SOFTWARE ENGINEERING PROJECT RELATION

- Introduction

This year, the Final Examination of the Software Engineering course asked the students to implement a client-server application that allows users to play the Santorini game, an abstract strategy board game.

- Architecture and Design Choices

The application is built using the Model View Controller architectural pattern: this has allowed us to manage separately all the game-specific logic and objects and the game-agnostic network and visualization requirements. The Model, as represented in the UML diagram included in the deliverables of the project, aims to depict a full match of the game. Our approach was to statically design all the elements in the game (such as towers, board, workers, gods) then to define a “Action” object (the “command” class) that encapsulates a command issued by the player, that contains all the information to modify the board and all the objects accordingly.
For the design of the Gods and the Gods’ powers we decided to apply standard java inheritance because we thought that all the override / overload mechanism would be useful to detail each god’s specific power, but also allowed us to reuse all the code in the abstract God class when a “standard” implementation of an action is required. We avoided solutions that involve decorator or proxies for that module.
The core of the Model module is the Board, where all the match objects are stored and organized. The board exposes an interface of public methods that allows players and the controller to modify the course of the game. It also contains and manages the game logic responsible for the handling of “wrong moves” such as out-of-bound movement or placing of workers. A fundamental part of this controls is also handled by the Gods themselves.
The Controller module includes the Controller itself and the two classes Command and PlayerCommand, that encapsulates the previously mentioned “player’s action”.
The View module contains all the classes to design and draw the CLI and the GUI, and also contains the server-side class RemoteView, that completes (alongside the Server and the ClientHandler classes) the distributed-MVC server side application. The remote view is in charge of the communication with the client during actual play time, while the Server and the ClientHandler classes manage the connection during game set-up and player’s login. As we designed the network interaction, we realized that the Board was the object that has to be sent to client, because it perfectly summarizes the game state, as a snapshot. To avoid sending the full board, we applied the Proxy pattern to it, creating a proxy object (the BoardProxy) that works as a static representation of the board itself, adding some other useful information the Client needs to properly describe the game state. To link the model to the view to the controller we implemented an Observer-chain:

Project name: Santorini
Elia Ravella
Marco Re
Gianluca Regis

every update in Board (issued by the Controller) also updates the BoardProxy (the first Observable) that is observed by the RemoteView. The RemoteView, on his update, sends the BoardProxy to the client. The RemoteView is also observed by the Controller: this is needed to notify the Controller of the commands the client has sent, to verify them and apply them to the Model.
The Observer pattern is also applied client-side: the BoardListener class encapsulates the functions and procedures to read from the socket and notify whenever a BoardProxy arrives from the server. This class is observable: every panel of the GUI and every element of the CLI re-implement a sub class Observer of BoardListener to receive updates.

- Network interaction
As depicted in the included UML interaction diagram, we focused on creating a simple network interaction, building apposite “container” classes (such as BoardProxy and PlayerCommand) to deliver the exact amount of information needed to perform an action. With the exception of the login phase (where the exchange is made with Strings parameters) all the communication is made via PlayerCommand (sent by the client) and BoardProxy (sent by the server). The client and server are semi-synchronous: every PlayerCommand sent by a player triggers a BoardProxy to be sent to all clients, but there could be updates that require more than a BoardProxy to be described; also, the client is not supposed to answer immediately with a command after an update.

- Disconnection management
The application handles disconnections. It can detect a client disconnection from the server (this causes the match to end) and a server disconnection from the client (this also triggers an end of the match).

- Tools used
We used IntelliJ Idea as IDE, GIT as VCS, Maven for the management of the dependencies and for the artifact creation, JUnit for the unit tests. For the synchronization inside the team, we used a Trello as board to manage the tasks assigned to each person. For the other documentation, we used draw.io and StarUML for the diagrams.

- Conclusion
The creation of this application, with all the design effort before it, gave us the possibility to learn a lot about software design and architecture, team work, and project and time management, alongside the technical competences we acquired.