Design Document - JAVAMAMAS

Specification

Our goal is to create a program that will allow users to play a game of monopoly locally with other players. When the program runs, each user will be able to assign themselves a character image, a character name and a fixed spot in the turn order. Once the game is started, a monopoly board will be created and players will take turns in the order assigned to them. On each players turn they will do the following:

- Roll/Move if they are still in the game.
- Buy Property - which we will call buyable tiles, and when they buy a whole lot (all properties of the same color), they can also start buying houses or hotels which increase the property value.
- Pay rent when they land on a property tile that is owned by another player.
- Collect 200$ from the bank if they passed the starting tile.
- Abide by tasks or consequences brought up when landing on a community chest tile.
- Go to jail if they land on the “Go to jail” tile

The program will control which players own each buyable tile. Additionally, the program will keep track of the number of tiles sold and remaining at all points in the game, as well as the funds of each player. There are 2 possible ways for the game to end: if all players except one go bankrupt (this remaining player wins), or if all the properties are sold (in which case the player with the highest net worth - balance plus property values - wins). A new feature - as per recommendation, was having a board class to remove some dependencies between controllers and entities. It also made the code for traversing the tiles more organized.

Design Choices and Why?

One major design decision we made was choosing to split up the code by class among one another, which we chose to do in order to make sure we have at least one person with an extremely thorough understanding of each class, and also so that we all implement a relatively equal amount of code. Also, we had a hard choice to make regarding the file structure of the project. Once a structure is chosen it should not change at all. So we decided to have every class in its own folder. This would allow the addition of extra helper files to a class if needed. This can be seen with the communityChestTile, multiple Card helper classes were needed, and are able to be added to the folder that communityChestTile is held in.

Additionally, a design choice we made to simplify development was to throw as many errors as possible. This allows for some using a class to know as quickly as possible when an issues happens. Stopping people from writing incorrect code assuming it is correct.

SOLID

Single Responsibility Principle

- All classes follow this principle. An example is, gameStatus file follows this principle because its only job is to determine different states of the game.

Open/ Closed Principle

- tile class with all of its subclasses.
- the tile class is closed for modification, however can add/ removed as many subclasses as needed to choose which tiles we want/ dont want

Liskov Substitution Principle

- Tile class uses this principle - it is a shared abstract interface
- Has subclasses of different tiles - buyable, community chest, jail, pass, prison, start, tile
- All the tiles implement from the interface and behave similarly as the objects from the interface

Interface Segregation Principle

- Initializable is a very small interface, which forces a single behaviour onto a child

Dependency Inversion Principle

- The Tile subclasses are all dependent on an abstraction in order to avoid the high-level, low-level class hierarchy and becoming dependent on higher levels

Clean Architecture and CRC Cards

- Entities: Our entities include the bank class, player class and the tile classes. The bank stores information about the number of houses/hotels and buyable tiles available. The player class stores information about each player (name, properties owned, balance). The tile classes stores information about players or tasks in relation to the corresponding tile (community chest, buyable, etc.). All in all, they do not enforce rules on any classes and are not dependent on other classes. Storage of the class data is done through classes in the outer layers.

- Use Cases: The board class creates a board which stores movement of players (an entity) in the game. The player status and game status check whether a player or the game can continue. This is done through checking properties in the entity classes such as player balance.

- Controllers: The Move controller controls movement through manipulation of the board class (use case class).

CRC Cards:
https://github.com/CSC207-UofT/course-project-javamamas/blob/main/phase%202/CRC%20cards.pdf

Scenario Walkthrough

In order to achieve our goal of creating a program that allows users to play Monopoly among other local players, we created a CRC model of our program to help visualize how we want our program to work. Our program starts with a Player which stores the player name and the tiles with respective houses or hotels for each tile. Additionally, the Player class stores their balance, which determines what the player can afford and whether they can keep playing. Players in this game can own tiles, which is represented by the abstract Tile class. The Tile class is a parent class to BuyableTile, CommunityChestTile, PrisionTile, StartTile, and GoToJailTile, which are each unique tiles in the game that extend the Tile parent class. The TileController class manages the Tiles part of the game. This class controls which players own which tiles, and what happens to players that land on specific tiles. The TileController class can also allow or block a player from purchasing a tile. The PropertyChecker class checks to see if a player can buy a specific tile, which refers to the balance the player has and whether that player can afford the purchase.

With all these purchases going on in the game, we have the Bank class to manage and keep track of what can be bought in the game. This class stores how many tiles and houses/hotels are able to be bought. As the game goes buy and players buy, sell, and pay rent on tiles, some players will run out of money. In our monopoly game, we deem a player unfit to continue playing if they are bankrupt. To keep track of a player's bankrupt and financial status, we have the PlayerStatus class. This class checks if a player is bankrupt, and checks a player's net worth which includes the sum of money the player has left and their tile values. With all these classes keeping track of statuses and managing purchases, we have a MoveController class to manage the moves during the game. This controller class controls whose turn it is in the game, the dice rolls, the current player positions on the board, and where the players can move respective to their dice rolls. The MoveController class also assigns a turn number for each player to help the game run more smoothly. With all these classes keeping the game functioning properly, we determine whether the game is finished with the GameStatus class. This use case class checks whether the game has ended if all the buyable tiles are sold, which is kept track by the bank. Additionally, the GameStatus class checks if all the players are bankrupt, which also indicates the game is over. Finally, we have our Main class which implements the GUI for the game and interacts with all of our classes to keep the game running.

Design Patterns

Observer Design Pattern

- have a GameObserver which is a lock on all game events, if one class requests a lock another cannot also have a lock. This allows for multiple objects to be related through a single class.

Strategy Design Pattern

- the Tile class uses this (implemented ihttps://github.com/CSC207-UofT/course-project-javamamas/pull/20/files). It has an onAction event which needs to do different things depending on the use case. The implementation of onAction is hidden within the subclasses.

Dependency Injection Design Pattern

- inside of PropertyChecker#canBuy take a Tile. This is because we want to be able to use any kind of tile, not just a buyable tile. Say if a player requests to buy a jail tile, they obviously cannot do this, but we must somehow check that a jail tile is not buyable, the common between jail and buyable tiles are is that they are both tiles. So we use a Tile type.

Template Method Design Pattern

- Inside of entities - we used an abstract class for Tile and had subclasses that were shared by the specific tiles (OnAction, canAddPlayer, addPlayer, removePlayer, getPlayers)

Use of GitHub Features

Our group used many features of GitHub including pull, commit, push, merge, pull requests and looking to Githubs conflict manager when trying to solve conflicts. More specifically, we pulled code as we went through everyone's classes when we met up, so that our local repository would be up to date, and we push any changes we made to our personal assigned classes. We also made sure to commit changes before merging or pulling so as not to lose any work. Additionally, when faced with the issue of accidentally overriding work, we had to use the revert pull request feature.

Code Style and Documentation

- When coding on IntelliJ, we struggled a lot with fixing warnings due to local issues regarding our programs
- By the end of the semester, we all seemed to understand how to use git a little better - earlier on…
    - The issue at hand was that for some of us (Rubaina and Dravin), only some classes were being read by the IntelliJ program, and others were not

    - Hence, we were constantly getting the error *"Cannot resolve symbol [insert class name]"*
- In terms of documentation, we have JavaDoc comments present to indicate what each class does
- We believe our code is understandable, as it is styled thoroughly with JavaDocs, and laid out in an organized manner

Testing

Most of our code was tested through j-unit tests aside from the following:

- Exceptions were difficult to test since j-unit is unable to handle it
- GUI was also not tested fully since it is a UI and requires using it, also due to the time constraint we were limited to what we could test at the end

Refactoring

It is evident we put in good effort to refactor our code, as multiple pull requests involve the same classes or code, which are edited to avoid code smells. For instance, one of many examples is where in the following pull request (https://github.com/CSC207-UofT/course-project-javamamas/pull/22) the Bank class was edited from what was originally submitted in pull request #21 (https://github.com/CSC207-UofT/course-project-javamamas/pull/21). We believe we avoided code smells thoroughly, and that our code avoids them at all times.

Code Organization & Packages

- For code organization, we decided to organize our classes based on the clean architecture model
- We have our individual classes within subfolders/packages of the different portions of the architecture model (interface, controller, entities, interface, observers, use_cases)
- Specifically within entities, we had split into between bank, player and tiles and we have our individual subclasses divided within them

Functionality

- We believe our program demonstrates what we specified it would to the best of our abilities considering the time constraint. We demonstrated this in our presentation for Phase 2
- Our program can store state state and load state

Accessibility Report

1. **Principles**

Principle 1: Equitable Use

- The game is designed for usage by anyone who is able to operate a computer
- No segregation based on operating system, device type, etc.
- Allows for anyone to play
    - If they do not know how to play, instructions are available for them to learn and facilitate in order to create an equity in game knowledge

Principle 2: Flexibility in Use

- Provides players choice in a variety of sections (player icon, number of players)
- Buttons are large and accessible for users - hard to miss

Principle 3: Simple and Intuitive Use

- Text used in instructions are simple and use easy language to get message across
- For next time, to make it more accessible, we would like to add prompts for the players so they know that it is their turn (in the case that they get distracted or confused about whose turn it is)

Principle 4: Perceptible Information

- Has limited text to provide straightforward information (no redundant text)
- Bolds essential information vs non important information
- Neutral colours that do not hurt the eye (colours from original Monopoly board)

Principle 5: Tolerance for Error

- Instructions on how to play the game is provided
- For future modifications, a fail safe should be implemented
    - If player presses wrong tile
    - Error shows up if they try to purchase a property that is not theirs

Principle 6: Low Physical Effort

- Board allows for minimal movements from players
- Maintain a neutral body position
- For future modifications, only one screen accommodates for multiple players which could be difficult for multiple people to come together
    - Make it so that multiple people can access the game from different devices

Principle 7: Size and Space for Approach and Use

- Clear line of sight for all board elements (players, tiles, cards, etc.)
- All components are easy to access for neutral standing/sitting position

2. There are two slightly different markets we would advertise our program to. Since Monopoly is a group or family game, we would promote our program to families and

college students, who are more likely to play this game with each other or with other family members and friends. The second market we would advertise our program to are younger people such as college students and teenagers. Since our program is an online version of Monopoly, the younger generations would be more likely to use our program more because of their high rate of technology use compared to older generations.

3. Our markets that we would focus on are also tied to how demographics play a role in which groups of people would use our program more. Middle to younger aged people such as families, students, teenagers, and children are more likely to use our program. Since Monopoly is family friendly and also a fun way for people to play a game together, anyone under between the ages of 8 and 50 or 60 are most likely to use this program and play Monopoly. A more specific demographic group would be college students and young adults between the ages 18-30, and teenagers and younger aged children between the ages 8 or 10 through 17. Since this is an online Monopoly program, tech-savvy generations would more likely use this program, which is why these two younger demographic groups are more likely to use this program.

   In contrast, there are a few demographic groups that will have few users of this program. As a virtual version of Monopoly, individuals who are visually impaired would need assistance in playing our game, for example someone who is able to read out the tiles and what is going on in the game. Another demographic group that wouldn't use our program are technologically challenged people. Since Monopoly already exists in the form of a popular board game, the chance of technologically challenged individuals using our program is slim.

Progress Report

Struggles
The following are struggles we have faced…

- JavaFX learning curve
- JavaFX animations run on a separate thread and they don't run fully sometimes.
- JavaFX doesn't allow for percentage positioning
- JavaFX cascading style sheets are very limited
- Different members use maven and gradle which made code syncing very difficult

What has worked so far?
Recap…
The process we used in Phase 1 that led to the most efficient outcome was splitting up classes amongst our group due to our conflicting schedules. This worked really well so we followed a similar process for phase 2.

Since Phase 1…
We have been able to meet more consistently in person. We had much better collaboration for the GUI (as it was a learning curve for the majority of us) and were able to work more collaboratively by assigning each member a GUI scene to work on which come together to form our entire GUI.

Work Breakup and Next Steps
After Phase 1 and the provided feedback, we continued on creating our GUI which will communicate with our code. We distributed the following GUI "scenes" as follows:

- Shivanshi:
  - Playing Page
- Dravin:
  - Loading Page
- Yusra:
  - Player Selection Page
- Dennis:
  - Home Screen
- Rubaina:
  - Help Page

We collaborated on combining all of these scenes together to form our game.

Giancarlo adjusted the board class based on feedback from phase 1.

Pull Requests:
Yusra:

1. https://github.com/CSC207-UofT/course-project-javamamas/pull/22
- I believe this pull request demonstrates a significant contribution as the Bank (which was edited in this part, but originally pushed in pull request #21) and PlayerStatus classes, as with all classes, are vital to our program. This pull request is the implementation/editing of these two classes which were my responsibility in our code, while my partners similarly had classes to implement that, when combined, lead to our coded program.
2. https://github.com/CSC207-UofT/course-project-javamamas/pull/39
- This pull request is to pull the code for the selection page of our GUI, which is what is used for players to be able to choose their name and icon for the game.

Dennis:

1. https://github.com/CSC207-UofT/course-project-javamamas/pull/10
- This pull request involved the skeleton code for our program, and so it played a significant role in the implementation of our program. Also, it set the method signatures and packages/files, which provided structure for our group.
2. https://github.com/CSC207-UofT/course-project-javamamas/pull/48
- This pull request is significant because I have added J-unit tests which allow us to verify that the code works as we want it to.

Giancarlo:

1. https://github.com/CSC207-UofT/course-project-javamamas/pull/13
   This pull request is important because it was necessary to have all the tiles in the board created and organized so that the MoveController could interact with each tile regardless of its type. The board was a use case class so it helped preserve clean architecture when called by the MoveController. It was later updated to read off a file as per the feedback.

Rubaina:

1. https://github.com/CSC207-UofT/course-project-javamamas/pull/28
   This pull request was important as it was necessary to have all the code up to date. It was specifically a pull merge request as many changes were made to the code on my end and it was necessary to make sure all the code was up to date. Specifically, changes were made to portions of the CommunityChest file.

Dravin:

1. https://github.com/CSC207-UofT/course-project-javamamas/pull/30
   Coded the GameStatus and PropertyChecker classes, which were the classes I was assigned to write for Phase 1. I committed them for the group to have access to and implement in the program, however, there were a few issues with merging, therefore my groupmate Dennis had to help merge for me.

Shivanshi:

1. https://github.com/CSC207-UofT/course-project-javamamas/pull/26

Coded StartTile for Phase 1 and pushed it. I learned a lot about github as there were a lot of issues in pushing my branch on git which my team helped me with.