sum of user weapon damage and shoot damage of the user plane itself. Hence, if user plane shoots a target, the health of target decreases by the sum of the damage amount of weapon and shoot damage of user plane.

- **context Level inv:**
  points >= 0

The points collected in a level should never be smaller than 0. Even though user loses points when he has 0 points, the points should stay at 0 value.

- **context Level inv:**
  coinCoefficient >= 1

The coinCoefficient for a level cannot be lower than 1. The coin coefficient denotes how many points collected in level corresponds to one coin. The user should never earn more than one coins for one point collected.

- **context Level inv:**

weapons->forAll(i: int | i >= 0)

Weapons denotes the list of numbers of weapons purchased. The number of weapons should decrease when used but should never drop below 0. This means a weapon cannot be used if its amount is 0.

- **context Level inv:**
  targetPlanes->intersection(createdTargetPlanes)->size = 0

The list of targetPlanes should not intersect with the list of created target planes. An element is added to targetPlanes list only after it is removed from createdTargetPlanes list. targetPlanes list holds all target planes appearing in level. createdTargetPlanes denote the list of created but not yet appeared target planes. Hence, an instance of target plane can either be on screen or waiting in list of created target planes.

- **context Level inv:**
  targets->intersection(createdTargets)->size = 0

The list of targets should not intersect with the list of created targets. An element is added to targets list only after it is removed from createdTargets list.

- **context Level inv:**
  obstacles->intersection(createdObstacles)->size = 0

The list of obstacles should not intersect with the list of created obstacles. An element is added to obstacles list only after it is removed from createdObstacles list.

- **context Level inv:**
  bonusPackages->intersection(createdBonusPackages)->size = 0

The list of bonusPackages should not intersect with the list of created bonusPackages. An element is added to bonusPackages list only after it is removed from createdBonusPackages list.

- **context Level inv:**
  targets->forAll(t: Target | t.health > 0)

A target can only be contained in the list of targets if its health is larger than 0. When the health of a target drops to 0 or below, it should be removed from the list, therefore from the screen.

- **context Level::createObjects(t, f) post:**
  createdTargets->select(tr: Target | tr.appearTime = t)->size = 0
  targets->contains(tr: Target | tr.appearTime = t)

After the createObjects(t,f) method of Level is called, any item that has the appear time given in the method is removed from createdTargets list and added to targets list. If the appear time of target is the same as the level time, the target should be deleted from createdTargets and added to actual targets that are drawn on the screen.

- **context TargetPlane inv:**
  shoots->size > 0

All instances of TargetPlane should contain more than 0 shoots. This means all target planes should send weapons to user plane.

- **context TargetPlane::removeShoot(s) pre:**
  shoots->contains(s)

A shoot can be removed from list of shoots in TargetPlane class only if the list contains shoots.

6. *Conclusions and Lessons Learned*

In this report the application domain of Sky Wars was examined and its requirements were extracted. These requirements were analyzed in order to plan the solution domain. Functionalities of the system were examined and modeled in the report. Moreover, the system design was examined in detail, subsystem decompositions, architectural styles and design decisions were explained. Finally, the object design was included in the report, solution domain objects were clarified with design pattern applications, interface specifications and constraints.

First of all, general themes and functions of the game were identified. The aim, context and control of the game were specified. Game concepts and items were explained in detail. Hence, a basic understanding of Sky Wars, what is it, how it is played, what are the properties and tasks of the game, was built.

Considering the description of the game, requirements were extracted. What Sky Wars should be able to do for certain functionalities were identified. Furthermore, non-functional requirements and system constraints were specified. In the guidance of requirements, sample scenarios were created for various functionalities. Then these scenarios were generalized as use cases. For a better understanding of the game, interface of Sky Wars was included in the report. Screen designs and pictures of game objects were provided.

Analysis part followed the Requirements Elicitation section. Class diagrams were created, system objects were identified and their operations and attributes were determined. The functional flow is represented with Activity Diagrams. For detailed explanation of interactions among objects, Sequence Diagrams were provided. These UML diagrams created a bridge between application and solution domains.

After determining details of application domain with requirements and analysis phases, the difference between application and solution domains were tried to be narrowed by system design. First of all, design goals and trade-offs were explained. The system was decomposed into subsystems and architectural patterns were identified. Deployment and Component diagrams were used for elaborating Hardware/Software mapping decisions. Other system decisions such as data management and software control were explained in detail.

The object design was examined after the system design details were clarified. The system was re-examined to detect problems with system functionality, performance, coherence, extensibility and robustness. Considering the discovered problems and the structure of our system, appropriate design patterns were applied. These patterns were helpful for finding missing classes and functionalities of the system. Then the class interfaces were examined in detail; the types, signatures, visibilities and functions of all class attributes and operations were listed. The attributes and operations were further clarified with OCL contracts which specified the invariants, preconditions and post conditions of the class interfaces. Hence a better understanding of Sky Wars architecture was constructed.

Consequently, this report is an example of Object Oriented Software Design. The system was built gradually; the software was produced from the problem statement after the application of all steps of object-oriented approach.

Sky Wars is basically a level-based Javanoid game. The player is controlling a user plane and he is trying to collect as many points as he can without depleting his health. Our game has many different objects. Various targets appear in level that behave differently and earn user various points. User gains coins in levels and can purchase different planes, pilots and weapons with these coins. Furthermore, bonus packages are included in the game to boost or harden levels.

Sky Wars aims to be a user-friendly, functional and reliable program. Flexibility, efficiency and high performance are also essential for our game. With the light of these design goals, Sky Wars is designed and implemented in a way that it consists of three hierarchic levels. Facade, Observer and Strategy patterns are applied to system. Consequently, we managed to reach to our functional, non-functional and

design goals with the help of object-oriented design principles.

Following object-oriented principles brought many conveniences as well as many challenges. Object-oriented design eases many steps of software development. Without OO principles, it is very hard to produce a successful software product in a limited time which is consistent with initial requirements. These techniques allowed us to specify, refine and perfect our system design and structure before implementing it. Requirements analysis step was crucial for creating a common and fundamental understanding of the system. Hence, any participant of the project can figure out what is required from the software product both from user and developer perspectives. Other development steps were all based on extracted requirements. Hence, we managed to maintain consistency within our developing system. Analysis, on the other hand, served as a basis for the functionality of the system. How Sky Wars is expected to function is demonstrated by diagrams. Static and dynamic behaviors of the system were clarified. Hence, the actual implementation became easier. Design process was also very helpful for the perfection of the project. System design allowed us to review our design goals and create strategies to achieve these non-functional requirements. Furthermore, considering concurrency, data management and boundary conditions before the implementation allowed us to foresee potential problems and find solutions to them. Therefore, implementation process was efficient. Object design helped us to increase functionality, extendibility and clarity of the system by the application of appropriate design patterns. Hence, the system of Sky Wars was designed in a way that it maximizes efficiency and flexibility. With the help of this design, it was possible for us to implement the code that is consistent with functional and nonfunctional requirements.

We learned that object-oriented principles are helpful for producing a maintainable and sustainable software. The detailed analysis and design of the system allowed us to divide the system to manageable components. Also, the principle of working with subsystems and subcomponents helped us to build the whole system from easy to manage smaller subsystems. If the software is produced from scratch, the system may function the same as a system developed with OO principles. However, it is very hard to change, upgrade or fix a system unless it is already developed to be open to development. OOP techniques allowed our system to be open to development because the documentation allows coders to understand the system to change it and low-coupling between subsystem and various design patterns allow flexibility and extendibility. Since we are building a game, it is crucial for our software to be open to upgrades and OO helped us about this goal.

However, during development we understood that writing code with object-oriented principles is much harder. It is possible to implement a functionality with a line of code for example, but OO requires almost 10 times more code writing, more classes, interfaces and connections. Moreover, documentation may also be a long and tedious process. It is also very hard to maintain consistency in documents and good documentation requires great effort. Even though OO software development has its downsides, it eases development in other respects. The more complex and comprehensive the software project becomes the more useful OOP becomes. As we continued developing our project and adding more details and functions, we understood that object-based design eased our job. It is now possible for example, to add a new feature to our game with a little effort and without changing the structure of the system.

Furthermore, we learned that revision is essential for object oriented development. After each development step, you need to revise all your requirements, diagrams, system structure in order to maintain consistency. In our project we changed class diagrams, sequence diagrams and subsystem structures after each development state. Even after each step of implementation, we needed to update the system design. Hence, managing consistency between documents and code is hard yet very important. We understood that if this principle is applied, it is possible produce a successful software.

Consequently, this project taught us the principles of object-oriented development, how to manage and maintain a system better. We had difficulties with applying some steps however, we learned in the end to produce a high quality software in limited time in a structured and professional manner.