University of Malta

Faculty of Information and Communication Technology

CPS-2004 Object -Oriented Assignment

 Coursework 2021-2022

Icon

Description automatically generated

|  |  |
| --- | --- |
| Faculty of ICT  Scholastic year: 2021/22 | Nathan Bonavia Zammit (198402L) |

**FACULTY OF INFORMATION AND**

**COMMUNICATION TECHNOLOGY**

Declaration

Plagiarism is defined as “the unacknowledged use, as one’s own work, of work of another person, whether or not such work has been published” (Regulations Governing Conduct at Examinations,1997, Regulation 1 (viii), University of Malta).

I / We\*, the undersigned, declare that the [assignment / Assigned Practical Task report / Final Year Project report] submitted is my / our\* work, except where acknowledged and referenced.

I / We\* understand that the penalties for making a false declaration may include, but are not limited to, loss of marks; cancellation of examination results; enforced suspension of studies; or expulsion from the degree programme.

Work submitted without this signed declaration will not be corrected, and will be given zero marks.

\* Delete as appropriate.

(N.B. If the assignment is meant to be submitted anonymously, please sign this form and submit it to the Departmental Officer separately from the assignment).

Graphical user interface, text, application

Description automatically generated

Contents

[DAG (in C++) 4](#_Toc93686281)

[Implementation of the code 4](#_Toc93686282)

[Test Cases 4](#_Toc93686283)

[UML 6](#_Toc93686284)

[Possible Improvements: 6](#_Toc93686285)

[Crypto Exchange (in Java) 7](#_Toc93686286)

[Implementation of the Code 7](#_Toc93686287)

[Test Cases 7](#_Toc93686288)

[UML 9](#_Toc93686289)

[Possible Improvements: 10](#_Toc93686290)

[Big Integers (in C++) 10](#_Toc93686291)

[Implementation of the Code 10](#_Toc93686292)

[Test Cases 11](#_Toc93686293)

[UML 12](#_Toc93686294)

[Possible Improvements 12](#_Toc93686295)

# DAG (in C++)

## Implementation of the code

The first task consisted of building a templatized directed acyclic graph data structure (DAG). A DAG consists of nodes which are connected by edges. The edge class creates an edge which returns a source node and a target node. The vertex class consists of multiple functions which are used to create the nodes in the graph, and populate the aforementioned source and target nodes. The dag class is used to build the final DAG. This is done through the use of an edge list. The edge list contains all the edges that will be used in our graph. For example, the first edge is looking for the source node ‘1’. The program will attempt to find said node in the graph that we are building. However, since the graph is currently empty, it will fail and add the node ‘1’ to the graph itself. This is also repeated with the target node ‘2’. After this step if there is any other node in the graph which is using node ‘1’ as a source node, the program will find it in the graph already and use the one in the graph. If any nodes are to be removed this can also be done via the remove\_vertex() function. The main method used will print out the DAG and the edge list. In some cases, one or multiple nodes are removed and in this case the DAG and edge list will be outputted again without the nodes that were removed.

## Test Cases

For the testing the following edge list was used:

Text

Description automatically generated

The DAG outputted like so:

|  |  |
| --- | --- |
| Text  Description automatically generated | A picture containing clock  Description automatically generated |

Fig.2: Output of DAG with a diagram of the graph on the right

Along with the edge list as seen below:

Text

Description automatically generated with low confidence

Fig.3: The Edge List

The first two nodes were removed from the DAG and the outputs changed, however the main issue is that disconnected nodes were not completely removed.

|  |  |
| --- | --- |
| Text  Description automatically generated | Text  Description automatically generated with low confidence |

Fig.4: The DAG after removing nodes 1 and 2

Diagram

Description automatically generated

Fig.5: Diagram of the graph for the without nodes 1 & 2

## UML

Diagram

Description automatically generated

Fig.6: UML Diagram of Task 1

## Possible Improvements:

Disconnected Nodes removal could make the output look cleaner.

* + - This was attempted in the DAG class however it caused bugs and errors and therefore it was left out

A function to check whether a cycle is present in the graph or not could have been added.

# Crypto Exchange (in Java)

## Implementation of the Code

The second task consisted of building a simple crypto exchange platform. The users that would use said crypto exchange would be Traders or Administrators. The Traders would have to register into the system and be approved by the Administrator. Upon running the program, the user would have 3 options: Register an account, Log in to an existing account, or Exit the system. If the user is an Administrator, then they have the option to approve any accounts waiting for approval, or log out of the system. If the user is an approved Trader, then they have the following options: View their Orderbooks, Buy Crypto, Sell Crypto, View any previous Crypto Transactions, Deposit Money into their account, View their balance, or log out of their account. The Trader and Administrator classes are subclasses to the User superclass since the user’s details such as username, password and ID are required for the Trader and Administrator’s functions to work and they are in the User() object.

## Test Cases

For the test cases three users were created, 2 Traders and 1 Admin.

Text

Description automatically generated

Fig.7: The users which were used for the Test Cases

In order to test out the Admin user, a user was registered so it culd be approved by the Admin.

|  |  |
| --- | --- |
| Text  Description automatically generated | Text  Description automatically generated |

Fig.8: Test user being regiestered and approved

The following menus are shown when logging into an Admin account (Left) or Trader account (right):

|  |  |
| --- | --- |
| Text  Description automatically generated | Text  Description automatically generated |

Fig.9: Logging into an Admin account (left) and Trader account (right).

The user ‘Beppe’ bought 10 of Polkadot and the order was successful since he had sufficient funds:

Text

Description automatically generated

Fig.10: User ‘Beppe’ buying 10 of Polkadot



Fig.11: User Beppe’s Balance

Graphical user interface, application

Description automatically generated

Fig.12: User Beppe’s Balance after buying 10 Polkadot

Text

Description automatically generated with low confidence

Fig.13: The successful transaction of the purchase

|  |  |
| --- | --- |
| Text  Description automatically generated | Text  Description automatically generated |

Fig.14: Depositing money into the account (left), and the balance after the addition of the money (right)

If at any point the user wishes to view their Order Book, the following is outputted:

Text

Description automatically generated

Fig.15: Viewing the Order Book

This can be used to abide by the FATF regulations so that any fraudelemt activity can be investigated by the authorities and scenarios recreated.

## UML

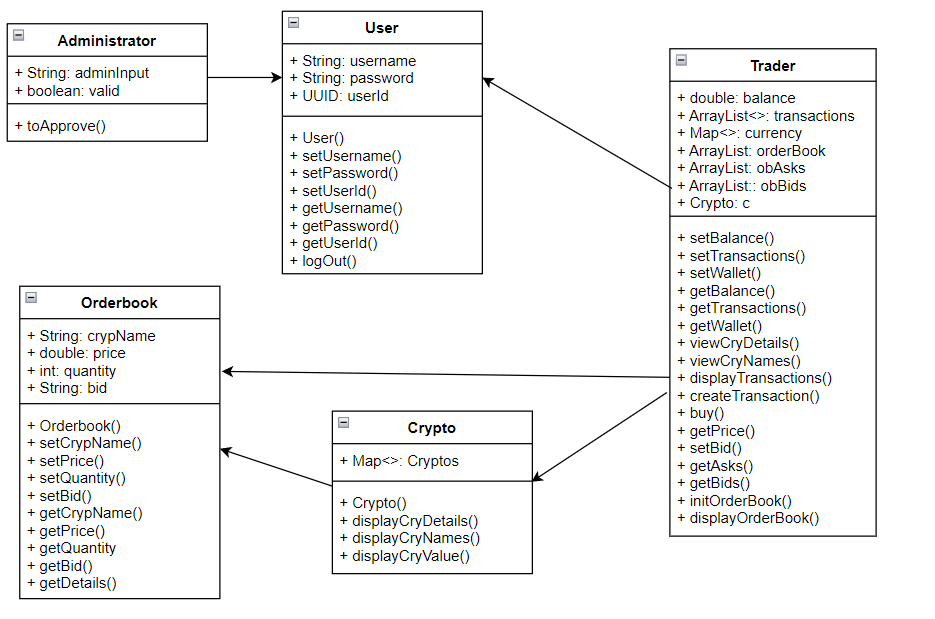


Fig.16: UML Diagram for Task 2

## Possible Improvements:

As can be seen in the test cases above, the user’s balance wasn’t being updated whenever a purchase occurred, however whenever money was deposited, the balance was updated.

The sell function wasn’t properly working as whenever the user tried to sell the crypto that they had bought, it perceived it as the user trying to buy it instead of selling it as seen below:

Text

Description automatically generated

Fig.17: Failure to sell the 10 Polkadot the user Beppe had bought

Due to time constraints and the deadline these errors couldn’t be fixed. If they were to be fixed all requirements for the task would have been met.

# Big Integers (in C++)

## Implementation of the Code

The third task consisted of writing a templated library that supports integers from 1-bit to 2048-bits. The libraray needed to only support unsigned integers with 2n bits which through the use of the function powerOf2(), this is achieved. All binary integer operators (+, -, <<, >>) needed to be implemented and overloaded in the library. This is done through the use of operator+(), operator-(), operator<<(), and operator>>(). The operators \*, /, % were implemented too: operator\*(), operator/(), and operator%(). Since vector<bool> is being used, the minimum number of bits possible are being used to store a number which increases the efficiency of the code and since the operations are binary operations this also makes the code more efficient. The functions Integer\_to\_Binary, reverseBinary, printNum(), and onesComp() are all helper functions to be able to use the operators implemented, print the numbers, or even calculate one’s compliment which is used in the operator-() function. The convertToInteger() function is used to convert the vector<bool> into an integer which is then used to display said integer during testing.

## Test Cases

For testing the numbers used were 86 stored in x, and 3 stored in y. As can be seen, the numbers are converted to binary and are stored as vector<bool>. When outputting the number the convertToInteger() function is used to convert it back to int.

Text

Description automatically generated

Fig.18: Numbers x(86) and y = 3 being declared

The operators =, -, <<, and >> were tested as seen below. The output was stored in various variables and then converted to integer.

Text

Description automatically generated

Fig.19: Operators +, -, <<, and >> being used.

Text

Description automatically generated

Fig.20: Operators / and % being used.

## UML

Diagram

Description automatically generated

Fig.21: UML Diagram of Task 3

## Possible Improvements

The operator\*() function tends to cause a segmentation fault and as such this could be fixed in the future.

Text

Description automatically generated

Fig.22: Segmentation fault cause by the operator\*() function

A fix for this segmentation fault was found below however it affected the multiplication calculation and as such wasn’t used.

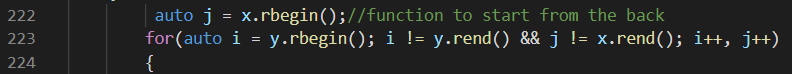


Fig.23: Line 223 with the fixed segmentation fault

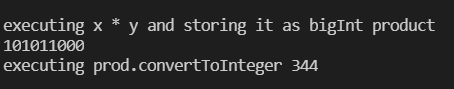


Fig.24: The calculation of 86 \* 3 is incorrect