<u>CS246 A5 Chess Design Document</u>

By: Nicholas Lee, Oditha Amarasinghe, and Gianpaulo Pineda

1.1 Overview

In our project, we have various abstract classes that represent the different key components of Chess. The **Board** class holds the actual board which is a 2-D vector of **Tiles**. It also holds the **Pieces** and **Players** of the game. The **Board** class runs and resets the game. It allows players to take turns, keeps track of the score, and allows **Player**s to make **Move**s. The **TextDisplay** class is what repeatedly prints out the text-based chess board to standard output.

The **Piece** class is an abstract class and holds general information about chess pieces such as the colour (“white” or “black”) and the **Tile** that the **Piece** is on. Each **Piece** also has a tracker which helps distinguish **Piece**s and a value which is based on the type of **Piece**. With every **Piece**, there is a map of validMoves which the **Player** can use on their turn.

The **Player** class is an abstract class which has two derived classes called **Human** and **AI**. The **AI** class is also an abstract class, which has one derived class called **Level1**. A chess game must be played with two players, which can be any combination of **Humans** or **AI** (with the **AI** being **Level1**). Each player type has their own characteristics and commands that they can input into the Chess game. For example, **Human** players must manually input their commands, and **Level1** (**AI**) players have their **Move**s randomly made for them.

The **AI** class represents a computer player, and derived classes associated with levels of difficulty as an opponent. So far, we only have implemented the **Level1** derived class which is meant to only make random valid **Move**s (without prioritizing any specific one). To make **Level1** create random **Move**s, we got a reference to the game board and stored it in *curBoard*. From there, **Level1** waits until it receives the “move” input command to start working. It creates a vector called *myPieces* which stores all of **Level1**’s owned pieces on the board, as it goes through each tile, checking if the piece belongs to and has any valid **Move**s to make. From there, we generate a random number between 0 and the size of *myPieces* - 1, to be used to index *myPieces* and select a random piece out of *myPieces*. That specific piece becomes the *chosenPiece*. A random move is selected by *chosenPiece->getRandomMove()* and is executed by the method. The *createMove()* function in **Level1** returns the move info, later to be used in the construction of a new **Move** object, used to track **Move**s that have occurred and document and important information regarding those **Move**s.

The **Move** class holds various information regarding a move that is made on the board. It knows useful information like the **Tile** where the move began, the **Tile** where it ended, the **Piece** that used to be on the destination **Tile**, and the Piece that currently stands on the destination **Tile**. Additionally, it can print all the information that was previously listed to standard output.

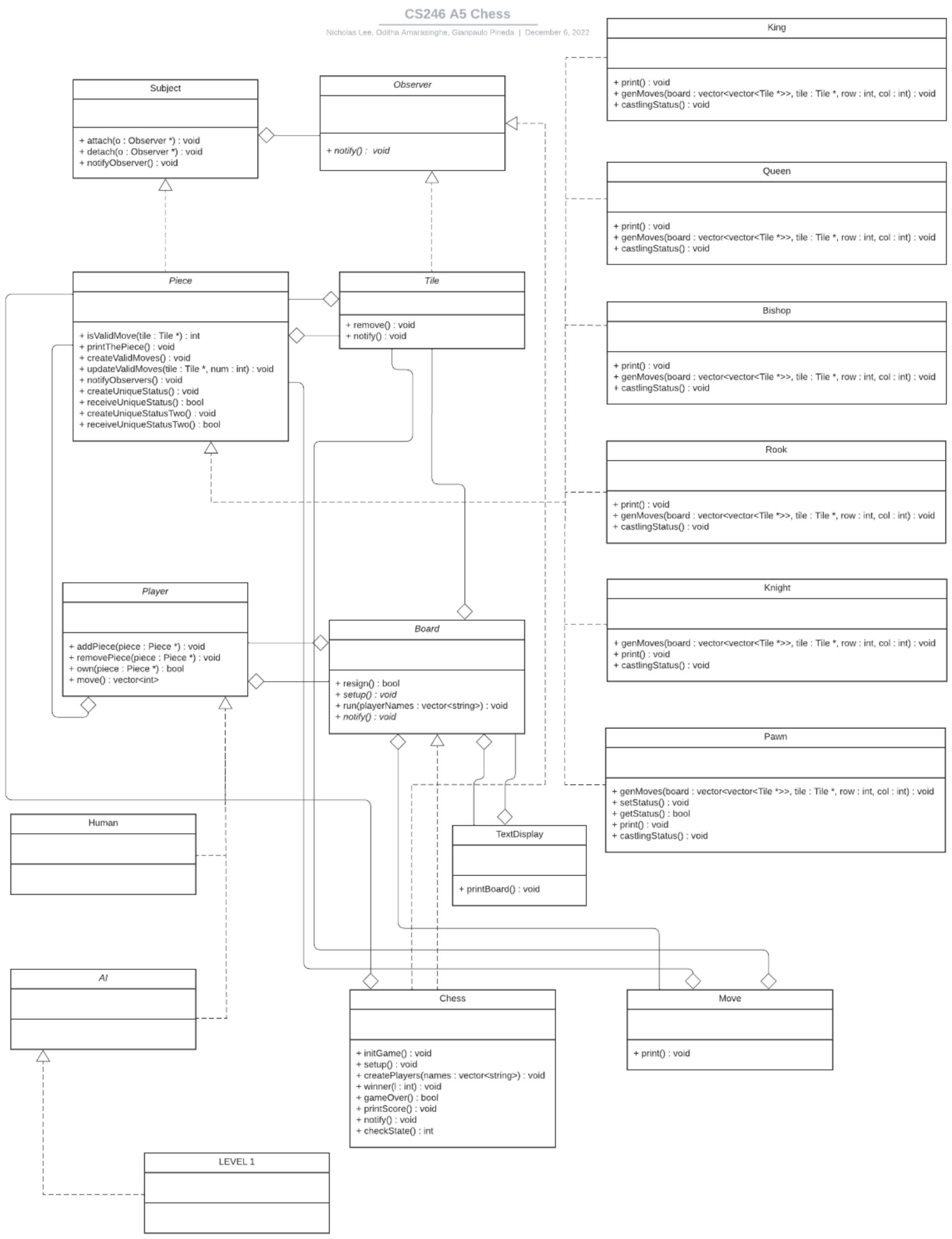
The **Chess** class is a concrete class that is derived from the **Board** class and focuses on the logic of the game of Chess. The functionality that is implemented includes *checkmate*, *check* and *stalemate*. Regarding subsidiary mechanisms, the class supports pawn promotion and castling. The class includes an *initGame()* function which sets up an 8-by-8 board and places pieces in the correct order. It also creates two players, one for each side.

An **Observer** and **Subject** class was also implemented following the methodologies taught in class. These classes work hand in hand and are abstract interfaces to implement the Observer/Subject pattern.

The **Tile** class represents each square the player can move on the **Board**. It holds space for a single **Piece** and is derived from the abstract **Observer** class, making it the concrete Observer. Regarding functionality, all **Tile**s on the board have a specific position that can be identified by a row and column; denoted using x and y coordinates. A piece may be able to attack or move to a Tile on the next turn which occurs when a tile is under "threat" by the pieces.

1.2 Updated UML

This updated UML has some changes compared to the original UML. In the original UML, the **Board** class was handling both the game logic and printing out the **Board**. As stated in the section below, we wanted to implement the MVC design pattern so we decided to separate the responsibilities of logic and displaying the board. In order to accomplish this we created a **Chess** class to handle the logic and a **TextDisplay** class to print out the **Board**. Due to time constraint issues, a Level 2 class was not created as shown in the original UML. We also created a **Move** class in order to hold information about how a move is made.


CS246 A5 Chess
Nicholas Lee, Oditha Amarasinghe, Gianpaulo Pineda | December 6, 2022
Subject
+ attach(o : Observer *) : void
+ detach(o : Observer *) : void
+ notifyObserver() : void
Observer
+ notify() : void
King
+ print() : void
+ genMoves(board : vector<vector<Tile *>>, tile : Tile *, row : int, col : int) : void
+ castlingStatus() : void
Queen
+ print() : void
+ genMoves(board : vector<vector<Tile *>>, tile : Tile *, row : int, col : int) : void
+ castlingStatus() : void
Piece
+ isValidMove(tile : Tile *) : int
+ printThePiece() : void
+ createValidMoves() : void
+ updateValidMoves(tile : Tile *, num : int) : void
+ notifyObservers() : void
+ createUniqueStatus() : void
+ receiveUniqueStatus() : bool
+ createUniqueStatusTwo() : void
+ receiveUniqueStatusTwo() : bool
Tile
+ remove() : void
+ notify() : void
Bishop
+ print() : void
+ genMoves(board : vector<vector<Tile *>>, tile : Tile *, row : int, col : int) : void
+ castlingStatus() : void
Rook
+ print() : void
+ genMoves(board : vector<vector<Tile *>>, tile : Tile *, row : int, col : int) : void
+ castlingStatus() : void
Player
+ addPiece(piece : Piece *) : void
+ removePiece(piece : Piece *) : void
+ own(piece : Piece *) : bool
+ move() : vector<int>
Board
+ resign() : bool
+ setup() : void
+ run(playerNames : vector<string>) : void
+ notify() : void
Knight
+ genMoves(board : vector<vector<Tile *>>, tile : Tile *, row : int, col : int) : void
+ print() : void
+ castlingStatus() : void
Pawn
+ genMoves(board : vector<vector<Tile *>>, tile : Tile *, row : int, col : int) : void
+ setStatus() : void
+ getStatus() : bool
+ print() : void
+ castlingStatus() : void
Human
TextDisplay
+ printBoard() : void
AI
Chess
+ initGame() : void
+ setup() : void
+ createPlayers(names : vector<string>) : void
+ winner(l : int) : void
+ gameOver() : bool
+ printScore() : void
+ notify() : void
+ checkState() : int
Move
+ print() : void
LEVEL 1

1.3 Design

In our project, we made use of the Observer/Subject pattern. This pattern was used to implement the dependencies between classes that were critical to the game of chess. More specifically, the **Tile** class and the **Piece** class act as our concrete observer and subject respectively. Note that **Tile**s must know which **Piece**s are posing a hazard to it. To do this, **Tile** contains a mapping of a *string* to an *int* (called *numThreats)* that indicates how many of that colour (black or white) are posing a threat to the **Tile**. Since the attacks will change frequently as the game continues, this map is updated during each move. **Tile** observes **Piece** (the subject) as part of the Observer pattern. Each **Tile** on the board runs its own *notify()* function and updates its map *numThreats* by checking every **Piece** and testing whether itself is a valid move to which the piece can move to in the next turn. This allows the class **Chess** to check if the **King Piece** has been checked or checkmated. If the **Tile** the King is on is being threatened by the opposite colour, the **King** is in check. If after subsequent **Move**s the status is still in check, we see if the **King**'s *validMove* mapping is empty, in which the **King** would be in checkmate and the game would end.

It is important to see that **Piece** also points to the **Tile** it is on. This is needed to track the valid **Move**s a **Piece** can make based on its current location. **Piece**s are either black or white and the players can only move their colour and capture the opposite colour. Each type of **Piece** also has its own *value* which is based on actual chess gameplay. The **King** has a value of 10, **Queen** has a value of 9, **Rook** has a value of 5, **Bishop** has a value of 4, **Knight** has a value of 3 and the **Pawn** has a value of 1. This functionality will be useful in higher levels of gameplay with an **AI** since the program can capture and prioritize **Pieces** based on this value. Each **Piece** having a value was not sufficient to uniquely identify each piece so a *tracker* field was added for each **Piece**. This is an integer value that is unique to each **Piece** generated and is required when tracking **Move**s in a game.

Each **Piece** also has a *notMoved* field that is a boolean which is useful for **Piece**s like the **Pawn**, **Rook**, and **King**. Every **Piece** also has a map of valid **Move**s (*validMoves*) that tells which **Tile**s that the **Piece** can move onto. *validMoves* maps a **Tile** pointer to an integer and this integer conveys what type of move is possible. If it is a 1 then the move is to an empty cell, a 2 means a capture move, a 3 means a piece with the same colour, 4 is a unique move such as castling, and -1 is a check move. After every turn, each **Piece** calls the *createValidMoves* function which regenerates the map. Each **Piece** generates its valid **Move**s based on the rules of how that particular **Piece** moves. For example, the Bishop can only move in a diagonal direction. When printing out the **Piece**s, a lowercase letter represents a black **Piece** and an uppercase letter represents a white **Piece**. The **Piece** class has a *printThePiece* function that calls a private virtual function called *print* which is overridden by each type of particular piece.

Our project also utilizes the Model View Controller (MVC) design pattern. In our original UML, our **Board** class was responsible for both the game logic and communicating the results. In order to apply MVC, we separated these responsibilities so now we have a **Chess** class that is responsible for the game logic such as check/checkmate, **Pawn** promotion, castling, and the state of the game. The **Chess** class acts as the Model, the **Board** class acts as the Controller, and the **TextDisplay** class acts as the View. The **Board** class delegates the task of outputting the actual chess **Board** to the **TextDisplay** class. The Non-Virtual Idiom was also utilized in our project. This can be seen in the **Piece** class where all public methods are non-virtual and all virtual methods are private except for the destructor. NVI allows for customizable behvaiour such as when each type of piece generates their own valid **Move**s.

Chess Class Nuances

In the **Chess** class, there are implementations of various functionalities that make Chess interesting. These include **Pawn** promotion, and castling. Starting with **Pawn** promotion, a fitting *pawnPromote()* function was created that does exactly that. If either **Pawn** (white or black) makes it to the back row of the opposite colour, the **Chess** class will prompt the user to select a **Piece** that the **Pawn** will “promote” to. A move is also created in this instance, with the *previousPiece* being the **Pawn**, *currentPiece* being the promoted **Piece**, and the *initialTile* and *destinationTile* being the current **Tile**. As for castling, a similar *Castle()* function was created. Note that the **Piece** class (and all its derived pieces), have a property called *notMoved.* This is important for castling because when this is false for the **Rook** and **King**, we check if the cells in between the two **Piece**s are empty and are not threatened by an opposite **Piece**. If so, the **King** moves two **Tile**s towards the **Rook** and the **Rook** moves to the other side of the **King**. For the Castling and PawnPromotion methods, the virtual methods *castlingStatus* and *getCastlingStatus* are overridden by each **Piece**, along with *promoteStatus* and *getPromoteStatus*. Only the **King** and **Pawn** classes truly use these functions, whereas all other **Piece**s simply call them. Inside the **Piece** class, *createUniqueStatusTwo()* calls *castlingStatus()* and *receiveUniqueStatusTwo()* returns *getCastlingStatus()* which changes the boolean *castle* status set as a parameter in **King**. Additionally, *createUniqueStatus()* calls *promoteStatus()* and *receiveUniqueStatus()* returns *getPromoteStatus()* which changes the boolean *isPromoted* status set as a parameter in **Pawn**. **Chess** checks for the *receiveUniqueStatus()*'s return in order to initiate *Castle()* inside the *notify()* function as per the Observer/Subject design pattern, and this is similar to *pawnPromote().*

1.4 Resilience to Change

We created a program which accommodates change through the use of modularization, which is the process of splitting a program into many smaller, manageable modules. This is highly evident as each class within our program has two modules, one to hold its interface (.h files) and another to hold its implementation files (.cc files). Having our program split up in this manner means that when changes need to be made to the program, all we need to do is target and recompile the relevant modules, preventing accidental edits from being made to irrelevant components.

As mentioned in the design section, our project utilizes the MVC design pattern as well, which further improves our ability to accommodate change. If we wanted to change how the **Board** looks, fortunately, this will not interfere with the logic of the game. The MVC design pattern would also allow for a graphical user interface to be more easily implemented as well since it would just be another type of view component in the MVC.

Also, to give our program more resilience to change, we made sure to use effective module design through designing modules which help us achieve low coupling and high cohesion.

Our program achieves low coupling because our modules only communicate with each other through function calls. However, our program only reaches the second-lowest level of coupling because of inheritance and the fact that some methods like the *moveCreate(*) function in the **Player** class, or the *getBoardRef()* function in the **Board** class, return vectors instead of simpler types. The benefits of low coupling are that changes to a single module will have little to no effect on other modules, and it makes it easy to reuse modules when needed.

Our program also showcases high cohesion because we specifically organized each module to only hold components like methods or fields that were relevant to itself. Specifically, our program achieves the second-highest level of cohesion as there are some instances where vectors are passed as parameters (this can be seen within the *genMoves()* function in the **Piece** class). Also, our code is split into modules which each have their own unique purpose. Examples of this are: the class/module **TextDisplay**, which completely handles the printing of the board, and the class/module **Move**, which stores information about **Piece** movements. We further increase cohesion by having readable code and effective commenting/communication throughout the program (example: **Human.cc**, **Level1**.cc) which makes it easier for unfamiliar users to understand the program's functionality and make changes where they understand they can be made.

The final version of our program heavily leverages encapsulation as well by making every single classes' fields private and only allowing access via functions/methods. This aids resilience to change as it keeps modifiers from touching the private fields, and forces them to handle information in a more secure manner (through functions that return values).

In regards to how our program will use these aspects for implementing changes provided by the program specification:

- For a book of standard openings, all the changes that are necessary occur within modules related to the **Chess** class, which prevents any unnecessary changes being made to other classes.
- The addition of an "undo move" ability would only require the modification to the **Piece** class (the addition of vector *pastMoves*), which is easy to understand and locate due to our program's high cohesiveness and low coupling.
- Adding the ability for the game to be played by four players, we would need to alter the specific modules related to **TextDisplay**, **Chess**, and **Board**. However, we do not need to make changes to any other modules which is why modularization, coupling, and cohesion are very helpful.
- Changing piece behavior can be achieved by modifying the methods within each class of the **Piece**s you want to modify, modularization allows for the changes to only affect those pieces and forces those specific pieces to be the only ones that are recompiled.
- Adding new **AI** player difficulties can be achieved by creating derived classes of the **AI** abstract class. This would only require the addition of an interface file and implementation file, and a little bit of changes to **Chess** (the *createPlayers()* function).

- We could modify chess's rules anytime we want because we are using a MVC design pattern, and since modifying rules relates to its logic, the only parts we need to touch are the **Controllers** (**Chess**, **Board**)

1.5 Answers to Questions

1. **Question:** Chess programs usually come with a book of standard opening move sequences, which list accepted opening **Move**s and responses to opponents' **Move**s, for the first dozen or so **Move**s of the game. Although you are not required to support this, discuss how you would implement a book of standard openings if required.
    - **Move**s would ideally be inputted using the Alphanumeric system (i.e A14).
    - Fundamentally, this implementation would require many conditionals to check if certain **Pieces** have moved to certain **Tiles** and then dictate whether there exists a counter move within a list.
    - In C++, we would use the <map> library which would allow us to hold key value pairs of two types. This would allow us to hold a move and a counter move within the same index where calling a move (a key) would return its counter move (the value).
    - All this functionality will be within the **Chess** class and we would simply call the move function on the counter move.

2. **Question:** How would you implement a feature that would allow a player to undo their last move? What about an unlimited number of undos?
    - To give players the ability to undo their last move, we plan to let **Pieces** track their past **Move**s by holding a vector of **Move** objects called *pastMoves*.
    - Everytime a **Move** is made, a new **Move** object is *emplaced_back()* to the end of pastMoves.
    - Using the **Move** class we implemented, we can use the fact that it holds relevant information like pointers to previous and surviving pieces, and initial location and destination tiles.
    - The information provided by the **Move** class could be used to undo the move, by recreating the old **Piece** within the location it was deleted, and moving the current piece back to its old position using the **Tile** information.
    - Afterwards, the last **Move** in *pastMoves* is popped.

- Since the pastMoves vector will keep track of all past **Move**s, completing an arbitrary amount of undos will just be undoing each move at the back of the pastMoves vector and popping it off, repeated any amount of times to reach “x” amount of undos.

3. **Question:** Variations on chess abound. For example, four-handed chess is a variant that is played by four players (search for it!). Outline the changes that would be necessary to make your program into a four-handed chess game.
   - In order for our program to handle a four-handed chess game, we would first need to allow the user to set up this game variant. Then we would have to create an additional two **Players** along with the additional **Tiles** and **Pieces**. The user should also have the option to play in teams of two or individually. In determining when a game is won, new rules need to be defined such as when a team vs an individual wins.
   - To go into more detail:
     - **i.** **Chess** class
       1. Inside the function that initializes the game (*initGame()*), we would have to set up a 14x14 **Board** on which a “playable” 8x8 grid represents where **Player**s can move their **Piece**s.
       2. The *printBoard()* function would thus need to be revamped to support this size of **Board**
       3. Since we would now need to support two more colours, the *notify()* function would need to be changed to check for threats and validMoves for each respective colour.
       4. The logic for checkmate and check would also have to be modified

1.6 Final Questions

1. **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?
   a. There are many lessons that this project taught us about developing software in teams. We learned how to use git and version control practices such as committing with meaningful messages. We learned of the importance of creating an UML and a plan document that allowed us to map out the project and create reasonable deadlines for ourselves. We also learned the importance of

communication which was vital since many of the components that we worked on intertwined with each other. Working in a team allowed us to rely on each other which was very useful in debugging our code. Overall, developing software in a team was a unique experience that allowed us to gain real world experience.

2. **Question:** What would you have done differently if you had the chance to start over?
    a. If we had the chance to start over, we would have been more careful forming the design document because we made some careless mistakes which could've been avoided. In the planning phase, we would have also tried and brainstormed together how to best implement certain aspects of the game, because most of the implementations were split up and done independently. Having everyone informed as to how things were implemented would've generally made debugging easier on everyone. In the coding phase, we would probably try to be more careful when coding because we witnessed a surprising amount of careless mistakes resulting in errors that ruined our program's functionality. Additionally, we would've made an effort to secure study rooms earlier because we didn't realize that study rooms are usually booked by students a week ahead of the actual time.