Requirements Analysis

The client needs an application which can read/write a .txt database of students.

The application should have functions to Add, Remove, Show All Students, and Find Students.

Students details should be:

* First Name
* Last Name
* Email
* Campus
  + Selection from Auckland, Wellington, or Christchurch

And all students should have an ID which is autogenerated from the first 3 letter of the first + last name, and the year.

Removing a student should be done by using the Student’s ID.

The database should be displayed and sorted by a few options:

* By ID
* By First Name
* By Last Name
* By Campus

And these should all be displayed in an ascending order.

An individual record can be searched for by:

* First Name
* Last Name
* Student ID

The terminal should display records in a way which is easy to read, and there should be a simple UI.

When the program is exited, the updated database should rewrite the .txt file to update it.

# My Two Proposed Solutions

## Solution 1: Quick Sort and Binary Search

## Solution 2: Quick Sort and Linear Search

This is my choosen solution

# Data Structure

My data has been stored in a python List.

I tossed up between Binary Tree(BST), Linked List or a python List and I decided on a python List for a few reasons:

1. Accessing an item in a List is O(1) – a constant time meaning that no matter the size of the list Accessing an item ill take the same amount of time. This is better than BST (O(log(n)) and significantly better than a Linked List (singly- or doubly-) O(n)
2. Appending an item in a List is also O(1), better compared Insertion to BST at O(log(n)) and a Linked List at O(n). As I don’t intend to have an order to my data I am able to append items to the end of the List and now have to insert at specific points.
3. Python Lists have, like Linked Lists and BST, the ability to grow and shrink as data is added or removed. It also is able to store different types of data.

Ultimately, for my purposes a python List is the best solution as it is faster at accessing and appending items than my other options. The benefits of a Linked List and BST which don’t normally come with an array do come with a python List data type and therefore a List is clearly the best option.

Table 1: Data Structure Comparison

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Algorithm | Average Time Complexity | | | Space Complexity |
|  | Access | Insertion | Deletion |  |
| Python List | O(1) | O(1) (for append) | O(n) | O(n) |
| Binary Tree | O(log(n)) | O(log(n)) | O(log(n)) | O(n) |
| Linked List | O(n) | O(1) | O(1) | O(n) |

# Sorting Algorithm

## Why Quick Sort over Bubble Sort?

Bubble Sort has a complexity of O(n^2). As the value of n increases, the speed of Bubble Sort quadratically increases with the value of n. This means that the bigger ‘n’ (the data) is, the slower bubble sort becomes, and it becomes slow rapidly. As more students join the university, the system would continue to get significantly slower at a fast pace. You can see from Table 1 below that Bubble Sort’s average time is the same as their worst case – which is not good.

Bubble sort can work well for small data sets, but with 500 records and potentially more in the future, bubble sort is not appropriate.

Quick Sort has a complexity of O(n log n). Quick Sort partitions the data into smaller sets and is significantly faster than a Bubble sort.

Quick Sort can perform poorly if the data is already sorted, decreasing to a quadratic time complexity, but the data will not be sorted so this will not be an issue. It is also larger than Bubble Sort but the space is worth the efficiency of the algorithm.

The worst-case quicksort is O(n^2) – the same as Bubble Sort. So there is not a case where Quick Sort is worse than Bubble Sort even in the worst scenario.

Table 2: Comparing Quicksort to Bubblesort

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Algorithm | Time | | | Space |
|  | Best | Average | Worst |  |
| Quick Sort | O(n log n) | O(n log n) | O(n^2) | O(log n) |
| Bubble Sort | O(n) | O(n^2) | O(n^2) | O(1) |

# Searching Algorithms

## Why Linear Search over Binary Search?

Binary Search only works when the data is sorted due to the nature of the search. A binary search works by examining the midpoint to see if the target is the same as, bigger, or smaller than the midpoint, then depending on bigger or smaller it recursively calls itself with a smaller array of data until the midpoint is equal to the target. You can see in Figure 3 I have run a binary search and it has given me the incorrect result due to an unordered algorithm.

If I were to choose to use Binary Search, I would have to re-sort my data by my target (so if I want to find a record by its fname value, I would have to re-order the array by fname). This would mean each time I called a Search I would also be doing a Sort.

As you can see from my graph this would take longer than a Linear Sort. This makes Linear Sort the better option of the two presented algorithms based on my selected datatypes, for this application.

Table 3: Search Algorithms

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Algorithm | Time | | | Space |
|  | Best | Average | Worst |  |
| Linear Search | O(1) | O(n) | O(n) | O(log(n)) |
| Binary Search (Recursive) | O(1) | O(log n) | O(log n) | O(1) |
| Unsorted List Binary Search  (Recursive, using Quicksort) | O(n log n + 1) | O((n log n) + log n) | O(log n + n^2) | O(log n + 1) |

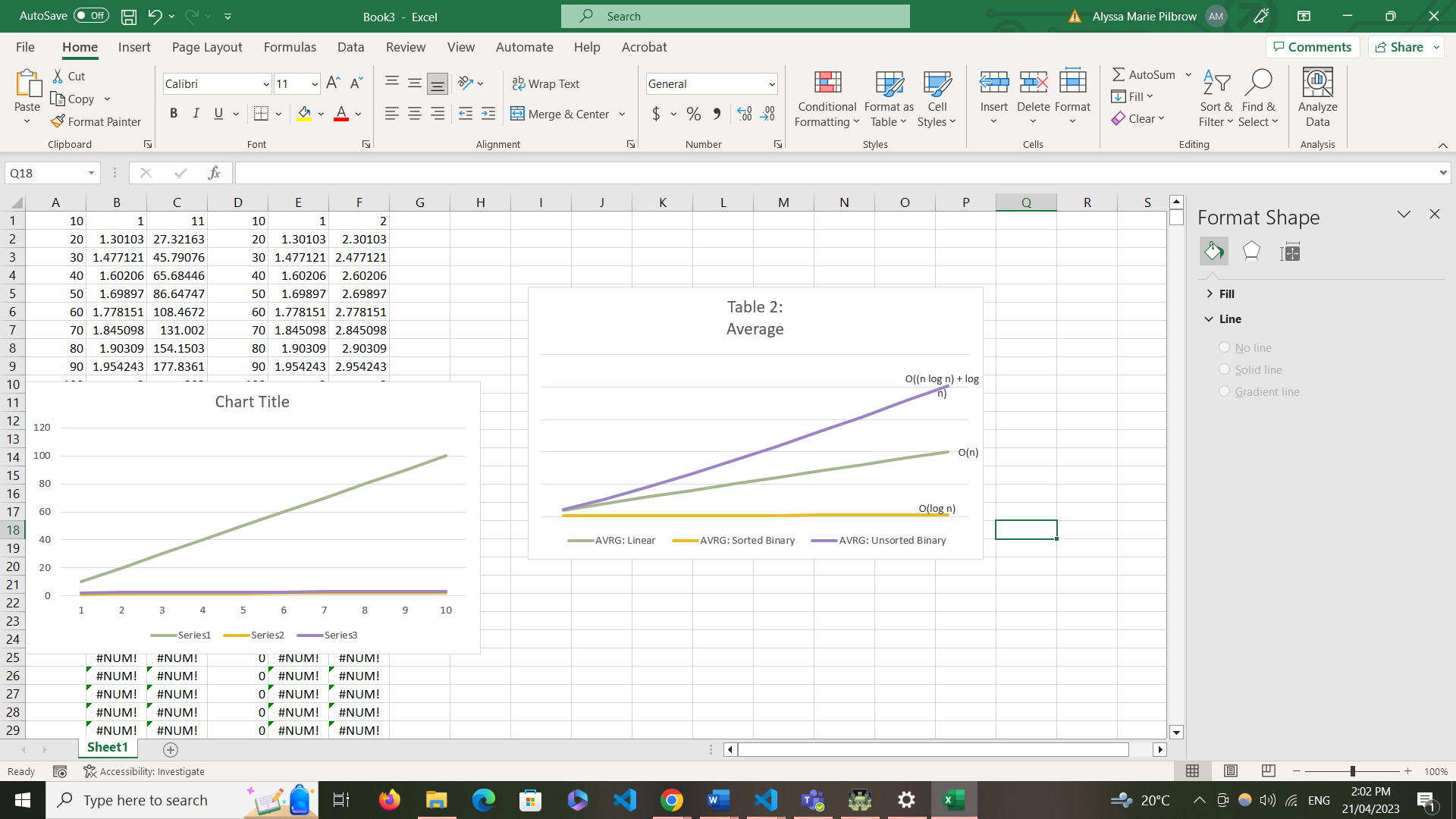


Figure 1: Graph showing comparisons of the Average from Table 3.

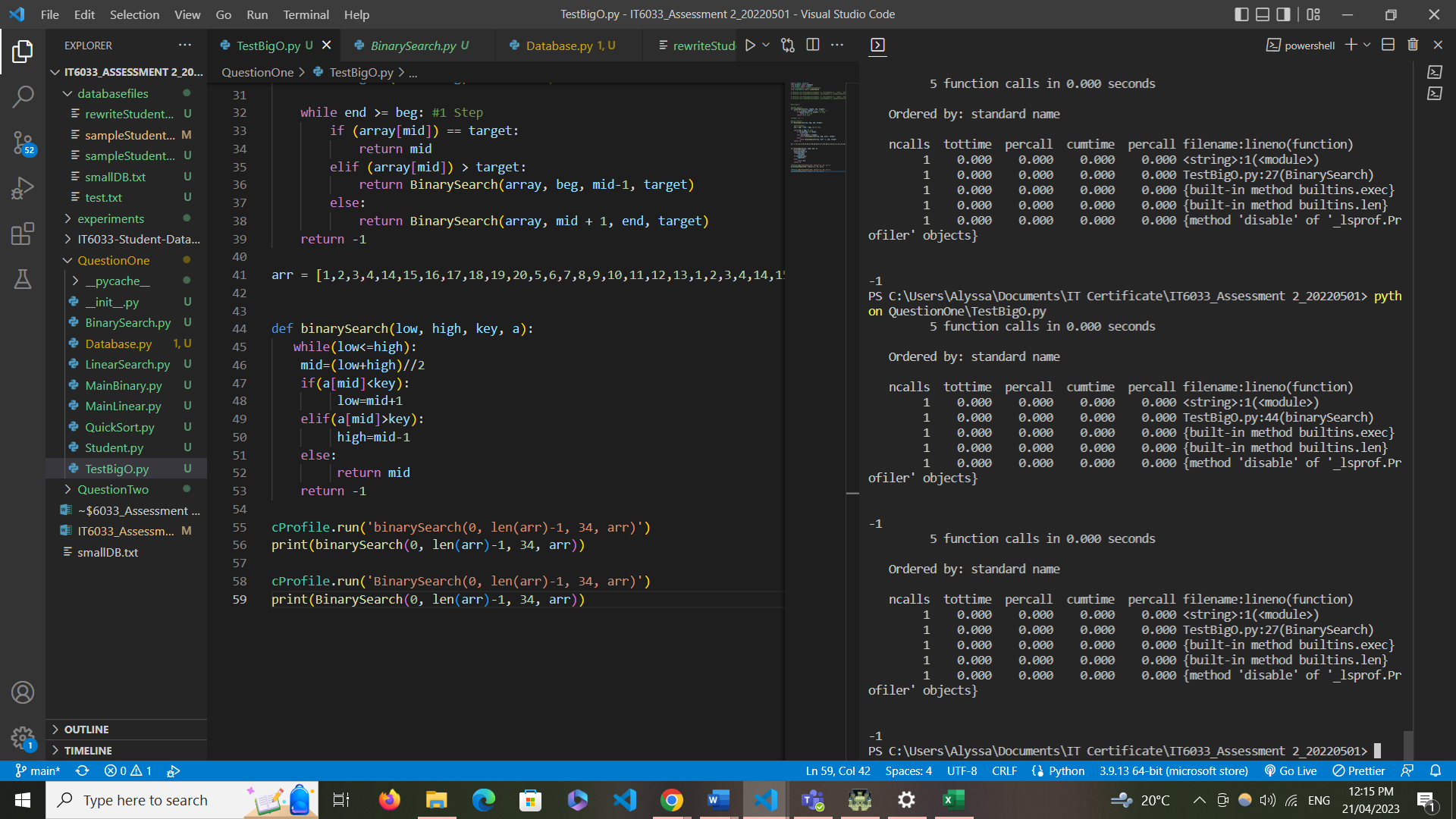


Figure 2: My Binary Search returning a negative index, indicating the item does not exist in the array. The item does exist, but the array is unsorted

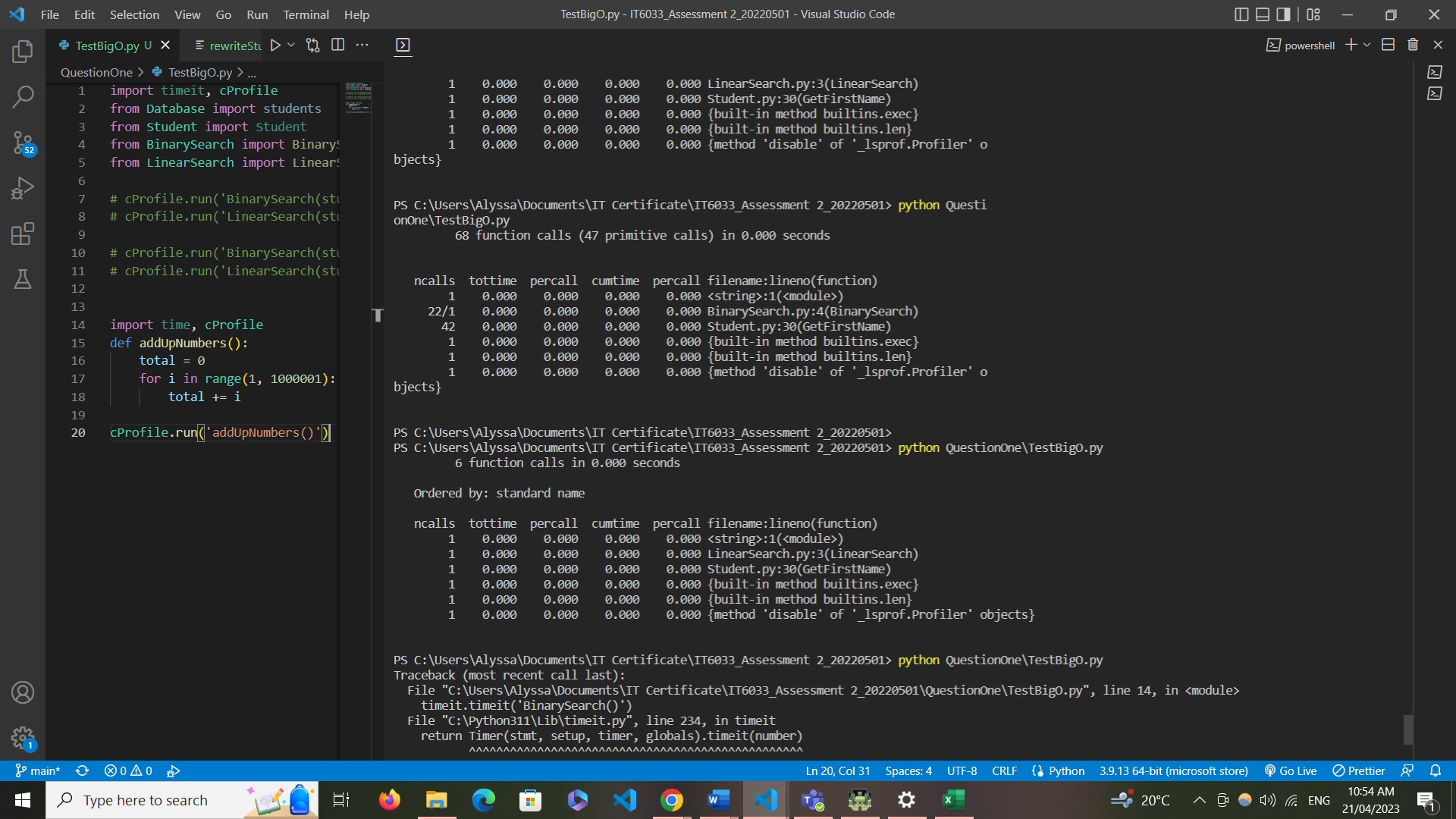


Figure 3: Linear Search

# Software Design Pattern

The Observer software design pattern should be used when:

* Objects are dependent on another
* A change in one object will mean changes are needed in other objects, but you don’t know how many.

An observer pattern works well when we need to notify all stakeholders (In my case the Student, Teacher and Course objects) when a state of an object has been changed.

### Use Cases in my Application

Subject: Courses

Observers: Teacher, Student

If there is an update for a Course Object, such as a course being deleted, Teacher and Student objects containing that specific instance of Course will need to update to reflect this.

Subject: Courses

Observers: Teacher

When a new student is enrolled in a course, the Teacher object will be notified. To extend to a real usage, an email or other sort of notification containing the update could be sent to the details contained in the Teacher instance which was affected by a new enrolment.

For the student database, the client would like to introduce the functionalities for:

* Teacher is notified when student is enrolled
* Teacher Announcements will go to all enrolled students

We use the observer pattern when we want to be able to inform/notify all stakeholders (an object or a group of objects) when the state of an object changes. An important feature of the observer pattern is that the number of subscribers/observers, as well as who the subscribers are, may vary and can be changed at runtime