L-Università ta' Malta
Faculty of Information &
Communication Technology

Assignment Report

Manwel Bugeja

January 12, 2020

Contents

0.1 Task 1

0.1.1 The Design

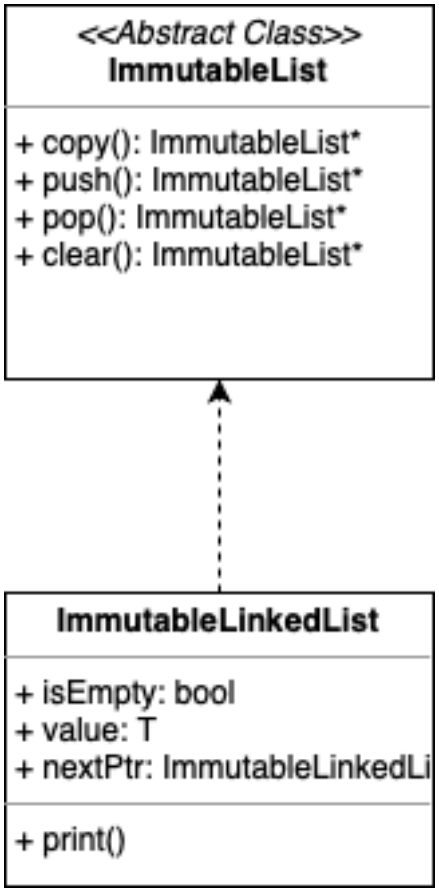

Figure 1: Task 1 UML Diagram

In this section a pure abstract class was created called 'ImmutableList' containing the required functions. Another class was created called 'ImmutableLinkedList' which inherited from the first class. This was done to implement the methods of the abstract class for a structure called a linked list. The class has 2 constructors and 3 variables.

The constructors

The key to this design is the fact that there are actually two constructors but one of them is inaccessible by the user. The hidden constructor has the ability to create nodes with custom values for the variables 'isHead',

'value' and 'nextPtr' whose use is explained later. On the other hand the public constructor is only able to set the value of a head node. The public constructor is implementer using the private, passing 'isHead' as true and 'nextPtr' as nullptr and the value containing what is passed by the user.

'isHead'

This variable is a boolean type which indicates if the current node is the head of a list or not. This is needed to distinguish the head from other nodes of the list. 'isHead' enables the possibility to have an empty list as the value head node is completely ignored by the api.

'value'

This variable contains the value of the node which is the same type as that indicated at the creation of the immutable list.

'nextPtr'

This variable contains a pointer towards the next node of the immutable list.

0.1.2 Critical evaluation and limitations

Initial testing

To test the immutability of a list, a list was created with some initial values. Then a new list was created using the old list and the push method. A third list was created using the first one and the push method with a different value passed as a parameter. All three lists were printed as should.

Concurrent testing

The list was also tested concurrently. A function was created that receives a list and a value to append. Two threads were created and the function was run in each thread with a different value passed. After the threads were joined, the list that was passed was printed to show that it was unchanged.

Limitations

To use this immutable list, the user would have to take care of the memory management manually. This could have been avoided by using smart

pointers.
This design can also be inefficient when compared to its mutable counterpart. An example of this is when adding a new member to the list. A simple task requires for a creation of a whole new list which takes O(n). On top of that, accessing the memory for every element makes the process even slower. Improvements to my immutable list to improve its preformance could be done by making the linked list doubly linked.

0.2 Task 2

0.2.1 The Design

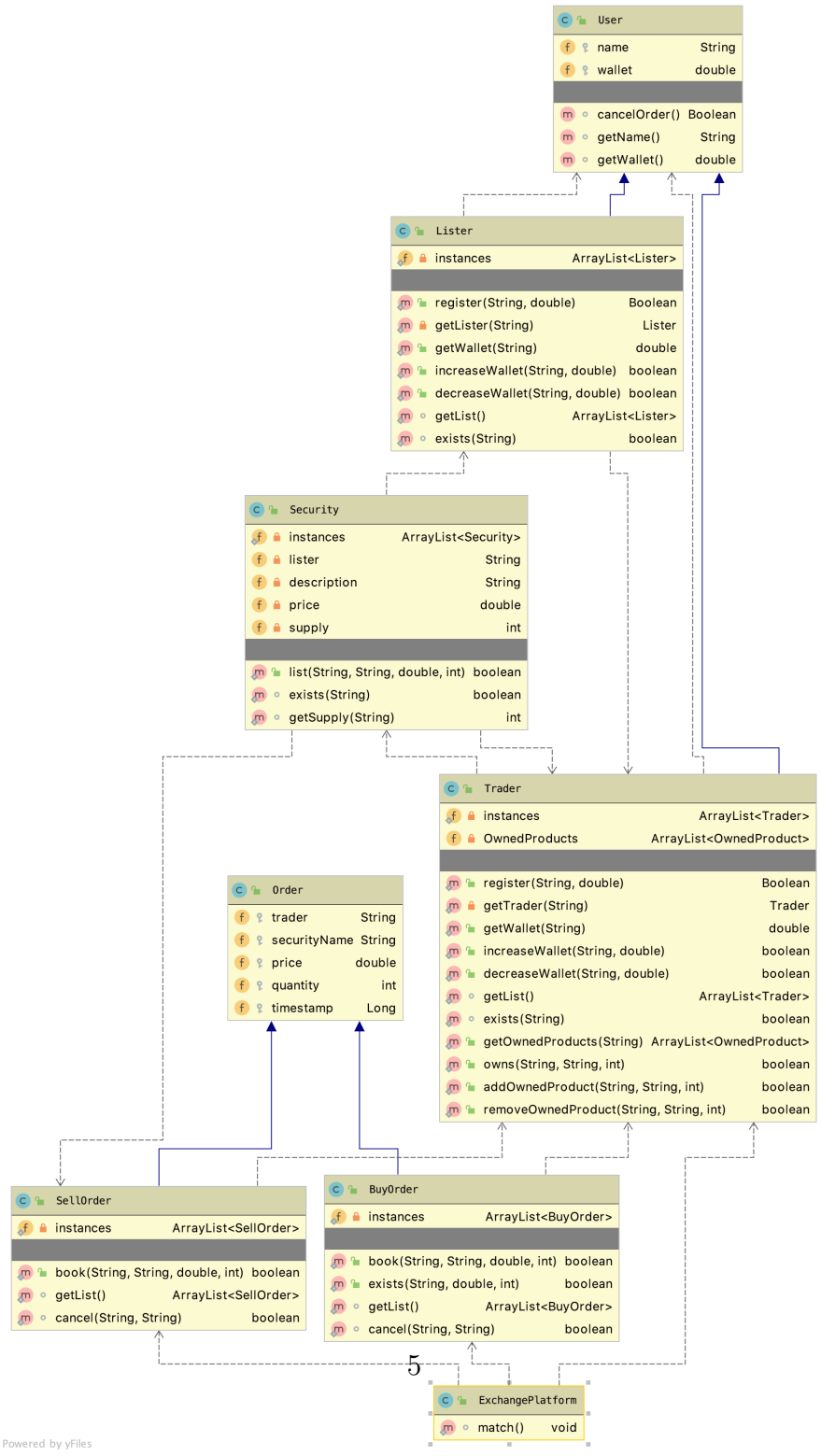

Figure 2: Task 2 UML Diagram

In my design for this task, three main classes were created for securities, orders and users. For the latter two, two further sub classes where created. These three main classes work in a very similar manner. Created instances are tracked by being added to a static array list and having a unique variable to identify them. Operations on these classes work by iterating through the created instances until the correct one is found.

0.2.2 Testing

For the testing, JUnit was used and new methods were added as the project was developed. The tests tried the functions with invalid and valid parameters.

0.2.3 Critical evaluation and limitations

My solution uses array lists to keep track of objects. This means that it takes O(n) to find the desired object during a search. This would have been more efficient if hash maps where used. On top of that, weak referencing would have made the design more memory efficient.

0.2.4 Polymorphism

To implement the log, a class can be created with a new list structure with only an add method (and no remove). Every time 'book()' or 'list()' or 'cancel()' is called, the action is added to this new list. Tampering this list can be avoided with proper encapsulation.
Apart from that, the matching engine could have been tested more rigorously by testing even more possible combinations.

0.3 Task 3

0.3.1 The Design

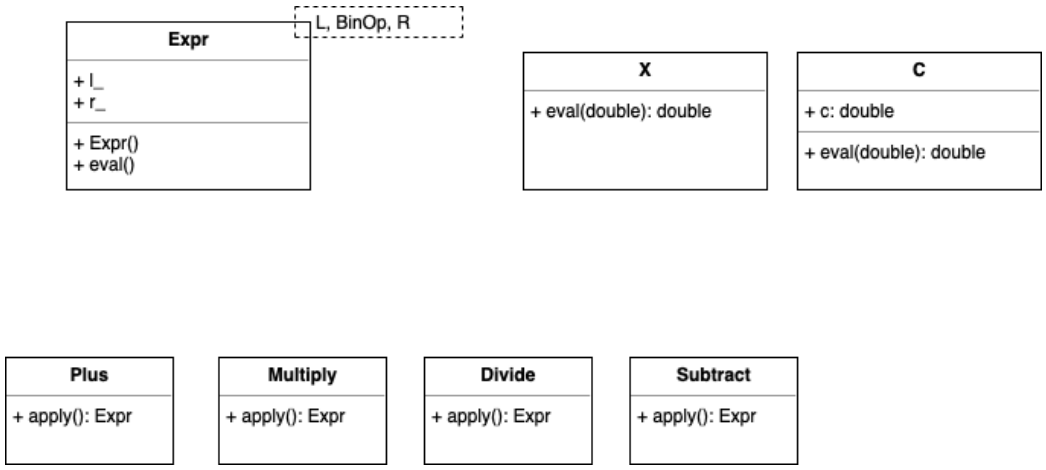

Figure 3: Task 3 UML Diagram

For this task, a template class 'Expr' was created. The purpose of this was to represent the AST and unwind it as should. Template classes for the operators were also created which server a similar purpose. The operators were also overloaded using templated functions.
The expression in the main is worked out as shown.

Listing 1: Example of an expression

```
((C(10) / X()) + (C(2) * X())).eval(4)

=
( (C(10) / X()).eval(4) ) + ( (C(2) * X()).eval(4))

=
( C(10).eval(4) / (X).eval(4) ) + ( C(2).eval(4) * X().
    eval(4) )

=
(10 / 4) + (2 * 4)
```

For the intergration function, a template function was created that received 'Expr' along with the other requirements as parameters. The function implemented the trapeziod rule.

0.3.2 Testing

For the testing, an expression that used all operators was used and the output was checked. As for the integration part, an expression was passed and checked against the correct answer.

0.3.3 Critical evaluation and limitations

It is difficult to know whether the inlined functions were done as should. Also, since the expression is evaluated in a kind of recursive manner (with the classes X and C being the base cases) the code generated could be very cumbersome especially for very long expressions.