Object Oriented Programming

CPS 2004

Daniel James Sumler

0107297A

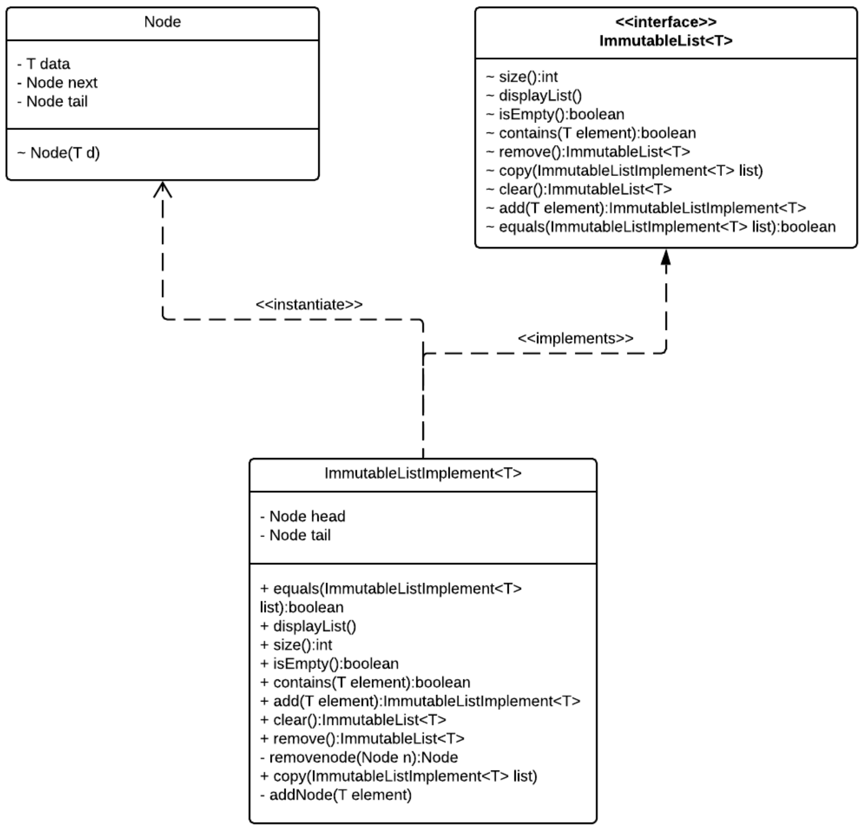
Gitlab repo : https://gitlab.com/Sumler/oop-assignment/tree/master

Final commit : `1b72056a`

Question 1

Immutable List (in Java)

UML Diagram


Node
- T data
- Node next
- Node tail
~ Node(T d)
<<interface>>
ImmutableList<T>
~ size():int
~ displayList()
~ isEmpty():boolean
~ contains(T element):boolean
~ remove():ImmutableList<T>
~ copy(ImmutableListImplement<T> list)
~ clear():ImmutableList<T>
~ add(T element):ImmutableListImplement<T>
~ equals(ImmutableListImplement<T> list):boolean
<<instantiate>>
<<implements>>
ImmutableListImplement<T>
- Node head
- Node tail
+ equals(ImmutableListImplement<T> list):boolean
+ displayList()
+ size():int
+ isEmpty():boolean
+ contains(T element):boolean
+ add(T element):ImmutableListImplement<T>
+ clear():ImmutableList<T>
+ remove():ImmutableList<T>
- removenode(Node n):Node
+ copy(ImmutableListImplement<T> list)
- addNode(T element)

Description of the approach:

In order to implement an *ImmutableList*, firstly an interface was created and the most important method signatures (namely add, copy, remove and clear). After this, the implementation class, *ImmutableListImplement*, was designed in order to retain a copy of the original list while returning an edited list. In this way, the original list is preserved, including the memory addresses.

A linked list approach was used in this implementation. This was to not only ensure that future changes could easily be made to the methods, but also to ensure that the lists were truly immutable.

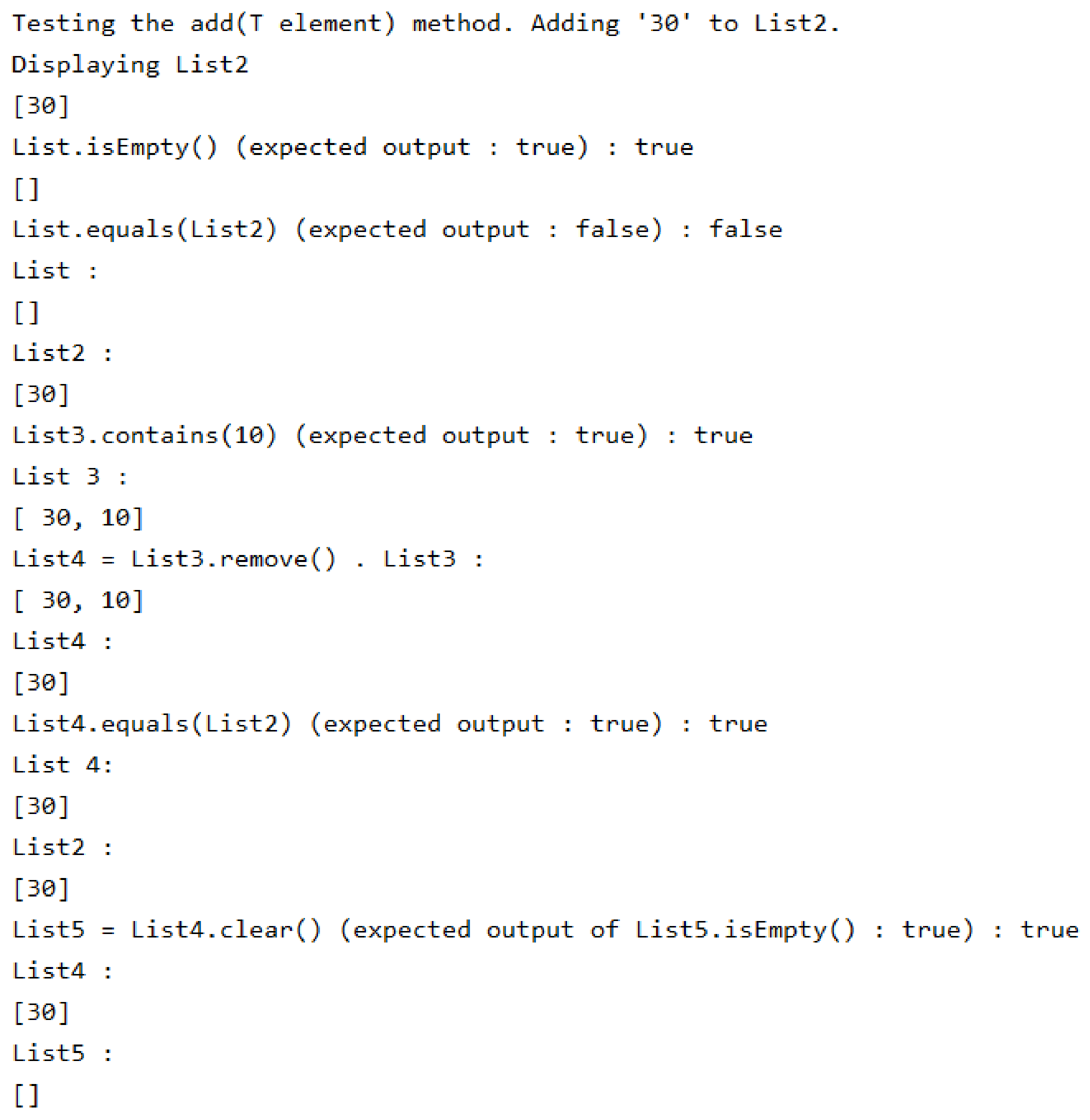
In order to implement a linked list, a *Node* class was created which can hold a piece of Data of a generic type T, as well as a pointer to the Node which follows it. Head and Tail Nodes were also employed for an accurate implementation.

The methods were thoughtfully designed in order to simplify the overall experience for the user. For example, when executing the *copy()* method, the copied List is being placed into the list the function is being used on (*List.copy(List2) – List2 is being copied into List*). The alternative would be to return an entirely new list, but this was adopted for neatness and convenience sake.

Test Cases:

In order to test the *ImmutableList* Implementation, every function was tested thoroughly and it was concluded that every function worked, and more importantly, did not alter the given list. A function was executed on one or more lists, and then both lists were displayed to show that nothing was altered.

The test output can be seen below :



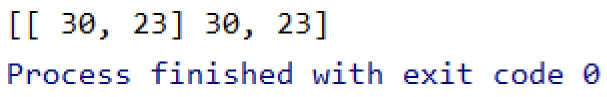
```
Testing the add(T element) method. Adding '30' to List2.
Displaying List2
[30]
List.isEmpty() (expected output : true) : true
[]
List.equals(List2) (expected output : false) : false
List :
[]
List2 :
[30]
List3.contains(10) (expected output : true) : true
List 3 :
[ 30, 10]
List4 = List3.remove() . List3 :
[ 30, 10]
List4 :
[30]
List4.equals(List2) (expected output : true) : true
List 4:
[30]
List2 :
[30]
List5 = List4.clear() (expected output of List5.isEmpty() : true) : true
List4 :
[30]
List5 :
[]
```

If one wishes to run the code which gives the above output, kindly run the ‘Main.java’ file located in the Question1 folder.

Concurrent Tests:

In order to concurrently test my solution as per the specification requirements, a custom class was created with inherits the *Thread* superclass. I then overrode the *run()* function in order to allow me to dictate what each instance of my Thread class would do. In this case, I called the *add(T element)* and *remove()* function multiple times.

When both threads were run together, they produced the same output, but since they finished at roughly the same time, the *displayList()* function became entangled within the two threads and the output was distorted as seen below:



```
[[ 30, 23] 30, 23]
Process finished with exit code 0
```

This therefore shows that when run in two threads, the list works as expected, and at roughly identical times.

Bonus Question:

This immutable list performs worse than its mutable list counterpart, mainly because of extra steps which must be carried out when executing functions. For example, when working with an immutable list, whenever a list is to be altered, a copy of that list is to be made in order to preserve the original one, whereas a mutable list would be able to skip this step. This is therefore time and resource consuming.

The most obvious optimization to make to this immutable list would be to create a very optimized version of the *copy()* function which uses less resources and time. Other than this glaring issue, its hard to suggest further optimizations other than to clean up existing functions.

Critical Evaluation and Limitations:

When taking a look back at the finished product with relation to the task specifications, I believe that I created a working solution. I believe that the basic functions are as optimized as possible, and that working with a Linked List was the best option in this case.

Although my solution is a valid one, it also comes with some limitations. One of them being the fact that not every method that is in the *List* class is available to use in my *ImmutableList* class. This was a design choice due to the fact that many of the *List* methods were repetitive of other methods in the same class.

Another limitation is that when using the class in a programming environment, the list is not strictly immutable, meaning that an already-existing list can easily be overwritten if the user so desires. (For example, using List4 = List4.remove();).

One of the biggest limitations of my solution is that certain methods do not exactly recreate their *List* class counterpart. For example, the *remove* function in my implementation only removes the last element from a list. This was done because an optimized solution was not found in order to recreate its counterpart.

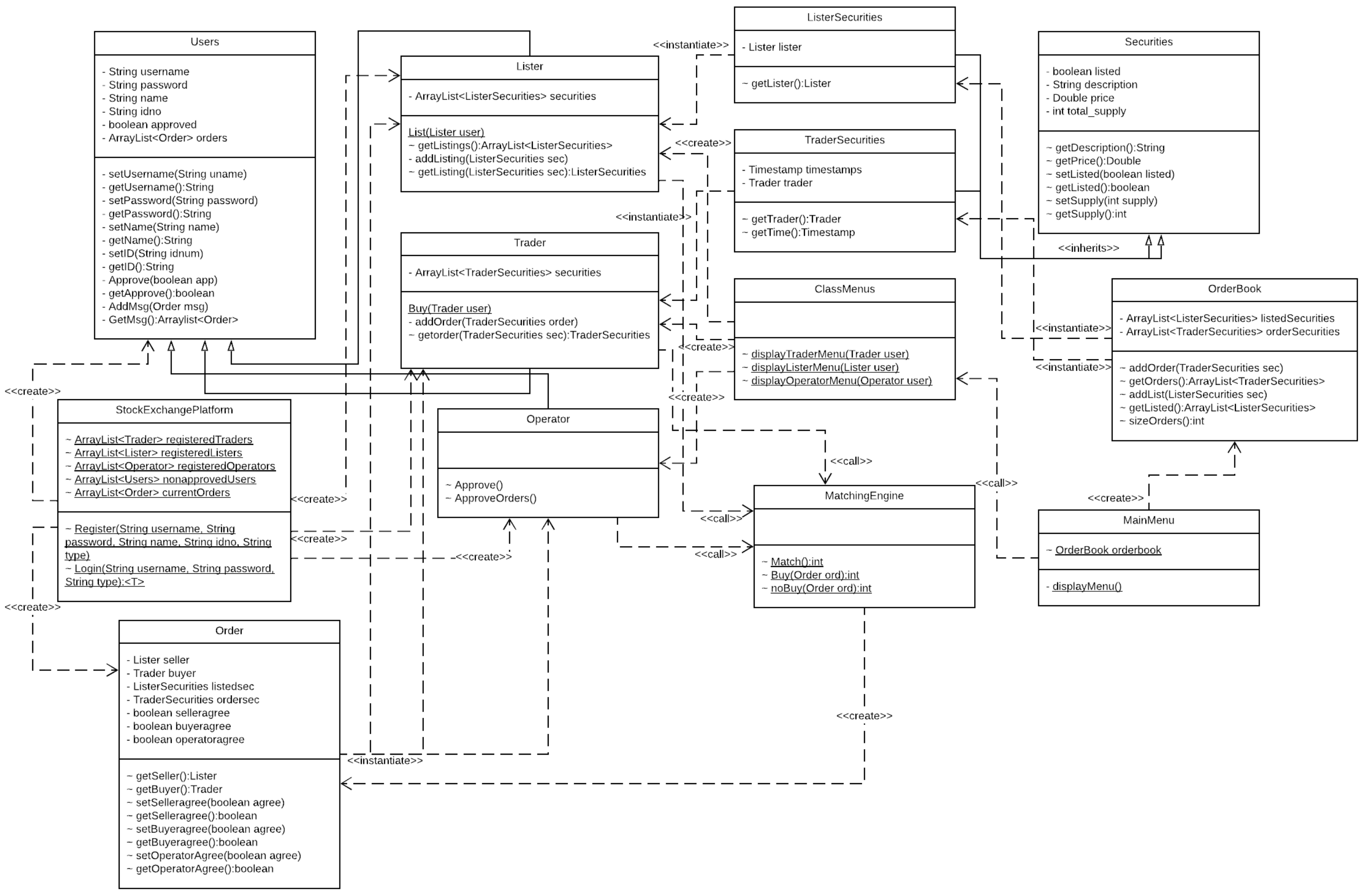
Improvements which could be made to this solution would be to redesign the *remove* method in order to act more like the original, make sure that an instance of *ImmutableList* cannot be altered and finally implement as many methods as feasible from the original *List* class.

All in all, I believe that this solution is a suitable one with respect to the given task.

Question 2

Stock Exchange System (in Java)

UML Diagram


Users
- String username
- String password
- String name
- String idno
- boolean approved
- ArrayList<Order> orders
- setUsername(String uname)
- getUsername():String
- setPassword(String password)
- getPassword():String
- setName(String name)
- getName():String
- setID(String idnum)
- getID():String
- Approve(boolean app)
- getApprove():boolean
- AddMsg(Order msg)
- GetMsg():Arraylist<Order>
Lister
- ArrayList<ListerSecurities> securities
List(Lister user)
~ getListings():ArrayList<ListerSecurities>
- addListing(ListerSecurities sec)
~ getListing(ListerSecurities sec):ListerSecurities
Trader
- ArrayList<TraderSecurities> securities
Buy(Trader user)
- addOrder(TraderSecurities order)
~ getorder(TraderSecurities sec):TraderSecurities
Operator
~ Approve()
~ ApproveOrders()
ListerSecurities
- Lister lister
~ getLister():Lister
TraderSecurities
- Timestamp timestamps
- Trader trader
~ getTrader():Trader
~ getTime():Timestamp
ClassMenus
~ displayTraderMenu(Trader user)
~ displayListerMenu(Lister user)
~ displayOperatorMenu(Operator user)
Securities
- boolean listed
- String description
- Double price
- int total_supply
~ getDescription():String
~ getPrice():Double
~ setListed(boolean listed)
~ getListed():boolean
~ setSupply(int supply)
~ getSupply():int
OrderBook
- ArrayList<ListerSecurities> listedSecurities
- ArrayList<TraderSecurities> orderSecurities
~ addOrder(TraderSecurities sec)
~ getOrders():ArrayList<TraderSecurities>
~ addList(ListerSecurities sec)
~ getListed():ArrayList<ListerSecurities>
~ sizeOrders():int
MatchingEngine
~ Match():int
~ Buy(Order ord):int
~ noBuy(Order ord):int
MainMenu
~ OrderBook orderbook
- displayMenu()
StockExchangePlatform
~ ArrayList<Trader> registeredTraders
~ ArrayList<Lister> registeredListers
~ ArrayList<Operator> registeredOperators
~ ArrayList<Users> nonapprovedUsers
~ ArrayList<Order> currentOrders
~ Register(String username, String password, String name, String idno, String type)
~ Login(String username, String password, String type):<T>
Order
- Lister seller
- Trader buyer
- ListerSecurities listedsec
- TraderSecurities ordersec
- boolean selleragree
- boolean buyeragree
- boolean operatoragree
~ getSeller():Lister
~ getBuyer():Trader
~ setSelleragree(boolean agree)
~ getSelleragree():boolean
~ setBuyeragree(boolean agree)
~ getBuyeragree():boolean
~ setOperatorAgree(boolean agree)
~ getOperatorAgree():boolean
<<instantiate>>
<<create>>
<<instantiate>>
<<create>>
<<create>>
<<inherits>>
<<instantiate>>
<<instantiate>>
<<call>>
<<call>>
<<call>>
<<call>>
<<create>>
<<create>>
<<create>>
<<create>>
<<create>>
<<create>>
<<create>>
<<instantiate>>

Description of the approach:

In order to create a solution for the given task, I approached each requirement separately.

First, I coded the main functions and classes which would be used throughout the entirety of the program (Such as the Users class, which is then inherited by Listers, Traders and Operators).

Secondly, I shifted my attention on Securities. I wanted to make sure that Securities could be differentiated between Ordered Securities (which are placed by a Trader) and Listed Securities (which are listed by a Lister). In order to do this, a superclass Securities was created, and two classes which inherit from it (namely ListerSecurities and TraderSecurities). Each Security has either a Lister or Trader attached to it (as can be seen within their respective class declarations), and this helps to keep track of which user placed or listed which security.

Thirdly, the Matching Engine was created. This was created by searching through *ArrayLists* of TraderSecurities and ListerSecurities and matched them together based on price per unit and total supply. If a match was found, an instance of the *Order* class would be created, and each user (Trader, Lister and Platform Operator) would receive an approval message detailing the match which had been found. If any one of the users denies the message, then the order would be cancelled and deleted entirely.

The approach that I took also allows for future changes, due to the fact that classes and methods have set tasks and never stray beyond said tasks. This implies that current features cannot be broken by adding new features. The fact that I chose to incorporate GUI into this task also helps when adding more features, as it helps said features to stand out more. This is important if you're rolling out new features to a wide number of people who are using the application.

Test Cases:

In order to test my solution, three test unit files were created. These files were made to test the main functions of the Exchange System, namely the *Register*, *Login* and *MatchingEngine* methods.

The main functions to be tested were whether a user is successfully registered and whether they can login, and whether matching securities are found and successfully bought by the Matching Engine. As one can see in the test files provided, all of these cases were considered and tested.

Not all of the methods in the program were able to be tested, due to the fact that many required user input to be able to function, which wasn’t possible with the Unit Testing library (JUnit).

Critical Evaluation and Limitations:

With respect to the given task, I believe that an adequate solution has been provided. Not only was care given when designing the classes to be reliable and use Object-Oriented Principles, but also when making sure that the design of the system was easy to follow and soft on the eyes, hence my choice to use a Graphical User Interface.

Although a lot of work went into designing my system, it still has its limitations.

Firstly, securities are matched when they have the same price and supply, rather than having a complex system which compares price and deducts supply from the order and listed security.

Secondly, traders are only allowed to place buy orders rather than sell, and listers are only allowed to list securities. Although this may go against the requirements of the task, the trader can also opt to create a lister account under the same details and list as many securities as required there.

In order to improve the system, more time can be put into developing these aspects and making them more complex.

Regarding future changes:

Making future changes to the system I created is relatively easy, made simpler by the use of GUI and how the created classes are used.

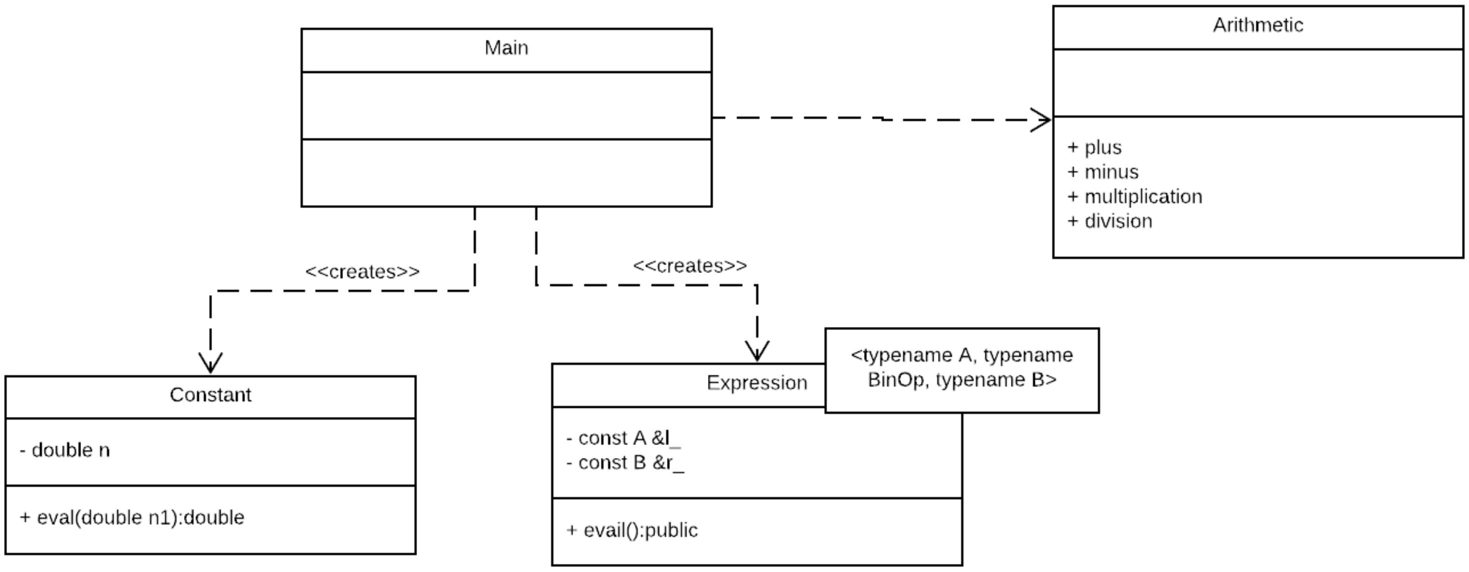
A method on how future changes can be made with specific reference to the Audit Trails change can be seen below :

Since every security and order has a corresponding lister and trader, each instance of *Order* contains all the details from both the buyer and the seller classes. This means, that in order to produce a logbook, all the programmer would need to do would be to print the details of each order once they have been fulfilled or cancelled.

Question 3

Expression Language (in C++)

UML Diagram



Description of the approach:

In order to approach this task with care, particular importance was given to the use of templates, as was asked within the question. This was done to allow generic input to be accepted by the methods.

Therefore, when coding the simple *plus*, *minus*, *division* and *multiplication* classes (found in *Arithmetic.h*), which are all classes which either add, subtract, multiply or divide two values, a template was added to each class.

After this, a General Expression template was added which accepts two values from a template as well as an operator command. For example, if one wished to execute the *plus* method, one would just have to call that method using the general template format.

Next, the operators were overloaded and replaced with a custom operator which would instead use the *Arithmetic.h* functions over the standard C++ definitions. Therefore, if the general expression template is to be used with one of the arithmetic functions, they would be redirected to use one of the *Artithmetic.h* functions.

Unfortunately, after all of this work, I was unable to find a working solution to the problem at hand. The code does not run as expected, but one can still find attached my attempt at solving said problem.

Critical Evaluation and Limitations:

Overall, I was unable to produce a working solution to the task, although I believe that I tried to understand the goings on of the task, but was ultimately unable to.