CS246 A5 Watopoly Design Document

Isaac Nguyen, Aaron Zhang, Rahel Tamrat

Introduction

This document will provide a thorough synopsis of our implementation of the game Watopoly as a part of CS246 Assignment 5. More specifically, we will elaborate extensively on the software engineering concepts and techniques that were utilized during the development process and the design decisions that came to our program structure. We will also refer to various engineering challenges that occurred and our approach(es) to overcoming them. Through reading our design document, instructors will have a better understanding of how our group applied numerous C++ concepts to create a robust program with clever software design.

Overview

Our program structure was designed with the **MVC design pattern** in mind. To implement this, we used **encapsulation** to separate each component of this design pattern such that we achieve **high coupling** and **low cohesion** across all modules of our program.

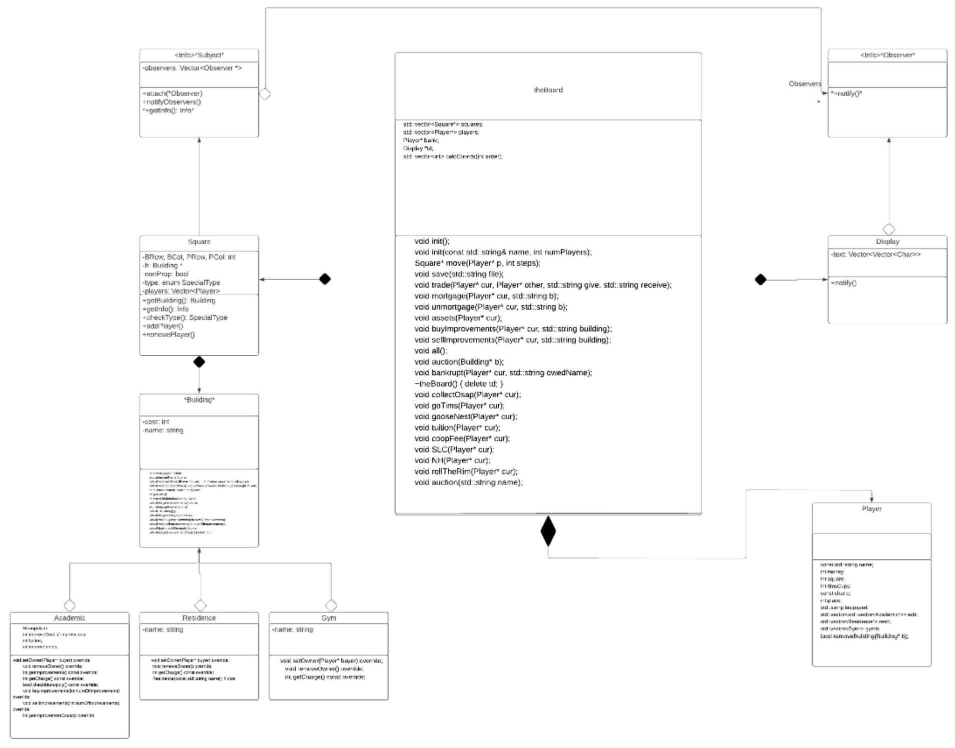
To illustrate this:

- Model: theBoard Class (board.h)
  - Purpose: contain and operate all necessary logic of our game. Communicates with our controller to receive necessary information on our p
  - Encapsulated all necessary game logic components such that no information about board's logic is accessible in main.
- View: Display Class (display.h)
  - Purpose: updates the graphical display whenever a player has moved, or an improvement has been made.
  - Note that these two actions are the only cases when changes to the display needs to be made, hence the only times display.h would be accessed
  - Only contains code related to what is displayed and is the only place where changes to the display are made

- Only has access to components necessary for updating the display:
  - square.h (Square Class) - required for knowing where on the board to update, and which players are on a given square
  - player.h (Player Class) - required for which players to display on the board

- Controller: Watopoly.cc (essentially main.cc)
  - Purpose: exclusively used for receiving input from our player entities and acts as a container for our model (board class).
  - Only has access to necessary components to carry out controller specific tasks:
    - shuffle.h (Shuffle Class) - required for **Dice Rolling** functionality used by our player entities.
    - player.h (Player Class) - required to initialize player entities that will be interacting with our program on game load/start. Notice that players interact with our model, our controller is the bridge to this relationship.
    - board.h (Board Class) - required to start and run a game. Since this is our model, it operates all our logic and must be contained by our controller.

UML



<Info>*Subject*
-observers: Vector<Observer *>
+attach(*Observer)
+notifyObservers()
+getInfo(): Info
theBoard
std::vector<Square*> squares;
std::vector<Player*> players;
Player* bank;
Display *td;
std::vector<int> calcCoords(int order);
void init();
void init(const std::string& name, int numPlayers);
Square* move(Player* p, int steps);
void save(std::string file);
void trade(Player* cur, Player* other, std::string give, std::string receive);
void mortgage(Player* cur, std::string b);
void unmortgage(Player* cur, std::string b);
void assets(Player* cur);
void buyImprovements(Player* cur, std::string building);
void sellImprovements(Player* cur, std::string building);
void all();
void auction(Building* b);
void bankrupt(Player* cur, std::string owedName);
~theBoard() { delete td; }
void collectOsap(Player* cur);
void goTims(Player* cur);
void gooseNest(Player* cur);
void tuition(Player* cur);
void coopFee(Player* cur);
void SLC(Player* cur);
void NH(Player* cur);
void rollTheRim(Player* cur);
void auction(std::string name);
Observers
<Info>*Observer*
+notify()
Display
-text: Vector<Vector<Char>>
+notify()
Square
-BRow, BCol, PRow, PCol: int
-b: Building *
-nonProp: bool
-type: enum SpecialType
-players: Vector<Player>
+getBuilding(): Building
+getInfo(): Info
+checkType(): SpecialType
+addPlayer()
+removePlayer()
Building
-cost: int
-name: string
Player
const std::string name;
int money;
int square;
int timsCups;
const char p;
int place;
std::string lastpayed;
std::vector<std::vector<Academic*>> acb;
std::vector<Residence*> resb;
std::vector<Gym*> gymb;
bool removeBuilding(Building* b);
Academic
Monopoly m;
int improveCost; // improve cost
int tuition;
int improvements;
void setOwner(Player* buyer) override;
void removeOwner() override;
int getImprovements() const override;
int getCharge() const override;
bool checkMonopoly() const override;
void buyImprovements(int numOfImprovements) override;
void sellImprovements(int numOfImprovements) override;
int getImprovementCost() override;
Residence
-name: string
void setOwner(Player* buyer) override;
void removeOwner() override;
int getCharge() const override;
Residence(const std::string name); Vcost
Gym
-name: string
void setOwner(Player* buyer) override;
void removeOwner() override;
int getCharge() const override;

Design

To effectively address the various design challenges inherent in our project, we employed a number of specific techniques. One such technique involved the implementation of a "**Building**" abstract superclass, which enabled us to polymorph different property types such as "**Academic**", "**Residence**", and "**Gym**", with each being represented as concrete derived classes. This approach facilitated an efficient system to classify and isolate type specific behaviour.

Furthermore, to ensure that the charging functionality varied appropriately between different building types, we utilized **pure virtual functions** for each building type. **Non-pure virtual functions** were mainly utilized for "**Academic**" types, as we needed to account for the specialization of monopolies. This design decision allowed us to implement polymorphic properties for the "**Building**" superclass. Furthermore, we ensure encapsulation for fields that are shared between all derived classes (i.e object name).

To manage players effectively, we chose to use a struct as they represent separate entities within the controller that only hold information and do not require encapsulation. Using classes to represent players will complicate our program and is not necessary in this case. Our player structure contains three different vectors, with each representing a building type to better organize their owned properties. This design choice allowed us to maintain clear and concise record-keeping for each player's properties.

For academic property types, we utilized a vector of size 8, with each containing vectors of pointers to "**Academic**" types. This approach allowed us to easily keep track of the 8 different monopoly types a player might own, making it straightforward to perform monopoly checks and access each monopoly type correctly.

Another design decision that contributed to the consistency of the program was the choice to use **maps** as an information conversion tool. For example, we made a map of each **"Academic"** building types with the identifier being the building name and the corresponding value being a vector of integers. Each index within the vector reflects a numeric property about each specific building. This includes monopoly type, improvement costs, buying cost, etc. We were able to efficiently obtain information about each building from within the "**Building**" class while also maintaining utmost consistency as the information is unchanging. You will be able to see the use of this technique throughout the entirety of our program to more efficiently store and retrieve invariating information across different classes.

In addition, the overall structure of our game revolves around the construction of a vector of **"Square"** objects in our "**theBoard**" class, where each has a field called **"SpecialType"** to specify the type of square it represents. This is implemented as an enum class to specify the 12 possible types (refer to line 11 in player.h). If it is one of the **"Academic"**, **"Residence"**, or **"Gym"** types, the **"Square"** object acts as a container for each of the derived building objects. A polymorphic pointer in **"Square"** pointing to the building object is used to connect the two objects. Else, this pointer is set to nullptr for other types. The order of its construction in **theBoard** corresponds to the order of squares the players are able to advance in.

Another crucial component in our program is the use of the **observer design pattern** to manually update the graphical display whenever a square object gets updated. Our **"Display"** class is essentially a vector of vectors of characters, similar to the grid class in A4Q3. It is an observer to every single square in "**theBoard**", and whenever a visually changing behaviour occurs in "**theBoard**, the vector of vectors of characters gets automatically updated. These two behaviours are improvement changes and movement of players; which is made possible by keeping track of the coordinates of where the visual display for building and player start on each specific square. This is done through passing an **"Info"** object in **notifyObservers().** When sent to **"Display"**, **"Info"** contains necessary information to update our display (refer to info.h). This is extremely useful as our graphic display will always be asynchronously updated without requiring any manual checks.

From a high level standpoint, the main function in our program is responsible for receiving input and calling the appropriate functions from our model, **"theBoard"**. This also include loading saved files and creating new games. Our **"theBoard"** class acts as the centrepiece and is responsible for initializing most aspects of the game, making it a critical component of our program.

Our main function is implemented in a manner that ensures program stability. To further elaborate, for each user input, we ensure that exception handling is implemented. Such that in the case of unexpected user behaviour, an exception will automatically be thrown and the program will exit before any system instability occurs. This restricts the user interaction with the program and only ensures valid behaviour to occur.

One particularly clever implementation in our main function was the initialization of player objects in a vector. This allowed us to keep track of each player's turn in the form of their index in the player vector. This made it easy to implement turn-based gameplay and ensured that each player had equal opportunities to make moves.

Another noteworthy implementation in our main function was the handling of player commands. We maintained a map of commands that each player had access to, and depending on the situation, the logic within the main function would update the available commands for the current player. At the start of each new player's turn, the command map was reset to ensure consistency in gameplay. This design decision was made to ensure consistent gameplay without any overlapping behaviour between each player entities.

Finally, the randomizing functionality of the program was implemented through creating a **"Shuffle"** class. The **"Shuffle"** class allows us to customize the data each **"Shuffle"** object randomizes by specifying the probabilities in a vector. This includes rolling dice or drawing Tims Cups at certain squares. This done through specifying the probabilities of each possible outcome in a vector and constructing a **"Shuffle"** object using the specified vector. For example, the vector used for constructing a **dice "Shuffle"** object would contain integers from 1 to 6. Refer to our plan.pdf for Due Date 1 for more information.

Overall, our program's design and implementation addressed several key challenges, including maintaining player and building data, handling player turns, and providing an intuitive interface for user input. Through our use of polymorphism, virtual functions, vectors, and observer design pattern, we were able to create a flexible and extensible program that meets the project requirements.

Resilience To Change

One of the key aspects of a well-designed program is its ability to adapt to changes in the program specification. Our program has been designed to be highly resilient to change by incorporating a number of design elements that facilitate flexibility and extensibility.

Firstly, our program has been designed using an object-oriented approach which allows us to encapsulate functionality within various objects. This means that any changes to a particular object can be made without affecting the rest of the program. For example, if we wanted to add a new building type, we could simply create a new derived class from our Building abstract superclass, and our program would be able to accommodate this change without affecting any other parts of the code.

In addition, our program utilizes design patterns which facilitate changes to the program specification. For example, we have implemented the Observer pattern in our display class, which allows the display to be updated whenever a change occurs in the program state. This means that if we were to make changes to the program logic, the display would be able to adapt and display the new state of the program without requiring any changes to the display code.

Furthermore, our program has been designed to be highly modular, with each component of the program encapsulating a specific functionality. This means that any changes to a particular component can be made without affecting the rest of the program. For example, if we wanted to change the way in which player movement is handled, we could make changes to the movement function within our Board class, without affecting any other parts of the program. More over this allows for high cohesion and low coupling between each module as there are little dependencies between each class inorder for them to be functional.

Finally, our program utilizes data structures such as vectors and maps, which facilitate changes to the program specification. For example, we use a vector to store our player objects, which allows us to easily add or remove players from the game. We also use a map to store the available commands for each player, which allows us to easily add or remove commands as required. These data structures will allow us to manually change invariating information depending on the specific changes to be implemented.

Our program has been designed to be highly resilient to change, incorporating object-oriented programming principles, design patterns, modular design, and flexible data structures. This means that our program is able to adapt to changes in the program specification while maintaining consistent gameplay behaviour and ensures strong encapsulating capabilities.

Answer to Questions

Q: Would the Observer Pattern be a good pattern to use when implementing a gameboard? Why or why not?

A: Yes. From our original plan, we have changed our implementation of the observer pattern to be used with just display and square. This decision was made because we recognized that other implementations would be unnecessary as relationship between other components/modules do not require asynchronous updates. Whereas the only relevant modules are display and squares that accurately represent this relationship

Q: Suppose that we wanted to model SLC and Needles Hall more closely to Chance and Community Chest cards. Is there a suitable design pattern you could use? How would you use it?

A: Yes, one possible design pattern that could be used to model SLC and Needles Hall more closely to Chance and Community Chest cards is the Decorator pattern. This pattern allows us to dynamically add new behaviors or features to an object by wrapping it with another object that has these behaviors or features. For example, we could create a BuildingDecorator class that wraps around a Building object and adds the behavior of drawing a card and executing its effects. We could then create SLCDecorator and NeedlesHallDecorator classes that inherit from BuildingDecorator and add their own card drawing and effect execution behaviors.

However, it's worth noting that we didn't use the Decorator pattern in our implementation of the Monopoly game. We instead utilized inheritance and polymorphism to model the different types of buildings and their behaviors. While the Decorator pattern could be a suitable alternative, it ultimately depends on the specific design goals and requirements of the project.

Q: Is the Decorator Pattern a good pattern to use when implementing Improvements? Why or why not?

A: In our implementation, we didn’t use the decorator pattern because we chose to implement a map to keep track/convert information on improvements for each building. This approach allows flexibility in how we access and retrieve information without having to implement the decorator pattern.

Final Questions

Q: What lessons did this project teach you about developing software in teams? If you worked alone, what lessons did you learn about writing large programs?

A: This project taught us several important lessons about developing software in teams. Firstly, we learned that communication is key to ensure that everyone is on the same page and that there is a shared understanding of the project requirements. We also learned the importance of having a well-defined project plan and dividing tasks appropriately to ensure that all team members are contributing effectively.

Another lesson we learned is the importance of testing and debugging. We found that it was essential to thoroughly test each component of our program and to have a plan in place for debugging any issues that arose. This was especially important when merging code from different team members to ensure that everything worked together smoothly.

If working alone, this project teaches the lesson of writing large programs. Specifically, it highlights the importance of breaking down complex problems into smaller, more manageable parts, and testing each part as you go along. It also emphasizes the importance of having a clear plan and structure for your code, as well as taking the time to document your code and implement best practices for readability and maintainability.

2. What would you have done differently if you had the chance to start over?

If we had the chance to start over, there are a few things we would do differently. Firstly, we would spend more time refining our initial project plan and requirements to ensure that everyone on the team had a clear understanding of the project's goals and specifications. Additionally, we would have incorporated more thorough testing and debugging practices earlier on in the development process to catch issues before they became more complex. Lastly, we would have implemented more frequent check-ins and code reviews to ensure that all team members were working cohesively and to catch any issues or inconsistencies earlier on.