SOEN 6441

Advanced Programming Practices

Instructor: Professor Amin Ranjbar

Build 3

Architectural Design Document

Team 20

Ali Molla Mohammadi …….40042597
Konstantin Chemodanov…. 40049285
Farzaneh Jamshidi…..…...…40075234
Negar Ashouri……40071530
Saman Safaei……. 40119399

March 2019

Department of Computer Science and Software Engineering

Introduction:

The purpose of this build is to add "Strategy Pattern" design pattern to the single-page web application of Online Risk game implementing an MVC architectural design model. We designed the project as web client-server application, where server running backend is implemented using Java language and client interface is using Angular. In order to implement the observer pattern design we used "WebSocket" in Spring-Boot and Angular. We implemented player behavior strategies using "Strategy Pattern" design.

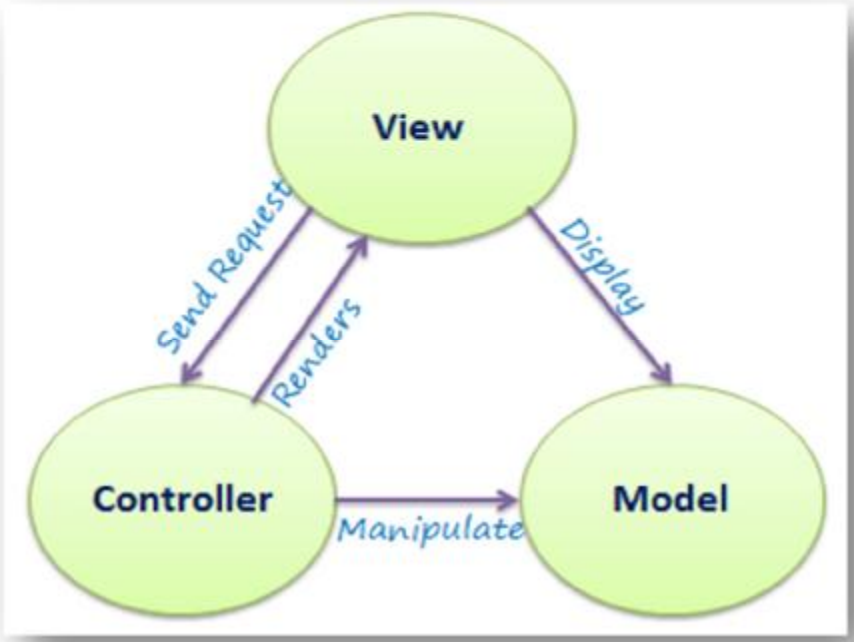
We have followed the extreme programming approach in development of our project:
Pair Programming: After dividing project into modules, two programmers or more worked on the same tasks in order to have better and quicker results and desired functionalities.
Simple Design: In order to avoid faults, software was designed as simple as possible and any extra complexity were removed.
Continuous Integration: Bitbucket is a web-based version control repository for development projects that use Git revision control systems, same as this project. Using Bitbucket made all the programmers on the same branch and eased the Maintenance of concurrent changes and errors detections by automating integrating tasks.
Collective Ownership: Using a repository also made all members able to make changes anywhere in the code anytime.
Testing: Several unit test have been written in order to make sure each unit is functioning as it should.
Coding Standards: Oracle coding standards were accepted by all programmers and was performed in the whole project. Like naming conventions, file organization, commenting conventions and etc. Following these standards made the code more readable and understandable for everyone which leaded us to a more maintainable and sustainable code that will be helpful in our further builds.
Refactoring: After receiving feedbacks of the first and second builds and requirements for the new build, deficiencies were found and refactoring has been applied as were needed, like in design patterns and player model and game modes.

Architectural Design:

Model View Controller:

The purpose of the MVC design pattern is to separate content from presentation and data-processing from content.
Separation of concerns (SoC) is a design principle for separating a computer program into distinct sections so that each section addresses a separate concern, which means each component should not do more than one thing. [2]

Model View Controller architecture aims for separation of Concerns, by dividing the components into three parts: Model, View, Controller.

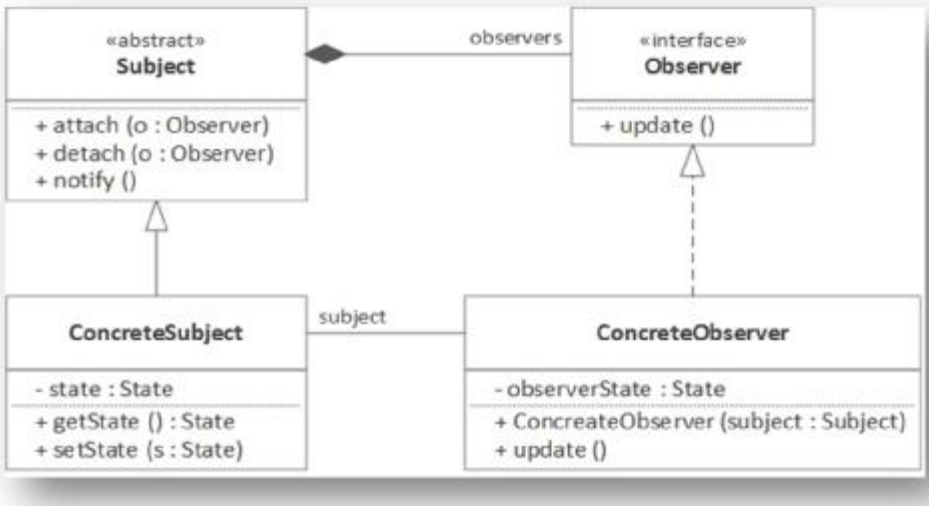


This is how MVC has been implemented in this project:

Models *(*GameMap, Player, Continent, Country, etc.) manages the data of the application domain. If the model gets a query for change state from the Views, they respond to the instruction via Controllers.
Views (SelectMap, mapGraph etc.) visualize the models and have interaction with the user. If the model data changes, the view must update its presentation as needed. Angular is a web application framework which is used to create the user interface of this project.
Controllers (StartupPhaseEndpoint, FortificationPhaseEndpoint, ReinforcementPhaseEndpoint, etc.) receive user input and initiate a response based by making calls on appropriate model objects.
The controller translates the user's interactions with the view it is associated with, into actions that the model will perform that may use some additional/changed data gathered in a user-interactive view. Controllers are also responsible for spawning new views upon user demands. [1]

Observer pattern
Observer pattern defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. It is also referred to as the publish-subscribe pattern.
In observer pattern, there are many observers (subscriber objects) that are observing a particular subject (publisher object). Observers register themselves to a subject to get a notification when there is a change made inside that subject.
An observer object can register or unregister from subject at any point of time. It helps is making the objects loosely coupled.

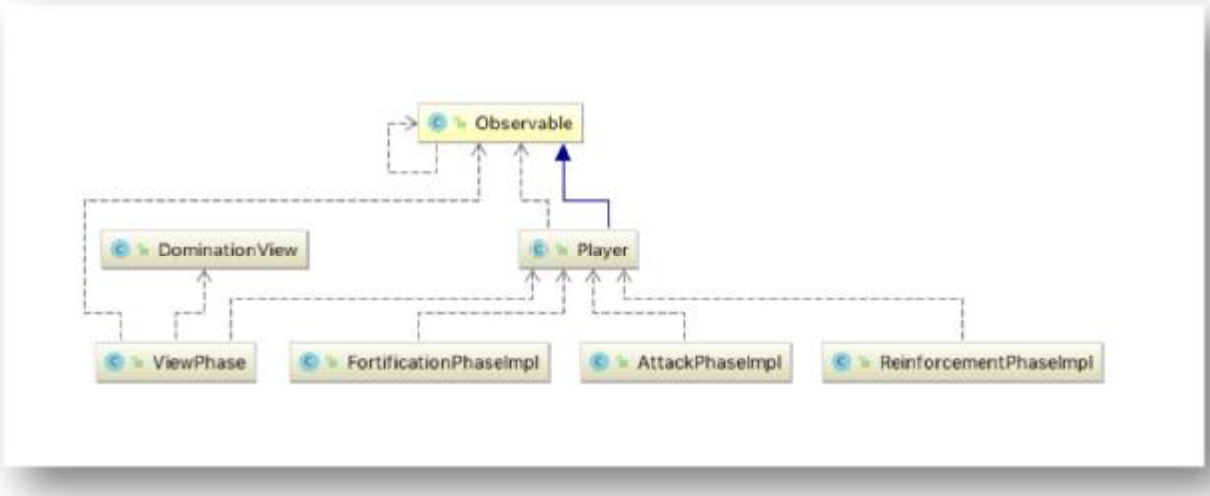
When to use observer design pattern:
As described above, when you have a design a system where multiple entities are interested in any possible update to some particular second entity object, we can use the observer pattern.
The flow is very simple to understand. Application creates the concrete subject object. All concrete observers register themselves to be notified for any further update in the state of subject.
As soon as the state of subject changes, subject notifies all the registered observers and the observers can access the updated state and act accordingly.

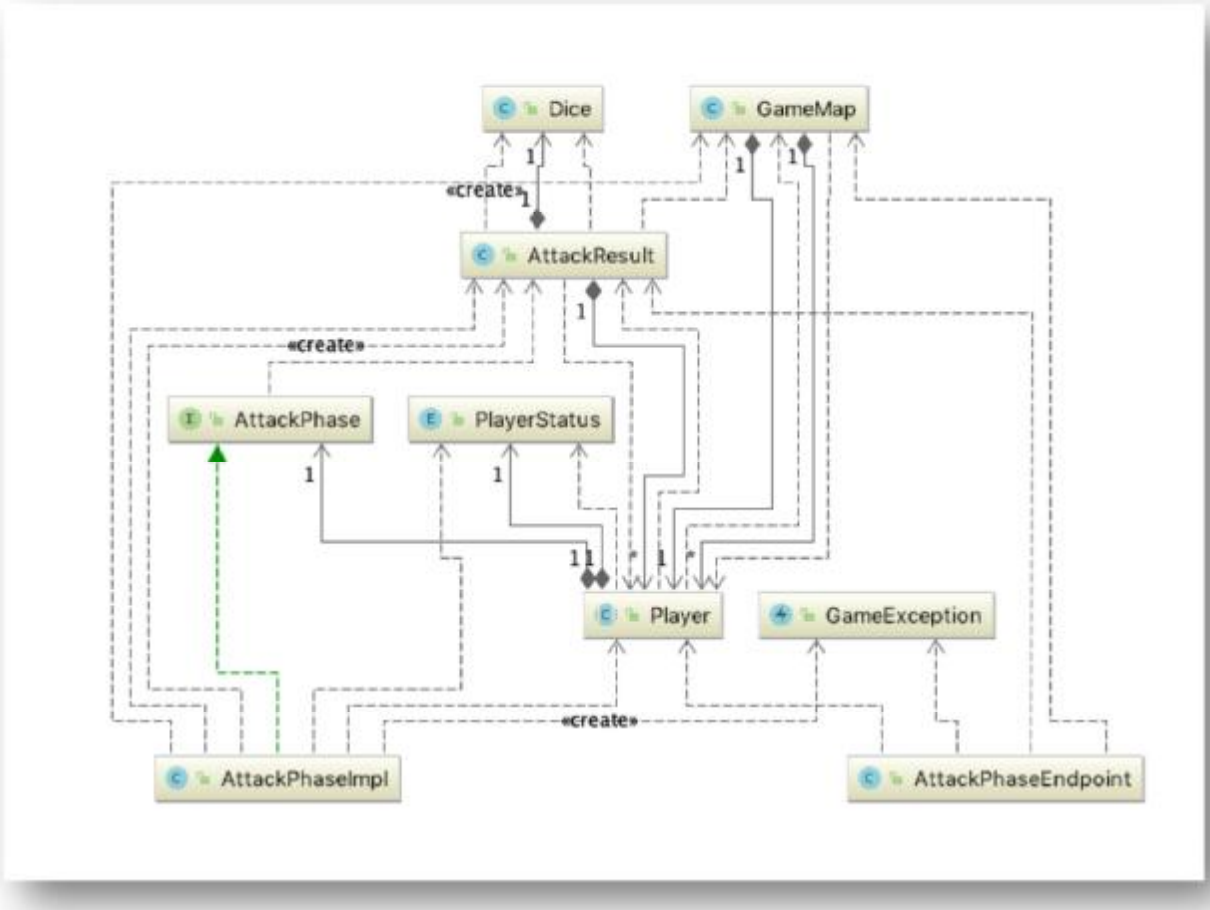


The observer pattern has four participants.

- Subject – interface or abstract class defining the operations for attaching and de-attaching observers to the subject.
- ConcreteSubject – concrete Subject class. It maintain the state of the object and when a change in the state occurs it notifies the attached Observers.
- Observer – interface or abstract class defining the operations to be used to notify this object.
- ConcreteObserver – concrete Observer implementations. [3]

Following diagrams are showing how we have implemented observer pattern in this project:





We have used webSocket in our project in both backend and frontend in order to apply the observer pattern for player view phase, players world domination view phase and card exchange view.

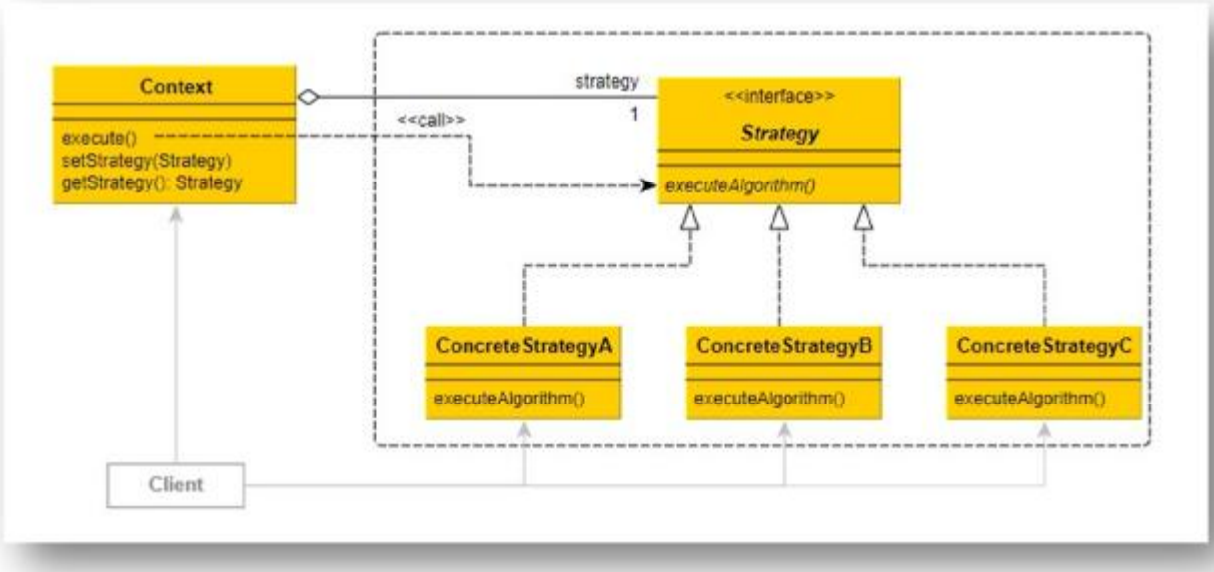
Strategy pattern:

In computer programming, the strategy pattern (also known as the policy pattern) is a behavioral software design pattern that enables selecting an algorithm at runtime. Instead of implementing a single algorithm directly, code receives run-time instructions as to which in a family of algorithms to use.

Strategy lets the algorithm vary independently from clients that use it Strategy is one of the patterns included in the influential book Design Patterns by The Gang of Four that popularized the concept of using design patterns to describe how to design flexible and reusable object-oriented software. Deferring the decision about which algorithm to use until runtime allows the calling code to be more flexible and reusable.

For instance, a class that performs validation on incoming data may use the strategy pattern to select a validation algorithm depending on the type of data, the source of the data, user choice, or other discriminating factors. These factors are not known until run-time and may require radically different validation to be performed. The validation algorithms (strategies), encapsulated separately from the validating object, may be used by other validating objects in different areas of the system (or even different systems) without code duplication.

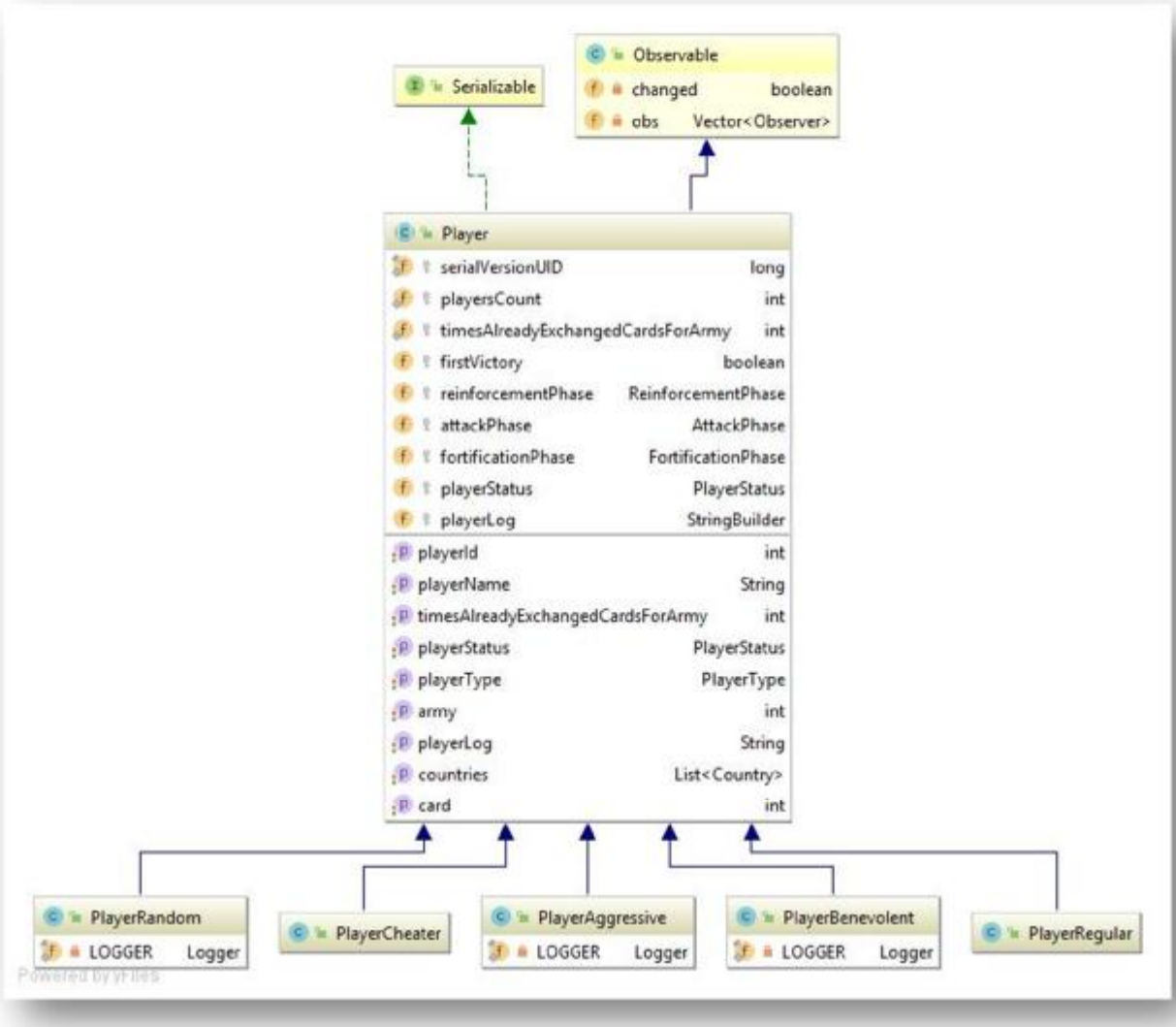
Typically the strategy pattern stores a reference to some code in a data structure and retrieves it. This can be achieved by mechanisms such as the native function pointer, the first-class function, classes or class instances in object-oriented programming languages, or accessing the language implementation's internal storage of code via reflection. [4]



We have used "strategy pattern" design in order to implement four new classes for four different behavior strategies of players:

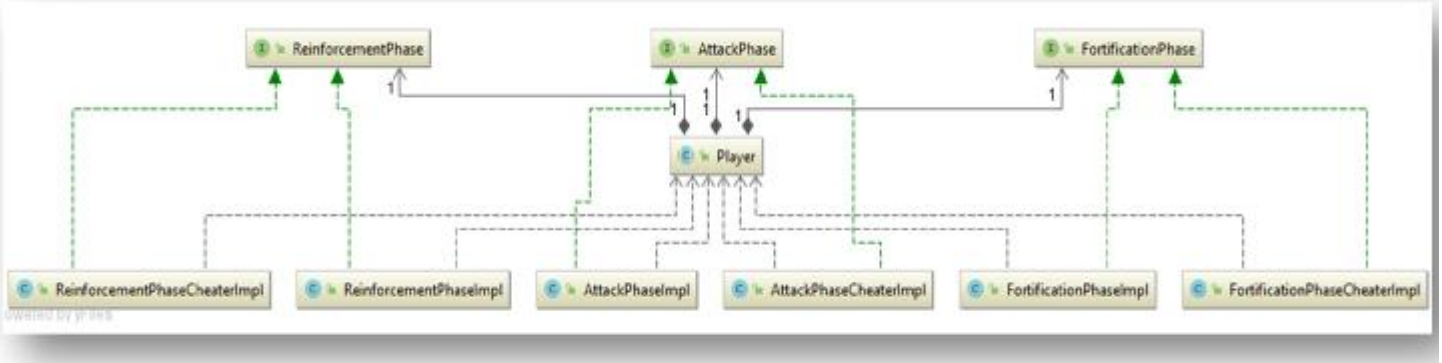
• An aggressive computer player strategy that focuses on attack (reinforces its strongest country, then always attack with it until it cannot attack anymore, then fortifies in order to maximize aggregation of forces in one country).

• A benevolent computer player strategy that focuses on protecting its weak countries (reinforces its weakest countries, never attacks, then fortifies in order to move armies to weaker countries).
• A random computer player strategy that reinforces random a random country, attacks a random number of times a random country, and fortifies a random country, all following the standard rules for each phase.
• A cheater computer player strategy whose reinforce() method doubles the number of armies on all its countries, whose attack() method automatically conquers all the neighbors of all its countries, and whose fortify() method doubles the number of armies on its countries that have neighbors that belong to other players.
All of these classes are children of Abstract class "player".



Game phases are all interfaces, using the strategy pattern to change the way each phase call methods. Based on the type of the player behavior, each phase will do some specific calculations for all parameters and arguments and based on these calculations it will call methods. We have an additional implementation in every phase for the "cheater" as it does not follow the game rules.

In following diagrams you can see how "Strategy Pattern" is implemented in our project:



References

[1] [Online]. Available: [Online]. Available: Tutorialsteacher.com. (n.d.). MVC Architecture. [online] Available at:https://www.tutorialsteacher.com/mvc/mvc-architecture.

[2] [Online]. Available: [Online]. Available: En.m.wikipedia.org. (n.d.). Separation of concerns. [online] Available at:https://en.m.wikipedia.org/wiki/Separation_of_concerns.

[3] [Online]. Available: [Online]. Available: Gupta, L. (n.d.). Observer Design Pattern - Observer Pattern in Java - HowToDoInJava. [online] HowToDoInJava. Available at: https://howtodoinjava.com/design-patterns/behavioral/observer-design-pattern/ .

[4] [Online]. Available: En.wikipedia.org. (n.d.). Strategy pattern. [online] Available at: https://en.wikipedia.org/wiki/Strategy_pattern .