BILKENT UNIVERSITY

CS 319 – Object-Oriented Software Engineering

Design Report

Intergalactica

Group 18

Ahmet Burak Şahin 21301755

Melis Kızıldemir 21401184

Selin Erdem 21401244

Ömer Sakarya 21301535

Table of Contents

1. Introduction

1.1 Purpose of the System

Among many versions of the *Galaga* [1], there is a missing element of ***adventure*** that might bring forgotten space-based games back to once-glorious days. *Intergalactica* is designed to fill that gap. It features some of the key concepts that was missing in the original implementation, such as travelling between planets, acquiring new ships throughout voyage and most importantly, having a *purpose* and an *end*.

Intergalactica project aims to furnish not only an adventurous, but also a graphically better and flawless experience to its aficionados.

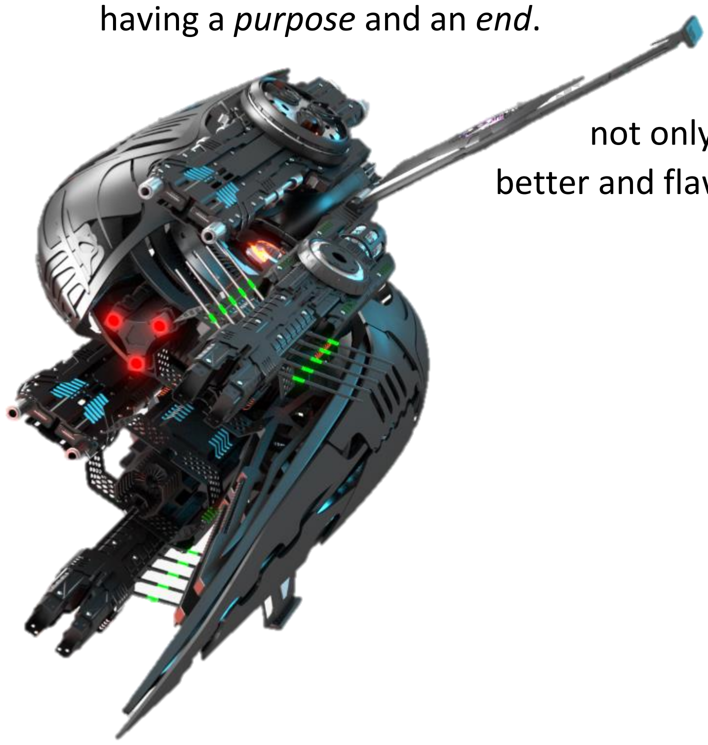


Image by MillionthVector [2].

1.2 Design Goals

➔ ***Accessibility***

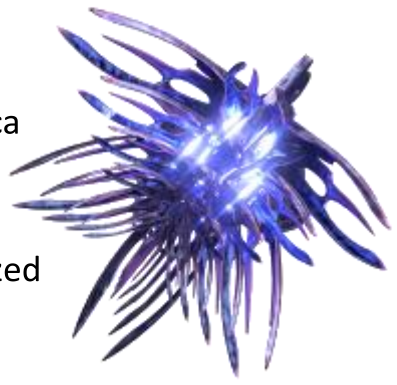
Intergalactica will be highly accessible since it is going to be free to use. We will distribute our game for the enthusiasts who may want to try.

➔ ***Efficiency:***

The main design goal in Intergalactica is being able to provide a smooth gaming experience to the user along with a dazzling UI design.

➔ ***Maintainability:***

Almost in every step of software development, a major care must be taken to ensure the final product is maintainable. So does the development of Intergalactica have taken maintainability as pivotal factor. Entire project is being kept under the VCS (version control system –Git) to ensure that the project remains organized and in any case of failure in the system, issue can be addressed by tracing the development background.



➔ ***Portability:***

Intergalactica is being implemented in Java platform. This specific language is chosen because it runs regardless of the underlying system as long as the machine contains Java Virtual Machine (JVM).

➔ ***Reliability:***

For possible run-time issues, any kind of exception will be handled thoroughly on the fly in order not to interrupt the gaming experience. Severe crashes in general, might as well corrupt the data that was collected previously by the user, however, Intergalactica's one of the core design goals is to be reliable and robust to not to result in data loss or cause any damage to the system.

➔ ***User-Friendliness:***

Intergalactica will have very plain, easy to use and intuitive interface even that –although we will include a brief one–a tutorial wouldn't be necessary for any regular user to get the hang of it.

All Images in this page by MillionthVector [2].
Images are 3D imaginary depictions of some of our alien ships.

2. Software Architecture

2.1 Subsystem Decomposition

The decomposition of our subsystems is depicted as follows:

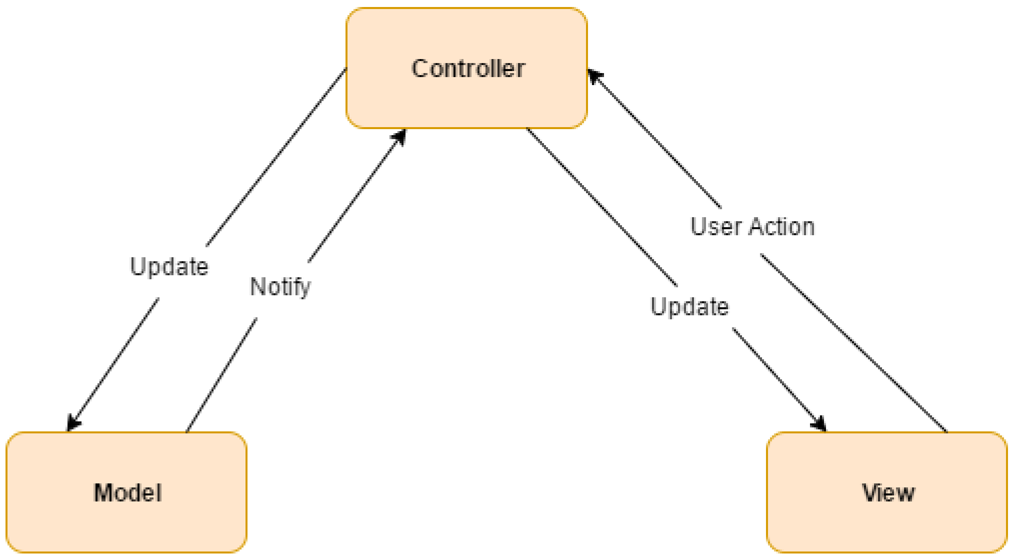


Diagram above shows Subsystem Decomposition of Intergalactica.

Intergalactica's subsystems suit well with the MVC architectural style. Every boundary and user-related operations are included in the Model Subsystem that deals with the user interface management and input issuing to the Controller Subsystem. Speaking of which, the Controller Subsystem takes and interprets user's actions and updates the system as a response, in either of graphical and structural ends. Lastly, the backbone of the whole system, the Model Subsystem's main task is to represent a relational structure between objects. Model Subsystem notify in any important change on its state, however, more often it applies the commands issued by the Controller Subsystem and reevaluates its state values accordingly. All packages' services will be discussed in Section 3.

2.2Hardware/Software Mapping

Our system shall be realized via both software and hardware point of view. From software perspective, since it is going to be developed in Java platform, any computer having Java Runtime Environment (JRE) will be able to run Intergalactica, independent of the underlying OS.

From the hardware point of view, the requirements are relatively minimal. I/O devices necessary to play the game consists of: a PC, a mouse, a keyboard and a monitor. Regardless of the size of the monitor, Intergalactica will run correctly, because the software will be window-based.

2.3 Persistent Data Management

Nonvolatile data management will be handled in Controller subsystem. Using the User's hard drive, only top 10 high scores along with the scorers' names will be stored. The file format we have chosen for the ease of parsing is .csv (comma separated value) for the ease of use. Moreover, since each entity will have its own appearance through a picture, each .png file will also be permanently stored. Finally, for background music and game sound, we will store .wav files in the hard disk.

2.4Access Control and Security

In Intergalactica there's no authentication process and users will not, by any means, be prompted to share any particular personal information.

2.5Boundary Conditions

Initialization:

- Intergalactica is downloaded from the source and placed somewhere in the hard disk drive.
- User starts the program.
- System boots up and User can start playing the game.

Termination:

- In any page, User presses ESC key which makes a menu pop up.
- From the Pop-up menu, User selects Quit.
- OR, User can directly exit from the main menu, selecting Quit option.

Failure:

- Since Intergalactica does not support save/load feature for any game, in case of a failure, all the progress achieved by the User will be lost.
- System also can face technical faults during processing alien motion or animation generation.
- If Java Runtime Environment doesn't exist, Intergalactica cannot start up.
- In the case of failure, system will halt itself and display information through a pop-up window once it guarantees that persistent data is safe.

3. Subsystem Services

3.1 View Subsystem Services

View subsystem constitutes a bridge between the User and the System. The main duty is to provide an uninterrupted –possibly via `Event Dispatch Thread`– interaction that will enable user to experience a reasonable game play. The View subsystem is composed of multiple layers of panels, each of which provide a different functionality. As apparent as it is, this subsystem is used mainly to gather inputs issued by the User and redirect it to appropriate components. Any entity corresponding to a game object is converted to its graphical representation and displayed on the game panel. In the game screen (a.k.a., game panel) reflects changes to the states of the application domain model-displayable objects by the commands issued from the Controller Subsystem.

3.2 Controller Subsystem Services

Controller's main duty, in simplest terms, is to organize and control over the whole system, respond incoming inputs from the View subsystem and generate outputs accordingly with the aid of the states encapsulated in the Model subsystem. Furthermore, it checks collisions with ammos and spaceships, either User's or an alien's, and does so by checking overlapping areas of game objects. In order to supply such collision checking utility, Controller subsystem keeps track of in-game entities to use them whenever necessary. Also, Controller subsystem manages aliens, allocates and scatters each Alien to the Game Map. Finally, it provides read-write file operations regarding the highscore and sound list retrieval and storing.

3.3 Model Subsystem Services

Model subsystem embodies the structure and behavior of the fundamental units of data for the system. Its constituents are realized from gathered application domain concepts. In terms of service, its sole purpose is to represent the realized structure and generate necessary responses to the controller and change to an appropriate state, accordingly. This subsystem acts like a repository and also a look-up model for the controller to enable it to make the best decisions in any situation. Its services are triggered by the Controller subsystem and the communication between them is bidirectional, meaning that result of any update is notified back to the Controller subsystem.

4. Low-level Design

4.1 Object Design trade-offs

Maintainability vs. Security

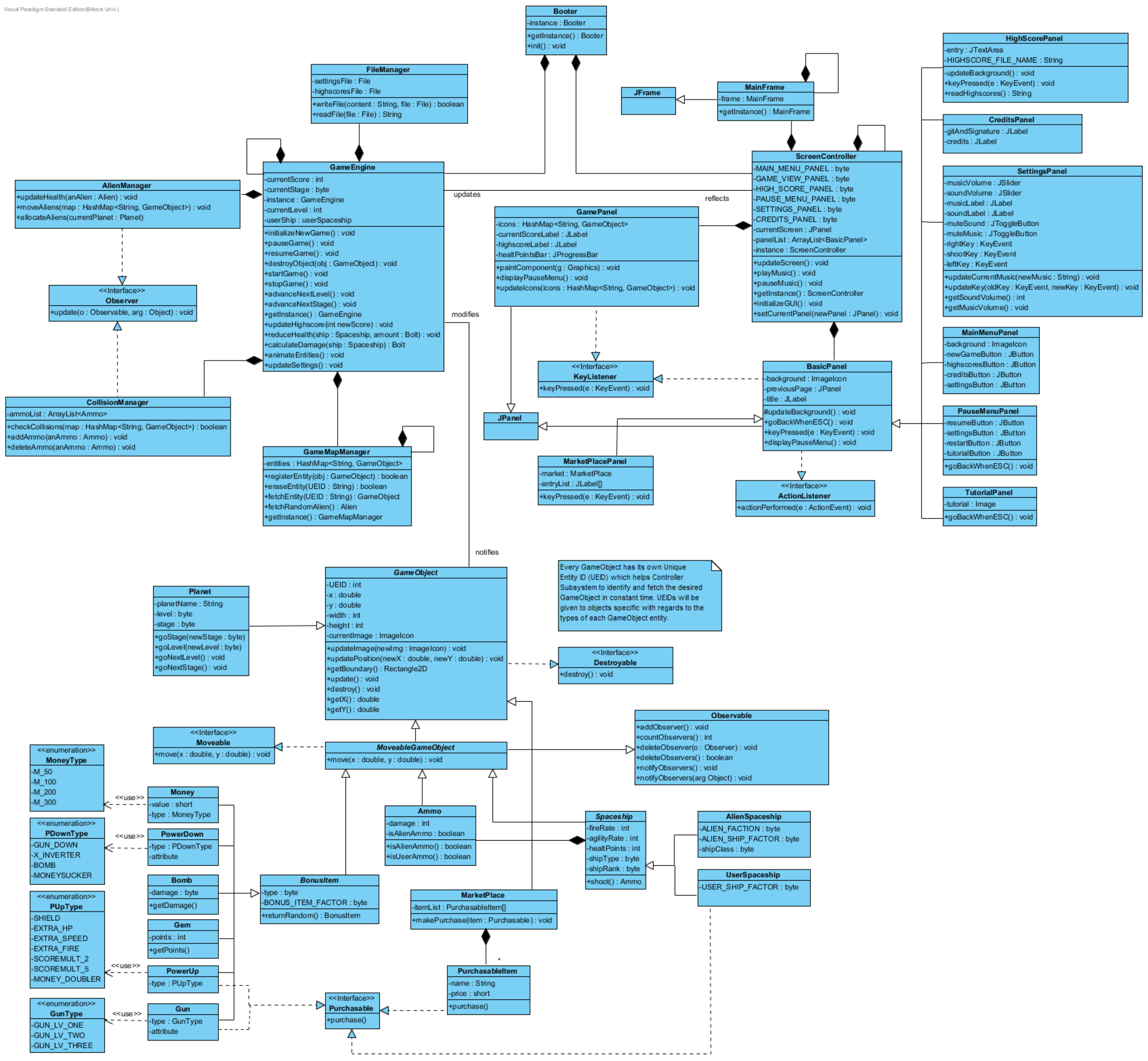
Implementing a secure system would increase the complexity of the design and would yield the final product even harder to maintain. We have favored maintainability in this case because since we do not obtain any of the personal information of the user while file storage and merely keep the name of him/her, we decided that maintaining of code for more stable future versions would be the best choice.

Performance vs. Level-of-Detail

Since Java runs on a virtual machine and the levels from Java source code to machine code is longer compared to languages like C++, we had to be careful about determining the level of details in our object design. Because defining objects for almost everything would give us an overhead of complexity and one of our design goal, that is to give the most efficient gaming experience wouldn’t be met. Hence, we decided to keep the level-of-specialization up to a decent level and tried to optimize the design as good as possible.

4.2 Final Object Design

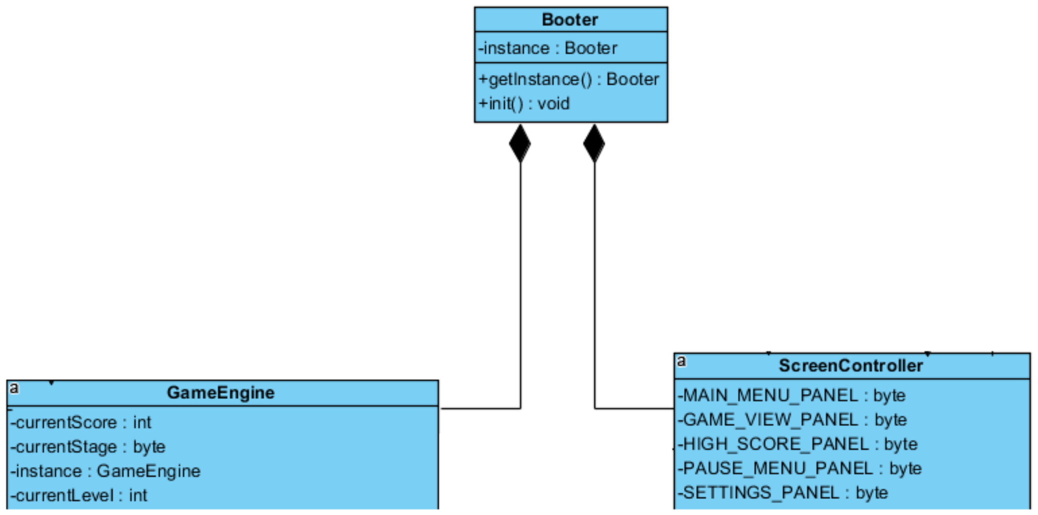
In this section, the choices behind design patterns and their reasoning will be articulated. Furthermore, the implementations for specific structures will be explained as well. The main class diagram is as follows:



Visual Paradigm Standard Edition(Bilkent Univ.)
Booter
-instance : Booter
+getInstance() : Booter
+init() : void
FileManager
-settingsFile : File
-highscoresFile : File
+writeFile(content : String, file : File) : boolean
+readFile(file : File) : String
JFrame
MainFrame
-frame : MainFrame
+getInstance() : MainFrame
HighScorePanel
-entry : JTextArea
-HIGHSCORE_FILE_NAME : String
-updateBackground() : void
+keyPressed(e : KeyEvent) : void
+readHighscores() : String
CreditsPanel
-gitAndSignature : JLabel
-credits : JLabel
SettingsPanel
-musicVolume : JSlider
-soundVolume : JSlider
-musicLabel : JLabel
-soundLabel : JLabel
-muteSound : JToggleButton
-muteMusic : JToggleButton
-rightKey : KeyEvent
-shootKey : KeyEvent
-leftKey : KeyEvent
+updateCurrentMusic(newMusic : String) : void
+updateKey(oldKey : KeyEvent, newKey : KeyEvent) : void
+getSoundVolume() : int
+getMusicVolume() : void
AlienManager
+updateHealth(anAlien : Alien) : void
+moveAliens(map : HashMap<String, GameObject>) : void
+allocateAliens(currentPlanet : Planet)
GameEngine
-currentScore : int
-currentStage : byte
-instance : GameEngine
-currentLevel : int
-userShip : userSpaceship
+initializeNewGame() : void
+pauseGame() : void
+resumeGame() : void
+destroyObject(obj : GameObject) : void
+startGame() : void
+stopGame() : void
+advanceNextLevel() : void
+advanceNextStage() : void
+getInstance() : GameEngine
+updateHighscore(int newScore) : void
+reduceHealth(ship : Spaceship, amount : Bolt) : void
+calculateDamage(ship : Spaceship) : Bolt
+animateEntities() : void
+updateSettings() : void
updates
reflects
modifies
notifies
ScreenController
-MAIN_MENU_PANEL : byte
-GAME_VIEW_PANEL : byte
-HIGH_SCORE_PANEL : byte
-PAUSE_MENU_PANEL : byte
-SETTINGS_PANEL : byte
-CREDITS_PANEL : byte
-currentScreen : JPanel
-panelList : ArrayList<BasicPanel>
-instance : ScreenController
+updateScreen() : void
+playMusic() : void
+pauseMusic() : void
+getInstance() : ScreenController
+initializeGUI() : void
+setCurrentPanel(newPanel : JPanel) : void
GamePanel
-icons : HashMap<String, GameObject>
-currentScoreLabel : JLabel
-highscoreLabel : JLabel
-healtPointsBar : JProgressBar
+paintComponent(g : Graphics) : void
+displayPauseMenu() : void
+updateIcons(icons : HashMap<String, GameObject>) : void
<<Interface>>
Observer
+update(o : Observable, arg : Object) : void
<<Interface>>
KeyListener
+keyPressed(e : KeyEvent) : void
BasicPanel
-background : ImageIcon
-previousPage : JPanel
-title : JLabel
#updateBackground() : void
+goBackWhenESC() : void
+keyPressed(e : KeyEvent) : void
+displayPauseMenu() : void
MainMenuPanel
-background : ImageIcon
-newGameButton : JButton
-highscoresButton : JButton
-creditsButton : JButton
-settingsButton : JButton
PauseMenuPanel
-resumeButton : JButton
-settingsButton : JButton
-restartButton : JButton
-tutorialButton : JButton
+goBackWhenESC() : void
TutorialPanel
-tutorial : Image
+goBackWhenESC() : void
JPanel
CollisionManager
-ammoList : ArrayList<Ammo>
+checkCollisions(map : HashMap<String, GameObject>) : boolean
+addAmmo(anAmmo : Ammo) : void
+deleteAmmo(anAmmo : Ammo) : void
GameMapManager
-entities : HashMap<String, GameObject>
+registerEntity(obj : GameObject) : boolean
+eraseEntity(UEID : String) : boolean
+fetchEntity(UEID : String) : GameObject
+fetchRandomAlien() : Alien
+getInstance() : GameMapManager
MarketPlacePanel
-market : MarketPlace
-entryList : JLabel[]
+keyPressed(e : KeyEvent) : void
<<Interface>>
ActionListener
+actionPerformed(e : ActionEvent) : void
Every GameObject has its own Unique Entity ID (UEID) which helps Controller Subsystem to identify and fetch the desired GameObject in constant time. UEIDs will be given to objects specific with regards to the types of each GameObject entity.
Planet
-planetName : String
-level : byte
-stage : byte
+goStage(newStage : byte)
+goLevel(newLevel : byte)
+goNextLevel() : void
+goNextStage() : void
GameObject
-UEID : int
-x : double
-y : double
-width : int
-height : int
-currentImage : ImageIcon
+updateImage(newImg : ImageIcon) : void
+updatePosition(newX : double, newY : double) : void
+getBoundary() : Rectangle2D
+update() : void
+destroy() : void
+getX() : double
+getY() : double
<<Interface>>
Destroyable
+destroy() : void
<<Interface>>
Moveable
+move(x : double, y : double) : void
MoveableGameObject
+move(x : double, y : double) : void
Observable
+addObserver() : void
+countObservers() : int
+deleteObserver(o : Observer) : void
+deleteObservers() : boolean
+notifyObservers() : void
+notifyObservers(arg Object) : void
<<enumeration>>
MoneyType
-M_50
-M_100
-M_200
-M_300
<<use>>
Money
-value : short
-type : MoneyType
<<enumeration>>
PDownType
-GUN_DOWN
-X_INVERTER
-BOMB
-MONEYSUCKER
<<use>>
PowerDown
-type : PDownType
-attribute
Bomb
-damage : byte
+getDamage()
Gem
-points : int
+getPoints()
<<enumeration>>
PUpType
-SHIELD
-EXTRA_HP
-EXTRA_SPEED
-EXTRA_FIRE
-SCOREMULT_2
-SCOREMULT_5
-MONEY_DOUBLER
<<use>>
PowerUp
-type : PUpType
<<enumeration>>
GunType
-GUN_LV_ONE
-GUN_LV_TWO
-GUN_LV_THREE
<<use>>
Gun
-type : GunType
-attribute
BonusItem
-type : byte
-BONUS_ITEM_FACTOR : byte
+returnRandom() : BonusItem
Ammo
-damage : int
-isAlienAmmo : boolean
+isAlienAmmo() : boolean
+isUserAmmo() : boolean
Spaceship
-fireRate : int
-agilityRate : int
-healtPoints : int
-shipType : byte
-shipRank : byte
+shoot() : Ammo
AlienSpaceship
-ALIEN_FACTION : byte
-ALIEN_SHIP_FACTOR : byte
-shipClass : byte
UserSpaceship
-USER_SHIP_FACTOR : byte
MarketPlace
-itemList : PurchasableItem[]
+makePurchase(item : Purchasable) : void
PurchasableItem
-name : String
-price : short
+purchase()
<<Interface>>
Purchasable
+purchase()

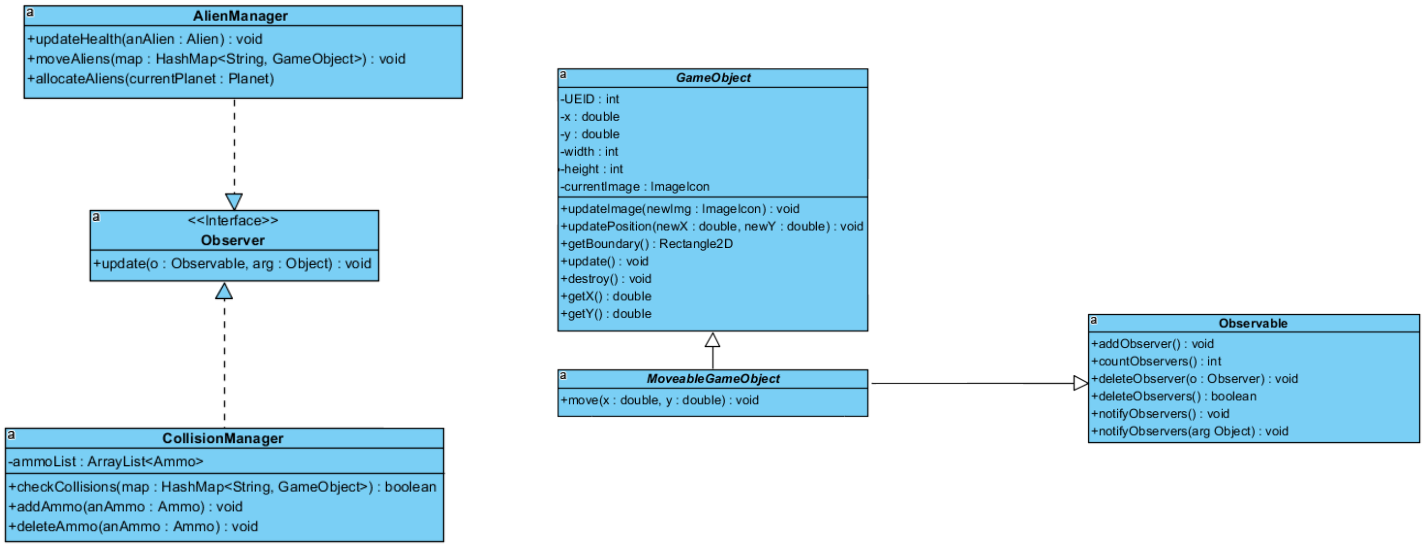
➔ Façade Design Pattern

Since the operation of the system depends on the correct instantiation of both View and Controller, we need to somehow reduce the coupling for better instantiation and booting of the system. We came up with a class `Booter` that boots the whole system properly from a single interface, and can be instantiated only once.



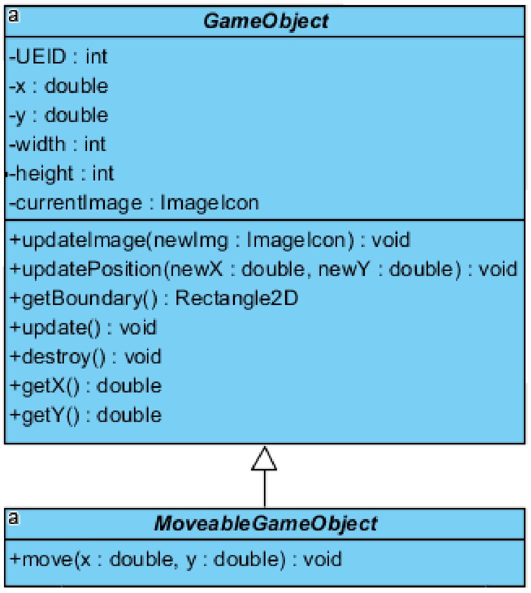
➔ Observer Design Pattern

Observation is necessary when an `Observer` seeks for a particular change in `Observee` and takes action accordingly. In our implementation, instead of constantly checking and updating the states of each item, which would be a waste of processing power, it is better to *notify* the change to the observers whenever it happens. When a `MovableGameObject` takes a hit or changes its desired state relative to its previous state, all of the `Observers, CollisionManager` and `AlienManager` in our case, checks, for example, whether a collision has occurred, if so, a health decrement is necessary for the given `Spaceship.` Thus, `Observer` calls corresponding methods. For such applications, we implemented Observer Design Pattern.


AlienManager
+updateHealth(anAlien : Alien) : void
+moveAliens(map : HashMap<String, GameObject>) : void
+allocateAliens(currentPlanet : Planet)
<<Interface>>
Observer
+update(o : Observable, arg : Object) : void
CollisionManager
-ammoList : ArrayList<Ammo>
+checkCollisions(map : HashMap<String, GameObject>) : boolean
+addAmmo(anAmmo : Ammo) : void
+deleteAmmo(anAmmo : Ammo) : void
GameObject
-UEID : int
-x : double
-y : double
-width : int
-height : int
-currentImage : ImageIcon
+updateImage(newImg : ImageIcon) : void
+updatePosition(newX : double, newY : double) : void
+getBoundary() : Rectangle2D
+update() : void
+destroy() : void
+getX() : double
+getY() : double
MoveableGameObject
+move(x : double, y : double) : void
Observable
+addObserver() : void
+countObservers() : int
+deleteObserver(o : Observer) : void
+deleteObservers() : boolean
+notifyObservers() : void
+notifyObservers(arg Object) : void

➔ Adapter Design Pattern

We have used Adapter Design Pattern in order to remain consistent in our Model design. Some objects in our Model does not conform to a specified behavior, namely most of our objects were *mobile objects* such as `Spaceships` and `BonusItems`, but some of were *immobile,* such as a Planet. To distinguish these two incompatible types, we have placed an Adapter class called `MoveableGameObject` to interface and differentiate mobile and immobile types.

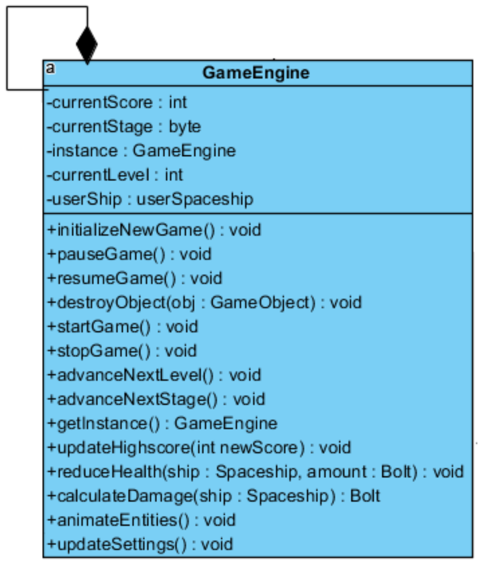

GameObject
-UEID : int
-x : double
-y : double
-width : int
-height : int
-currentImage : ImageIcon
+updateImage(newImg : ImageIcon) : void
+updatePosition(newX : double, newY : double) : void
+getBoundary() : Rectangle2D
+update() : void
+destroy() : void
+getX() : double
+getY() : double
MoveableGameObject
+move(x : double, y : double) : void

➔ Singleton Design Pattern

Singleton Design Pattern is used in our project to guarantee that there will be always a single instance of the desired class, such as a `GameEngine` or a `Booter` to ensure that there will be at most one `GameEngine` or a `Booter` because otherwise, there would be race conditions and confusions in state, etc. which is undesirable. Hence, we have prevented this by making some of our classes in our Object Design Singleton. They are

- `Booter`
- `GameEngine`
- `ScreenController`
- `GameMapManager`
- `MainFrame`
- `UserSpaceship`

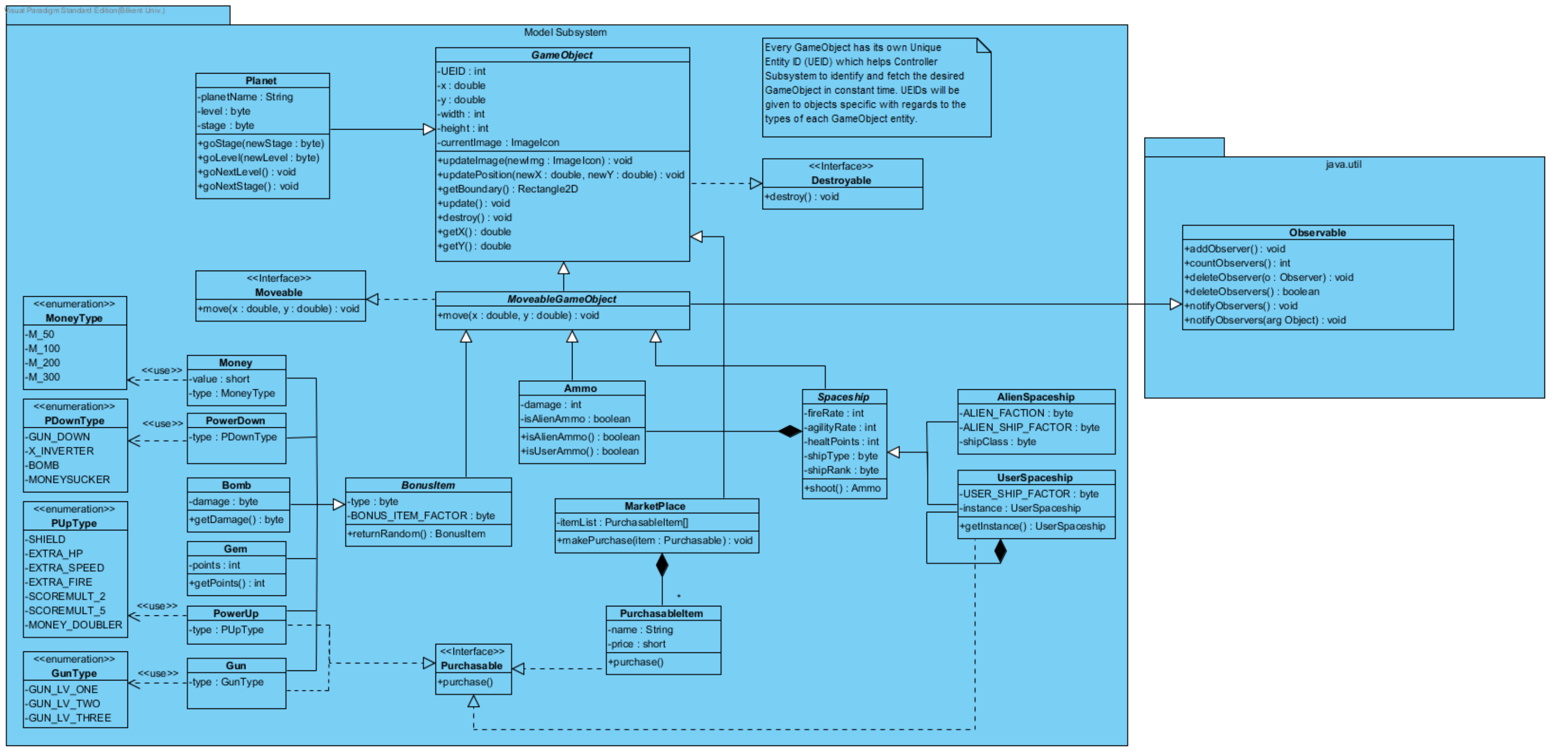
The depiction of Singleton is the same as the following `GameEngine` example:


a
GameEngine
-currentScore : int
-currentStage : byte
-instance : GameEngine
-currentLevel : int
-userShip : userSpaceship
+initializeNewGame() : void
+pauseGame() : void
+resumeGame() : void
+destroyObject(obj : GameObject) : void
+startGame() : void
+stopGame() : void
+advanceNextLevel() : void
+advanceNextStage() : void
+getInstance() : GameEngine
+updateHighscore(int newScore) : void
+reduceHealth(ship : Spaceship, amount : Bolt) : void
+calculateDamage(ship : Spaceship) : Bolt
+animateEntities() : void
+updateSettings() : void

4.3 Packages

➔ Model Package

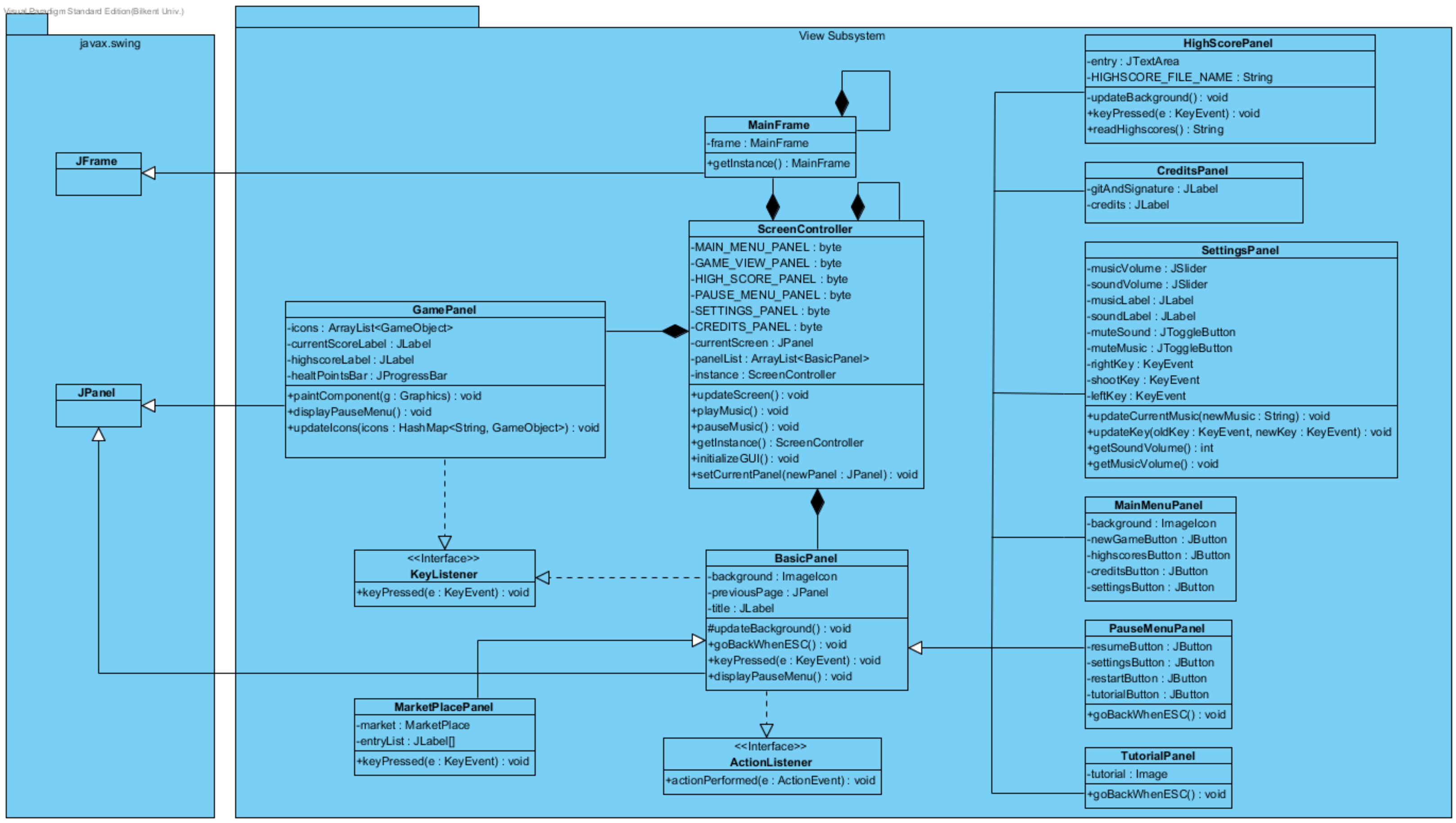
Model package consists of modelling objects used to reflect the application domain concepts. Controller accesses and manipulates `GameObjects` which reside on the top of the hierarchy inside the package. In our model we have 2 types of `GameObjects;` either mobile or immobile objects. Each object will have a representation in the actual game panel, thus we need to have x, y coordinates for every one of them, also an icon for graphical representation. Every game object must be `Destroyable` since there is, for all of them, a point in time where they should disappear from the screen due to game logic. Furthermore, we can only purchase an item from `MarketPlace` of type `Purchasable` for consistency. Each entry in the `MarketPlace` will be represented by the `PurchasableItem` to avoid the confusion of mobility (Since a `BonusItem` is-a `Moveable` and a `PurchasableItem` is just an entry in the `MarketPlace`, so it's immobile and the distinction is made there). The Package is as shown below:



Every GameObject has its own Unique Entity ID (UEID) which helps Controller Subsystem to identify and fetch the desired GameObject in constant time. UEIDs will be given to objects specific with regards to the types of each GameObject entity.

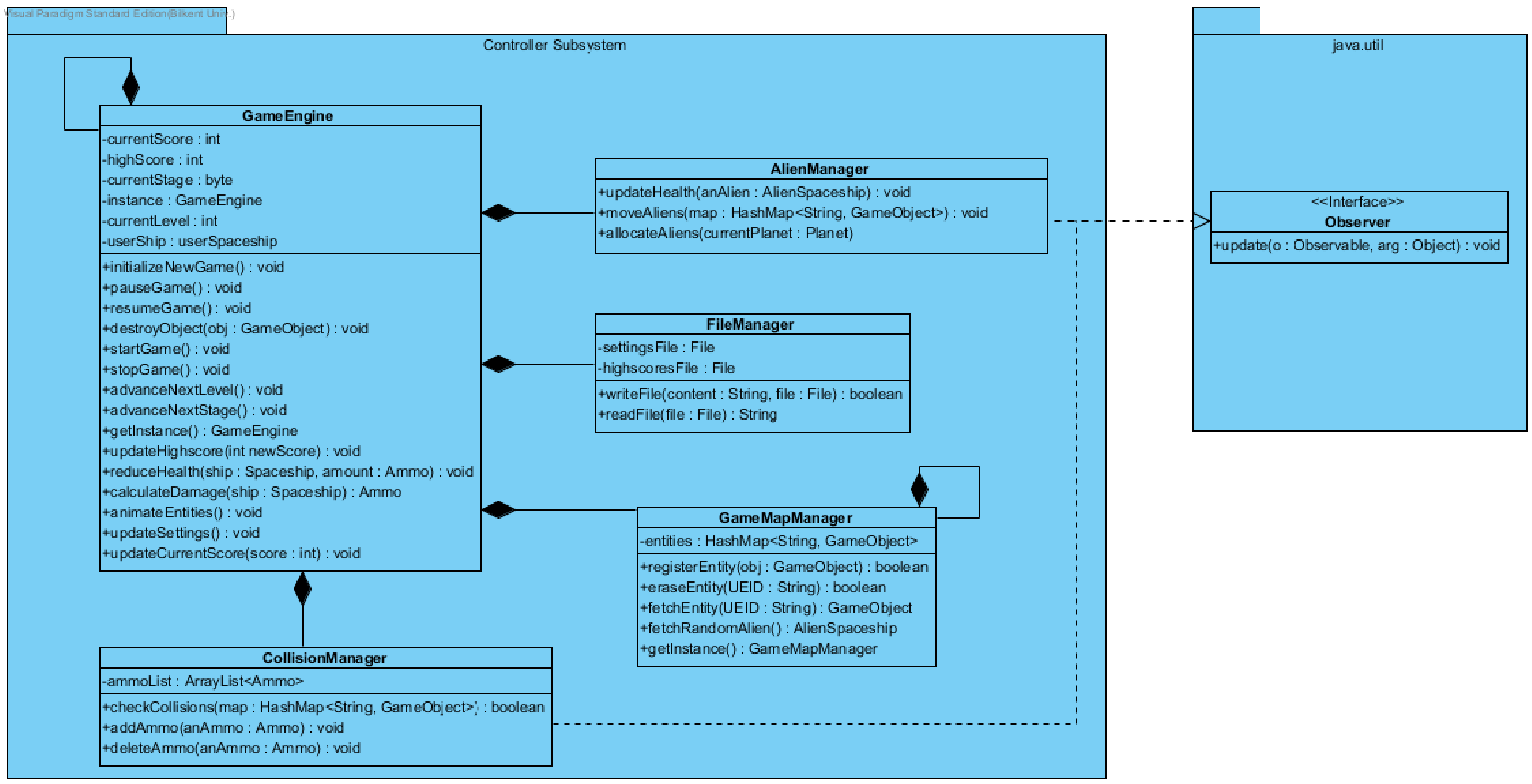
➔ View Package

The View package nearly consists entirely of panels, each of which has a different purpose and different duties. For example, `GamePanel` is a panel that corresponds to the main gaming panel of the system, so whenever user prompts program to start a new game, this is the canvas that the `GameObjects` will be placed upon. Every panel is a specialization of the `BasicPanel` which implements both `KeyListener` and `ActionListener` interfaces to enable user interact through buttons and key hits. The control mechanism and interface to the Controller subsystem is provided by the `ScreenController` class. It takes actions according to the commands issued by the Controller subsystem and does the necessary adjustments. The Package is as below:



➔ Controller Package

The controller package is the backbone of the the system. It manages resources, updates model entities, and refreshes visual representations as a result of the stimuli by the User. At the top, we have `GameEngine` that manages the most fundamental functions of the system. We also have `FileManager` for file operations, specifically storing and restoring highscores and settings. Also, `AlienManager's` duty is to update aliens by observing their states, allocate aliens in the beginning of each stage and move aliens around during combat. `GameMapManager` is the keeper of the entities in the system. It keeps a `HashMap<String, GameObject>` so that continuously retrieval operations will be limited to the complexity of $\boldsymbol{O}(\mathbf{1})$. (The reason of this choice will be addressed in Sec. 4.4, at first occurrence of UEID). Finally, The `CollisionManager` keeps track of `Ammo` objects thrown by either aliens or the User. If any hit occurs it returns true and lets the `GameEngine` take from there. Lastly, observe that `CollisionManager` and `AlienManager` are both `Observer`s. They observe existing `MovableGameObject` of interested type, and updates accordingly. The Package is depicted as the following:



Visual Paradigm Standard Edition(Bilkent Univ.)
Controller Subsystem
GameEngine
-currentScore : int
-highScore : int
-currentStage : byte
-instance : GameEngine
-currentLevel : int
-userShip : userSpaceship
+initializeNewGame() : void
+pauseGame() : void
+resumeGame() : void
+destroyObject(obj : GameObject) : void
+startGame() : void
+stopGame() : void
+advanceNextLevel() : void
+advanceNextStage() : void
+getInstance() : GameEngine
+updateHighscore(int newScore) : void
+reduceHealth(ship : Spaceship, amount : Ammo) : void
+calculateDamage(ship : Spaceship) : Ammo
+animateEntities() : void
+updateSettings() : void
+updateCurrentScore(score : int) : void
AlienManager
+updateHealth(anAlien : AlienSpaceship) : void
+moveAliens(map : HashMap<String, GameObject>) : void
+allocateAliens(currentPlanet : Planet)
FileManager
-settingsFile : File
-highscoresFile : File
+writeFile(content : String, file : File) : boolean
+readFile(file : File) : String
GameMapManager
-entities : HashMap<String, GameObject>
+registerEntity(obj : GameObject) : boolean
+eraseEntity(UEID : String) : boolean
+fetchEntity(UEID : String) : GameObject
+fetchRandomAlien() : AlienSpaceship
+getInstance() : GameMapManager
CollisionManager
-ammoList : ArrayList<Ammo>
+checkCollisions(map : HashMap<String, GameObject>) : boolean
+addAmmo(anAmmo : Ammo) : void
+deleteAmmo(anAmmo : Ammo) : void
java.util
<<Interface>>
Observer
+update(o : Observable, arg : Object) : void

4.4 Class Interfaces

➔ View Classes

MainFrame Class

Attributes:

private MainFrame frame: This is the main frame of the system. The panels such as GamePanel, HighScoresPanel will always be located on this frame.

Constructor:

private MainFrame(): Singleton constructor for Mainframe.

Methods:

private MainFrame getInstance(): This function returns reference of the only instance of the MainFrame. If the reference has not yet initialized, it first gets initialized.

ScreenController Class

Attributes:

private final byte MAIN_MENU_PANEL: A constant to enumerate the MainMenuPanel.

private final byte GAME_VIEW_PANEL: A constant to enumerate the GamePanel.

private final byte HIGH_SCORE_PANEL: A constant to enumerate the HighScorePanel

private final byte PAUSE_MENU_PANEL: A constant to enumerate the PauseMenuPanel.

private final byte SETTINGS_PANEL: A constant to enumerate the SettingsPanel.

private final byte CREDITS_PANEL: A constant to enumerate the CreditsPanel.

private JPanel currentScreen: The current JPanel that is under display.

`private ArrayList<BasicPanel> panelList:` The list of panels that are going to be placed on top of `GamePanel` basis.

`private ScreenController instance:` The only instance of `ScreenController.`

Constructors:

`private ScreenController():` The constructor cannot be called by any other class.

Methods:

`public void updateScreen():` This method reflects the changes after a possible change of currentPanel.

`public void playMusic():` System starts playing music.

`public void pauseMusic():` System pauses playing music.

`public ScreenController getInstance():` This function returns reference of the only instance of the `ScreenController`. If the reference has not yet initialized, it first gets initialized.

`public void initializeGUI():` This function initializes GUI, calls corresponding functions.

`public void setCurrentPanel(JPanel newPanel):` This function replaces the `currentPanel` with the `newPanel`.

`BasicPanel` **Class**

Attributes:

`private ImageIcon background:` The background image for all descendant panels.

`private JPanel previousPage:` When user hits ESC key, previous panel will be fetched and placed on top of the z-order.

`private JLabel title:` Title of the panel

Constructors:

`public BasicPanel():` Default constructor.

Methods:

`protected void updateBackground():` This method updates the background image, even though it will never be replaced, its used to initialize.

`public void goBackWhenESC():` Triggers the previousPage to be displayed.

`public void keyPressed(KeyEvent e):` Whenever a key action occurs by the user, this method will be called and will perform the specified action.

`puvlic void displayPauseMenu():` This method sets the current panel to be the invisible and sets `PauseMenu` to be displayed util User presses ESC.

`MarketPlacePanel` Class

Attributes:

`private MarketPlace market:` This is the market instance that contains all the necessary information about what do display.

`private JLabel[] entryList:` This is the list of graphical representations of the entries in the `MarketPlace`.

Constructors:

`public MarketPlacePanel():` Default constructor

Methods:

`public void keyPressed(KeyEvent e):` Whenever a key action occurs by the user, this method will be called and will perform the specified action.

`GamePanel` Class

Attributes:

`private ArrayList<GameObject> icons:` It is the list that, when iterated, will refresh and place every object's icon on the screen.

`private JLabel currentScoreLabel:` It is the label that will indicate the current score of the User

`private JLabel highScoreLabel:` It is the label that will indicate the highest score so far.

`private JProgressBar healthPointsBar:` It is the progress bar that will be used to show the health percentage of the user.

Constructors:

`public GamePanel():` Default constructor for the `GamePanel`

Methods:

`public void paintComponent(Graphics g):` This method uses this panel as a canvas and draws desired entities.

`public void displayMenu():` This function makes the current panel invisible and makes the `PauseMenuPanel` visible.

`public void updateIcons(ArrayList<GameObject> icons):` This function updates each function' icons.

`HighScorePanel` **Class**

Attributes:

`private final String HIGHSCORE_FILE_NAME :` This constant string designates the path and name of the file where the highscores are stored.

`private JTextArea entry:` This is the place where the highscores will be listed.

Constructors:

`public HighScorePanel():` Default constructor

Methods:

`private void updateBackground():` Updates the background picture, used of initialization or replacement of background picture.

`public void keyPressed(KeyEvent e):` It seeks for a Key event that will trigger for a certain event.

`public String readHighscores():` returns the highscores contained in the JTextArea.

`CreditsPanel` Class

Attributes:

`private JLabel gitAndSignature:` It contains the label that will specify the git and other signatures related to the developers.

`private JLabel credits:` Credits to be displayed

Constructors:

`public CreditsPanel():` Default constructor

`SettingsPanel` Class

Attributes:

`private JSlider musicVolume:` A slider to set the music volume.

`private JSlider soundVolume:` A slider to set the sound volume.

`private JLabel musicLabel:` Label that will indicate the type of slider.

`private JLabel soundLabel:` Label that will indicate the type of slider

`private JToggleButton muteSound:` Toggle button to decide whether or not to mute or unmute.

`private JToggleButton muteMusic:` Togggle button to decide whether of not to mute or unmute.

`private KeyEvent rightKey:` Setting of the key that will make spaceship to go right.

`private KeyEvent leftKey:` Setting of the key that will make spaceship to go left.

`private KeyEvent shootKey:` Setting of the key that will make spaceship shoot an `Ammo`.

Constructors:

`public SettingsPanel():` Default constructor

Methods:

`public void updateCurrentMusic(String newMusic):` It will replace the current music with the one whose path is specified via parameter `newMusic`.

`public void updateKey(KeyEvent oldKey, KeyEvent newKey):` It updates the key setting for the `oldKey`.

`MainMenuPanel` **Class**

Attributes:

`private ImageIcon background:` The secondary background icon for the intro.

`private JButton newGameButton:` It is the button that will initiate the initialization of the new game.

`private JButton highScoresButon:` It is the button that will make the high scores display

`private JButton creditsButton:` It is the button that will make the credits display.

`private JButton settingsButton:` It is the button that will make the settings display.

Constructors:

`public MainMenuPanel():` Default constructor

`PauseMenuPanel` **Class**

Attributes:

`private JButton resumeButton:` It is the button that will make the game resume what was doing before pausing.

`private JButton settingsButton:` It is the button that will make the settings display.

`private JButton restartButton:` It is the button that will restart the game.

`private JButton tutorialButton:` It is the button that will make the tutorial display.

Constructors:

`public PauseMenuPanel():` Default constructor

Methods:

`public void goBackWhenESC():` It sets visible the previous page and make itself invisible.

`TutorialPanel` **Class**

Attributes:

`private Image tutorial:` It is the tutorial image that will teach our users how to play.

Constructors:

`public TutorialPanel():` Default constructor

Methods:

`public void goBackWhenESC():` It sets visible the previous page and goes invisible

➔ Model Classes

`GameObject` Class

Attributes:

`private final int UEID:` Every entity, precisely `GameObject,` in the game has its own `Unique Entity ID (UEID)` so that we can refer to each individual object with its own ID number and also to use it to fetch the actual object rather than keepings lists of `GameObject`s around the system. ID will be calculated based on its type and a range of IDs. Since those properties will be known, an constant acces to a HashTable will result in near constant time retrieval for each kind of `GameObject` since we will know beforehand what IDs correspond to aliens or which ones correspond to PowerUps and so on.

`private double x:` It is the x coordinate of the object.

`private double y:` It is the y coordinate of the object.

`private int width :` Width of the object.

`private int height:` Height of the object.

`private ImageIcon currentImage:` Image of the object

Constructors:

`No Constructors since this class is abstract.`

Methods:

`public void updateImage(ImageIcon newImage):` Updates the image

`public void updatePosition(double newX, double newY):` New coordinates for the object.

`public Rectangle2D getBoundary():` Returns the boundary of the object.

`public void update():` Updates the object

`public void destroy():` Destroys the object

`public double getX():` Returns the X coordinate

`public doule getY():` Returns the Y coordinate

`Planet` Class

Attributes:

`private String planetName:` Name of this planet

`private byte level:` Level of this planet

`private byte stage:` Stage of this planet

Constructors:

`public Planet():` Default constructor

Methods:

`public void goStage(byte newStage):` Goes to the `newStage`.

`public void goLeel(byte newLevel):` Goes to the `newLevel`.

`public void goNextLevel():` Advances to the next level.

`public void goNextStage():` Advances to the next stage.

`Money` Class

Attributes:

`private short value:` Value of this `Money`.

`private MoneyType type:` The type of the `Money` selected among enum MoneyType.

Constructors:

`public Money():` Default constructor

`PowerDown` Class

Attributes:

private PDownType type: Type of this PoweDown among enum PDownType.

Constructors:

public PowerDown(): Default constructor

Bomb Class

Attributes:

private byte damage: The amount of damage it carries.

Constructors:

public Bomb(): Default constructor

Methods:

public void getDamage(): Returns the damage of this bomb

Gem Class

Attributes:

private int points: amount of points it carries.

Constructors:

public Gem(): Default constructor

Methods:

public int getPoints(): Returns the amount of points this gem carries

---

PowerUp Class

Attributes:

private PUpType type: The type of this power up among the enum PUpType

Constructors:

`public PowerUp():` Default constructor

`Gun` **Class**

Attributes:

`private GunType type:` The type of this gun among the types specified by the enum GunType

Constructors:

`public GunType():` Default constructor

`BonusItem` **Class**

Attributes:

`private final byte BONUS_ITEM_FACTOR:` The factor to be used in the calculation of `UEID`.

`private byte type:` The type of the bonus it carries

Methods:

`public BonusItem returnRandom():` Returns a random BonusItem object among its descendants.

`PurchasableItem` **Class**

Attributes:

`private String name:` Name of this `PurchasableItem`.

`private short price:` Price of this `PurchasableItem`.

Constructors:

`public PurchasableItem():` Default constructor

Methods:

`public void purchase():` Makes the purcase of this item.

`MarketPlace` **Class**

Attributes:

`private PurchasableItem[] itemList:` The list of the items that will be for sale in the `MarketPlace.`

Constructors:

`public MarketPlace():` Default constructor

Methods:

`public void makePurchase(Purchasable item):` Makes the purchase of the item in the argument

`Ammo` **Class**

Attributes:

`private int damage:` The damage this ammo carries, which will be dealt to the receiver when a collision occurs.

`private Boolean isAlienAmmo:` if true then it is Alien ammo otherwise User ammo.

Constructors:

`public Ammo(int damage):` Constructor for ammo.

Methods:

`public boolean isAlienAmmo():` True if its alien ammo

`public boolean isUserAmmo():` True if its user ammo.

***`MoveableGameObject`* Class**

Methods:

`public void move(double x, double y):` Moves to object to some x offset

***`Spaceship`* Class**

Attributes:

`private int fireRate:` Fire rate of this ship.

`private int agilityRate:` Agility rate of this ship.

`private int healthPoints:` Health points of this ship.

`private byte shipType:` Type of this ship

`private byte shipRank:` Rank of this ship

Methods:

`public Ammo shoot():` Returns an ammo when called.

`AlienSpaceship` Class

Attributes:

`private final byte ALIEN_FACTION:` The faction this alien belongs to.

`private final byte ALIEN_SHIP_FACTOR:` The factor to be used in the calculation of `UEID`.

`private byte shipClass:` Class of this alien ship

Constructors:

`public AlienSpaceship():` Default constructor

`UserSpaceship` Class

Attributes:

`private final byte USER_SHIP_FACTOR:` the factor to be used in the calculation of `UEID`.

`private UserSpaceship instance:` Single instance of the User ship.

Constructors:

`private UserSpaceship():`

Methods:

`public UserSpaceship getInstance():` Returns the only instance of the `UserSpaceship`.

➔ Controller Classes

`GameEngine` Class

Attributes:

`private int currentScore:` Momentary score

`private int highScore :` Current high score

`private byte currentStage:` the stage the game is currently at

`private GameEngine instance:` GameEngine's own instance since it is going to be the mere GameEngine.

`private int currentLevel:` The Level the game is currently at

`private UserSpaceship userShip:` The spaceship of the user

Constructors:

`private GameEngine()`

Methods:

`public void initializeNewGame():` Initializes the game from the beginning

`public void pauseGame():` Pauses the game

`public void resumeGame():` Resumes the game

`public void destroyObject(GameObject obj):` destroys the game object specified.

`public void startGame() :` Starts the game

`public void stopGame():` Stops the game

`public void advanceNextLevel():` Advances to the next level

`public void advanceNextStage():` Advances to the next stage

`public GameEngine getInstance():` Returns the only instance of this clas

`public void updateHighscore(int newScore):` Updates the highscore

`public void reduceHealth(Spaceship ship, Ammo amount) :` reduces the spaceship's health by the ammo's damage.

`public Ammo calculateDamage(Spaceship ship) :` Calculates the spaceship's power rate and creates a Ammo, this method is used when a ship shoots.

`public void animateEntities():` Animates GameObjects, merely updates for any changes

`public void updateSettings():` If any change occurs in the settings, update everything

`public void updateCurrentScore(int score):` Adds score amount to the current score.

`CollisionManager` **Class**

Attributes:

`private ArrayList<Ammo> ammoList:` A list of ammos is kept to constantly check whether or not a spaceship and an Ammo have overlapped.

Constructors:

`public CollisionManager():` Default constructor

Methods:

`private boolean checkCollisions(HashMap<String, GameObject> map):` Checks whether or not a collision has occurred. If so returns true.

`public void addAmmo(Ammo anAmmo):` Adds an ammo to the list

`public void deleteAmmo(Ammo anAmmo):` Removes the specified ammo from the list.

`AlienManager` **Class**

Constructors:

`public AlienManager():` Default constructor.

Methods:

`private void updateHealth(AlienSpaceship anAlien):` updates Health of the specified alien.

`private void moveAliens(HashMap<String, GameObject> map):` Moves aliens for the next position.

`private void allocateAliens(Planet currentPlanet):` Allocates alien to the map according to the level and stage of the planet.

`FileManager` **Class**

Attributes:

`private File settingsFile:` Path of the settings file

`private File highScoresFile:` Path of the highscores file

Constructors:

`public FileManager():` Default constructor.

Methods:

`private boolean WriteFile(String content, File file):` Writes to a file.

`private String ReadFile(File file):` Reads from the specified file and returns its contents

`GameMapManager` **Class**

Attributes:

`private HashMap<String, GameObject> entities:` All of the `GameObject` entities are kept in this HashMap.

Constructors:

`private GameMapManager()`

Methods:

`private boolean registerEntity(GameObject obj):` Registers the provided `GameObject`

`private boolean eraseEntity(Strign UEID):` Erases the `GameObject` with the specific UEID.

`private GameObject fetchEntity(String UEID):` Fetches the `GameObject` with the specific UEID.

`private AlienSpaceship fetchRandomAlien():` Returns a random Alien.

`private GameMapManager getInstance():` Returns the only instance of this class.

---

5. References

[1] “Galaga.” Wikipedia. Wikimedia Foundation, n.d. Web. 11 Oct. 2016. (https://en.wikipedia.org/wiki/Galaga)

[2] “Free Sprites.” MillionthVector:. N.p., n.d. Web. 11 Oct. 2016. (http://millionthvector.blogspot.com.tr/p/free-sprites.html)