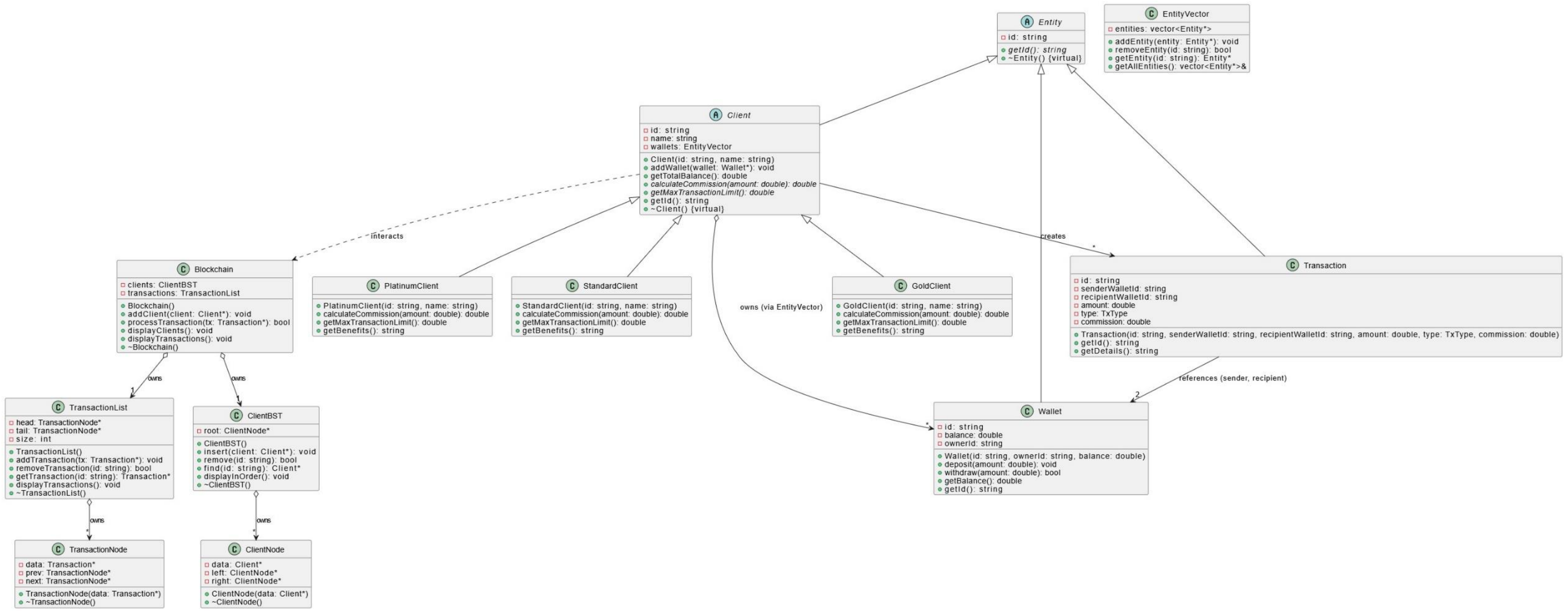
PROJECT OF END COURSE – LABORATORY OF FUNDATIONS OF ALGORITHMS AND PROGRAMMING

SIMPLIFIED SIMULATION OF A BLOCKCHAIN TRANSACTION SYSTEM

Professor: Holger Espinola Rivera

UML DIAGRAM:



Entity
- id: string
+ getId(): string
+ ~Entity() {virtual}
EntityVector
- entities: vector<Entity*>
+ addEntity(entity: Entity*): void
+ removeEntity(id: string): bool
+ getEntity(id: string): Entity*
+ getAllEntities(): vector<Entity*>&
Client
- id: string
- name: string
- wallets: EntityVector
+ Client(id: string, name: string)
+ addWallet(wallet: Wallet*): void
+ getTotalBalance(): double
+ calculateCommission(amount: double): double
+ getMaxTransactionLimit(): double
+ getId(): string
+ ~Client() {virtual}
interacts
creates
*
owns (via EntityVector)
Blockchain
- clients: ClientBST
- transactions: TransactionList
+ Blockchain()
+ addClient(client: Client*): void
+ processTransaction(tx: Transaction*): bool
+ displayClients(): void
+ displayTransactions(): void
+ ~Blockchain()
PlatinumClient
+ PlatinumClient(id: string, name: string)
+ calculateCommission(amount: double): double
+ getMaxTransactionLimit(): double
+ getBenefits(): string
StandardClient
+ StandardClient(id: string, name: string)
+ calculateCommission(amount: double): double
+ getMaxTransactionLimit(): double
+ getBenefits(): string
GoldClient
+ GoldClient(id: string, name: string)
+ calculateCommission(amount: double): double
+ getMaxTransactionLimit(): double
+ getBenefits(): string
Transaction
- id: string
- senderWalletId: string
- recipientWalletId: string
- amount: double
- type: TxType
- commission: double
+ Transaction(id: string, senderWalletId: string, recipientWalletId: string, amount: double, type: TxType, commission: double)
+ getId(): string
+ getDetails(): string
references (sender, recipient)
2
owns
1
owns
1
TransactionList
- head: TransactionNode*
- tail: TransactionNode*
- size: int
+ TransactionList()
+ addTransaction(tx: Transaction*): void
+ removeTransaction(id: string): bool
+ getTransaction(id: string): Transaction*
+ displayTransactions(): void
+ ~TransactionList()
ClientBST
- root: ClientNode*
+ ClientBST()
+ insert(client: Client*): void
+ remove(id: string): bool
+ find(id: string): Client*
+ displayInOrder(): void
+ ~ClientBST()
*
Wallet
- id: string
- balance: double
- ownerId: string
+ Wallet(id: string, ownerId: string, balance: double)
+ deposit(amount: double): void
+ withdraw(amount: double): bool
+ getBalance(): double
+ getId(): string
owns
*
owns
*
TransactionNode
- data: Transaction*
- prev: TransactionNode*
- next: TransactionNode*
+ TransactionNode(data: Transaction*)
+ ~TransactionNode()
ClientNode
- data: Client*
- left: ClientNode*
- right: ClientNode*
+ ClientNode(data: Client*)
+ ~ClientNode()

- **Description of Relationships:**

1. Inheritance:
   - Entity ← Client, Wallet, Transaction: Entities inherit from Entity for ID management.
   - Client ← GoldClient, PlatinumClient, StandardClient: Client types override commission and limit methods.
2. Composition:
   - Blockchain → ClientBST: Blockchain owns the ClientBST (1 to 1).
   - Blockchain → TransactionList: Blockchain owns the TransactionList (1 to 1).
   - ClientBST → ClientNode: ClientBST owns ClientNode objects (1 to *).
   - TransactionList → TransactionNode: TransactionList owns TransactionNode objects (1 to *).
3. Aggregation:
   - Client → Wallet: Client owns Wallets via EntityVector (1 to *), but Wallets could be reassigned.
4. Association:
   - Client ↔ Blockchain: Clients interact with Blockchain for transactions and queries (* to 1).
   - Client → Transaction: Clients create Transactions (1 to *).
   - Transaction → Wallet: Transactions reference sender and recipient wallets (1 to 2).

- **Notes:**

- Attributes: Private (-), Public (+).
- Multiplicity: 1 (one), * (many).
- TxType: Enum in Transaction (TRANSFER).
- **EntityVector** manages 1-to-many relationships.
- **ClientBST** orders clients by total balance.
- **TransactionList** uses a doubly-linked list for transactions.

- **Description of Classes, Attributes, and Methods**

Below are the simplified class specifications:

1. **Entity** (Abstract Base Class):
    - **Purpose**: Base class for entities (Client, Wallet, Transaction).
    - **Attributes**:
        - id: string (unique identifier)
    - **Methods**:
        - getId(): string (pure virtual)
        - ~Entity() (virtual destructor)
    - **Abstract**: Yes.
2. **EntityVector**:
    - **Purpose**: Manages a 1-to-many relationship for entities (e.g., wallets owned by a client).
    - **Attributes**:
        - entities: vector<Entity*> (stores entity pointers)
    - **Methods**:
        - addEntity(entity: Entity*): void
        - removeEntity(id: string): bool
        - getEntity(id: string): Entity*
        - getAllEntities(): vector<Entity*>&
3. **Client** (Abstract Base Class):
    - **Purpose**: Base class for client types.
    - **Attributes**:
        - id: string (inherited from Entity)
        - name: string
        - wallets: EntityVector (stores wallets)
    - **Methods**:
        - Client(id: string, name: string)
        - addWallet(wallet: Wallet*): void

        - getTotalBalance(): double (sum of wallet balances)
        - calculateCommission(amount: double): double (pure virtual)
        - getMaxTransactionLimit(): double (pure virtual)
        - getId(): string (override)
        - ~Client() (virtual destructor)

4. **GoldClient** (Inherits from Client):
    - **Purpose**: Premium clients with low commissions.
    - **Attributes**: (Inherited)
    - **Methods**:
        - GoldClient(id: string, name: string)
        - calculateCommission(amount: double): double (e.g., 1%)
        - getMaxTransactionLimit(): double (e.g., 10000)
        - getBenefits(): string (“Priority processing, low fees”)
5. **PlatinumClient** (Inherits from Client):
    - **Purpose**: Mid-tier clients with moderate commissions.
    - **Attributes**: (Inherited)
    - **Methods**:
        - PlatinumClient(id: string, name: string)
        - calculateCommission(amount: double): double (e.g., 2%)
        - getMaxTransactionLimit(): double (e.g., 5000)
        - getBenefits(): string (“Bonus rewards, moderate fees”)
6. **StandardClient** (Inherits from Client):
    - **Purpose**: Standard clients with higher commissions.
    - **Attributes**: (Inherited)
    - **Methods**:
        - StandardClient(id: string, name: string)
        - calculateCommission(amount: double): double (e.g., 5%)
        - getMaxTransactionLimit(): double (e.g., 1000)
        - getBenefits(): string (“Standard access”)
7. **Wallet** (Inherits from Entity):

- **Purpose**: Stores funds for transactions.
- **Attributes**:
  - id: string (inherited)
  - balance: double
  - ownerId: string (client ID)
- **Methods**:
  - Wallet(id: string, ownerId: string, balance: double)
  - deposit(amount: double): void
  - withdraw(amount: double): bool
  - getBalance(): double
  - getId(): string (override)

8. **Transaction** (Inherits from Entity):
   - **Purpose**: Represents a transfer between two wallets.
   - **Attributes**:
     - id: string (inherited)
     - senderWalletId: string (sender’s wallet ID)
     - recipientWalletId: string (recipient’s wallet ID)
     - amount: double
     - type: TxType (enum: {TRANSFER})
     - commission: double
   - **Methods**:
     - Transaction(id: string, senderWalletId: string, recipientWalletId: string, amount: double, type: TxType, commission: double)
     - getId(): string (override)
     - getDetails(): string
9. **TransactionNode**:
   - **Purpose**: Node for the doubly-linked list of transactions.
   - **Attributes**:
     - data: Transaction*
     - prev: TransactionNode*

        - next: TransactionNode*
    - **Methods**:
        - TransactionNode(data: Transaction*)
        - ~TransactionNode()
10. **TransactionList**:
    - **Purpose**: Doubly-linked list for transactions.
    - **Attributes**:
        - head: TransactionNode*
        - tail: TransactionNode*
        - size: int
    - **Methods**:
        - TransactionList()
        - addTransaction(tx: Transaction*): void
        - removeTransaction(id: string): bool
        - getTransaction(id: string): Transaction*
        - displayTransactions(): void
        - ~TransactionList()
11. **ClientNode**:
    - **Purpose**: Node for the binary search tree of clients.
    - **Attributes**:
        - data: Client*
        - left: ClientNode*
        - right: ClientNode*
    - **Methods**:
        - ClientNode(data: Client*)
        - ~ClientNode()
12. **ClientBST**:
    - **Purpose**: Binary search tree for clients, ordered by balance.
    - **Attributes**:
        - root: ClientNode*

- **Methods**:
  - ClientBST()
  - insert(client: Client*): void
  - remove(id: string): bool
  - find(id: string): Client*
  - displayInOrder(): void
  - ~ClientBST()

13. **Blockchain**:
    - **Purpose**: Manages clients and transactions.
    - **Attributes**:
      - clients: ClientBST
      - transactions: TransactionList
    - **Methods**:
      - Blockchain()
      - addClient(client: Client*): void
      - processTransaction(tx: Transaction*): bool
      - displayClients(): void
      - displayTransactions(): void
      - ~Blockchain()

- **Instructions**

Implement the system which simulates a creative and very simplified version of BlockChain financial transaction operations between wallets of Clients. The core of the system is the BlockChain, which contains information about the Clients and the transactions. It is necessary do operations of deposits and withdraws. Implement the system in C++ following OOP paradigm using classes, methods, constructors, destructors and relationships between classes like inheritance, association, aggregation or composition when it is necessary. Use the data structures of Binary Tree Search and Double Linked List. Prepare necessary methods which can make reports of List of Clients and the List of transactions contained in the BlockChain. If you want, you can add or simplify the ideas expressed in the UML diagram and this description (it is not necessary implement entire schema – all classes, attributes or methods, but the most critic ideas and functionalities are mandatory). If it is possible, implement some overload of operations in some classes for processes of assignation of new objects or comparison between 2 objects. Store the information of clients and the information of transactions in 2 files: **Clients.txt** and **Blockchain_transactions.txt** using functionalities to read and write in files through the functions **fopen, fscanf, fprintf** and **fclose** used in the lesson 09 of 1st semester in course Algorithmizing and Programming. The assignment should be completed in **groups of 4 students**. A report should be written following the instructions given by the theory teacher. The presentation will take place on the day of the laboratory evaluation. The project need be published in the **Github repository** of the leader of the group. In the day of the presentation of the Project of end course, is necessary come with the printed version of the report.

- **Commentaries**

- The implementation of **Entity Vector** can be based on data structure and algorithms defined in **lab02_q3**. Define one class for each entity and after that, implement some vectorize class of this entity.
- The implementation of classes **Client Node** and **Client BST** can be based on data structure and algorithms defined in **lab06_q1**. The code need have some adaptations to can works under objects of class Client. It is necessary choose some numerical attribute from class Client which can be useful to apply the invariant of the binary tree building process.
- The implementation of classes **Transaction Node** and **Transaction List** can be based on data structure and algorithms defined in **lab06_q3**. The same idea of Double Linked List for objects of class Circle can implement, respecting the proper structure and relationships that the **Transaction** class has.
- The implementation of functionalities read and write in files can be guided with some adaptations following the **week09** (Github repository **Algorithms-5130203**).