CS246 Assignment 5 - Design Document

Aumio Islam (a38islam)
David Movsisyan (dmovsisy)
Sabina Gorbachev (sgorbach)

Introduction:

The game of Watopoly, is a University of Waterloo branded Monopoly game. The game consists of 2 to 7 players. Players play the game until they reach bankruptcy, at which point they are removed from the game. The game continues until one player remains. A game can be started, saved, loaded in, and quit. Other specifics are outlined within the assignment document.

Overview:

Our implementation is made up of 7 different classes: Game, Board, Dice, Player, State, Square, and Block. This separation of classes allows for each set of challenges we face to be solved through one class, and its methods. As a result, if changes need to be made with one particular part of the program, it can be done through minimal change (changing the class in question, rather than the whole program). Also due to the fact each class is focused on solving its specific set of challenges, the methods within each respective class are closely interconnected. This promotes high cohesion. In order to present the structure of our program, it will be helpful to present the task of each class, and the main methods which carry the tasks out.

To begin, we have ***Game***. ***Game*** is implemented as a class which holds attributes such as: whether or not the game is in testing mode, or whether or not the current game has been loaded in. ***Game*** also holds a pointer (*currentPlayer) of Player type which allows us to easily keep track of the player whose turn it currently is throughout the game. ***Game*** also holds methods to construct, load, save, play, and readPlayers which allows for the making of a game, through loading of a previously saved game or creation of a new game by reading in players piece choice (along with the piece's char that represents the piece on the board). ***Game*** has ownership of a ***Board*** object (composition relationship).

The ***Board*** class is a class which handles the logic of the board, and interfaces between user input, and the states of the squares and players. ***Board*** holds attributes/variables such as a vector of Player object pointers and a vector of square object pointers, as well as the blocks used to represent monopolies. ***Board*** also holds methods which carry out the logic of the board such as roll, next, purchase, trade, improve, mortgage, unmortgage, and bankruptcy. ***Board*** also holds methods which interface the state of the squares such as assets, printPlayers (prints players on square), and printImprovements (prints improvements on a square). ***Board*** owns 8 ***Block***s (composition), ***Board*** owns 40 ***Squares*** (composition).

Subsequently, we have ***Dice***. ***Dice*** is implemented through a class, which stores the values of 2 dice being rolled, and if these values result in a double (which is important in our game as it results in an extra turn/going to DC Tims Line/getting out of DC Tims Line). In order to promote high cohesion, ***Dice*** also holds the methods which allow for 2 dice to be rolled (in and out of the testing environment), and methods which check if a double was rolled. ***Dice*** is allocated as a stack object when needed, and is destructed when it falls out of scope.

Now, we look at our ***Player***. ***Player*** is implemented as a class which holds all relevant information/attributes of the player in question. This information is stored in variables such as, name (player name), piece (the piece the player is playing with), buildingsOwned (the buildings the player owns, stored in a vector to act as a dynamic-length array), gymsOwned and resOwned (stored as a integer), and DC Tims Line relevant variable (checking if the player is currently in the Tims Line, checking if they have any timsCups so they can leave, and checking the number of rounds the player has been in the Tims Line (if number of rounds is 3, the player must perform the steps to exit)). The ***Player*** also holds methods for the getting and setting of the information and attributes discussed above, as well as a player constructor and destructor. Above that ***Player*** also holds methods bankrupt (clears the players assets) print, and assets (which prints the information stored in ***Player*** to the screen at the request of the current player), roll (which uses the allocated ***Dice*** object as described above), and next (changes *currentPlayer, to next player). ***Player*** owns a ***State*** (composition), ***Square*** has a ***Player*** (aggregation).

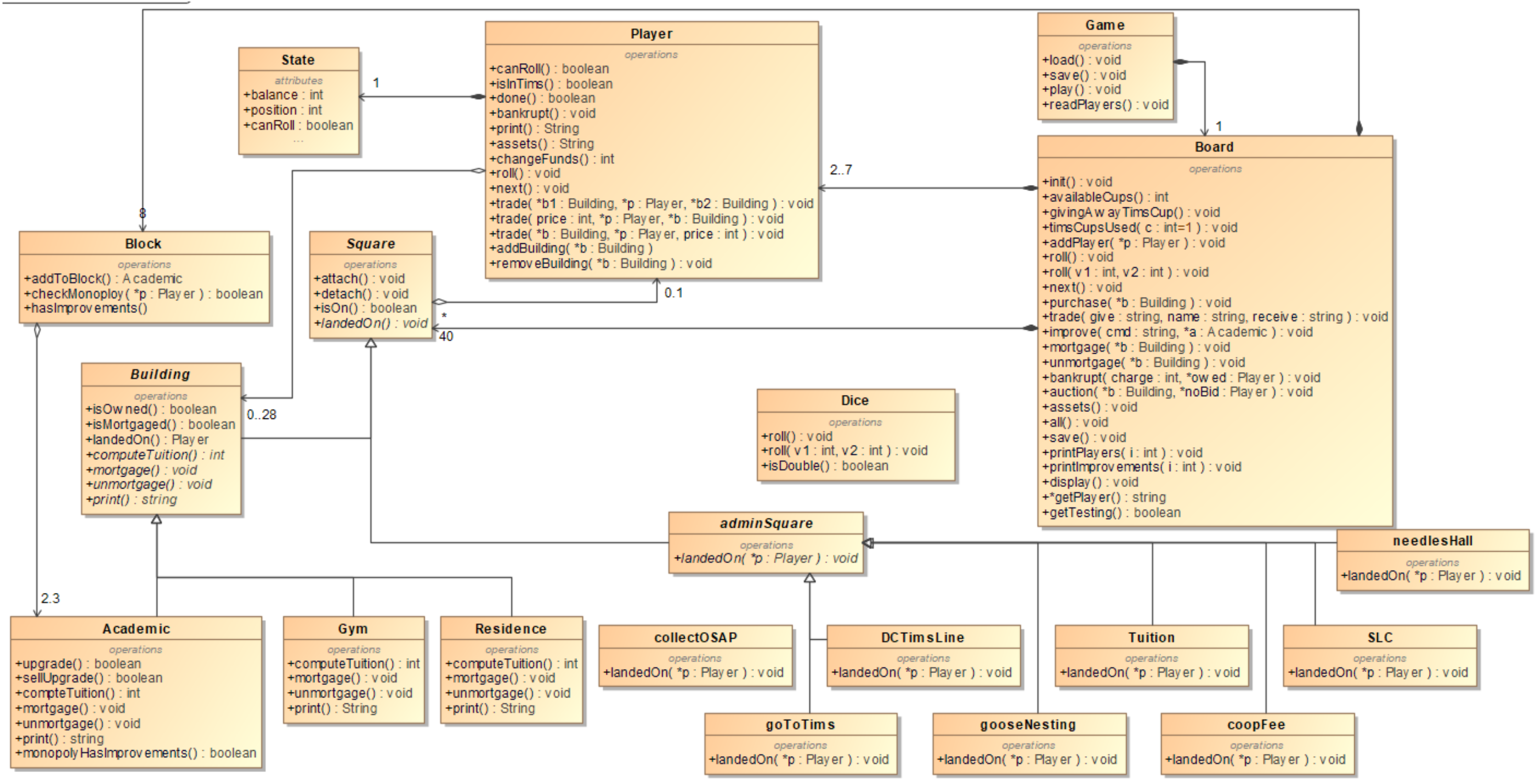
Then, we look at ***State***. ***State*** is implemented as a struct, (or a fully public class) which holds information about *currentPlayer at a specific instance (or at a state). ***State*** holds attributes such as balance, assets, and position, and canRoll (the current players balance, assets, position, and canRoll). ***State*** is owned by ***Player*** (composition).

Following this we have ***Square.*** The ***Square*** is implemented as a class which stores information/attributes about each square. This information is stored in variables like its name, type and playersOnBoard (which stores a squares name, type (academic/residence/gym etc.) and the players which are currently on the board within a vector). ***Square*** holds methods which act as getters and setters for this information/attributes within ***Square***. ***Square*** also has its own constructors and destructors. ***Square*** holds methods attach and detach (acts as attaching/detaching a player to/from a square), and virtual method landedOn. ***Square*** holds subclasses ***Building*** and ***adminSquare*** (which each have their own subclasses) which allows for the differentiation of squares with limited coupling. Also these subclasses hold any relevant methods (ex. mortgaging/remortgaging buildings) that are relevant to said subclass, but not to ***Square*** as a whole. ***Board*** owns a ***Square*** (composition), and ***Square*** has a ***Player*** (aggregation).

Lastly we have ***Block***. ***Block*** is implemented as a class to represent a block of buildings, or a monopoly (ex. Art1 is a block of AL and ML). ***Block*** stores information about each block in a variable name, and a vector of the member of the block in block_members. ***Block*** also holds a block constructor and destructor, as well as methods checkMonoploy (which checks for a Monopoly (letting us know if a player can purchase improvements), and hasImprovements (which checks for improvements). ***Board*** owns a ***Block*** (composition), and ***Block*** has 2-3 ***Academic***s.

This overview of our program represents the functionalities of each class through their attributes and methods, as well as their relationships. Through this breakdown, the general structure of our Watoploy implementation is presented.

Updated UML:



Design:

The final implementation of our Watopoly differs from our original design to a degree, while the overarching concept/approach stays the same. To begin we went away from the observer pattern that was discussed within UML 1.0. Also within UML 1.0 we had a design that had relatively low cohesiveness as multiple classes had overlapping tasks.

In our final implementation instead of using an observer pattern between the ***Board*** and the ***Player***s, we simply moved the composition relationship (which was between ***Game*** and ***Square***) to a composition relationship between ***Board*** and ***Square***. Also we added methods to ***Board*** to make up for the observers design patterns functionality.

In our final implementation each class had its own unique functionality, for example, we allowed ***Board*** to deal with the logic of the game, ***Player*** to interface with players and squares, and the ***Game*** object interfaces with ***Player***. This extra compartmentalization which was added to our final implementation promotes cohesiveness through eliminating multiple functionality for classes, and reduces coupling through allowing one class to work on each task, rather than needing information from many classes to perform a task.

Some challenges that we faced within this project were, efficiently overloading methods which are used by multiple subclasses, type checking, dealing with up and down casting, and allowing subclasses to access the methods of their parent class. We solved these problems through the use of virtual functions, dynamic casting, and inheritance respectively.

First, through the use of virtual functions we were able to overload methods for any number of subclasses. The best example of this within our implementation is the virtual function ***adminSquare*** and its virtual method landedOn() which allows for a separate and possibly different implementation of landedOn() of each of the 8 ***adminSquare*** subclasses.

Next, throughout the project we faced situations which required type checking, also we faced possible problems with down casting. For example if we used static_cast when downcasting from ***Building*** to ***Academic*** static_cast would not compile in this situation. As a result the solution of using dynamic_cast worked well, and is used throughout our implementation.

Finally, we faced the problem of allowing subclasses to access the attributes and methods of their parent classes. A solution to this problem would be greatly efficient as it would limit the repetition of code from subclasses and their parent classes. Within this course we learned of inheritance. This was the perfect solution to the problem we faced. An example of the use of inheritance in our solution is between ***Building*** and ***Academic*** where ***Academic*** inherits computeTuition(), mortgage(), unmortgage() and print ().

Through redesigning our implementation after the first due date, with increasing cohesion and decreasing coupling in mind we were able to produce a more efficient implementation. Also, through using mechanisms learned in class we were able to solve problems which arise in an elegant and concise fashion.

Resilience to Change:

Throughout the planning of our implementation of Watopoly, we kept object-oriented design within our minds. As a result we planned and produced an implementation, which applied solid object-oriented design. To begin, we applied object-oriented concepts such as: Use of Objects/Classes (represented through our 7 main classes, and and their subclasses), the use of information hiding/encapsulation (within our classes), the use of inheritance (used in our subclasses (ex. ***Academic*** inherits from ***Building*** which inherits from ***Square***), the use of interfaces (our .h files), and the use of polymorphism (ex. the function landOn() in ***Square*** is overloaded to allow for the same function/method to perform different tasks depending on which class it is a part of).

Through understanding object-oriented design, it became evident that the goal of these object-oriented practices is in fact two things. To minimise coupling (minimise the degree of dependency between methods and functions, as well as objects and classes) and to maximise cohesion (maximise the degree which elements within a module/function and a class/object relate to each other).

First when thinking about coupling, we realised that if we were to absolutely minimise coupling (0 coupling), our program would in fact be non-efficient. As a result, we allowed for coupling wherever imperative, and negated it elsewhere. For example we allowed for coupling between our ***Player*** and ***State*** classes (where ***Player*** depends on ***State***) as it allowed for an elegant solution of giving us the current players state when needed. Nevertheless through implementing our through 7 classes, there is limited overlap between the functionalities which each class serves. Through this implementation style changes to design, or program functionality would require limited changes in our program. This represents our low coupling.

Next, when thinking about cohesion, we realised to maximise cohesion we had to keep all related code together. This was achieved through the fact that each of our 7 main classes is made to complete one specific task, and as a result all of the attributes and methods of the class are related to the completion of this task. This represents our high cohesion.

To represent our consideration for object oriented design, let us see what would occur if we were to change the rules. For example, let us add a rule where no two pieces can occupy the same square at the same time (excluding collectOSAP and DCTimsLine). To implement this change all we would need to do is add a method to ***Player*** which checks if the square in question is occupied, if yes, it calls roll() again, if no, the move continues as normal.

To further represent our consideration for object oriented design, let us add a new ***adminSquare*** named newSquare. Here to add this newSquare all we need to do is add a subclass of ***adminSquare*** and adjust our ***Board*** class by adjusting the vector which holds the ***Square***s and adjusting our display accordingly. So through changing just two classes, we can completely change the game through adding a new ***adminSquare***. This represents low coupling, as through only dealing with one class we can completely change the rules of the game. We also maintain high cohesion by changing the rule in the relevant class.

The use of automatic dependency management within our makefile automatically updates the source files, streamlining recompilation, and increasing resilience to change.

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?

For the case of the game-board, the Observer Pattern would be an ideal method of implementing the desired functionality. The Observer Pattern is meant to establish a one-to-many dependency between objects such that all observers of a particular subject are notified when the state of the subject changes. In the case of the game-board example, the board observes every player (subject). Every time a player rolls the dice and moves to a new square, their state (position on the board) has changed. Given that each square has an action that is defined for when a player lands on that specific square, the Observer Pattern is a natural fit, as the board is notified of the player's new position, it can call an appropriate function to simulate the play that should occur when a player lands on that new square.

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?

One way to model the SLC and Needles Hall squares, in such a manner that it resembles the Chance and Community Chest cards is to use the Factory Pattern. The Factory Pattern creates an interface for object creation, or it is essentially a virtual constructor. If a State structure is created for each player, and contains variables representing the position and balance of the player, both the SLC and Needles Hall squares simply modify a player's state. Following the probabilities listed in the tables, the Factory Pattern could be used to create an object that modifies the player's state (position or balance).

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

The Decorator Pattern is intended to allow the program to add functionality or features to an object at run-time. In the Improvements example, the Decorator Pattern is a sub-optimal method of implementation. Looking at the table of the tuition required for each building, there is no consistency for the ratio of tuition between each improvement across all the buildings. So, a general improvementOne class, improvementTwo class, and so on, would not work because the multiplication factor would have to be stored within the class. And since the multiplication factors are different for each building, instead of storing the factors, with the same amount of space, the actual tuition values could be stored instead in a vector. Therefore, the Decorator pattern is simply not a good fit for the Improvements feature.

Final 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?

Through working through this project we learned about the difficulties and importance of effective communication, and cooperation. The importance of fully understanding relevant jargon was in full display when working through this project. At the beginning, it was difficult to fully wrap our heads around each other's ideas. As time went on, we got better at understanding each other's ideas quickly. Nevertheless, when it came to the implementation of others ideas, it would often take several revisions or reviews to ensure the implementation matched the idea, and caused no merging problems.
Next we learned that each of us had our own strengths as the project went on. As a result it was efficient to break up the project in a fashion based around each individual's strengths and weaknesses. Through this the importance of sharing knowledge was put on view. By working together our strengths cover a significantly larger area of knowledge. This advantage offsets the extra time that has to be spent on communication and collaboration, and makes rather large and difficult projects manageable.

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

If we had the chance to start over, we would first have a formal conversation at the very beginning of the project discussing each member's capabilities, strengths and weaknesses, so the project can be split up in an efficient manner. We would also spend more time in the brainstorming phase to ensure each member has an equal understanding of the chosen approach. This will eliminate the need to possibly switch responsibilities later on, and allow obtaining a timeline more reasonable. Looking back, our original design (UML 1.0) was not fully complete, and did not fully implement all specifications of Watopoly. As a result, there was no concrete road map, and header files were constantly changing, making it difficult to follow another person's ideas. Overall this stalled our project by a couple days, and would have been counteracted through a UML of a complete implementation to begin with.