Concordia University

Department of Computer Science and Software Engineering

Advanced Programming Practices

SOEN 6441 - Fall 2018

Software Architecture & Methodologies

Team 35

Suruthi Raju	4008 4709
Rohit Sharma	4008 2190
Hamid Reza Anvari	2713 4541
Jatan Gohel	4007 8112

Guided by

Dr. Joey Paquet

Architectural Design Document

Introduction

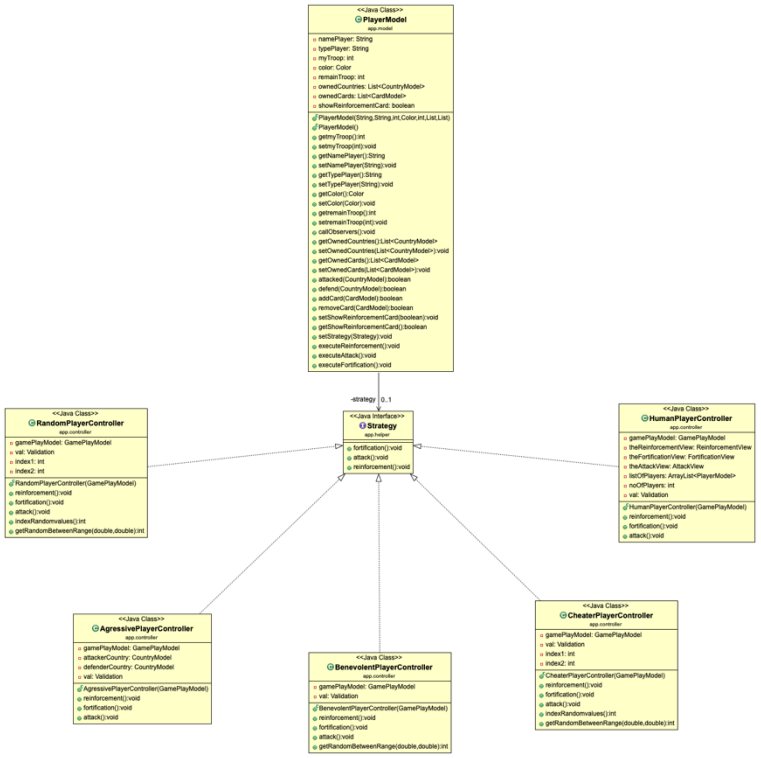
Our purpose is to build an Online Risk game implementing a Model View Controller (MVC) architectural design model. We have studied and made sure we follow the extreme programming approach for the smooth development of software by implementing features like:

- **Pair Programming:** There were modules where more than two programmers worked on the same tasks to complete the functionalities. This not only increased the pace, but also the productivity. It efficiently resulted in knowing our responsibilities too.
- **Simple Design:** Complex designs were avoided in order to make the software more user-friendly, and to also avoid faults. Smooth features were implemented with the help of Simple designs which resulted in quick, structured and clear game flow.
- **Continuous Integration:** Git was used as Revision Version Control which made all the programmers on the same branch. Maintenance of concurrent changes, rollback sequences increased the productivity by automating integrating tasks. Errors detected by integration were rectified quickly.
- **Coding Standards:** All the programmers were on the same page to follow all the coding standards including the naming conventions, File organization, etc. which helped us to peer reviewing and maintaining the source codes. Because of the standardized implementation, the sustainability of the source codes would be helpful in further builds.
- **Refactoring:** Change in the Views, PlayerController, Console panel for bot movements in the game

1. **Change in the Views:** Refactoring multiple Views in order to incorporate bots with the flow of game.

2. **PlayerController:** To add <u>strategies</u> for each player *viz.* Aggression bot, Random bot, Benevolent bot and Cheater bot, Strategy Pattern was implemented. Encapsulation of Reinforcement, Attack and Fortification phase is done in *PlayerController.* In *GamePlayModel,* the data flows into data flow model object and updates the view whenever data changes.

3. **Console panel for bot movements in the game**: To manage and track bot movements in the game, we refactored *PlayConsoleView*



Model View Controller

Model View Controller architecture aims for separation of Concerns, meaning the components should not do more than one thing by dividing it into three parts a Model, a View and a Controller.

Model: It is the lowest level of the pattern which is responsible for maintaining data. In enterprise software, a model often serves as a software approximation of a real-world process.

View: It is responsible for displaying all or portion of the data to the users. If the model data changes, the view must update its presentation as needed. This can be achieved by using a push model, in which the view registers itself with the model for change notifications, or a pull model, in which the view is responsible for calling the model when it needs to retrieve the most current data.

Controller: It is a software code that controls the interaction between the model and the view. The controller translates the user's interactions with the view into actions (ActionListener) that the model will perform. In a stand-alone application, user interactions could be button clicks or mouse over events. A controller may also change the view as and when the action wants.

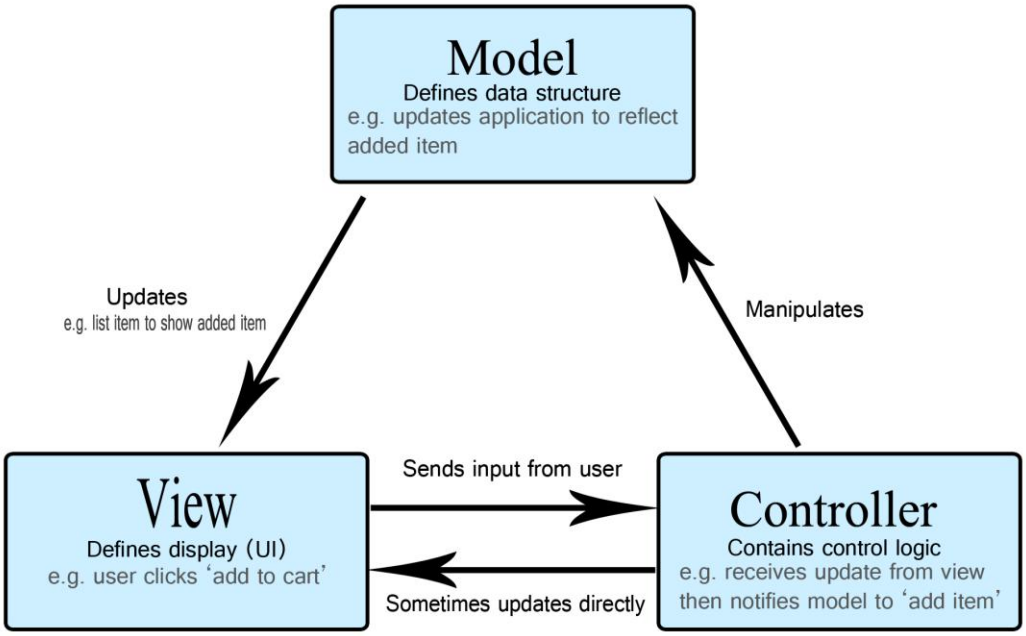


Fig. 1. Basic MVC architecture

As shown in Fig. 2, this is how MVC has been implemented in the project.

- **Models** *(GameMapModel, PlayerModel, ContinentModel, CountryModel, 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** *(CreateCountryView, StartUpView, etc.)* on the other hand renders the model into a form suitable for visualization or interaction, in a form of UI (user interface). If the model data changes, the view must update its presentation as needed. In our case, it is implemented using Java Swing.
- **Controllers** *(NewGameController, WelcomeScreenController, etc.)* are designed to handle user input and initiate a response based on the event by making calls on appropriate model objects. Thus, accept various input from the user and instruct the model to perform operations.

- 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 invoking new views upon conditions.

Following the layout of Fig. 2, the Fig. 3, 4 and 5 describes the actual class diagrams inside each and every packages namely Model Package in Fig. 3, View Package in Fig. 4 and Controller Package in Fig. 5.

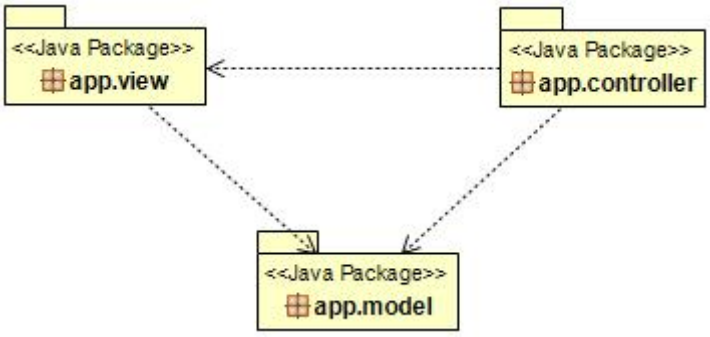


Fig. 2. Class Diagram for overall MVC structure

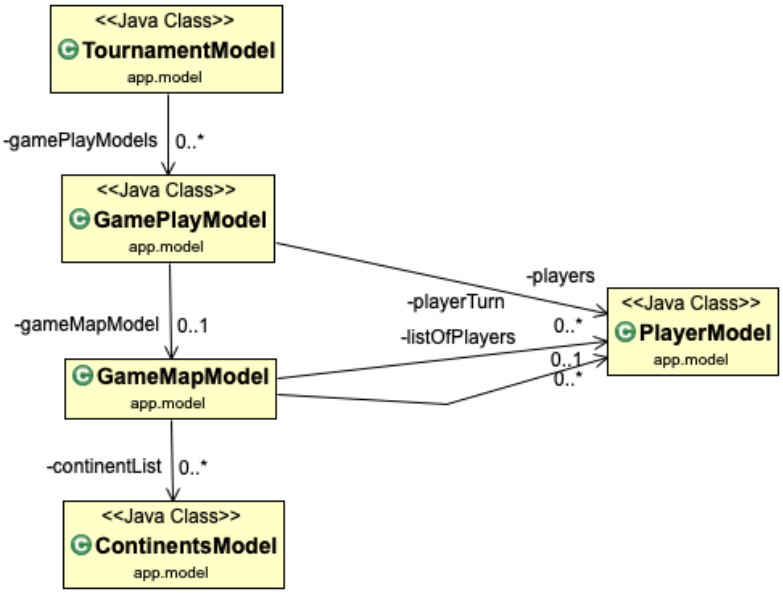


Fig. 3. Model Class Diagram

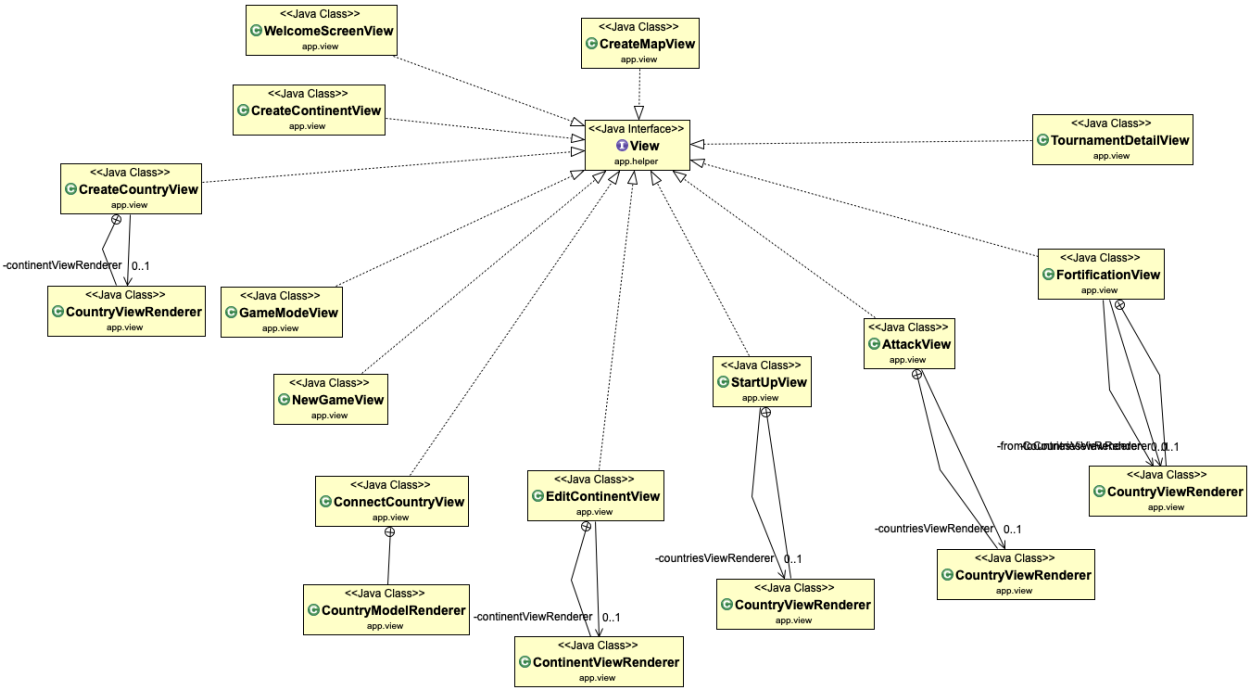


Fig. 4. View Class Diagram

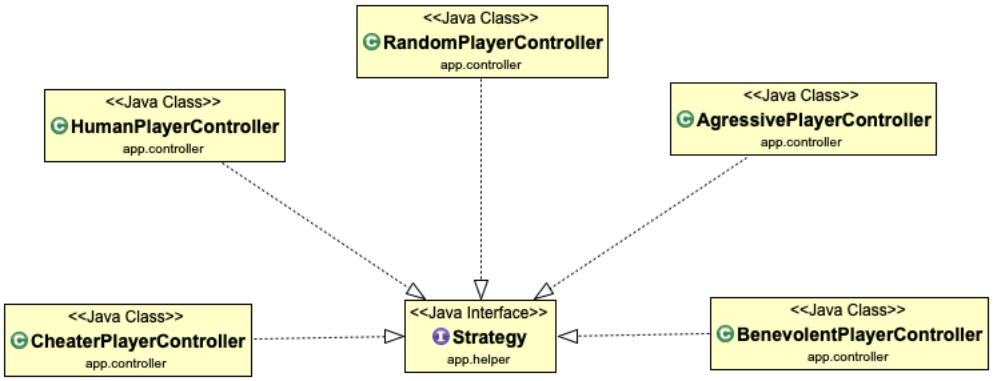


Fig. 5. Controller Class Diagram

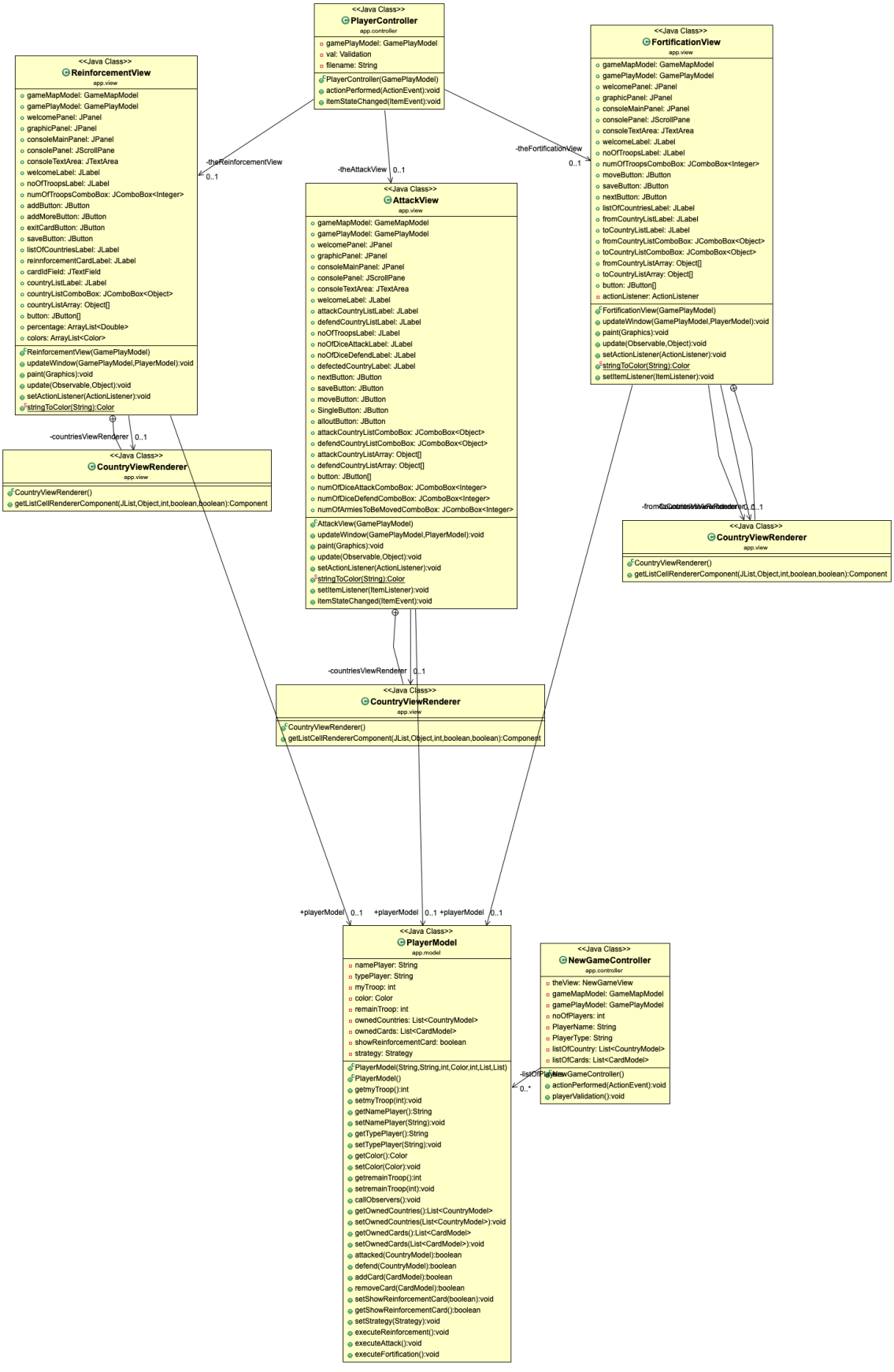


Fig. 6. Player architecture class diagram

Fig. 6 shows one particular class diagram (of many such possible class linkage possible) where in it is shown how MVC structure is implemented in the actual sense. With each and every package of model controller and view linking with each other in event-driven system.

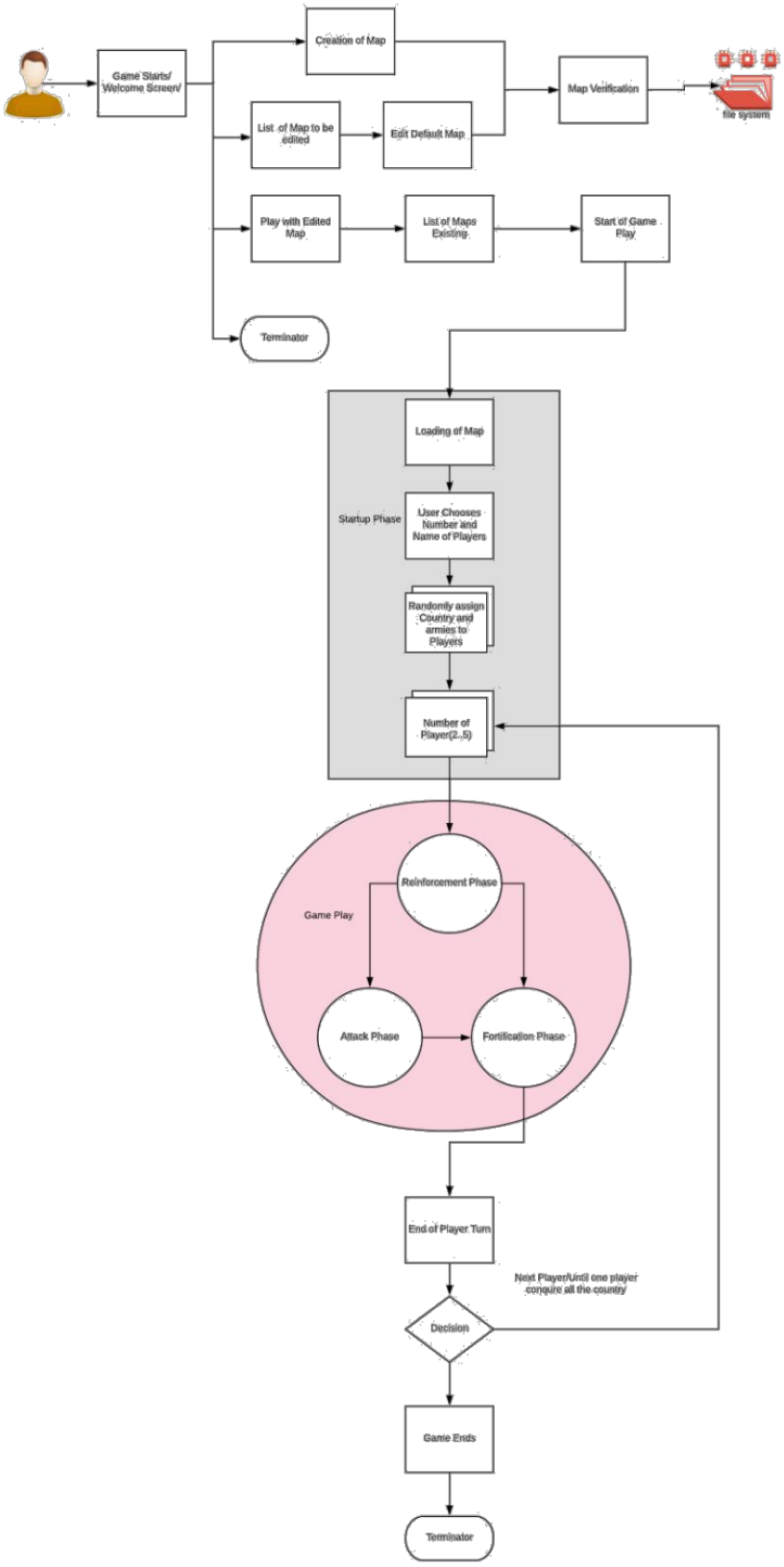


Fig. 7. Block Diagram

References

- http://www.oracle.com/technetwork/articles/javase/index-142890.html
- https://mdn.mozillademos.org/files/16042/model-view-controller-light-blue.png