Mafia

Andrew Chang | Dhonovan Hauserman | Callum Moore | Shawn Sowersby

April 29, 2016

Abstract

We have designed a system that comprehensively coordinates and assists users in playing the game of Mafia. In Mafia, all players have different roles, so each player's available actions are specified to their assigned role. A player is assigned their role at the beginning of the game by a moderator. Players also need to communicate with one another throughout the game, and their decisions during each phase of the game determine the winning team. This communication is sensitive; the moderator oversees events that are only available to a subset of players. Our project eliminates the need for a human moderator by replacing this role with an automated service. In essence, this app controls both the interaction between players and their actions as well as the task of the moderator in terms of guiding players through the cycles of the game.

Rules of the Game

Setup

The game is played with seven players. They are divided into two teams: the Town, and the Mafia. Every player is assigned a role. The Town consists of a Cop, a Medic, and three Vanilla, and the Mafia consists of two Mafioso. Mafia members know who their partner is, but members of Town only know their own role. This lack of knowledge is what drives the game forward. As players question each other in order to discern the identity of the Mafia, anyone could be lying about what side they are truly on. This means players have to think critically about each player's actions and statements, and look for clues that would give away a lie. Each Night has a 12 second timer, Day has a 12 second timer, and the formal has a 8 second timer.

Gameplay

The game is divided into two phases: Day and Night. Every Night, the roles on both teams pick a target in the night. The Mafioso attempt to *kill* that person, the Cop attempts to *check* that person, the Medic attempts to *save* that person, and the Vanillas do nothing. The Mafia team can only kill one person during the night, so the Mafioso must agree on who to kill. If the Mafia pick a target who is also picked by the Medic, that player does not die. Selecting nobody in the night is also a valid strategy for the Mafioso and Medic. The Cop's check will tell him what team his target is on: the Town or Mafia. Then, during the next day, the Town discusses who they suspect was behind the murder. At any point during the day, players may accuse each other of being Mafia. Eventually, Town votes to *lynch* whoever they believe is the most likely to be Mafia based on evidence and arguments of all players. This is done by putting them under *formal* and counting the votes on them at the end of the timer. After a formal it goes to Night.

Game Over

Players who are killed are out for the rest of the game. They are not allowed to vote or discuss the game. After death, a player's role is displayed to all other players in the game: this is known as a *role-flip*. When all of the mafia are eliminated, Town wins. However, if the number of Town and Mafia players were match, Mafia wins. This is because Mafia can never be lynched due to the fact that Town do not hold the majority of players in the game, so Mafia will just always vote no on killing each other. It does not matter if you are alive or dead when the game is over: the results of your team is your result.

Goals

Application Goals

<u>Create a mobile platform for Mafia</u>

Above all, we are looking to create a mobile platform where users can play the game of Mafia. Often times, Mafia requires a lot of setup time and negotiation between parties about the rules. Our app provides a simple rule set and virtually no setup time to play Mafia on a mobile device.

<u>Automate the human moderator</u>

The moderator is often seen as an "unfun" role in the game of Mafia. Our application aims to automate the moderator role to make the game fun for all participants, as well as require one less human player to create a full game. This idea is fundamental to the original motivation for this project.

<u>Remove bias from the game</u>

Another concern is the bias that a human moderator may have. A moderator may be more or less likely to assign certain players to certain roles. To combat this, our app assigns roles randomly. This ensures everyone is given equal chance at every role, every game.

High Level Goals

<u>Testability</u>

One of our first high level goals was to code for testability. To ensure that we implemented a project that was intended for high testability, we worked to decouple our code so that we could facilitate unit testing.

Readability

In conjunction with testability, we worked to write readable code. In essence, we wanted to incorporate design and architecture patterns wherever they could apply. Utilizing these resources would ensure our code followed a structured pattern to facilitate reader understanding. Often times, we can code until functionality is achieved and just move on, leading to chaotic code. While implementing this project we focused on consistent design and architecture to produce a well formatted application.

Usability

Ultimately, testable and well-designed code is ineffectual without a usable application. As detailed in the application goals subsection above, we wanted to provide a mobile app that made playing Mafia easier. Use of our application needed to be simple and provide useful feedback to the user while providing a friendly, easy to use interface.

Extendability

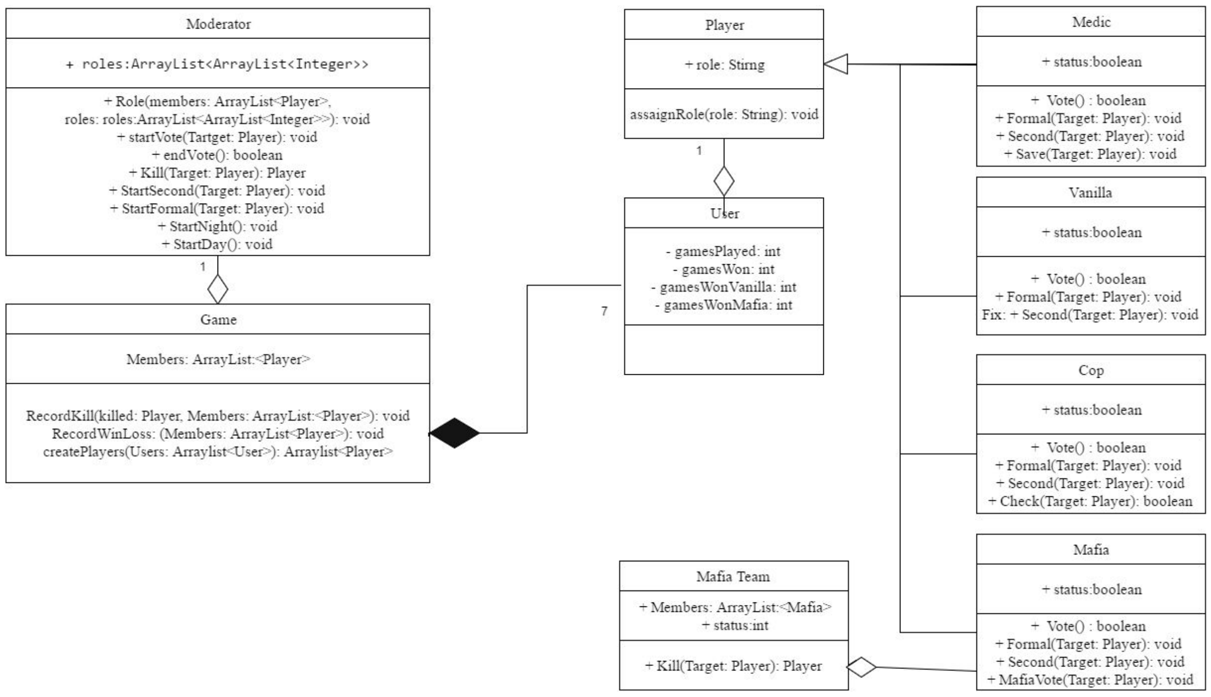
For future goals, we wanted to write code that would require a low level of modifications in order to extend additional functionality. This way we can expand our app's capabilities and facilitate systematic reuse.

Methodology

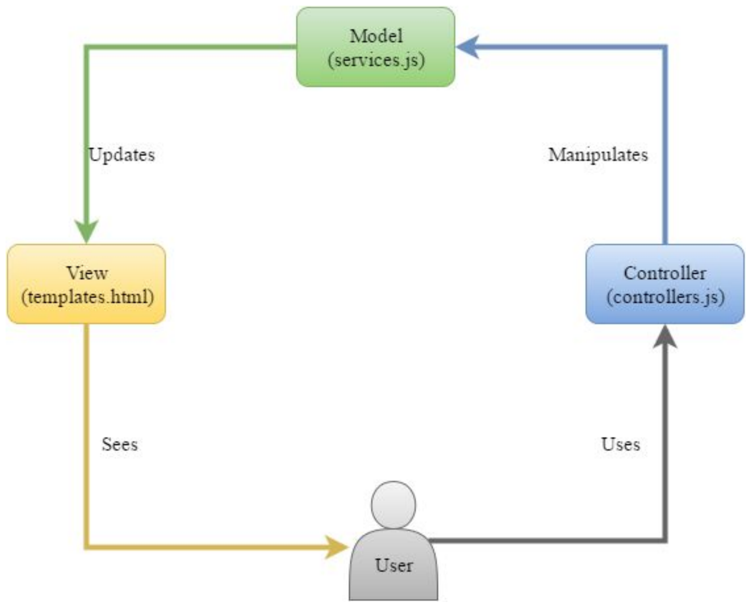
In our original proposal, we had decided to create a dynamic user interface with console messages that would act on behalf of the game's moderator. The messages that would be displayed to players throughout the game would eliminate the need for a human moderator. We stuck to this original plan. We also decided that the best way to eliminate bias from the game was to randomize the role assignment process as well as all of the automated players' selections for their Night actions and their votes that are cast during the Day.

We had also originally planned on developing this app using several different objects that would interact with each other throughout the game. We had planned on using the decorator design pattern to make better use of classes that were very similar to one another. This lead us towards a Java development environment that we could apply to the Android market. However, our interest grew away from this market and towards web development and cross platform development. In order to use the Mafia app across all devices, we decided to use the Ionic framework. Details on the installation and exectution of our Ionic app are laid out in the README.md file included.

Ionic is capable of launching our app on Android, iOS, and web services. However, this now meant that we had to abandon our Java project and work towards a full web development stack consisting of cascading style sheets generated by Ionic, Javascript files for functionality, and html views. This allowed us to utilize the MVC architectural pattern.


Moderator
+ roles:ArrayList<ArrayList<Integer>>
+ Role(members: ArrayList<Player>, roles: roles:ArrayList<ArrayList<Integer>>): void
+ startVote(Tartget: Player): void
+ endVote(): boolean
+ Kill(Target: Player): Player
+ StartSecond(Target: Player): void
+ StartFormal(Target: Player): void
+ StartNight(): void
+ StartDay(): void
1
Game
Members: ArrayList:<Player>
RecordKill(killed: Player, Members: ArrayList:<Player>): void
RecordWinLoss: (Members: ArrayList<Player>): void
createPlayers(Users: Arraylist<User>): Arraylist<Player>
Player
+ role: Stirng
assaignRole(role: String): void
1
User
- gamesPlayed: int
- gamesWon: int
- gamesWonVanilla: int
- gamesWonMafia: int
7
Medic
+ status:boolean
+ Vote() : boolean
+ Formal(Target: Player): void
+ Second(Target: Player): void
+ Save(Target: Player): void
Vanilla
+ status:boolean
+ Vote() : boolean
+ Formal(Target: Player): void
Fix: + Second(Target: Player): void
Cop
+ status:boolean
+ Vote() : boolean
+ Formal(Target: Player): void
+ Second(Target: Player): void
+ Check(Target: Player): boolean
Mafia Team
+ Members: ArrayList:<Mafia>
+ status:int
+ Kill(Target: Player): Player
Mafia
+ status:boolean
+ Vote() : boolean
+ Formal(Target: Player): void
+ Second(Target: Player): void
+ MafiaVote(Target: Player): void

Original class diagram (rejected after transition to Ionic)



MVC diagram provides better insight of our project's structure

The Controllers are found within www/js/controllers.js. In this file, there is a Controller for every View. The Controllers call functions that are provided within the Model and manipulate the results of those functions. For instance, the Night controller calls the isGameOver() function from the Model and redirects the player to the Results page if the game is over at the start of that Night. It also generates pop-up boxes that display messages depending on the target whom the player picks in the Night. This uses functions from the Model to validate roles. It also keeps a countdown so that players know when Night will end. The pop-up, timeout, and redirection functionality are all Ionic provided services. Other Controllers have similar functionality.

The Model is found within www/js/services.js. All objects, arrays, and functions that are called from the Controller or modified in the View exist within this file. Rather than create a class for each role and team as was originally proposed, we simply held one array of seven user objects. The objects are all identical in their default form. Each user object contains fields for their username, assigned role, the action their role allows, alive status, button display (what appears on their button may vary depending on their alive status), most recent vote, whether they were saved by the Medic in the previous Night, who their target in a Night was, a message specific to their role if need be, and a function for their Night action. The last field is the most interesting and it warranted the use of the strategy design pattern, which is explained further in the Design Challenges subsection within this paper.

The Views are found within the www/templates directory. Each html file here represents one View of the app. Some files, such as gameDay.html send user input to www/js/controllers.js where it is transferred to www/js/services.js where fields are modified. Early mockups of the Views are available within the Images directory of this project, as well as an early page flow diagram.

We implemented this app to be easily extendable for future modifications. We made sure to abide by the Open/Closed principle such that our code is available for extension with only minor additions necessary, but no modifications necessary. This will allow us to easily implement some of the future goals that will be mentioned later on such as opening the app up for multiplayer use, or dynamically adding more roles or players to the game from within the user interface.

Testing

As one of our goals was to write code for testability, we wanted to be able to use unit testing in order to check the functionality we wrote. Our unit tests in this case would all be considered white box testing since we wrote the tests with knowledge of how the program works. We used Karma to write unit tests for Javascript. Karma works by simulating the web app through a browser window. The browser remains empty while the console displays which tests failed and which passed. In that sense, Karma operates very similarly to JUnit tests.

The main difference is that with Karma, all of the web app besides the section we are testing is ignored. We only used unit testing on the Model portion of our program since this contained the most testable functionality. The Views and their respective Controllers were tested by user interactions since the Views had to be visually checked, and the Controllers determined routing paths and countdowns. This was all functionality that was easily testable by simply playing through the game.

The unit tests were written in tests/unit-tests/servics/serviceTests.js. The unit tests were all a form of white box testing since we wrote the tests with knowledge of how the app was implemented. For user tests, we allowed friends outside of our development group to play the game after reading the rules. This ensured no bias in our testing set. It is also an example of black box testing since the participants did not know about the details of our implementation. We also played the game amongst ourselves and attempted to break it by playing for all edge cases. This ensured a wide coverage of our testing and is a form of white box testing since we played with the intent of breaking the game based on details of the implementation.

Challenges and Solutions

Design Challenges

It became self-evident after implementing the majority of the functionality for this app that we were violating the software engineering policy that is coding for testability. We also wanted to be able to extend our code for future additions, as stated in our Goals section. In order to resolve these issues, we worked to decouple our code and generalize attributes to allow extension and reusability. We also ran into an issue regarding the synchronization of player actions. Since there are multiple players that all need to perform their actions before moving between Night and Day, we needed to ensure that no player would be unable to perform their action unless it were intentional.

Design Solutions

In order to code for testability, we had to decouple our Model and Controller more than it currently was. Mostly, this meant moving functionality that existed within the controllers over to our services.js file. This allowed us to keep all of the app's functions within one file and ensure that no one function contains too much functionality. There are eighteen functions within this file that all get called from controllers.js, and they are all testable. One example of the refactoring that went into this involves the last field of the user object, the function nightAction.

Rather than determine which action to have any given player perform during each Night by checking their role within a large conditional block in controllers.js, we implemented the strategy pattern and refactored our code. Each user has a function that is undefined as default. When the roles of a game are randomly assigned within services.js, each player's function is also assigned accordingly. Then, in controllers.js, there is simply a for-loop that calls users[i].nightAction(). This will perform that specific player's action without needing to check their role.

In order to ensure extendability, we had to code with the Open/Closed principle in mind. Our code is generalized so that if we ever wanted to add an eighth player to the game, we could do so by adding one line of code in services.js that adds another user to our users array. All functions and controllers are written so that they are not dependent on any definite size. The same may be said for roles. We could add an entirely new role to the game by adding its name to our roles array and its respective function in our setRole function within services.js.

In order to solve our synchronization issue, we decided to use an AJAX tool called timeout. We then set the amount of seconds that that particular page would stay up before redirecting to another page. This now allows us to cycle between Night and Day, as well as the voting pages as necessary. It also ensures that every player has a set amount of time that is displayed to them indicating how long they have to perform an action. If no action is performed, it will not break the game. However, we found that if the user were redirected to another page before the timeout, that the timeout would still count down and a second, unwanted redirection would occur. To fix this, we added functionality within each controller that consisted of a countdown that would reset all data upon entering the page and cancel the timer whenever a rerouting takes place. Now, the cycle between Night, Day, and voting within the game all occurs at the same time for all players involved.

Testing Challenges

The hardest part of testing was actually getting the running suite set up for unit tests. Installing Karma and getting it to correctly run took most of the time spent on unit testing. Other problems we faced were syntax and how to write the actual tests. Karma has specific syntax that was foreign to us at the time. Once we had the syntax down though, it was just a matter of figuring out how to test each of the functions. The user tests were not as difficult, but it needed the user to be experienced with the rules and rare situations of the game to be able to test it.

Testing Solutions

In order to successfully use Karma, we had to research a wide range of troubleshooting forums in order to get through the install process. We looked at many examples on how to test Javascript using Karma so that we could efficiently test our code. For the user tests, we originally checked for errors by playing the game within our development team. In order to ensure valid testing that was absent of any bias, we allowed others to play with the app after informing them of the rules. We also played in a manner that attempted to break the game in all of its edge cases in order to test as many scenarios as possible.

Future Work

After this development cycle, our development team is looking to extend and improve our application. With continued support, we will look to better our Mafia application in a variety of areas, although the primary focus of further development will be the implementation of a multiplayer version of the app. Multiplayer will vastly increase the user base of our application. As mentioned earlier, our code is flexible and built for multiple players with individual actions. For this reason, an extension to multiplayer should prove to be relatively simple. However, some expected difficulties would be the implementation of a server-client relationship, implementation of a database, and communication between different devices running our application. Presently, our application prohibits automated users from killing the host for the sake of gameplay.

Our current version of the app already has views built for statistics and a full user authentication. However, these are more of a formality at this point. The stats page is just a static html and the user authentication just needs any string to be entered for the username while the password field is just a placeholder. With multiplayer implementation, the stats page would be populated with data extracted from the database for that user. User authentication would also require the database so that users may be added from the Create Account page or verified from the Login page. The other necessary adjustment for a move to multiplayer to be complete is the act of inviting players to a host's game lobby. The host is the user who started the game and invited players via username. Those users could then accept or deny the invite to the lobby.

Another interesting feature requiring extreme code flexibility would be the creation of new rule sets and roles. These rule sets could be created by the community to increase popularity of the app. These new rule sets could adjust the difficulty of the game based on player count and role count. New roles could also add interesting and new ways to play the game. In that sense, our Mafia app has the ability to remain constantly fresh and exciting for the consumer.

Conclusions

In the projects infancy, we learned to choose our framework and language carefully. We started this application in Java because that was the language we were most familiar with. We were ignorant to our interests and the needs of the application. As a result, we lost a lot of work switching to Javascript. We realized we would have a far more interesting application if we developed it as a full stack web app that could be supported on mobile devices. We learned that the initial investment into framework and language exploration could save time and resources.

Another lesson learned was the transcendence of software architecture and design patterns. We implemented certain design and architecture patterns when we began this project in Java, and even though we switched to developing a web app, those patterns still applied. Many design and architecture patterns are applicable to most programming languages, hence their high usability. In particular we found architecture patterns to be the key to the success of our project. Our application consists of a large number of communicating elements. To handle this degree of communication and still maintain readability, an architecture pattern such as MVC was a necessity. Without an architecture pattern we would not have met any of our high level goals.

One of the major oversights of our project plan was testing. As often is the case in software development, we overestimated our progress on this project and as a result we did not have much time for testing. While this project was smaller in scope than most real-world applications, it should still hold that testing consist of a large majority of our development timeline. However, we performed several user interaction tests after each significant change to the code along the way. In the future we will look to more accurately estimate our progress.

In reflection the application does not fulfill the original specification. However, with the lessons that we learned and the application that we produced, we are thrilled and are looking forward to further developing and marketing our application in the future.