SCORE 2018 Summary Report
Epic-Monopoly: A Novel Monopoly Game

Ziqiang Li Yulian Mao Xizi Ni Yilin Zheng Chenyu Zhou
{11510352, 11510086, 11510602, 11510506, 11510374}@mail.sustc.edu.cn
Yuqun Zhang zhangyq@sustc.edu.cn

January 16, 2018

1 Abstract

In this project, we concentrate on developing a new web-based monopoly game mechanism which called Epic-Monopoly. We developed this project by following a modified Waterfall Model which helped us develop this project efficiently. We designed this novel monopoly in an object-oriented manner which enables a better design for large programs. Some popular and advanced frameworks were used and we combined in this project that results in a modern web game architecture. Reverse proxy servers, cache servers, and database are considered in the extended architecture design. Since the original monopoly game is rigid in rules and less exciting for players after some turns. There are many new and interesting elements evolved in this game. An economy factor, similar to GDP or its relative index, is added into the game as an approach to model the reality. The index of assets value, tax rate and punishment intensity of chance cards are all influenced by this mechanism. For more user flexibility, the difficulty of the game can be adjusted at the very beginning of a game by changing the initial cash and some other indices. In the course of a game, we use a dynamic map to generate a more interesting but still balanceable experience. The player falling behind might catch up again with this change. In the reality, there is much cooperation between entrepreneurs that might influence tax and warfare, so, this game includes alliance and countries which can also be seen as a loyalty program to model this situation. Besides those important and creative new silver points, there are some other modifications, such as the rules that allow enterprisers to enter the game which is ongoing. All the details are introduced in the following sections, including the requirements of the program, concrete implementation and the UI design of the game.

2 Software Engineering Process

The software engineering process chosen to manage the creation of our project is much like Waterfall Model but not exactly the same. We called it **Campus Waterfall Model** since we use this model on campus and many student project processes will act like this model.

It is well-observed that developing a software on campus is much different from developing that(most projects on companies are much larger and more complex than those on campus) from requirements, tools, conditions, and situations. On campus, the project doesn't face the requirement of scalability and security in most time. So, the situation might differ from that in commerce. The campus project pays more attention to the design and implementation of the functions and the effect of the final result which are only parts of the goals in commercial products. The tools for students might be limited to the free or open source or something costs less since the project is not profitable but the cost is still needed to be measured. Besides, the conditions of the colleges or supervisors can provide are not as advanced as those on corporations where the staff are under pressure. However, the biggest difference is the students have no superior skills and enough engineering experience about the software development even they have learned some related courses. So, we modified this model to adapt our real situation.

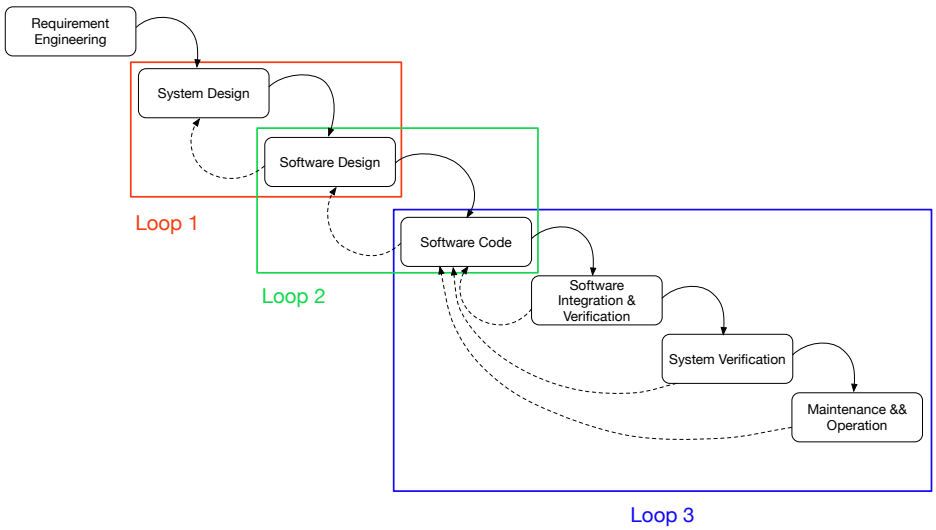

Figure 1: Campus Waterfall Model

In this model, we add four back curved arrows (dashed line) back from certain stages to the previous stages and three squares which indicate development loops. The dashed line means the feedback to the previous steps and returning to the previous stages for improvements or bug fixing and go along the steps again which

finally will be looped. The later loops will be faster than before or even skip some steps since we are more familiar with the operations and less modification is needed or even no changes take place. For small projects, this loop will be feasible and useful. At least, in our project, this model helps us go quickly in the development and overcome the disadvantages of the original Waterfall Model.

1. *Requirement Engineering*

In this stage, requirements were collected and analyzed. In this project, the requirements came from sponsors of this project and we further considered more in an engineering way with the situations such as the increment of users or servers, the cost of the project and the expandability of the product.

2. *System Design*

In this stage, we divided the whole development into two parts, the front end system and the back end system according to the requirements. Between the front end and the back end, the important part is communication.

- For the front end, we choose game engine Phaser combining JS/HTML/CSS to implement all the pages and animation in the game.
- For the back end, we use a normal game framework, which includes a Nginx reverse proxy server to handle massive requests from the front end, Tornado servers as application servers to run game instantly. We also plan to use Redis as the cache servers and MongoDB as the database if necessary. We leave the cache and database part as an extended development plan since we don't need them temporarily(more details in later sections).
- For communication part, we use JSON to store data(so it can be used in MongoDB or cached in Redis) and transfer message data between the front and the back end.

3. *Software Design*

Based on the previous stage, this stage is to further design the software in the framework. State diagram and UML were designed in this stages and all the game logic was split into various code modules to guide the code implementation in next stage. Classes were also designed under the principles of object-oriented programming since we were going to implement this project in an OOP manner. In this stage, we can also go back to previous stages to adjust the system design to make it more feasible to implement since the software or tools we choose might not fully support what we need.

4. *Software Code*

In this stage, all the codes are implemented under the guide of the UML designed in the previous stage and state diagram was used to validate the running result of codes. Also, there is a loop here to indicate the coding process will also influence the UML design since some designs are not easy to implement so we will modify the UML to lower the difficulty of coding which also lowers the existence of bugs.

5. *Software Integration & Verification*

In previous stages, codes were developed in independent units and can be tested for its functionality by using unit test. In this stage, unit codes were integrated and tested as a whole by using integration test to see if all modules cooperate as expected. Feedback still needs to be returned to previous stages for code improvement.

6. *System Validation*

In this stage, the three parts were to be assembled into a complete entity then we conducted the system test for this software on the hardware we designed before and fed back the bugs or problems to coding stage.

7. *Operation & Maintenance*

This stage means we almost finish this project but we need to do something for further maintenance of our project. We conducted an online test by inviting some friends to be the initial users and give us feedback. According to the feedback, we might need to go back to improve our codes.

The software engineering process is much important for developing a real software involved with engineering technique from design to release so that it can work as expected.

3 Requirements

This project is a web-based game design project, so it involves web development techniques and the requirements of this production include all the fundamental features and some new features the sponsor wants:

1. allow players to enable a random sequence of spots as well as a periodic, random change of the properties pricing, rents, interests and taxes rates (in both spots and cards);

2. allow a loyalty program to enable the use of virtual currency with random rules based on existing international alliances programs;

3. allow new players to join the game as entrepreneurs

4. allow complexity setting in three levels (e.g., easy, medium, hard) based on rules and different configurations from exploring variabilities on the previous three bullets.

3.1 Design & Tradeoff

To meet the basic requirement and achieve better user experience, we add some new features in this game. However, during the implementing process, we encounter many difficulties. We solved some of the difficulties and we did make some tradeoff in this project. The new design and tradeoff are shown as follow:

1. *New Features*

a) Economic Factor

In the requirement, property value, rent pricing, tax, etc. should be set as dynamic values in the new monopoly game. After our discussion, we thought that randomly change all the values individually may influence the balancing of the game. Therefore, we analogized from GDP or some relative economic index, we introduced Economy Factor(EF) to the Epic-Monopoly. EF act as a global rule which in charge of the changes of the prices of properties, tax and value of other objects.

For a specific period, EF will change between a certain range, and it will have an influence on the rent, pricing of the house, etc. Because of its characteristics, the average value of all property is same. It is seldom for a property devalue every period, which is unfortunate for the owner of this property.

This new mechanism enables the game closer to our real life.

b) Dynamic Map

The sequence of the spot in the Epic-Monopoly will change in a specific period, to re-pick the fate of players who fall behind. To unify the map, the change of color blocks, stations and utilities will be controlled by some rules. Therefore, some wired situation, such as successive chance block, will not happen in Epic-Monopoly.

c) Country & Alliance

There is no benefit for the corporation in the old version monopoly. Therefore, countries and alliances are introduced to the Epic-Monopoly. Every player will assign to one country and corresponding alliances. The trade and rent between the member of alliances will make a discount. And some new events will add to opportunity, to add more fun. In reality, tax and warfare differ depends on country's policy. Therefore, the tax and some of the cards in chance and chest will depend on which country the player belongs.

d) Difficulty Setting

In the old version of Monopoly, players can custom some game configuration such as limitation of houses, etc. However, in the Epic-Monopoly, the old custom configuration is kept, and we introduce the difficulty settings. Again, to balance the game, the difficulty depends on the fluctuation of the game parameter EF. For example, easy model the EF setting will within 5% while normal model will be 10%.

e) New Player

Enterprisers can join the game after the game began. However, the enterprisers are only able to join when unacquired real estate more than 50%. And the initial cash of the enterprisers depends on how many turns the game has begun.

f) Loyalty Program

Instead of return cash or virtual currency, Epic-Monopoly will make a discount for enterpriser. (The effect is same but simpler) By the time enterpriser enter other's spot, the rent of the spot to this enterpriser will decrease. Same rules for prison. By the time an enterpriser bail from the prison, the price will increase, which is the same as reality (The more into prison, the more bail fee it must pay).

2. *User Interface Design*

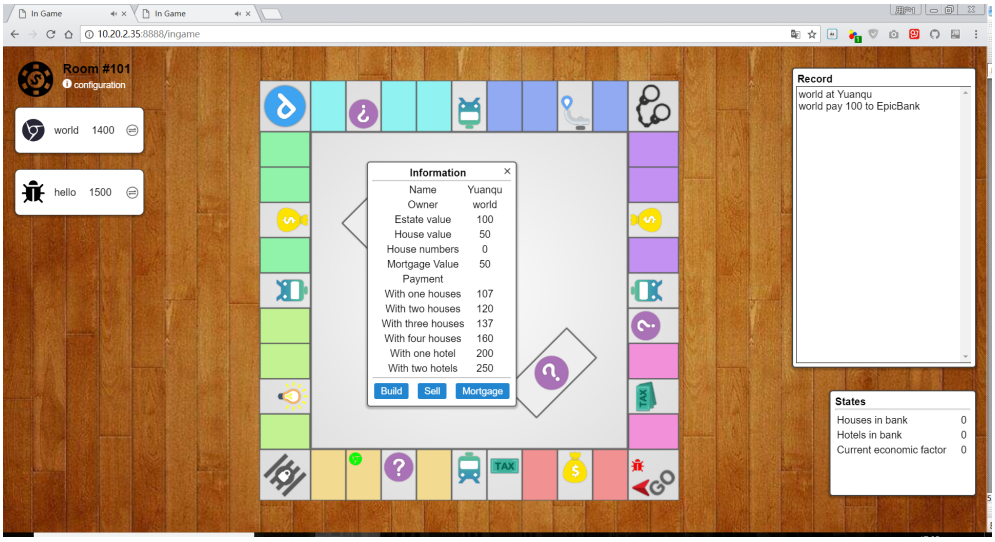

Figure 2: Management page

a) UI Design

To pay our respects to the old version monopoly which is created in 1903, the key design principle of Epic-Monopoly UI is classic. The design of card and chessboard are old fashioned. We used a wooden background which represented the wooden table people used to play in.

More about UI is attached in Appendix II-UI.

b) Animation

Epic-Monopoly, as a web-based game, users want to open it within seconds. We have to trade off the expectation about animation and 3D chessboard. In the current design, we used 2D chessboard and simply movement animation. For the final implementation, we may switch to 2.5D, which known as fake 3D.

3. *Game Logic Design*

a) Object Oriented

Epic-Monopoly has many objects in the game which can be extracted to classes. Thus we designed the game logic in an object-oriented manner. We designed with as less as possible classes by using inheritance mechanism and polymorphism. The classes have no overlapped in its function, greatly simplified our codes, and also made the codes much easier to maintain. The relations of the classes the code are constructed based on the state diagram.

The state diagram is attached in Appendix I-State Diagram. The UML is followed in Appendix I-UML.

4. *Server Design*

We divided server into three parts, request, gaming room and communication. The request part takes in charge of analyzing the request from a user and acts the

corresponding operation, like building a room or enter a room. If all users agree to start the game, the server will build a process for the gaming room to start the game logic. For the final part, we defined the JSON format to transmit data from game logic to user interface. We use WebSocket to implement the communication part. For the further implementation, we will use MongoDB to store user information and data of every game and Redis to cache data if necessary. Those data will be very helpful for change the gaming parameter to have better game balancing. For consideration of scalability and concurrence, we plan to use Nginx as the reverse proxy server to distribute requests from the front end to distributed servers which provide the software scalability and provide a better concurrence.

4 Implementation

4.1 Team assignment

There are five developers in our team: two for back-end development, two for front-end development and the other one applied the communication of back end and front end. In the design stage, all five developers designed the game logic together. Then one of the back-end developers and communication developer drew the UML and state diagram. Later two back-end developers confirmed the JSON format for data transmission together. In implementation stage, simultaneously, one front-end developer designed the UI and the other front end developer implemented the animation in the game and web pages while two back-end developers implemented the game logic.

However, as the project proceeded, it was difficult to finish the task independently. There is some time we worked together on same part of work, especially the front end part involved with much work that even dragged our progress.

4.2 Platform choices

4.2.1 Back end

The back end is implemented by Python 3.6.4 with Tornado framework.

1. *Object Oriented*

Epic-Monopoly is designed by object-oriented ideas. At the beginning, it took us over one week to design the UML which further helped us a lot during the implementation. The main game logic is divided into several parts and every part is defined by some superclasses. For example, the ”Block” class is the superclass which represents all the block on the chess board. The ”Block” can be further divided to ”Asset” which represents the properties in the game, ”CardPile” which represents the ”Chance” and ”Chest” block on the chess board. And those sub-classes can be divided into even smaller class, ”Asset” can be divided into ”Estate”, ”Utility” and ”Station”, which are the three main properties that players own in during the game. Every sub-class have their own game logic and attributes but share many same

attributes like ”name”, ”id” and ”position”. The benefits of using object-oriented design, on the one hand, can reduce the lines of code, on the other hand, let us focus more on the difference between classes. Also, it is very convenient to debug and further maintain as well. At the end of this project, we found an extremely wired bug. After analysis and tests on it, we realized it happened in the common codes among ”Estate”, ”Utility” and ”Station”, thus we are able to find the mistake and repair it.

Moreover, there are many similar methods used in the program. For example, payment is used by many objects to process a trading case. Therefore, we extract this kind of methods to a file named ”operation”, and almost all classes will import this module since they need the common method. It will also prevent the re-import problem. We do the same procedure when we designed the ”messenger” class again. It embedded all the methods of communications between clients and servers.

2. *Speed*

Epic-Monopoly is a web-based game. Users are not willing to wait for several seconds or even minutes for the initialization of a game. While we are loading the files from the server, the user can use those time to create a room or join a room. The loading time is much tougher to observe. Also, we use CDN to deliver the data quickly, which speeds up the initialization and reduces the pressure on our server.

3. *Security*

The game runs only on the server, and both the front end and the back end will check the data input by users and the results generated by the game, ensuring the correctness of the game and no possibility for players to cheat.

4. *Balancing*

There are many features introduced in the new monopoly, to balance the game and maximize the happiness of players. Epic-Monopoly was tested to balance the new features and original game settings.

4.2.2 Front end

Front end includes **page design** and **animation design**.

For **page design**:

There are two pages in Epic-Monopoly. The first is login page and the second is the game page.

In the login page, we use a <canvas> element as a container, a <form> to collect players' personal configuration (nickname and avatar) including an <input type=”text”> element and a <select> element.

Then if a player chooses to create a room, a hidden element will be displayed, which contains another <form>. That <form> contains an <input type=”text”> element, several groups of <input type=”radio”> elements and <input type=”checkbox”> element, to set up all configurations of the new room. After a player confirm his settings, the function *createRoom* will be called. It then calls *getPlayerJson* and *getRoomJson* to obtain all settings of the player in JSON form and sends an HTTP POST to transfer the aggregated JSON to the server.

Then it reads the JSON returned by the server and stores them in sessionStorage, and finally let the window jump to the game page.

And if a player chooses to join a room, the function *getRoomList* will be called. It sends an HTTP GET request to the server to get all information of existing rooms at that time, then reads the JSON returned by the server and adds data into a <table> element in another hidden element, and finally displays the hidden . Then a player can join any room that is still not filled by 6 players. After that function *joinRoom* will be called. It then calls *getPlayerJson* to obtain all settings of the player in JSON form and also the room whom player chose to join and sends an HTTP POST to transfer the aggregated JSON to the server, then read the JSON returned by the server and stores them in sessionStorage, and finally lets the window jump to the game page.

The game page is divided into three parts: left part, mid part, and right part. The left part contains two elements: one to display room information, another to display all players and their basic information in the room. Each player is displayed in a elements containing an <img> (to display avatar), two (to display player's name and cash) and an <a> (as a trade button). The player elements are all hidden and will be displayed when a new player joins the room.

The mid part contains a <button> to start gaming and a hidden . When a player presses the start button and the room is permitted to start a game, the button will be hidden and the hidden containing the chessboard will be displayed.

The right part contains two , record and states. The record contains an inner to display all record of the game and the states contain a <table> to display some dynamic data during the game.

For **animation design**:

1. sessionStorage: store room id, users' id, users' avatar. And these data will be cleaned when the browser closed.

2. Chessboard part:

The chessboard is a canvas, there are two main parts in chessboard to implement the animation design: *preload* and *create*. We put *update* part into the *create* part since we need JSON data when initializing the chessboard.

We only use **Phaser** game framework in our project.

- The first part is preload. In this part, we load our elements including images and spritesheets. We also use another **WebSocket** object here to build communication with back end.

- The second part is a function: *WebSocketTest*
  We implement the actions according to the JSON files sent by the back end

  - Init:

    * Create the part of chessboard which will not change during the game. Such as the four corners, chest blocks, and chance blocks.

        * Create the rest part of chessboard according to the JSON file, such as estate, station, and utility.
        * Create the players on the "Go" block.
    - Update:
        * Update the block information
        * Update the players' information including their position, cash
        * Update estate, station, utility information
        * Bank information
        * Economic Factor
    - Hint:
        * Show message on the screen last two seconds.

- Some isolated functions:
    - create_house: creates houses or hotels according to the house number of the block information
    - listener & show_block_infoContect: shows the corresponding information window(estate information window,) when click the block sprite
    - roll_dice: control dice rolling action
    - mortgage: send a mortgage request when click the mortgage button

4.2.3 Communication

Our server is based on **Tornado** which is a Python web framework and asynchronous networking library. Its two features are what our game program needs. Tornado deals with all requests from the client in our present architecture. The server will monitor all room members (clients) if their links are alive. If not, the server will destroy the room, close the socket, empty the memory for the rest of other games.

The program doesn't have an account management system, and we consider that a player who creates or joins a room is ready for starting games. We also allow that players join a game which is ongoing but follow the rules on initial cash for balancing the game.

If a room owner creates a room, the server will not create a game immediately. Until the room has enough players and one of them clicks start games the server will create a new game instance in another threading. Also, the server threading communicates with the game threading through a pipe, which notifies the game threading the clients activities. The game message uses a queue to preventing from blocking on received or sent messages.

In a further development, if take the scale of the multiplayer game, single Tornado server cannot bear. So, we can add the Nginx server as the reverse proxy server to balance the server pressure with distributed Tornado servers.

4.3 Design changes

We recently changed our design of the front end part.

1. Add a loading page

Before loading all the items from the server, we add a loading page to show the rate of progress.

2. Add creative room and join room layout on the login page

We do not design the create room and join room layout in our first design version.

3. Change the left part of player list on the game page

The ”in game” page does not show anything before all the players click on the ”start game” button. However, it may not accord with the formal condition because the players do not know how many players are ready in the room and who are the other players. So we change the trigger function of player list, the player list may first show the player his/her own information and add the other players' information when then the other players clicking the start game button.

4. UI style

We change the color schemes to make the chessboard more striking while implementing our UI design.

5 Verification & Validation activities & outcomes

At the beginning, we thought about what this game flow should be like, and determined what programming language and framework should we use. Then, we drew a state diagram to describe it. At the same time, we also drew a class diagram which shows the skeletons of Epic-Monopoly classes containing the attributes, the functions and the relation of the classes.

The UML can help other members of our team understand the structure of the program as soon as possible, and if we modify it in the future, UML can explain it well for us. However, the most important is UML explain clearly the game's complex logic, multi-level state transition. Whenever we implemented the code, we would not get lost in the game logic.

We verified our game's validation and usability in modules. Each module has several test cases. After we finished all game logic modules, we assembled it into an integral game logic and verified whether each module worked together smoothly.

Firstly, we tested the game logic on our local server. Then we added commutation module and put it on to the web server. We tested it on a prototype website without game UI, which showed all web socket worked well. Finally, when the UI part was gradually finished, we integrated both the front end and the back end to ensure that both ends cooperate as expected.

6 Implementation operation

When implementing our design, we encounter some frustrations which were caused by original inconsiderate design mode. So, we enriched our UML and rebuilt the details several times to achieve the ideal goal.

Taking commutation part as an example, we first tested whether poll to get information from the server is a good idea. We found that it was not since polling means server passively sending the information to the client. If a server has some new game responses, they will not be sent until next polling. Then we tested implementing connection over WebSocket, which can exchange information between both sides and has a great multiplex feature.

7 Post-mortem Analysis

This project is to develop a novel monopoly game with more flexible rules and mechanisms. Different from traditional local software entity, this project is web-based which means a different design from original monopoly version like typical V5.

The design of the project is not finished at a time but modified many times since we have less experience and knowledge on design. The design of architecture is very hard at the beginning even though we had played the original game for some times. After referencing some engineering architecture we finally designed the architecture.

The coding of the back end was smooth since we both follow the same coding style much like PEP8 and we did not encounter many problems when conducting the integrated test. However, peer view and collaborative programming are necessary sometimes so as to keep a high and similar code quality.

Besides, the communication between front end and back end is a crucial part and we need to consider from both ends. It was the first time for us to design such thing so what we paid more attention to the data transmission, JSON here helped us a lot.

The test is identical difficult since this the turn-based game so when and where the imperceptible bugs might happen we can not know and those bugs are usually unrepeatable with the random result of dices. So test should not only involves with correct methods but also enough patience.

And some other engineering situations need to be taken into consideration like the scalability, concurrency, and security.

8 Lessons

This is the first time we've tried such a complicated project. We had to learn many things in order to accomplish the whole project. It's to develop a software but not to realize an algorithm or solve a specific problem. It's about designing rather

than just coding. So more aspects of engineering should be taken into consideration before and during coding. There're mainly two lessons we've learned from this project.

The first is that teamwork plays an important role in development. We realized this and reached an agreement on an appropriate work division and how to integrate everyone's work before starting coding. So, we finished our integration easily and quickly. Many important fundamental structures, if settled as early as possible, can help avoid recoding afterward. That's why we modified the traditional Waterfall model and paid much attention to the design before concrete implementation. We combined the whole project from time to time to check if there are inconsistencies.

The second is that timing is vital for development. We arranged a schedule properly and completely as well as strictly observe it so we can finish almost all the work that should be done. If not scheduled properly, we may not be able to finish this project.

9 Others

We had made some new friends through this project and the team became more tacit. We gained a lot of software design and development experience. We also learned some new tools, software, and frameworks.

Appendix I-State Diagram

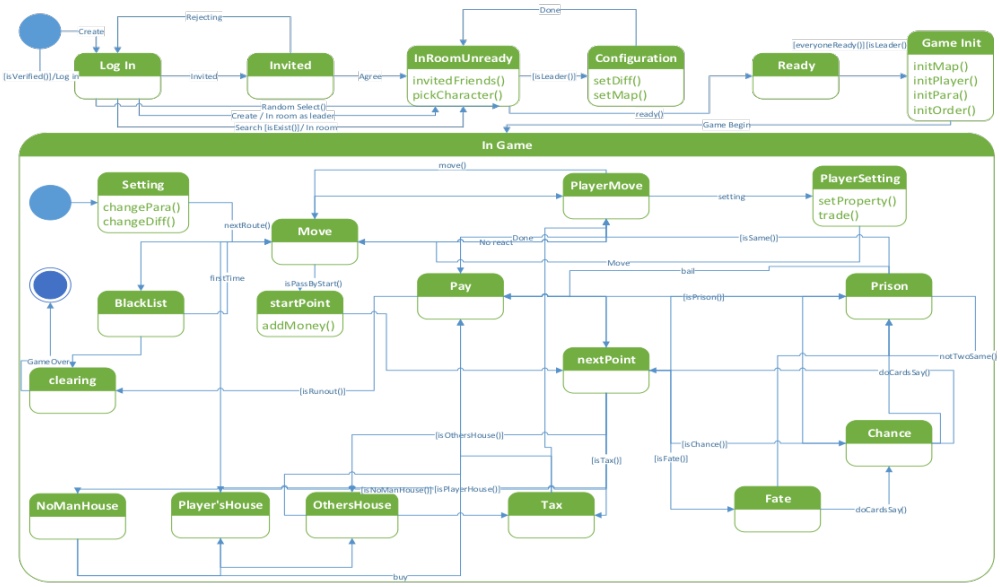

Figure 3: State Diagram

Appendix I-UML

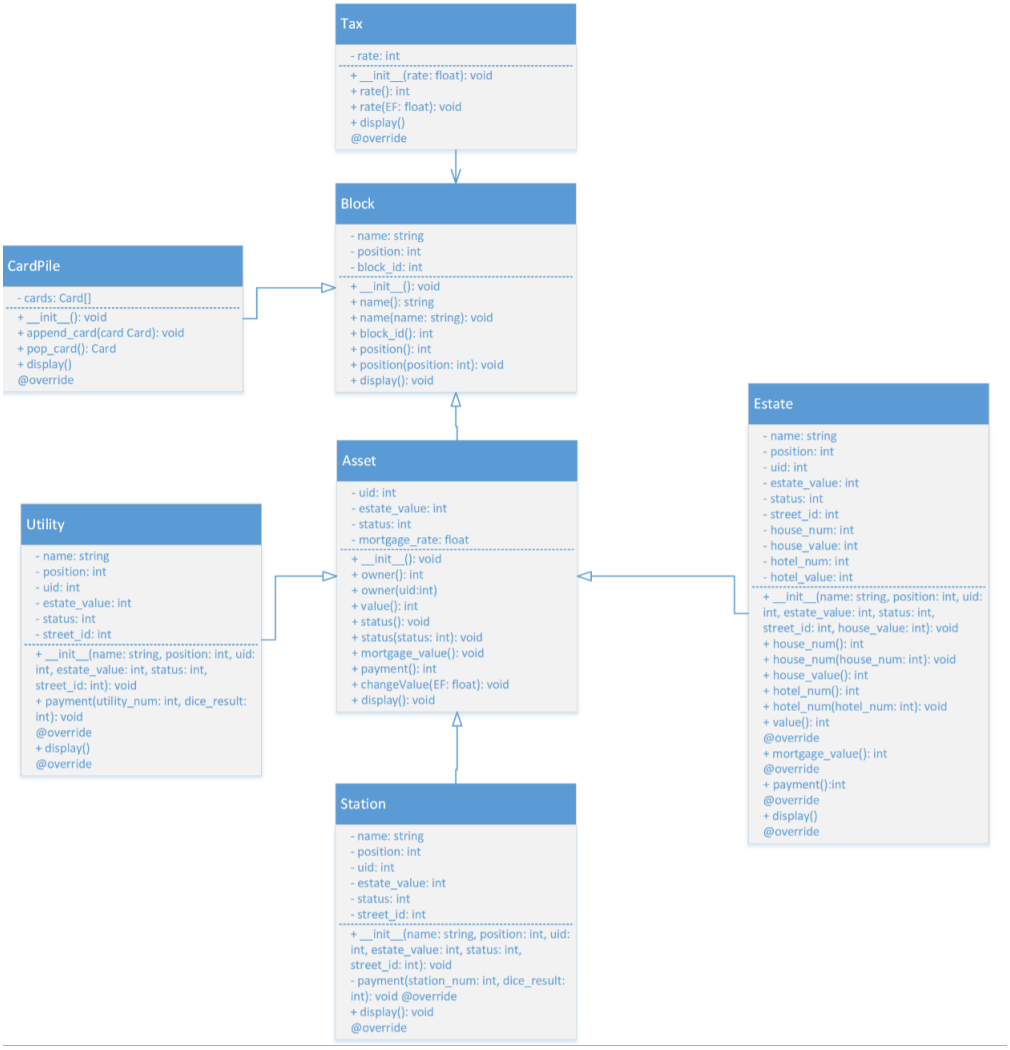

Figure 4: Block

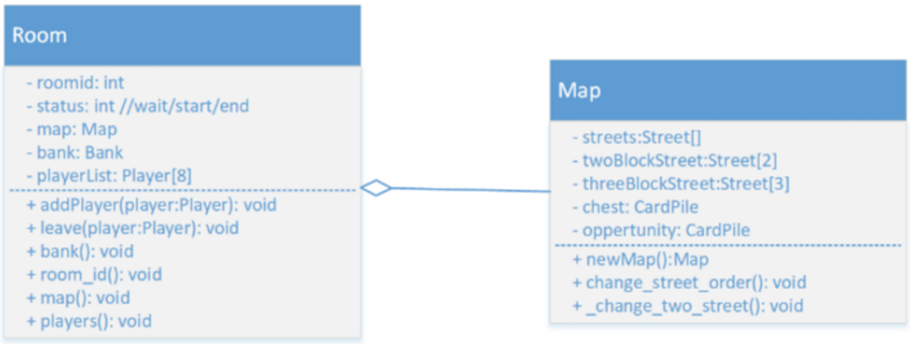

Figure 5: Room & Map

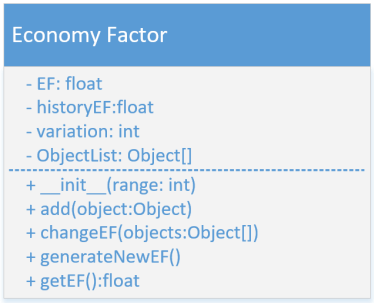

Figure 6: Economy Factor

Figure 7: Street

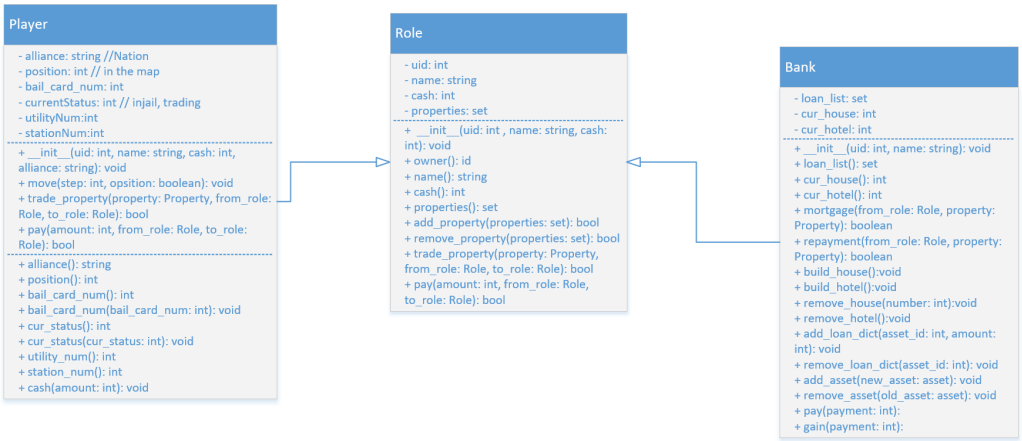

Figure 8: Role

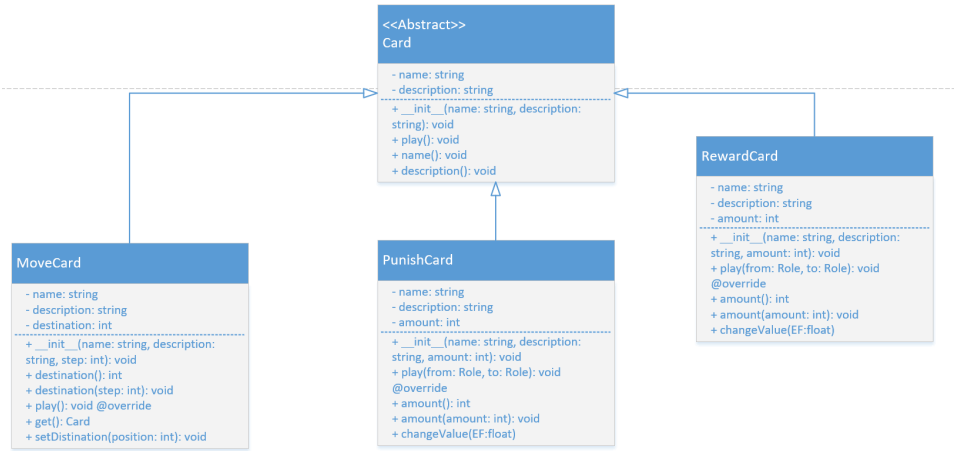

Figure 9: Card

Appendix II-UI

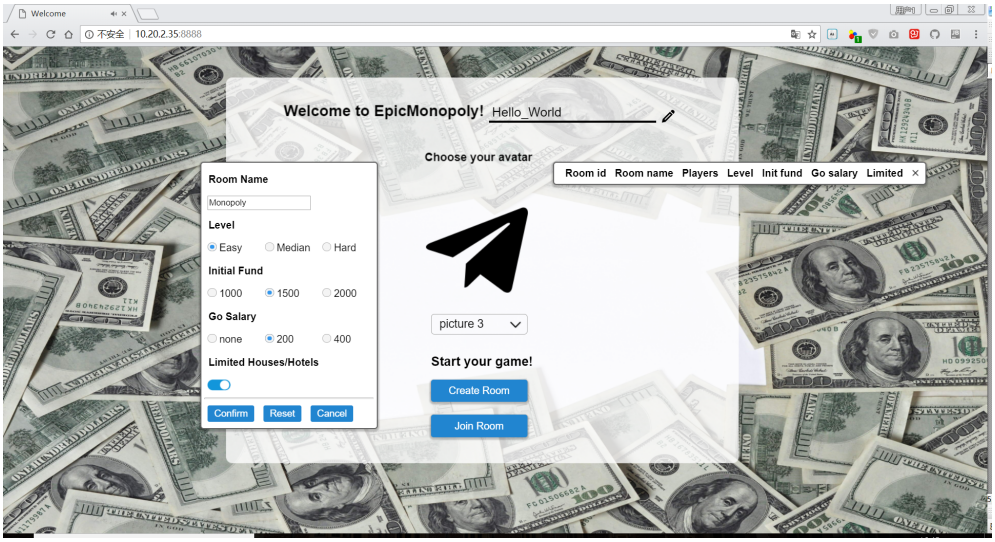

Figure 10: Settings

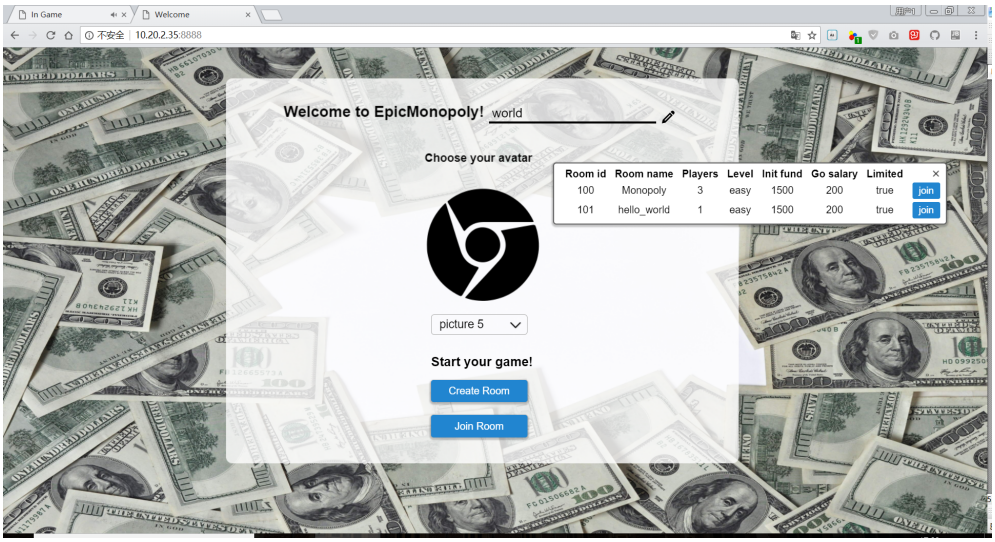

Figure 11: Room List

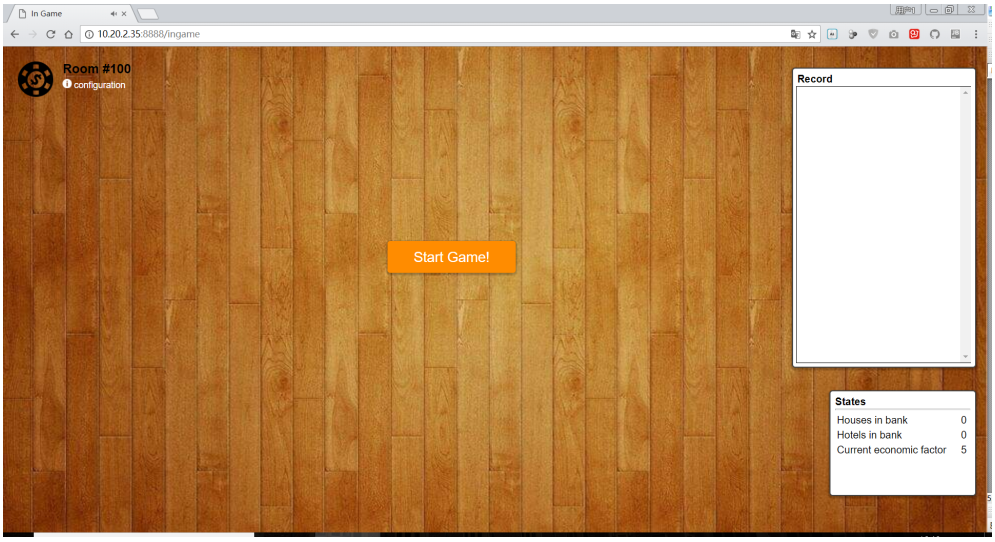

Figure 12: Start page

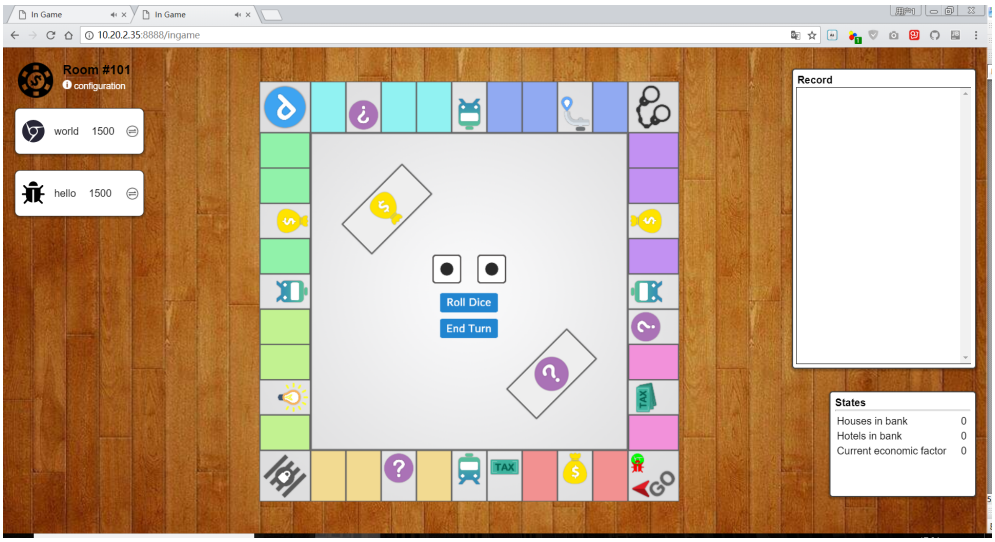

Figure 13: Main page 1

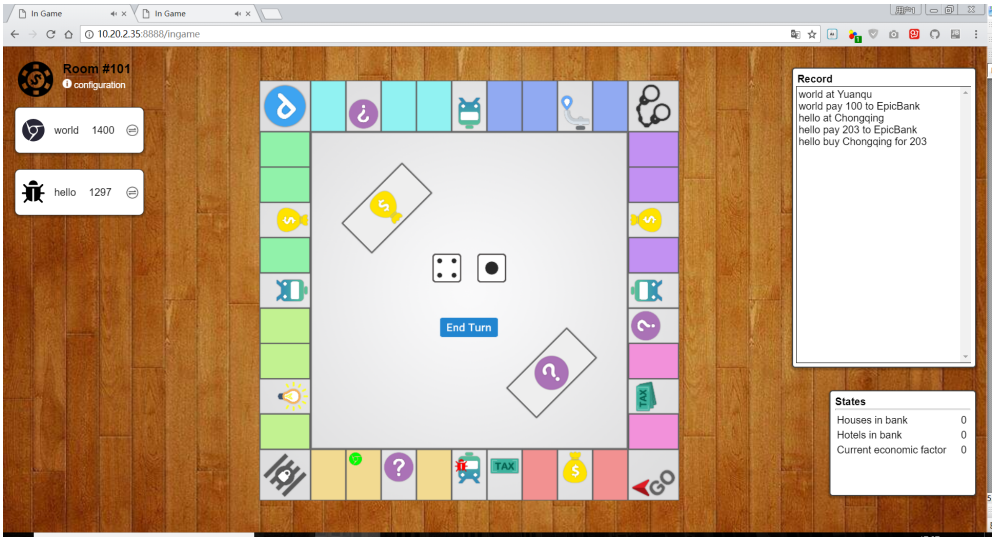

Figure 14: Main page 2

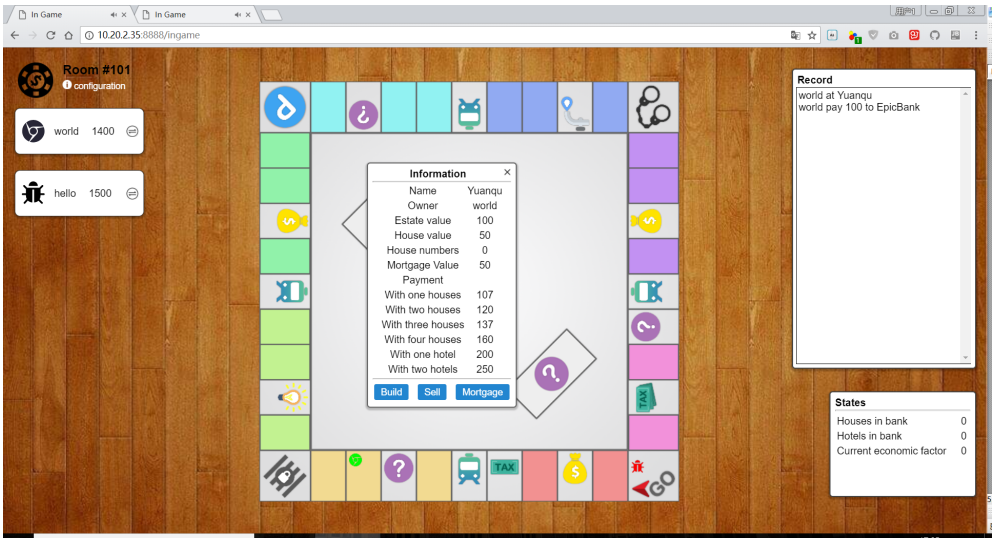

Figure 15: Management page

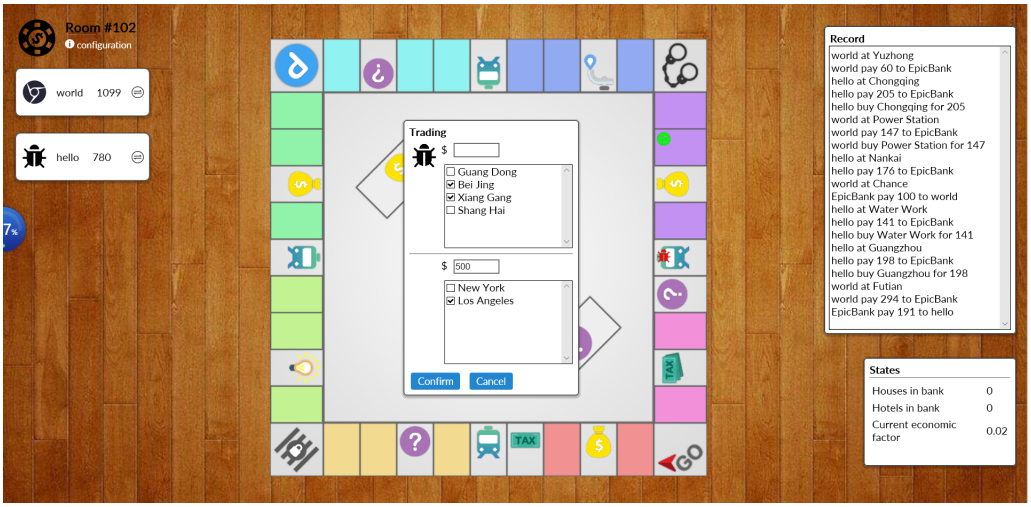

Figure 16: Trade page