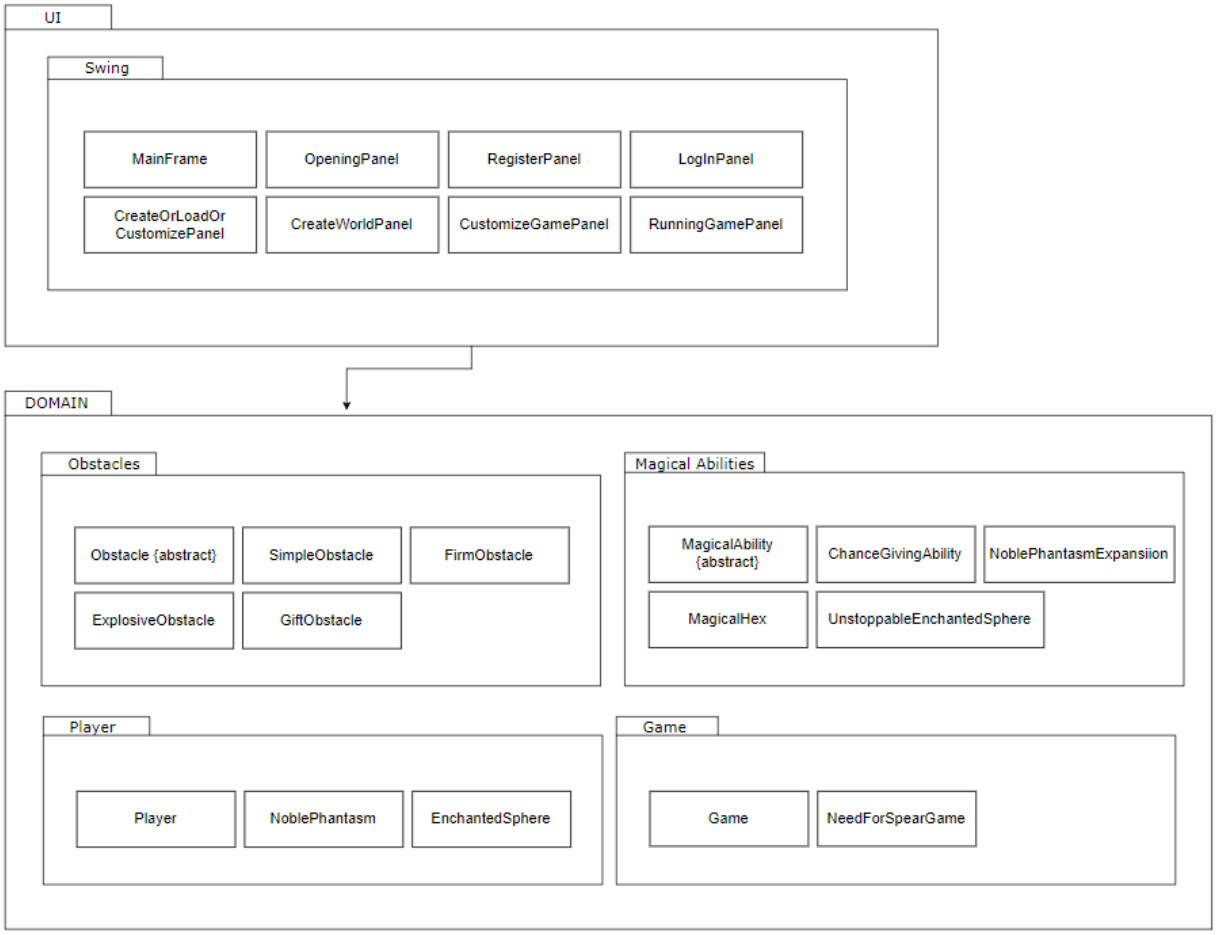
COMP-302 TERM PROJECT D1

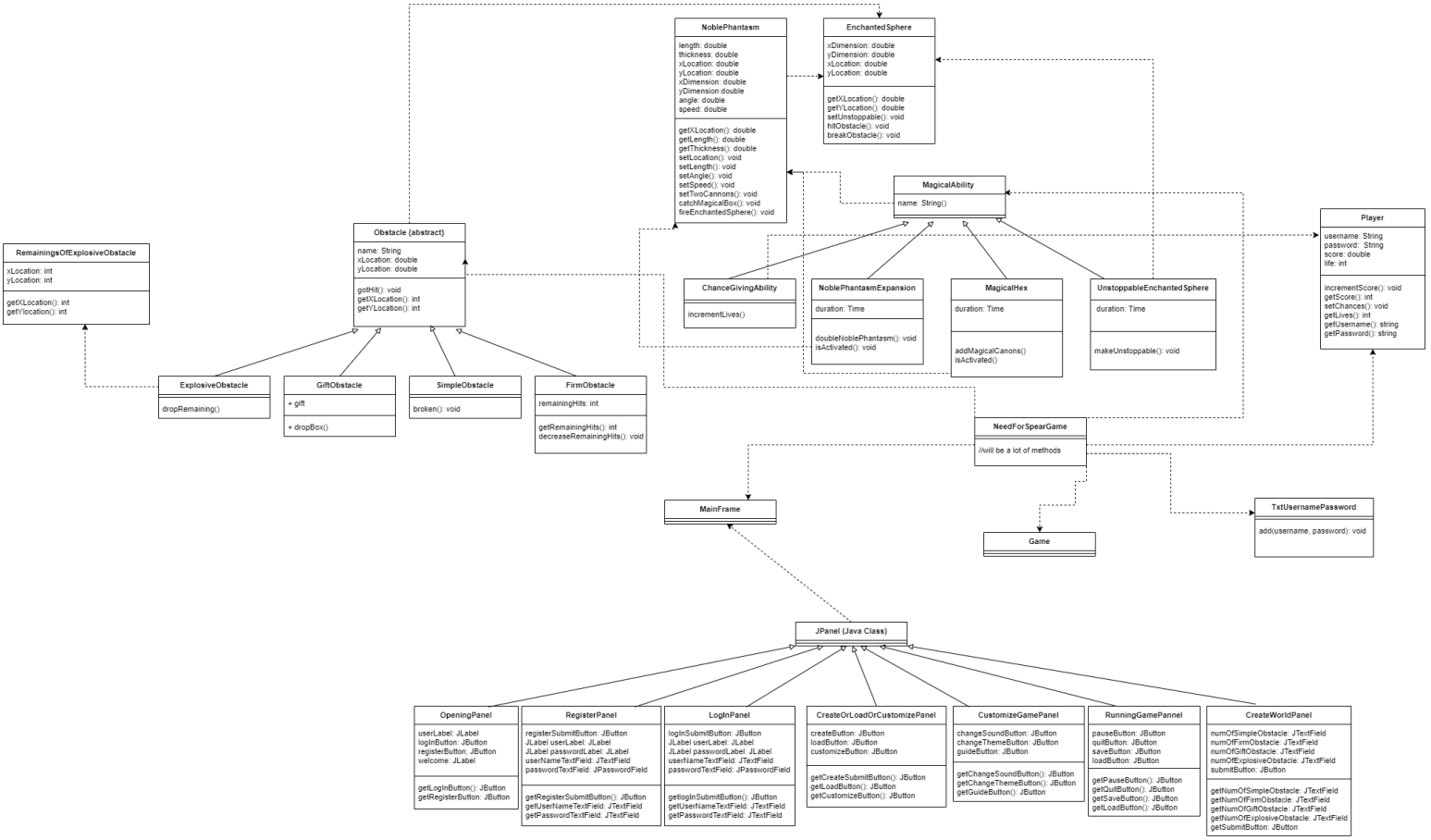
Group Name: Brogrammers
Date: 14.11.2021

Table Of Contents

UML PACKAGE DIAGRAM



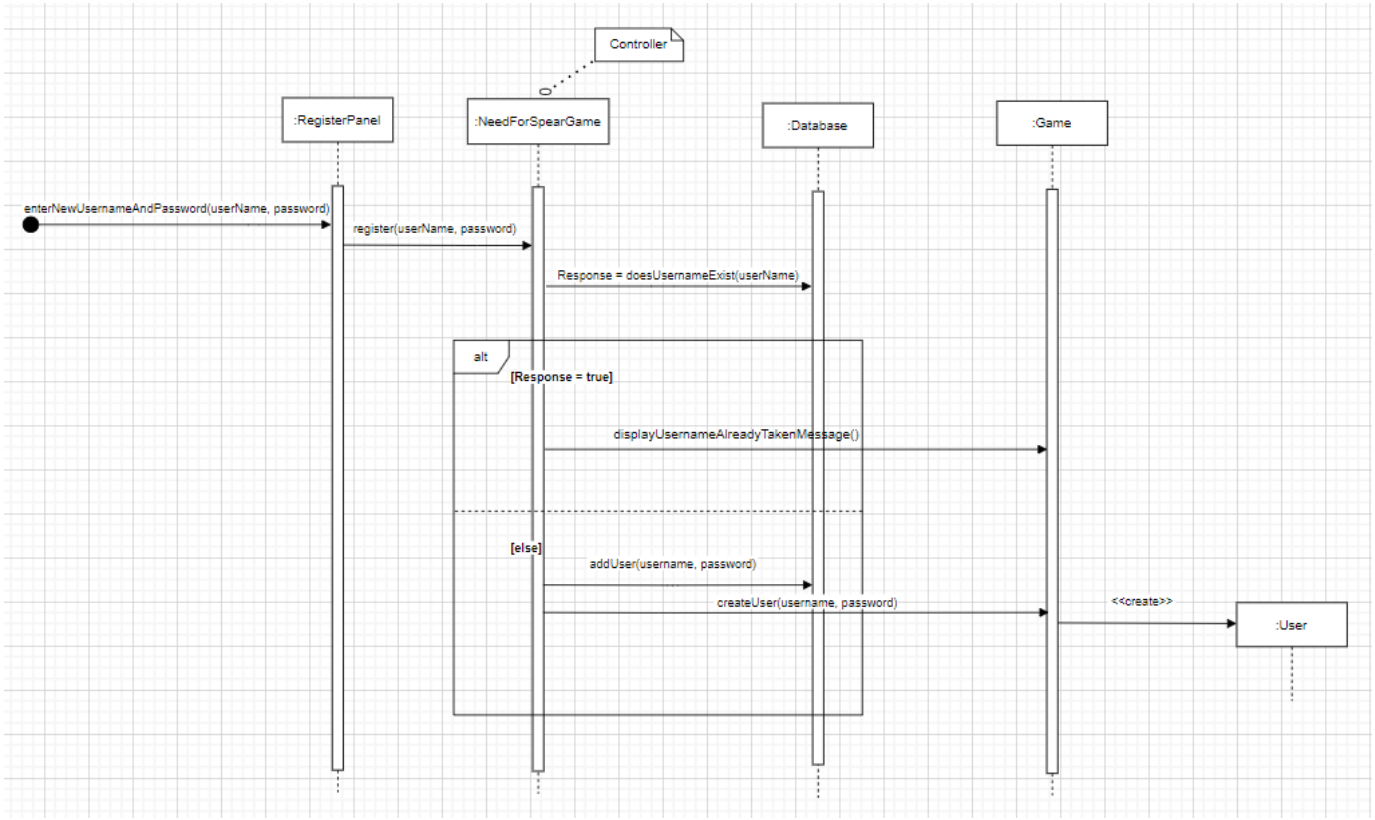
UML CLASS DIAGRAM



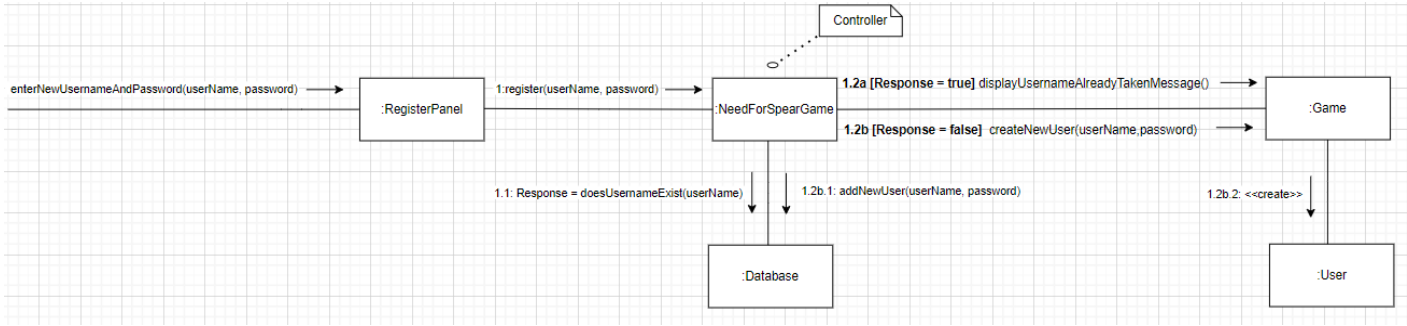
Note: Class diagram is subject to development (we did not include all methods since it would be huge, and also we did not decide all methods yet.)

INTERACTION DIAGRAMS

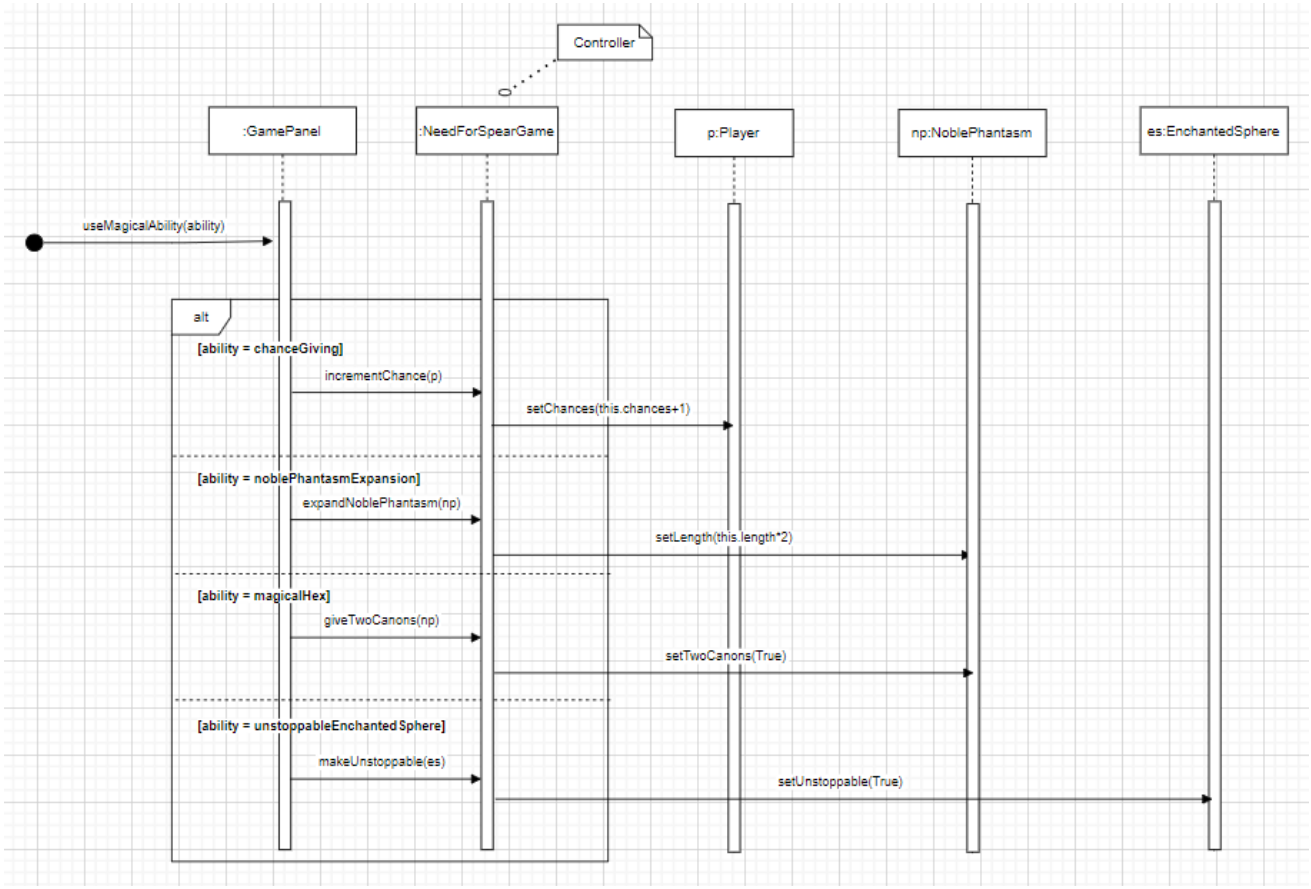
Sequence Diagram-1: *enterNewUsernameAndPassword(username, password)*



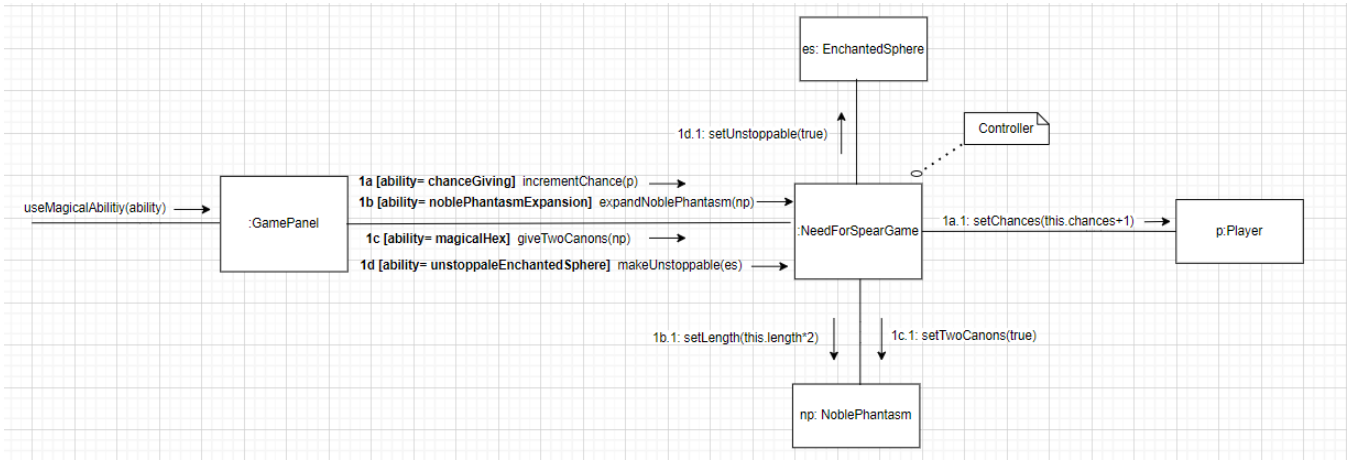
Communication Diagram-1: *enterNewUsernameAndPassword(username, password)*



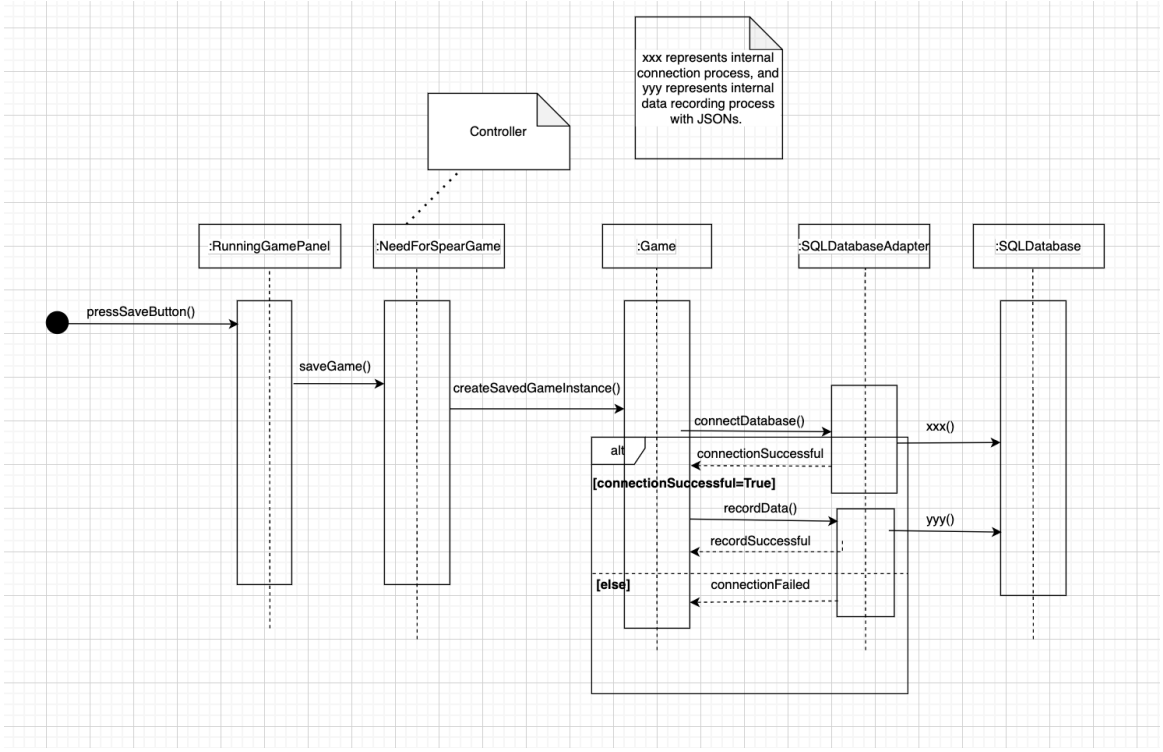
Sequence Diagram-2: *useMagicalAbility(ability)*



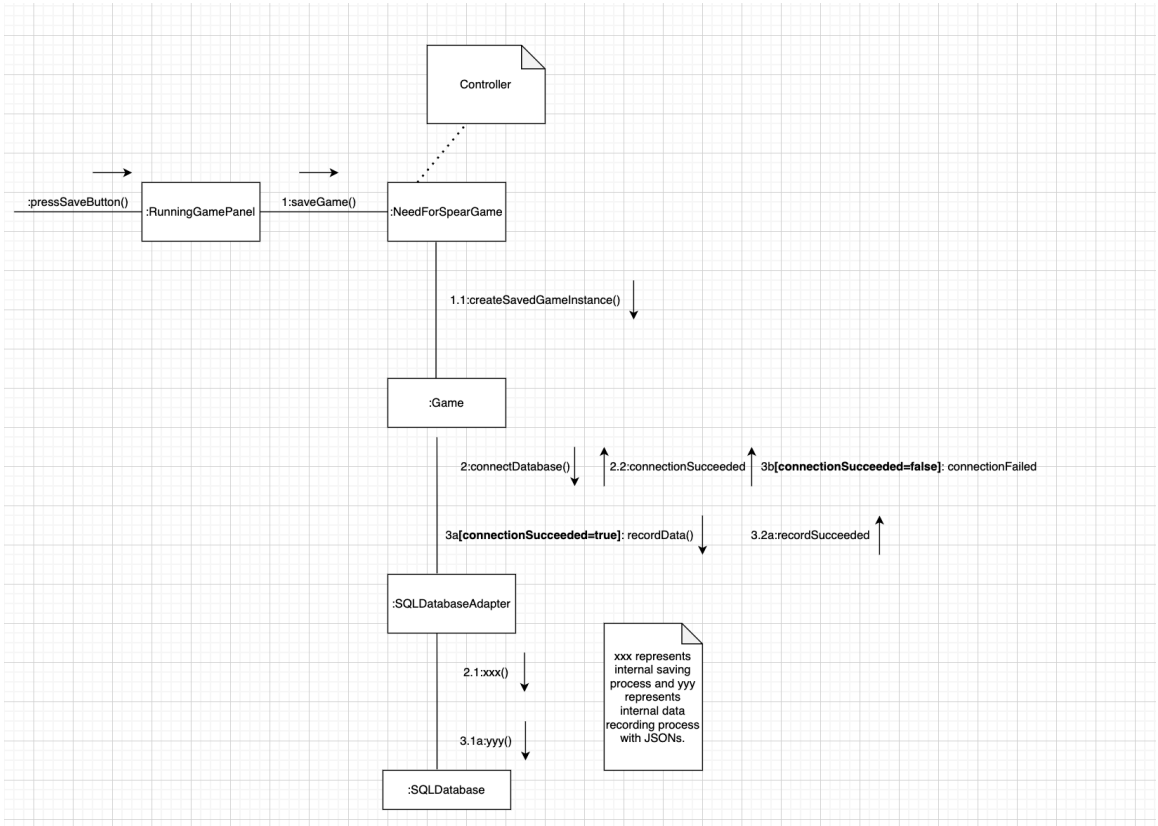
Communication Diagram-2: *useMagicalAbility(ability)*



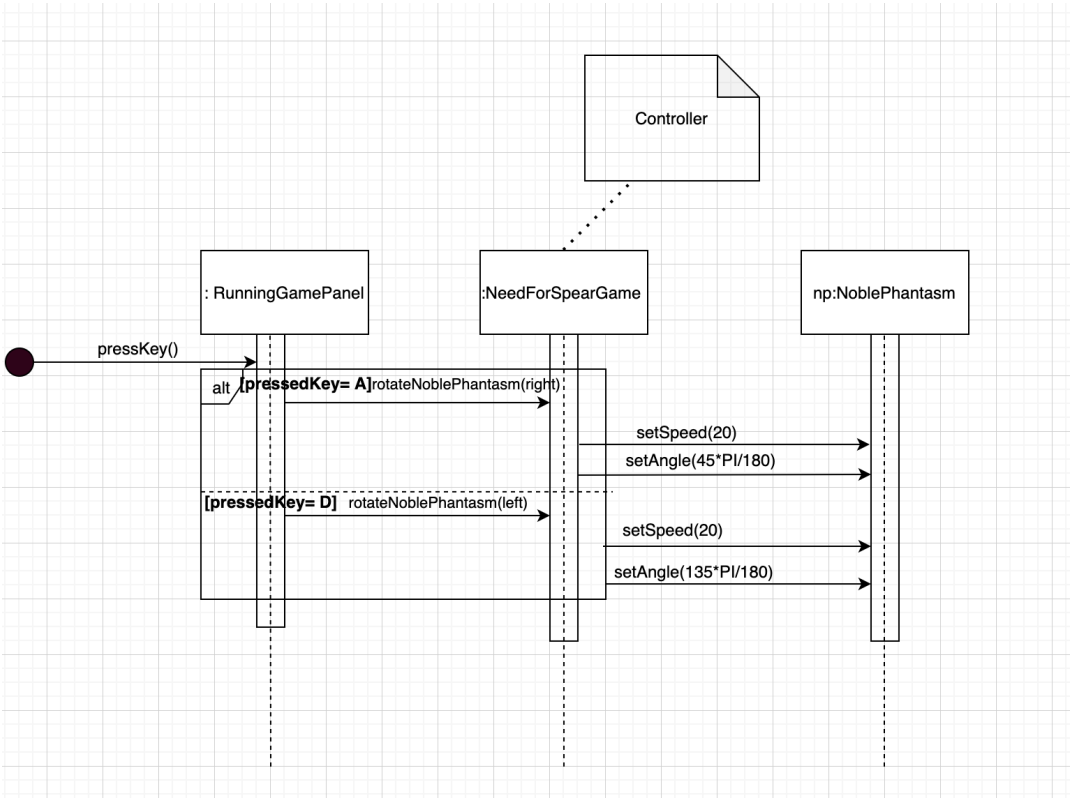
SequenceDiagram-3: *pressSaveButton()*



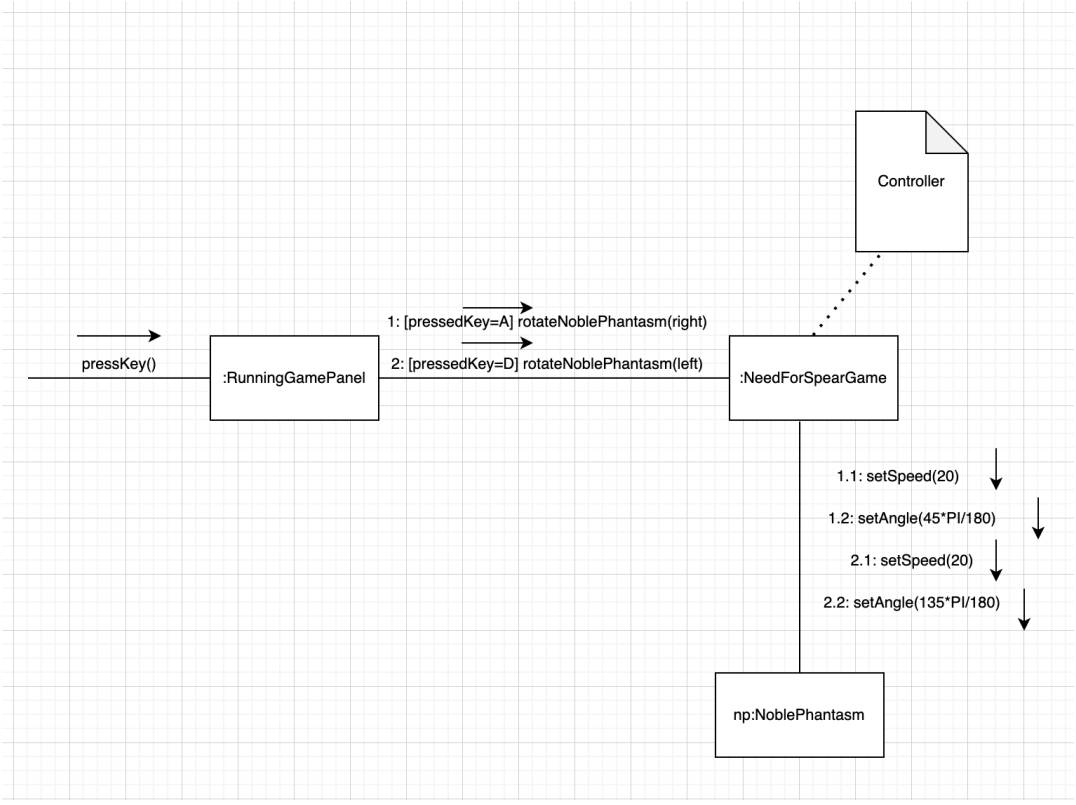
Communication Diagram-3: *pressSaveGameButton()*



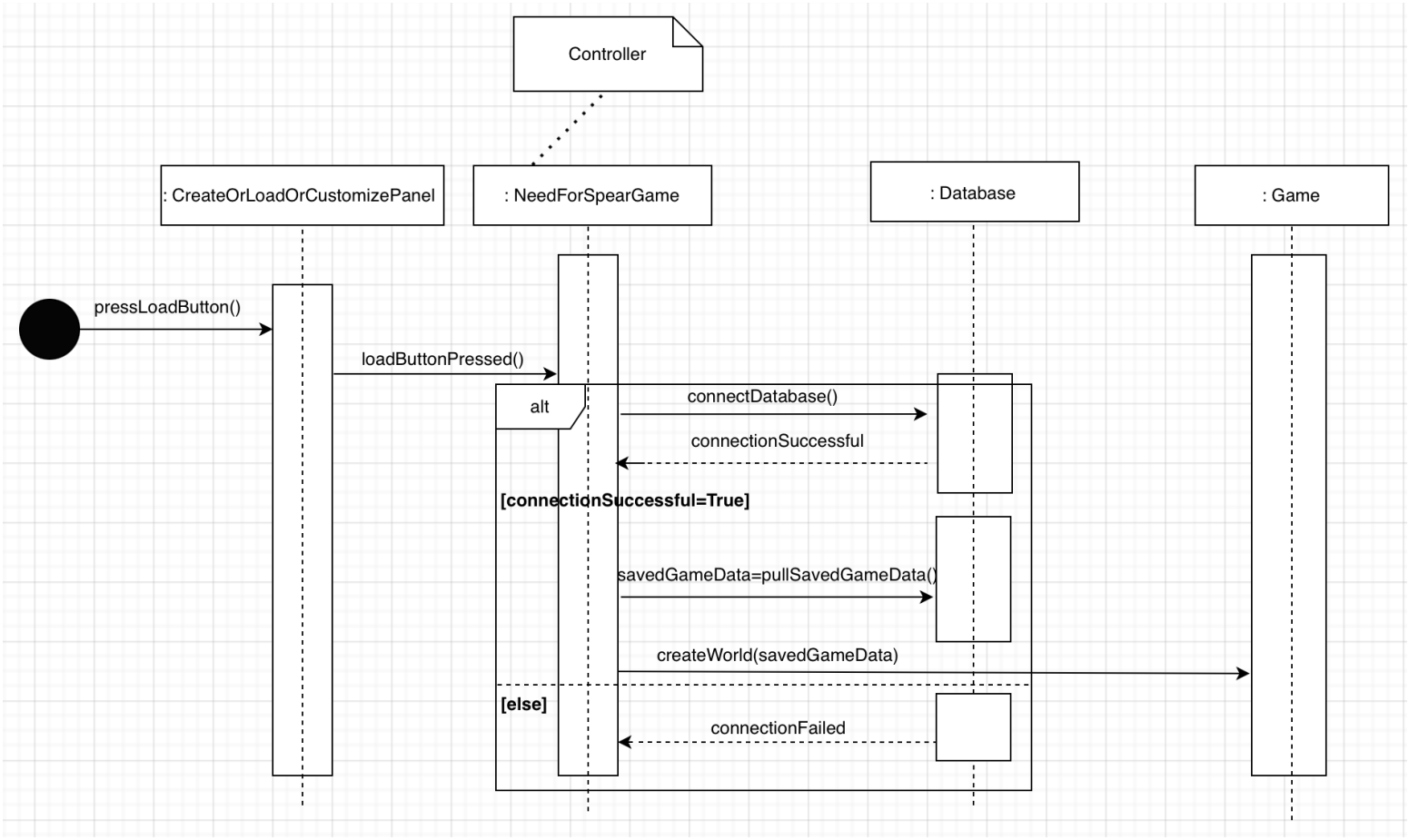
Sequence Diagram-4: *rotateNoblePhantasm()*



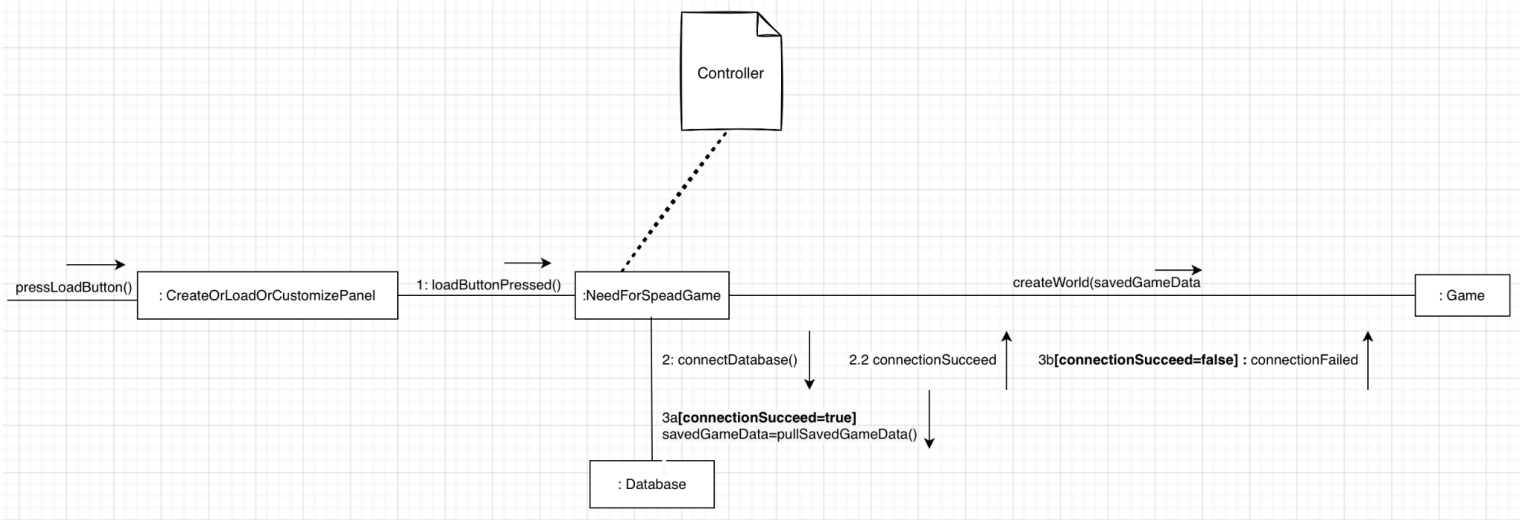
Communication Diagram-4: *rotateTheNoblePhantasm()*



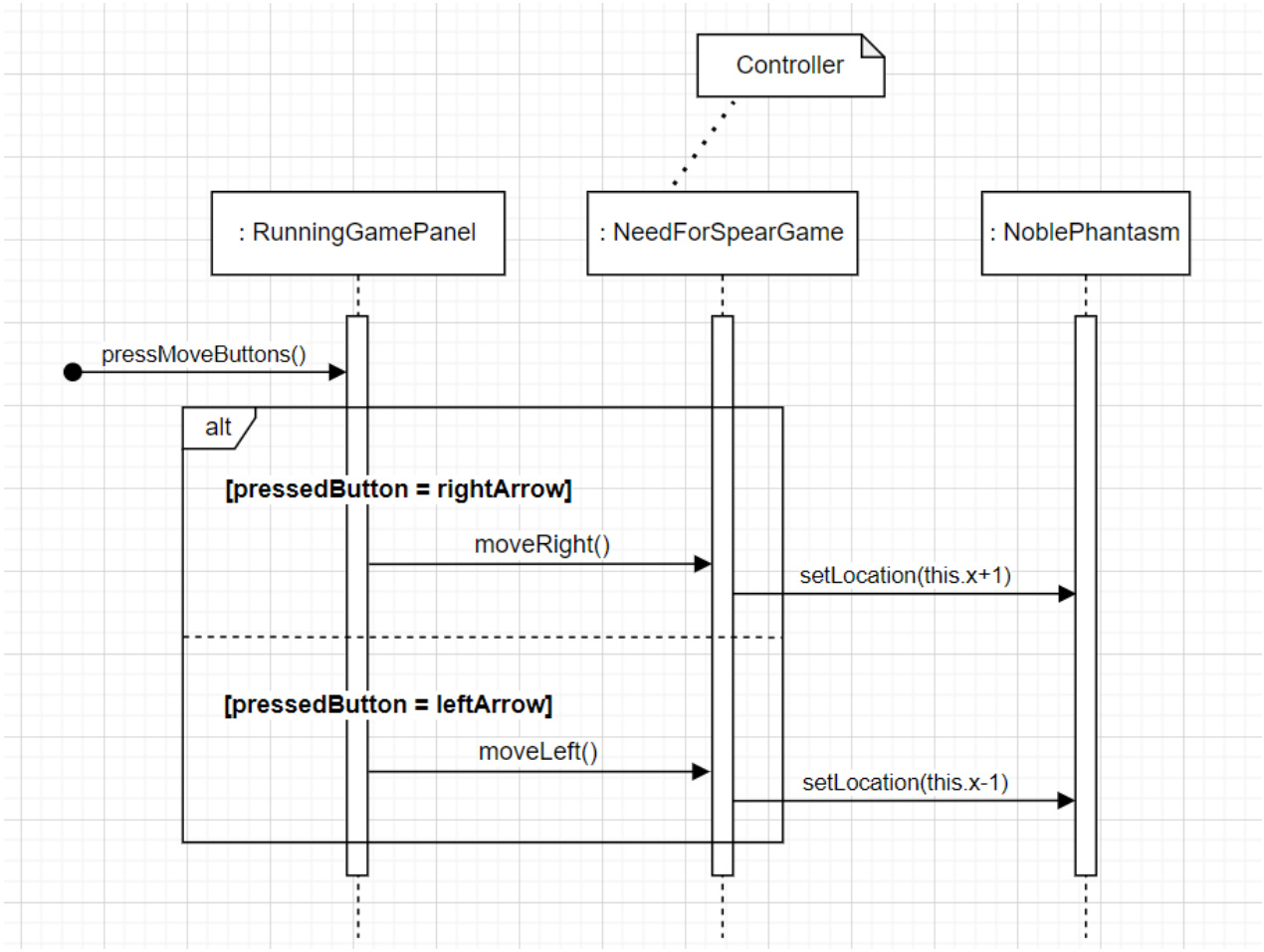
Sequence Diagram-5: *pressLoadButton()*



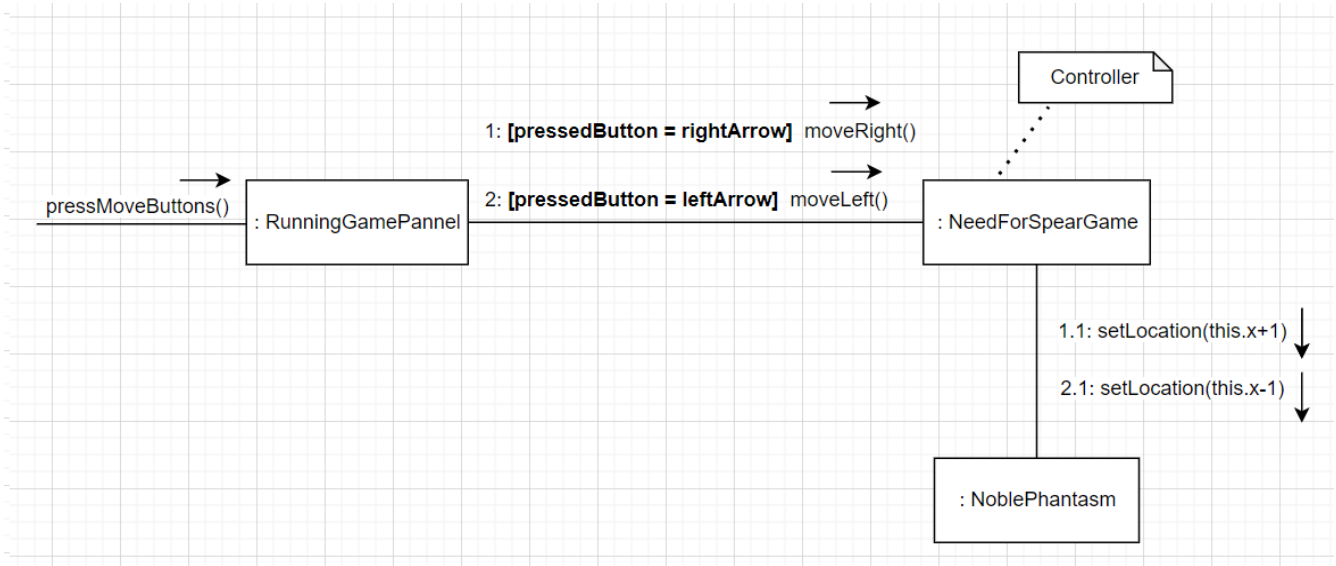
Communication Diagram-5: *pressLoadButton()*



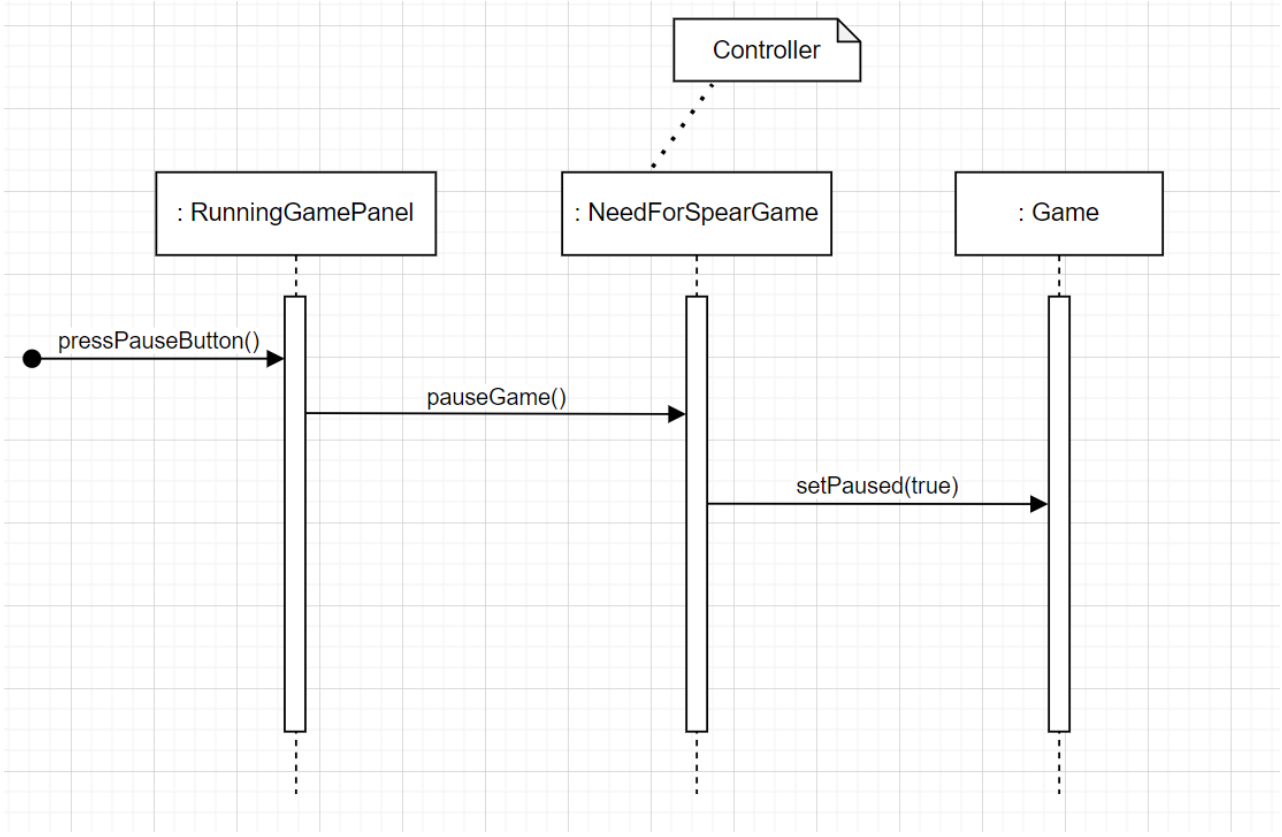
Sequence Diagram-6: *moveNoblePhantasm()*



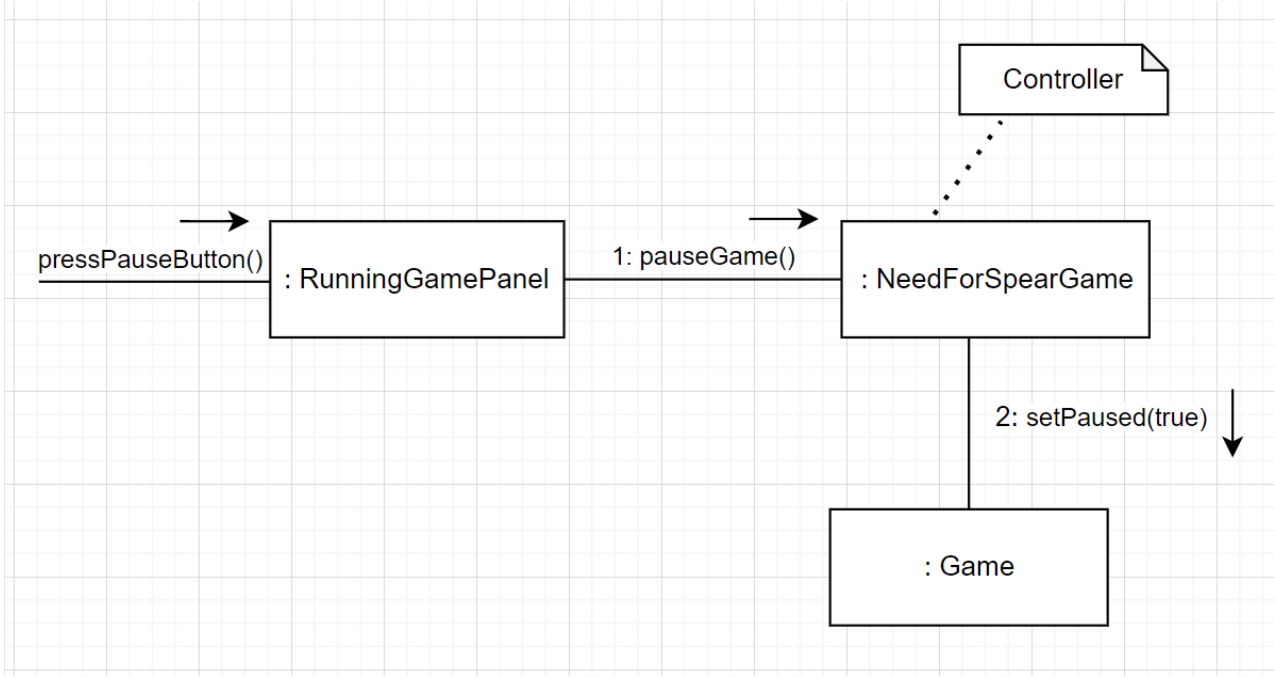
Communication Diagram-6: *moveNoblePhantasm()*



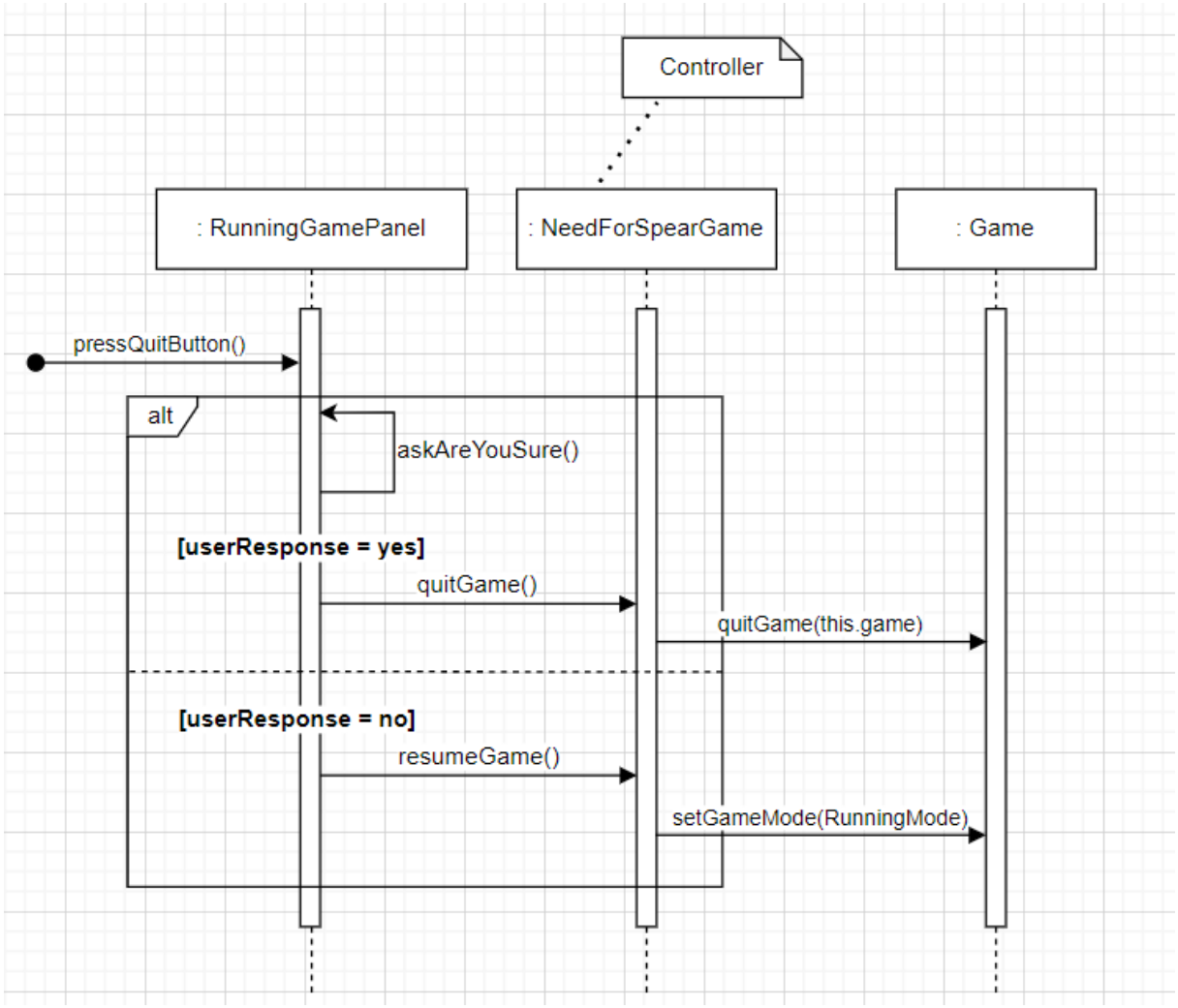
Sequence Diagram-7: *pressPauseButton()*



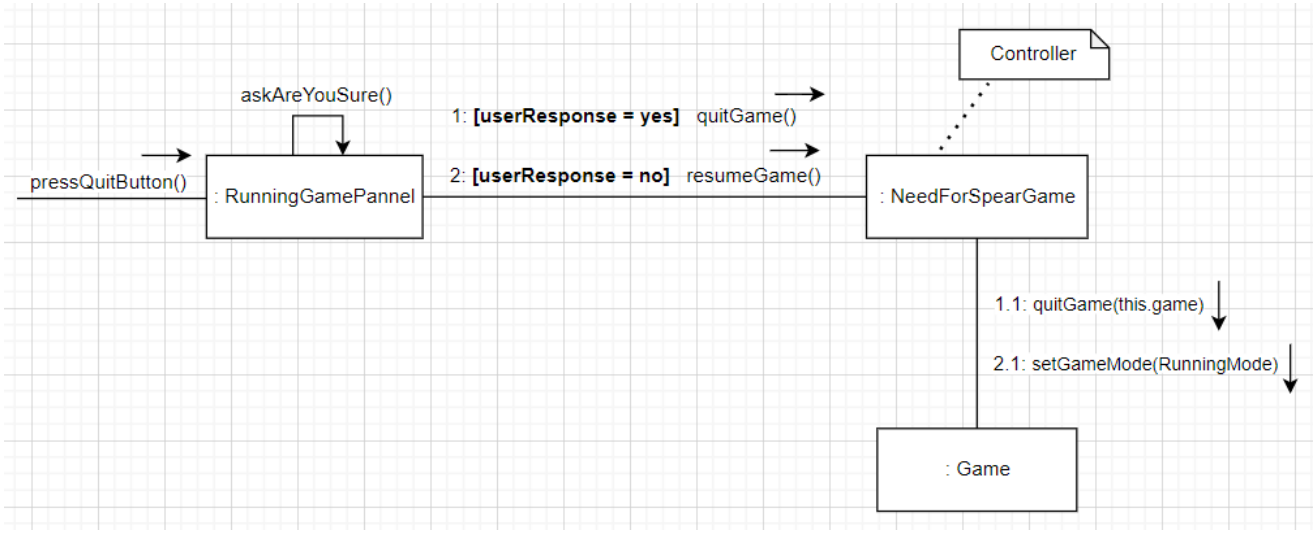
Communication Diagram-7: *pressPauseButton()*



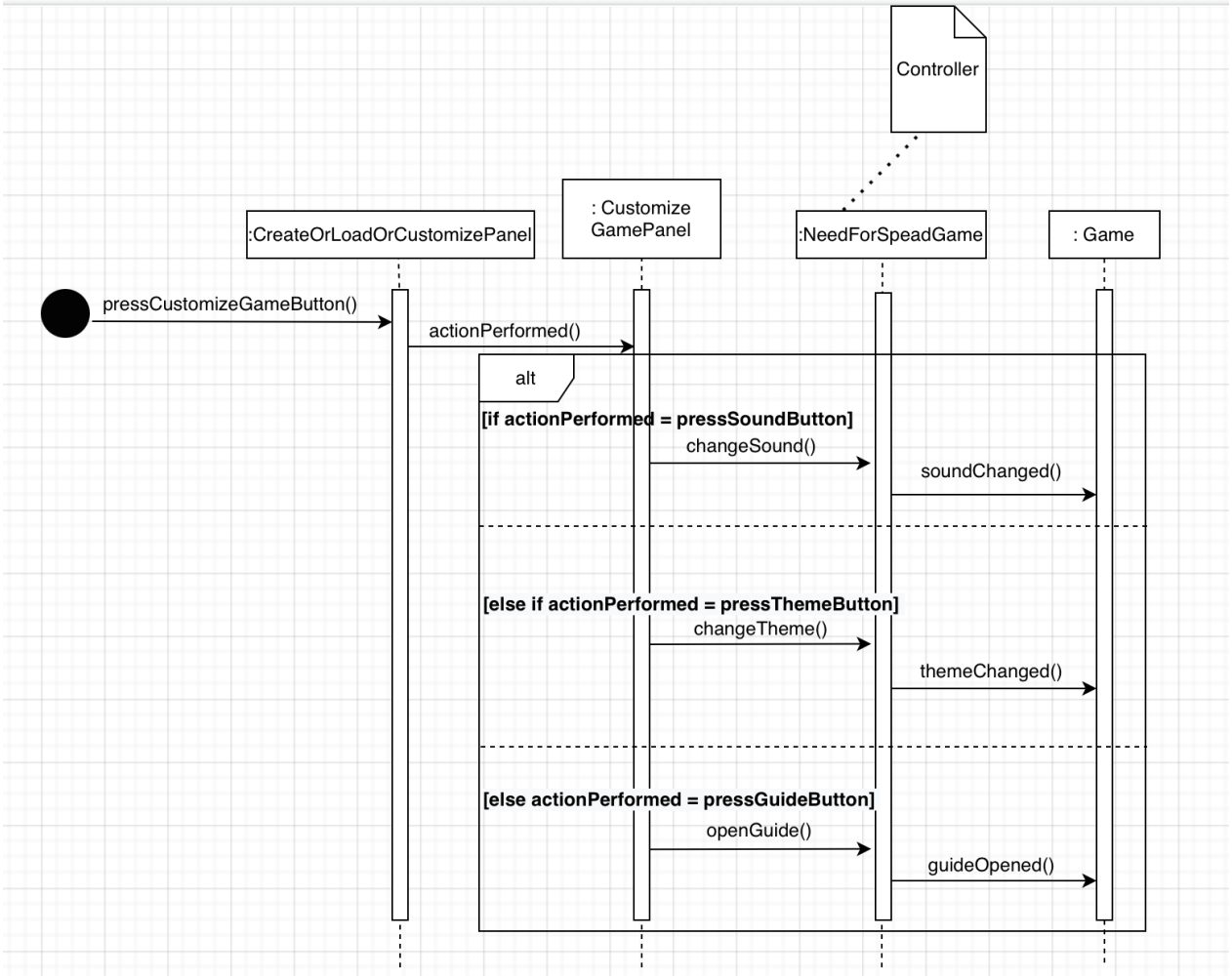
Sequence Diagram-8: *pressQuitButton()*



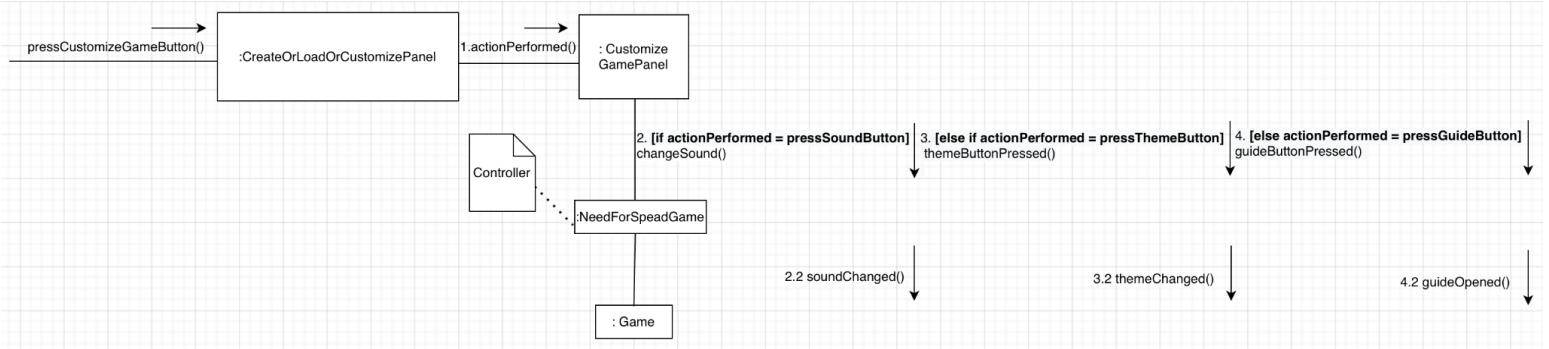
Communication Diagram-8: *pressQuitButton()*



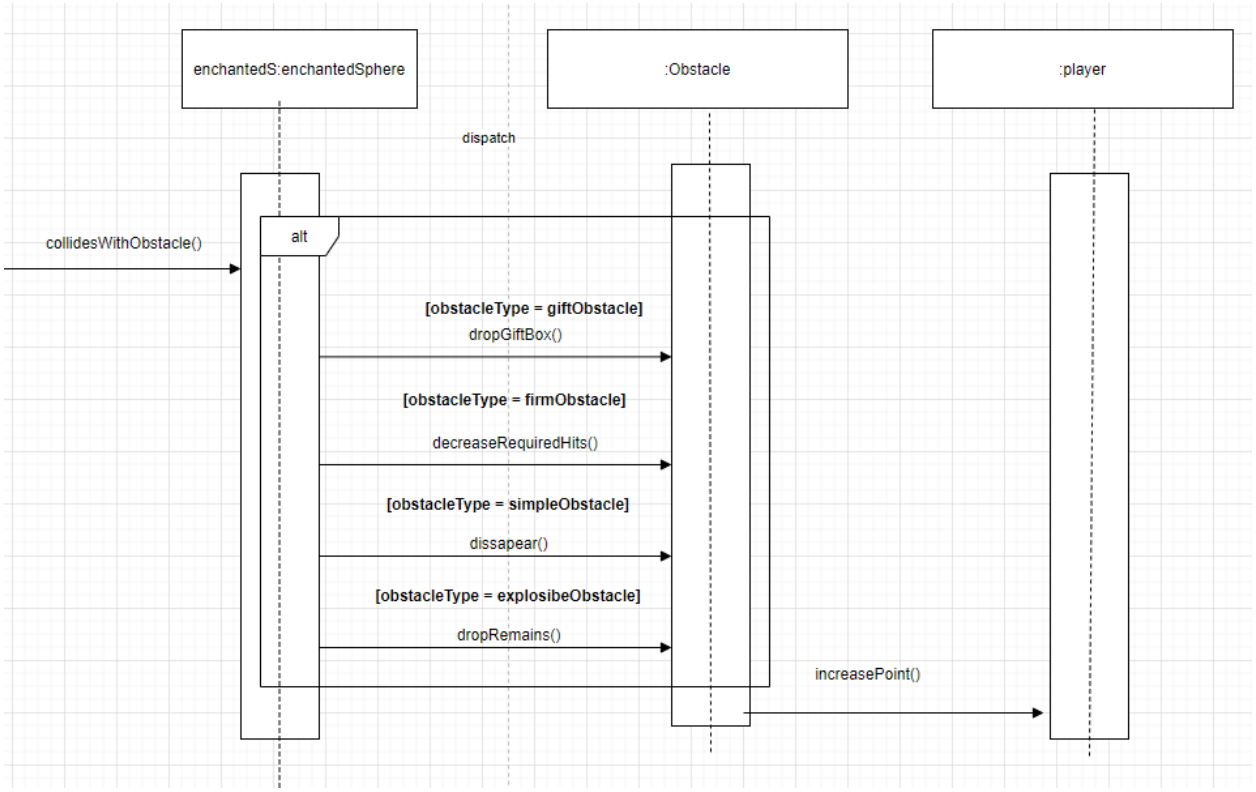
Sequence Diagram-9: *customize()*



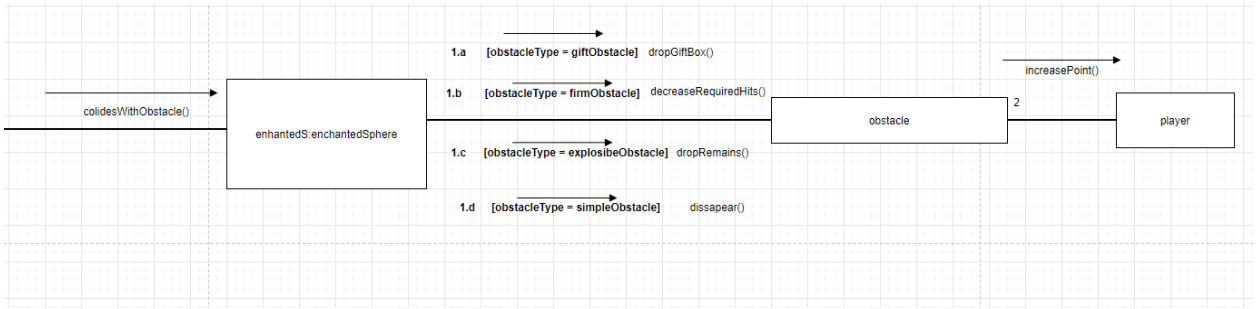
Communication Diagram-9: *customize()*



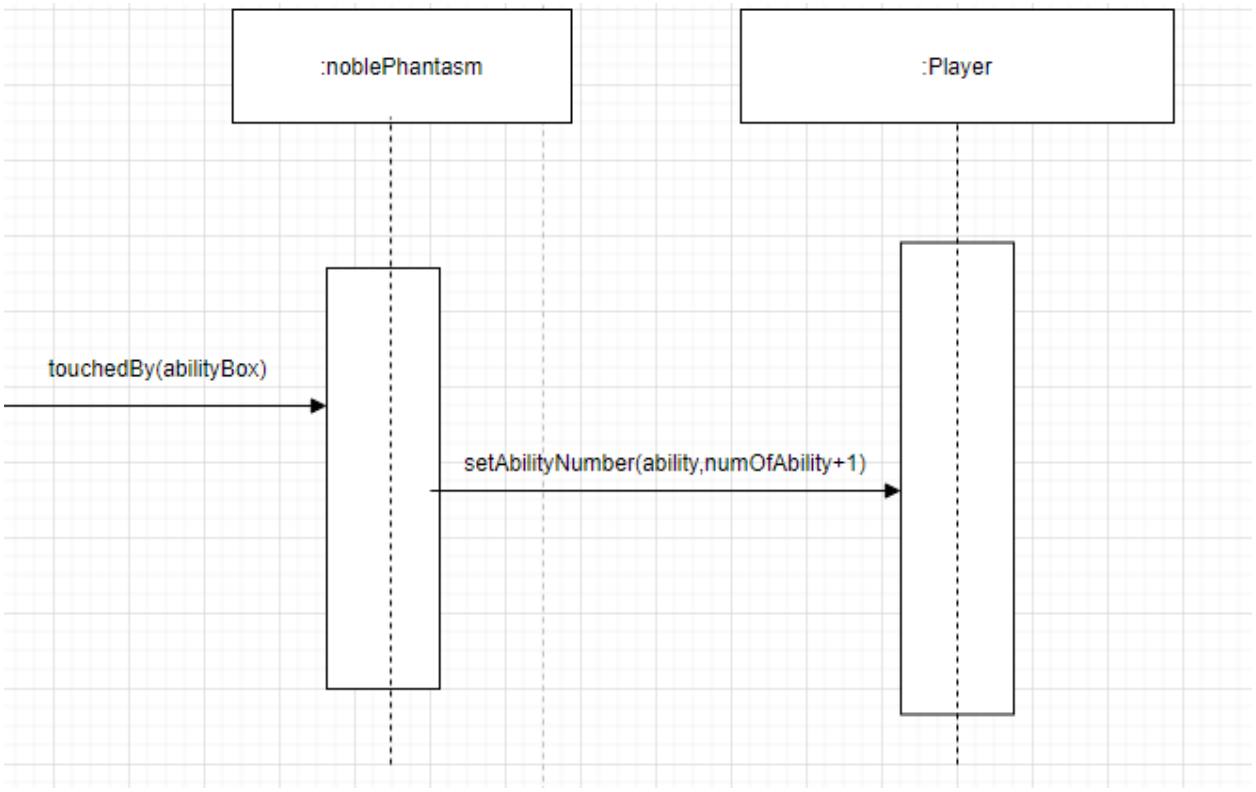
Sequence Diagram-10: *hitObstacle()*



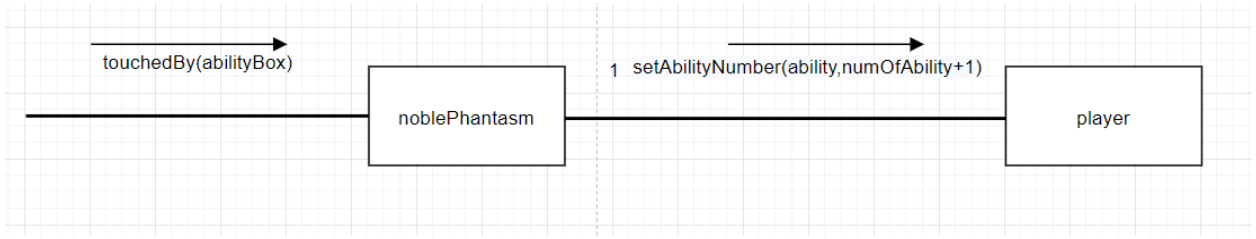
Communication Diagram-10: *hitObstacle()*



Sequence Diagram-11: *getAbility()*



Communication Diagram-11: getAbility()



DESIGN PATTERNS

Controller: We will create a controller class which is responsible for the whole game called NeedForSpearGame. It will receive inputs from the UI layer, then distribute them to the related domain layer objects. Therefore, it will also be helpful for Model-View Separation. In short, it will be the connection from the UI to the domain layer.

We can easily manage the executions of the main steps of the game (such as starting the game) by adding a few lines of code in the NeedForSpearGame (GameController) class (instead of adding more code to the related methods in the Game class). Thus, we can avoid code redundancy via the usage of the controller design pattern. Moreover, controller design pattern shortens the project development by making the project more manageable & readable.

High Cohesion: Our controller class gets input from UI and sends them to the mostly related domain objects, so it helps us achieve high cohesion.

Observer: For the passing information from domain model to UI layer, we will implement listeners to sustain model-view separation. For example, we will create a listener for Score. Whenever the score is changed, Game class will call the onScoreEvent() of each class who implements the listener of score, so the listener will notify the UI layer. Then, onScoreEvent(...) which belongs to the MainFrame class or the RunningGamePanel class, will adjust the score displayed on the screen.

While keeping track of hitting the obstacles, there is a need for an Observer pattern (when the enchanted sphere hits an obstacle, with the help of the Observer pattern; UI layer is notified and the obstacles remain unchanged or destroyed in UI). In this hitting process, we need the Observer pattern also to update the number written on the FirmObstacle. Whenever a firm obstacle is hit, with the help of the Observer pattern, the UI is notified and the number on the Firm obstacle decreases.

In the rotateNoblePhantasm() sequence diagram, the application of the Observer Pattern can also be seen. For the rotateNoblePhantasm() sequence diagram, the observers are KeyListener objects.

Polymorphism: We will use the polymorphism pattern for Obstacles (and Magical Abilities) since we have different kinds of them. Thanks to polymorphism, we will implement common methods of obstacles (like is*Hit()*) for each obstacle separately. For example when we call *isHit()* for an obstacle object, it will behave depending on the type of the obstacle. This saves us from using if-else statements for checking type and allows us to add new obstacles if the product owner wants in the future.

Factory Pattern: We will use factory pattern to create obstacles and magical abilities easily. As an example, our factory for obstacles will contain such a method:

```
public Obstacle createObstacle(String type) {
        if(type.equals("SimpleObstacle")) return new SimpleObstacle();
        else if(type.equals("FirmObstacle")) return new FirmObstacle();
        else if(type.equals("ExplosiveObstacle")) return new ExplosiveObstacle();
        else if(type.equals("GiftObstacle")) return new GiftObstacle();
}
```

Pure Fabrication: Since we will give the duty of the creation object to a new class (factory) which does not exist in our domain model instead of existing classes, we will also be applying pure fabrication and this will increase the cohesion.

Singleton: We will apply the Singleton pattern for factory classes. Instead of creating objects for factories, we will directly reach them by calling their static functions. This will enable us to save memory.
The one disadvantage can be that: For the factory classes of the Singleton, the Singleton and Factory classes are connected, and the Factory class calls the Pure Fabrication classes, so this makes the reading complex, and more complicated classes.

Information Expert: We plan to have a class named Game which contains all information about obstacles, the enchanted sphere etc. (like their locations). We will use this class for several functionalities such as detecting collisions. (Nevertheless, we are not sure whether we are going to do that. We may change our decision during implementation.) Also, in general we will give importance to assigning the responsibilities to the classes which are most related and have information to fulfill that responsibility.

Creator: In the creator pattern, the main idea is that the object which is highly associated with (for example contains) some smaller objects should be the creator of them. For the Explosive Obstacle (Pandora's Box), after the obstacle is exploded, the remains of the obstacle are created. Since the Explosive Obstacle contains the remains of the obstacle, the responsibility of creating a remain instance should be given to the ExplosiveObstacle. The usage of the creator pattern provides more manageable and easily understandable creation processes of the remains coming from the Explosive Obstacle.

In the "enterNewUsernameAndPassword(username, password)" sequence diagram (for the Register use case), the Game object creates a User object. Since a User instance is recorded by the Game instance, we can say that the "Creator Design Pattern" is also used in the sequence diagram for the Register use case.

In general, these patterns increase the complexity of our codes (by increasing the number of packages, classes etc.). But it helps us have a better design.
As another cons, following the design patterns may prevent our creativity. However, it is clear that following these patterns would provide better design than our creativity.

These patterns and strategies are powerful, and they can be used the wrong way easily, creating way more complex code to read and execute for the programmers. In short, they can be easily misused and it would be very harmful rather than being beneficial.

However, despite the cons, in general these patterns will be really useful for designing and implementing our project.

Note: We implemented Register use case by using Controller pattern, and pushed it to our GitLab repository.