**Задание 1.**

На основе тестов нужно было реализовать основную логику приложения по следующим функциональным требованиям:

● Пользователь имеет возможность регистрации и авторизации в приложение;

● Пользователь имеет возможность создания одного или нескольких счетов, а также их закрытие;

● Пользователь имеет возможность перемещения денежных средств между своими счетами или счетами других пользователей;

● Все классы с логикой являются реализацией соответствующего интерфейса (например UserService -> UserServiceImpl);

● Хранение данных о пользователях осуществляется с помощью словаря HashMap, где ключом является идентификатор пользователя, а значением - объект (ссылка на объект) User;

● Хранение данных о счётах осуществляется с помощью словаря HashMap, где ключом выступает идентификатор счёта, а значением - объект Account;

● Для ассоциации счёта и пользователя, в структуре данных Account предусмотрено поле userId, с идентификатором пользователя - владельца счёта;

● Операции с поиском данных осуществляться с помощью методов StreamAPI (например, поиск счётов по значению поля userId);

● В качестве UI выступает консольное окно.

В целях выполнения задания были созданы объект User, который выступает в качестве владельца счетов, а также объект Account, который олицетворяет счет. Были созданы интерфейсы UserService, AccountService, ProcessingService.

Интерфейс UserService отвечает за:

1. Регистрацию пользователя по почте (если такой почты еще не было).
2. Проверку по Id существование пользователя.
3. Проверку, что пользователь успешно авторизовался по почте и паролю.

Интерфейс AccountService отвечает за:

1. Создание счета для пользователя.
2. Закрытие счета клиента.
3. Проверку, что к определенному счету имеет допуск только его владелец.
4. Проверку, что этот счет принадлежит кому-то.
5. Возвращение счета по id счета.
6. Обновление данных счета.

Интерфейс ProcessingService отвечает за:

1. Перевод денег с одного счета на другой счет.

Реализацию этих методов можно найти в соответствующих классах с суффиксом –Impl (UserServiceImpl, AccountServiceImpl, ProcessingServiceImpl). Также хранения клиентов в виде HashMap осуществляется в UserServiceImpl, а хранение счетов происходит в AccountServiceImpl.

В тестовых классах (UserServiceTest, AccountServiceTest, ProcessingServiceTest) происходит тестирование различных операций между клиентами и счетами. В класс UserServiceTest был добавлен метод тестирования для закрытия счета – userIsAbleToCloseAccount(). Все остальные методы в классах тестирования были реализованы согласно исходным (прописанных заранее).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Название метода | Описание теста | Значение аргументов для тестирования | Ожидаемый резуль-тат | Получен-ный ре-зуль-тат |
| isUserRegisterCorrectly()  1) *assertTrue*(userService.isUserRegistered(user.getId())); | Проверка, что пользователь регистрируется в in-memory базу | User("John", "Doe", "johndoe@mail.com", "password") | *1)True* | *1)True* |
| isUserAuthCorrectly()  1)  *assertTrue*(userService.isUserAuthenticated(user.getEmail(), user.getPassword())); | Проверка, что пользователь успешно авторизовался | User("John", "Doe", "johndoe@mail.com", "password") | *1)True* | *1)True* |
| userIsAbleToCreateAccount()  1)  *assertTrue*(optionalAccount.isPresent()); | Проверка, что пользователь может создавать счёт | User("John", "Doe", "johndoe@mail.com", "password")  Account(1, 0.0) | *1)True* | *1)True* |
| userIsAbleToCloseAccount()  1)  *assertFalse*(optionalAccount.isPresent()); | Проверка, что пользователь может закрыть счет | User("John", "Doe", "johndoe@mail.com", "password")  Account(1, 0.0) | *1)False* | *1)False* |
| createdAccountIsBelongToUser()  1)  *assertTrue*(accountService.getAccountById(user, accountID).isPresent()); | проверяет, что при создании счёта он действительно принадлежит нужному (конкретному) пользователю | User("John", "Doe", "johndoe@mail.com", "password")  Account(1, 0.0) | *1)True* | *1)*True |
| nonAuthorizedUserHasNoAccessToAccount()  1)  *assertFalse*(accountService.getAccountById(user2, accountID\_1).isPresent()); | проверяет, что неавторизован-ный пользователь или другой юзер не может получить доступ к чужому счёту | User1("John", "Doe", "johndoe@mail.com", "password")  User2("Jane", "Doe", "janedoe@mail.com", "password")  Account(1, 0.0) | *1)False* | *1)False* |
| accountIsBelongToUser()  1)  *assertNotNull*(account.getUserId()); | проверка, что поле userId класса Account не пустое (то есть создаваемому счёт принадлежит какому-либо пользователю) | User("John", "Doe", "johndoe@mail.com", "password")  Account(1, 0.0) | *1)True* | *1)True* |
| transactionsBetweenAccountsIsValid()  1)  *assertTrue*(processingService.transfer(account1, account2, 25.0));    2)  *assertEquals*(75.0, account1.getBalance())  3)  *assertEquals*(75.0, account2.getBalance()) | проверка работоспособ-ности транзакции между счетами разных пользователей | User1("John", "Doe", "john.doe@example.com", "password")  User2("Jane", "Doe", "jane.doe@example.com", "password")  Account1(1,100.0)  Account2(2,50.0)  Цена перевода – 25.0 | *1)True*  *2)Equals*  *3)Equals* | *1)True*  *2)Equals*  (True)  *3)Equals*  (True) |
| transactionsBetweenAccountsOfOneUserIsValid()  1)  *assertTrue*(processingService.transfer(account1, account2, 25.0));    2)  *assertEquals*(75.0, account1.getBalance())  3)  *assertEquals*(75.0, account2.getBalance()) | проверка работоспособности транзакции между счетами одного пользователя | User("John", "Doe", "john.doe@example.com", "password")  Account1(1,100.0)  Account2(2,50.0)  Цена перевода – 25.0 | *1)True*  *2)Equals*  *3)Equals* | *1)True*  *2)Equals*  (True)  *3)Equals*  (True) |
| transactionRollbackByError()  1)  *assertFalse*(processingService.transfer(account1, error, 25.0));  2)  *assertEquals*(100.0, account1.getBalance())  3)  *assertEquals*(50.0, account2.getBalance()) | проверка, что транзакция откатывает изменения при ошибке передачи  (с счета Account1 пытаемся перевести на несуществующий счет Account100) | User("John", "Doe", "john.doe@example.com", "password")  Account1(1,100.0)  Account2(2,50.0)  Account100 – которого не существует  Цена перевода – 25.0  Account error = new Account(new User("Sam", "Sam", "sam@example.com", "password"),  100, 0); | *1)False*  *2)Equals*  *3)Equals* | *1)False*  *2)Equals*  (True)  *3)Equals*  (True) |
| transactionNotPermitIfNegativeDebt()  1)  *assertFalse*(processingService.transfer(account1, account2, 25.0));    2)  *assertEquals*(-100.0, account1.getBalance())  3)  *assertEquals*(50.0, account2.getBalance()) | проверка, что транзакция не осуществляет-ся при отрицатель-ном балансе счёта - отправителя | User("John", "Doe", "john.doe@example.com", "password")  Account1(1, -100.0)  Account2(2, 50.0)  Цена перевода – 25.0 | *1)False*  *2)Equals*  *3)Equals* | *1)False*  *2)Equals*  (True)  *3)Equals*  (True) |

Приложение.

Package test.

Class UserServiceTest.

class UserServiceTest {  
  
 private UserService userService;  
 private AccountService accountService;  
  
 @BeforeEach  
 void setUp() {  
 userService = new UserServiceImpl();  
 accountService = new AccountServiceImpl();  
 }  
  
 @Test  
 void isUserRegisterCorrectly(){  
 *//Проверка, что пользователь регистрируется в in-memory базу* User user = new User("John", "Doe", "johndoe@mail.com", "password");  
 userService.register(user);  
 *assertTrue*(userService.isUserRegistered(user.getId()));  
 }  
  
 @Test  
 void isUserAuthCorrectly(){  
 *//Проверка, что пользователь успешно авторизовался* User user = new User("John", "Doe", "johndoe@mail.com", "password");  
 userService.register(user);  
 *assertTrue*(userService.isUserAuthenticated(user.getEmail(), user.getPassword()));  
 }  
  
 @Test  
 void userIsAbleToCreateAccount(){  
 *//Проверка, что пользователь может создавать счёт* User user = new User("John", "Doe", "johndoe@mail.com", "password");  
 userService.register(user);  
 int accountID = accountService.createAccount(user, 250);  
 Optional<Account> optionalAccount = accountService.getAccountById(user, accountID);  
 *assertTrue*(optionalAccount.isPresent());  
 }  
  
 @Test  
 void userIsAbleToCloseAccount(){  
 *//Проверка, что пользователь может закрыть счет* User user = new User("John", "Doe", "johndoe@mail.com", "password");  
 userService.register(user);  
 int accountID = accountService.createAccount(user, 250); *// счет создался* accountService.closeAccount(user, accountID ); *// счет закрылся* Optional<Account> optionalAccount = accountService.getAccountById(user, accountID);  
 *assertFalse*(optionalAccount.isPresent());  
 }  
}

Class AccountServiceTest.

class AccountServiceTest {  
  
 private AccountService accountService;  
 private UserService userService;  
  
 @BeforeEach  
 void setUp() {  
 userService = new UserServiceImpl();  
 accountService = new AccountServiceImpl();  
 }  
  
 @Test  
 void createdAccountIsBelongToUser(){  
 *//проверяет, что при создании счёта он действительно принадлежит нужному (конкретному) пользователю* User user = new User("John", "Doe", "johndoe@mail.com", "password");  
 userService.register(user);  
 int accountID = accountService.createAccount(user, 250);  
 *assertTrue*(accountService.getAccountById(user, accountID).isPresent());  
 }  
  
 @Test  
 void nonAuthorizedUserHasNoAccessToAccount(){  
 *//проверяет, что неавторизованный пользователь или другой юзер не может получить доступ к чужому счёту* User user1 = new User("John", "Doe", "johndoe@mail.com", "password");  
 User user2 = new User("Jane", "Doe", "janedoe@mail.com", "password");  
 userService.register(user1);  
 userService.register(user2);  
 int accountID\_1 = accountService.createAccount(user1, 250);  
 *assertFalse*(accountService.getAccountById(user2, accountID\_1).isPresent());  
 }  
  
 @Test  
 void accountIsBelongToUser(){  
 *//проверка, что поле userId класса Account не пустое (то есть создаваемому счёт принадлежит какому-либо пользователю)* User user = new User("John", "Doe", "johndoe@mail.com", "password");  
 userService.register(user);  
 int accountID = accountService.createAccount(user, 250);  
 Account account = user.getAccounts().get(accountID);  
 *assertNotNull*(account.getUserId());  
 }  
  
}

Class ProcessingServiceTest.

class ProcessingServiceTest {  
  
 private ProcessingService processingService;  
 private AccountService accountService;  
 private UserService userService;  
  
 @BeforeEach  
 void setUp() {  
 userService = new UserServiceImpl();  
 accountService = new AccountServiceImpl();  
 processingService = new ProcessingServiceImpl(accountService, userService);  
 }  
  
 @Test  
 void transactionsBetweenAccountsIsValid(){  
 *//проверка работоспособности транзакции между счетами разных пользователей  
  
 // Создаем двух пользователей* User user1 = new User("John", "Doe", "john.doe@example.com", "password");  
 userService.register(user1);  
 User user2 = new User("Jane", "Doe", "jane.doe@example.com", "password");  
 userService.register(user2);  
  
 *// Создаем счета для пользователей* int accountID\_1 = accountService.createAccount(user1, 0);  
 Account account1 = accountService.getAccountById(user1, accountID\_1).get();  
 account1.setBalance(100.0);  
  
 int accountID\_2 = accountService.createAccount(user2, 0);  
 Account account2 = accountService.getAccountById(user2, accountID\_2).get();  
 account2.setBalance(50.0);  
  
 *// Переводим деньги между счетами  
 assertTrue*(processingService.transfer(account1, account2, 25.0));  
  
  
 *// Проверяем, что балансы изменились правильно  
 assertEquals*(75.0, account1.getBalance());  
 *assertEquals*(75.0, account2.getBalance());  
 }  
  
 @Test  
 void transactionsBetweenAccountsOfOneUserIsValid(){  
 *//проверка работоспособности транзакции между счетами одного пользователя  
  
 // Создаем пользователя* User user = new User("John", "Doe", "john.doe@example.com", "password");  
 userService.register(user);  
  
 *// Создаем два счета для пользователя* int accountID\_1 = accountService.createAccount(user, 0);  
 Account account1 = accountService.getAccountById(user, accountID\_1).get();  
 account1.setBalance(100.0);  
 int accountID\_2 = accountService.createAccount(user, 0);  
 Account account2 = accountService.getAccountById(user, accountID\_2).get();  
 account2.setBalance(50.0);  
  
 *// Переводим деньги между счетами  
 assertTrue*(processingService.transfer(account1, account2, 25.0));  
  
 *// Проверяем, что балансы изменились правильно  
 assertEquals*(75.0, account1.getBalance());  
 *assertEquals*(75.0, account2.getBalance());  
 }  
  
 @Test  
 void transactionRollbackByError(){  
 *//проверка, что транзакция откатывает изменения при ошибке передачи  
  
 // Создаем пользователя* User user = new User("John", "Doe", "john.doe@example.com", "password");  
 userService.register(user);  
  
 *// Создаем два счета для пользователя* int accountID\_1 = accountService.createAccount(user, 0);  
 Account account1 = accountService.getAccountById(user, accountID\_1).get();  
 account1.setBalance(100.0);  
 int accountID\_2 = accountService.createAccount(user, 0);  
 Account account2 = accountService.getAccountById(user, accountID\_2).get();  
 account2.setBalance(50.0);  
  
 Account error = new Account(new User("Sam", "Sam", "sam@example.com", "password"),  
 100, 0);  
  
 *// Переводим деньги между счетами, но передаем неверный ID счета  
 assertFalse*(processingService.transfer(account1, error, 25.0));  
  
 *// Проверяем, что балансы не изменились  
 assertEquals*(100.0, account1.getBalance());  
 *assertEquals*(50.0, account2.getBalance());  
 }  
  
 @Test  
 void transactionNotPermitIfNegativeDebt(){  
 *//проверка, что транзакция не осуществляется при отрицательном балансе счёта - отправителя  
 // Создаем пользователя* User user = new User("John", "Doe", "john.doe@example.com", "password");  
 userService.register(user);  
  
 *// Создаем два счета для пользователя* int accountID\_1 = accountService.createAccount(user, 0);  
 Account account1 = accountService.getAccountById(user, accountID\_1).get();  
 account1.setBalance(-100.0);  
  
 int accountID\_2 = accountService.createAccount(user, 0);  
 Account account2 = accountService.getAccountById(user, accountID\_2).get();  
 account2.setBalance(50.0);  
  
 *// Переводим больше денег, чем есть на счету  
 assertFalse*(processingService.transfer(account1, account2, 25.0));  
  
 *// Проверяем, что балансы не изменились  
 assertEquals*(-100.0, account1.getBalance());  
 *assertEquals*(50.0, account2.getBalance());  
 }  
}

Package init.

Interface UserService.

public interface UserService {  
 void register(User user);  
 boolean isUserRegistered(int userId);  
 boolean isUserAuthenticated(String email, String password);  
}

Interface AccountService.

public interface AccountService {  
 int createAccount(User user, int balance);  
 boolean hasAccess(User user, int accountId);  
 Optional<Account> getAccountById(User user, int accountId);  
 void updateAccount(User user, Account account);  
 void closeAccount(User user, int accountId);  
}

Interface ProcessingService.

public interface ProcessingService {  
 boolean transfer(Account from, Account to, double amount);  
}

Class UserServiceImpl.

public class UserServiceImpl implements UserService {  
 private int lastUserId = 0;  
 private static InMemoryBase *base*;  
  
 public UserServiceImpl() {  
 this.*base* = new InMemoryBase(new HashMap<Integer, User>());  
 }  
  
 @Override  
 public void register(User user) {  
 *// регистрирует пользователя, если не было такой почты* if (*base*.getUsers().values().stream().anyMatch(us -> us.getEmail().equals(user.getEmail()))) {  
 return;  
 }  
 user.setId(++lastUserId);  
 *base*.getUsers().put(user.getId(), user);  
 }  
  
 @Override  
 public boolean isUserRegistered(int userId) {  
 *// проверяет, что такой пользователь существует* return *base*.getUsers().values().stream()  
 .anyMatch(user -> user.getId() == userId);  
 }  
  
 @Override  
 public boolean isUserAuthenticated(String email, String password) {  
 *// проверяет, что пользователь успешно авторизовался* return *base*.getUsers().values().stream()  
 .anyMatch(user -> user.getEmail().equals(email) && user.getPassword().equals(password));  
 }  
}

Class AccountServiceImpl.

public class AccountServiceImpl implements AccountService {  
  
 private int lastAccountId = 0;  
  
 @Override  
 public int createAccount(User user, int balance) {  
 *// создает счет для пользователя, если он существует* Account account = new Account(user, balance, ++lastAccountId);  
 user.getAccounts().put(account.getId(), account);  
 return lastAccountId;  
 }  
  
 @Override  
 public void closeAccount(User user, int accountId) {  
 user.getAccounts().remove(accountId);  
 }  
  
 @Override  
 public boolean hasAccess(User user, int accountId) {  
 *// проверяет, что неавторизованный пользователь или другой юзер не может получить доступ к чужому счёту* Optional<Account> optionalAccount = Optional.*ofNullable*(user.getAccounts().get(accountId));  
 if (optionalAccount.isPresent()) {  
 return true;  
 }  
 return false;  
 }  
  
 @Override  
 public Optional<Account> getAccountById(User user, int accountId) {  
 *// возращает счет по Id* return Optional.*ofNullable*(user.getAccounts().get(accountId));  
 }  
  
 @Override  
 public void updateAccount(User user, Account account) {  
 *// обновляет данные счета* user.getAccounts().put(account.getId(), account);  
 }  
}

Class ProcessingServiceImpl

public class ProcessingServiceImpl implements ProcessingService {  
  
 private final AccountService accountService;  
 private final UserService userService;  
  
 public ProcessingServiceImpl(AccountService accountService, UserService userService) {  
 this.accountService = accountService;  
 this.userService = userService;  
 }  
  
  
 @Override  
 public boolean transfer(Account from, Account to, double amount) {  
 *// осуществляет перевод денег amount с счета fromAccountId на toAccountId* if (!userService.isUserRegistered(from.getUserId()) || !userService.isUserRegistered(to.getUserId())) {  
 return false;  
 }  
  
 if (from.getBalance() < amount) {  
 return false;  
 }  
  
 from.setBalance(from.getBalance() - amount);  
 to.setBalance(to.getBalance() + amount);  
  
 return true;  
 }  
}

Package ob.

Class User.  
public class User {  
 private int id;  
 private String name;  
 private String lastName;  
 private String email;  
 private String password;  
 HashMap<Integer, Account> accounts;  
  
 public User(String name, String lastName, String email, String password) {  
 this.name = name;  
 this.lastName = lastName;  
 this.email = email;  
 this.password = password;  
 accounts = new HashMap<>();  
 }  
  
 public int getId() {  
 return id;  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public String getLastName() {  
 return lastName;  
 }  
  
 public void setLastName(String lastName) {  
 this.lastName = lastName;  
 }  
  
 public String getEmail() {  
 return email;  
 }  
  
 public String getPassword() {  
 return password;  
 }  
  
 public void setId(int id) {  
 this.id = id;  
 }  
  
 public HashMap<Integer, Account> getAccounts() {  
 return accounts;  
 }  
}

Class Account.

public class Account {  
 private int id;  
 private final int userId;  
 private double balance;  
  
 public Account(User user, int balance, int accountId) {  
 this.userId = user.getId();  
 this.balance = balance;  
 this.id = accountId;  
 user.getAccounts().put(accountId, this);  
 }  
  
 public int getId() {  
 return id;  
 }  
  
 public int getUserId() {  
 return userId;  
 }  
  
 public double getBalance() {  
 return balance;  
 }  
  
 public void setBalance(double balance) {  
 this.balance = balance;  
 }  
  
 public void setId(int id) {  
 this.id = id;  
 }  
  
}

class InMemoryBase

public class InMemoryBase {  
 private final HashMap<Integer, User> users;  
 public InMemoryBase(HashMap<Integer, User> users){  
 this.users = users;  
 }  
 public void addUser(User user){  
 users.put(user.getId(), user);  
 }  
 public HashMap<Integer, User> getUsers() {  
 return users;  
 }  
  
}