САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО

Институт компьютерных наук и кибербезопасности

Высшая школа технологий искусственного интеллекта

Направление: **02.03.03** Математическое обеспечение и администрирование информационных систем

**Отчёт о выполнении курсовой работы**

По дисциплине«Основы программирования и алгоритмизации»

«Упрощённая симуляция системы блокчейн-транзакций»

Студенты,

группа 5130203/40002 Сикорский А.В.

Слинкина В.Д.

Чигринский З.Н.

Преподаватель: Хольгер Э.Р.

Санкт-Петербург, 2025

Содержание

Введение4-5

Реализация классов и их методов6-29

1) Класс: Entity6

Виртуальный деструктор: Entity6

Чисто виртуальный метод: get\_id6

2) Класс: Wallet6-8

Конструктор: Wallet6

Метод: deposit7

Метод: withdraw7

Метод: get\_balance7

Метод: get\_id8

Метод: set\_owner\_id8

3) Класс: Client8-11

Конструктор: Client8

Метод: addWallet8

Метод: getTotalBalance8

Чисто виртуальный метод: calculateCommission9

Чисто виртуальный метод: getMaxTransactionLimit9

Чисто виртуальный метод: getBenefits10

Метод: get\_name10

Метод: get\_type10

Метод: get\_wallet\_at10

Метод: get\_max\_n\_wallets11

Деструктор: Client11

4) Дочерние классы Client12

4.1) Класс: StandartClient и его методы12

4.2) Класс: PlatinumClient и его методы12

4.3) Класс: GoldClient и его методы12

5) Класс: Transaction13-14

Конструктор: Transaction13

Метод: get\_id13

Метод: get\_details13

Метод: get\_type14

6) Класс: ClientNode14-15

Конструктор: ClientNode14

Деструктор: ClientNode15

7) Класс: СleintBST15-19

Метод: insertNode15

Метод: removeNode15

Метод: findNode17

Метод: showTree17

Конструктор: ClientBST18

Метод: insert18

Метод: remove18

Метод: find19

Метод: displayInOrder19

Деструктор: ClientBST19

8) Класс: TransactionNode19-20

Конструктор: TransactionNode20

Деструктор: TransactionNode20

9) Класс: TransactionList20-22

Конструктор: TransactionList20

Деструктор: TransactionList20

Метод: addTransaction21

Метод: removeTransaction21

Метод: getTransaction22

Метод: displayTransactions22

10) Класс: Blockchain22-29

Метод: writeClientNode23

Метод: writeTransactionList23

Конструктор: Blockchain24

Метод: addClient27

Метод: processTransaction27

Метод: displayClients29

Метод: displayTransactions29

Деструктор: Blockchain29

Описание тестирования разработанного решения30-37

Заключение38

**Введение**

В отчёте представлено описание выполнения курсовой работы по дисциплине “Основы программирования и алгоритмизации”. Работа представляет из себя код на языке C++ в объектно-ориентированном стиле, реализацию системы, которая имитирует упрощенную версию блокчейн-транзакций между кошельками клиентов, где ядром системы является Blockchain, содержащий информацию о клиентах и транзакциях.

**Главная цель** – разработать упрощённую имитацию системы блокчейн-транзакций между клиентами на языке C++, реализующую основные принципы объектно-ориентированного программирования и работы с файлами.

Согласно главной цели, в работе необходимо выполнить несколько задач, чтобы система могла обеспечивать:

* Управление клиентами и их кошельками;
* Выполнение финансовых операций (депозиты, снятия, переводы);
* Хранение данных в структурах (двусвязный список для транзакций, двоичное дерево для клиентов);
* Сохранение и загрузку данных в файлы;

Определим основные термины, используемые в отчёте:

*Блокчейн* – децентрализованная цифровая система учёта транзакций, в которой данные хранятся в виде цепочки блоков.

*Виртуальный метод* – метод класса, который может быть переопределён в дочерних классах.

*Двоичное дерево клиентов (двоичное дерево, дерево) –* структура данных для хранения и поиска клиентов по id, состоящая из связанных между собой узлов, в которой каждый из них имеет не более двух потомков.

*Двусвязный список транзакций (двусвязный список, список)* – структура данных для хранения транзакций, где каждый узел содержит ссылки на предыдущий и следующий элементы.

*Деструктор* – метод класса, вызываемый при уничтожении объекта и освобождающий ресурсы.

*Дочерний класс* – это класс, который наследует свойства и методы от базового класса

*Инициализация* – присвоение начального значения полям объекта при его создании.

*Клиент* – пользователь системы, владеющий одним или несколькими кошельками.

*Конструктор* – метод класса, вызываемый при создании объекта и инициализирующий поля объекта.

*Кошелёк* – виртуальный счёт клиента для хранения средств.

*Поддерево* – узел двоичного дерева и всего его потомки.

*Поля объекта (поля)* – это переменные, которые определяют характеристики или свойства конкретного объекта.

*Потомок узла (потомок)* – узел двоичного дерева, находящийся ниже этого узла по иерархии.

*Рекурсивный обход (рекурсивно)* – метод обхода структуры данных, который подразумевает обработку узла, а затем обработку всех его потомков.

*Транзакция* – операция перевода средств.

*Узел* – отдельный элемент с данными и ссылками, базовая единица двоичного дерева и двусвязного списка.

*Чисто виртуальный метод* – метод класса, реализация которого отсутствует в базовом классе, который должен быть реализован в дочернем классе.

**Реализация классов и их методов**

**1) Класс: Entity**

Базовый абстрактный класс для всех сущностей системы.

Атрибут класса: id (string) – уникальный идентификатор сущности.

**Методы класса:**

virtual ~Entity() – виртуальный деструктор.

Параметры и возвращаемое значение: -

Назначение: Корректное удаление объектов дочерних классов.

Реализация: Объявлен виртуальным в базовом классе, переопределяется в наследниках.

Entity::~Entity(){}

virtual string get\_id() = 0 – чисто виртуальный метод для получения id, реализованный в дочерних классах.

Параметры и возвращаемое значение: -

Назначение: Переопределение метода в дальнейшем у дочерних классов.

**2) Класс: Wallet**

Дочерний класс Entity, реализующий функционал кошелька.

**Методы класса:**

Wallet(string id, double balance, string owner\_id) – конструктор по параметрам.

Параметры:

id (string) – id кошелька

balance (double) – начальный баланс

owner\_id (string) – id владельца

Возвращаемое значение: -

Назначение: Создание кошелька с указанными параметрами.

Реализация: Наследует id от Entity, инициализирует balance и owner\_id.

Wallet::Wallet (string id, double balance, string owner\_id):

    Entity(id){

        this->balance = balance;

        this->owner\_id = owner\_id;}

deposit(double amount)

Параметры:

amount (double) – сумма для пополнения

Возвращаемое значение: -

Назначение: Пополнение баланса кошелька.

Реализация: Проверяет, что amount > 0, если это верно – увеличивает баланс кошелька на указанную сумму, иначе – вывод сообщения об ошибке.

void Wallet::deposit (double amount){

    if (amount > 0) {

        this->balance += amount;

    }

    else{

        cout<<"Incorrect amount of deposit"<<endl;

    }

}

withdraw(double amount)

Параметры:

amount (double) – сумма для снятия

Возвращаемое значение: bool (совершена операция или нет)

Назначение: Снятие средств с кошелька.

Реализация: Проверяет, что amount > 0 и баланс достаточен, если это верно – уменьшает баланс кошелька на указанную сумму, иначе – вывод сообщения об ошибке.

bool Wallet::withdraw (double amount){

    if (amount > 0 && this->balance >= amount) {

        this->balance -= amount;

        return true;

    }

    cout<<"Error withdraw"<<endl;

    return false;

}

get\_balance()

Параметры: -

Возвращаемое значение: double

Назначение: Возвращает текущий баланс кошелька.

double Wallet::get\_balance(){

    return this->balance;

}

get\_id()

Параметры: -

Возвращаемое значение: string

Назначение: Возвращает id кошелька.

Реализация:

string Wallet::get\_id(){

    return this->id;

}

set\_owner\_id(string id)

Параметры:

id (string) – id клиента-владельца кошелька

Возвращаемое значение: string

Назначение: Присваивает кошельку id его владельца.

Реализация:

void set\_owner\_id(string id) { owner\_id = id; };

**3) Класс: Client**

Дочерний класс Entity, абстрактный базовый класс для всех типов клиентов, реализует функционал клиента.

**Методы класса:**

Client(string id, string name) – конструктор по параметрам.

Параметры:

id (string) – id клиента

name (string) – имя клиента

Возвращаемое значение: -

Назначение: Создание клиента с указанными параметрами.

Реализация: Наследует id от Entity, инициализирует name.

Client::Client(string id, string name):

    Entity(id) {

    this->name = name;

}

addWallet(Wallet\* wallet)

Параметры:

wallet (Wallet\*) – кошелек для добавления

Возвращаемое значение: -

Назначение: Добавление кошелька в список кошельков клиента.

Реализация: Добавляет указатель на кошелек в вектор vWallets.

void Client::addWallet(Wallet\* wallet) {

    vWallets.push\_back(wallet);

}

getTotalBalance()

Параметры: -

Возвращаемое значение: double – суммарный баланс всех кошельков.

Назначение: Возвращает общий баланс клиента.

Реализация: Суммирует балансы всех кошельков из vWallets через цикл, от первого к последнему кошельку.

double Client::getTotalBalance() {

    double total\_balance = 0.0;

    for (int i = 0; i < vWallets.size(); i++) {

        total\_balance += vWallets.at(i)->get\_balance();

    }

    return total\_balance;

}

* 3 следующих метода будут реализованы в дочерних классах Client, но их назначение, параметры и возвращаемое значение будут идентичны.

virtual double calculateCommission(double amount) = 0

Назначение: Рассчитывает комиссию для транзакции в зависимости от типа клиента. Позволяет каждому дочернему классу реализовать свои правила расчёта.

Параметры:

amount (double) – сумма транзакции

Возвращаемое значение: double – размер комиссии.

virtual double getMaxTransactionLimit() = 0

Назначение: Возвращает максимально допустимую сумму для одной транзакции. Ограничения зависят от типа клиента.

Параметры: -

Возвращаемое значение: double – лимит транзакции.

virtual string getBenefits() = 0

Назначение: Возвращает строку с описанием привилегий клиента.

Параметры: -

Возвращаемое значение: string – описание benefits.

get\_id()

Параметры: -

Возвращаемое значение: string

Назначение: Возвращает id клиента.

Реализация:

string get\_id() { return id; };

get\_name()

Параметры: -

Возвращаемое значение: string

Назначение: Возвращает имя клиента.

Реализация:

string get\_name() { return name; };

get\_type()

Параметры: -

Возвращаемое значение: char

Назначение: Возвращает символьный идентификатор типа клиента.

Реализация:

    char get\_type() {

        switch (type)

        {

        case CLIENTTYPE\_STANDARD: return 's'; break;

        case CLIENTTYPE\_PLATINUM: return 'p'; break;

        case CLIENTTYPE\_GOLD: return 'g'; break;

        default: return 'n'; break;

        }

    }

get\_wallet\_at(int n)

Параметры:

n (int) – индекс кошелька в векторе vWallets

Возвращаемое значение: Wallet\*

Назначение: Возвращает указатель на кошелек клиента по индексу.

Реализация:

Wallet\* get\_wallet\_at(int n) { return (n >= 0 && (n < vWallets.size()) ? vWallets[n] : nullptr;  };

get\_max\_n\_wallets()

Параметры: -

Возвращаемое значение: int – размер вектора vWallets

Назначение: Возвращает текущее количество кошельков клиента.

Реализация:

int get\_max\_n\_wallets() { return static\_cast<int>(vWallets.size()); };

~Client() –деструктор.

Параметры и возвращаемое значение: -

Назначение: Удаление объектов класса.

Реализация:

Client::~Client(){}

**4) Дочерние классы Client**

* Методы каждого дочернего класса включают в себя конструктор по параметрам, наследуемым от Client, где также явно устанавливается тип клиента, и реализацию трёх чисто виртуальных методов из класса Client. Типы клиентов в enum объявлены в Client.h.

enum ClientType {

    CLIENTTYPE\_NONE,

    CLIENTTYPE\_STANDARD,

    CLIENTTYPE\_PLATINUM,

    CLIENTTYPE\_GOLD

};

**4.1) Класс: StandartClient**

Дочерний класс Client, реализующий функционал клиента с наименее выгодными условиями обслуживания, среди всех остальных.

**Методы класса (см. Чисто виртуальные методы класса Client):**

Реализация: Устанавливает комиссию клиента на 5% и лимит на транзакции в 1000.

class StandardClient: public Client {

public:

    StandardClient(string id, string name) : Client(id, name) { this->type = CLIENTTYPE\_STANDARD; };

    virtual double calculateCommission(double amount) { return amount \* 0.05; };

    virtual double getMaxTransactionLimit() { return 1000; };

    virtual string getBenefits() { return "Standard access"; };

};

**4.2) Класс: PlatinumClient**

Дочерний класс Client, реализующий функционал клиента с условиями более выгодными, чем у StandardClient.

**Методы класса (см. Чисто виртуальные методы класса Client):**

Реализация: Устанавливает комиссию клиента на 2% и лимит на транзакции в 5000.

class PlatinumClient : public Client {

public:

    PlatinumClient(string id, string name) : Client(id, name) { };

    virtual double calculateCommission(double amount) { return amount \* 0.02; };

    virtual double getMaxTransactionLimit() { return 5000; };

    virtual string getBenefits() { return "Bonus rewards, moderate fees"; };

};

**4.3) Класс: GoldClient**

Дочерний класс Client, реализующий функционал клиента с наиболее выгодными условиями обслуживания, среди всех остальных.

**Методы класса (см. Чисто виртуальные методы класса Client):**

Реализация: Устанавливает комиссию клиента на 1% и лимит на транзакции в 10000.

class GoldClient : public Client {

public:

    GoldClient(string id, string name) : Client(id, name) { this->type = CLIENTTYPE\_GOLD; };

    virtual double calculateCommission(double amount) { return amount \* 0.01; };

    virtual double getMaxTransactionLimit() { return 10000; };

    virtual string getBenefits() { return "Priority processing, low fees"; };

};

**5) Класс: Transaction**

Дочерний класс Entity, реализующий функционал транзакции.

**Методы класса:**

Transaction(string id, string senderWalletId, string recipientWalletId, double amount, TxType type, double commission) – конструктор по параметрам.

Параметры:

id (string) – id транзакции

senderWalletId, recipientWalletId (string) – id кошельков отправителя и получателя средств соответственно

amount (double) – сумма

type (TxType) – тип транзакции

commission (double) – комиссия

Возвращаемое значение: -

Назначение: Создание транзакции с указанными параметрами.

Реализация: Инициализирует все поля класса.

Transaction::Transaction(string id, string senderWalletId, string recipientWalletId, double amount, TxType type, double commission):

    Entity(id){

        this->senderWalletId=senderWalletId;

        this->recipientWalletId=recipientWalletId;

        this->amount=amount;

        this->type=type;

        this->commission=commission;

    }

get\_id()

Параметры: -

Возвращаемое значение: string

Назначение: Возвращает id транзакции.

Реализация:

string Transaction::get\_id(){

    return this->id;

}

getDetails()

Параметры: -

Возвращаемое значение: string (описание транзакции)

Назначение: Возвращает детали транзакции в читаемом виде.

Реализация: Использует switch для преобразования TxType в строку.

string Transaction::getDetails(){

    string typeStr;

    switch (type) {

        case TxType::TRANSFER:   typeStr = "TRANSFER"; break;

        case TxType::DEPOSIT:    typeStr = "DEPOSIT"; break;

        case TxType::WITHDRAWAL: typeStr = "WITHDRAWAL";

    }

    return "Transaction ID: " + id +

            "\nSender Wallet ID: " + senderWalletId +

            "\nReceiver Wallet ID: " + recipientWalletId +

            "\nAmount: " + to\_string(amount) +

            "\nType: " + typeStr +

            "\nCommission: " + to\_string(commission);

}

get\_type()

Параметры: -

Возвращаемое значение: char

Назначение: Возвращает символьный идентификатор типа транзакции.

Реализация:

char get\_type() {

        switch (type)

        {case TxType::TRANSFER: return 't'; break;

        case TxType::DEPOSIT: return 'd'; break;

        case TxType::WITHDRAWAL: return 'w'; break;

        default: break;}

    };

**6) Класс: ClientNode**

Узел двоичного дерева поиска для хранения клиентов.

**Методы класса:**

ClientNode(Client\* data) – конструктор.

Параметры:

data (Client\*) – хранимый объект клиента

Возвращаемое значение: -

Назначение: Создание узла двоичного дерева с указанными данными клиента.

Реализация: Инициализирует поле data переданным значением. Устанавливает указатели left и right в nullptr.

ClientNode(Client\* data) { this->data = data; left = nullptr; right = nullptr; };

~ClientNode() – деструктор.

Параметры и возвращаемое значение: -

Назначение: Уничтожение узла двоичного дерева.

Реализация: Обнуляет указатели на узлы, связанные с удаляемым узлом. Не удаляет сам объект Client (управляется классом ClientBST).

~ClientNode() { left = nullptr; right = nullptr; };

**7) Класс: ClientBST**

Двоичное дерево поиска для хранения и управления клиентами.

**Методы класса:**

insertNode(ClientNode\*& current, Client\* client)

Параметры:

current (ClientNode\*&) – текущий узел для рекурсивного обхода

client (Client\*) – клиент для добавления

Возвращаемое значение: -

Назначение: Рекурсивная вставка узла.

Реализация: Если текущий узел пуст – создаёт новый. Если id меньше – идёт в левое поддерево. Если id больше – идёт в правое поддерево. При совпадении id – выводит предупреждение.

void ClientBST::insertNode(ClientNode\*& current, Client\* client) {

    if (current == nullptr) {

        ClientNode\* newNode = new ClientNode(client);

        current = newNode;

    }

    else if (client->get\_id() < current->data->get\_id()) {

        insertNode(current->left, client);

    }

    else if (client->get\_id() > current->data->get\_id()) {

        insertNode(current->right, client);

    }

    else {

        cout << "Client is already in the tree" << endl;

    }

}

removeNode(ClientNode\*& current, string id)

Параметры:

current (ClientNode\*&) – текущий узел для рекурсивного обхода

id (string) – id клиента для удаления

Возвращаемое значение: ClientNode\* - указатель на изменённое поддерево.

Назначение: Рекурсивное удаление узла.

Реализация: Для узла без потомков удаляет узел. Для узла с одним потомком заменяет узел на потомка. Для узла с двумя потомками находит минимальный элемент в правом поддереве копирует его данные и рекурсивно удаляет его.

ClientNode\* ClientBST::removeNode(ClientNode\*& current, string id) {

    if (current == nullptr) {

        return nullptr;

    }

    if (current->data->get\_id() == id) {

        if (current->left == nullptr && current->right == nullptr) {

            delete current;

            return nullptr;

        }

        else if (current->left != nullptr && current->right == nullptr) {

            ClientNode\* orphan = current->left;

            delete current;

            return orphan;

        }

        else if (current->right != nullptr && current->left == nullptr) {

            ClientNode\* orphan = current->right;

            delete current;

            return orphan;

        }

        else {

            if (current->right->left == nullptr) {

                current->data = current->right->data;

                current->right = current->right->right;

            }

            else {

                ClientNode\* lefmostNode = current->right;

                while (lefmostNode->left != nullptr) {

                    lefmostNode = lefmostNode->left;

                }

                current->data = lefmostNode->data;

                removeNode(current->right, lefmostNode->data->get\_id());

            }

            return current;

        }

    }

    else if (id < current->data->get\_id() && current->left != nullptr) {

        current->left = removeNode(current->left, id);

        return current;

    }

    else if (id > current->data->get\_id() && current->right != nullptr) {

        current->right = removeNode(current->right, id);

        return current;

    }

    else {

        cout << "No such node!" << endl;

        return current;

    }

    return current;

}

findNode(ClientNode\* current, string id)

Параметры:

current (ClientNode\*) – текущий узел для рекурсивного обхода

id (string) – id искомого клиента

Возвращаемое значение: ClientNode\* (или nullptr) - указатель на найденный узел.

Назначение: Рекурсивный поиск узла по id.

Реализация: Сравнивает id с текущим узлом. Рекурсивно идёт в левый/правый узел.

ClientNode\* ClientBST::findNode(ClientNode\* current, string id) {

    if (current == nullptr) {

        return nullptr;

    }

    else if (current->data->get\_id() == id) {

        return current;

    }

    else if (id < current->data->get\_id()) {

        return findNode(current->left, id);

    }

    else {

        return findNode(current->right, id);

    }

}

showTree(const ClientNode\* current)

Параметры:

current (const ClientNode\*) – текущий узел для рекурсивного обхода

Возвращаемое значение: -

Назначение: Рекурсивный вывод двоичного дерева, отсортированного по возрастанию id.

Реализация: Для каждого узла дерева выводит id и имя.

void ClientBST::showTree(const ClientNode\* current) {

    if (current == nullptr)

        return;

    showTree(current->left);

    cout << current->data->get\_id() << '\t' << current->data->get\_name() << '\n';

    showTree(current->right);

}

ClientBST() – конструктор по умолчанию.

Параметры и возвращаемое значение: -

Назначение: Инициализация пустого двоичного дерева поиска.

Реализация: Устанавливает root в nullptr.

ClientBST() { root = nullptr; };

insert(Client\* client)

Параметры:

client (Client\*) – клиент для добавления в дерево.

Возвращаемое значение: -

Назначение: Добавление нового клиента в двоичное дерево поиска, сохраняя упорядоченность по id.

Реализация: Вызывает приватный метод insertNode.

void insert(Client\* client) { insertNode(root, client); };

remove(string id)

Параметры:

id (string) – id клиента для удаления.

Возвращаемое значение: bool – true, если клиент найден и удален; false, если клиент не найден.

Назначение: Удаление клиента с указанным id из дерева, сохраняя его свойства.

Реализация: Использует приватный метод findNode для проверки – существует ли такой узел, если да – удаляет его при помощи метода removeNode.

bool ClientBST::remove(string id) {

    if (findNode(root, id) != nullptr) {

        removeNode(root, id);

        return true;

    }

    else {

        return false;

    }

}

find(string id)

Параметры:

id (string) – id искомого клиента

Возвращаемое значение: Client\* (или nullptr) – указатель на найденного клиента (нулевой указатель, если не найден)

Назначение: Осуществление поиска клиента по id в дереве.

Реализация: Вызывает приватный метод findNode.

Client\* ClientBST::find(string id) {

    ClientNode\* p = findNode(root, id);

    if (p != nullptr) {

        return p->data;

    }

    else {

        return NULL;

    }

}

displayInOrder()

Параметры и возвращаемое значение: -

Назначение: Вывод списка всех клиентов в отсортированном порядке (по возрастанию id).

Реализация: Вызывает приватный метод showTree.

    void displayInOrder() { return showTree(root); };

~ClientBST() – деструктор.

Параметры и возвращаемое значение: -

Назначение: Удаление объектов класса

Реализация:

~ClientBST() {};

**8) Класс: TransactionNode**

Узел двусвязного списка для хранения транзакций.

**Методы класса:**

TransactionNode(Transaction\* tx) – конструктор

Параметры:

tx (указатель на Transaction) – транзакция для хранения в узле.

Возвращаемое значение: -

Назначение: Создание узла списка с заданной транзакцией.

Реализация: Инициализирует поле data переданной транзакцией. Устанавливает указатели prev и next в nullptr.

TransactionNode::TransactionNode(Transaction\* tx):

    data(tx), prev(nullptr), next(nullptr){};

~TransactionNode() – деструктор

Параметры и возвращаемое значение: -

Назначение: Освобождение ресурсов узла.

Реализация:

TransactionNode::~TransactionNode(){};

**9) Класс: TransactionList**

Двусвязный список для хранения и управления транзакциями.

**Методы класса:**

TransactionList() – конструктор по умолчанию

Параметры и возвращаемое значение: -

Назначение: Инициализация пустого списка транзакций.

Реализация: Устанавливает head, tail в nullptr. Инициализирует size нулём.

TransactionList::TransactionList():

    head(nullptr), tail(nullptr), size(0) {}

~TransactionList()

Параметры и возвращаемое значение: -

Назначение: Освобождение памяти, занимаемой списком и транзакциями.

Реализация: Последовательно удаляет все узлы списка и их данные (т.е. транзакции).

TransactionList::~TransactionList() {

    TransactionNode\* current = head;

    while (current) {

        TransactionNode\* next = current->next;

        delete current->data;

        delete current;

        current = next;

    }

}

addTransaction(Transaction\* tx)

Параметры:

tx (указатель на Transaction) – транзакция для добавления.

Возвращаемое значение: -

Назначение: Добавление транзакции в конец списка.

Реализация: Создаёт новый узел. Если список пуст, устанавливает head и tail на новый узел. Иначе связывает новый узел с tail и обновляет tail.

void TransactionList::addTransaction(Transaction\* tx) {

    TransactionNode\* newNode = new TransactionNode(tx);

    if (!head) {

        head = tail = newNode;

    } else {

        tail->next = newNode;

        newNode->prev = tail;

        tail = newNode;

    }

    size++;

}

removeTransaction(string id)

Параметры:

id (строка) – id транзакции для удаления.

Возвращаемое значение: bool (совершено удаление или нет).

Назначение: Удаляет транзакцию с указанным id из списка.

Реализация: Находит узел с заданным id. Связывает соседние узлы, минуя удаляемый. Обновляет head или tail, если удаляется первый или последний узел.

bool TransactionList::removeTransaction(string id) {

    TransactionNode\* current = head;

    while (current) {

        if (current->data->get\_id() == id) {

            if (current->prev) current->prev->next = current->next;

            if (current->next) current->next->prev = current->prev;

            if (current == head) head = current->next;

            if (current == tail) tail = current->prev;

            delete current;

            size--;

            return true;

        }

        current = current->next;

    }

    return false;

}

getTransaction(string id)

Параметры:

id (string) – id транзакции для поиска.

Возвращаемое значение: Transaction\* (найденная транзакция или nullptr).

Назначение: Возвращает транзакцию с указанным id.

Реализация: Линейный поиск по списку.

Transaction\* TransactionList::getTransaction(string id) {

    TransactionNode\* current = head;

    while (current) {

        if (current->data->get\_id() == id) {

            return current->data;

        }

        current = current->next;

    }

    return nullptr;

}

displayTransactions()

Параметры и возвращаемое значение: -

Назначение: Вывод деталей всех транзакций в списке.

Реализация: Выводит результат getDetails() для каждой транзакции из списка.

void TransactionList::displayTransactions(){

    TransactionNode\* current = head;

    while (current) {

        std::cout << current->data->getDetails() << endl << "-----------------" << endl;

        current = current->next;

    }

}

**10) Класс: Blockchain**

Класс, реализующий блокчейн-систему для управления клиентами и их транзакциями. Обеспечивает:

* Хранение клиентов в двоичном дереве поиска (посредством **ClientBST**)
* Обработку транзакций (посредством **TransactionList**)
* Сохранение и загрузку данных в файл
* Валидацию транзакций перед добавлением

**Методы класса:**

writeClientNode(ClientNode\* currentNode, string path)

Параметры:

currentNode (ClientNode\*) – указатель на текущий узел дерева клиентов

path (string) – путь к файлу для сохранения

Возвращаемое значение: -

Назначение: Рекурсивно записывает данные клиента и его кошельков в файл.

Реализация: Открывает файл в режиме добавления (ios\_base::app). Сохраняет id, имя и тип клиента. Сохраняет все кошельки клиента. Рекурсивно обрабатывает левое и правое поддеревья.

void Blockchain::writeClientNode(ClientNode\* currentNode, string path) {

    ofstream out;

    out.open(savefile, ios\_base::app);

    Client\* currentClient = currentNode->data;

    out << currentClient->get\_id() << 'i' << endl;

    out << currentClient->get\_name() << 'n' << endl;

    out << currentClient->get\_type() << 't' << endl;

    for (int i = 0; i < currentClient->get\_max\_n\_wallets(); i++) {

        Wallet\* wal = currentClient->get\_wallet\_at(i);

        out << wal->id << 'w' << endl;

        out << wal->balance << 'b' << endl;

    }

    out << endl;

    if (currentNode->left != nullptr)

        writeClientNode(currentNode->left, path);

    if (currentNode->right != nullptr)

        writeClientNode(currentNode->right, path);

}

writeTransactionList(TransactionNode\* currentNode, string path)

Параметры:

currentNode (TransactionNode\*) – текущий узел списка транзакций

string path - путь к файлу для сохранения (не используется, вместо этого используется savefileTransactions)

Возвращаемое значение: -

Назначение: Рекурсивно записывает все транзакции из связанного списка в файл.

Реализация: Открывает файл savefileTransactions в режиме добавления (app). Если текущий узел существует, то получает данные транзакции и записывает в файл все поля транзакции символами с соответствующими обозначениями: i – id транзакции, s – id кошелька отправителя, r – id кошелька получателя, a – сумма, t – тип транзакции, c – комиссия. Закрывает файл. Рекурсивно вызывает себя для следующего узла (если он существует).

void Blockchain::writeTransactionList(TransactionNode\* currentNode, string path)

{

    ofstream out;

    out.open(savefileClients, ios\_base::app);

    if (currentNode != nullptr) {

        Transaction\* currentTransact = currentNode->data;

        out << currentTransact->get\_id() << 'i' << endl;

        out << currentTransact->senderWalletId << 's' << endl;

        out << currentTransact->recipientWalletId << 'r' << endl;

        out << currentTransact->amount << 'a' << endl;

        out << currentTransact->get\_type() << 't' << endl;

        out << currentTransact->commission << 'c' << endl;

        out << endl;

    }

    out.close();

    if (currentNode->next != nullptr)

        writeTransactionList(currentNode->next, path);

}

Blockchain(string cpath, string tpath) – конструктор

Параметры:

cpath (тип string) — путь к файлу клиентов (по умолчанию “c.txt”)

tpath (тип string) — путь к файлу транзакций (по умолчанию “t.txt”)

Возвращаемое значение: -

Назначение: Инициализирует объект Blockchain и загружает данные клиентов и транзакций из файла.

Реализация: Сохраняет пути к файлам в savefileClients и savefileTransactions. Загружает клиентов; читая файл построчно, создаёт объекты StandardClient, PlatinumClient или GoldClient на основе маркеров. Добавляет кошельки к клиентам. Загружает транзакции; Создаёт объекты Transaction с типами TRANSFER, DEPOSIT или WITHDRAWAL. Обрабатывает их через processTransaction.

.h

Blockchain(string cpath = "c.txt", string tpath = "t.txt");

.cpp

Blockchain::Blockchain(string cpath, string tpath) {

    savefileClients = cpath;

    savefileTransactions = tpath;

    ifstream inc;

    inc.open(savefileClients);

    if (inc.is\_open()) {

        string s = "";

        Client\* lastClient = nullptr;

        string setup\_id = "";

        string setup\_name = "";

        Wallet\* lastWallet = nullptr;

        string setup\_wallet\_id = "-";

        while (getline(inc, s)) {

            if (s.size() < 1) {

                this->addClient(lastClient);

                continue;

            }

            char si = s.back(); // string identifier

            s.pop\_back();

            switch (si)

            {

            case 'i':       // client id

                setup\_id = s;

                break;

            case 'n':       // client name

                setup\_name = s;

                break;

            case 't':       // client type

                if (s == "s") {

                    lastClient = new StandardClient(setup\_id, setup\_name);

                }

                else if (s == "p") {

                    lastClient = new PlatinumClient(setup\_id, setup\_name);

                }

                else if (s == "g") {

                    lastClient = new GoldClient(setup\_id, setup\_name);

                }

                break;

            case 'w':       // wallet id

                setup\_wallet\_id = s;

                break;

            case 'b':       // wallet balance + creation of wallet

                lastWallet = new Wallet(setup\_wallet\_id, stod(s), "-");

                lastClient->addWallet(lastWallet);

                break;

            default:

                break;

            }

        }

    }

    ifstream intr;

    intr.open(savefileTransactions);

    if (intr.is\_open()) {

        string s = "";

        string setup\_id = "";

        string setup\_sender\_w\_id = "";

        string setup\_recipient\_w\_id = "";

        double setup\_amount = 0.0;

        double setup\_commission = 0.0;

        while (getline(intr, s)) {

            if (s.size() < 1) {

                continue;

            }

            char si = s.back(); // string identifier

            s.pop\_back();

            switch (si)

            {

            case 'i':       // client id

                setup\_id = s;

                break;

            case 's':       // sender wallet id

                setup\_sender\_w\_id = s;

                break;

            case 'r':       // recipient wallet id

                setup\_recipient\_w\_id = s;

                break;

            case 'a':       // amount

                setup\_amount = stod(s);

                break;

            case 'c':       // commission

                setup\_commission = stod(s);

                break;

            case 't':       // client type

                if (s == "t") {

                    this->processTransaction(new Transaction(setup\_id, setup\_sender\_w\_id, setup\_recipient\_w\_id, setup\_amount, TxType::TRANSFER, setup\_commission));

                }

                else if (s == "d") {

                    this->processTransaction(new Transaction(setup\_id, setup\_sender\_w\_id, setup\_recipient\_w\_id, setup\_amount, TxType::DEPOSIT, setup\_commission));

                }

                else if (s == "w") {

                    this->processTransaction(new Transaction(setup\_id, setup\_sender\_w\_id, setup\_recipient\_w\_id, setup\_amount, TxType::WITHDRAWAL, setup\_commission));

                }

                break;

            default:

                break;

            }

        }

    }

addClient(Client\* client)

Параметры:

client (Client\*) – указатель на добавляемого клиента

Возвращаемое значение: -

Назначение: Добавляет клиента в двоичное дерево.

Реализация: Вызывает метод insert у объекта clients (метод класса **ClientBST**)

void Blockchain::addClient(Client\* client) {

    clients.insert(client);

}

processTransaction(Transaction\* tx)

Параметры:

tx: Указатель на обрабатываемую транзакцию

Возвращаемое значение: bool (успех/неудача)

Назначение: Валидирует и обрабатывает транзакцию.

Реализация: Проверяет корректность суммы и комиссии. Проверяет лимиты транзакций для отправителя. Обрабатывает по типу: TRANSFER: Проверяет существование получателя, DEPOSIT/WITHDRAWAL: Проверяет совпадение кошельков. Добавляет валидную транзакцию в список.

bool Blockchain::processTransaction(Transaction\* tx) {

    if (tx->amount <= 0 || tx->commission <= 0) {

        return false;

    }

    if (clients.find(tx->senderWalletId) != nullptr &&

        (tx->amount > clients.find(tx->senderWalletId)->getMaxTransactionLimit())){

        return false;

    }

    if (tx->commission <= 0) {

        return false;

    }

    if (tx->type != TxType::DEPOSIT && tx->commission >= tx->amount) {

        return false;

    }

        switch (tx->type)

    {

case TxType::TRANSFER:

    if (tx->senderWalletId != tx->recipientWalletId) {

        Client\* sender = nullptr;

        for (int i = 0; i < clients.root->data->get\_max\_n\_wallets(); i++) {

            if (clients.root->data->get\_wallet\_at(i)->get\_id() == tx->senderWalletId) {

                sender = clients.root->data;

                break;

            }

        }

        if (sender) {

            transactions.addTransaction(tx);

            ofstream out(savefileTransactions, ios::app);

            out << tx->get\_id() << 'i' << endl

                << tx->senderWalletId << 's' << endl

                << tx->recipientWalletId << 'r' << endl

                << tx->amount << 'a' << endl

                << tx->commission << 'c' << endl

                << static\_cast<char>(tx->type) << 't' << endl << endl;

            out.close();

            return true;

        }

    }

    return false;

    break;

    case TxType::DEPOSIT:

        if (tx->senderWalletId == tx->recipientWalletId) {

            transactions.addTransaction(tx);

            return true;

        }

        else {

            return false;

        }

        break;

    case TxType::WITHDRAWAL:

        if (tx->senderWalletId == tx->recipientWalletId) {

            transactions.addTransaction(tx);

            return true;

        }

        else {

            return false;

        }

        break;

    default:

        cout << "ERROR" << endl;

        return false;

        break;

    }

    return false;

}

displayClients()

Параметры и возвращаемое значение: -

Назначение: Вывод всех клиентов в отсортированном порядке.

Реализация: Вызывает displayInOrder у объекта clients (метод класса **ClientBST**).

void Blockchain::displayClients() {

    clients.displayInOrder();

}

displayTransactions()

Параметры и возвращаемое значение: -

Назначение: Вывод всех транзакций

Реализация: Вызывает displayTransactions у объекта transactions (метод класса **TransactionList**).

void Blockchain::displayTransactions() {

    transactions.displayTransactions();

}

~Blockchain() – деструктор

Параметры и возвращаемое значение: -

Назначение: Сохранение данные перед уничтожением объекта.

Реализация: Открывает файл для перезаписи. Вызывает writeClientNode для корня дерева клиентов. Закрывает файл.

Blockchain::~Blockchain() {

    ofstream out;

    out.open(savefileClients);

    if (out.is\_open()) {

        writeClientNode(clients.root, savefileClients);

    }

    out.close();

    out.open(savefileTransactions);

    if (out.is\_open()) {

        writeTransactionList(transactions.head, savefileTransactions);

    }

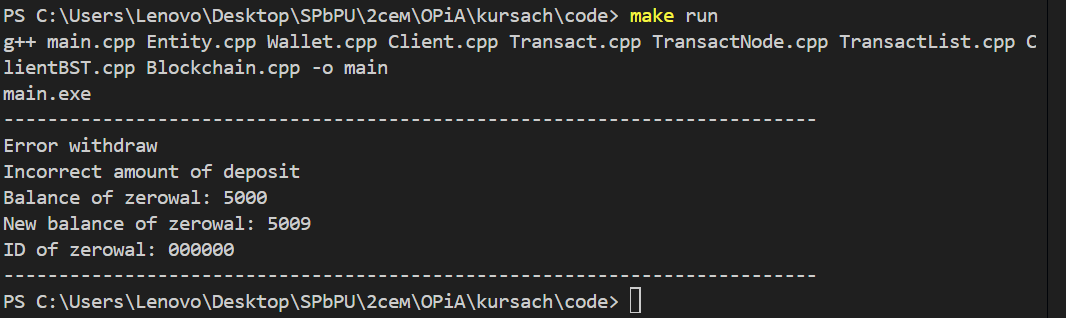
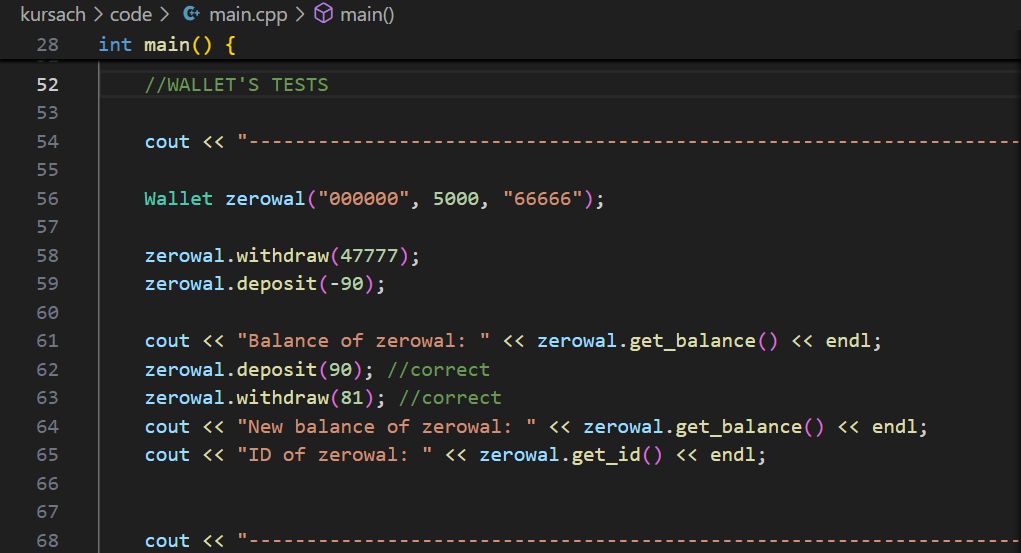
    out.close();

}

**Описание тестирования разработанного решения**

Тестирование проводилось по модулям, начиная с базовых классов и заканчивая комплексной проверкой системы Blockchain.

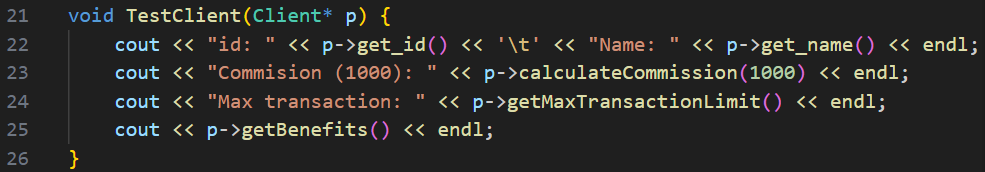
Тестирование класса Wallet

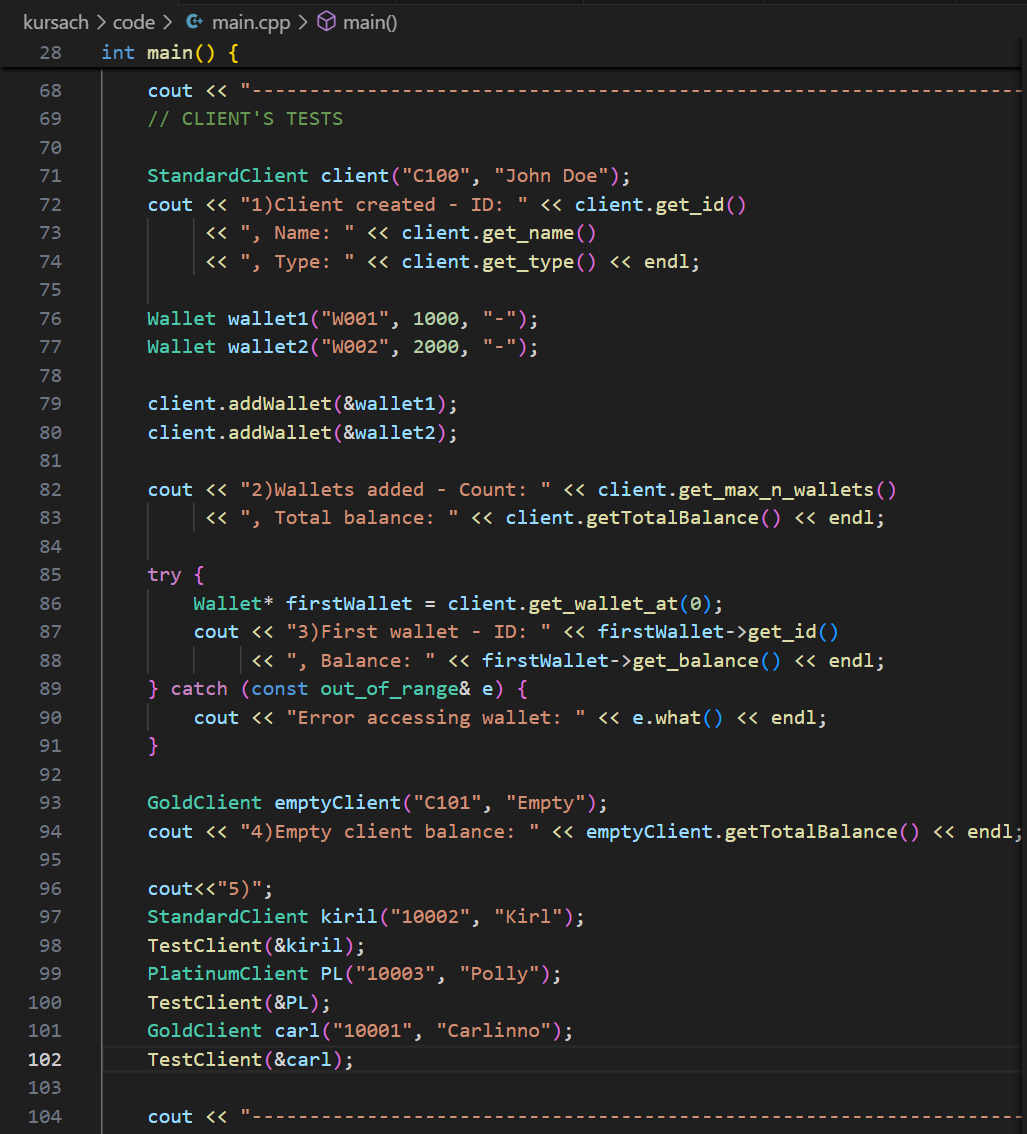


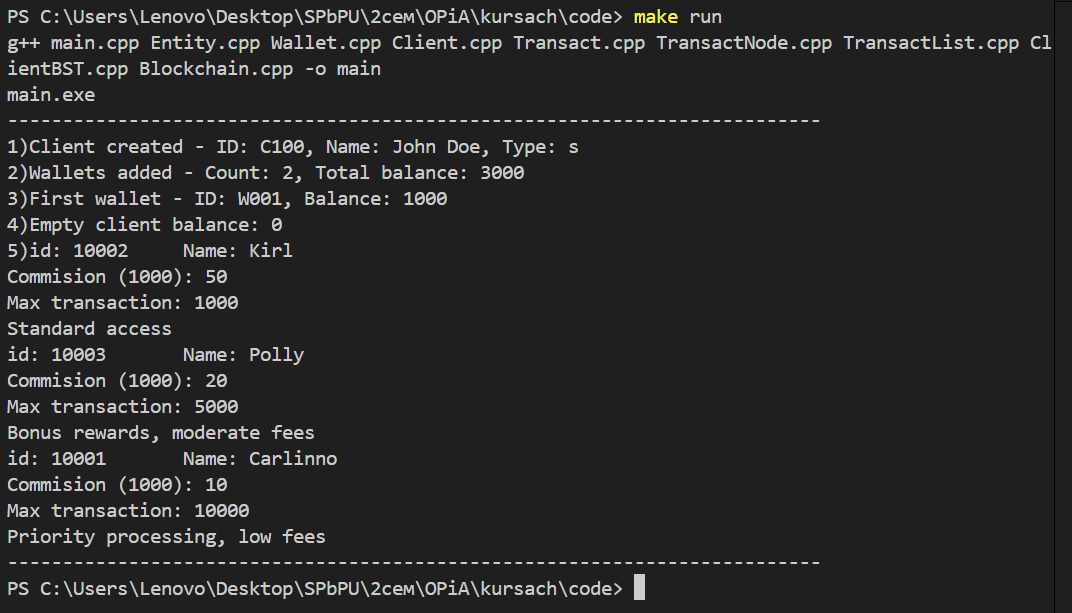
Проверяется:

* Обработка некорректных операций (снятие больше баланса, отрицательный депозит);
* Корректность выполнения валидных операций;
* Получение информации о кошельке;

Тестирование класса Client и его дочерних классов



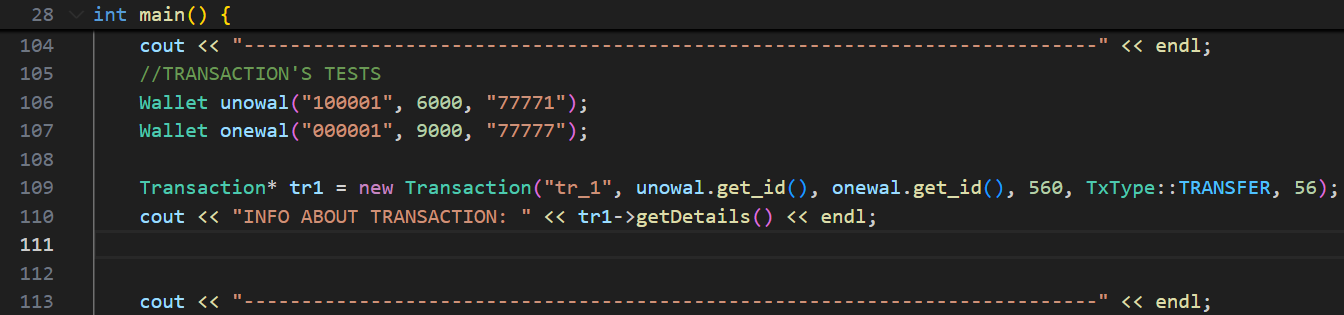


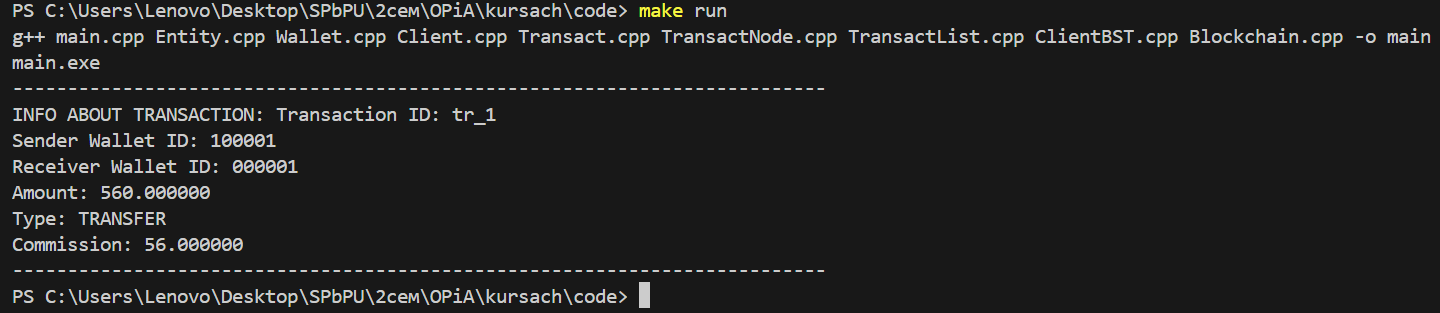


Проверяется:

* Создание клиента и проверка базовых свойств;
* Тестирование работы с кошельками, проверка доступа к ним;
* Проверка пустого клиента;
* Проверка разных типов клиентов и вывода их параметров;

Тестирование класса Transaction



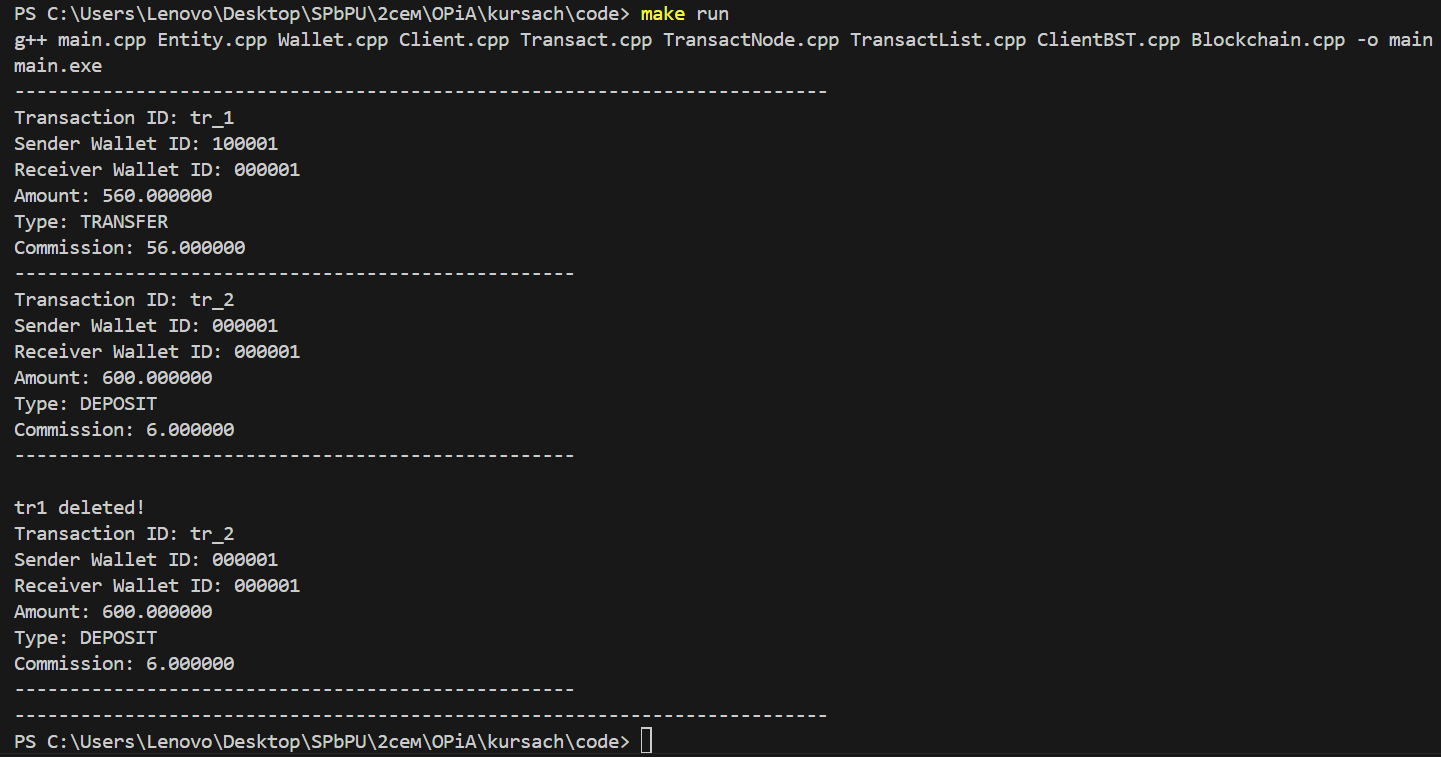


Проверяется:

* Создание транзакции;
* Корректность вывода информации о транзакции;

Тестирование класса TransactionList



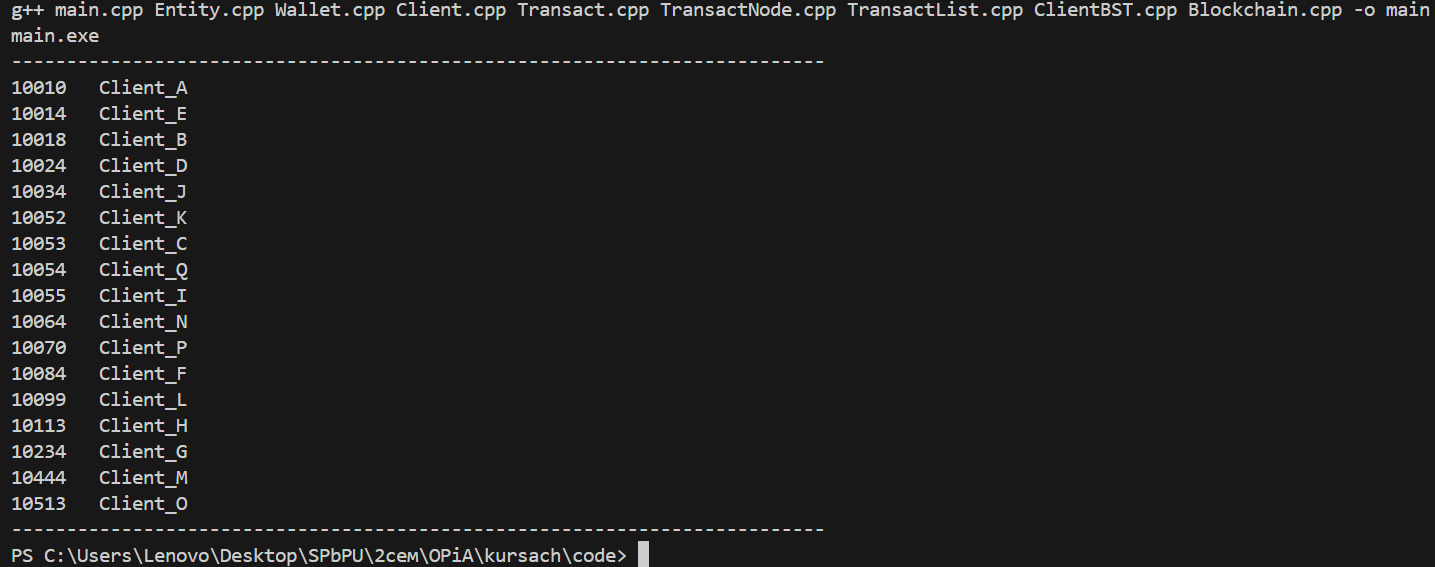


Проверяется:

* Добавление транзакций в список;
* Отображение списка транзакций;
* Удаление транзакции по id;
* Обработка случая, когда транзакция не найдена;

Тестирование класса TransactionList

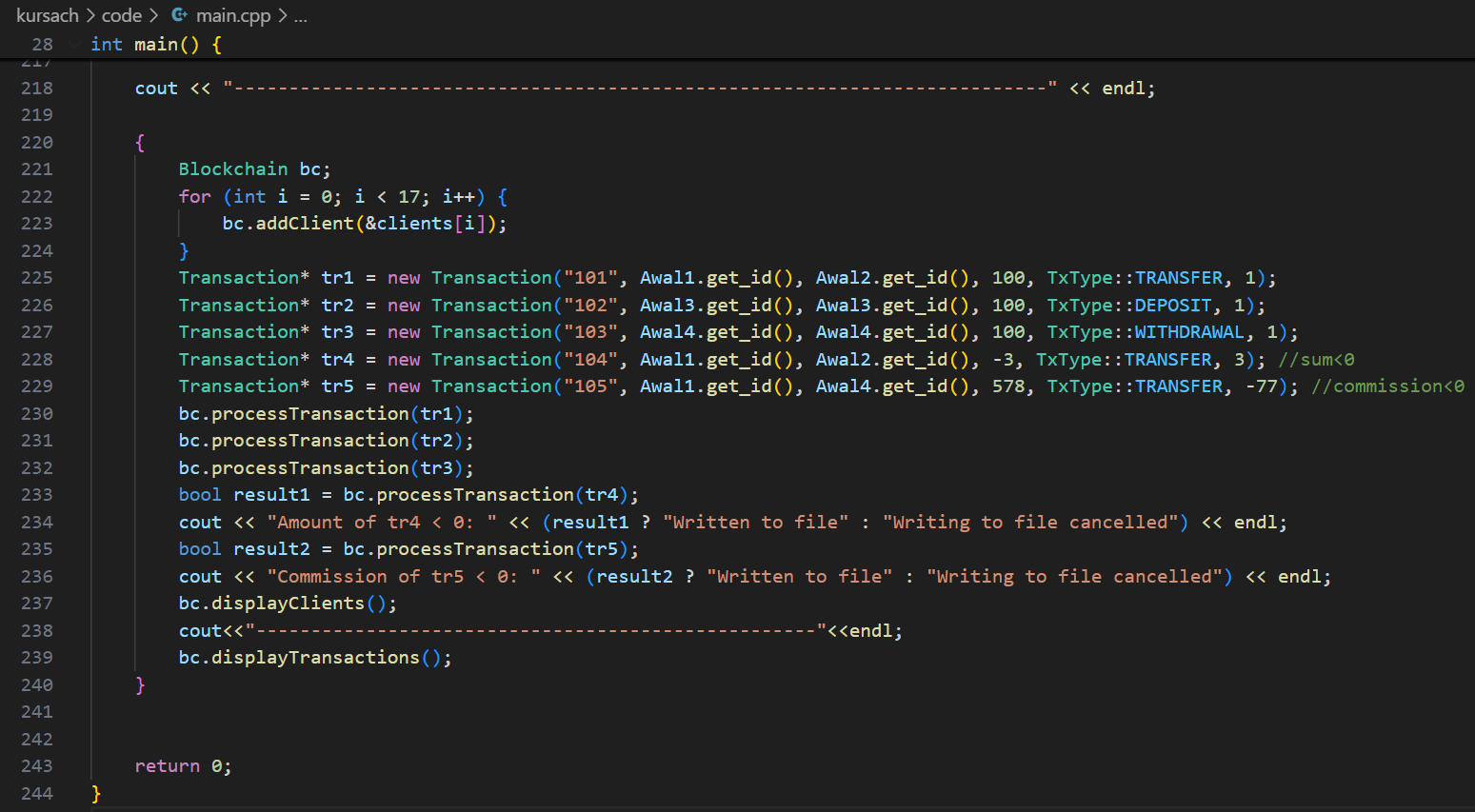


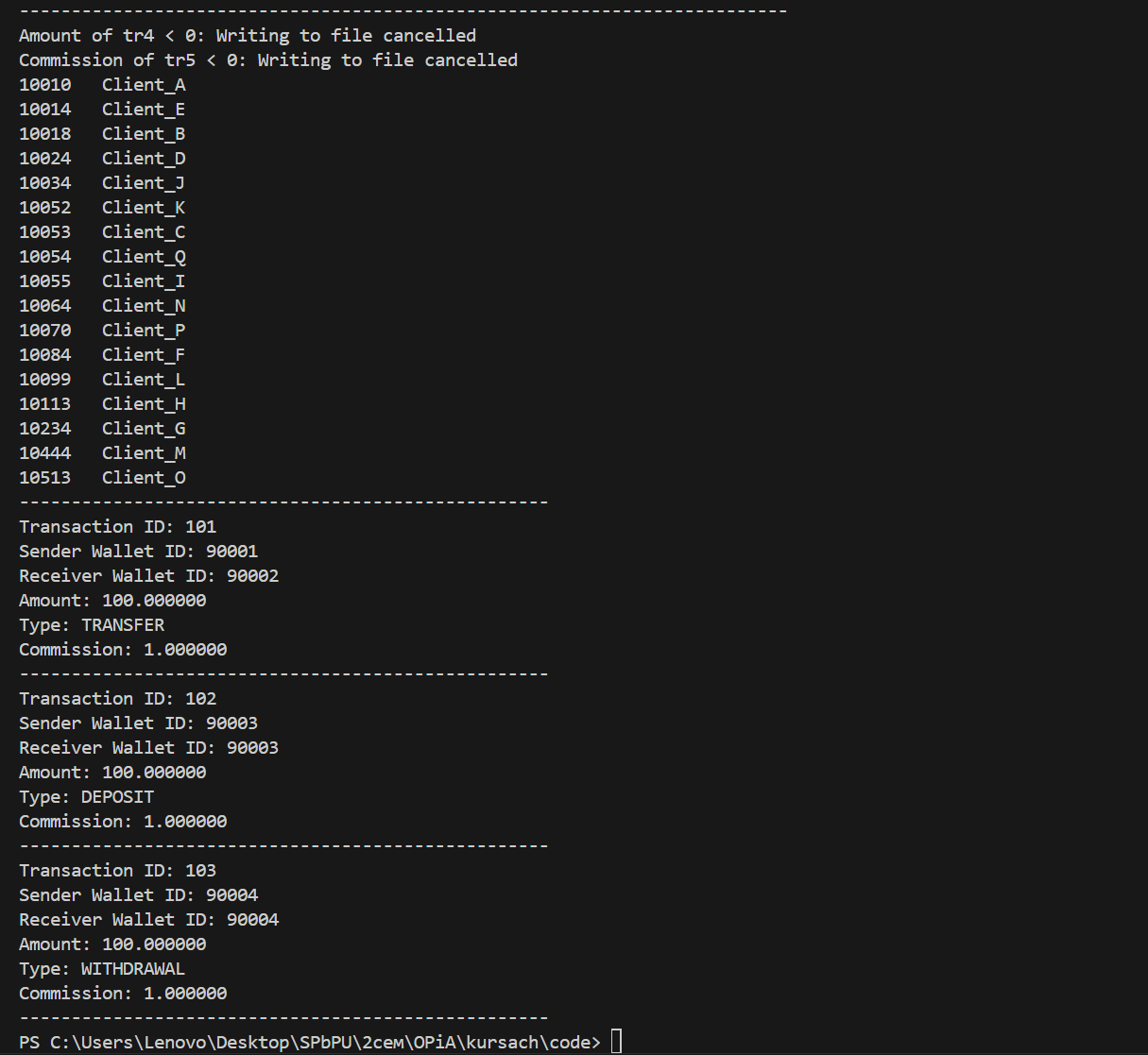


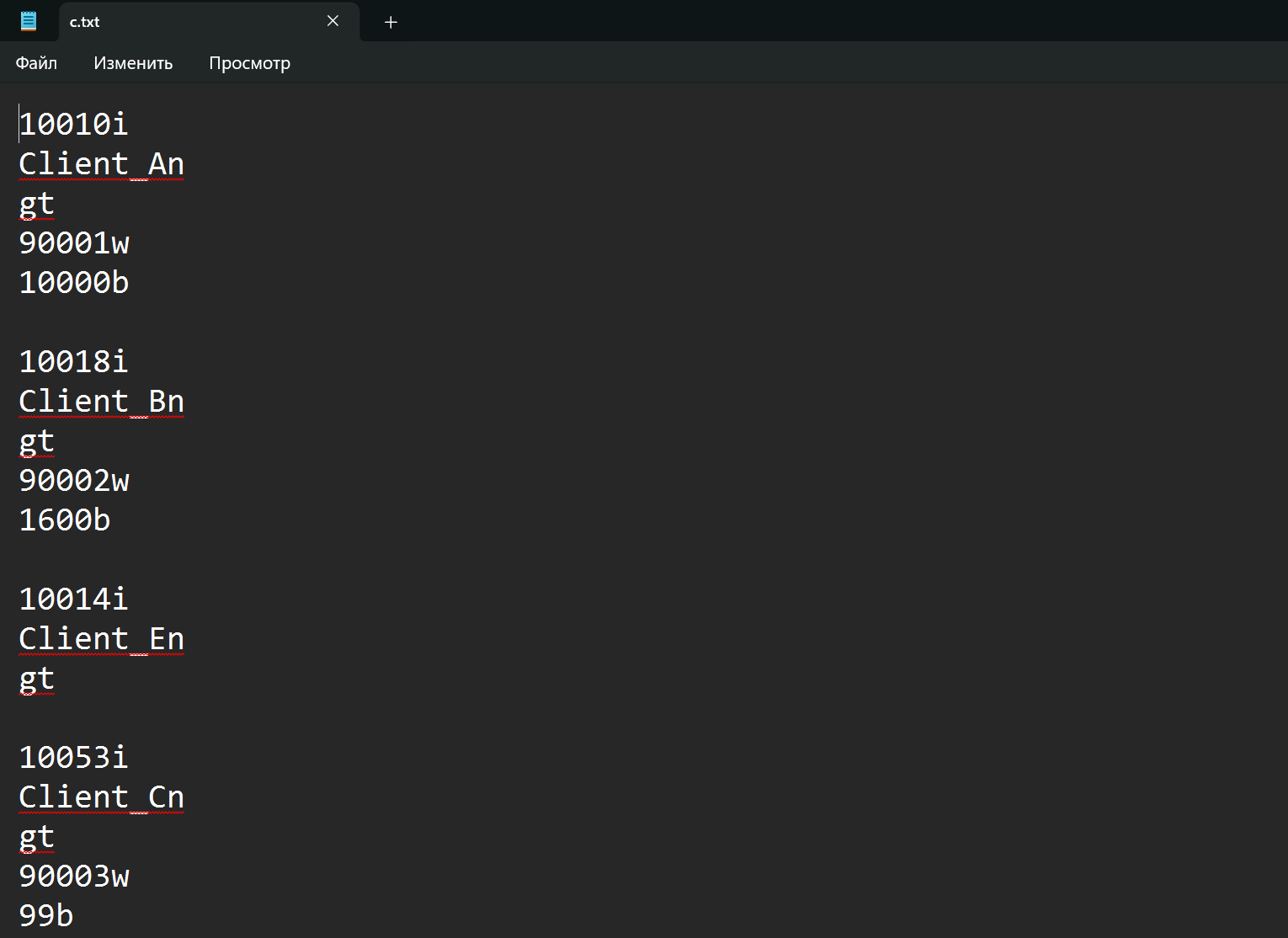
Проверяется:

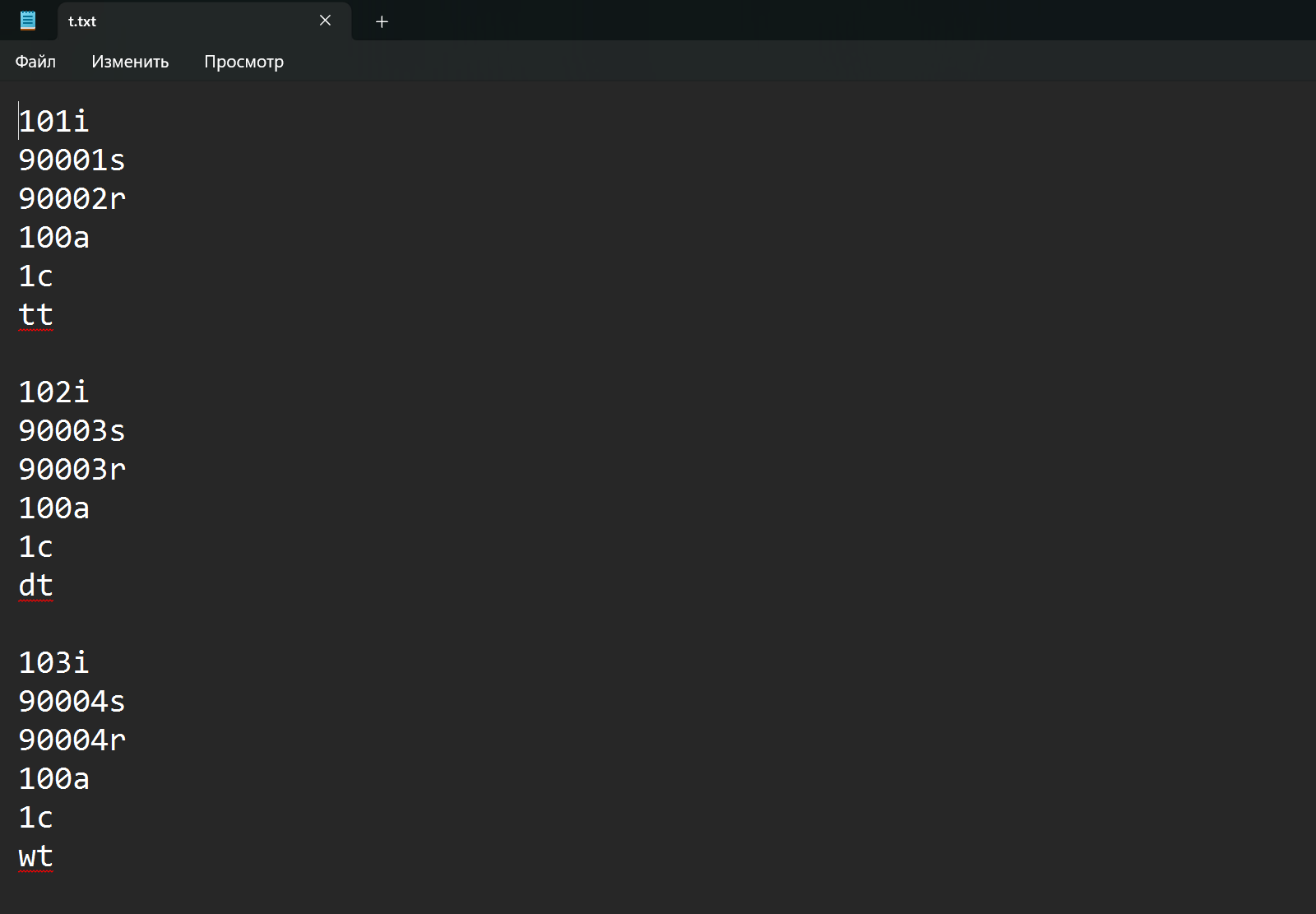
* Добавление клиентов в дерево;
* Отображение дерева в порядке возрастания;
* Удаление клиентов из дерева;

Тестирование класса Blockchain









Проверяется:

* Запись и чтение всех типов транзакций;
* Запись и чтение клиентов;
* Попытка записать в файл транзакцию с отрицательным значением – отклоняется;
* Попытка записать в файл транзакцию с отрицательной комиссией – отклоняется;

Проблемы, возникшие в ходе тестирования:

В ходе тестирования реализации блокчейна мы столкнулись с рядом технических сложностей, которые потребовали внесения срочных корректировок в исходную логику обработки транзакций, чтобы обеспечивалась хотя бы минимальная работоспособность системы. Следовало пересмотреть условия обработки комиссий, так как комиссия не была напрямую связана с типом клиента и по структуре нашего кода не могла получить соответствующее значение методом calculateCommission – проверка комиссии за транзакцию упрощена до условия, чтобы она не была больше самой суммы транзакции (кроме транзакций типа DEPOSIT) и не была отрицательной. Главной проблемой оказалась строгая система проверок при записи в файл, которая в первоначальной реализации не позволяла записывать транзакции типа TRANSFER. Чтобы обеспечить базовую функциональность системы, мы оставили проверку только для клиента-отправителя. Эти изменения позволили создать работоспособную систему, которая корректно обрабатывает и записывает большинство транзакций.

**Заключение**

В результате работы главная цель была достигнута – реализована упрощённая система блокчейн-транзакций. Нами было освоено эффективное структурирование данных для работы с файлами и памятью. Удалось реализовать несколько функциональных модулей, которые образуют законченную и работоспособную систему. Нами была оценена важность тщательного проектирования системы проверок на ранних этапах. Кроме того, мы убедились в необходимости заблаговременного комплексного тестирования всех сценариев использования. Но несмотря на возникшие в процессе разработки сложности, нам удалось создать решение, отвечающее основным требованиям. Можно сделать вывод, что данное задание требовало от нас креативности и умения адаптировать исходные требования к практическим условиям реализации. В заключение хочется сказать, что этот проект стал ценным опытом, наглядно показавшим, как теоретические принципы ООП применяются в реальных разработках.