KOÇ UNIVERSITY

COMP 302 – SOFTWARE ENGINEERING

FALL 2016

TERM PROJECT

Project Assignment 6

Final Design Report and Source Code

Team Name: UC

Instructor: Attila Gürsoy

TA: Farideh Halakou

Arda Arslan

Atahan Bekiroğlu

Aykut Aykut

Oğulcan Büyüksandalyacı

Zafer Çavdar

Submission Deadline: 05.01.2017

TABLE OF CONTENTS

1. Summary of the final design

1.1 Program flow

Our final design has explicitly and implicitly implemented 7 design patterns, 7 interfaces,59 classes and 11 packages.

In the program flow, Main class calls GameWindow GUI class and all of the user interactions are captured by UI classes like BeginningPanel, BoardPanel, BuildingPanel and GameWindow. Actions are delegated to Controller classes and controllers decide which method to call from model classes. Also controller classes creates required threads according to user inputs and threads perform jobs according to their individual responsibilities. We have 4 thread classes:

- BallThread for the movement of first ball
- SecondBallThread for the movement of second ball
- CezeryeThread for the creation and deletion of cezerye object.
- GizmoThread for the continuous rotation of Firildak and,
- Tokat's running mode rotations are handled by inner Runnable classes that acts as different threads in LeftTokat and RightTokat classes.
- In any case, Swing and Main threads already work.

We have a singleton BoardModel class that control domain objects and lots of domain objects as a part of game. Any change in domain is notified to observers and observers call required UI methods to change the appearance of the GUI. UI-Domain separation is handled by patterns and we attached lots of importance to low coupling and high coherence while we were implementing classes.

1.2 Classes and interfaces in patterns

In the assignment 4 of the project, we had decided to use 7 design patterns in the project implementation and we had efficiently implemented and used these 7 patterns.

1. Observer Pattern
2. Simple Factory Pattern
3. Strategy Pattern
4. Singleton Pattern
5. Creator Pattern
6. Controller Pattern
7. Expert Pattern

How did we implement observer pattern?

To implement observer pattern, we have both observable and observer object. Simply observable object is a java object that implements Observable interface which has two methods with the following signatures:

```
public void attachObserver(Observer observer);
```

public void notifyAllObservers();

Observable objects have a field named “observers” whose type is ArrayList<Observer> and attachObserver method adds the observer parameter to this list. notifyAllObserver method calls update methods of observers in the list.

Moreover, observer object is a java object that extends Observer abstract class with the method named update(Observable object). Each observers’ update method takes one observable object and calls appropriate GUI repaint method according to type of this object.

The classes that implement Observable interface in our project are below:

- Ball
- Cezerye
- Cezmi
- Firildak
- Takoz
- Tokat
- TriangleTakoz
- Player
- BoardModel

The classes that extend Observer class in our project are below:

- BallObserver
- CezeryeObserver
- CezmiObserver
- GizmoObserver
- PlayerObserver
- WinPanelObserver

While update method calls observer with the argument **this,** observer object down-casts object into appropriate observable object and calls GUI’s one of the specialized repaint methods with passing the object as a parameter. Player has PlayerObserver and it does not know which type of observers are alerted when the score has changed. In this pattern, observable objects do not interact with the UI classes or methods, UI changes are controlled by observers and handled by UI classes.

How did we implement Simple Factory pattern?

We have 2 Simple Factory classes in our implementation:

- BoardObjectFactory
- EventFactory

The former one controls the creation of all board objects and latter one controls the event creation. During the building mode or loading the game from an XML file, these factory objects are created. To create a board object such as Ball, Cezmi or various Gizmos, BoardObjectFactory’s one of the 2 methods are called with the BoardObjectEnum parameter that stores the type of which object to create. EventFactory has 1 method to create

event. This method simply chooses 1 event randomly from 3 different event types and this factory is called by Cezerye when new cezerye is created in the game within some time intervals.

By using simple factory pattern, we separated creation of most of the model objects from inner class methods. To change the types of events or creation logic of the board objects, we now just change the factory classes. Other classes remain same.

How did we implement Strategy pattern?

We have an interface called Event and 3 classes implements Event interface. Event interface has 2 method signatures:

public void applyEvent(ArrayList<Cezmi> cezmis);
public void finishEvent();

We have 3 classes that implements Event interface:

MagnifyEvent
ShrinkEvent
FreezeEvent

Each event class implements these two methods, these events are created by EventFactory randomly and stored in Cezerye. While the ball collides with cezerye, cezerye's applyEvent method is called. Depending on the type of the event, applyEvent magnifies, shrinks or freezes the target cezerye. The strategy (event type) is selected randomly and this pattern is applied with the help of classes that implements same interface.

How did we implement Singleton pattern?

Following classes are implemented as Singleton:

GameWindow (from gui package)
BoardObjectFactory (from factory package)
EventFactory (from factory package)
BoardModel (from model package)

The main purpose of implementing these classes as singletons are to create some objects that are globally accesible by other classes and to ensure the class only has one instance. By using singleton pattern, we stored all board related variables in a single BoardModel, created all domain objects using 2 singleton factories and GameWindow is created once and integrated with other GUI components.

To implement the singleton pattern, private constructor is used and private static instance variable of type the class itself has been stored. Static getMethod has returned the instance itself if it is not created, else new class object is returned.

How did we implement the Information Expert pattern?

Information expert is a pattern used to determine where to navigate responsibilities. These responsibilities include methods, computed fields, and so on.

Using the information expert pattern, a general approach to assigning responsibilities is to look at a given responsibility, consider the information that is needed for this responsibility, and then look up where that information is stored. Information expert will help the

developers navigate the responsibilities to classes that contain most of the needed information to fulfill it.

In our implementation, this principle is used in many classes. For example, to move the gizmo in building mode, MoveController calls BoardModel's move methods, because the information of which places are free to move or which ones are occupied are known by the BoardModel. Another example is the movement of the Ball. Ball knows its next position and it can handle its next direction, state or velocity if it knows the collision. By using the singleton BoardModel, ball class gathers other objects position information and checks whether it collides with another object or not. The responsibility of check is fullfilled by ball class. Moreover, shrinking of the ball during running the game is again handled by the ball itself. Repainting the next position of any moved object in the game is not handled by the domain class. Since their representation is not directly related with the interface representation, repaint methods are fullfilled by GUI classes and they decide how to draw domain objects according to their domain representation.

How did we implement the controller pattern?

We have created multiple controller classes to connect our GUI and domain layer. In our singleton GameWindow class, which is our main GUI class, we declared these controllers as the fields of our GameWindow class. Our controller classes are as follows:

- AddGizmoController
- DeleteGizmoController
- LoadController
- MoveGizmoController
- PauseController
- PlayController
- RotateGizmoController
- ResumeController
- SaveController
- QuitController

Except for the PlayController class, which is the controller that handles the in-game keyboard clicks that is used to play the game by both players, all the controllers have a button on the GUI layer that can be pressed to call their buttonClick() method embedded inside them. Inside these buttonClick() methods we set the state Enum of our Singleton BoardModel class accordingly. Later we use the state Enum of our BoardModel to determine what to do with the next input from the users. One of which is the board click inputs. The board click inputs are handled according to the current state Enum inside the boardClick() method of the controllers that are listed below:

- AddGizmoController
- DeleteGizmoController
- MoveGizmoController
- RotateGizmoController

There are controllers that handle the input i/o situations. These controllers have loadBoard and saveBoard methods that operate when the enum allows them to. These controllers are listed below:

- SaveController
- LoadController

And there are controllers that handle flow of the game. These contollers only have the buttonClick() method and they are used to pause, resume, play or to quit the game. These controllers are listed below:

- PauseController
- PlayControllerr
- ResumeController
- QuitController

This way we can easily establish the relation between our GUI classes and Domain classes. We can now implement more classes to our Domain without having the need to change the GUI updating to the controller class.

How did we implement the creator pattern?

Creation of objects is one of the most common activities in an object-oriented system. Which class is responsible for creating objects is a fundamental property of the relationship between objects of particular classes.

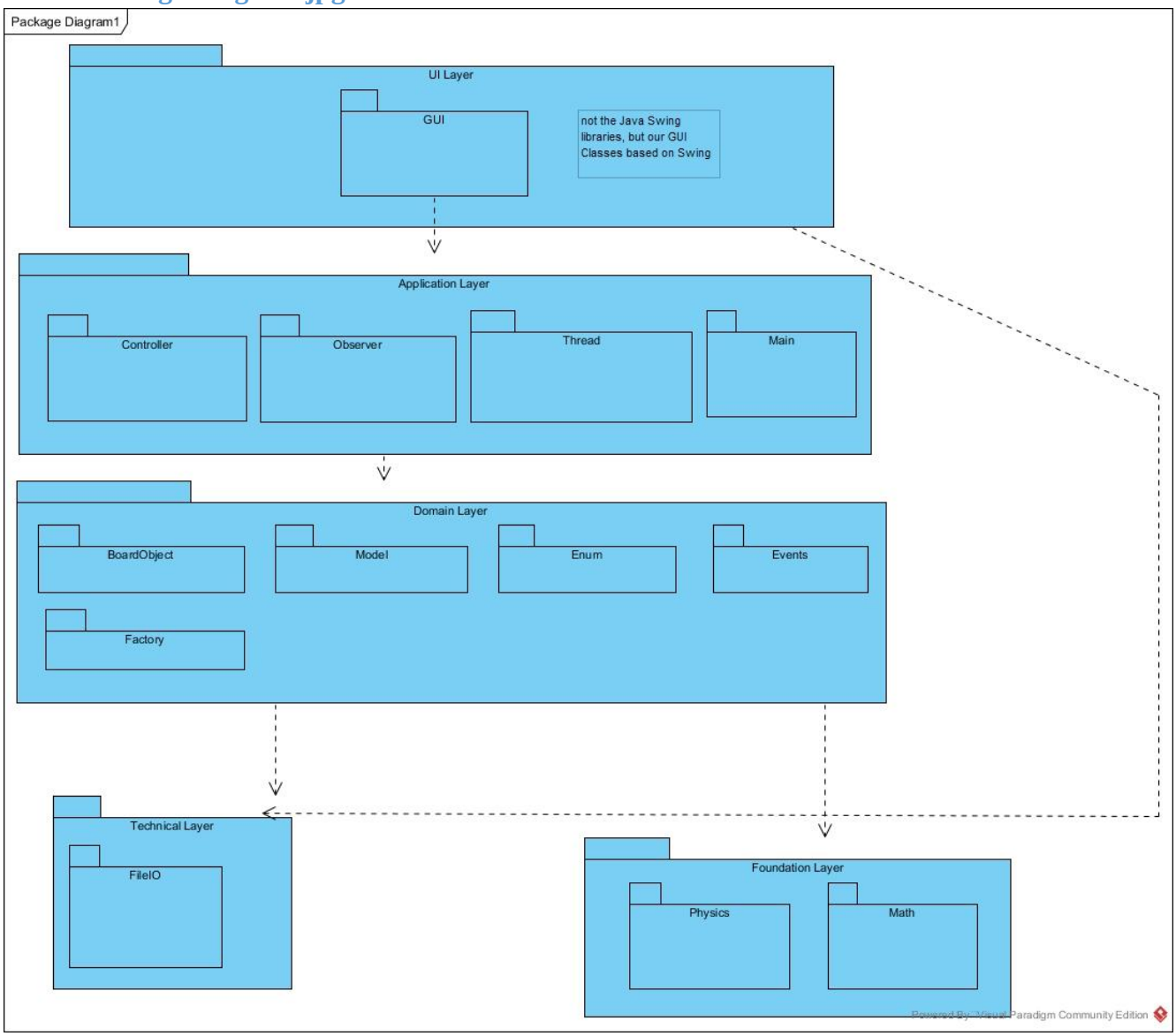
In our implementation, we used a singleton BoardModel class that holds all the board objects as fields and generates them when it is required via calling one of its initialize methods. These initialize methods then call the singleton boardObject Factory to create the object. We keep this BoardModel instance as a field inside our singleton GameWindow class which calls the BoardModel's getInstance() method to begin the initialization cycle.

1.3 Final Class Diagram

Final Class diagram that shows all of the interactions between all classes, interfaces and enumerations in the project is attached to this file in the submission folder and can be located at **Final Diagrams** folder with name **"UC Final Class Diagram.jpg"** . Since it has a high resolution, we could not insert it to this page.

1.4 Final Package Diagram

You may find the latest package interaction diagram below. Also it is attached to submission folder and can be located at **Final Diagrams** folder with name **UC Final Package Diagram.jpg.**



2. Changes in the design from the Design Parts 1-2

While we were working on Design Part1-2, we tried to specify each class relation, class variables and class methods as we are developing the final version and drew them carefully. After finishing the implementation, we recognized that we have needed lots of new private and public methods, variables but our latest design has remained similar to our previously submitted design work. Followings are the changes that we needed to modify:

1. We used concurrency in our project with maximum 12 threads (depending on players' tokat number) and we thought that BoardModel and Main class are suitable for managing these threads; but in the implementation, we realized that threads must be controlled by GameWindow. Therefore, thread managing core is GameWindow now in our implementation.

2. We also changed our observer classes. In design, we planned that observers have subject which is observed, but in implementation, we changed our observing logic and instead of keeping observed objects as an attribute in observers, we give the observed object to the update method of observers as a parameter.

3. In design and use cases, we planned to use 1 button for each building mode gizmo operations, but after that we realized that if we implement our program like in this way then our program will not be very user-friendly. Thus, we changed our design here and created 2 buttons for each building mode gizmo operations, 1 for player1 and 1 for player2.

4. In building mode moving gizmo use case, we planned to use drag-drop method for moving gizmos, but then thought that first selecting a gizmo and then moving it to any available location on the player's editing area looks better and we implemented our move gizmo use case like that.
5. In design, we did not need an observer for BoardModel, but in the implementation, in order to finish the game and show a popup on the screen, we need to add an observer to BoardModel.

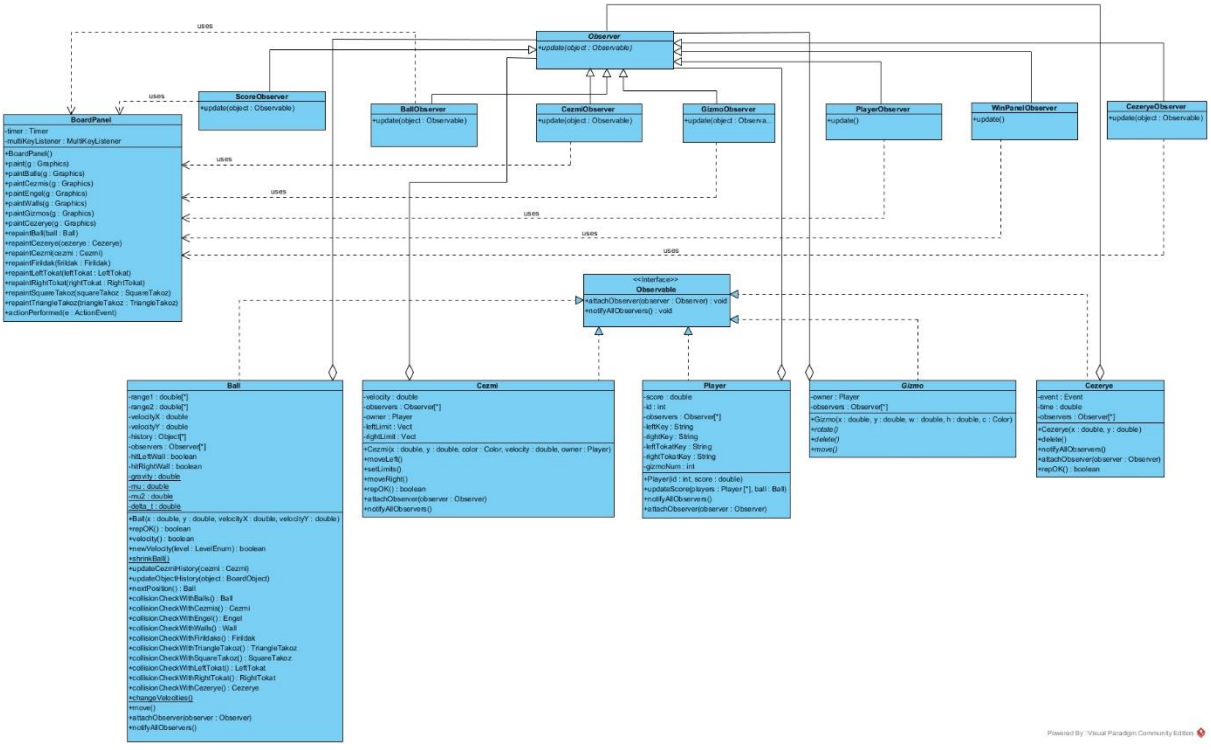
3. All design patterns with individual diagrams

We thought of using GRASP and GoF patterns in our project. In total, we used these 7 design patterns:

a. Observer Pattern
b. Simple Factory Pattern
c. Strategy Pattern
d. Singleton Pattern
e. Creator Pattern
f. Controller Pattern
g. Expert Pattern

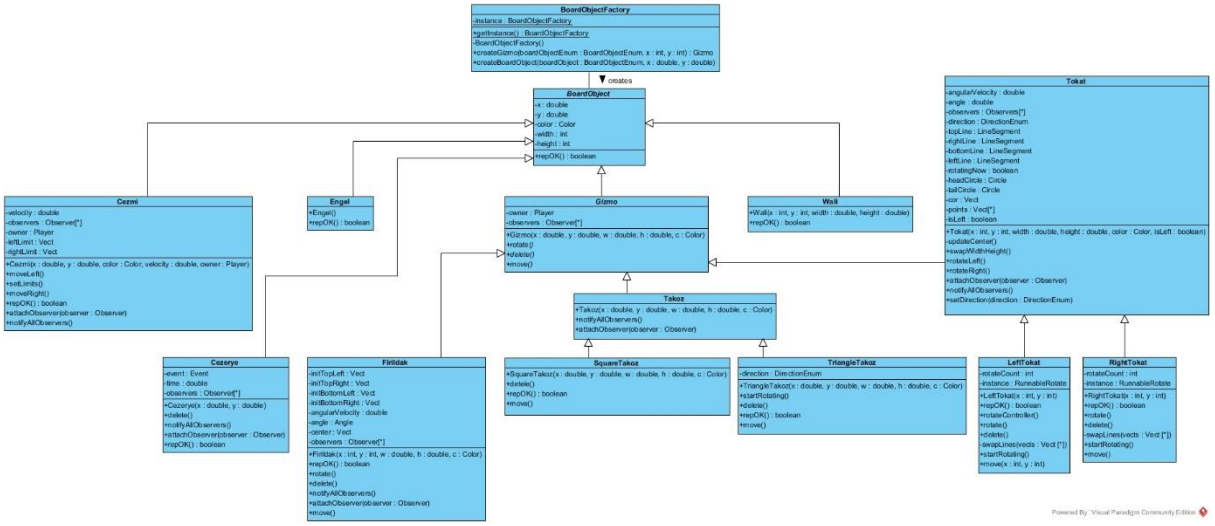
a. Observer Pattern (GoF)

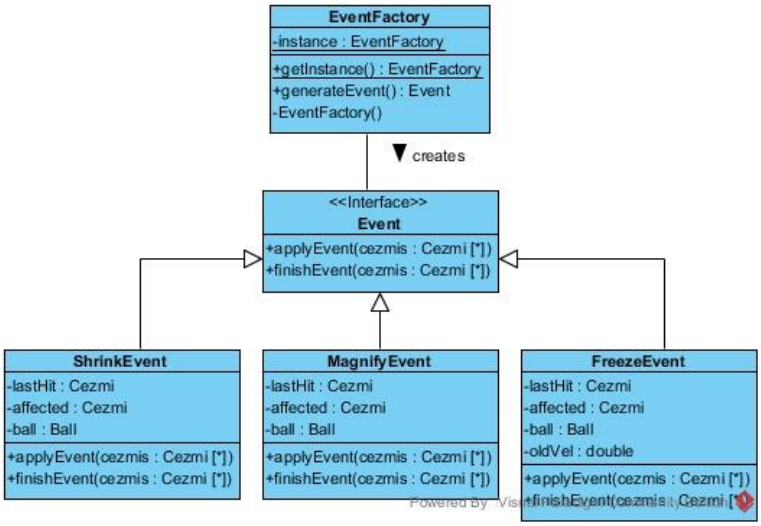
We wanted to keep the domain layer and the UI layer separate. It was appropriate for the UI layer to reach the domain layer but if the domain layer reaches the UI layer directly, this violates the Model View Separation methodology. Therefore, we used the observer pattern for each domain object we implemented. In our project, we used 6 observers of different types. These were BallObserver, CezeryeObserver, CezmiObserver, GizmoObserver, PlayerObserver and WinPanelObserver. We have also a class named "Observer" which was extended by these 6 classes. All observers have "update" methods whose inputs were the objects that were being observed. BallObserver, CezeryeObserver, CezmiObserver and GizmoObserver were used for calling the "repaint" methods of the objects that were being observed. PlayerObserver was used for only reflecting updated scores on the screen. And WinPanelObserver was used for only displaying a pop-up message which indicated the end of the game. In "**ObserverPatternClassDiagram.jpg**" file **located at Pattern Diagrams folder**, one can see how we used Observer Pattern in our design. A thumbnail of this image is as follows:



b. Simple Factory Pattern (GoF)

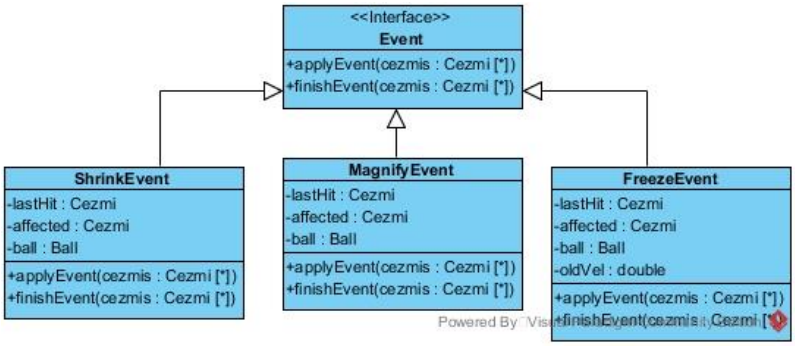
We thought the Simple Factory Pattern would be the suitable creational pattern to create different type of objects which were board objects and events. We used two separate simple factories for this reason. Since we had only two types of simple factories, we did not implemented an abstract factory. We combined this pattern with the singleton pattern so that we could reach the factories in a static way. This allowed us to ensure that there was only one instance of a factory at a time which prevented the system from using too much source of memory. BoardObjectFactory was used for creating new instances of gizmos, engel, walls, cezerye, ball and cezmi. EventFactory was used for creating different types of Event objects randomly. These events were MagnifyEvent, FreezeEvent and ShrinkEvent. In "**FactoryPatternClassDiagram (BoardObjectFactory).jpg**" and "**FactoryPatternClassDiagram (EventFactory).jpg**" files **located at Pattern Diagrams folder**, one can see how we used Observer Pattern in our design. A thumbnail of this image is as follows:





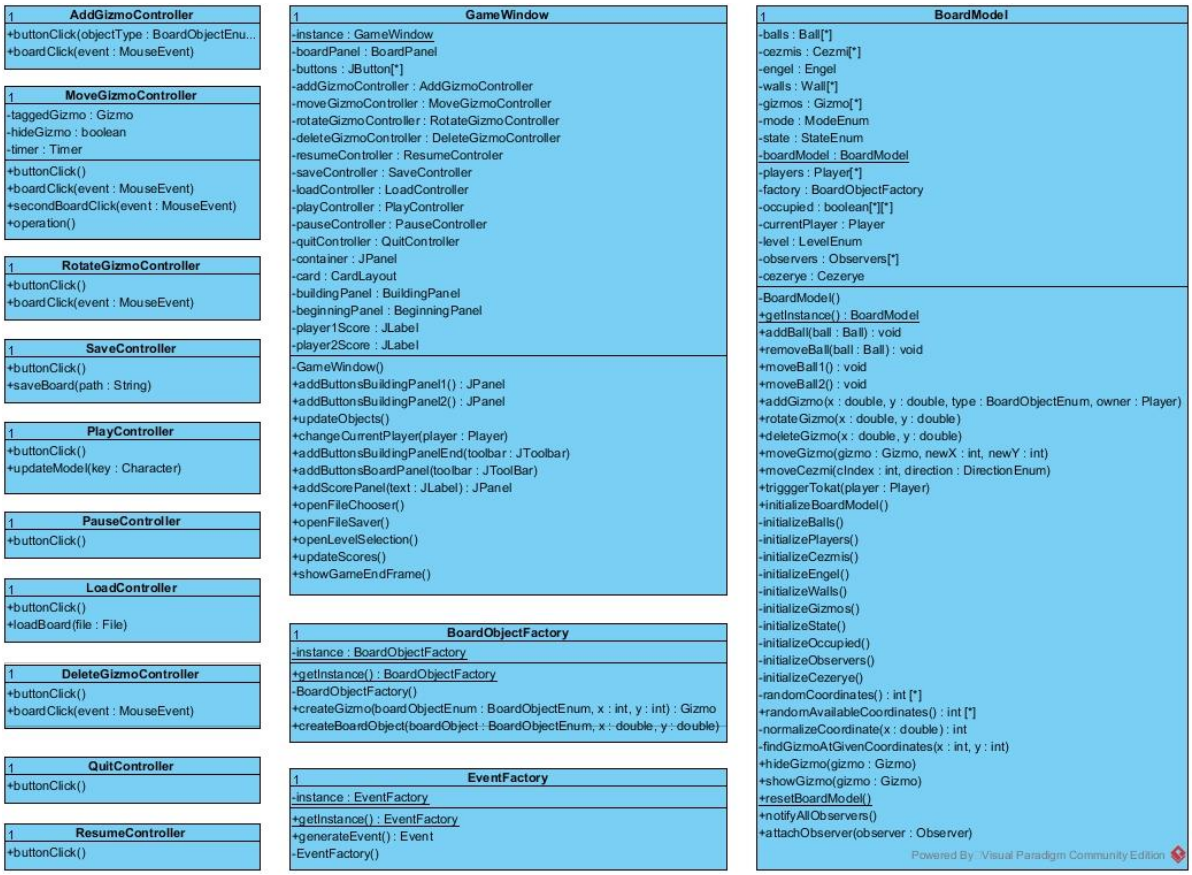
c. Strategy Pattern (GoF)

There were three different events that changed the attributes of Cezmis. We thought that the requirements might have changed and there might be additional events that we might had to handle. Thus, grouping these events under a common interface seemed logical. The strategy pattern was used for this issue; it allowed us to have a hierarchical structure over our events under the interface which is named as Event. This pattern provided high cohesion and low coupling between the concrete classes. In "**StrategyPatternClassDiagram.jpg**" file **located at Pattern Diagrams folder**, one can see how we used Strategy Pattern in our design. A thumbnail of this image is as follows:



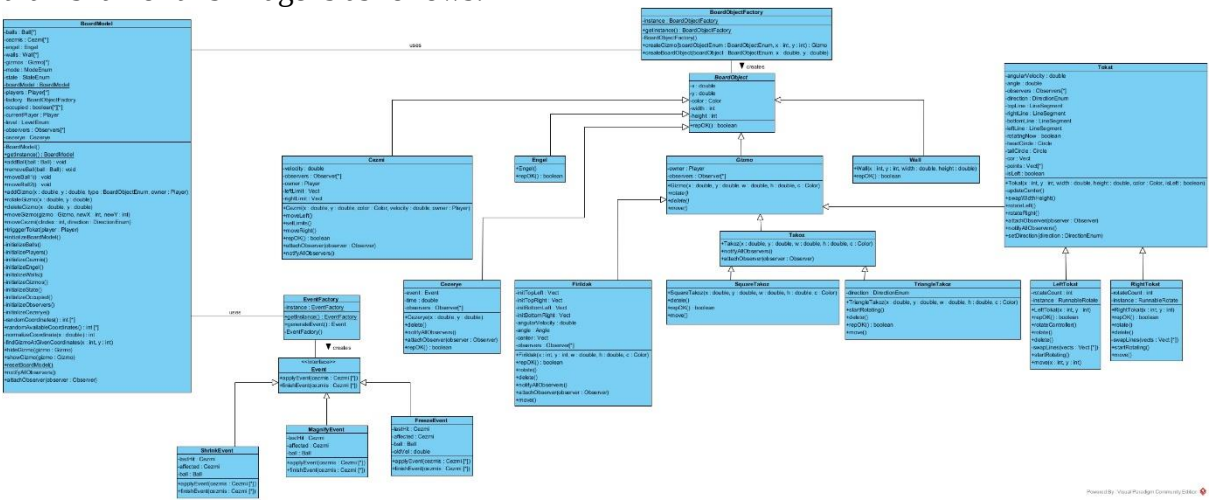
d. Singleton Pattern (GoF)

It was logical to have one instances of BoardModel object, GameWindow object, EventFactory object, BoardObjectFactory object and all types of controllers in our game. And the singleton pattern was used for this issue. It allowed us to reach these objects in a static way which solved the creational concerns. And this also allowed us to ensure that there was only one instance of a factory at a time which prevented the system from using too much source of memory. In "**SingletonPatternClassDiagram.jpg**" file **located at Pattern Diagrams folder**, one can see how we used Singleton Pattern in our design. A thumbnail of this image is as follows:



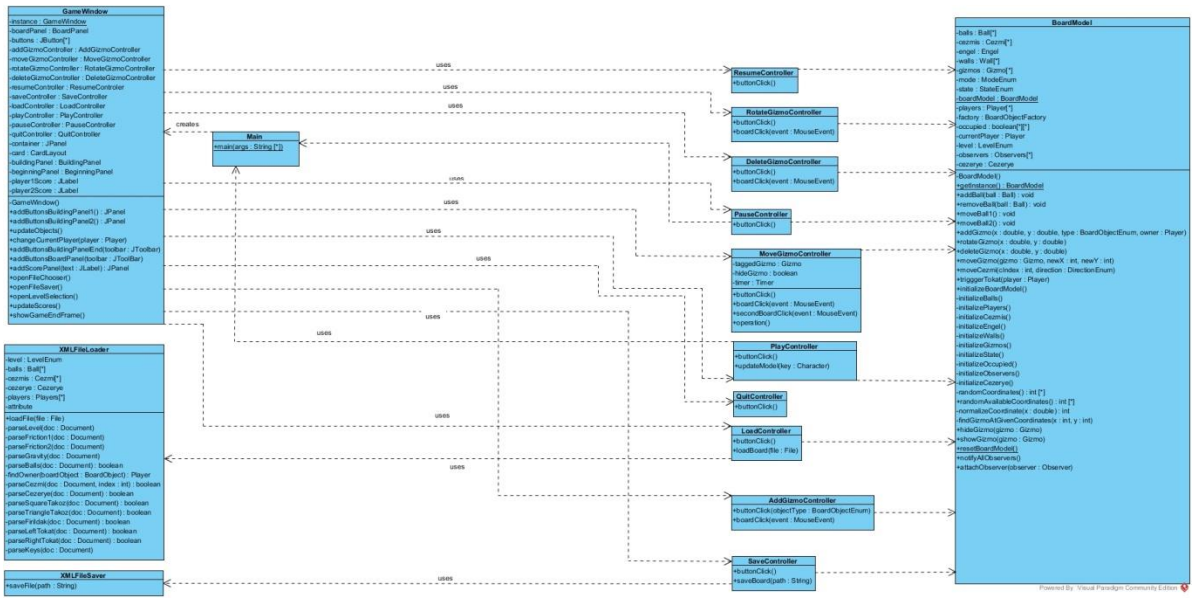
e. Creator Pattern (GRASP)

Creator pattern was used to determine the object that was responsible to create the other objects. We used this pattern implicitly as it was not an actual class in our project. Since BoardModel recorded all the domain objects as its attributes, it was appropriate for this particular class to create all the domain objects using the factories. In "**CreatorPatternClassDiagram.jpg**" file **located at Pattern Diagrams folder**, one can see how we used Creator Pattern in our design. A thumbnail of this image is as follows:



f. Controller Pattern (GRASP)

The best practice was to have a controller for each use case. We had 11 use cases and ended up with 10 controllers in the end. We did not add a controller for the use case “saving game in running mode”, because we had already added a controller for the use case "saving game in building mode". And we decided to use this controller for both use cases. UI layer received input from users and it was responsible for sending the input to the corresponding controller. The chosen controller delegated the process to the appropriate object(s) which would later on be decided by the expert pattern. Name of the controller objects were AddGizmoController, DeleteGizmoController, LoadGizmoController, MoveGizmoController, PauseController, PlayController, QuitController, ResumeController, RotateGizmoController, SaveController. In "**ControllerPatternClassDiagram.jpg**" file **located at Pattern Diagrams folder**, one can see how we used Controller Pattern in our design. A thumbnail of this image is as follows:



g. Expert Pattern (GRASP)

Just like the creator pattern, we used this pattern implicitly. Controller pattern and expert pattern were closely related to each other. Expert pattern was used for delegating responsibilities to the object which contained the information to fulfill the process. In our design, we followed this pattern while assigning the controllers’ destination. Since we used Expert Pattern implicitly, we could not provide any class diagram to show how we used this pattern in our design.

4. User manual

1. Introduction

The aim of this document is to help players get familiarized with our game, *Hadi Cezmi*. This is a step-by-step guide which will aid players not only through the installation process but also teach them how to play the game as well.

2. Install & Run

Since there are no external dependencies it is easier for the players to run the game. A player only needs the game folder loaded in their storage to run the game. The game is designed in a click-and-play manner; Thus, players can open this executable and run the game without any installation process. Our game consists of a single executable file named ***HadiCezmi.jar***.

3. Inside the Game

Our game has some elements that players must get familiar with before playing the game;

1. Game Glossary

- **Board:** The 25x25 unit square area where all the ***game elements*** are located.

- **Cezmi:** A ***game element*** which belongs to a player**.** These are the **Cyan** semi-circles on the ground where players can move with certain buttons;
  - The **A** key moves the **Left Cezmi** to the left.
  - The **S** key moves the **Left Cezmi** to the right.
  - The **K** key moves the **Right Cezmi** to the left.
  - The **L** key moves the **Right Cezmi** to the right.

  These are the keys by default, however, a player can change the keys if they feel the need to.

- **Ball:** A ***game element*** that a player can interact with its **Cezmi.** The player can control their **Cezmi** to hit and bounce it off their area.

- **Engel:** A ***game element*** that divides the **Board** to half. It is a stationary object which makes a **Ball** bounce off if they collide.

- **Cezerye:** A ***game element*** that appears randomly on the **Board.** It appears as a stationary **Red** object in the game. When it is hit by a **Ball,** it triggers a random **Event;**
  - **Freeze Event:** This event makes the opponents **Cezmi** freeze.
  - **Shrink Event:** This event makes the opponents **Cezmi** to shrink by half.
  - **Magnify Event:** This event makes the players own **Cezmi** become twice as big.

  All the events occur for 2 seconds, later, their effect will perish.

- **Gizmo:** A ***game element*** which belongs to a player. There are different type of **Gizmos** in the game;

- **Square Takoz:** A stationary square object which makes a **Ball** bounce off when hit. A **Square Takoz** cannot be rotated. It appears as **White** in the game.

- **Triangle Takoz:** A stationary perpendicular triangle object which makes a **Ball** bounce off when hit. A **Triangle Takoz** can be rotated while setting up the **Board** and it has 4 distinct appearances. It appears as **Red** in the game.

- **Fırıldak:** A rotating square object which makes a **Ball** bounce off when hit. A **Fırıldak** will consistently rotate around its centre during the game. A **Fırıldak** cannot be rotated. It appears as **Pink** in the game.

- **Right Tokat:** A slim rectangle which makes a **Ball** bounce off when hit. A **Right Tokat** can be rotated while setting up the **Board** and it has 4 distinct appearances. It appears as **Red** with an inner **Green** circle in the game. While playing the game, a player can also make it rotate 90^{o} ***clock-wise*** by hitting certain buttons;
  - The **T** key rotates the **Right Tokat** for the player on the **Left.**
  - The **P** key rotates the **Right Tokat** for the player on the **Right.**

- **Left Tokat:** A slim rectangle which makes a **Ball** bounce off when hit. A **Left Tokat** can be rotated while setting up the **Board** and it has 4 distinct appearances. It appears as **Blue** with an inner **Green** circle in the game. While playing the game, a player can also make it rotate 90^{o} ***counter clock-wise*** by hitting certain buttons;
  - The **R** key rotates the **Left Tokat** for the player on the **Left.**
  - The **O** key rotates the **Left Tokat** for the player on the **Right.**

2. **Main Menu**

After opening the game, players are greeted by the **Main Menu**. There are 3 options at this point to choose from;

- **Starting a New Game:** There will be 2 separate grids that indicates the corresponding players' building area. Players will be asked to set up a valid **Board** before starting the game. A valid board consists of 8 **Gizmos;** 4 for each player. Players can add different types of **Gizmos** to their corresponding building areas from their own drop-down **Gizmo** lists. A player can also remove or rotate their **Gizmos** if they are not pleased with their own **Gizmo** configuration. Once there are 8 gizmos in total, they can either hit the **Save button** to save their configured board or hit the **Play** button to select a **Level** and play afterwards.

- **Loading an Existing Game:** The game can be saved at any time while playing or building to continue playing later. This option will present a window where a player can choose a valid game file to continue a game. If the game file is incomplete or corrupt, players will receive a feedback accordingly.

- **Quitting the Game:** This option simply closes the program. Players must make sure they save their games if they don't want to lose their configured board.

3. **Playing the Game**
   The game will start either when the players load an existing game or they are done with building a new **Board.** There will be different number of **Balls** depending on the chosen **Level**. The **Ball(s)** will start moving in random direction(s) with random velocity(-ies).

1. **The Goal**
   - The Goal of the game is to land a **Ball** in the opponent's area. That way, the player earns a point.
   - If a player manages to land a **Ball** in their opponent's area after making the **Ball** bounce off a **Gizmo,** they earn twice as much points.
2. **Penalties**
   - If a player fails and lands a **Ball** inside their own area, they will lose one point instead.
   - If a player hits the **Engel,** they lose half a point.
3. **Determining the Winner**
   - When a player reaches 10 points, they are crowned as the winner.
   - When a player reaches -10 points, their opponent is crowned as the winner.
4. **Tips and Tricks**
   - The **Events** triggered by **Cezerye** are always beneficial for the player. Try to hit those before landing a goal.
   - There is a gravity factor in the game. Make sure the **Ball** is not on your side after a couple of bounces.
   - **Tokats** are very powerful tools in the game as they cover 4 times larger area than other **Gizmos** and they can rotate when needed but are harder to use. Mastering their usage is the key to success.

5. Source Code

The source code is attached to the submission folder and it is under the folder named **Source Code**.