

Отчет о разработке приложения для учета финансов "ВШЭ-Банк"

Эдуард Альтшуль

18 марта 2025 г.

Содержание

1	Общая идея решения	2
1.1	Реализованный функционал	2
2	Архитектура приложения	2
2.1	Структура проекта	3
3	Принципы SOLID и GRASP в проекте	4
3.1	Принципы SOLID	4
3.1.1	S - Single Responsibility Principle (Принцип единственной ответственности)	4
3.1.2	O - Open/Closed Principle (Принцип открытости/закрытости)	5
3.1.3	L - Liskov Substitution Principle (Принцип подстановки Лисков)	6
3.1.4	I - Interface Segregation Principle (Принцип разделения интерфейсов)	7
3.1.5	D - Dependency Inversion Principle (Принцип инверсии зависимостей)	7
3.2	Принципы GRASP	8
3.2.1	High Cohesion (Высокая связность)	8
3.2.2	Low Coupling (Низкая связанность)	8
3.2.3	Creator (Создатель)	9
3.2.4	Controller (Контроллер)	9
3.2.5	Information Expert (Информационный эксперт)	9

4	Паттерны GoF в проекте	9
4.1	Порождающие паттерны	9
4.1.1	Factory (Фабрика)	9
4.2	Структурные паттерны	10
4.2.1	Facade (Фасад)	10
4.2.2	Proxy (Прокси)	11
4.3	Поведенческие паттерны	12
4.3.1	Command (Команда)	12
4.3.2	Decorator (Декоратор)	12
4.3.3	Template Method (Шаблонный метод)	13
4.3.4	Visitor (Посетитель)	14
5	Тестирование проекта	15
6	Выводы	16

1 Общая идея решения

Разработанное приложение "ВШЭ-Банк" представляет собой программный комплекс для учета личных финансов. Основная цель проекта - создать надежную архитектуру с использованием современных принципов и паттернов проектирования.

1.1 Реализованный функционал

- Управление банковскими счетами (создание, редактирование, удаление)
- Работа с категориями доходов и расходов (создание, редактирование, удаление)
- Учет финансовых операций (доходы, расходы)
- Аналитические инструменты (статистика по категориям, периодам)
- Экспорт данных в различные форматы (CSV, JSON)
- Импорт данных (CSV, JSON, YAML)
- Мониторинг производительности операций
- Высокий процент покрытия кода тестами (более 65%)

2 Архитектура приложения

Приложение спроектировано с использованием многослойной архитектуры, где четко разделены ответственности между слоями:

- **Доменный слой** - содержит основные бизнес-сущности и их поведение
- **Сервисный слой** - реализует бизнес-логику приложения
- **Уровень представления** - консольный интерфейс пользователя

2.1 Структура проекта

```
/FinanceApp
|-- /Domain
|   |-- BankAccount.cs
|   |-- Category.cs
|   |-- Operation.cs
|
|-- /Services
|   |-- /Facade
|   |   |-- BankAccountFacade.cs
|   |   |-- CategoryFacade.cs
|   |   |-- OperationFacade.cs
|   |   |-- AnalyticsFacade.cs
|   |
|   |-- /Command
|   |   |-- ICommand.cs
|   |   |-- CreateOperationCommand.cs
|   |   |-- UpdateBalanceCommand.cs
|   |   |-- DeleteOperationCommand.cs
|   |   |-- TimeMeasureDecorator.cs
|   |
|   |-- /Export
|   |   |-- IVisitor.cs
|   |   |-- CsvExportVisitor.cs
|   |   |-- JsonExportVisitor.cs
|   |
|   |-- /Import
|   |   |-- ImportTemplate.cs
|   |   |-- CsvImport.cs
|   |   |-- JsonImport.cs
|   |   |-- YamlImport.cs
|   |
|   |-- /Proxy
|   |   |-- BankAccountProxy.cs
|   |
|   |-- /Implementations
|   |   |-- FinancialObjectFactory.cs
|
|-- /Util
|   |-- NonClosingStringWriter.cs
|
|-- Program.cs
|-- usings.cs
|-- FinanceApp.csproj
|-- README.md
|
/FinanceApp.Tests
|-- BasicTests.cs
```

```

|-- BankAccountTests.cs
|-- CategoryTests.cs
|-- OperationTests.cs
|-- AnalyticsTests.cs
|-- PerformanceTests.cs
|-- FileOperationsTests.cs
|-- ImportExportInterfaceTests.cs
|-- MoreTests.cs
|-- YamlImportTests.cs
|-- DomainModelTests.cs
|-- ComplexIntegrationTests.cs
|-- FactoryTests.cs
|-- OperationDetailedTests.cs
|-- OperationFacadeTests.cs
|-- AnalyticsAdvancedTests.cs
|-- EdgeCaseTests.cs

```

3 Принципы SOLID и GRASP в проекте

3.1 Принципы SOLID

3.1.1 S - Single Responsibility Principle (Принцип единственной ответственности)

Этот принцип реализован во множестве классов проекта:

- **Доменные классы** (BankAccount, Category, Operation) - отвечают только за свои данные и поведение без внешней логики.
- **Фасады** (BankAccountFacade, CategoryFacade, OperationFacade, AnalyticsFacade) - каждый из них отвечает только за операции с соответствующим типом объектов.
- **Экспортеры/Импортёры** (CsvExportVisitor, JsonExportVisitor, CsvImport, JsonImport) - каждый класс отвечает только за свой формат данных.

Пример из кода BankAccountFacade:

```

// BankAccountFacade is responsible only for bank account
// operations
public class BankAccountFacade
{
    private readonly Dictionary<int, BankAccount> _accounts =
        new Dictionary<int, BankAccount>();
    private readonly FinancialObjectFactory _factory;

```

```

public BankAccountFacade(FinancialObjectFactory factory)
{
    _factory = factory ?? throw new ArgumentNullException
(nameof(factory));
}

public BankAccount CreateAccount(string name, decimal
initialBalance, AccountType type = AccountType.Checking)
{
    var account = _factory.CreateBankAccount(name,
initialBalance, type);
    _accounts[account.Id] = account;
    return account;
}

// Other methods for account operations
}

```

3.1.2 O - Open/Closed Principle (Принцип открытости/закрытости)

Этот принцип реализован в нескольких компонентах:

- **IVisitor и его реализации** - система экспорта позволяет добавлять новые форматы без изменения существующего кода.
- **ImportTemplate** - абстрактный шаблонный класс, закрытый для изменений, но открытый для расширения через наследование.
- **ICommand и его реализации** - команды для различных операций можно добавлять без изменения кода, использующего интерфейс.

Пример с реализацией паттерна Посетитель (Visitor):

```

// Visitor interface is open for new implementations
public interface IVisitor
{
    void Visit(BankAccount account);
    void Visit(Category category);
    void Visit(Operation operation);
}

// CsvExportVisitor - concrete implementation
public class CsvExportVisitor : IVisitor
{

```

```

private readonly StringBuilder _sb = new StringBuilder();

public void Visit(BankAccount account)
{
    _sb.AppendLine($"BankAccount;{account.Id};{account.
Name};{account.Balance}");
}

// Implementations for other types...

public string GetCsvResult() => _sb.ToString();
}

```

3.1.3 L - Liskov Substitution Principle (Принцип подстановки Лисков)

Этот принцип реализован в классах, которые наследуют общие абстракции:

- **ImportTemplate** и его наследники (CsvImport, JsonImport, YamlImport) - корректно реализуют абстрактные методы.
- **ICommand** и его реализации - любая команда может использоваться через интерфейс.
- **TimeMeasureDecorator** - корректно расширяет функциональность ICommand.

Пример с шаблонным методом:

```

// Base abstract class with template method
public abstract class ImportTemplate
{
    public void ImportFile(string path)
    {
        var fileContent = File.ReadAllText(path);
        var parsedData = ParseData(fileContent);
        ProcessData(parsedData);
    }

    protected abstract object ParseData(string fileContent);
    protected abstract void ProcessData(object data);
}

// Descendant correctly implements abstract methods
public class CsvImport : ImportTemplate
{

```

```

protected override object ParseData(string fileContent)
{
    return fileContent.Split('\n');
}

protected override void ProcessData(object data)
{
    var lines = data as string[];
    foreach(var line in lines ?? Array.Empty<string>())
    {
        Console.WriteLine($"Importing CSV line: {line}");
    }
}
}

```

3.1.4 I - Interface Segregation Principle (Принцип разделения интерфейсов)

Этот принцип применен к интерфейсам проекта:

- **IVisitor** - содержит только необходимые методы для посещения различных типов объектов.
- **ICommand** - имеет минимально необходимый набор методов (только Execute).
- **IBankAccountProxy** - содержит только методы, необходимые для кэширования банковских счетов.

Пример интерфейса ICommand:

```

// Minimal command interface
public interface ICommand
{
    void Execute();
}

```

3.1.5 D - Dependency Inversion Principle (Принцип инверсии зависимостей)

Этот принцип реализован через:

- **Внедрение зависимостей** - фасады и другие сервисы принимают необходимые зависимости через конструктор.

- **Зависимость от абстракций** - классы зависят от интерфейсов и абстрактных классов, а не конкретных реализаций.
- **DI-контейнер** - в Program.cs используется ServiceCollection для управления зависимостями.

Пример внедрения зависимостей:

```
// Class depends on abstraction (interface), not concrete
// implementation
public class OperationFacade
{
    private readonly Dictionary<int, Operation> _operations =
        new Dictionary<int, Operation>();
    private readonly FinancialObjectFactory _factory;
    private readonly BankAccountFacade _accountFacade;

    public OperationFacade(FinancialObjectFactory factory,
        BankAccountFacade accountFacade)
    {
        _factory = factory ?? throw new ArgumentNullException
            (nameof(factory));
        _accountFacade = accountFacade ?? throw new
            ArgumentNullException(nameof(accountFacade));
    }

    // Class methods...
}
```

3.2 Принципы GRASP

3.2.1 High Cohesion (Высокая связность)

- **Доменные классы** - каждый класс имеет высокую связность, содержит только свои данные и методы.
- **Фасады** - группируют родственные операции (например, операции со счетами в BankAccountFacade).
- **AnalyticsFacade** - объединяет всю аналитическую функциональность в одном месте.

3.2.2 Low Coupling (Низкая связанность)

- **Использование интерфейсов** - компоненты взаимодействуют через интерфейсы (ICommand, IVisitor).

- **Фасады** - служат точками входа и скрывают внутреннюю реализацию, снижая связанность.
- **Внедрение зависимостей** - зависимости явно передаются через конструктор.

3.2.3 Creator (Создатель)

- **FinancialObjectFactory** - централизованно создает доменные объекты.
- **Фасады** - управляют созданием объектов через фабрику, у которой есть вся необходимая информация.

3.2.4 Controller (Контроллер)

- **Program** - координирует действия на высоком уровне, делегируя их фасадам.
- **Фасады** - служат контроллерами для операций с соответствующими типами объектов.

3.2.5 Information Expert (Информационный эксперт)

- **Доменные классы** - имеют всю информацию для вычисления своих свойств.
- **AnalyticsFacade** - собирает информацию из разных источников для расчета аналитики.

4 Паттерны GoF в проекте

4.1 Порождающие паттерны

4.1.1 Factory (Фабрика)

Реализация: Класс **FinancialObjectFactory** в файле `FinanceApp/Services/Implementati`

Обоснование важности: Фабрика централизует создание объектов, что позволяет:

- Инкапсулировать логику создания объектов
- Обеспечить валидацию входных параметров

- Генерировать уникальные идентификаторы для объектов
- Упростить добавление новых типов объектов в будущем

```
public class FinancialObjectFactory
{
    private int _nextAccountId = 1;
    private int _nextCategoryId = 1;
    private int _nextOperationId = 1;

    public BankAccount CreateBankAccount(string name, decimal
        initialBalance, AccountType type = AccountType.Checking)
    {
        if (initialBalance < 0)
            throw new ArgumentException("Initial balance
                cannot be negative", nameof(initialBalance));

        return new BankAccount(_nextAccountId++, name,
            initialBalance, type);
    }

    // Methods for creating other objects...
}
```

4.2 Структурные паттерны

4.2.1 Facade (Фасад)

Реализация: Классы `BankAccountFacade`, `CategoryFacade`, `OperationFacade`, `AnalyticsFacade` в директории `FinanceApp/Services/Facade/`

Обоснование важности: Фасады предоставляют высокоуровневый интерфейс к подсистемам:

- Скрывают сложность внутренней реализации
- Предоставляют единую точку входа для операций над определенным типом объектов
- Повышают читаемость и поддерживаемость кода
- Упрощают взаимодействие с доменными объектами

```
public class AnalyticsFacade
{
    private readonly OperationFacade _operationFacade;
    private readonly CategoryFacade _categoryFacade;
```

```

    public AnalyticsFacade(OperationFacade operationFacade,
        CategoryFacade categoryFacade)
    {
        _operationFacade = operationFacade ?? throw new
        ArgumentNullException(nameof(operationFacade));
        _categoryFacade = categoryFacade ?? throw new
        ArgumentNullException(nameof(categoryFacade));
    }

    public decimal CalculateIncomeExpenseDifference(DateTime
start, DateTime end)
    {
        var totalIncome = _operationFacade.GetIncomeTotal(
start, end);
        var totalExpense = _operationFacade.GetExpenseTotal(
start, end);
        return totalIncome - totalExpense;
    }

    // Other analytics methods...
}

```

4.2.2 Proxy (Прокси)

Реализация: Класс **BankAccountProxy** в файле **FinanceApp/Services/Proxy/BankAccountProxy.cs**

Обоснование важности: Паттерн Прокси обеспечивает:

- Кэширование объектов в памяти для ускорения доступа
- Отложенную загрузку данных
- Контроль доступа к объектам
- Повышение производительности приложения

```

public interface IBankAccountProxy
{
    BankAccount? GetById(int id);
    BankAccount Save(BankAccount account);
    bool Remove(int id);
}

public class BankAccountProxy : IBankAccountProxy
{
    private readonly Dictionary<int, BankAccount> _cache =
new Dictionary<int, BankAccount>();
}

```

```

public BankAccount? GetById(int id)
{
    _cache.TryGetValue(id, out var acct);
    return acct;
}

public BankAccount Save(BankAccount account)
{
    _cache[account.Id] = account;
    return account;
}

public bool Remove(int id)
{
    return _cache.Remove(id);
}
}

```

4.3 Поведенческие паттерны

4.3.1 Command (Команда)

Реализация: Интерфейс **ICommand** и его реализации в директории `FinanceApp/Services/Command/`

Обоснование важности: Паттерн Команда позволяет:

- Инкапсулировать действия в объекты
- Параметризовать клиентов с разными запросами
- Создавать последовательности команд
- Реализовать транзакционность (возможность отмены операций)
- Разделить ответственность между объектами

```

public interface ICommand
{
    void Execute();
}

public class CreateOperationCommand : ICommand
{
    private readonly OperationFacade _facade;
    private readonly OperationType _type;
    private readonly int _accountId;
    private readonly decimal _amount;
}

```

```

private readonly int _categoryId;
private readonly string _description;

// Constructor with parameters...

public void Execute()
{
    _facade.CreateOperation(_type, _accountId, _amount,
DateTime.Now, _categoryId, _description);
}
}

```

4.3.2 Decorator (Декоратор)

Реализация: Класс **TimeMeasureDecorator** в файле **FinanceApp/Services/Command/Time**

Обоснование важности: Паттерн Декоратор обеспечивает:

- Динамическое добавление функциональности объектам без изменения их структуры
- Альтернативу наследованию для расширения функциональности
- Возможность комбинировать множество дополнительных поведений
- Соблюдение принципа открытости/закрытости (ОСР)

```

public class TimeMeasureDecorator : ICommand
{
    private readonly ICommand _command;
    private readonly string _commandName;

    public TimeMeasureDecorator(ICommand command, string
commandName = null)
    {
        _command = command;
        _commandName = commandName ?? command.GetType().Name;
    }

    public void Execute()
    {
        var stopwatch = Stopwatch.StartNew();

        try
        {
            _command.Execute();
        }
    }
}

```

```

        finally
        {
            stopwatch.Stop();
            Console.WriteLine($"Command execution time {
                _commandName}: {stopwatch.ElapsedMilliseconds} ms");
        }
    }
}

```

4.3.3 Template Method (Шаблонный метод)

Реализация: Абстрактный класс **ImportTemplate** в файле **FinanceApp/Services/Import/** и его наследники

Обоснование важности: Паттерн Шаблонный метод позволяет:

- Определить скелет алгоритма, делегируя конкретные шаги под-классам
- Избежать дублирования кода
- Обеспечить расширяемость алгоритма
- Сохранить контроль над последовательностью выполнения шагов алгоритма

```

// Base abstract class with template method
public abstract class ImportTemplate
{
    public void ImportFile(string path)
    {
        var fileContent = File.ReadAllText(path);
        var parsedData = ParseData(fileContent);
        ProcessData(parsedData);
    }

    protected abstract object ParseData(string fileContent);
    protected abstract void ProcessData(object data);
}

// Descendant correctly implements abstract methods
public class CsvImport : ImportTemplate
{
    protected override object ParseData(string fileContent)
    {
        return fileContent.Split('\n');
    }
}

```

```

protected override void ProcessData(object data)
{
    var lines = data as string[];
    foreach(var line in lines ?? Array.Empty<string>())
    {
        Console.WriteLine($"Importing CSV line: {line}");
    }
}
}

```

4.3.4 Visitor (Посетитель)

Реализация: Интерфейс **IVisitor** и его реализации в директории FinanceApp/Services/Exp

Обоснование важности: Паттерн Посетитель обеспечивает:

- Отделение алгоритмов от структуры объектов
- Добавление новых операций к классам без изменения их кода
- Выполнение разных операций над разными типами объектов
- Сбор связанных операций в одном классе

```

public interface IVisitor
{
    void Visit(BankAccount account);
    void Visit(Category category);
    void Visit(Operation operation);
}

public class JsonExportVisitor : IVisitor
{
    private readonly List<object> _entities = new List<object>
    >();

    public void Visit(BankAccount account)
    {
        _entities.Add(new
        {
            Type = "BankAccount",
            account.Id,
            account.Name,
            account.Balance
        });
    }

    // Other Visit methods...
}

```



```

public string GetJsonResult()
{
    return JsonSerializer.Serialize(_entities, new
JsonSerializerOptions
    {
        WriteIndented = true
    });
}
}

```

5 Тестирование проекта

Для проекта создан комплексный набор тестов, охватывающих различные аспекты системы:

- **Модульные тесты** - проверяют отдельные компоненты системы
- **Интеграционные тесты** - проверяют взаимодействие нескольких компонентов
- **Тесты производительности** - оценивают эффективность работы кода
- **Краевые случаи** - проверяют работу системы в экстремальных ситуациях

Общее покрытие кода тестами составляет более 65%, что соответствует поставленным требованиям.

6 Выводы

В рамках проекта "ВШЭ-Банк" была успешно разработана система учета личных финансов с применением современных принципов и паттернов проектирования. Проект демонстрирует:

- Четкую структуру и разделение ответственности между компонентами
- Применение принципов SOLID и GRASP на практике
- Использование различных паттернов GoF для решения конкретных задач

- Высокую тестируемость и надежность кода
- Распираемую архитектуру, готовую к добавлению новых функций

Разработанное приложение может служить хорошей основой для дальнейшего развития и расширения функциональности системы финансового учета.