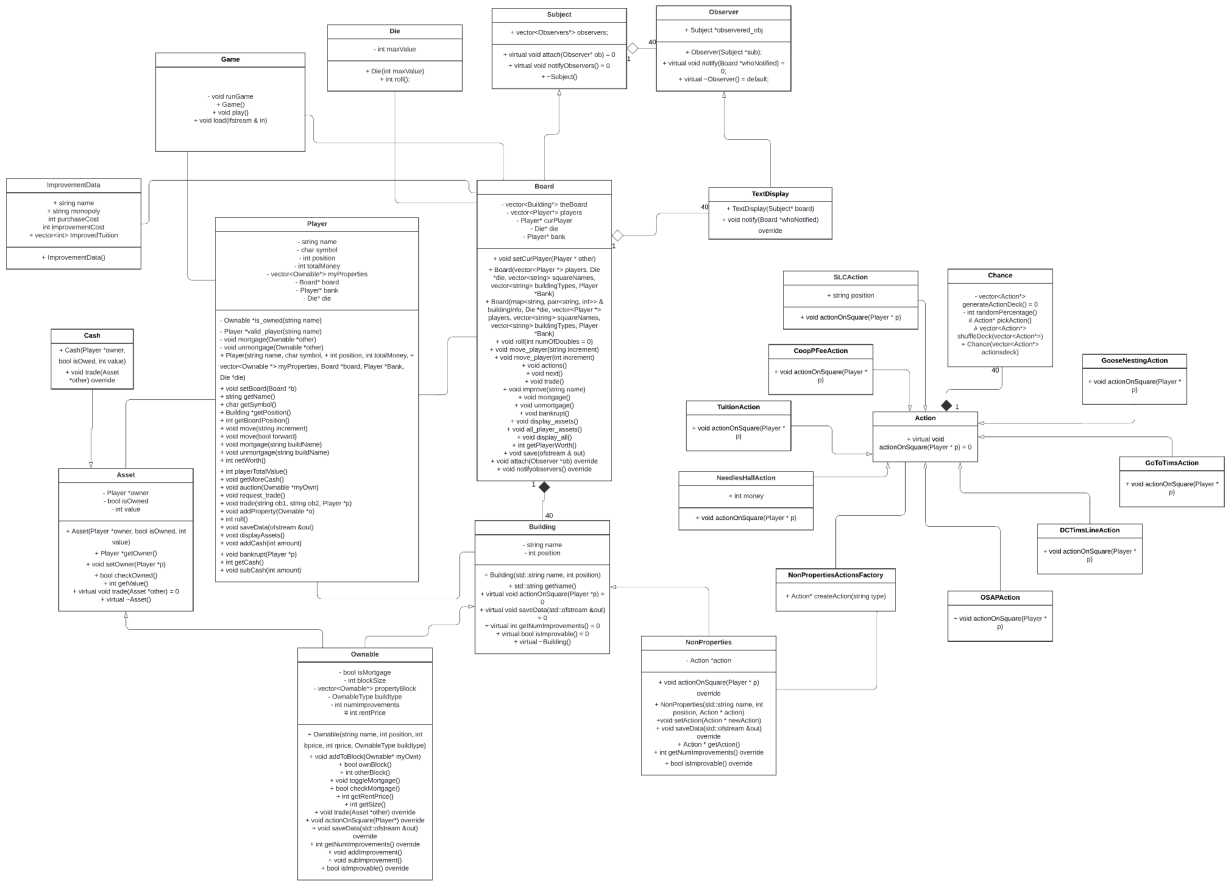
CS246 - DD2 Documentation

For the CS246 assignment 5 project, our group (William Liu, Allan Yin, Jeffrey Zhao) have chosen to implement the video game Watapoly which is a variant of the game, Monopoly, but with a board based on the University of Waterloo campus. As described in the project guidelines, our implementation will be broken down into the following components: property buildings, non-property buildings, players, the board, and finally a class which would allow the users to play the game.

Updated UML:


Game
- void runGame
+ Game()
+ void play()
+ void load(ifstream & in)
Die
- int maxValue
+ Die(int maxValue)
+ int roll();
Subject
+ vector<Observers*> observers;
+ virtual void attach(Observer* ob) = 0
+ virtual void notifyObservers() = 0
+ ~Subject()
Observer
+ Subject *observered_obj
+ Observer(Subject *sub);
+ virtual void notify(Board *whoNotified) = 0;
+ virtual ~Observer() = default;
40
1
ImprovementData
+ string name
+ string monopoly
int purchaseCost
int improvementCost
+ vector<int> ImprovedTuition
+ ImprovementData()
Board
- vector<Building*> theBoard
- vector<Player*> players
- Player* curPlayer
- Die* die
- Player* bank
+ void setCurPlayer(Player * other)
+ Board(vector<Player *> players, Die *die, vector<string> squareNames, vector<string> buildingTypes, Player *Bank)
+ Board(map<string, pair<string, int>> & buildingInfo, Die *die, vector<Player *> players, vector<string> squareNames, vector<string> buildingTypes, Player *Bank)
+ void roll(int numOfDoubles = 0)
+ void move_player(string increment)
+ void move_player(int increment)
+ void actions()
+ void next()
+ void trade()
+ void improve(string name)
+ void mortgage()
+ void unmortgage()
+ void bankrupt()
+ void display_assets()
+ void all_player_assets()
+ void display_all()
+ int getPlayerWorth()
+ void save(ofstream & out)
+ void attach(Observer *ob) override
+ void notifyobservers() override
TextDisplay
+ TextDisplay(Subject* board)
+ void notify(Board *whoNotified) override
40
1
Player
- string name
- char symbol
- int position
- int totalMoney
- vector<Ownable*> myProperties
- Board* board
- Player* bank
- Die* die
- Ownable *is_owned(string name)
- Player *valid_player(string name)
- void mortgage(Ownable *other)
- void unmortgage(Ownable *other)
+ Player(string name, char symbol, + int position, int totalMoney, + vector<Ownable *> myProperties, Board *board, Player *Bank, Die *die)
+ void setBoard(Board *b)
+ string getName()
+ char getSymbol()
+ Building *getPosition()
+ int getBoardPosition()
+ void move(string increment)
+ void move(bool forward)
+ void mortgage(string buildName)
+ void unmortgage(string buildName)
+ int netWorth()
+ int playerTotalValue()
+ void getMoreCash()
+ void auction(Ownable *myOwn)
+ void request_trade()
+ void trade(string ob1, string ob2, Player *p)
+ void addProperty(Ownable *o)
+ int roll()
+ void saveData(ofstream &out)
+ void displayAssets()
+ void addCash(int amount)
+ void bankrupt(Player *p)
+ int getCash()
+ void subCash(int amount)
Cash
+ Cash(Player *owner, bool isOwed, int value)
+ void trade(Asset *other) override
Asset
- Player *owner
- bool isOwned
- int value
+ Asset(Player *owner, bool isOwned, int value)
+ Player *getOwner()
+ void setOwner(Player *p)
+ bool checkOwned()
+ int getValue()
+ virtual void trade(Asset *other) = 0
+ virtual ~Asset()
1
40
Building
- string name
- int position
+ Building(std::string name, int position)
+ std::string getName()
+ virtual void actionOnSquare(Player *p) = 0
+ virtual void saveData(std::ofstream &out) = 0
+ virtual int getNumImprovements() = 0
+ virtual bool isImprovable() = 0
+ virtual ~Building()
Ownable
- bool isMortgage
- int blockSize
- vector<Ownable*> propertyBlock
- OwnableType buildtype
- int numImprovements
# int rentPrice
+ Ownable(string name, int position, int bprice, int rprice, OwnableType buildtype)
+ void addToBlock(Ownable* myOwn)
+ bool ownBlock()
+ int otherBlock()
+ void toggleMortgage()
+ bool checkMortgage()
+ int getRentPrice()
+ int getSize()
+ void trade(Asset *other) override
+ void actionOnSquare(Player*) override
+ void saveData(std::ofstream &out) override
+ int getNumImprovements() override
+ void addImprovement()
+ void subImprovement()
+ bool isImprovable() override
NonProperties
- Action *action
+ void actionOnSquare(Player * p) override
+ NonProperties(std::string name, int position, Action * action)
+void setAction(Action * newAction)
+ void saveData(std::ofstream &out) override
+ Action * getAction()
+ int getNumImprovements() override
+ bool isImprovable() override
SLCAction
+ string position
+ void actionOnSquare(Player * p)
Chance
- vector<Action*> generateActionDeck() = 0
- int randomPercentage()
# Action* pickAction()
# vector<Action*> shuffleDeck(vector<Action*>)
+ Chance(vector<Action*> actionsdeck)
40
1
CoopPFeeAction
+ void actionOnSquare(Player * p)
GooseNestingAction
+ void actionOnSquare(Player * p)
TuitionAction
+ void actionOnSquare(Player * p)
Action
+ virtual void actionOnSquare(Player * p) = 0
GoToTimsAction
+ void actionOnSquare(Player * p)
NeedlesHallAction
+ int money
+ void actionOnSquare(Player * p)
DCTimsLineAction
+ void actionOnSquare(Player * p)
NonPropertiesActionsFactory
+ Action* createAction(string type)
OSAPAction
+ void actionOnSquare(Player * p)

Design:

For our design, we utilized several levels of class hierarchy as well as select design patterns that we thought would make our program more abstract. From the top, our program contains a game file that encapsulates the Watopoly game. This way, we could enable a way to decide if we were playing Watopoly in a testing environment, reading in from a saved file, or starting a game completely from scratch. We decided that such logic is not directly related to the game, and so, should be handled by a wrapper class. In addition, the use of a game class to control the board enabled us to keep objects that were crucial for the game, but not part of the board, to be able to interact with the board. Examples of these include the dice, the text display class, and the players. The game object also allowed us to ensure certain invariants of the board, such as a maximum of 40 buildings, a minimum of 2 players, a maximum of 8 players, and valid player character/symbol selections. To start the game, we simply passed pointers of the necessary objects to the board constructor, so that the board would have access to the pieces it needed to begin the game. To maintain polymorphic principles, our board was implemented as a 1 dimensional vector containing building object pointers. In our project, buildings were abstract base class objects, that separated into ownable objects and non-property objects. Through this, the board's ownable and non-property objects would be accessed through base class pointers.

Each time a player lands on a square or building, a certain action takes place. We created an abstract base class called building and used that class as a basis for the non-property class which were used to represent squares such as DC Tims Lane, Collect ASAP, Goose Nesting, etc. as well as for the ownable class which were used to represent the residence, gym, and academic building squares in the game. The building superclass contained a virtual function called actionOnSquare, which would run the action we define for each building. For the ownable objects, the action taken depends on what type of ownable property it is, the number of improvements, who owns it, etc. This action does not change often and the payment method is an inherent part of an ownable building. Contrastingly, for all non-properties squares, we used Action objects that define a virtual function called runAction. This is useful because we can separate the functionality of landing on the square from the class itself. Moreover, it allowed us to use the Factory method when initializing our squares. If we ever wanted to modify how to create our squares, or add in different types of non-properties squares, we just needed to modify the factory. Also, it came in very handy when making the SLC and Needle's Hall square act more like Chance/Community Chest cards. We would have not been able to have a deck and a discard pile of just functions, but with the Action class, we can do so.

In addition, the observer design pattern was employed to print the board the output stream. This was implemented through the TextDisplay class. To print the board, a TextDisplay object was instantiated and was added as an observer of the Watopoly board. Every time a player moved to a new square, or if an improvement was made to an Academic Building, the board would notify its observers of the change. As an observer, the TextDisplay object would print the board, with the players' positions updated and the improvements of buildings displayed. The observer pattern was primarily employed to separate the code that operated the logic of the board from the code that was only concerned with printing the state of the board. This way, if the implementation of the board would change, the code to display the code would not change, and vice versa. This helps us to increase the maintainability and reusability of our code.

Resilience to Change:
In many ways, our program is able to adapt to change. For instance, to support trading between players, ownable buildings was made into a subclass of Assets. Here, we created another class called Cash, which also inherited from Assets. When the program reads in the input for trading, instead of directly processing the trade, the trade function creates a Cash object. Now, we are effectively attempting to trade a Cash object and an Ownables object, which can be done polymorphically through the use of base class pointers. This way of coding allows to not only trade buildings with money, but anything that inherits from the Assets abstract base class. For instance, suppose we introduce an additional member to each player called Stocks. In this new game, in addition to owning and being able to read money and buildings, we now can also trade a Stock object. Instead of overriding the trade function, we can make Stock inherit from the Asset class and again, execute trades through base class pointers. In addition, our use of the factory method allows for lots of flexibility. Each square has an action player must execute upon landing. What if we wanted to change what action occurs? Instead of

rewriting the code for the building, we only need to change the building's action object. Through this, the risk of errors is significantly lower since the building does not know what action it needs to perform, it just knows it must perform an action. Likewise, we looked to reduce the amount of hardcode. In areas where we needed to hardcode (such as giving buildings their names or assigning each building its cost and improvement costs), we outsourced this data into files and read from the files. This way, if we wanted to change the name of a building or the type of building at a select location on the board, instead of changing the code, we only needed to change the file we were reading from. For our project, we read in files that contain information such as the types of each building, the names of the buildings, and the data on improvements for the buildings. Then, by reading from a file stream, we could fill the appropriate data structure and access that data when needed throughout the program.

Answers to questions:
Question: After reading this subsection, would the Observer Pattern be a good pattern to use when implementing a game-board? Why or why not?

Answer:
Yes because the board will be the subject to the display which is an observer to the board. Whenever the board state updates, it will need to notify the display to change accordingly. Moreover, the board should also be an observer to the player class and improvable class. Whenever the player moves, it should update the board to update their location. From there, we can notify the text display to update the location. Similarly, when an improvable property has improvements or improvements, it should be reflected in the board.

Question: 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?

Answer:

A good pattern to use is the Factory pattern. Both spaces require us to determine what action that will occur to the player based on a predetermined probability table. We will need to have a random number generator in the super-class since it will be used for both subclasses. We can define abstract methods called pickAction() and action(), which we can later implement in our SLC and Needle’s Hall class. Both classes should contain two vectors of Actions, one for the deck that we should draw from and one for the discarded deck of Actions. We just pop the top card from the deck whenever we use pickAction(), then call the action from the Action class. Once the deck is empty, we just swap the deck with the discarded deck and shuffle. Generating the actions can be done with a factory so we can modify our algorithm for creating the Action objects in a separate class.

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

Answer:
No. The decorator pattern is best used to add functionality to an object at runtime, but improvements shouldn’t be additional functionality. Those functionalities should be buying/selling improvements and changing the tuition price and those respective functions would be getTuition(), buyImprovements(), etc. Instead, we should make the Building class extend an Improvable class since those buildings should by default always be Improvable. If we ever wanted to change the max number of improvements from 5 to 10, we can just change the maxNumber of improvements variable in Improvable.

Extra Credit Features:
One extra credit feature we implemented was modeling the SLC and Needles Hall squares as community and chest cards. As mentioned before, we used the factory design pattern to dynamically generate a deck of cards at build time. To do this, we created two vectors of “cards", which were action objects in our implementation. After the two vectors are made, one vector is a full stack of cards, and is loaded with action

objects. As expected, the discard vector is empty initially. Every time a player lands on SLC, an action object is returned. By shuffling the deck of objects and returning the top one to the building, we have effectively modeled the community and chance cards. We chose to make these independent from the SLC and Needles Hall objects since the buildings themselves should not be working with the logic that shuffles the cards. It is better to separate this implementation to increase maintainability and flexibility, should the functionality of the community and chance cards change. When the deck of actions becomes empty, we shuffle the discard deck and swap the two decks. Now, we again have a full loaded actions deck of cards, and an empty discard vector.

Final 2 questions:

Question: 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?

Answer:

To answer this question, the three of us spent some time after finishing the project, to review what we did and discuss what challenges we encountered. The first, and most important, lesson we learned was the significant amount of collaboration needed for such projects to succeed. For most of us, we were relatively new to working on such a large project with others. Although we have experience working on smaller projects (e.g. Hackathons), this experience was definitely new for us. To begin the project, we created the UML as a guide, creating the necessary classes and forming the required relationships. To ensure efficiency, we split the project into 3 main chunks and each spent a couple of days working on our designated part. However, we did not realize how difficult it would be to integrate our work together. Following the design of the UML was a good place to start, but we underestimated how much the design of a program can change throughout its creation. Shortly after attempting to place our parts together, we realized that in many areas, we were not on the same page. Some member functions our own parts required from others turned out different, there were bugs in our own code, and new code needed to be written to accommodate the changes. Very quickly, a majority of our time was spent on assembling our individual components together, while ensuring our initial design invariants are held.

Looking back, it may not have been the best idea to divide such a large project into three separate pieces. Rather, it would have been better to break the project into more, smaller, chunks, and then work on those chunks together. This way, we could make sure we were all on the same page, and everyone's ideas were heard for each section of the assignment. Similarly, we also learned the importance of clear and effective communication. With such a large project, communicating to your partners about what you need from their classes is crucial. Without clearly telling your peers what you need and don't need, can create unnecessary work and incompatible code, which is a frustrating experience for everyone.

Question: What would you have done differently if you had the chance to start over?

Answer:
There are many areas which we would have done differently. To begin, as stated previously, we would have begun working on the project differently. Aside from that change, we would have decided to meet every day in person to do the project. Sometimes, online meetings are difficult to clearly express our ideas, which lead to miscommunication and avoidable bugs. For instance, DC Tims Line was capitalized in some parts of the code and not in other parts, which led to many bugs with our factory. Moreover, combining our code was painful. We did not use a code management tool such as Git since some of the members didn't know how to use it. This leads to many code conflicts amongst different files. A key contributor to that issue was that we worked on separate files from scratch individually, rather than work on the same file together. This meant that just compiling was difficult since we have many errors in many different files. If we worked together on the same file, we could at least try compiling each feature, which is much easier than compiling many features at once.

In addition we should have increased the frequency at which we updated our code on the github repository. Sometimes, a group member would use another member's code that was outdated, causing issues to arise when integrating. If we were to redo this project, we would update our code on the repository on a daily basis, to ensure our group mates were using our updated code for reference. This would have prevented some of the integration errors we had, as well as help create a more efficient team environment.

Finally, if we had the opportunity to redo this, we may have structured the hierarchies of our classes differently and sought more ways to make our code more abstract. For example, instead of reading in the building types from a file, perhaps there are more abstract ways of passing this information to our program. Also, with another opportunity, we would seek to make our user interface more appealing as well as seek the most time efficient solutions to some of the problems we faced.

Conclusion:

Although this was a rather challenging project for the three of us to complete, it served as a monumental learning opportunity for all of us as we were introduced to our first major opportunity to code within a team environment. There were many instances where the project seemed too daunting, however, by devising a schedule, collaborating together in person, and countless hours of debugging, we were able to overcome the hardships associated with this project. We were able to make a final product that was functional, incorporates object design, and something that we are proud of. Overall, we thoroughly enjoyed the collaborative aspect of this project and hope to have further opportunities both in future classes and the workplace to work on another group programming project.