Министерство образования Республики Беларусь
Учреждение образования
«Брестский государственный технический университет»
Кафедра ИИТ

Лабораторная работа №2
По дисциплине: “СПП”

Выполнил:
Студент 3 курса
Группы ПО-12
Боричевский Д. О.
Проверила:
Кулик А. Д.

Брест 2026

Цель работы: Закрепление навыков объектно-ориентированного программирования на языке Python.

Задание 1

2) Равносторонний треугольник, заданный длинами сторон – Предусмотреть возможность определения площади и периметра, а также логический метод, определяющий существует или такой треугольник. Конструктор должен позволять создавать объекты с начальной инициализацией. Переопределить метод __eq__, выполняющий сравнение объектов данного типа.

Решение:

```
import math


class EquilateralTriangle:
    """Класс равностороннего треугольника"""

    def __init__(self, side: float):
        """Конструктор с начальной инициализацией"""
        self._side = side

    @property
    def side(self) -> float:
        """Геттер для стороны"""
        return self._side

    @side.setter
    def side(self, value: float):
        """Сеттер для стороны с проверкой"""
        if value <= 0:
            raise ValueError("Сторона должна быть положительной")
        self._side = value

    def is_valid(self) -> bool:
        """Проверка существования треугольника"""
        return self._side > 0

    def perimeter(self) -> float:
        """Вычисление периметра"""
        if not self.is_valid():
            raise ValueError("Треугольник не существует")
        return 3 * self._side

    def area(self) -> float:
        """Вычисление площади"""
        if not self.is_valid():
            raise ValueError("Треугольник не существует")
        return (math.sqrt(3) / 4) * self._side**2

    def __str__(self) -> str:
        """Строковое представление"""
        status = "существует" if self.is_valid() else "не существует"
        return (
            f"Равносторонний треугольник (сторона={self._side}), " f"статус:
{status}"
```

```
        )

    def __eq__(self, other) -> bool:
        """Сравнение треугольников по площади"""
        if not isinstance(other, EquilateralTriangle):
            return NotImplemented
        return math.isclose(self.area(), other.area())


# === Демонстрация работы ===
if __name__ == "__main__":
    print("=" * 50)
    print("ЗАДАНИЕ 1: РАВНОСТОРОННИЙ ТРЕУГОЛЬНИК")
    print("=" * 50)

    # Создание объектов
    t1 = EquilateralTriangle(5.0)
    t2 = EquilateralTriangle(5.0)
    t3 = EquilateralTriangle(10.0)
    t_invalid = EquilateralTriangle(-3.0)

    # Вывод информации
    print(f"\nТреугольник 1: {t1}")
    print(f"  Периметр: {t1.perimeter():.2f}")
    print(f"  Площадь: {t1.area():.2f}")

    print(f"\nТреугольник 2: {t2}")
    print(f"\nТреугольник 3: {t3}")

    # Проверка невалидного
    print(f"\nНевалидный: {t_invalid}")
    print(f"  Существует? {t_invalid.is_valid()}")

    # Сравнение
    print(f"\nСравнение t1 == t2 (одинаковые): {t1 == t2}")
    print(f"Сравнение t1 == t3 (разные): {t1 == t3}")

    # Изменение стороны
    print(f"\nИзменение стороны t1 на 10...")
    t1.side = 10.0
    print(f"Теперь t1 == t3: {t1 == t3}")
```

Задание 2

2) Система Платежи. Клиент имеет Счет в банке и Кредитную Карту (КК). Клиент может оплатить Заказ, сделать платеж на другой Счет, заблокировать КК и аннулировать Счет. Администратор может заблокировать КК за превышение кредита.

Решение:

```
from abc import ABC, abstractmethod
from datetime import datetime
from typing import Optional
from enum import Enum


class PaymentStatus(Enum):
    """Статусы платежа"""
```

```
    PENDING = "в обработке"
    COMPLETED = "выполнен"
    FAILED = "отклонен"


class AccountStatus(Enum):
    """Статусы счета"""

    ACTIVE = "активен"
    BLOCKED = "заблокирован"
    CLOSED = "закрыт"


class CardStatus(Enum):
    """Статусы карты"""

    ACTIVE = "активна"
    BLOCKED = "заблокирована"


# ==================== КЛАССЫ СИСТЕМЫ ====================


class Order:
    """Заказ (ассоциация с Client)"""

    def __init__(self, order_id: str, amount: float, description: str):
        self.order_id = order_id
        self.amount = amount
        self.description = description
        self.is_paid = False

    def __str__(self):
        status = "оплачен" if self.is_paid else "не оплачен"
        return (
            f"Заказ {self.order_id}: {self.description}, {self.amount} руб.
({status})"
        )


class CreditCard:
    """Кредитная карта (агрегация с Client)"""

    def __init__(self, card_number: str, limit: float):
        self.card_number = card_number
        self.limit = limit
        self.balance = 0.0  # Текущий долг
        self.status = CardStatus.ACTIVE

    @property
    def available_credit(self) -> float:
        """Доступный кредит"""
        return self.limit - self.balance

    def charge(self, amount: float) -> bool:
        """Списание с карты"""
        if self.status == CardStatus.BLOCKED:
            print(f"Карта {self.card_number}: операция отклонена (карта
заблокирована)")
            return False
        if amount > self.available_credit:
            print(f"Карта {self.card_number}: недостаточно средств")
```

```
            return False
        self.balance += amount
        print(f"Карта {self.card_number}: списано {amount} руб.")
        return True

    def block(self):
        """Блокировка карты"""
        self.status = CardStatus.BLOCKED
        print(f"Карта {self.card_number}: заблокирована")

    def __str__(self):
        return (
            f"КК {self.card_number}: лимит={self.limit}, "
            f"долг={self.balance}, статус={self.status.value}"
        )


class BankAccount:
    """Банковский счет (агрегация с Client)"""

    def __init__(self, account_number: str, initial_balance: float = 0.0):
        self.account_number = account_number
        self._balance = initial_balance
        self.status = AccountStatus.ACTIVE

    @property
    def balance(self) -> float:
        return self._balance

    def deposit(self, amount: float) -> bool:
        """Пополнение счета"""
        if self.status != AccountStatus.ACTIVE:
            print(f"Счет {self.account_number}: операция невозможна")
            return False
        self._balance += amount
        print(f"Счет {self.account_number}: зачислено {amount} руб.")
        return True

    def withdraw(self, amount: float) -> bool:
        """Списание со счета"""
        if self.status != AccountStatus.ACTIVE:
            print(f"Счет {self.account_number}: операция невозможна")
            return False
        if amount > self._balance:
            print(f"Счет {self.account_number}: недостаточно средств")
            return False
        self._balance -= amount
        print(f"Счет {self.account_number}: списано {amount} руб.")
        return True

    def close(self):
        """Аннулирование счета"""
        self.status = AccountStatus.CLOSED
        print(f"Счет {self.account_number}: аннулирован")

    def __str__(self):
        return (
            f"Счет {self.account_number}: баланс={self._balance}, "
            f"статус={self.status.value}"
        )


class User(ABC):
```

```
    """Абстрактный класс пользователя (обобщение)"""

    def __init__(self, user_id: str, name: str):
        self.user_id = user_id
        self.name = name

    @abstractmethod
    def get_role(self) -> str:
        pass

    def __str__(self):
        return f"{self.get_role()}: {self.name} (ID: {self.user_id})"


class Client(User):
    """Клиент (реализация User, агрегация Account и Card)"""

    def __init__(self, user_id: str, name: str):
        super().__init__(user_id, name)
        self.account: Optional[BankAccount] = None
        self.credit_card: Optional[CreditCard] = None

    def get_role(self) -> str:
        return "Клиент"

    def assign_account(self, account: BankAccount):
        """Привязка счета"""
        self.account = account
        print(f"{self.name}: привязан {account}")

    def assign_card(self, card: CreditCard):
        """Привязка карты"""
        self.credit_card = card
        print(f"{self.name}: привязана {card}")

    def pay_order(self, order: Order) -> bool:
        """Оплата заказа со счета"""
        if not self.account:
            print(f"{self.name}: нет привязанного счета")
            return False

        print(f"\n--- Оплата заказа {order.order_id} ---")
        if self.account.withdraw(order.amount):
            order.is_paid = True
            print(f"Заказ {order.order_id} оплачен!")
            return True
        return False

    def pay_order_by_card(self, order: Order) -> bool:
        """Оплата заказа картой"""
        if not self.credit_card:
            print(f"{self.name}: нет привязанной карты")
            return False

        print(f"\n--- Оплата заказа {order.order_id} картой ---")
        if self.credit_card.charge(order.amount):
            order.is_paid = True
            print(f"Заказ {order.order_id} оплачен картой!")
            return True
        return False

    def transfer_to_account(self, target_account: BankAccount, amount: float) ->
bool:
```

```
        """Перевод на другой счет"""
        if not self.account:
            print(f"{self.name}: нет привязанного счета")
            return False

        print(
            f"\n--- Перевод {amount} руб. на счет {target_account.account_number}
---"
        )
        if self.account.withdraw(amount):
            target_account.deposit(amount)
            print("Перевод выполнен!")
            return True
        return False

    def block_own_card(self):
        """Самоблокировка карты"""
        if self.credit_card:
            self.credit_card.block()

    def close_account(self):
        """Аннулирование счета"""
        if self.account:
            self.account.close()
            self.account = None


class Administrator(User):
    """Администратор (реализация User)"""

    def get_role(self) -> str:
        return "Администратор"

    def block_card_for_debt(self, card: CreditCard):
        """Блокировка карты за превышение кредита"""
        if card.balance >= card.limit:
            print(f"\n[АДМИН] Блокировка карты {card.card_number} за превышение
лимита")
            card.block()
        else:
            print(
                f"\n[АДМИН] Карта {card.card_number} в норме, блокировка не
требуется"
            )


# ==================== ДЕМОНСТРАЦИЯ ====================

if __name__ == "__main__":
    print("=" * 60)
    print("ЗАДАНИЕ 2: СИСТЕМА ПЛАТЕЖИ")
    print("=" * 60)

    # Создание пользователей
    client1 = Client("C001", "Иван Петров")
    admin = Administrator("A001", "Админ Системы")

    print(f"\n{client1}")
    print(f"{admin}")

    # Создание счетов и карт
    account1 = BankAccount("40817810000000000001", 10000.0)
    account2 = BankAccount("40817810000000000002", 5000.0)
```

```
    card1 = CreditCard("4276123456789012", 50000.0)

    # Привязка к клиенту
    print("\n--- Привязка счетов и карт ---")
    client1.assign_account(account1)
    client1.assign_card(card1)

    # Создание заказов
    order1 = Order("ORD-001", 3000.0, "Ноутбук")
    order2 = Order("ORD-002", 500.0, "Мышь")

    print(f"\n{order1}")
    print(f"{order2}")

    # Оплата заказа со счета
    client1.pay_order(order1)
    print(f"\nТекущий баланс счета: {account1.balance} руб.")

    # Оплата заказа картой
    client1.pay_order_by_card(order2)
    print(f"\nТекущий долг по карте: {card1.balance} руб.")

    # Перевод на другой счет
    client1.transfer_to_account(account2, 2000.0)
    print(f"\nБаланс счета 1: {account1.balance} руб.")
    print(f"Баланс счета 2: {account2.balance} руб.")

    # Симуляция превышения кредита
    print("\n--- Симуляция превышения кредита ---")
    big_order = Order("ORD-003", 50000.0, "Автомобиль")
    client1.pay_order_by_card(big_order)  # Исчерпываем лимит

    # Проверка администратором
    admin.block_card_for_debt(card1)

    # Попытка оплаты заблокированной картой
    order3 = Order("ORD-004", 1000.0, "Бензин")
    client1.pay_order_by_card(order3)

    # Самоблокировка и аннулирование
    print("\n--- Закрытие счета ---")
    client1.close_account()

    print("\n" + "=" * 60)
    print("ДЕМОНСТРАЦИЯ ЗАВЕРШЕНА")
    print("=" * 60)
```

Вывод: В результате выполнения лабораторной работы были закреплены знания объектно-ориентированного программирования Python при решении задач.