Kian Parnis (0107601L)

CPS 2004 Assignment Documentation

Repository URL:

https://github.com/WyvernCore/OOP

Last Commit hash:

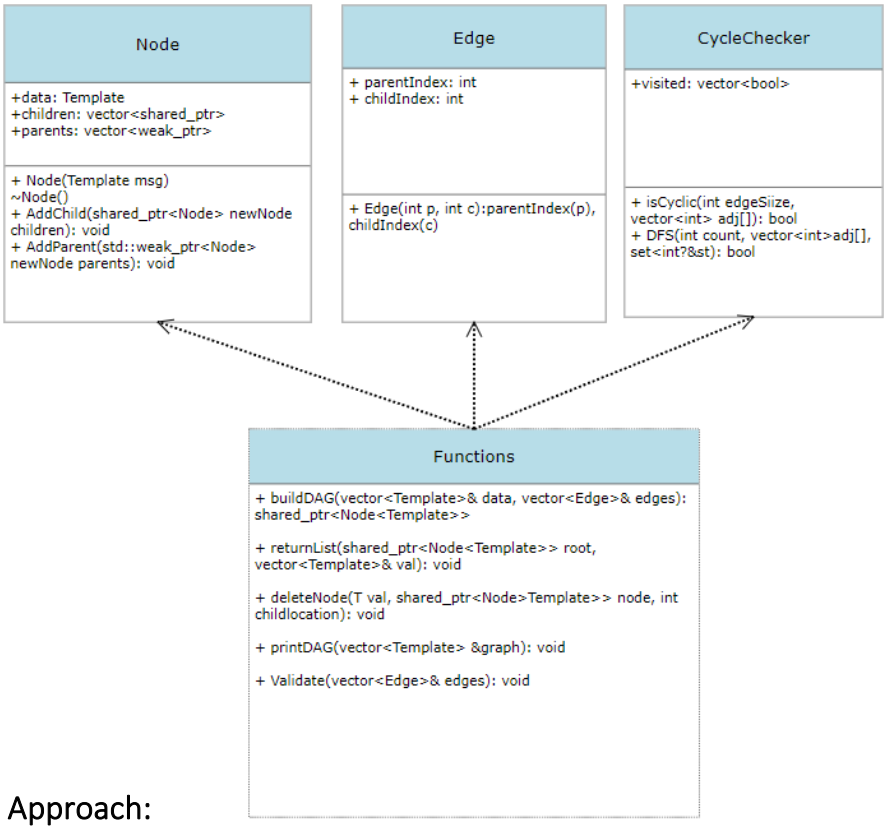
75c32c42ad3a8082966658e9bd81d7f48f01afa9

Contents

Kian Parnis (0107601L)

Task 1: DAG

UML Diagram:



Design and Approach:

The Directed Acyclic graph was designed with several factors in mind, firstly the user can decide what data they would like to store and decide on the type, so vectors and templates were used to store the dataset. After the user could be able to choose the links between the data and this was implemented using an **edge** class that stores the parent and child within the vector.

Before the initialization of the graph, validation checks where implemented to check if 1) The link connection was valid (stays within the boundaries) and 2) if a cycle was present. To check for cycles, an adjacency list was constructed, and a depth first search was used to see if a cycle was present. If the validation failed, the appropriate error message came up and the program was exited. If the validation succeeded the data and edges are passed to the **buildDAG** function which populates the node class (the DAG itself) to construct the graph and lets the graph **own** the individual objects.

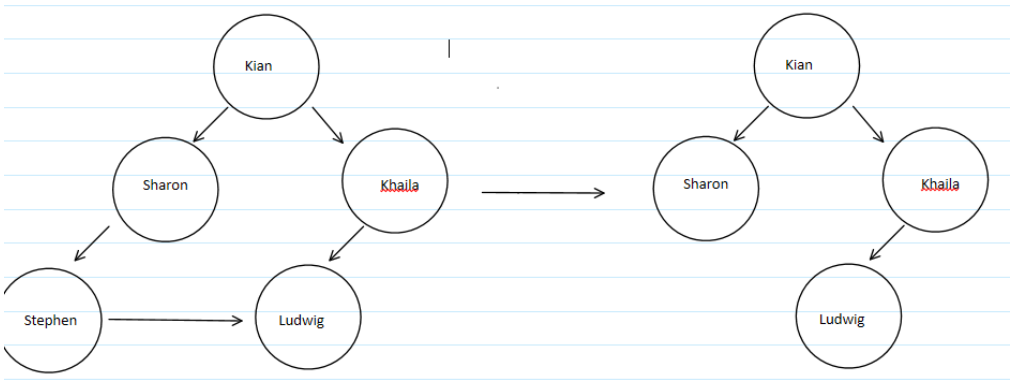
This was done with the help of shared pointers to create the links between the nodes and additionally, weak pointers where also set from the parents back to the children. Once this was done a couple of features where added such as a method **returnList** which returns the list of edges back to the user and stores them in a vector list. The user also has the option to print the graph in the order of breath first search (this is visualized in the testing section). Finally, the user can pass the value of a parent node and the location of the child is relative to the parent, so the child can be deleted. Since the graph is constructed using shared

pointers any subsequent nodes will also be removed preventing any memory links from occurring.

Test Cases Considered

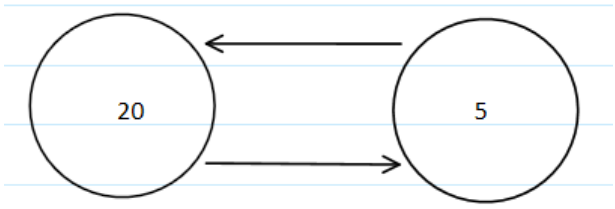
Below are three test cases which are present within the code, kindly comment out/in any one you would like to test.

Testing Case 1: (Deletion of Stephen node)



Calling TestCase1() successfully constructs the above DAG and then removes the Stephen node. Return list successfully returns DFS: Kian, Sharon, Stephen, Ludwig, Khaila, Ludwig and Kian, Sharon, Khaila, Ludwig respectively.

Testing Case 2: Cyclic Presents



Calling TestCase2 attempts to generate the above cycle but exits successfully due to a cyclic presence.

Testing Case 3: Invalid Link

The user attempts to link node 11.2 with a node which isn't present in the list and an error is successfully returned upon doing so.

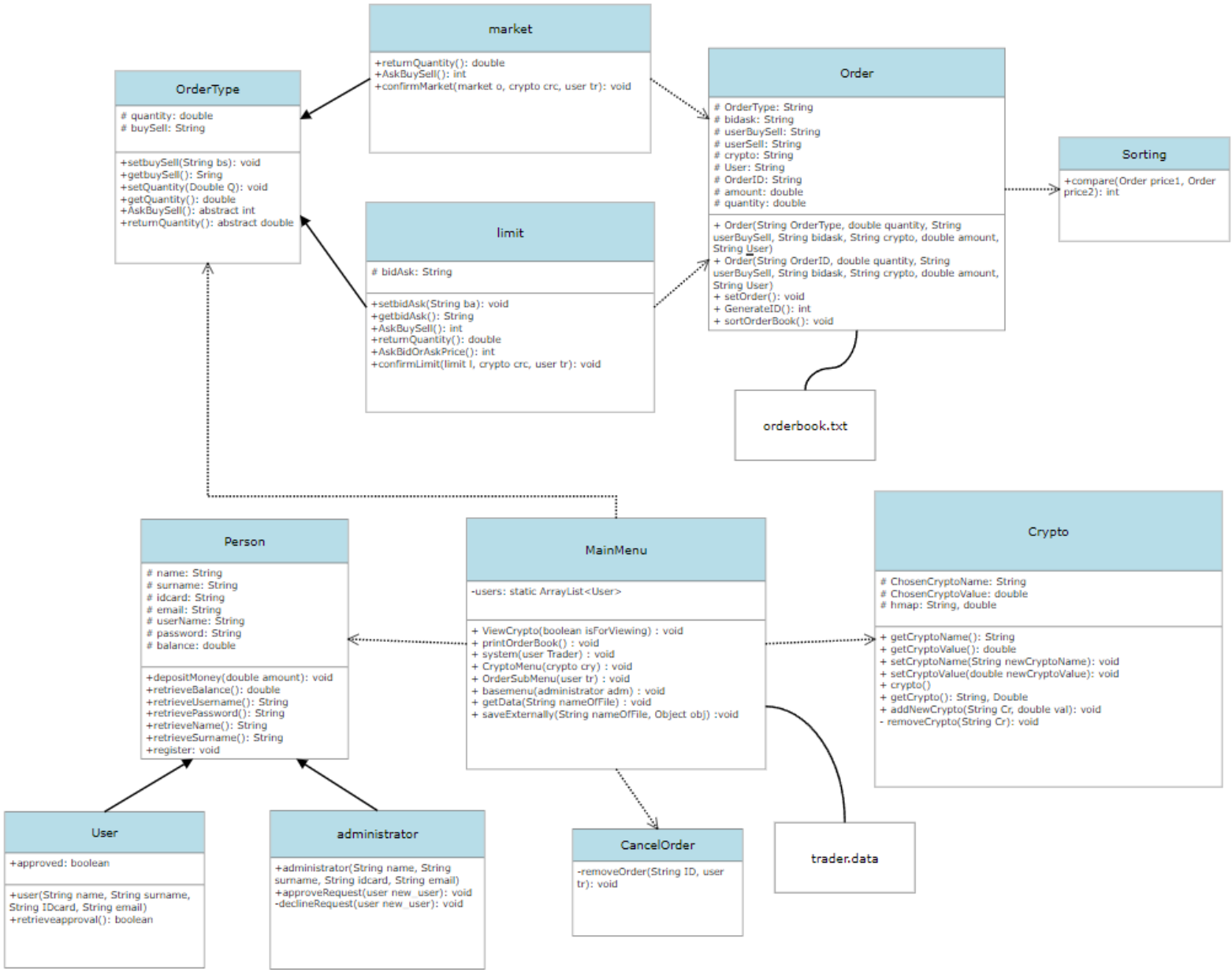
Limitations:

The limitation present is with regards to the deletion, although it works as intended further optimizations would allow the user to choose directly the node they want to delete, and it is deleted without considering the parent and the child location.

Kian Parnis (0107601L)

Task 2: Crypto Exchange

UML Diagram: *representation of the implementation of the crypto exchange platform*



Kian Parnis (0107601L)

Design and Approach:

For the implementation of the Crypto Exchange, classes where designed and modelled with a very simple CLI that was used as a means of testing the different interactions and functionality between each class. Detailed tests will be provided in the testing section showing what is expected to occur and what actually occurs for different aspects of the system. The following is a detailed explanation of how the classes where implemented: Class Person was implemented as the top-level hierarchical structure between class **User** and **Administrator**. The Person class contains types which are common between both sub-level classes such as {name, surname, idcard etc} as well as the respective getters for each type.

A function '*register()'* has been added that upon creating an instance of Person a process occurs which allows new registries to set a username and password and each instance's full details is then saved to an external file *'trader.data'* by the system which allows for multiple users to be stored and are then able to log back into the system. The **administrator** class has the functionality of **approving** the user to access the system and this Boolean value is stored within the **User** class. A **Crypto** class was implemented which uses a hashmap to store each crypto in a key, value pair where the key is the name of the Crypto Currency, and the value is the price for the whole crypto. For added functionality an **approved** user is allowed to view the available crypto currencies as well as choose later down the line, when making an order which Crypto they would like to buy/sell as well as the **quantity** of Crypto wanted.

To facilitate future expansion for the system additional functions were added to the Crypto class which would let an Administrator add additional crypto to the baseline four crypto currencies and another functionality is present which would permit the ability of changing the price of each individual crypto currency due to the volatility of crypto's currencies ever changing price.

Before the OrderBook was constructed the two types of Orders where considered, **Market** order and **Limit** order and for these two types a hierarchy was established with the top-level being **OrderType.** This class temporarily stores the data which is common between both order types which is then added to the **Order Book,** that being the **quantity** and whether the user wants to either **buy** an order or **sell** an order. **Limit** and **Market** differentiate in the fact that Limit order contains the additional temporary data storage for whether the user would like to **bid** or have an **ask** price.

After the user chooses the respective ordertype, a confirmation is asked from the user before creating the order to prevent any undesired mistakes from occuring. The data is then passed to the **Order** class that saves each order externally to an *orderbook* file which keeps track of:

1) A unique id which is generated for each order
2) The order type {Market / Limit}
3) The value of the crypto
4) Whether it's a buy or sell order

5) For Limit order whether its a bid or ask price (null is set for a market order)
6) The total *price*
7) The username of the user which made the order

The orderbook is maintained by keeping each order sorted by *price* which would be used by a matching system, and this is done with the aid of the **Sorting** class. Finally, a way for the user to cancel the order was implemented which lets the user select by id the order they want **cancelled** and if the user and id match the order is then removed from the orderbook.

Limitations:

Currently the system does not have the functionality of the matching engine present. The groundworks of this feature have been implemented {i.e having the orderbook being sorted} and given enough time this additional feature can also be added to the system. Apart from this due to the system being simple in nature, features such as that of security have been omitted and only a basic storage was implemented to store user information, this can be changed to be securely stored in a database instead. Apart from this several details where omitted such as detailed functionality for maintaining an individual's balance and having it save/change depending on whether the user sells or cancels an order.

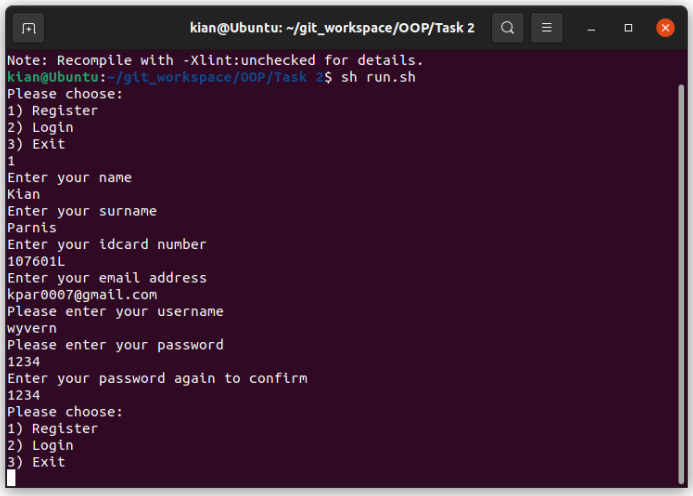
Audit Trails:

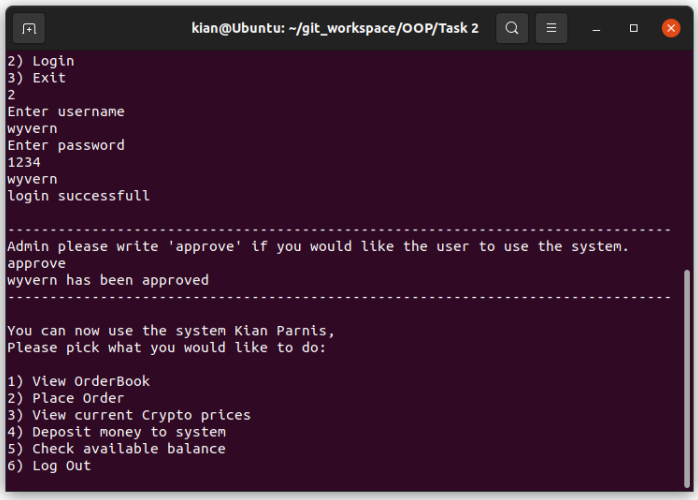
With the current implementation of the system a list of each order is stored externally to an orderbook. A log can easily be added by having each order be saved both to the normal orderbook as well a log file which maintains the activity of all users. This file would have to be encrypted and not allow deletion of records in order to not get tampered with. This solution would not need to modify any classes just the additional saving to a log file thus no modification is present.

Test Cases Considered

Below testing was done on various parts of the system for what the user intends to do vs what happens by the system.

<u>Test one</u>:

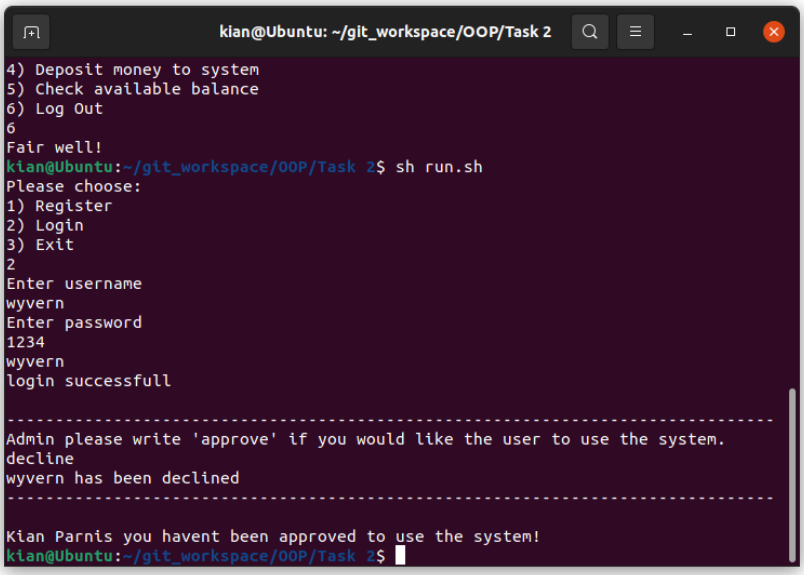




Expected: User successfully registered, logs in and is approved by an admin.

Actual: Output matches what has been expected successfully.

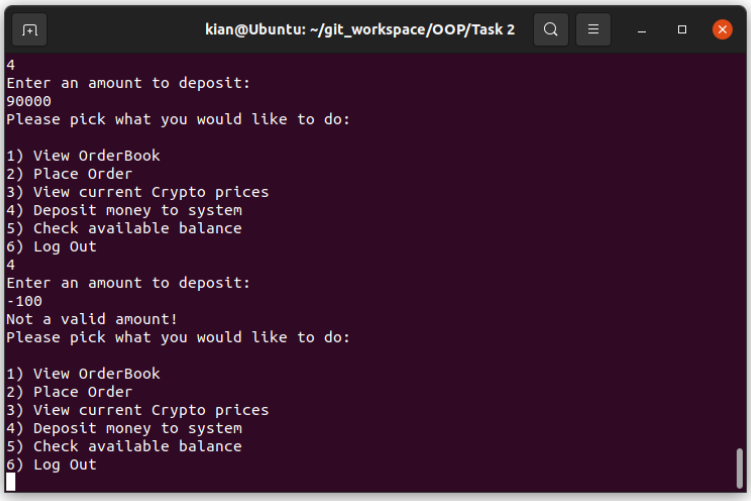
Test two:

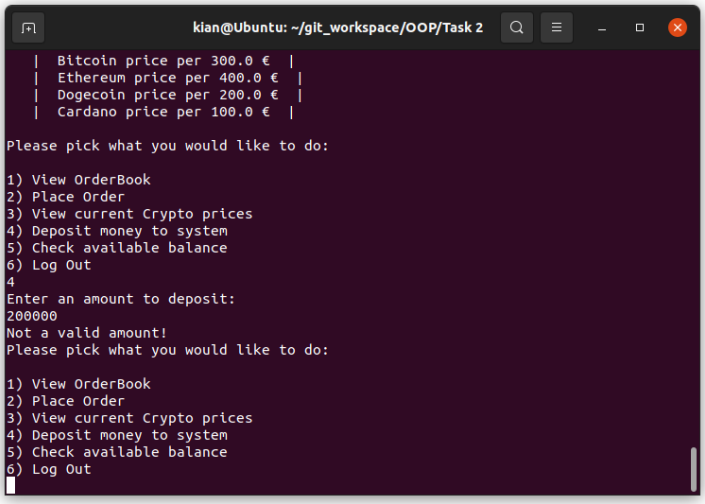


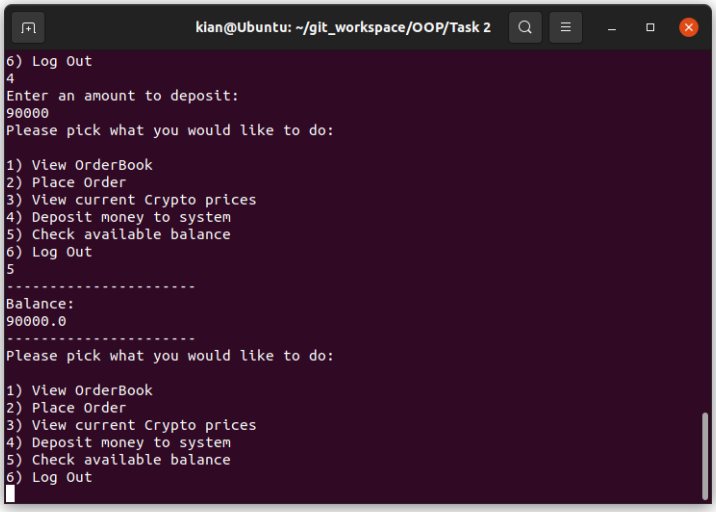
Expected: User successfully logs in, but is declined by the admin.

Actual: Output matches what has been expected successfully.

Third test:



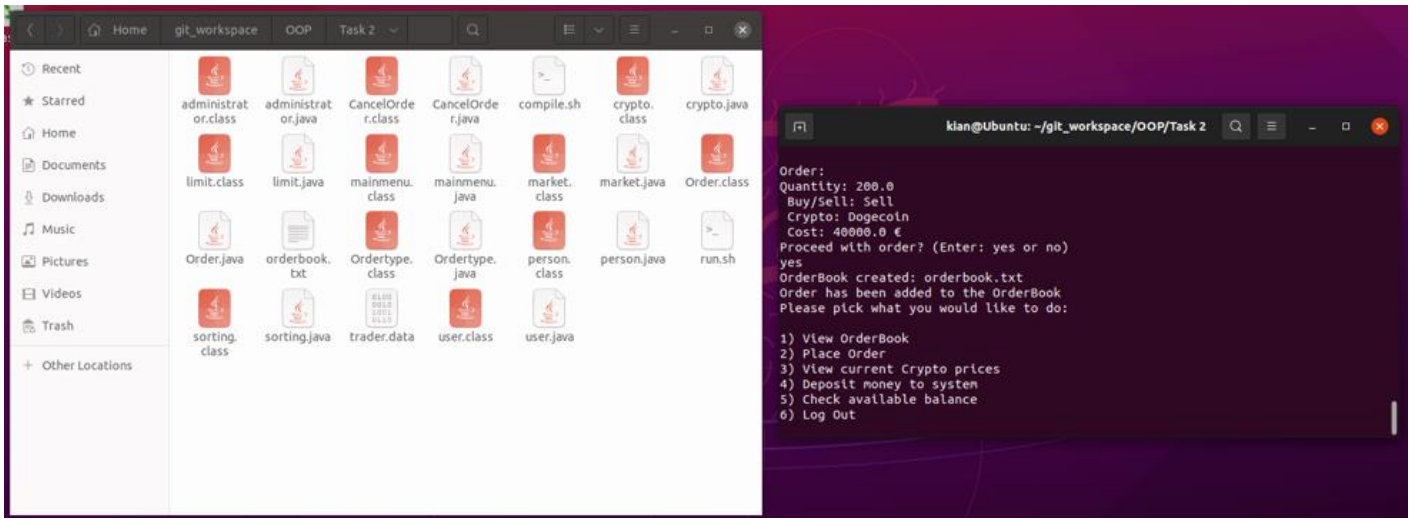




Expected: User attempts to deposit various amounts and only the value in range is accepted and inputted to the system.

Actual: 90k is successfully deposited by the user in the system.

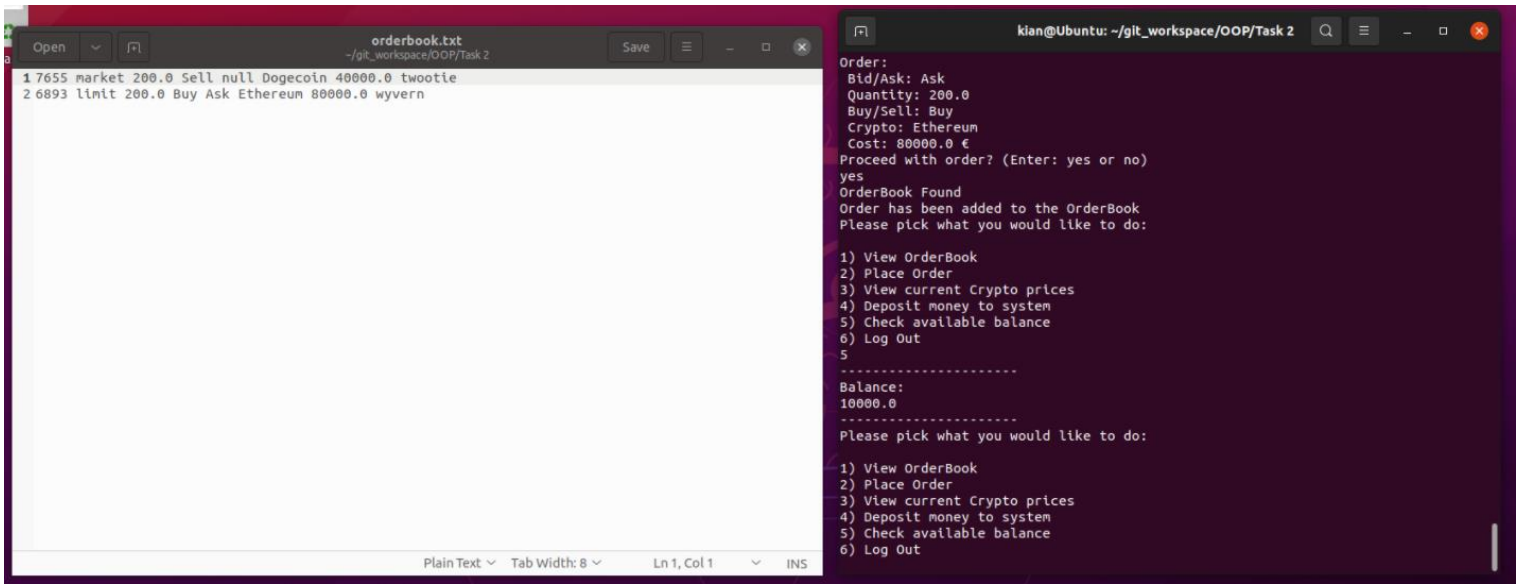
Fourth test:



Expected: User attempts place an order, since no orderbook is present it is created, and the order is successfully added to the orderbook to the system.

Actual: Output matches what has been expected successfully.

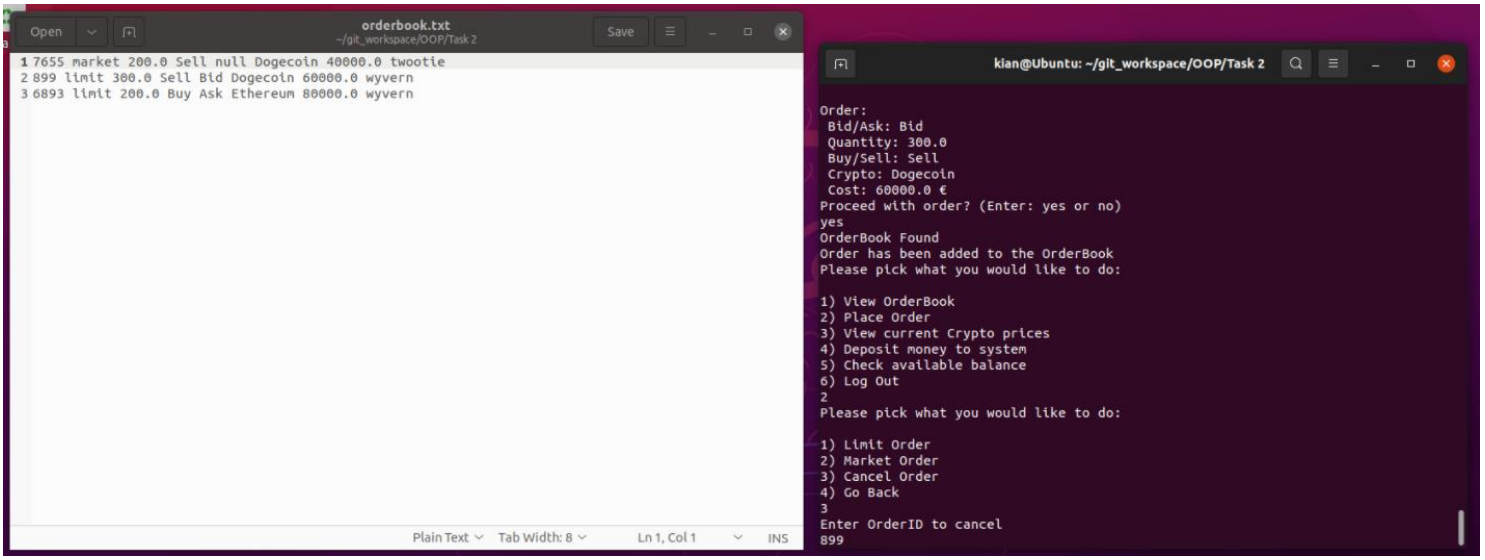
Fifth test:

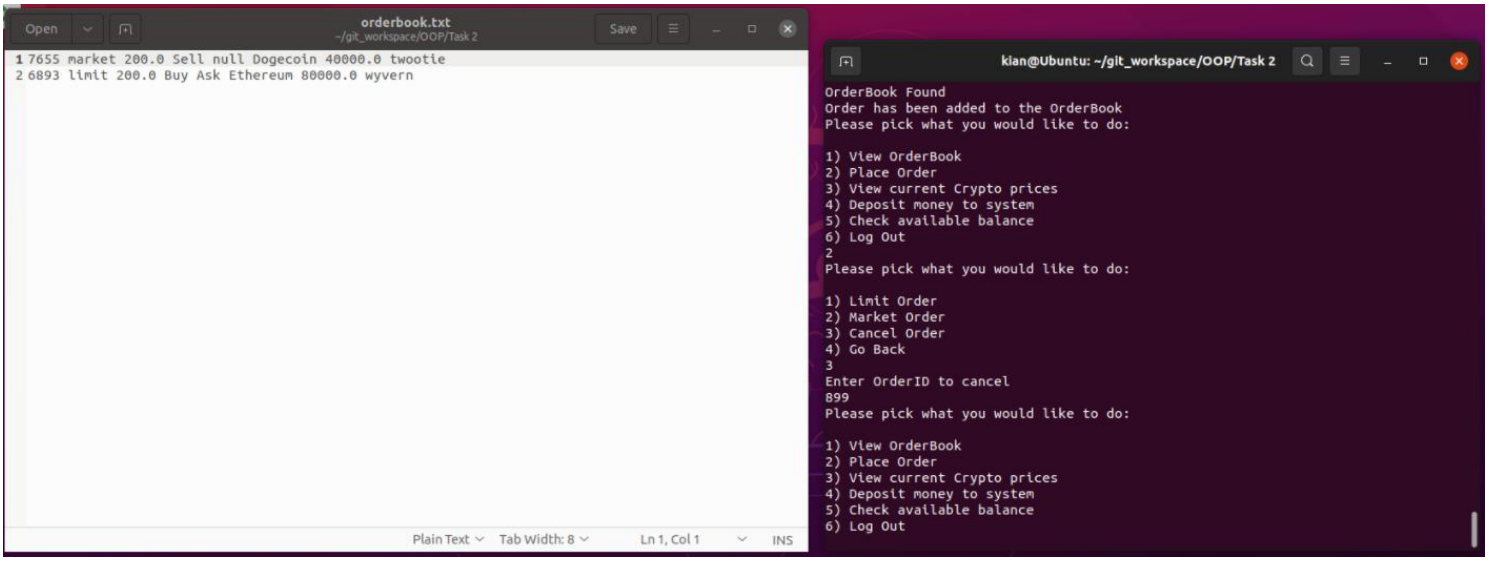


Expected: User attempts place an order; the order book is found the new order is added and the orderbook is sorted. User placed a buy order therefore it is expected that they lose the amount the amount the ordered.

Actual: Output matches what has been expected successfully.

Fifth test:





Expected: User attempts place an order; the order book is found the new order is added and the orderbook is sorted. User placed a buy order therefore it is expected that they lose the amount the amount the ordered.

Actual: Output matches what has been expected successfully.

Task 3: Big Integers

UML Diagram does not serve any functionally for this question therefore it has been omitted.

Design and Approach:

The approach taken to support big integers was to represent each number in a vector of type short where each slot represents a number between 0 and 9 and the length of the vector is given as the template parameter 'Template'. If the number is less than the size of Template the rest of the vector is padded with 0's.

With the use of static assert an error is given if the user attempts to enter a length integer which isn't in the form 2^n where n is the number size of the Template and Template > 0 and is a real number N. The user may pass either a string as an input, for when Big Integer inputs need to be taken and otherwise function overloading is done for regularly sized inputs of type int.

All required operator overloading's have been implemented including Arithmetic Operators {+, -, *, /, %}, Increment and Decrement Operators, Assignment Operators, Relational Operators and finally the << and >> operators. As a bonus the << operator has been overloaded to print any value stored in myuint. *Please refer to the myuint.hpp file for full list of what operators have been overloaded.*

For code writing efficiency, helper functions such as {Multiply, Add, Subtract etc} where implemented to be used by operators of a similar nature to avoid rewriting certain functionalities. Finally a ***convert_to()*** function has been implemented to convert any myuint to a datatype of the users choice.

Test Cases Considered

The first test case considered is implementation of *foo():*

```
int foo()
{
    myuint<512> i(5);
    myuint<512> j = (i<<1000) + 23;
    return j.convert_to<int>();
}
```

Output:

```
"/home/kian/git_workspace/OOP/Task 3/build/myuint"kian@Ubuntu
3/build/myuint"
23
```

Which is as expected.

Various functions were implemented to test out the functionalities of the implemented operators. The following are the results of these functions:

1) Calling test function ***AdditionTests()*** which gives expected output:

```
Addition of 3980 + 20 is: 4000
Adding 20 to previous answer: 4020
Adding 100 to previous answer 4120
Adding 100 to 20: 120
Add4: 1 k: 2
Add5: 2 L: 2
```

2) Calling test function ***RelationalTests()*** which gives the following expected output:

```
kian@Ubuntu:~/git_workspace/OOP/Task 3/build$ "/home/kian/git_workspace/OOP/Task 3/build/myuint"
5 is equal to 5
3213123123123173218 is greater then 3231243
32000 is greater or equal to 2000
32000 is not equal to 2000
```

3) Calling test function ***SubtractionTests()*** which gives the following expected output:

```
kian@Ubuntu:~/git_workspace/OOP/Task 3/build$ "/home/kian/git_workspace/OOP/Task 3/build/myuint"
Subtracting 300000000000000000000 from 2000000000000000000000: 3000000000000000000000
Subtracting 20 from 40: 20
Subtracting 2000 and 15000 from 30000: 13000
sub2: 1 n: 1
sub3: 2 p: 1
```

4) Calling test function ***MultiplicationTests()*** which gives the following expected output:

```
kian@Ubuntu:~/git_workspace/OOP/Task 3/build$ "/home/kian/git_workspace/OOP/Task 3/build/myuint"
33 * 0: 0
22 * 42: 924
```

5) Calling test function ***DivisonTests()*** which gives the following expected output:

```
kian@Ubuntu:~/git_workspace/OOP/Task 3/build$ "/home/kian/git_workspace/OOP/Task 3/build/myuint"
100 divided by 5 is 20
20 divided by 20 is 1
Error: attempting to divide by 0
```

6) Calling test function ***ModulusTests()*** which gives the following expected output:

```
kian@Ubuntu:~/git_workspace/OOP/Task 3/build$ "/home/kian/git_workspace/OOP/Task 3/build/myuint"
625 mod 5 is: 0
2000 mod 12 is: 8
```

Limitations:

The most present limitation in this implementation is the inefficiency in storing values within vector<short>, given more time this code can be optimized to instead use vector<bool> and represent each number as a binary value true or false where the value 1 represents true and 0 represents false, and simple binary addition subtraction etc could be used for the operators. Another present limitation is that the library does not allow conversion from one type to another and thus such as the example foo ‘i’ and ‘j’ must be of the same size.