Project Iteration 1

In order to support the host being able to select from multiple rule variations of a game, changes need to be made to the Rules interface, the GameFactory interface, and the GameFactoryFactory class. The PlayController, GameController, and MatchController would remain unchanged. To keep changes minimal, we add options for each combination of games and rules to the list of supported games. Those options are then decoded to create a GameFactory and a corresponding set of rules. The GameFactory creates new instances of the Rules it holds.

Inside GameFactoryFactory, each game variant string would also have a rule appended to it, which specifies the rules for a given game. Then, in its getGameFactory method, based on the game-rule combination which has been added to GameFactoryFactory's supported variable and which the host chose, a new GameFactory with the specified Rules gets returned.

The GameFactoryFactory class might look something like this:

```
static final String PU52MPRULE1 = "PU52MP:r1";
static final String PU52MPRULE2 = "PU52MP:r2";
static final String PU52SP = "PU52SP";

String gameIds[] = {PU52MPRULE1, PU52MPRULE2, PU52SP};
List<String> supported = Arrays.asList(gameIds);

public GameFactory getGameFactory(String selector) {
    if (game.equals(PU52MPRULE1)) {
        return new P52MPGameFactory(new P52Rules1());
    } else if (game.equals(PU52MPRULE2)) {
        return new P52MPGameFactory(new P52Rules2());
    } else if (game.equals(PU52SP)) {
        return new P52SPGameFactory(new DefaultRule());
    }
    return null;
}
```

GameFactory would become an abstract class that contains a private Rules variable (rules). Its constructor would accept a Rules instance, and assign that Rules instance to the private Rules variable (rules). The createRules() method would call returnNewInstance on the private rules variable in order to return a new instance of the given rules. This method would not get overridden by any extending class in order to default to the parent's implementation.

The GameFactory class might look something like this:

```
public abstract class GameFactory extends PlayerFactory, ViewFactory {
    private Rules rules;
    public GameFactory(Rules rules) {
        this.rules = rules;
    }

    public Rules createRules() {
        return rules.createNewInstance();
    }

    public Table createTable();

    public PlayerFactory createPlayerFactory();

    public Player createPlayer(Integer position, String socketId);
```

A concrete implementation of the modified GameFactory class might look like this:

```
public class P52MPGameFactory implements GameFactory, PlayerFactory, ViewFactory {
    @Override
    public Table createTable() {
        return new TableBase(this);
    }

    @Override
    public View createView(PartyRole role, Integer num, String socketId, RemoteTableGateway gw ) {
        return new P52PlayerView(num, socketId, gw);
    }

    @Override
    public Player createPlayer( Integer position, String socketId) {
        return new P52Player(position, socketId);
    }

    @Override
    public PlayerFactory createPlayerFactory() {
        return this;
    }

}
```

To support this, the Rules interface would have a getNewInstance() method added, which in any concrete Rule class, would return a new instance of itself.

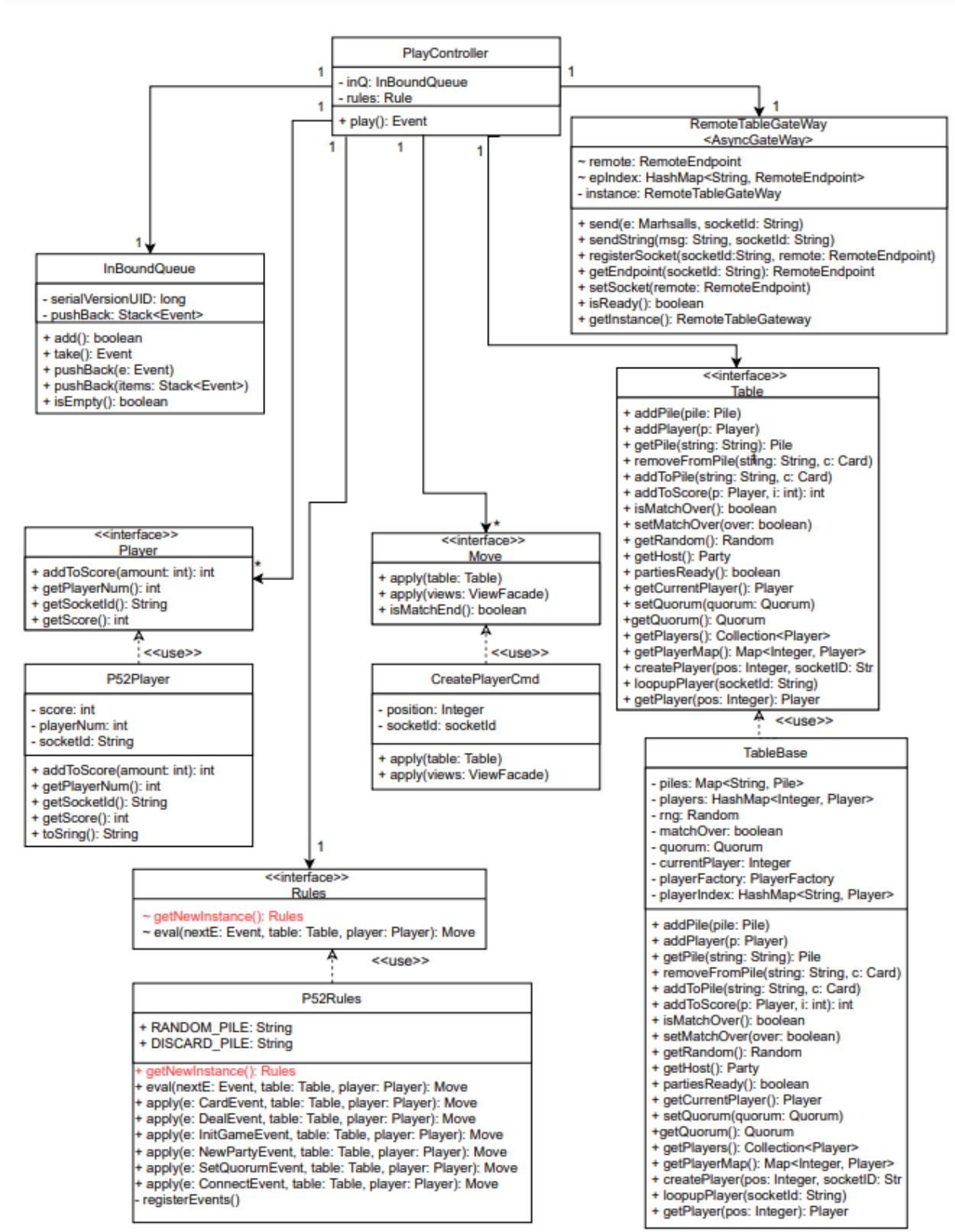
```
public interface Rules {

    Rules getNewInstance();
    Move eval(Event nextE, Table table, Player player);

}
```

An implementation of a specific Rule might look something like this:

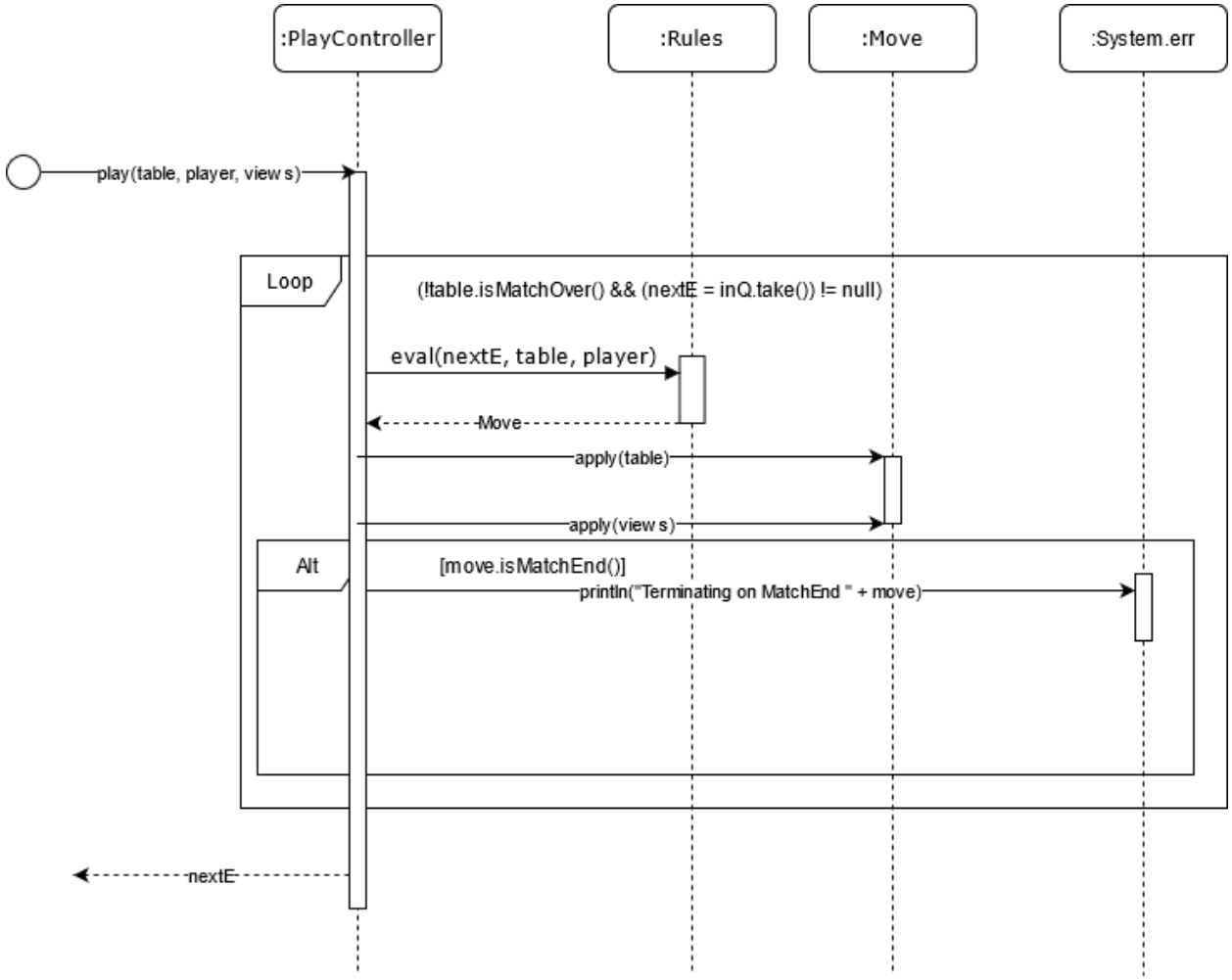
```
public class P52Rules extends RulesDispatchBase
implements Rules, RulesDispatch {

    public static final String RANDOM_PILE = "randomPile";
    public static final String DISCARD_PILE = "discardPile";

    public getNewInstance() {
        return new P52Rules();
    }
```

All changes are highlighted in red in the below Class Diagrams and Sequence Diagrams.

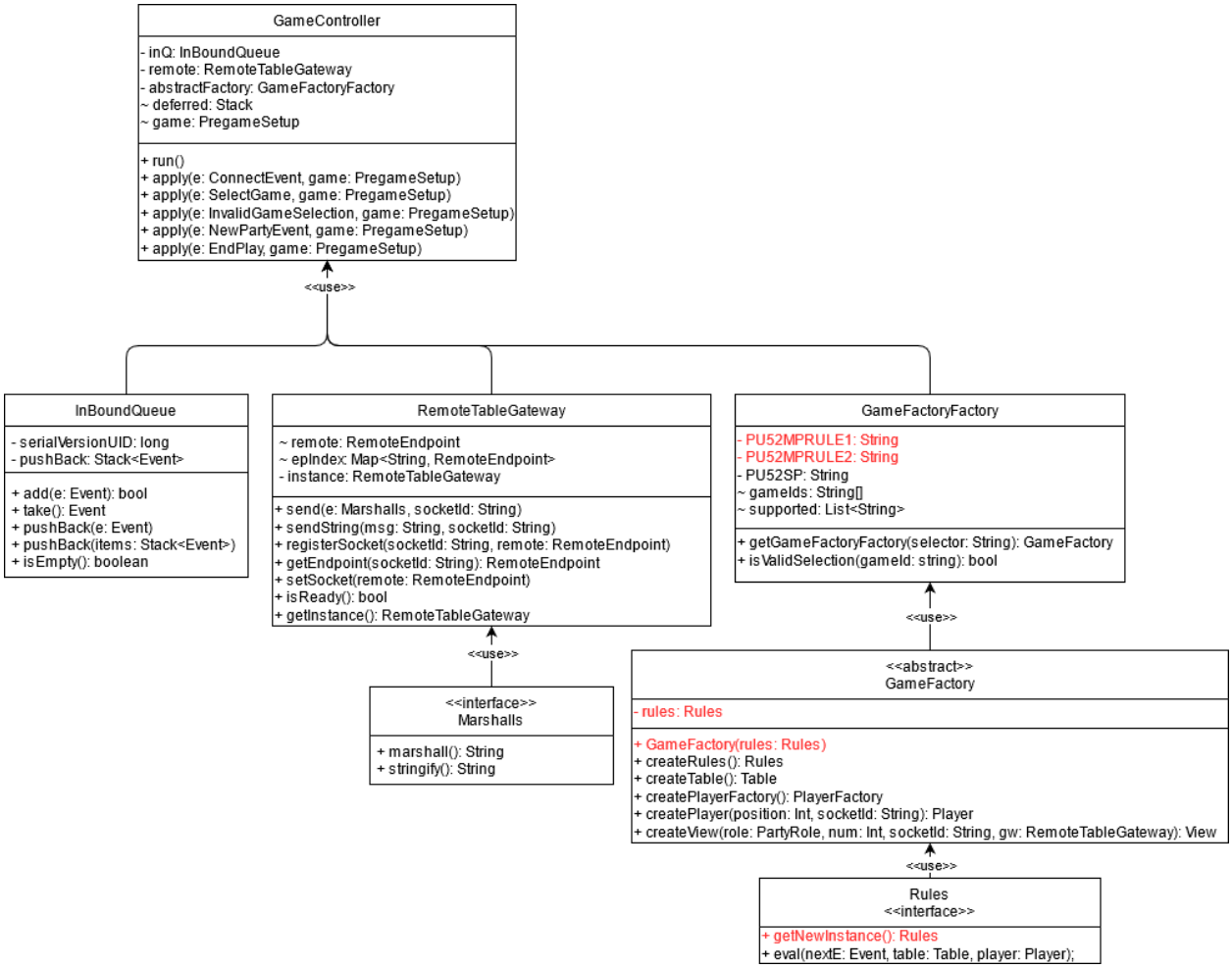
Nika: PlayController Class Diagram



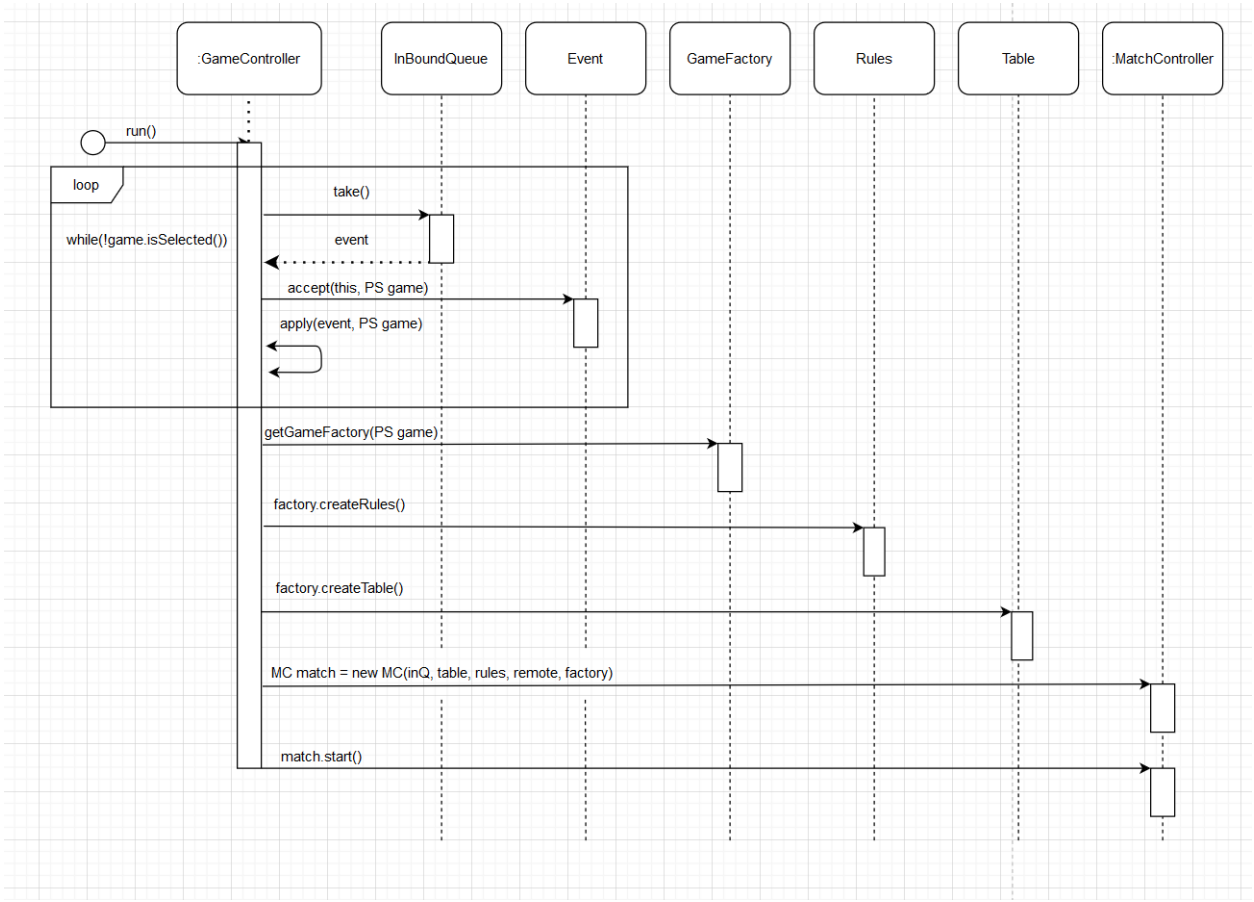
Collin: PlayController sequence Diagram



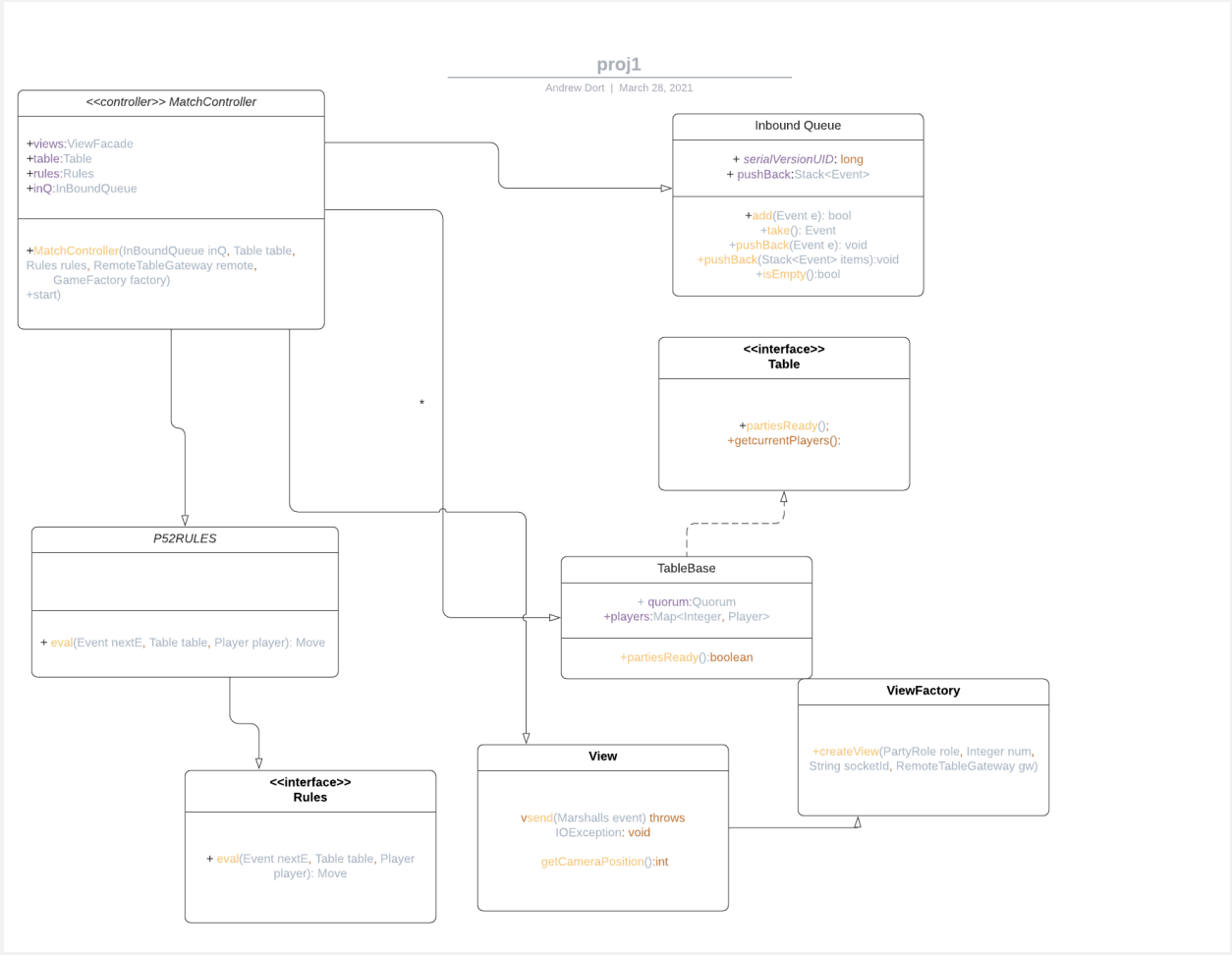
AJ: GameController Class Diagram



Keyes : GameController Sequence Diagram



Andrew: MatchController Class Diagram



JL: MatchController sequence diagram