Bilkent University

Department of Computer Engineering

CS319 - OBJECT-ORIENTED SOFTWARE ENGINEERING

Design Report

Group 1E : Tank Zone

Aybüke Ceren Duran

Hakan Türkmenoğlu

Mert Sezer

Supervisor: Eray Tüzün

TABLE OF CONTENTS

1. Introduction

1.1 Purpose of the system

Tank Zone game is expected to be a portable game which has the purpose of entertaining player by taking them into tank wars. The goal of the player is to destroy all enemies or other enemy teams.

Tank Zone is improved version of ‘diep.io’ game with extra features. Compared to diep.io’ game, the user will have more fun while playing ‘Tank Zone’ game because ‘Tank Zone’ game provides game modes such as climate mode, difficulty mode and team mode, difficulty levels, sound, different power-ups and high-quality tank classes. By these extra features, ‘Tank Zone’ game will address different users to make them enjoy and teach the strategies of the war between tanks.

1.2 Design goals

1.2.1 End User Criteria

• **Ease of use:** For the computer games, it is important to use common controls to increase the ease of use. Menu navigations can be controlled by the mouse pointer. While the gamer is playing ‘Tank Zone’ game, s/he will be able to move the tank in any direction with using 4 keyboard keys, and s/he will be able to decide where to take aim at while shooting by using mouse pointer.
• **Ease of learning:** When the user plays the game for the first time, s/he will not understand how ‘Tank Zone’ game works. So, it is crucial for providing the gamer with a smooth learning tip. ‘Help’ button on the menu navigator stands for this purpose. In addition, by choosing easy-game mode, the gamer can learn the strategies of the war between tanks. After these simple levels, the player will be pushed towards greater challenges to increase the fun factor.

1.2.2 Maintenance Criteria

• **Maintainability:** Maintaining, adding and removing feature is crucial for the lifecycle of a software. We give priority to determine what the game will be and what it will look like. Because of that, the tasks that our system should perform changes easily with every new idea. After the ‘Tank Zone’ is launched, we may think of developing an improved version of the game. So, ‘Tank Zone’ game must be modifiable and we should be able to make changes in the software readily for our future systems.

‘Tank Zone’ game system design favors composition over inheritance which increases flexibility. Therefore, this design architecture reduces the possibility of breaking encapsulation principle in case of changes needed in our design in the future.
Additionally, as a rule of thumb, we have decided that classes should not be more than 300 lines of code to increase partition. This will help ensure that we will achieve high coherence, in turn, increase maintainability of the system.

• **Portability:** ‘Tank Zone’ game will be Java-based game. Therefore, ‘Tank Zone’ game will work in JRE installed platforms.

• **Readability**: Developing a video game is a complex task. Because of this many games start readable but projects without readability as a goal in mind end up in a spaghetti code. For this reason we aim to use as much as design patterns as possible since they are proven to work.

1.2.3 Cost Criteria

• **Deployment cost:** The game will be a freeware since it’s mainly made for fun.

• **Development cost:** Since the game is not a commercial product, it should be as cheap as possible to develop. This is because we won’t have financial security provided by the sales unlike commercial products.

1.2.4 Tradeoffs

•**Usability vs Functionality:** From the beginning to the end, we consider addressing different users such as ones who enjoy the feeling of victory with having hardness, ones who prefer to focus on learning game and its strategies in the easiest level because Tank Zone game is designed for entertainment of a wide target of audience. Therefore, taking ones who like learning game and ones who like challenge into consideration, we must keep the game as simple and understandable. We do not need a high functionality for our game because once we increase the usability, the functionality is decreased as well.

•**Performance vs Memory**: ‘Tank Zone’ game will be designed via Java programming language. We chose Java programming language due to famous for its effectiveness to produce binaries which can run on any processor architecture. However, this situation leads to problem that in all the applications being run in an interpreter called the Java Virtual Machine. When interpreting another perspective, Java will decrease the development time of our project due to having wide range of libraries available. However this has a downside of having higher memory and CPU requirements than using a low-level language like C/C++/Rust. But still we are in favor of this kind of tradeoff.

Similarly, as we will discuss in software architecture chapter in detail, our system is designed to provide optimizations that require higher memory requirements in order to achieve better performance. For instance, a game object that can move might be tracked by a class that is responsible from objects moving and a class responsible from objects being drawn. Conversely, the movement class won't hold references to immovable objects in order to increase performance. So if in total there are 1000 objects and only 10 of them move, the movement class won't have to deal with the other 990 objects which substantially improves performance.

2. Software Architecture

2.1 Subsystem Decomposition

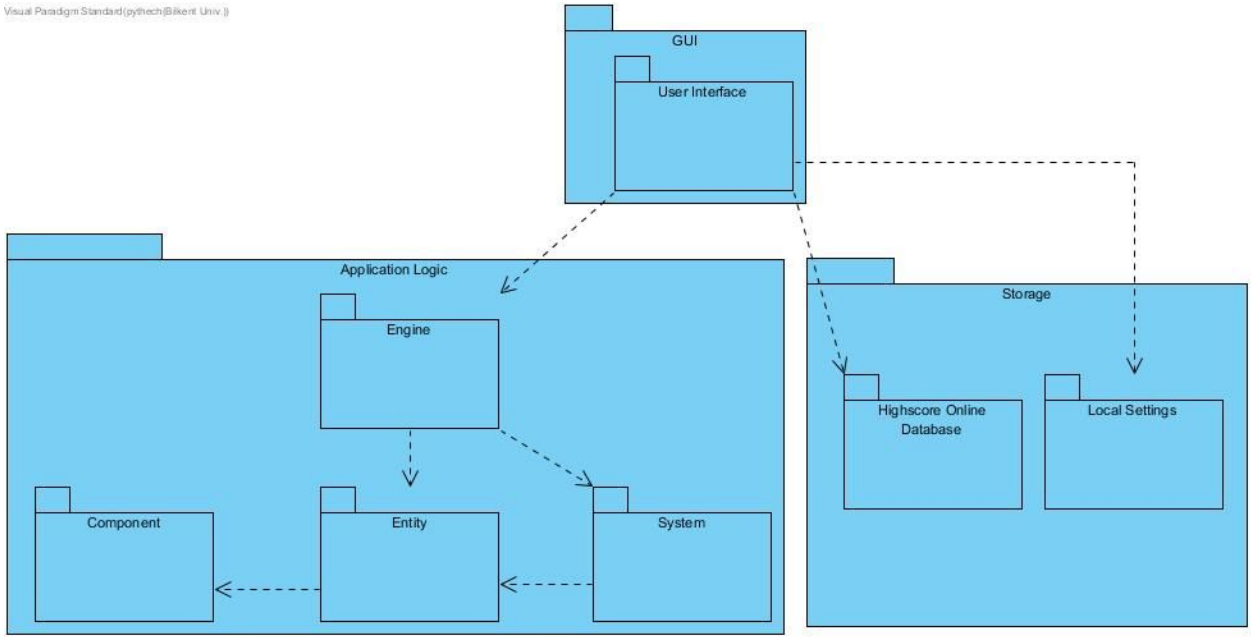


Figure-01: High Level Representation of Subsystem Decomposition

We have chosen three-tier architecture as since it's best suited for our design goals. Specifically, three-tier or n-tier architecture allows any layer to be replaced or upgraded independently. This way, we aim to achieve our portability and maintainability design goals. To give an example, later on if we want to port our game to a platform other than desktop like Android, we can replace our UI layer with something else without major issues. Maintainability is also achieved because we would only have to maintain a single application logic layer and a

single storage layer, instead of writing a lot of platform dependent code and thus maintaining them. The layers are:

- **GUI layer:** Responsible for providing an easy to use interface between the game and the user.
- **Application Logic layer:** Contains the business logic code for game to function.
- **Storage layer:** Responsible for taking care of storing high score information in online database and saving settings.

Additionally, our application logic layer uses “Entity-Component-System” architectural design which is widely used in game development. Popularized by Unity Engine, Entity-Component-System encourages composition over inheritance which greatly increases flexibility that is needed in game design.

- **Component:** Holds raw data without any business logic. It describes how an entity interacts with the world. E.g SpeedComponent, AngleComponent
- **Entity:** A general purpose game object that holds certain components. E.g TankBody, Obstacle
- **System:** Performs business logic on entities that have specific components. For instance, a MovementSystem would only move entities that have SpeedComponent and PositionComponent together.

This design removes ambiguity related with traditional deep and wide inheritance designs. To give an example, let’s say we have decided that GameObjects divide into two groups, StaticObjects and MovableObjects. We thought that each and every entity in our game would have a texture. But later on we have decided that there should be an invisible entity that should actually move. Then we have to redefine inheritance hierarchy. Then we discover that some objects need angles and some don’t. We can add angle to every object and set angle to a sentinel value but that’s wasteful, not all entities need angle property. With this design however, all the combinations of components can be added to an entity without any problems.
One more major advantage of this architecture is that an entity can have a component added or removed in runtime. This fits in our game well since the ‘Tank Zone’ game has powerups to be collected. These powerups can be modeled as components that will be attached to the entities during runtime.
Our design encourages high coherence and low coupling. Adding or removing a component, entity or a system is easy. It’s even possible to do this in runtime, so they are lowly coupled. Each class is highly related to each other inside the subsystem so they have high coherence. We used a closed architecture to increase maintainability and flexibility.

2.2. Hardware/Software Mapping

Software: Our game is developed in Java as the programming language. Therefore, Java Runtime Environment will be needed to run it.

Hardware: In terms of hardware requirements, the player will need a basic mouse and a keyboard.

2.3 Persistent Data Management

2.3.1 Local Settings

In the ‘Tank Zone’ game, the number of passed levels and the number of upgraded tanks will be recorded in a comma separated file. This will be used for Achievement system.

As we described in analysis report, Tank Zone game has sound on/off feature. Sound on/off state will be recorded in a file. If the sound is on, 1 will be recorded in the file. If the sound is off, 0 will be recorded in the file. This state of sound will be remembered even if the player will exit the Tank Zone game.

Additionally, we will store the last used username. This username is used to submit highscores, it would be useful to remember this information in case the user wants to play the game some time later.

2.3.2 Online Highscore Database

Our online database is located in a centralized server accessible with HTTP. The back-end for this database is coded with PHP. Our database of choice is MySQL. The reason we decided to use a database instead of a file management system is that there might be cases where multiple writes may occur at the same. Such facility is easily provided by a database engine without additional effort. We are familiar with PHP and MySQL and since they both are open source projects, they are also free of charge. These two factors led us to this decision. When the user decides to upload their score, a score is calculated and sent to this server via HTTP. Then corresponding tables are updated accordingly. The client (Java/game) can fetch the new top 10 scores.

2.4 Access Control and Security

‘Tank Zone’ game does not provide a login/signup mechanism for users to use. Highscores are identified by the username, therefore multiple users can submit highscores for the same username. However users should not able to modify the online database directly. They can only add new entries to the database system and retrieve them. Therefore the users will have limited write ability.

Similarly, if the player passes a level, the user can choose between increasing MaxHealth, MaxSpeed and so on. These features are not provided in the beginning of the game to the user, so we limit upgrade features according to the level of the player.

2.5 Boundary Conditions

2.5.1 Execution

‘Tank Zone’ requires Java Runtime Environment installation on the players’ computers. The game will be provided as a .jar file.

2.5.2 First Run

'Tank Zone' game does not have the feature of the saved game state. Therefore, the user will have only chance to play game. However it stores local settings in settings.csv file. If this file exists, username and Sound On/Off setting will be retrieved otherwise default values will be inserted.

2.5.3 Termination

While the game is running, if the player decides to press the "ESC" button, the system will display pause menu. Inside the menu, the user can decide to exit the game or continue. Similarly the user can exit the game in the main menu.

2.5.4 Failure

When the game is over, the user can submit their highscore. However if the network connection cannot be established, the program will display an error.

3. System Model

3.1 Libraries

The game was designed with both AWT Swing and LibGDX. LibGDX was used minimally, it provides facilities for drawing images on the screen. Additionally we have used its data structures which is claimed to be better suited for game development. These are:

- **Array** class which corresponds to Java's **ArrayList** class.
- **ObjectMap** class, corresponds to Java's **HashMap**.

These two data structures have almost identical interfaces as those provided by Java but with have additional concerns regarding memory. These help us to achieve higher performance.

Additionally, we also used these classes provided by LibGDX:

- **SpriteBatch:** Used to draw images on the screen.
- **Texture:** Used to hold image related information.
- **OrthographicCamera:** Provides facilities to simulate a camera on a 2D game so that we can have maps bigger than what screen provides.
- **Vector2:** Used to manipulate Vectors in 2D space which holds x and y coordinates.

Finally, the library uses floats for coordinates instead of integers.

3.2 Design Patterns

3.2.1 Façade Pattern

GameManager is a Façade class that care of starting the engine with appropriate map, engine and entities. By providing GameManager class, we abstract away the details of starting the game, making an easy to use interface to start the game.

3.2.2 Observer Pattern

Observer pattern is used in event-driven software. In our system, **Engine** class is the subject and each **System** is an observer. We use observer pattern in threeways, the first one is that in each frame the observers are notified with **update()** method. This way the systems will apply business logic for the next frame. The second one is whenever an entity is added or removed, the systems are notified about this change with **notifySystemsAboutAddedEntity()** and **notifySystemsAboutRemovedEntity()** functions that call **entityUpdated(entity : Entity, added : boolean)** method of each system. Finally, whenever an entity has a component added or removed the same **entityUpdated** method will be called.

3.2.3 Singleton Pattern

World class is a singleton object that holds information about the world the game is running that is shared among all the systems. By using this pattern, we ensure that only a single game/world will be created at a time. Similar reasoning applies to **Storage** class.

3.2.4 Strategy Pattern

Since our game has three difficulties we have decided that we should come up with different AI behaviours for each difficulty. For this reason, AI will use different algorithms depending on the difficulty. To achieve this, we have used strategy pattern in **AISystem** class. AIs can move, target and shoot enemies. They have 2 strategies for each.

- **ShootStraightStrategy:** Shoots directly at the enemy, used in easy and medium difficulties.
- **ShootWithPredictionStrategy:** There is a delay between the time bullet takes off and hits the enemy. Using the speed of the bullet and the enemy, the AI tries land the bullet on enemy instead of naively shooting straight ahead. Used in hard difficulty.
- **MoveStraightStrategy:** AI moves directly to the enemy. Used in easy difficulty.
- **MoveCoordinatedStrategy:** AI moves as a collective unit. Used in medium and hard difficulty.
- **TargetClosestStrategy:** Targets the closest enemy. Used in easy and medium.
- **TargestMostCrowdedStrategy:** Targets the closest enemy by considering the crowds so that it will ignore lonely enemies. Used in hard difficulty.

3.2.5 Marker Interface Pattern

Although components have no common attributes, they do implement the empty Component interface. This gives the compiler the opportunity to check whether added class has the correct type or not. This also is useful for polymorphism.

3.2.6 Factory Method Pattern

EntityFactory class provides alternative constructors for entities. For instance, a tank has two entities, **TankBarrel** and **TankBody** that actually act individually in the game although they are of course related. By providing and easy interface for creating a tank with **createTank()** method for instance, we make creating entities with wrong attributes harder. Also the methods take care of adding the newly created entities into the **Engine** class.

3.3 Subsystems

Full class diagram is displayed in the following page.

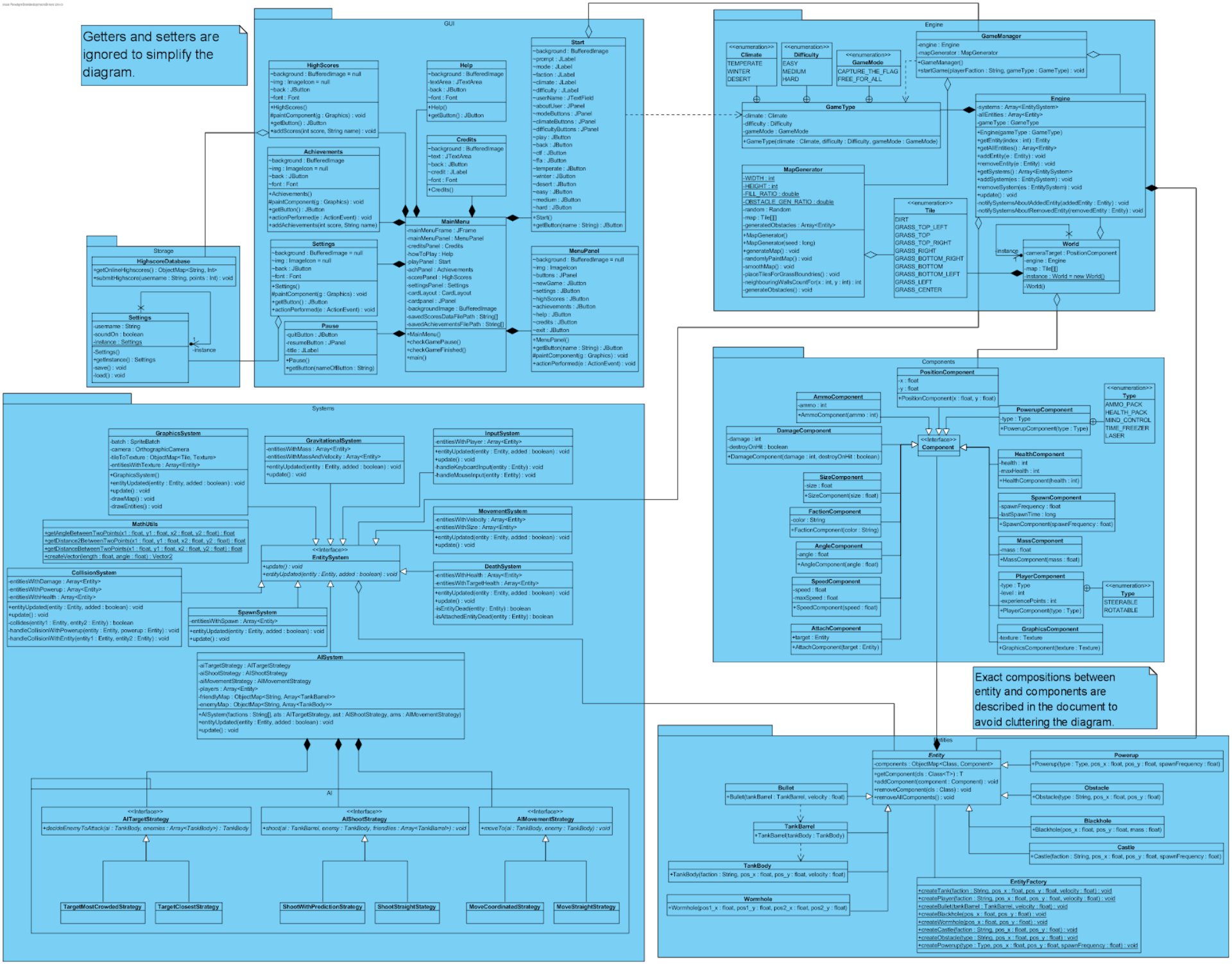

Getters and setters are ignored to simplify the diagram.
GUI
HighScores
-background : BufferedImage = null
-img : ImageIcon = null
-back : JButton
-font : Font
+HighScores()
#paintComponent(g : Graphics) : void
+getButton() : JButton
+addScores(int score, String name) : void
Help
-background : BufferedImage
-textArea : JTextArea
-back : JButton
-font : Font
+Help()
+getButton() : JButton
Start
-background : BufferedImage
-prompt : JLabel
-mode : JLabel
-faction : JLabel
-climate : JLabel
-difficulty : JLabel
-userName : JTextField
-aboutUser : JPanel
-modeButtons : JPanel
-climateButtons : JPanel
-difficultyButtons : JPanel
-play : JButton
-back : JButton
-ctf : JButton
-ffa : JButton
-temperate : JButton
-winter : JButton
-desert : JButton
-easy : JButton
-medium : JButton
-hard : JButton
+Start()
+getButton(name : String) : JButton
Credits
-background : BufferedImage
-text : JTextArea
-back : JButton
-credit : JLabel
-font : Font
+Credits()
Achievements
-background : BufferedImage
-img : ImageIcon = null
-back : JButton
-font : Font
+Achievements()
#paintComponent(g : Graphics) : void
+getButton() : JButton
+actionPerformed(e : ActionEvent) : void
+addAchievements(int score, String name)
MainMenu
-mainMenuFrame : JFrame
-mainMenuPanel : MenuPanel
-creditsPanel : Credits
-howToPlay : Help
-playPanel : Start
-achPanel : Achievements
-scorePanel : HighScores
-settingsPanel : Settings
-cardLayout : CardLayout
-cardpanel : JPanel
-backgroundImage : BufferedImage
-savedScoresDataFilePath : String[]
-savedAchievementsFilePath : String[]
+MainMenu()
+checkGamePause()
+checkGameFinished()
+main()
Settings
-background : BufferedImage = null
-img : ImageIcon = null
-back : JButton
-font : Font
+Settings()
#paintComponent(g : Graphics) : void
+getButton() : JButton
+actionPerformed(e : ActionEvent) : void
Pause
-quitButton : JButton
-resumeButton : JPanel
-title : JLabel
+Pause()
+getButton(nameOfButton : String)
MenuPanel
-background : BufferedImage = null
-img : ImageIcon
-buttons : JPanel
-newGame : JButton
-settings : JButton
-highScores : JButton
-achievements : JButton
-help : JButton
-credits : JButton
-exit : JButton
+MenuPanel()
+getButton(name : String) : JButton
#paintComponent(g : Graphics) : void
+actionPerformed(e : ActionEvent) : void
Storage
HighscoreDatabase
+getOnlineHighscores() : ObjectMap<String, Int>
+submitHighscore(username : String, points : Int) : void
Settings
-username : String
-soundOn : boolean
-instance : Settings
-Settings()
+getInstance() : Settings
-save() : void
-load() : void
1
-instance
Engine
GameManager
-engine : Engine
-mapGenerator : MapGenerator
+GameManager()
+startGame(playerFaction : String, gameType : GameType) : void
<<enumeration>>
Climate
TEMPERATE
WINTER
DESERT
<<enumeration>>
Difficulty
EASY
MEDIUM
HARD
<<enumeration>>
GameMode
CAPTURE_THE_FLAG
FREE_FOR_ALL
GameType
-climate : Climate
-difficulty : Difficulty
-gameMode : GameMode
+GameType(climate : Climate, difficulty : Difficulty, gameMode : GameMode)
Engine
-systems : Array<EntitySystem>
-allEntities : Array<Entity>
-gameType : GameType
+Engine(gameType : GameType)
+getEntity(index : int) : Entity
+getAllEntities() : Array<Entity>
+addEntity(e : Entity) : void
+removeEntity(e : Entity) : void
+getSystems() : Array<EntitySystem>
+addSystem(es : EntitySystem) : void
+removeSystem(es : EntitySystem) : void
+update() : void
-notifySystemsAboutAddedEntity(addedEntity : Entity) : void
-notifySystemsAboutRemovedEntity(removedEntity : Entity) : void
MapGenerator
-WIDTH : int
-HEIGHT : int
-FILL_RATIO : double
-OBSTACLE_GEN_RATIO : double
-random : Random
-map : Tile[][]
-generatedObstacles : Array<Entity>
+MapGenerator()
+MapGenerator(seed : long)
+generateMap() : void
-randomlyPaintMap() : void
-smoothMap() : void
-placeTilesForGrassBoundries() : void
-neighbouringWallsCountFor(x : int, y : int) : int
-generateObstacles() : void
<<enumeration>>
Tile
DIRT
GRASS_TOP_LEFT
GRASS_TOP
GRASS_TOP_RIGHT
GRASS_RIGHT
GRASS_BOTTOM_RIGHT
GRASS_BOTTOM
GRASS_BOTTOM_LEFT
GRASS_LEFT
GRASS_CENTER
instance
World
-cameraTarget : PositionComponent
-engine : Engine
-map : Tile[][]
-instance : World = new World()
-World()
Components
PositionComponent
-x : float
-y : float
+PositionComponent(x : float, y : float)
AmmoComponent
-ammo : int
+AmmoComponent(ammo : int)
PowerupComponent
-type : Type
+PowerupComponent(type : Type)
<<enumeration>>
Type
AMMO_PACK
HEALTH_PACK
MIND_CONTROL
TIME_FREEZER
LASER
DamageComponent
-damage : int
-destroyOnHit : boolean
+DamageComponent(damage : int, destroyOnHit : boolean)
<<Interface>>
Component
HealthComponent
-health : int
-maxHealth : int
+HealthComponent(health : int)
SizeComponent
-size : float
+SizeComponent(size : float)
SpawnComponent
-spawnFrequency : float
-lastSpawnTime : long
+SpawnComponent(spawnFrequency : float)
FactionComponent
-color : String
+FactionComponent(color : String)
MassComponent
-mass : float
+MassComponent(mass : float)
AngleComponent
-angle : float
+AngleComponent(angle : float)
PlayerComponent
-type : Type
-level : int
-experiencePoints : int
+PlayerComponent(type : Type)
<<enumeration>>
Type
STEERABLE
ROTATABLE
SpeedComponent
-speed : float
-maxSpeed : float
+SpeedComponent(speed : float)
GraphicsComponent
-texture : Texture
+GraphicsComponent(texture : Texture)
AttachComponent
+target : Entity
+AttachComponent(target : Entity)
Systems
GraphicsSystem
-batch : SpriteBatch
-camera : OrthographicCamera
-tileToTexture : ObjectMap<Tile, Texture>
-entitiesWithTexture : Array<Entity>
+GraphicsSystem()
+entityUpdated(entity : Entity, added : boolean) : void
+update() : void
-drawMap() : void
-drawEntities() : void
GravitationalSystem
-entitiesWithMass : Array<Entity>
-entitiesWithMassAndVelocity : Array<Entity>
+entityUpdated(entity : Entity, added : boolean) : void
+update() : void
InputSystem
-entitiesWithPlayer : Array<Entity>
+entityUpdated(entity : Entity, added : boolean) : void
+update() : void
-handleKeyboardInput(entity : Entity) : void
-handleMouseInput(entity : Entity) : void
MathUtils
+getAngleBetweenTwoPoints(x1 : float, y1 : float, x2 : float, y2 : float) : float
+getDistanceBetweenTwoPoints(x1 : float, y1 : float, x2 : float, y2 : float) : float
+getDistanceBetweenTwoPoints(x1 : float, y1 : float, x2 : float, y2 : float) : float
+createVector(length : float, angle : float) : Vector2
MovementSystem
-entitiesWithVelocity : Array<Entity>
-entitiesWithSize : Array<Entity>
+entityUpdated(entity : Entity, added : boolean) : void
+update() : void
<<Interface>>
EntitySystem
+update() : void
+entityUpdated(entity : Entity, added : boolean) : void
CollisionSystem
-entitiesWithDamage : Array<Entity>
-entitiesWithPowerup : Array<Entity>
-entitiesWithHealth : Array<Entity>
+entityUpdated(entity : Entity, added : boolean) : void
+update() : void
-collides(entity1 : Entity, entity2 : Entity) : boolean
-handleCollisionWithPowerup(entity : Entity, powerup : Entity) : void
-handleCollisionWithEntity(entity1 : Entity, entity2 : Entity) : void
DeathSystem
-entitiesWithHealth : Array<Entity>
-entitiesWithTargetHealth : Array<Entity>
+entityUpdated(entity : Entity, added : boolean) : void
+update() : void
-isEntityDead(entity : Entity) : boolean
-isAttachedEntityDead(entity : Entity) : boolean
SpawnSystem
-entitiesWithSpawn : Array<Entity>
+entityUpdated(entity : Entity, added : boolean) : void
+update() : void
AISystem
-aiTargetStrategy : AITargetStrategy
-aiShootStrategy : AIShootStrategy
-aiMovementStrategy : AIMovementStrategy
-players : Array<Entity>
-friendlyMap : ObjectMap<String, Array<TankBarrel>>
-enemyMap : ObjectMap<String, Array<TankBody>>
+AISystem(factions : String[], ats : AITargetStrategy, ast : AIShootStrategy, ams : AIMovementStrategy)
+entityUpdated(entity : Entity, added : boolean) : void
+update() : void
AI
<<Interface>>
AITargetStrategy
+decideEnemyToAttack(ai : TankBody, enemies : Array<TankBody>) : TankBody
<<Interface>>
AIShootStrategy
+shoot(ai : TankBarrel, enemy : TankBody, friendlies : Array<TankBarrel>) : void
<<Interface>>
AIMovementStrategy
+moveTo(ai : TankBody, enemy : TankBody) : void
TargetMostCrowdedStrategy
TargetClosestStrategy
ShootWithPredictionStrategy
ShootStraightStategy
MoveCoordinatedStrategy
MoveStraightStrategy
Exact compositions between entity and components are described in the document to avoid cluttering the diagram.
Entities
Entity
-components : ObjectMap<Class, Component>
+getComponent(cls : Class<T>) : T
+addComponent(component : Component) : void
+removeComponent(cls : Class) : void
+removeAllComponents() : void
Powerup
+Powerup(type : Type, pos_x : float, pos_y : float, spawnFrequency : float)
Bullet
+Bullet(tankBarrel : TankBarrel, velocity : float)
Obstacle
+Obstacle(type : String, pos_x : float, pos_y : float)
TankBarrel
+TankBarrel(tankBody : TankBody)
Blackhole
+Blackhole(pos_x : float, pos_y : float, mass : float)
Castle
+Castle(faction : String, pos_x : float, pos_y : float, spawnFrequency : float)
TankBody
+TankBody(faction : String, pos_x : float, pos_y : float, velocity : float)
Wormhole
+Wormhole(pos1_x : float, pos1_y : float, pos2_x : float, pos2_y : float)
EntityFactory
+createTank(faction : String, pos_x : float, pos_y : float, velocity : float) : void
+createPlayer(faction : String, pos_x : float, pos_y : float, velocity : float) : void
+createBullet(tankBarrel : TankBarrel, velocity : float) : void
+createBlackhole(pos_x : float, pos_y : float) : void
+createWormhole(pos_x : float, pos_y : float) : void
+createCastle(faction : String, pos_x : float, pos_y : float) : void
+createObstacle(type : String, pos_x : float, pos_y : float) : void
+createPowerup(type : Type, pos_x : float, pos_y : float, spawnFrequency : float) : void

Figure-02: Subsystems of Tank Zone Game

Available in http://i.hizliresim.com/Plgl5v.png

3.3.1 User Interface Subsystem

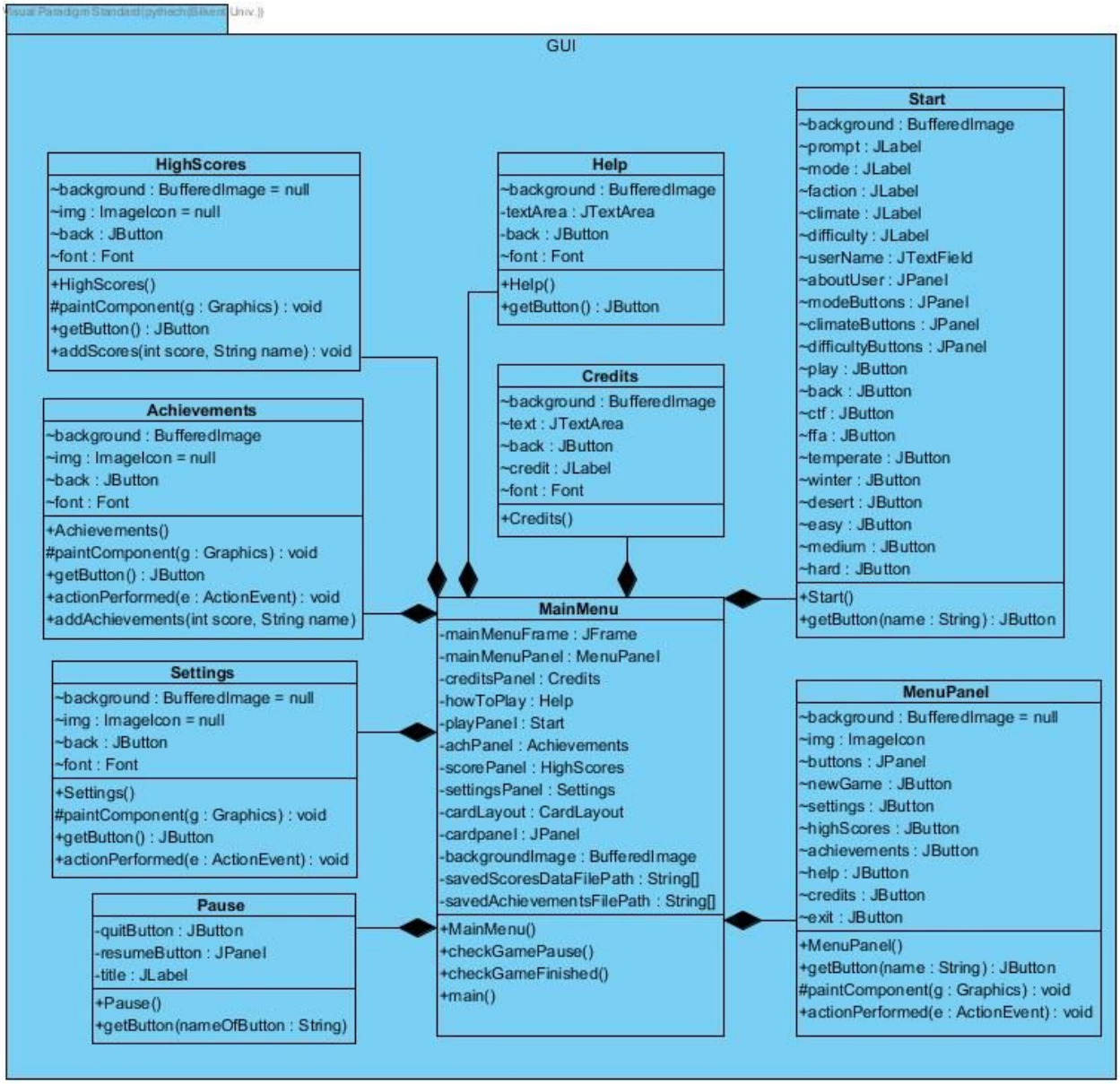


Figure-03: User Interface Subsystem

This subsystem includes the classes which have the feature of the interface of the game. It includes 8 classes which are HighScores, Achievements, Settings,Help, Pause,MenuPanel, Start and Credits. MainMenu is the class that uses all the panels.

MainMenu Frame

Attributes:

-**mainMenuFrame: JFrame**

It is a frame that will display the MainMenu contents.

-creditsPanel: Credits

It is a panel that will display the developers of Tank Zone game.

-howToPlay: Help

It is a panel that includes a text of the description of Tank Zone game.

-playPanel: Start

It is a panel that will display the contents of start options for users.

-achPanel: Achievements

It is a panel that will display the high scores of the user

-scorePanel: HighScores

It is a panel that will display the high scores of the users from all over the world.

-settingsPanel: Settings

It is a panel that provides users sound on/off option.

-cardLayout: CardLayout

It is a CardLayout to provide the transition among all panels of the game which are put in one frame. It is put into cardPanel which is a JPanel.

-cardPanel: JPanel

It is a JPanel which holds all panels that are added into. It shifts the panels once their buttons are clicked on.

-backgroundImage: BufferedImage

It is an image which will be seen on the background of the menu.

-savedScoresDataFilePath: String[][]

It is an array to hold the high scores of the given user.

-savedAchievementsFilePath: String[]

It is an array to hold the top scores.

Constructor: MainMenu() is the default constructor.

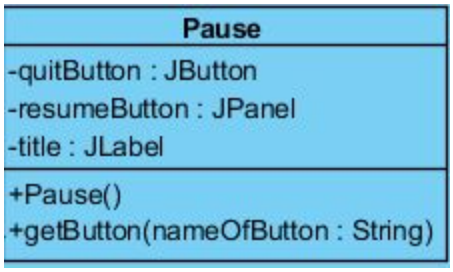
Methods:

main: it is a main method to initiate game.

checkGamePause(): returns true if the game is paused and open the PauseGame panel.

checkGameFinished(): checks if the game is finished.

Pause Panel



Attributes:

-quitButton: JButton

It is a JButton which provides functionality of prompting user to menu.

-resumeButton: JButton

It is a JButton which returns the game from the pause panel.

-title: JLabel

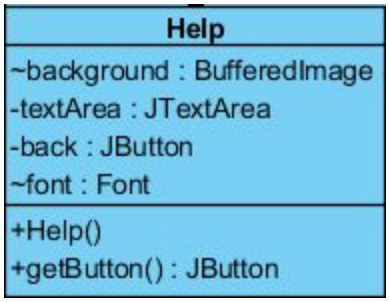
It is a JLabel which asks the player to quit game.

Constructor: Pause() is a default constructor which creates a pop-up that indicates the game is paused, resume button and return to menu.

Methods:

+getButton(nameOfButton: String) : this method takes the button of this panel as a String parameter, and this parameter is used by ActionListener. ActionListener will decide which panel to transform player as the player clicks on one of the buttons on Pause panel.

Help Panel



Attributes:

-background: BufferedImage

It is an image which will be seen on the background of the help panel.

-textArea: JTextArea

It is a text area which includes a text area which provides information about how to play game.

-back: JButton

It is a JButton which transfers the player to main menu.

-font: Font

It is a Font which arranges the style of the text of how to play game.

Constructor: Help() is a default constructor which initializes the help text, image and button.

Methods:

+getButton(): returns back button.

Start Panel

Attributes:

-background: BufferedImage

It is an image which will be seen on the background of the start panel.

-prompt: JLabel

It is a label that prompts user with entering name message.

- mode: JLabel

It is a label that includes mode title.

-faction: JLabel

It is a label that includes faction title.

-climate: JLabel

It is a label that includes climate title.

-difficulty: JLabel

It is a label that includes difficulty title.

-userName : JTextField

It is a JTextField that provides user an area for entering his/her name.

-aboutUser: JPanel

It is a panel that holds prompt and userName.

-**play: JButton**

It is a JButton which initiates the game.

-back: JButton

It is a JButton which transfers the player to main menu.

-ctf: JButton

It is a JButton which arranges the mode of game into capture the flag mode.

-ffa:JButton

It is a JButton which arranges the mode of game into free for all mode.

-temperate: JButton

It is a JButton which arranges the climate of game into temperate.

-winter: JButton

It is a JButton which arranges the climate of game into winter.

-desert: JButton

It is a JButton which arranges the mode of game into desert.

-easy: JButton

It is a JButton which arranges the difficulty of game into easy.

-medium: JButton

It is a JButton which arranges the difficulty of game into medium.

-hard: JButton

It is a JButton which arranges the difficulty of game into hard.

-modeButtons: JPanel

It is a JPanel that holds mode buttons.

-climateButtons: JPanel

It is a JPanel that holds climate buttons.

-difficultyButtons: JPanel

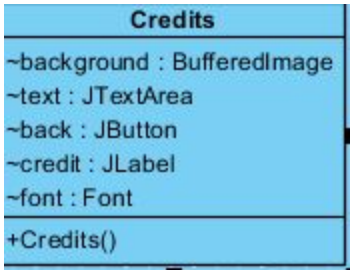
It is a JPanel that holds difficulty buttons.

Constructor: Start() is a default constructor that initializes game.

Methods:

+getButton(nameOfButton: String): this method takes the button of this panel as a String parameter, and this parameter is used by ActionListener. ActionListener will decide which mode, climate and difficulty the game will be.

Credits Panel



Attributes:

-background: BufferedImage

It is an image which will be seen on the background of the credits panel.

-text: JTextArea

It is a text area which includes a text area which provides information about developers of the game.

-back: JButton

It is a JButton which transfers the player to main menu.

-credit: JLabel

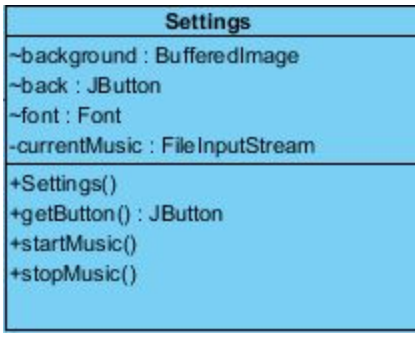
It is a label that includes credits title.

-font: Font

It is a Font which arranges the style of the text of credits.

Constructor: Credits() is a default constructor that initializes credits panel.

Settings Panel



Attributes:

-background: BufferedImage

It is an image which will be seen on the background of the settings panel.

-back: JButton

It is a JButton which transfers the player to main menu.

-font: Font

It is a Font which arranges the style of the text of settings.

-currentMusic: FileInputStream

music on the background of the game.

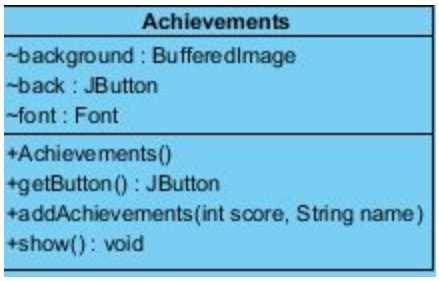
Constructor: Settings() is a default constructor that initializes settings panel.

Methods:.

+startMusic(): This method makes music play.

+stopMusic(): This method makes music stop.

Achievements Panel



Attributes:

-background: BufferedImage: It is an image which will be seen on the background of the achievements panel.

-back: JButton : It is a JButton which transfers the player to main menu.

-font: Font: It is a Font which arranges the style of the text of achievements.

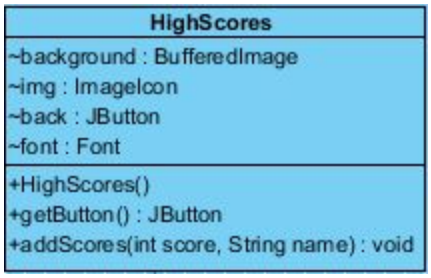
Constructor: Achievements() is a default constructor that initializes achievements panel.

Methods:

+addAchievements(int score, String name): This method allows new achievements added to be system according to score and name parameters.

+show(): This method allows the achievements displayed on the Achievements panel.

HighScores Panel



Attributes:

-background: BufferedImage: It is an image which will be seen on the background of the highscores panel.

-back: JButton -> It is a JButton which transfers the player to main menu.

-**font: Font:** It is a Font which arranges the style of the text of highscores.

Constructor: HighScores() is a default constructor that initializes highscores panel.

Methods:

+addScores(int score, String name): This method allows new highscores added to the system according to score and name parameters.

3.3.2 Engine Subsystem

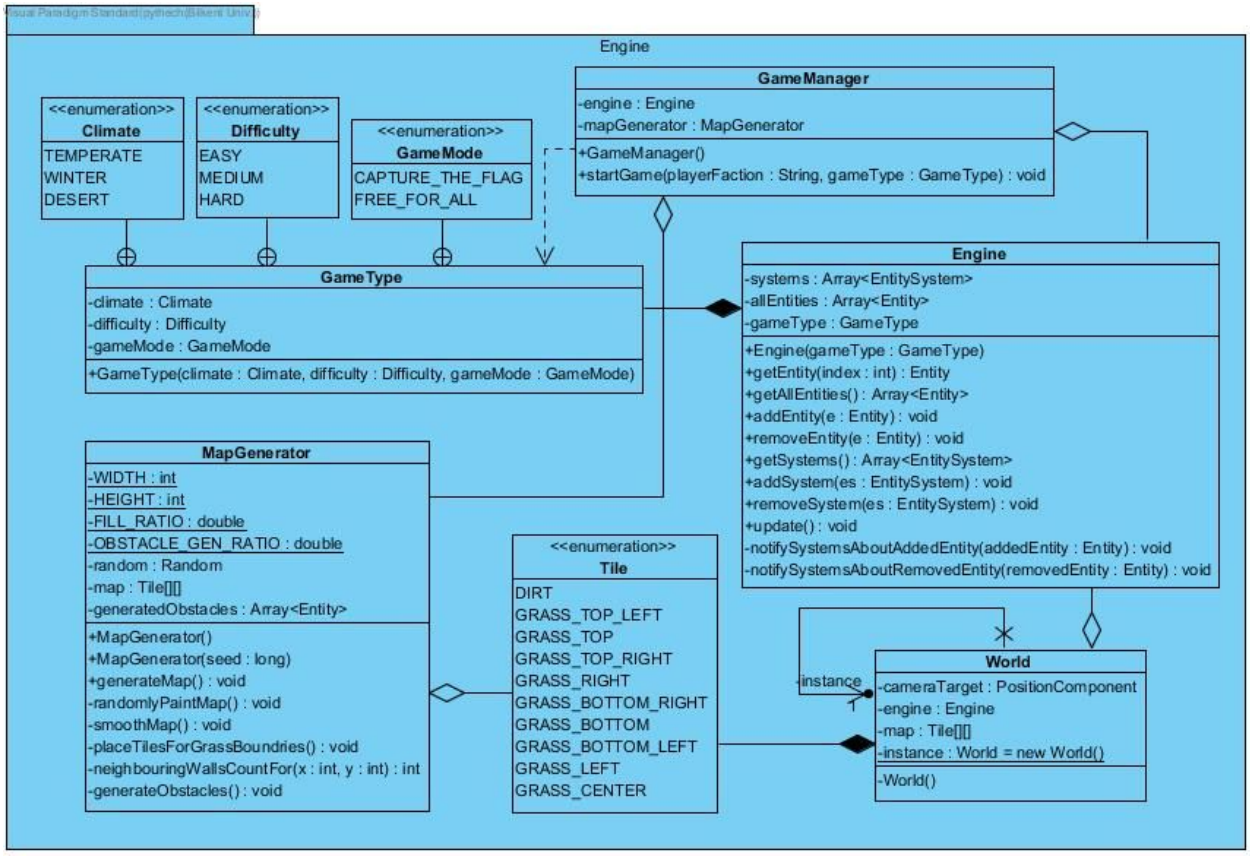
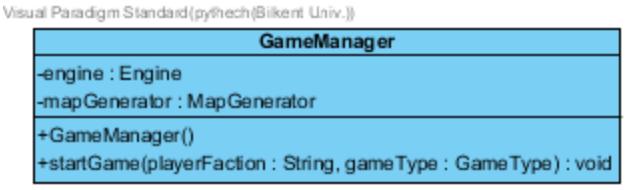


Figure-04: Engine Subsystem of Tank Zone Game

The figure illustrates the overall composition of the Engine Subsystem. This subsystem is responsible for creating game types such as game mode, game difficulty and game climate, and game loop is also started in this subsystem. The updating of Tank Zone game entities are also done in this subsystem, more specifically in Engine class. The map is generated, a set of entities and systems are added into the Engine using GameManager Façade class upon request and the game starts.

GameManager Class:

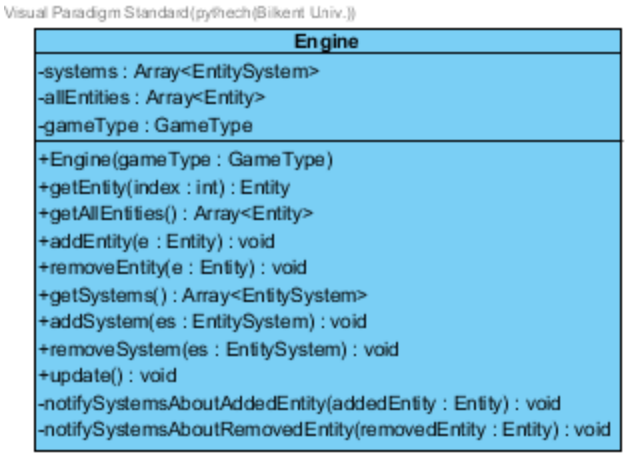


GameManager is a Façade class that takes care of the complexities of generating a map, creating entities according to the GameType and finally inserting EntitySystems into the Engine and World class.

Attributes:

-enige : Engine -> Stores an Engine instance.
-mapGenerator : MapGenerator -> Stores a MapGenerator instance to generate a map suited for the game type.
Constructor:
+GameManager() -> Initializes the attributes of the GameManager class.
Methods:
+startGame(playerFaction : String, gameType : GameType): void -> Prepares a game instance by creating necessary map according to the game type and starts the game.

Engine Class:

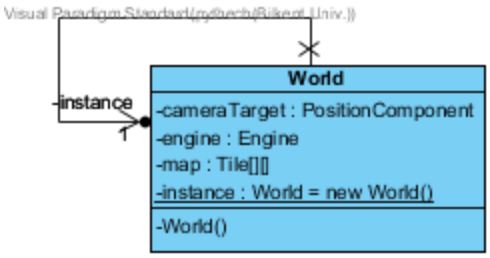


This class is responsible for managing the entities and systems (like addition, removal of them) of the game.
Attributes: -systems: Array<EntitySystem> -> this attribute stores all entity systems of the game(entity systems are the systems like CollisionSystem, InputSystem)
-allEntities: Array<Entity> -> this attribute stores all entities in the game

Methods:
+update(): void -> Updates all the systems per each frame.
+getEntity(index: int): Entity -> returns the entity at the specified index.
+addEntity(e : Entity): void -> this method adds Entity e to allEntities array.
+removeEntity(e: Entity): void -> this method removes Entity e from allEntities array
+addSystem(es: EntitySystem): void -> this method adds system es to EntitySystem array
+removeSystem(es: EntitySystem): void -> this method removes system es from EntitySystem array
-notifySystemsAboutAddedEntity(addedEntity: Entity): void -> when entity addedEntity is added, for all entitySystems(es), this method calls es.entityUpdated(addedEntity, true). Notifies systems about the added entity.

-notifySystemsAboutRemovedEntity(removedEntity: Entity): void -> when entity removedEntity is removed, for all entitySystems(es), this method calls es.entityUpdated(removedEntity, false). Notifies systems about the removed entity.

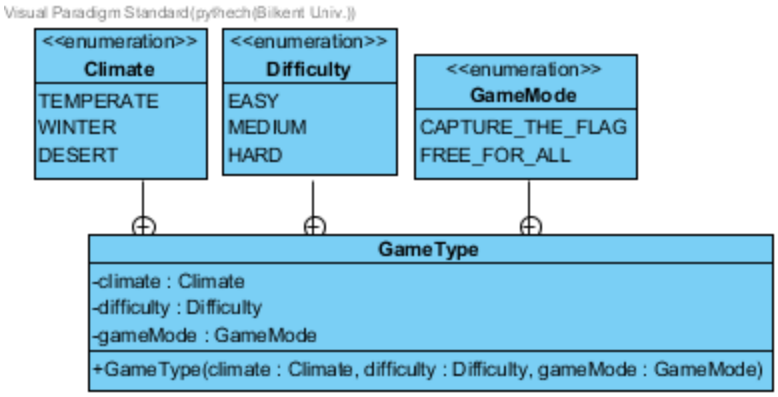
World Class:



This is a Singleton class used to store information about the world generated.
Attributes:
-cameraTarget: PositionComponent -> Current entity that the camera focuses on.
-engine: Engine -> game engine that is responsible for creating entities and entity systems
-map: Tile[][] -> 2 dimensional array that stores all tiles.

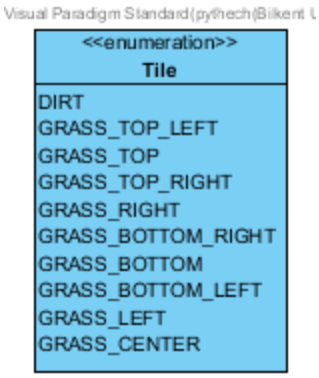
Constructor:
-World() -> It constructs the class World

Game Type Class:



GameType stores information related about the preferences about the game that is going to be played. These options are selected in the user interfacelayer.
Attributes:
-climate: Climate -> can be temperate, winter, desert
-difficulty: Difficulty -> can be easy, medium, hard
-mode: Mode -> can be capture_the_flag, free_for_all

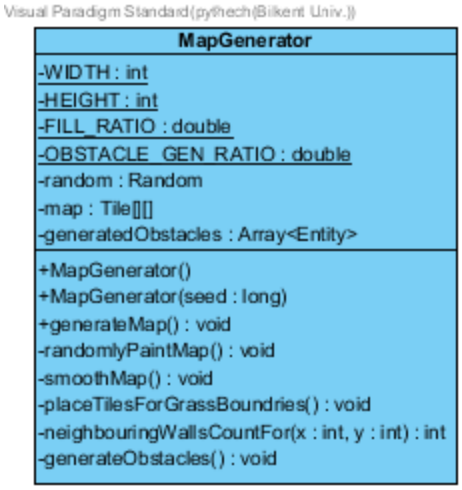
Tile Class



Tile enumeration has DIRT attribute and attributes for the grass center(GRASS_CENTER) and grass boundaries like GRASS_TOP_LEFT. We have chosen ‘Temperate’ climate as the default but the same values can be used for different climates. For instance, GRASS acts as SNOW in the Winter climate and SAND in the desert climate. Obstacles are generated inside the GRASS tiles while the DIRT tile is left empty so that the user will have challenges navigating inside the GRASS tiles.

Attributes: **DIRT** -> dirty looking grass. This attribute distinguishes the looking of the grass
GRASS_TOP_LEFT -> the grass shape for top left of grass
GRASS_TOP ->the grass shape for top of grass
GRASS_TOP_RIGHT ->the grass shape for top right of grass
GRASS_RIGHT ->the grass shape for right of grass
GRASS_BOTTOM_RIGHT ->the grass shape for bottom right of grass
GRASS_BOTTOM ->the grass shape for bottom of grass
GRASS_BOTTOM_LEFT ->the grass shape for bottom left of grass
GRASS_LEFT ->the grass shape for left of grass
GRASS_CENTER ->the grass shape for center of grass

MapGenerator Class



Attributes: **-WIDTH: int** -> width of the map
-HEIGHT: int -> height of the map
-FILL RATIO: double -> the ratio of tile dirt to all tiles in a map
-OBSTACLE_GEN_RATIO: double -> fullness ratio (obstacle ratio in map)
-random: Random -> random number generator
-map: Tile[][] -> Map consist of tiles with (x, y) positions. Also Map[width][height] is a 2 dimensional array that stores these tiles.
-generatedObstacles: Array<Entity>-> the array of entity obstacles

Constructor: **MapGenerator()** -> The class draws the map randomly
MapGenerator(seed:long) -> Again the class draws the map randomly(but the random map is determined according to seed value)

Methods:

+generateMap(): void -> This method creates and assigns 2 dimensional tile array[width][height] to the variable map. Initially every tile is a GRASS_CENTER. Calls the following methods to generate the map.

-randomlyPaintMap(): void -> This method assigns Tile.DIRT to the tiles randomly according to the FILL_RATIO.

-smoothMap(): void -> This method makes the map smooth by connecting close tiles.

- placeTilesForGrassBoundaries(): void -> This method replaces GRASS_CENTER tiles with GRASS_TOP, GRASS_RIGHT, GRASS_BOTTOM etc. if they are located in the boundaries.

-neighbouringWallsCountFor(x: int, y: int): int -> Returns the number of DIRT tiles neighbouring a given x,y coordinate.

- generateObstacles(): void -> This method generates entity obstacles inside GRASS tiles.

3.3.3 Component Subsystem

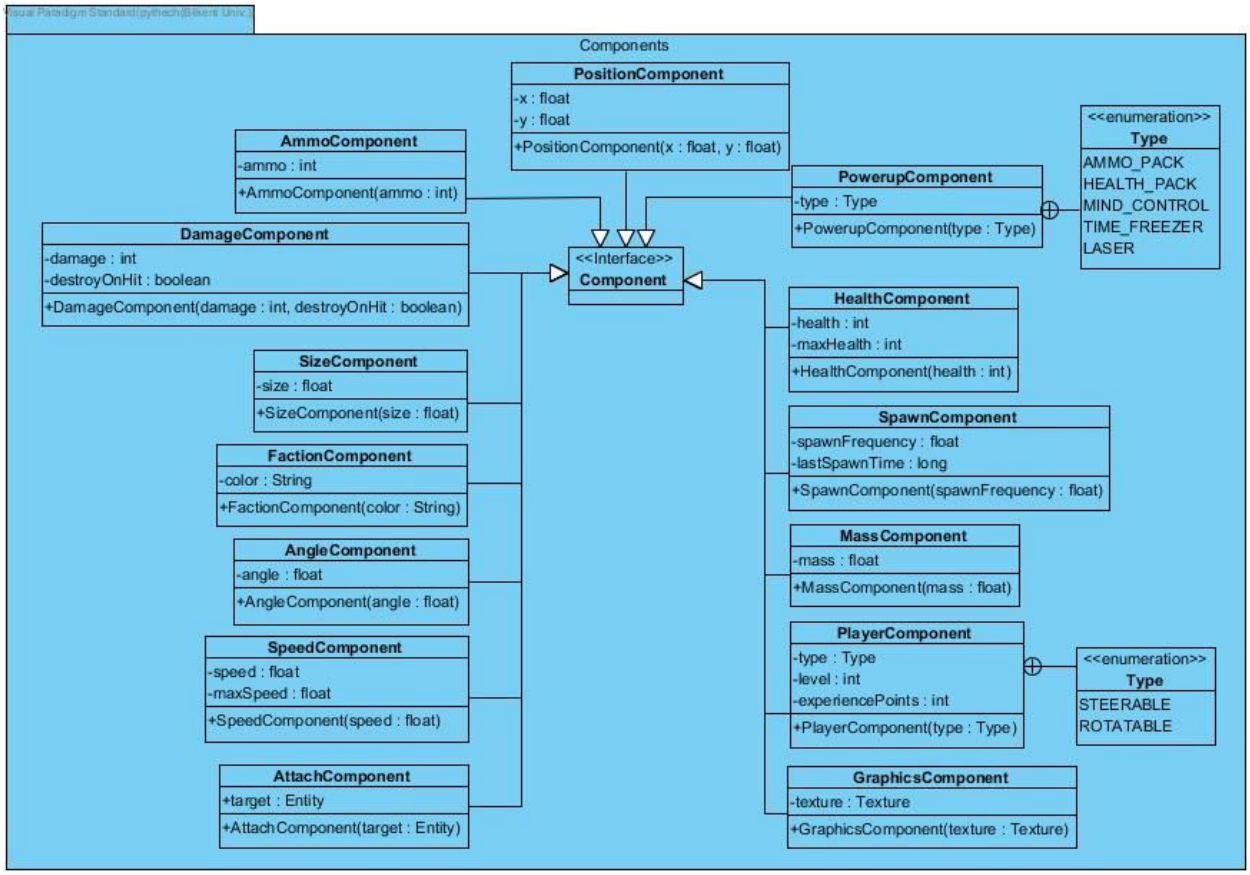
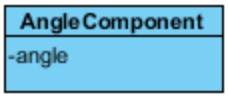


Figure-05: Component Subsystem of Tank Zone Game

This subsystem is responsible for handling and creating the components.

AngleComponent Class:

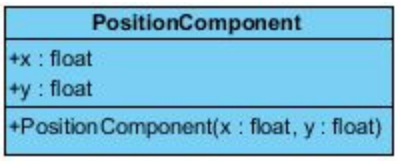


Angle component class implements Component class and has only 1 attribute: angle. Angles are described in degrees and 0 degree means upwards, 90 degrees mean right and so on.

Attributes:

-angle: the value of angle(degree)

PositionComponent Class:

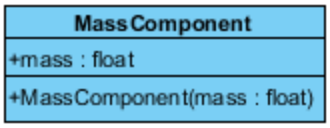


Position component class implements Component class and has 2 variables: (0,0) coordinate is the origin, located in lower-left corner.

Attributes: **+x: float** -> x coordinate
+y: float -> y coordinate

Constructor: It initializes instances(x and y) of PositionComponent object.

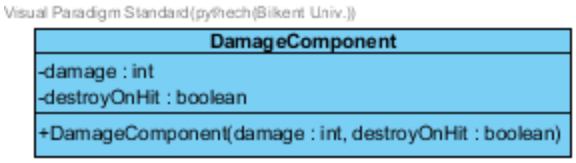
MassComponent Class:



MassComponent implements Component class and is also related to gravity and also has 1 variable. Some entities like TankBody, Blackhole have MassComponent. For example, TankBody's mass is too small and therefore its' gravitation is almost ignorable, whereas Blackhole's mass and gravity is too big compared to TankBody, thus in turn it pulls the nearest tanks to itself. Mass is expressed in kilograms.

Attributes:
+mass: float -> The value of mass of an entity.
Constructor:
+MassComponent(mass: float) -> It initializes the instance of MassComponent object.

DamageComponent Class:

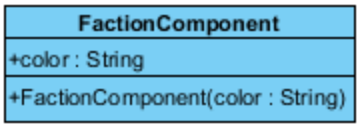


Damage component class implements Component class and handles the amount of damage that the entities can give. Also some entities are destroyOnHit(being destroyed when hits) and some are not destroyOnHit. For example, blackholes and obstacles have damage; but they are not destroyOnHit. Bullet also has damage and it is destroyOnHit.

Attributes: **+damage: int** -> the amount of damage that that some entities can give.
+destroyOnHit: boolean -> destroyOnHit(disposable) or not

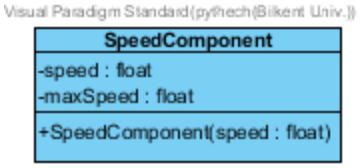
Constructor: +DamageComponent(damage: int, destroyOnHit: boolean) -> takes 2 parameters and initializes the 2 attributes.(damage, destroyOnHit)

FactionComponent Class:



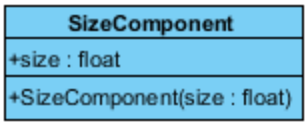
FactionComponent class implements Component class, TankBody and TankBarrel have FactionComponent and FactionComponent has only 1 attribute: color. This feature is in capture the flag mode and it is to distinguish the teams. Different teams' TankBody's and TankBarrel's have different faction components (different colors)
Attributes:
+color: String -> the color
Constructor:
+FactionComponent(color: String) -> It initializes the color instance of FactionComponent.

SpeedComponent Class:



SpeedComponent implements Component class and has only 1 attribute.
Attributes:
-speed: float -> This attribute can be thought as the speed of an entity in any angle, so is a scalar magnitude.
-maxSpeed: float -> When the keyboard key for moving in any direction is pressed, the velocity increases and becomes closer to maximum velocity as delta time increases(the time during Input Key is pressed)
Constructor: +SpeedComponent(speed: float) -> It initializes speed value of the SpeedComponent.

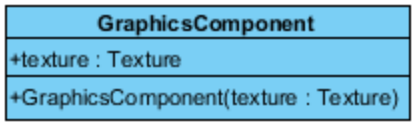
SizeComponent Class:



SizeComponent implements component class and has only 1 attribute: size. Note that the size of an entity is calculated with using its' width and height. Size is used to check whether entities collide or not. Collision system uses circular collision detection, so size is actually the radius of the given entity.

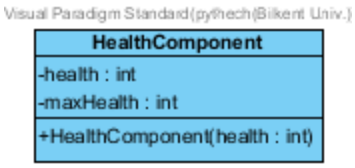
Attributes:
+size: float -> the size/radius of an entity.
Constructor:
+sizeComponent(size: float) -> It initializes size instance of a sizeComponent.

GraphicsComponent Class:



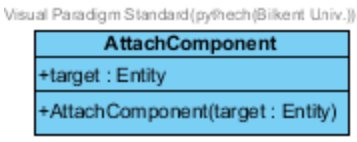
GraphicsComponent implements Component class and has only 1 attribute: texture
Attributes: +texture -> texture of an entity
Constructor: +GraphicsComponent(texture: Texture) -> It initializes the instance texture of the GraphicsComponent.

HealthComponent Class:



HealthComponent class implements component class and some entities like TankBody have HealthComponent. Entities that have HealthComponent have health value. For example TankBody's health value decreases when there is collision between tanks and blackholes, tanks and obstacles or tanks and bullets.
Attributes:
-health: int -> the health value of an entity
-maxHealth: int -> Maximum health(initial health of an entity). For example, TankBody's maximum health(initial health) is 100.
Constructor:
+HealthComponent(health: int) -> It initializes instance of a HealthComponent object.

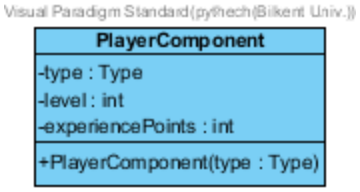
AttachComponent Class:



AttachComponent class implements component class and it defines where the TankBarrel is attached to.
Attributes: +target: Entity -> The target entity the entity is attached to

Constructor: +AttachComponent(target: Entity) -> It initializes the instance of AttachComponent class.

PlayerComponent Class:



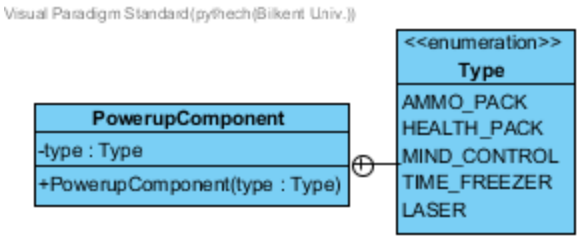
Attributes:

- **type: Type** -> steerable or rotatable
- **-level: int:** -> the level of the game; which increases according to experincePoints
- **- experiencePoints: int** -> this variable increases if the player is successful; for example if he/she destroys enemy TankBody

Constructor:

+PlayerComponent(type: Type) Player component class that takes the type parameter can be of two types: steerable, rotatable. Note that when creating a player, we are adding to TankBody the Steerable, adding to TankBarrel the Rotatable PlayerComponent type. Also rotatable type(tankbarrel) can be controlled with mouse, whereas steerable(tankbody) type can be controlled with keyboard keys.

PowerupComponent Class:



PowerupComponents are used in two ways. Each powerup is collected by moving over to the corresponding Powerup entity in the battlefield. When the user collects these powerups, this component is added to the collecting tank entity at runtime.

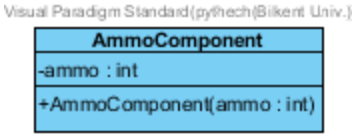
Attributes:

-type: Type -> the type of power-up(ammo_pack, health_pack, mind_control, time_freezer or laser)

Constructor:

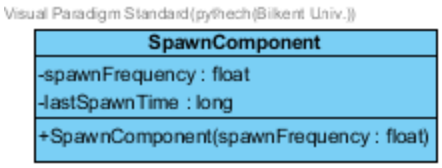
+ PowerupComponent(type: Type) This constructor determines the instance(type) of the PowerupComponent.

AmmoComponent Class:



Tanks have limited ammunition and they can be obtained by fighting the enemies. This component stores the current ammo left inside the tank.
Attributes:
-ammo: int -> Tank's current ammo.
Constructor:
+AmmoComponent(ammo: int)

SpawnComponent Class:



Certain entites can spawn according to the spawnFrequency. For instance, some powerups can re-generate every 60 seconds. Similarly, tanks will be created inside Castles.
Attributes:
-spawnFrequency: float -> Spawn frequency expressed in seconds.
-lastSpawnTime: long -> Used to check whether an entity should spawn by comparing it to the current time and spawnFrequency. When an entity spawns, this attribute is set to current time.
Constructor:
+SpawnComponent(spawnFrequency: float) ->

3.3.4 Entity Subsystem

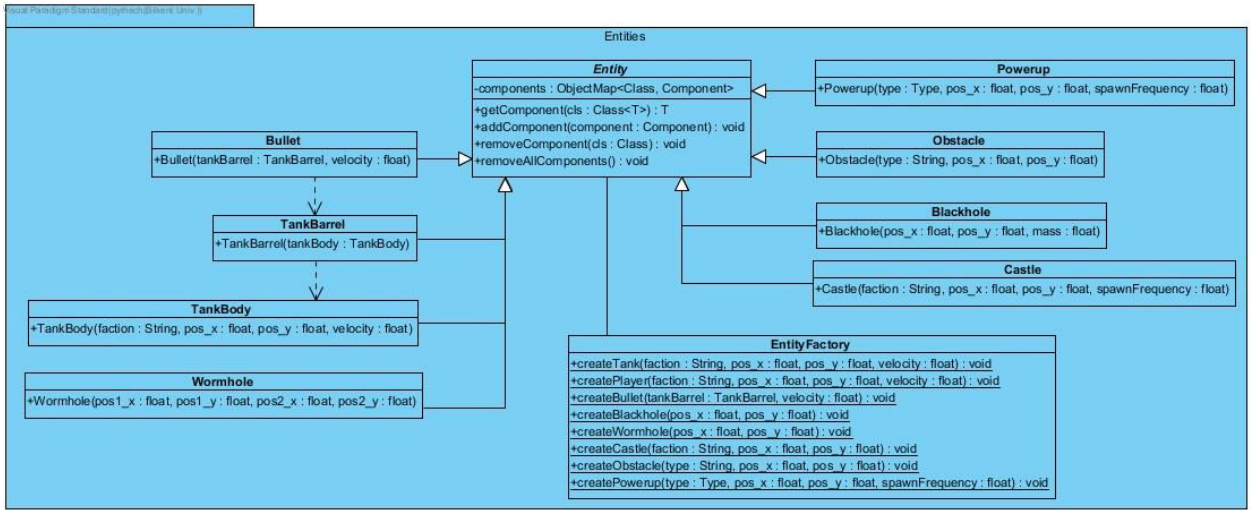
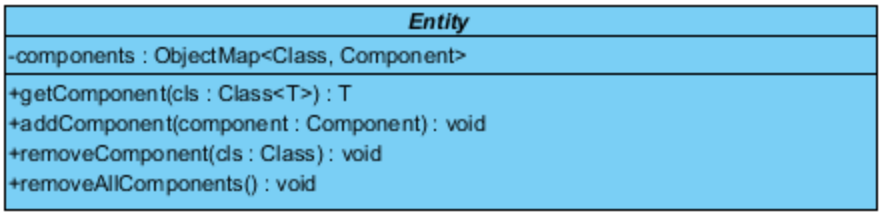


Figure-06: Entity Subsystem of Tank Zone Game

The figure above illustrates the overall composition of the Entity Subsystem.This subsystem is responsible for creating and handling the main entities of Tank Zone game such as tanks in game, player's tank,castle, blackhole and obstacle. In this subsystem, the positions, sizes, angles, textures of the entities are set. Adding of the components of the entities is done in this subsystem. By adding components to entities, we use composition based design.

Entity Class:



Entity is an abstract class and represents the game entities and is later implemented by this game entities like obstacle, TankBody

Attributes: -components: ObjectMap<Class, Component> Game entities have components; but not all game entities have same components. For example, BlackHole does not have VelocityComponent.

Methods:

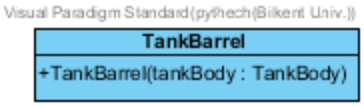
+getComponent(cls: Class<T>): T -> returns the component of an entity of a given type. If it does not exist, it returns null.

+addComponent(component: Component): void ->adds a component to the entity

+removeComponent(cls: Class): void -> removes the specified component from the entity

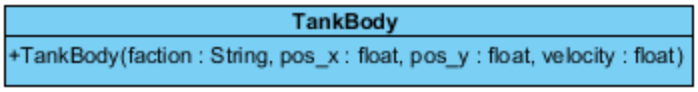
+removeAllComponents(): void -> clear all components

TankBarrel Class:



TankBarrel class extends Entity abstract class and it has GraphicsComponent, AttachComponent and AngleComponent and FactionComponent.
Constructor: +TankBarrel(tankBody : TankBody) -> Note that the constructor has only 1 variable in the parameter. String faction of TankBarrel is formed by getting the FactionComponent from the TankBody that is defined in the constructor. Also AttachComponent is defined by using the TankBody. GraphicsComponent is determined according to texture defined in the body of this constructor. Also AngleComponent is defined randomly in the constructor body.

TankBody Class:

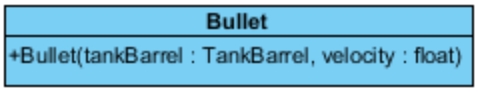


TankBody class extends Entity abstract class and it has GraphicsComponent, PositionComponent, VelocityComponent, AngleComponent, FactionComponent(only in capture the flag game mode), MassComponent, HealthComponent, and SizeComponent.
Constructor:
+TankBody(faction: String, pos_x: float, pos_y:float, velocity: float) -> It initializes the instances of TankBody; but note that it has only 4 parameters: 2 of them is for PositionComponent(x and y), 1 of them is for VelocityComponent and 1 of them is for FactionComponent. Other components are defined in constructor body. Initial HealthComponent is 100, initial MassComponent is 0.01f, initial AngleComponent is randomly generated and initial SizeComponent is determined according to texture defined in this constructor.

BlackHole Class:

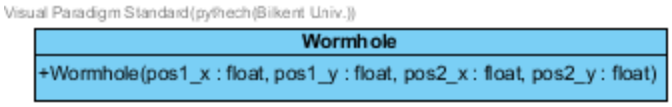
Blackhole class extends Entity abstract class and it has PositionComponent, MassComponent. Blackhole is an invisible object but it pulls tanks that are close to them like blackholes in real life.
Constructor: +Blackhole(pos_x: float, pos_y: float, mass: float) -> It takes 3 parameters(2 of them(x and y) is for PositionComponents and 1 of them(mass) is for SizeComponent) to construct Blackhole object.

Bullet Class:



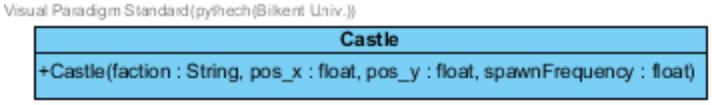
Bullet class extends Entity abstract class. It has GraphicsComponent, AngleComponent, VelocityComponent and PositionComponent.
Constructor: +Bullet(tankBarrel: TankBarrel, velocity: float) -> It initializes the instances of Bullet; but note that it has only 2 parameters. Velocity component is initialized according to constructor declaration. AngleComponent and PositionComponent are determined through calling and getting the TankBarrel's AngleComponent and PositionComponent correspondingly. Also GraphicsComponent is determined according to texture defined in the body of this constructor.

Wormhole Class:



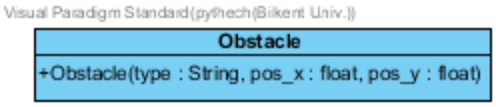
Wormhole class extends Entity abstract class and it has two PositionComponents, a GraphicsComponent. A wormhole has two ends, each end teleports a tank to the other side. That's why we store two PositionComponents.
Constructor: +Wormhole(pos1_x: float, pos1_y: float, pos2_x: float, pos2_y: float) -> A wormhole entity is created with the PositionComponents using (pos1_x, pos1_y) and (pos2_x, pos2_y) coordinates. GraphicsComponent will be added to display this component and will displayed in the both sides.

Castle Class:



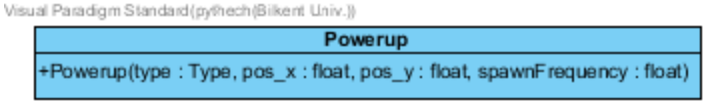
Castle class extends Entity abstract class and includes SizeComponent, FactionComponent, SpawnComponent, HealthComponent, PositionComponent and GraphicsComponent. Enemy and friendly tanks will spawn inside these castles in the Capture the Flag game mode. Castles have different textures depending on the climate. Player can destroy castles.
Constructor: +Castle(faction: String, pos_x : float, pos_y : float, spawnFrequency: float) -> Creates a Castle entity using the coordinates (pos_x, pos_y) to determine PositionComponent. spawnFrequency is used in SpawnComponent. Faction parameter is used for FactionComponent that describes which faction this castle belongs to. Finally

GraphicsComponent is added together with the SizeComponent and HealthComponent have health set to 500.



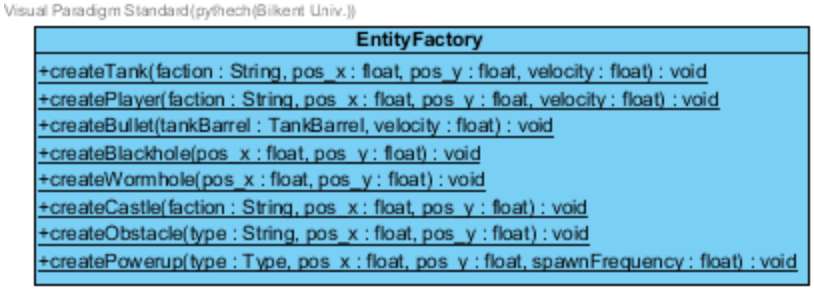
Obstacle class extends Entity abstract class and gives damage to other entities like TankBody. This class have HealthComponent, DamageComponent, GraphicsComponent, PositionComponent and SizeComponent.
Constructor: +Obstacle(type: String, pos_x: float, pos_y: float) -> Obstacle type can be tree1, tree2 and rock. pos_x and pos_y is the PositionComponents. Obstacles' texture(GraphicsComponent) is determined according to its' type. Also obstacles' SizeComponent is defined according to its' texture's height and width. DamageComponent has damage set to 10 health points. HealthComponent has health set to 20.

Powerup Class:



Power-ups are the entities that player can take if his/her tanks collides with it. It includes PowerupComponent, PositionComponent, GraphicsComponent and SpawnComponent.
Constructor:
+Powerup(type: Type, pos_x: float, pos_y:float, spawnFrequency: float) -> This constructor initializes the instances of power-up like its' type, its' positions and its' spawn frequency. If spawnFrequency is 0 then SpawnComponent won't be added.

EntityFactory Class:



This class provides Factory methods for creating entities with an easier interface.
Methods:
+createTank(faction: String, pos_x: float, pos_y: float, velocity: float): void -> create TankBody and TankBarrel.
+createPlayer(faction: String, pos_x: float, pos_y: float, velocity: float): void -> create a special player TankBody and TankBarrel with PlayerComponent added.

+createBullet(tankBarrel: TankBarrel, velocity: float): void -> create bullet at the tip of the given TankBarrel.
+createBlackhole(pos_x: float, pos_y: float): void -> creating blackhole
+createWormhole(pos_x: float, pos_y: float): void -> creating wormhole
+createCastle(faction: String, pos_x: float, pos_y: float): void -> creating castle
+createObstacle(type: String, pos_x: float, pos_y: float): void -> creating Obstacle with its' type and position
+createPowerup(type: Type, pos_x: float, pos_y: float, spawnFrequency: float): void -> creating Powerup with its' type, PositionComponent and spawnFrequency

3.3.5 System Subsystem

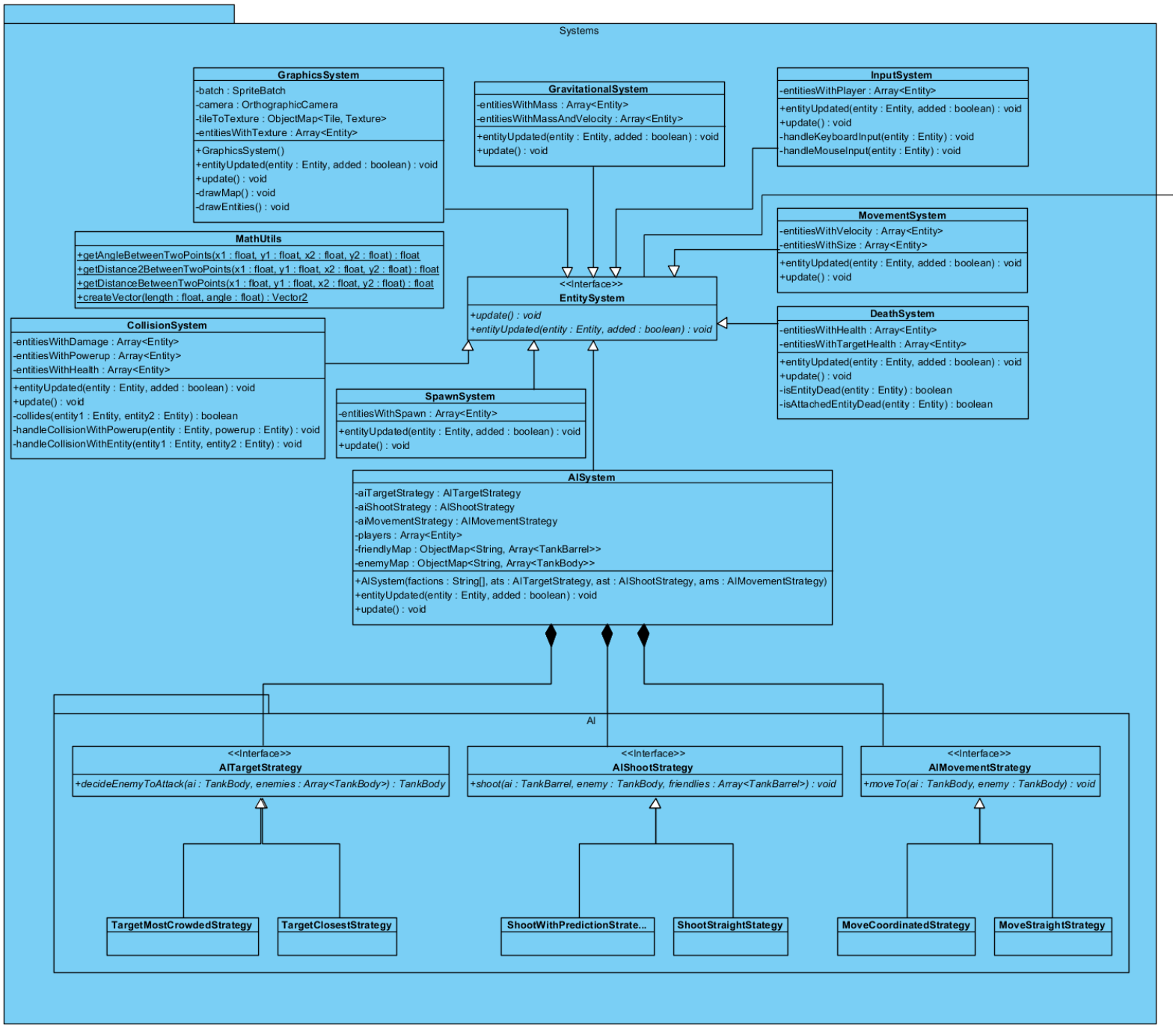
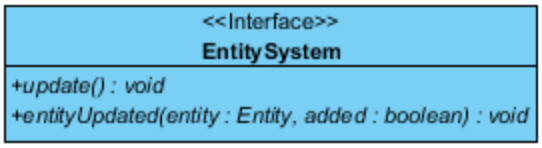


Figure-07: Systems Subsystem of Tank Zone Game

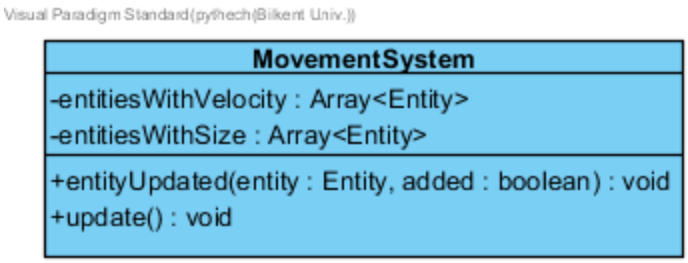
EntitySystem Interface:



EntitySystem is an interface and is later implemented by some game systems like GraphicsSystem, InputSystem, MovementSystem. These classes operate on entities that have certain components related to its function. For instance, MovementSystem only considers entities that have SpeedComponent since the entitites that don't have SpeedComponent cannot move by definiton.

Methods:
+update(): void -> Called by the Engine for each frame. Applies application logic to the entities of interest.
entityUpdated(entity: Entity, added: boolean): void -> Called when the engine adds or removes an entity. The system gets notified about the event and checks whether the added/removed entity belongs to this system. If it is added (indicated by added parameter being true), it stores the given entity to one of the Array data structure. If it is removed (indicated by the added parameter being false), then again it is checked whether this entity belongs to this system and removed accordingly.

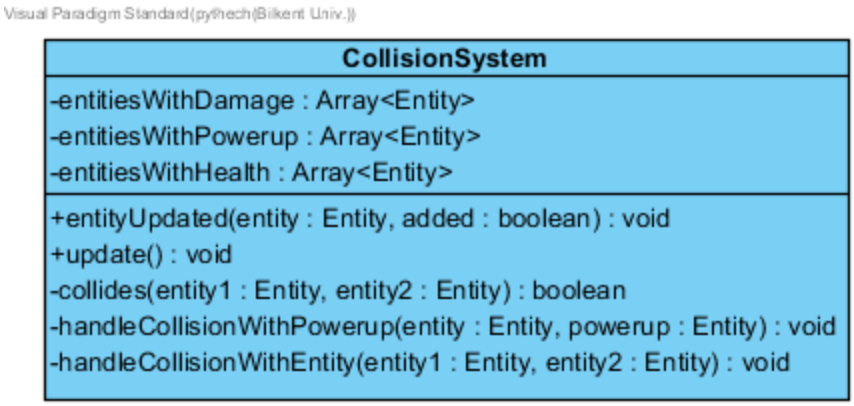
MovementSystem Class:



MovementSystem implements EntitySystem and this class is calculating where the entity will move (the last position of an entity).
Attributes:
-entitiesWithVelocity: Array<Entity> -> The array of entities that have velocity
-entitiesWithSize: Array<Entity> -> The array of entities that have size
Methods:
+entityUpdated(entiy: Entity, added: boolean): void -> In this method, PositionComponent, VelocityComponent and AngleComponent of an entity is defined from getting these components from the entity class accordingly. If both 3 components(position, velocity and angle) is NULL, this method does nothing. Otherwise this method adds this entity to array entitiesWithVelocity and boolean added is true in this case.

+update(): void -> In this method, PositionComponent, VelocityComponent and AngleComponent of an entity is defined from getting these components from the entity class accordingly. Also deltaTime is defined. Also the last positions(changes in x and y) is determined by using entity's AngleComponent and VelocityComponent

CollisionSystem Class:



This class implements EntitySystem and also 2 arrays: entitiesWithDamage and entitiesWithHealth. We use Circle-To-Circle collision system to determine if a collision occurs. The class also determines the amount of damage of the collisions through looking damagecomponent of the entities and also determines the amount of health of the entities before and after collision occurs. Note that to determine whether the collision occurs, this class gets the PositionComponent of the 2 entities, SizeComponent of the 2 entities and the distance between them through using another method getDistanceBetweenTwoPoints().

Attributes:

-entitiesWithDamage: Array<Entity> -> the entities that give damage to tankbody like bullet, obstacles when collision occurs.

-entitiesWithHealth: Array<Entity> -> the entities that have health value like tankbody

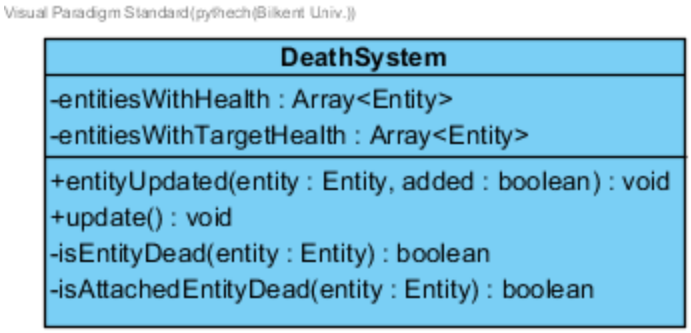
Methods:

+entityUpdated(entity: Entity, added: boolean): void -> This method creates variables PositionComponent, SizeComponent, DamageComponent and HealthComponent by getting the corresponding variables from the Entity entity. If this entities' positionComponent and sizeComponent is NULL, collision can not be defined. Also if this entity has DamageComponent, this method adds this entity to entitiesWithDamage or removes this entity from entitiesWithDamage depending on boolean added. If this entity has HealthComponent, this method adds this entity to entitiesWithHealth or removes this entity from entitiesWithHealth depending on boolean added..

+update(): void -> This method gets position, size and damage components of all entities in the entitiesWithDamage and entitiesWithHealth array. This method also creates the variable distance. If distance is smaller than the size of entities, collision occurs. If the entity is destroyOnHit, it is removed from the game. Also if the entity is not destroyOnHit, this entity's health value decreases proportional to the damage of other entity.

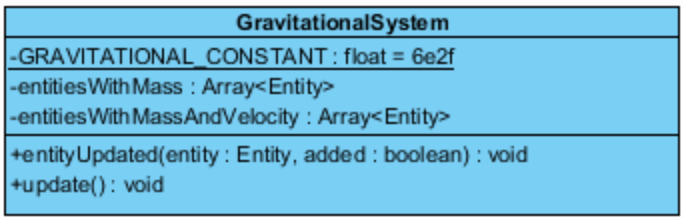
-collides(entity1 : Entity, entity2 : Entity) : boolean -> Determines whether the two entities collide.
-handleCollisionWithPowerup(entity : Entity, powerup : Entity) -> Executes a sequence of actions depending on the type of the power up over the entity that collides with the powerup.
-handleCollisionWithEntity(entity1 : Entity, entity2 : Entity) -> Handles a collision between two entities. Depending the on the components of the two entities, a collision may result in health loss or even destruction.

DeathSystem Class:



DeathSystem class implements EntitySystem. It removes the entities that health lower than 0.
Attributes:
-entitiesWithHealth: Array<Entity> -> For entities which have health
-entitiesWithTargetHealth: Array<Entity> -> For entities which don't have health but their attached entity has health.
Methods:
+entityUpdated(entity: Entity, added: boolean): void ->
+update(): void -> In this method, a AttachComponent variable and HealthComponent variable of an entity are initialized. If the HealthComponent of an entity is smaller than or equal to 0, this entity is removed from the system. Also if there is no target alive(AttachComponent's HealthComponent is smaller than or equal to 0), again the entity is removed from the system. Note that tank and tankbody has healthComponent and tankbarrel has targetComponent.
-isEntityDead(entity : Entity) : boolean -> Checks whether the given entity has health lower than 0.
-isAttachedEntityDead(entity : Entity) : boolean -> Checks whether the given entity's attached entity is dead.

GravitationalSystem Class:



This class determines gravitational force; which applies to all entities that have massComponent.

Attributes:

-GRAVITATIONAL_CONSTANT: float = 6e2f -> This constant is needed to calculate the gravitational force together with using masses

-entitiesWithMass: Array<Entity> -> array of entities with mass like TankBody and BlackHole

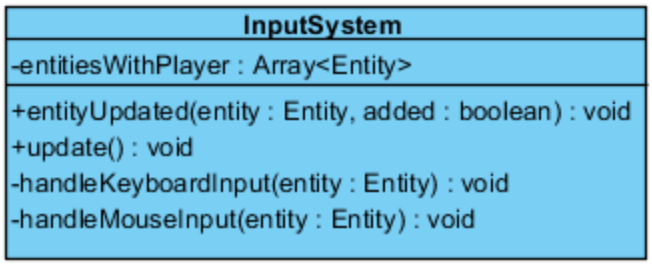
-entitiesWithMassAndVelocity: Array<Entity> -> array of entities with mass and velocity like TankBody

Methods:

+entityUpdated(entity: Entity, added: boolean): void -> Collects entities that have mass and/or speed component.

+update(): void -> Applies the Newton's gravitational force formulas to the entities inside the system. Acceleration of the entities are calculated and added to their speeds and angles accordingly (this is because speed is a scalar value but the force is applied using vectors).

InputSystem Class:



Responsible for responding to Keyboard and Mouse related events.

Attributes:

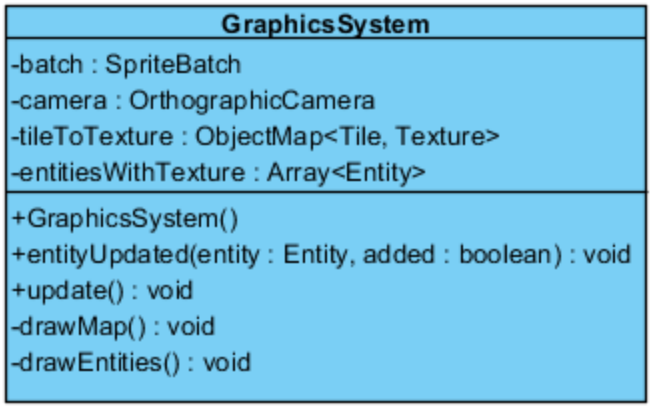
-entitiesWithPlayer: Array<Entity> -> the player that leading the tankbody and tankbarrel

Methods:

+entityUpdated(entity: Entity, added: boolean): void -> This method gets PlayerComponent from an entity like TankBody. If playerComponent is NULL, entity is not updated and added is false. If playerComponent is not NULL and if added is true, this methods adds the entity to

entitiesWithPlayer. If playerComponent is not NULL wheras added is false, this methods removes this entity from entitieswWithPlayer.
+update(): void -> This method gets input from the user and in case steerable(tankbody), the angle and velocity changes according to direction keys(Right, Left, Up, Down) pressed by the user. In case rotatable(tankbarrel), the TargetComponent(shooting angle) is determined by the user mouse input.
-handleKeyboardInput(entity : Entity) -> Handles keyboard related events on the entities that have STEERABLE type.
-handleMouseInput(entity : Entity) -> Handles mouse related events on the entities that have ROTATABLE type.

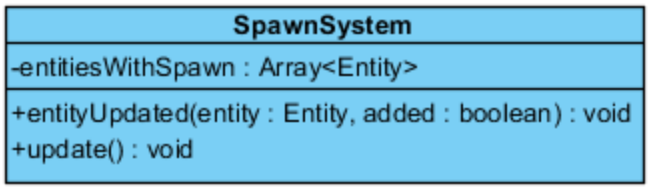
GraphicsSystem Class:



GraphicsSystem implements EntitySystem and this system is responsible from drawing the map and entities inside the camera. Camera follows the cameraTarget provided by the World class.
Attributes:
-batch: SpriteBatch -> batch is a new SpriteBatch to draw textures.
-camera: OrthographicCamera -> camera is an OrthographicCamera to show only a frame of the entire map.
-tileToTexture: ObjectMap<Tile, Texture> -> mapping of tiles and corresponding textures
-entitiesWithTexture: Array<Entity> -> array of some entitites that have texture
Methods:
+GraphicsSystem() -> Constructs the GraphicsSystem
+entityUpdated(entity: Entity, added: boolean): void -> This method creates a GraphicsComponent after getting GraphicsComponent from entity Entity. If the GraphicsComponent is not NULL, this method adds this entity to array entitiesWithTexture or removes this entity from array entitiesWithTexture depending of value boolean added.
+update(): void -> This method creates PositionComponent by calling getCameraTarget() from world. It also creates map by calling the class World. This method later decides texture to draw(grass or dirt according to map).

-drawMap(): void -> Draws the background map using tileToTexture mapping. Note the map is not implemented using entities.
-drawEntities() : void -> Draws the entities that have textures.

SpawnSystem Class:

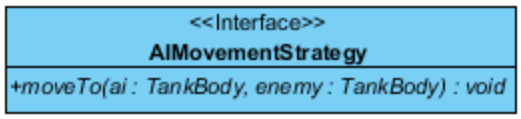


SpawnSystem class implements EntitySystem. It is responsible for spawning entities.
Attributes:
-entitiesWithSpawn: Array<Entity> -> For entities which have SpawnComponent
Methods:
+entityUpdated(entity: Entity, added: boolean): void -> Checks whether the entity has SpawnComponent and adds/remove them.
+update(): void -> Using the spawnFrequency and lastTimeSpawned attributes of SpawnComponent, the SpawnSystem will generate entities inside the system whenever needed.

AIShootStrategy Interface:

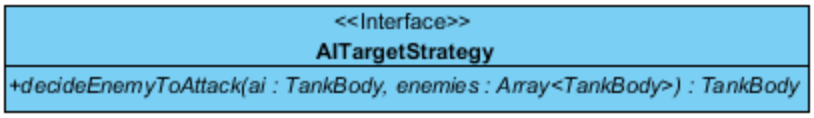
It is an interface with only 1 method. This interface is later implemented by ShootStraightStrategy class.
Methods:
+shoot(ai: TankBarrel, enemy: TankBody, friendlies: Array<TankBarrel>): void -> This method is to calculate the shooting angle and is later overrided in ShootStraightStrategy class.

AIMovementStrategy Interface:



It is an interface with only 1 method. This interface is later implemented by MoveStraightStrategy class.
Methods:
+moveTo(ai: TankBody, enemy: TankBody): void -> This method is to determine in which angle the TankBody(ai and enemy) will move. This method is later overrided in MoveStraightStrategy class.

AITargetStrategy Interface:



It is an interface with only 1 method. This interface is later implemented by TargetClosestStrategy class.

Methods:

+decideEnemyToAttack(ai: TankBody, enemies: Array<TankBody>): TankBody -> This method is to determine which enemy TankBody the the ai TankBody will attack. This method is later overrided in TargetClosestStrategy class.

AISystem Class:

AISystem implements EntitySystem and this class determines the TargetStrategy, MovementStrategy and ShootStrategy of the TankBody AI's. Also friendly tanks(same team) and enemy tanks(another team's tanks) can not have the same faction(color).

Attributes:

-aiTargetStrategy: AI TargetStrategy -> The TargetStrategy of TankBody AI's.

-aiShootStrategy: AIShootStrategy -> The ShootStrategy of TankBody AI's

-aiMovementStrategy: AIMovementStrategy -> The MovementStrategy of TankBody AI's

-player: Entity -> Note that player can use TankBody and TankBarrel

-friendlyTanksForFaction: ObjectMap<String, Array<TankBarrel>> -> The array; which stores the same team member's TankBarrel's. (all team members have the same faction(color) in Capture The Flag Game Mode)

-enemyTanksForFaction: ObjectMap<String, Array<TankBody>> -> The array; which stores different team member's TankBody's. (different team's TankBody's have different factions(colors) in Capture The Flag Game Mode)

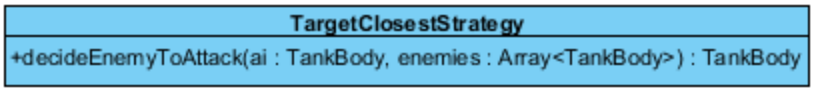
Constructor:

+AISystem(factions: String[], aiTargetStrategy: AITargetStrategy, aiShootStrategy:AIShootStrategy, aiMovementStrategy:AIMovementStrategy,) -> Constructs the AISystem using the given strategies and factions. If factions is null then Free For All mode is assumed.
+entityUpdated(entity: Entity, added:boolean): void -> Whenever an entity is added it does a sequence of complex tasks. This involves calculating friendlyTanks for and enemyTanks mappings using the added/removed entity's faction.
+update(): void -> Using the strategies, it first determines which enemy to attack, then how to move to the enemy and finally shoot it.

ShootStraightStrategy Class:

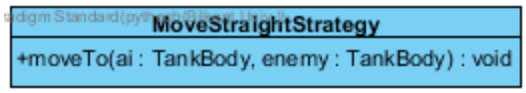
This class implements the AIShootStrategy Interface. Shoot straight strategy class determines the angle between the ai and enemy tanks through getting their positions(PositionComponents).
Methods:
-collides(lineStart:Vector2, lineEnd:Vector2, sphereCenter:Vector2, radius:float): Boolean -> This method tries to avoid hitting friendly entities. A bullet's pathway is imagined as a line and it will be checked whether this line collides with the given sphere, which is a friendly tank. If this returns true the AI will avoid shooting.
+shoot(ai: TankBarrel, enemy: TankBody, friendlies: Array<TankBarrel>): void -> This method determines the angle between ai and enemy tanks through getting their positions(PositionComponents). It also creates Entity Target from using TankBarrel's TargetComponent. This method creates variable shootangle determined by the method call (getAngleBetweenTwoPoints). Also friendly tanks are stored in array and this method also prevents AI and enemy TankBody's shooting at friendly tanks. TankBody should only shoot at enemy tanks.

TargetClosestStrategy Class:



This class implements AITargetStrategy and according to this strategy, tankbody attacks to the closest enemy.
Methods:
+decideEnemyToAttack(ai: TankBody, enemies: Array<TankBody>): TankBody -> This method creates a minimum distance variable using Float.MAX_VALUE and if the distance(determined by getDistance2Between2Points() method) between ai and enemy is smaller than minimum distance, variable "closest" defined in this method is this enemy and this method returns the variable closest.

MoveStraightStrategy Class:



This class creates and initializes both variables(velocity and angular velocity) to 100.

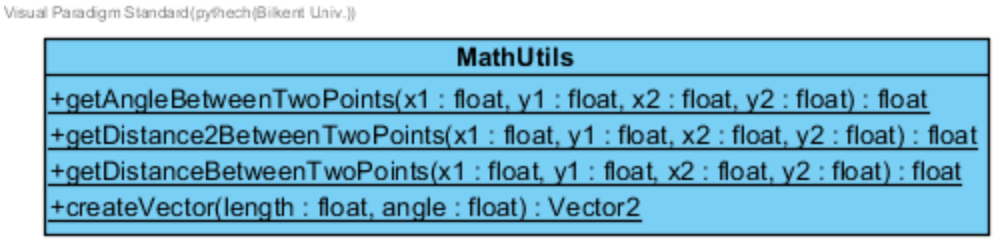
Methods:

+moveTo(ai: TankBody, enemy: TankBody): void -> This method creates currentAngle by getting the current positions(PositionComponents) of aiPos and enemyPos. The method also creates targetAngle by calculating the angle (getAngleBetweenTwoPoints() method) between ai and enemy positions.

This method also creates variable newAngle for TankBody's with using `Math.max(currentAngle - ANGULAR_VELOCITY * deltaTime, targetAngle);`

Also if (newAngle - targetAngle) > 10, and if distance<100000, the TankBody ai's velocity decreases as deltaTime passes. Otherwise the TankBody ai's velocity increases as deltaTime passes.

MathUtils Class:



Methods:

+getAngleBetweenTwoPoints(x1: float, y1: float, x2: float, y2: float): float -> This method calculates the angle between 2 points (2 (x,y) positions). Expressed in degrees.

+getDistance2BetweenTwoPoints(x1: float, y1: float, x2: float, y2: float): float -> This method calculates the distance between 2 points (2 (x,y) positions) but the distance is actually squared. This is because of the formula, the distance between two points is sqrt((x1 - x2)^2 + (y1 - y2)^2). Sometimes we need the squared distance so having a seperate function gives us slight performance advantage. Expressed in meters.

+getDistanceBetweenTwoPoints(x1: float, y1: float, x2: float, y2: float): float -> Same as above but instead returns the square root, so it gives the exact distance in meters.

+createVector(length: float, angle: float): Vector2 -> Creates a Vector2 class with the specified length (meters) and angle (degrees).

3.3.6 Storage Subsystem

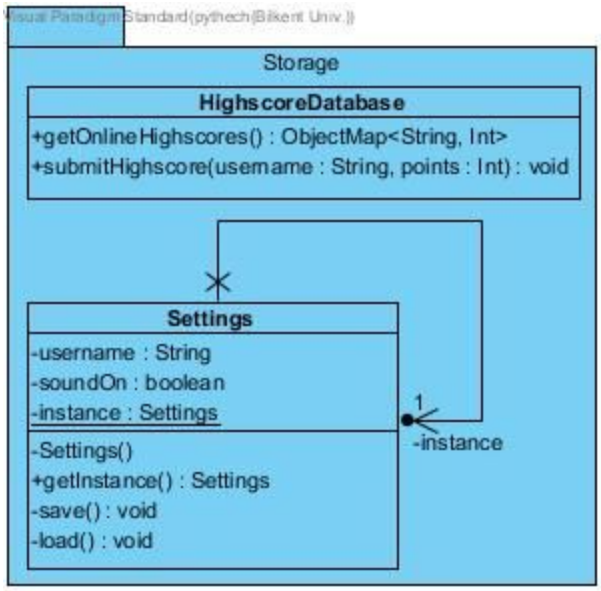
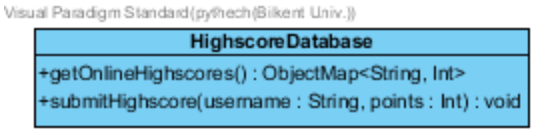


Figure-08: Storage Subsystem of Tank Zone Game

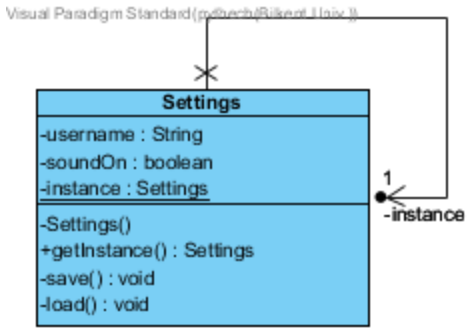
This subsystem is responsible for collecting highscores of different users in a database and submit them on the storage. This subsystem is also in charge of sound settings. Last state of the sound will be remembered with the existence of soundOn and username variable.

HighscoreDatabase Class:



Methods:
+getOnlineHighScores() : ObjectMap<String, int> -> This method is responsible for getting online highscores of different users in an ObjectMap.
+submitHighscore(username: String, points: int):void-> This method is responsible for storing online highscore of a specific user.

Settings Frame:



Attributes:

-username: String
This attribute exists for holding the name of the user.
-soundOn: boolean
Last state of the sound will be remembered in our game according to username. For this reason, soundOn variable also exists.
-instance: Settings
This attribute will store 1 if the soundOn is true. Otherwise, it will store 0.

Constructor: constructs Settings frame.

Methods:
+save(): void -> this method saves the last state of the sound state of the user.
+load(): void -> this method loads the last state of the sound state of the user whenever the user starts playing the game.
+getInstance(): Settings -> this method returns the Settings object.

4. Improvement Summary
Compared to our draft design report, we added the feature of remembering the last state of the sound of the specific user in our game.We have changed our architectural design from ground-up so all the diagrams had to be changed. Using the feedbacks we received, we have refined our design goals. We have added design patterns. Trade-offs section was improved. Online highscore database section is added.