# CPS2004 Report

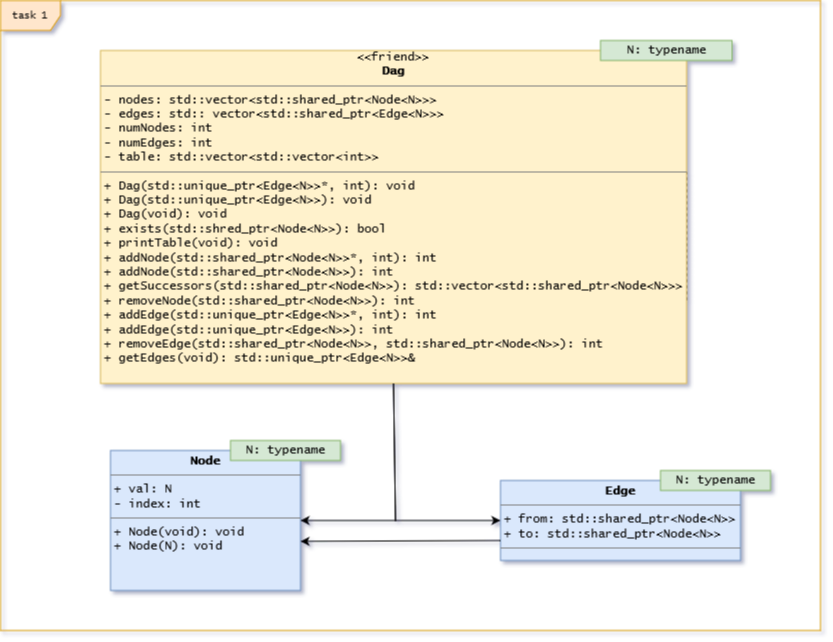
By Gabriel Hili

Contents

[CPS2004 Report 1](#_Toc91683639)

# Task 1

## UML Diagram



## Description

The Directed Acyclic Graph is implemented using 2 classes (Dag and Node) and 1 struct (Edge). Dag is a friend class to Edge and Node in order for the Dag Class to access private variables such as val and index. An edge consists of 2 nodes. The relationship of nodes and edges within the Directed Acyclic Graph is stored in an adjacency matrix. An element in the adjacency matrix is set to a non-zero value to represent an edge between 2 nodes.

Apart from the adjacency matrix, there also exists a vector of all nodes and a vector of all edges. The element in the adjacency matrix between 2 nodes is +1 of the index of the edge in this vector. Hence as an optimization when a node is trying to be removed, the row/column of the adjacency matrix will hold the index of all edges depending on that node.

Another optimization is in the function for checking cycles. If the user adds a list of edges, calling the cycle checking function for each one is slow and redundant. Hence the function is set to call only after the list of edges has been adding.

An instance of Dag can only be of a static type, but that type may be chosen through templates. Hence all instances of node and edges must match the template parameter of the corresponding Dag.

The user can manipulate the Dag class in many ways. The user can add Nodes, either singular or through a list. A list of edges can be passed to the Dag. An edge may refer a new node or a node already in the graph. In both cases the adjacency matrix is updated accordingly. **Removing a node will cause all edges dependent on that node to automatically be removed.**

The Dag accepts a list of unique pointer edges. Hence the edge ownership is safely moved to the Dag and the user cannot change any edges directly and must only interface with the Dag class to do so. The Dag accepts nodes in a shared pointer format.

The user can decide to instantiate a Dag with a list of edges, a singular edge or no edges at all, adding edges and nodes later on.

The Dag owns the individual edges by passing them as unique pointers. It has methods to remove individual nodes and edges. Since edges are unique pointers, edges are removed by passing in the from and to node to the function. Moreover the user can request a **reference** to the vector of edges in the Dag. However, the return type is set to const in order to prevent the user from directly altering the underlying data structure of the Dag without the proper checks and updates.

The Dag is declared as a friend class to the Node and Edge class. This is to prevent private members from being directly accessed from the launcher class but also allowing the Dag to manipulate its internal structure, such as accessing the index of a node.

## Extra Features

* A formatted way to view the underlying adjacency matrix
* The user can initialize the Dag with a single edge, or no edges at all, instead of only a list of edge.
* Checking for cycles only happens when necessary and not for every added edge.
* Throws exception in case of cycle.

## Test Cases

|  |  |  |
| --- | --- | --- |
| **Test Case** | **Expected Result** | **Actual Result** |
| Adding a node | Will add a record of that node in the adjacency matrix and the vector of all nodes | Match |
| Adding a node already in the graph | The Dag detects the node exists and ignores it. | Match |
| Adding an Edge of 2 new nodes. | The 2 Nodes are added to the graph. The element referenced by the 2 new nodes in the adjacency matrix is the +1 of the index of the new edge in the vector of all edges | Match |
| Adding an Edge of 1 node already in the Graph and a new node. | The New node is added to the graph and the old one is ignored. A link between the 2 nodes is created in the adjacency matrix. | Match |
| Adding an Edge of 2 nodes already in the graph, without an edge. | A link between the 2 nodes is created. No node is added. | Match |
| Adding an Edge between 2 nodes that already are linked. | The number of edges remains the same. The edge passed becomes obsolete and is lost. The previous edge takes precedence. | Match |
| Passing a nullptr as a Node. | Invalid Argument Exception is thrown. | Match |
| Passing nullptr as an Edge | Invalid Argument Exception is thrown. | Match |
| Removing an edge | The nodes are not removed from the adjacency matrix and the vector of all nodes. The edge is removed from the vector of all nodes. The adjacency matrix updates the elements to reflect the new indices of the vector of all edges. | Match |
| Removing an edge with nodes that are not in the graph | The graph remains unaffected. | Match |
| Adding a node which would create cycles. | An exception is raised. | Match |

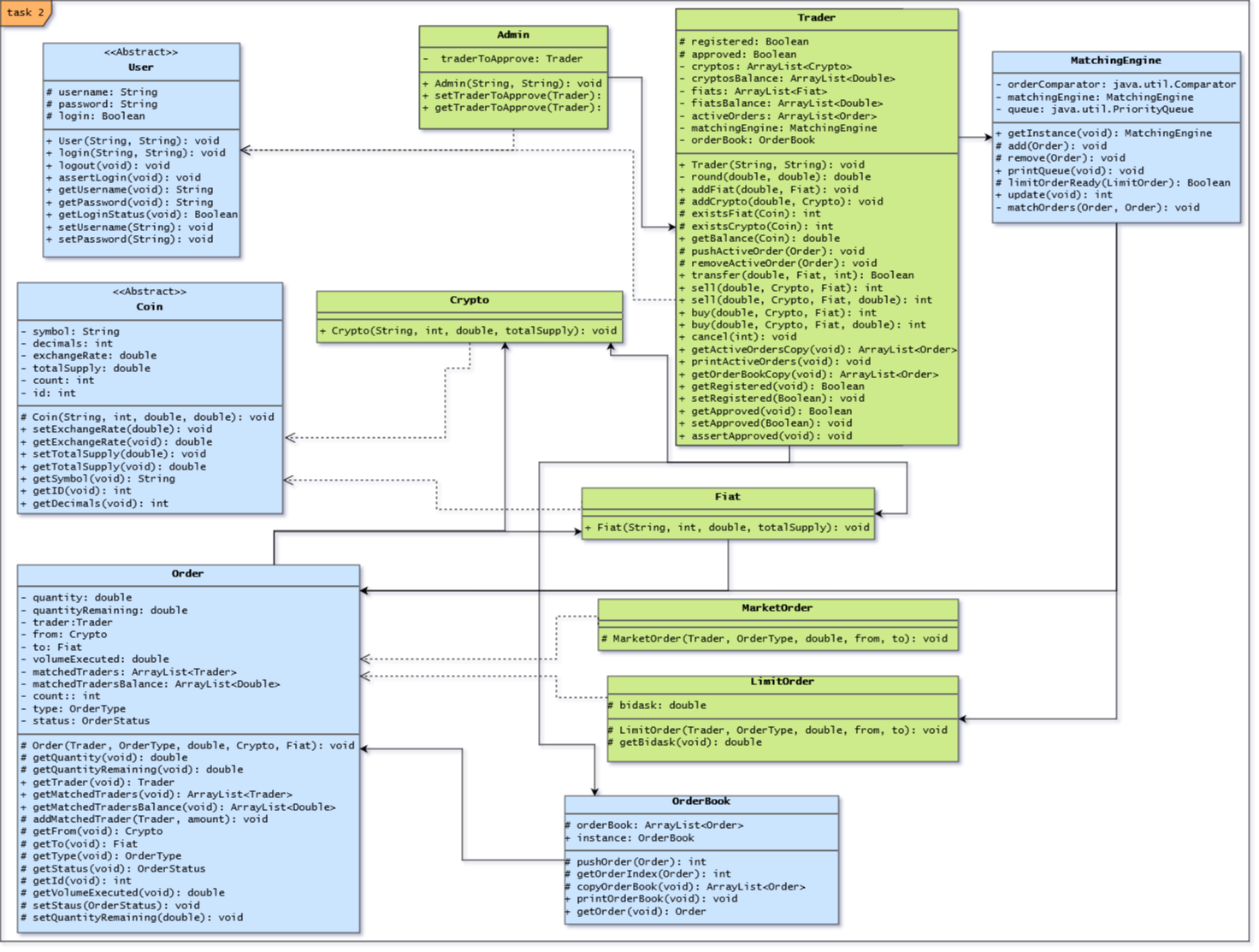
## Critical Evaluation

The Dag was created to have the fastest look up time. Removing a node gives you the index of the edge to remove with it without performing a linear search through the vector of all edges. Moreover checking if an edge exists between 2 nodes is performed in O(1) time. The Dag has been optimized to have O(1) look up time. However this comes at a cost with respect to space complexity. The adjacency matrix has O(n2) best space complexity. Despite this, the choice was still preferable than using a vector of linked lists, which although would have better space complexity, would not have constant look up time.

Shared and Unique pointers were used to automatically allocate and delete their memory, as opposed to using new and delete. Unique pointers for edges met the constraint that the edges need to be owned by the Dag. Unique pointers were not used for nodes as multiple individual edges may reference the same singular node.

# Task 2

## UML Diagram



## Description

The Crypto Exchange Platforms allows traders to place and cancel orders. For a trader to do so, he must be approved by an admin. This is done by first setting the registered flag to true. The trader instance is passed to an admin instance where the admin keeps a reference of the trader. The admin is then passed to the approve function in the Trader. If the admin’s reference to the trader matches, then the approved flag is set to true. Admins and Traders are subclasses of User. For any user to perform actions, they must be logged in.

The trader may place buy/sell market/limit orders between a Fiat Currency and a Crypto Currency. Fiat and Crypto are subclasses of the Coin Class. A Coin holds information such as exchange rate, which is always set to Euros in each coin. Exchanging from one coin to another involves first exchanging to euros then from euros to the second coin. Before placing an order, the system checks whether the trader possesses enough coins to carry the order if it were to execute. Before executing the order, the balances of the 2 traders are checked again, as they might fluctuate between placement and execution.

Each Order a Trader places contains information such as the Traders which helped execute the order and the volume executed of the Order. Every Order contains a unique ID which may be used to get that Order information from the OrderBook. The Trader can cancel any active order aslong as it hasn’t been fulfilled. Partially filled Orders may be cancelled, but obviously the Trader cannot get back the already traded currency back. MarketOrder and LimitOrder are subclasses of Order. An instance of Order cannot be instantiated as its constructor is protected. Only MarketOrders and LimitOrders are allowed to be instantiated in the Launcher class.

As the order progresses the OrderBook is updated. The OrderBook makes use of an ArrayList of type Order to store all incoming Orders. Hence MarketOrders and LimitOrders are stored in the same list. A deep copy of this ArrayList may be obtained at any time by all users. The OrderBook class is a singleton instance, since all Orders should belong to the same OrderBook.

All non-filled orders are placed in a Priority Queue in the singleton MatchingEngine class. The Priority Queue is sorted by the quantity of an Order, buy, sell, market or limit. The Priority Queue is implemented using Java’s inbuilt java.util.PriorityQueue. A static java.util.Comparator method is used to sort the Priority Queue by largest Quantity. When update is called the MatchingEngine attempts to find a pair of Buy and Sell Orders, which can be matched together. When a suitable pair is found, the Order’s remaining quantity is updated, which updates the volume executed variable. When an Order’s remaining quantity reaches 0 then the order is set to be fully filled, and removed from the Priority Queue.

Exceptions are promptly raised in case of illogical functionality, such as setting a negative quantity for an Order. Hence the Launcher file is required to catch these potential Exceptions and handle the flow of the program accordingly.

## Extra Features

* All Users must log in first.
* A Trader can filter the OrderBook to view only his current active orders.
* A function exists to add Fiat Currency to a Trader Profile. The function takes a bankid variable. This is so a Bank API can be used within this function to actually transfer funds in a real world application.
* Order Quantities are rounded to the number of decimals the Coin has.
* Every Order returns an Order ID which may be used to find that Order in the OrderBook.
* Proper Exception Handling
* An Error class which can be implemented to take a particular action given a certain thrown Exception.
* The Crypto Exchange Platform is in a package.

## Test Cases

|  |  |  |
| --- | --- | --- |
| **Test Case** | **Expected Result** | **Actual Result** |
| Trader attempts to buy/sell while not logged in. | Exception is thrown. | Match |
| Admin attempts to approve a Trader while not logged in. | Exception is thrown. | Match |
| User attempts to log in with incorrect username or password. | login flag is not set to true and an Exception is thrown. | Match |
| Admin attempts to approve Trader with registered flag set to false. | Trader is not approved and exception is thrown. | Match |
| Coin instance is initialized with negative values. | Values are clamped at 0 and cannot be negative. | Match |
| Trader wants to buy 1.001 of a Coin with decimals value set to 2. | The quantity is rounded to the nearest decimals, so the quantity becomes 1. | Match |
| Trader attempts to sell Crypto more than they have. | Exception is thrown. | Match |
| Trader places an Order of negative Quantity. | Exception is thrown. | Match |
| Trader attempts to buy with a currency they do not own. | Exception is thrown. | Match |
| Trader cancels order. | The order is removed from the MatchinngEngine Priority Queue and the Order’s status is set to CANCELLED. | Match |
| Trader adds new Fiat Coin to his profile. | A new element is added to the fiats and fiatsBalances ArrayList. | Match |
| Traders adds an existing Fiat/Crypto Coin to his profile, (either through trade or transfer). | The element in the balances Array is updated to reflect the new balance. | Match |
| A trader does not have the funds to execute an order. | The Matching Engine does not match the Trader’s order. | Match |
| 2 Orders are matched. | The Orders’ remaining quantities are updated. The transaction of Fiat and Crypto Currencies occurs between the 2 Traders. Fulfilled Orders are removed from the Matching Engine. | Match |
| Limit Orders are placed. | Matching Engine checks if a LimitOrder’s condition has been met before attempting to match it with another order. | Match |
| Trader attempts to copy the OrderBook. | Value returned is a deep copy and cannot affect the original in any way. | Match |
| An Order’s remaining Quantity reaches 0. | The Order’s status is set to FULFILLED and removed from the MatchingEngine Priority Queue. | Match |
| User logs out | After logging out Users login flag is set to false and they cannot perform any actions. | Match |
| Order’s quantity remaining value changed. | The volume executed is recomputed and shows the percentage of the quantity of the Order still to be Traded. | Match |

## Critical Evaluation

My implementation makes use of object oriented features such as inheritance, polymorphism, encapsulation and abstraction in order to create the structure a real-world crypto exchange platform would use. The User and Coin class are abstract classes and both are super classes to the Admin and Trader classes, and the Fiat and Crypto classes respectively. This removes any repeated and redundant code which were to be present in the subclasses.

Moreover, variables protection levels are set accordingly. Many of the variables are set to protected and private. Protected variables are hidden from the Launcher file, but are still allowed access within the package for more complex functionality. All private variables, those worth accessing or not, have getter and setter functions. This is to allow further development to take places using these variables without implementing any extra functionality. Calling the setter for the private variable relating to an order’s remaining quantity also recomputes the volume executed variable. Hence the remaining quantity cannot be changed without also changing the volume executed variable.

There also exists the Error class. The aim of the creation of the error class was to setup the foundations of an error handling system. A real world application could add code to the already existing class in order to catch exceptions and take appropriate actions. Due to the scope of this assignment, currently, handling an error only prints the message to the error stream and exits the program.

The Matching Engine and OrderBook class are implemented as singleton classes. This is to ensure that there only exists one instance of each across all orders. I preferred singleton classes over static ones in order to allow these classes to be passed to other functions in future development, despite not being so right now.

## Audit Trails

A Logger Class can be created. This Logger class will keep a reference of any active trader in the platform. This can be done by adding a trader to the Logger in the trader’s constructor. The Logger can check if any trader is logged in, and if so can periodically serialise the instance of the Trader class into a log alongside the current time, every fixed interval. Changes to the Trader profile will result in discrepancies between serialisations in the logs. Moreover the serialized profiles can be reinstantiated and the scenario taking place may be reconstructed.

Orders which a trader may place are always kept in the Order Book. The OrderBook may be modified to also keep track of the current time which an order was put. This then can be synced with a Trader’s logs to see if he placed any orders.

The logs may be saved in the company’s storage and encrypted, in order to avoid tampering.

# Task 3