CS246 A5 DD2: Watopoly Design Document

Introduction

In general, our group stuck fairly well to the original plan that we had, except for a few changes. Within those changes, the vast majority were additions rather than taking away a part of the pre-planned structure. Some examples of this include extra helper classes for displaying the output, extra public and/or static methods for utility (ex. for gathering or processing input), and other such things that were hard to account for when we were initially designing our project. One notable detraction from our starting plan was our decision not to use the decorator pattern for making improvements to properties, the reasons for which will be discussed later in this document.

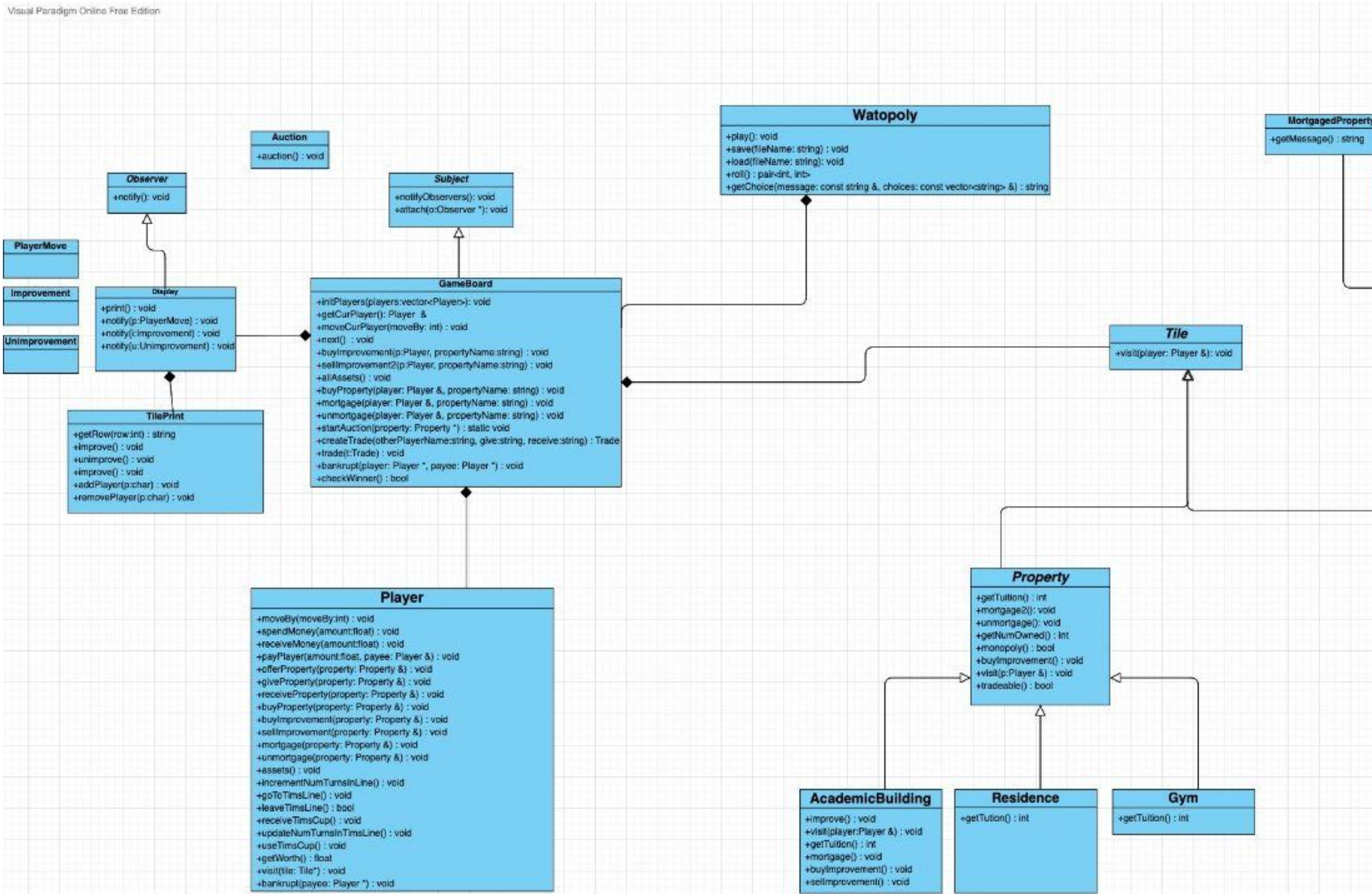
Overview

An overview of the project is as follows: the program begins in main.cc. From there, the command line arguments are read as necessary, and once that's done successfully, a new default Watopoly game is initialized if there was no load file to read, else a Watopoly game using the load file is started. Testing mode is turned on if that was specified. The game functionality is split into these main classes: GameBoard, Player, Tile, and Display. GameBoard handles the logic related to the state of the game. This includes a vector of players, a 2d vector of tiles, and other utility private members. GameBoard uses these to be able to initialize a board, keep track of which player's turn it is, buy and sell improvements, print the assets of the players, buy and mortgage properties, auction properties, and trade properties. Several of these call functions from other classes, such as player in most cases. Player keeps track of the player's name, symbol, location, money, properties, tims cups, and their jailed status. On player, one can spend or receive money, pay another player, list assets, mortgage/unmortgage, deal with the tims line, and visit tiles. Tile is a superclass that has subclasses that handle the different types of tiles a Watopoly game board can have. These include the buildable buildings (one class for all of these), the ones that are buyable but not improvable (special classes for most of these), and the ones that are not buyable and not improvable (each have their own special class). On the buildable tiles, you can mortgage, buy/sell improvements, get the cost for landing on that tile, and visit that tile. On non-buildable and ownable tiles, you can mortgage and visit that tile, and on non-ownable tiles you can visit that tile. In each case, visiting a tile calculates whatever consequence is necessary for the player, and calls the appropriate methods for it. The final main class is display. This explicitly handles outputting the game board. As such, when it is initialized, it creates a display of the starting game state, and it receives messages to update the game state via notify. You can notify the display either of a player move, an improvement, or an unimprovement, and pass it the corresponding data. For each, Display would change the display it stores as necessary. Display then has the print function,

which just displays the contents of whatever Display has stored at the moment. This concludes the high level overview of our project. An updated DD2 UML is included below to show the final structure of the entire program.

Updated UML

A link to an svg of the UML is linked here: A5 DD2 - UML.svg
There is also a picture of our UML here:



Design

The design implementations that will be discussed here include: the design of the error classes, using the visitor pattern for player, using the observer pattern for display, having the extra class TilePrint to aid with outputting the display, the auction class, and why we chose not to use the decorator pattern to implement improvements.

First, the error classes. These classes are designed in such a way that some errors are inheritors of other errors, which all inherit from the main Error parent class (ie. some errors are two generations below, some are only one). This was done to avoid having to print error messages every time something went wrong in other functions, such as a player not having enough money to buy something, a property not being tradeable, a player not being able to mortgage a property, and the myriad other exceptions that needed to be taken into account. We wanted to keep printing to output in only the classes where it was most necessary (in this case, Display), which meant we needed some other way of communicating what went wrong in other classes. This was done via these errors. Whenever you threw one, it would bubble up uncaught until the Watopoly class, which handled each type. There, Watopoly would grab the error message (a function on each error class) and print it out (via Display). Allowing the error to bubble up to Watopoly saves us the trouble of cancelling whatever else was going on at the time of the error as well. This makes it so that no unintended side effects happen, and the player needs to re enter their command. A special note is that some of the error classes that inherit multiple generations down have their getMessage() function in a decorator styled manner. This means that they call their parent's getMessage() function before their own, in order to modularize the error messaging process as much as possible.

The second point is the visitor pattern we used for players visiting tiles. This was done because it made the most sense for our project, and allowed us to code the process of a player landing on a tile really naturally. In the player class, we have a visit() function that receives a tile as a parameter, and we call this function every time that player moves to another tile. This function just calls tile.visit(*this), and passes the player into it. In each tile subclass, we have an overridden visit() function that handles whatever behaviour is supposed to happen there. For example, if we called the visit function on SLC, it would generate the random number and change the player's monetary status accordingly. If we called it on AL, it would charge the player as necessary if someone owned the tile, and if someone didn't it would offer the tile to the player to buy.

The third point of discussion is the observer pattern used for the display class. Display inherits from Observer, and observes GameBoard. Every time GameBoard changes via a player having moved or an improvement or unimprovement being made, the Display class is notified accordingly. The Display class has 3 overloaded functions for each type of notify, which each take in a different type of struct. These include PlayerMove, Improvement, and Unimprovement, which hold the corresponding information for the Display board to make the changes it needs to

(for example, PlayMove contains the old location, the new location, and the symbol of which character moved). When the Display class is notified of a change, it modifies the output that it stores accordingly. Then, when it is time to print, print() on the Display can be called. Using the Observer pattern here allowed us to separate the output functionality from the GameBoard, which again aided in the output modularization goal.

The fourth point of discussion would be the output details for displaying the board. To do this, we created another class called TilePrint. This class was responsible for returning the appropriate strings to print out when we called print() on display. Display held a 2d vector of these TilePrints, in order to mimic the shape of the board itself. Display then goes through the vector top to bottom and prints out each row of each tile (each tile has 7 rows of output). Since there were some specific situations where tiles had to be printed differently from others, boolean flags that indicated so could be passed in at the time of initialization. Each TilePrint also holds a vector of characters (for the player symbols that are on that tile), the name of the tile, and the number of improvements it has. These are all printed out dynamically when it is asked to return the specific row requested of it. Implementing the board output this way allowed us to have the shape of our output state mimic that of the actual board, and saved us from having to use a wall of print statements and careful formatting for the board.

Second to last, we will discuss the auction class. This was implemented to make starting an auction seamless at any point in time. Manually starting an auction in the middle of the GameBoard class was quite clunky, and made it so that we needed to have output in there, which we were trying to avoid. To this end, we moved the functionality over to the Auction class. We could then easily just pass a vector of players participating in the auction as well as the property up for auction, then we could call auction.auction() to begin. This class didn't use any design pattern in particular, but we felt it was a good point to mention for a good example of when we extrapolated functionality into another class.

Finally, we discuss why we did not end up going through with using the decorator pattern to implement the improvement system. As indicated in the design plan for DD1 as well as our UML, we planned to use the decorator pattern for this. However, after we began our project and were implementing the improvement system, we decided that it would be best to go without it for the following reasons: time constraints, its unnatural role in the program, and us needing to store the values for each improvement anyway. Our first thought when doing the improvements was that we would have a vector that stored the tuition of a property at each improvement level, and then increment the improvement level field on each property as necessary. We then thought that using the Decorator pattern might be better. However, we soon saw that using Decorator would add another level of complexity that we didn't have the temporal resources to handle. Using the Decorator pattern would require writing multiple extra classes with their decorations, on top of having to set each decorations field manually when initializing them. This meant that for this project's purposes, there was pretty much no difference between that and having a vector for each improvement level. There is no way to extrapolate the improvement cost based on the level, since they do not go up linearly. As such, one would need to have a table of improvement levels, and manually initialize each decorator with the corresponding level.

This was not much different than just using a single vector of improvements for the properties that needed it. Due to time constraints, and since dropping Decorator didn't lose our program on current functionality, we decided to move forward as such. We recognize that using Decorator would have allowed us to have more extensibility, but that depends on what kind of improvements those would be (ie. can each decorator be initialized independently or not), because if the decorations are like the ones in this project, there isn't much added benefit. Overall, using Decorator here was a lot of overhead for what we decided was too little gain, so we decided not to go through with it.

Before we move onto the next section, we would briefly like to discuss one smaller design idiosyncrasy in our program. In GameBoard, to make it so that we never needed to loop to find the pointer to a program when given its name, we had GameBoard additionally store a map of strings to property pointers, so we could immediately look it up. This was useful because many of our GameBoard functions received the name of a property and then had to access its pointer to pass as a parameter to another function, or to perform some sort of manipulation on the property itself.

Resilience To Change

In this section, we will describe our program's resilience to change with respect to implementing some of the bonus features, as well as other aspects. These include turning the output of the board into a graphics display and creating different board themes, implementing house rules (specifically money in the middle and landing on OSAP), the even build rule, changing the auction rules, and changing the tiles (ie. adding new ones, changing the present values for existing ones). We will also discuss how our program is not very resilient to implementing SLC or Needles Hall in the same way as Chance or Community chest cards.

The first potential change is changing the graphics from standard output to a graphics display and creating different board themes. Our program is quite resistant to both of these changes. The reason for this is that the only modifications one would need to make would be in the display class. The display class handles all of the output for our program, which means that it can be adapted to suit any means of output, not just standard output. This would require one to write a separate display class, and have that observe the GameBoard. Iit would need to have the same notify methods as Display (because that's how the output would be notified that it needs to change), and then would need to handle receiving those messages differently. This would apply to having different themes on our board as well; if it were in the terminal, it would only require you to print things differently in Display, and if it was a graphics display, that would just be a situation specific to that new graphics display class. Since accommodating this change only implies writing one extra class, our program is pretty resilient to it.

The second change to examine is the implementation of the suggested house rules. The first is money in the middle. This would be a pretty simple change to make, since it would only require having another field on GameBoard to keep track of the money pile in the centre, as well as modifying the visit function for the corresponding tile. Our player class already has all of the

methods necessary for paying and receiving money, so beyond those, there wouldn't be much else needed. This also applies if the starting amount in the middle is supposed to be $500 instead of $0. The last house rule is collecting double the money from OSAP. This is an extremely simple change, since we have a class for the OSAP tile. All we would need to do would be to change the visit function there to have the player receive double instead, and no other changes would be necessary. As such our program is resistant to implementing these house rules.

Two more changes that require nothing more than small changes to one class would be implementing the even build rule, and changing how auctions work. As a part of writing the monopoly() function, which needs to determine if a player has all of the properties of a specific type, we had an enumeration list of each type of monopoly triple anyway. This means that we could easily also check to see the number of improvements on each triple and make sure they are built evenly. Due to this, our program is resilient to implementing the even build rule. In addition, changing the auction rules would also be quite simple. Needless to say, this does depend on what exactly the rules were to be, however in general, this only requires changes to the auction class, since that handles all of the logic related to starting and running an auction. As a result, our program is resilient to this change.

The next change to examine is adding or removing the tiles on the monopoly board. In terms of handling the logic related to these tiles, adding or removing some would be very simple, since you would either construct additional tiles (whose classes you would need to write if they were completely new and unique tiles), or not construct as many as there are now. The logic would all be covered, since that is handled by GameBoard and only relies on interacting with the tiles it possesses as private members, which were the ones initially constructed. The one caveat is that in order to accommodate different tiles, one would need to change the Display class. Display is not suited for printing something that's very different from how the original 40-square game board of Watopoly is supposed to look, and it cannot dynamically print the locations of tiles when only being given a single list of them. As such, one would need to write another display class for the board with a different layout. In conclusion, our program is moderately resistant to this change.

The last change to explore is implementing SLC and Needles Hall in the same manner as Community Chest and Chance cards. This is behaviour that's quite different than what we currently have. Solutions to this include having the squares hold a vector of functions, with those functions being the behaviour of that corresponding card. This is an alright solution, but requires you to write out a ton of extra functions inside of SLC and Needles Hall, and is not very clear in its intent. Another potential solution would be to use a secondary visitor pattern on SLC and Needles Hall, which changes what the visit() function does on that square. This could be done be having classes for each type of card, then calling an appropriate .draw() method (analogous to .visit() on the corresponding card.This is a better solution since it is more extensible, and you can extract each function (ie. each different type of card) into its own class. However, since this extra pattern was not already implemented previously, this would require more overhead, and be pretty close in equivalency to starting the creation of those tiles' functionalities from the

beginning. As such, our program is not that resilient to this change, since extending what the current behaviour is into modelling Community Chest and Chance cards would be approximately the same amount of work as just redoing those two tiles.

Assignment Questions

Note: These questions are mostly unchanged from DD1, as our answers to them are relatively the same (except for Q2)

1. Would the Observer Pattern be a good pattern to use when implementing a game-board? How come?

The Observer Pattern would be a good choice for an implementation pattern for the game board due to its extensibility and cohesion. By making a Display class for all of the game board output and having that observe the game board, we are able to separate the game board from how it communicates the state of the game to the player. This allows for two things: the game board to focus only on game logic and manipulating the various states of the board, and for the display class to manage all of the printing. This means that to change the game logic, one would only need to change the game board class (and potentially other classes such as player, depending on which modifications you're making), and to change how the state of the game is outputted to the player, one would only need to change the Display class. Additionally, using the observer pattern allows one to be able to add more observers if necessary. One example where this could be useful would be extending this project to use a GUI. If that were the case, all we would need to do would be to add one more observer for the GUI, and have that handle how the game state is displayed, and we wouldn't need to change anything about the other classes. As such, using the observer pattern is a good choice for handling the job of outputting the contents of the game board.

2. Is the Decorator Pattern a good pattern to use when implementing improvements? How come?

We believe that the Decorator pattern has its pros and cons for this project specification. In general, the Decorator pattern would be a good choice when implementing improvements because of how flexible you can make it further down the line. Since the decorator pattern employs a linked list of classes, at each improvement stage, you could easily change the behaviour of the improvement to be essentially whatever you want by only modifying the class that improvement pertains to. Furthermore, adding more improvements of a different type is simple, as you would only need to write what that improvement does in its own class. This is all great behaviour for the Decorator pattern and works well if you were to build this project for extensibility. However, given that this project has time constraints, and that Decorator adds more overhead and complexity that we potentially don't have time to deal with in time for the deadline, the Decorator pattern for us is only a considerable idea, not a great one. The required behaviour can be achieved simply by using a vector of integers, which, while not allowing for

much functionality outside of using those, makes the tiles much more simple and less error prone. In conclusion, while the Decorator pattern is in general a good choice for this type of problem, we believe it to be much more work than necessary for this project specification, despite the extensibility benefits.

3. Suppose 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?

If we wanted to have the behaviour of SLC and Needles Hall more closely model Chance and Community cards, we could use the visitor pattern. This would be implemented in the following manner: when the board is initialized, randomly generate a list of chance and community cards (these would represent the deck that one could possibly choose from), using whichever random number scheme determined by the game rules. When a player lands on a corresponding square, call p.draw(), which would then call card.drew() on the appropriate card type. Since this uses the visitor pattern, whenever we want to add a new card, we need to add a method on player as well as card, which is inherent to how the visitor pattern works. Using this method, one could define as many different cards as they wanted, and would only need to create a class for that card, and an overloaded draw() method in the player that would call drew() on that card class. Overall, the visitor pattern does not have minimal coupling, since the player and card classes would need to be coupled out of necessity here, but it is a solid solution to use out of the different design patterns we have discussed in class, due to its extensibility in this scenario, and other design patterns being less applicable here.

Extra Credit Features

Our group was somewhat pressed for time to complete this project, so the only extra feature we attempted to implement was the “no new and delete” bonus that was available for every project specification. We planned to complete this right from the get-go, and so we started off our project using smart pointers to initialize anything that required memory management. There are other pointers throughout the program (ex. as private members of classes and whatnot), however those are all expressing non-ownership relationships, and are needed to call functions or gain access to values related to that pointer, not to delete it. Due to this, our program doesn't have any new and delete statements, and (as of writing this document), hopefully doesn’t leak. Given that we have not allocated anything except for the initialization of the smart pointers, and we never explicitly delete anything, this should not be a problem.

Final Questions

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

The biggest thing that this project taught us about developing software in teams was how important the planning stage was. If we were not forced to create a UML of our whole project

right at the beginning and make a detailed timeline for who would write which part of the project at each step of its process, I can be reasonably confident in saying that we'd be pretty disorganized. In our time doing the assignments alone, we underestimated the value of simply knowing everything that has been written in the program (since we wrote the content of each assignment ourselves). In a team, you need to have a good sense that your teammates will implement their functions in certain ways, so what you write doesn't conflict. To this end, having that planning stage at the start and maintaining open communication is key. Secondly, we realized that another facet of why the planning stage was important was simply for unifying everyone's ideas on what the structure of the program would look like. If we didn't do this, it would be hard to write much code without receiving constant checks from the rest of the group, since everyone would continually need to be on the same page. This is time needlessly spent reviewing that could instead be spent actually writing the code for the project. Overall, this project taught us that working on groups is much different than doing an assignment on your own; creating a solid plan to start off is key, and will save people much more time down the line.

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

If we were given a chance to start over, we definitely would have allocated more time to each part of the project. When the project specifications first came out, everyone thought that implementing the base game would take much less time, and immediately started thinking of implementing the bonus features. I think we all underestimated the quantity of work that was given just to write the base game. The game itself doesn't have much logic that's very complicated, but to design and implement it well, it actually took considerable time. If we started over, I don't think the structure of our program would change very much, but instead we would have a more rigorous schedule that would plan for the completion of the entire project even earlier than what we initially planned. Doing this would allow us to have a grace period for the completion of the base game, and hopefully let us implement more of the bonus features, which was our second biggest regret. Since we were strapped for time, we only were able to implement the one bonus feature that we started the project with the intent of doing: the non-explicit memory management bonus. Looking at the bonus questions, we are reasonably confident that they could be implemented with not much extra time spent on the project, the only problem being that we didn't have that much time this first run around. In conclusion, the two things that we would make sure to do differently given another run at this project would be to create a faster-paced schedule to stick to initially, and to implement more of the bonus features aside from the non-explicit memory management one.