ИДЗ №2 «Паттерны проектирования»

Коробов Вячеслав Сергеевич, группа 246

Инструкция по запуску

Клонирование проекта

```
git clone https://github.com/StudentSK0/FinanceTracker
cd FinanceTracker
```

Первый запуск (с созданием базы данных)

Перейдите в терминал:

```
cd FinanceTracker
dotnet build
dotnet run --project Finance.App
```

При первом запуске выполняется автоматическая инициализация:

- создаётся SQLite база данных: `finance.db`;
- создаются таблицы: `Accounts`, `Categories`, `Operations`;
- система готова к использованию.

Путь сохранения экспортируемых файлов

Результаты экспорта помещаются в каталоги:

- `out/json` — экспорт в формате JSON;
- `out/csv` — экспорт в формате CSV.

Сценарий использования

Здесь показан пример запуска программы с созданием какого-то счета и демонстрацией основных функций программы. Ниже рассказывается про паттерны проектирования в проекте

После запуска приложения отображается главное меню. Выберем пункт 1:

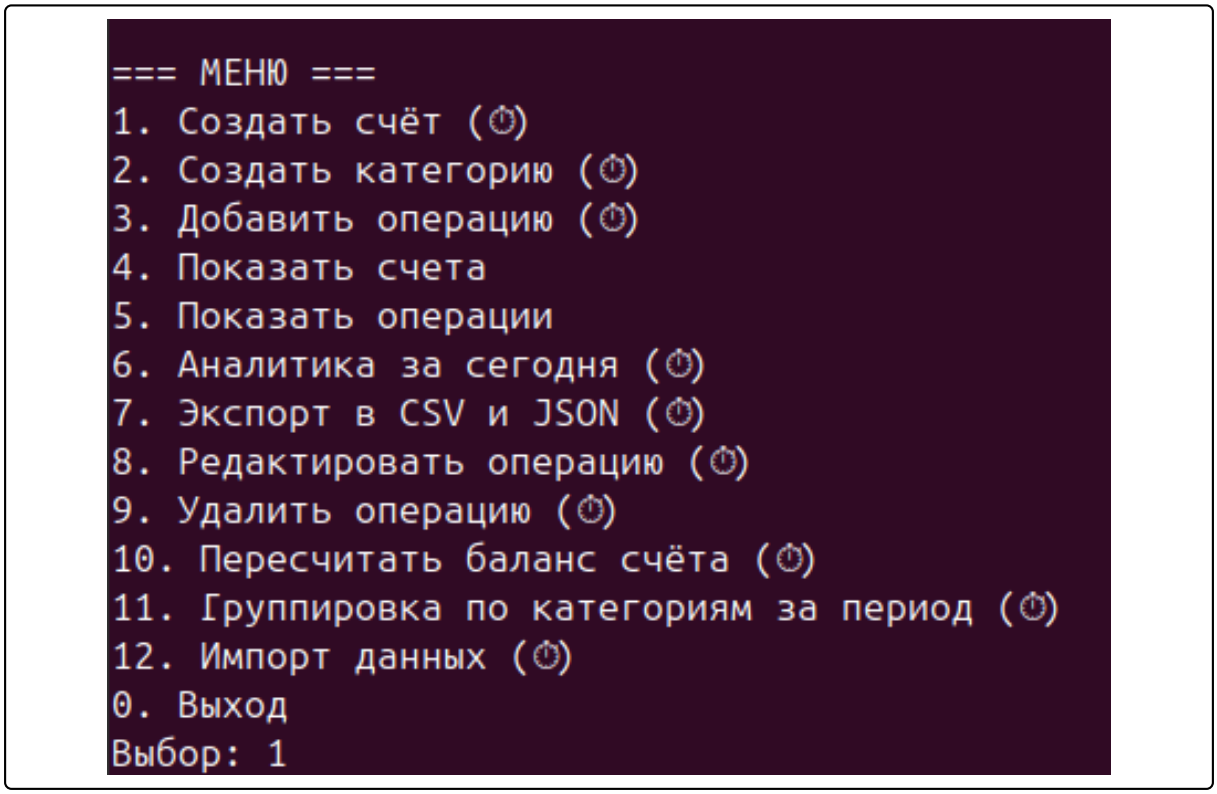

Рис. 1: Меню

Теперь можно ввести данные для создания счета:

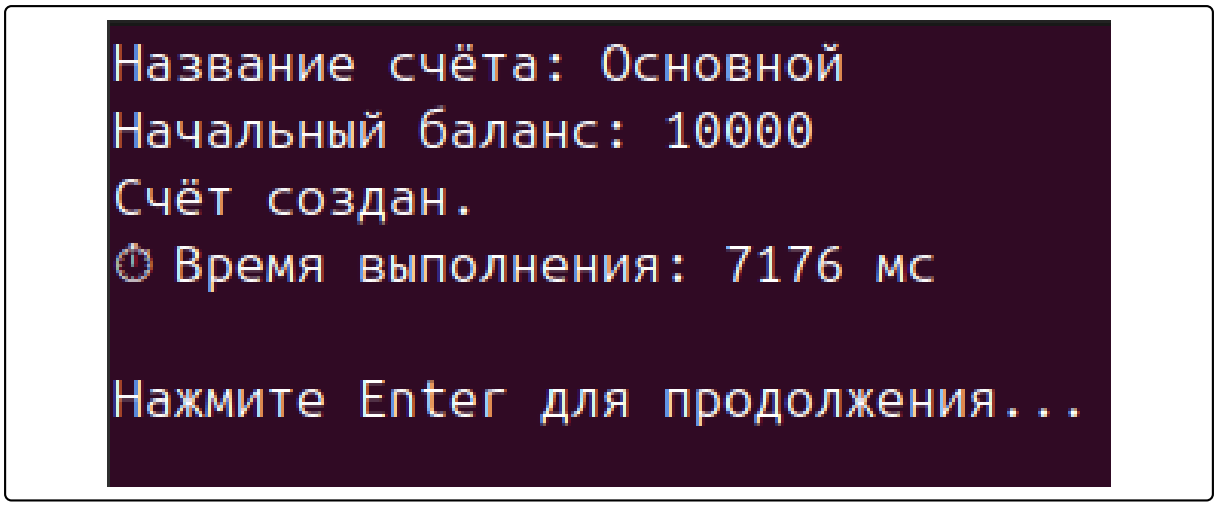

Рис. 2: Создания счёта

Открываем таблицу `Accounts` в DB Browser for SQLite. В системе появляется счёт, который доступен для операций доходов и расходов:

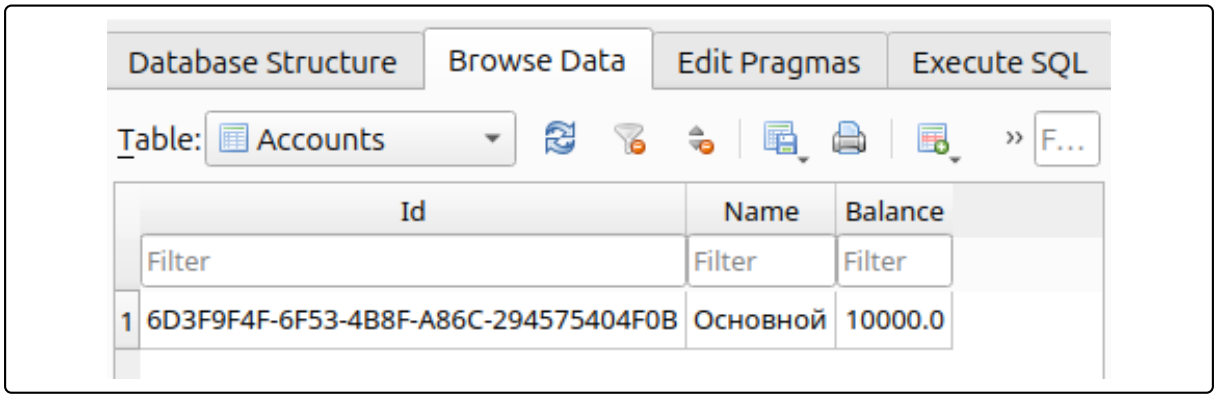

Рис. 3: Наш счёт в базе данных

Далее выберем пункт 2 (создание категории) и создадим, например, такую категорию:

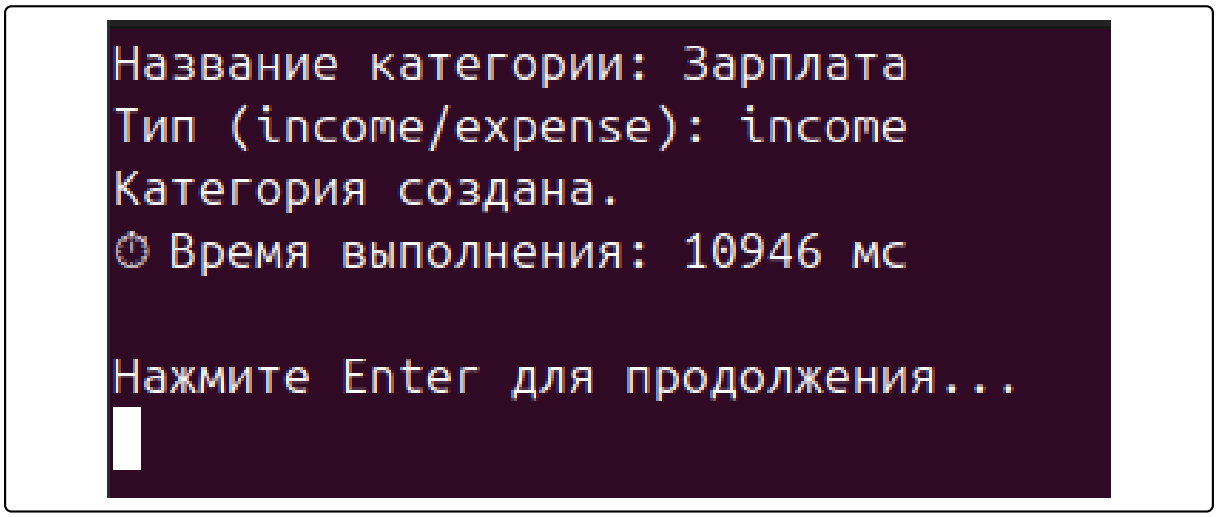

Рис. 4: Пример категории

Добавим операцию с типом income:

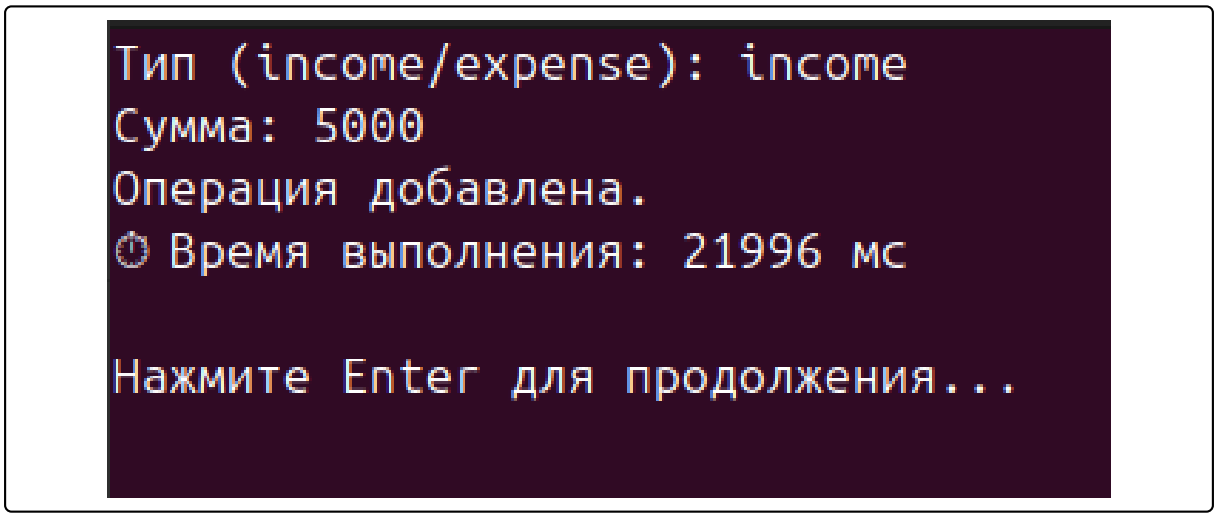

Рис. 5: Добавление операции

Наш баланс автоматически пересчитывается (используется фасад и репозиторий с корректировкой), поэтому в базе данных мы уже видим:

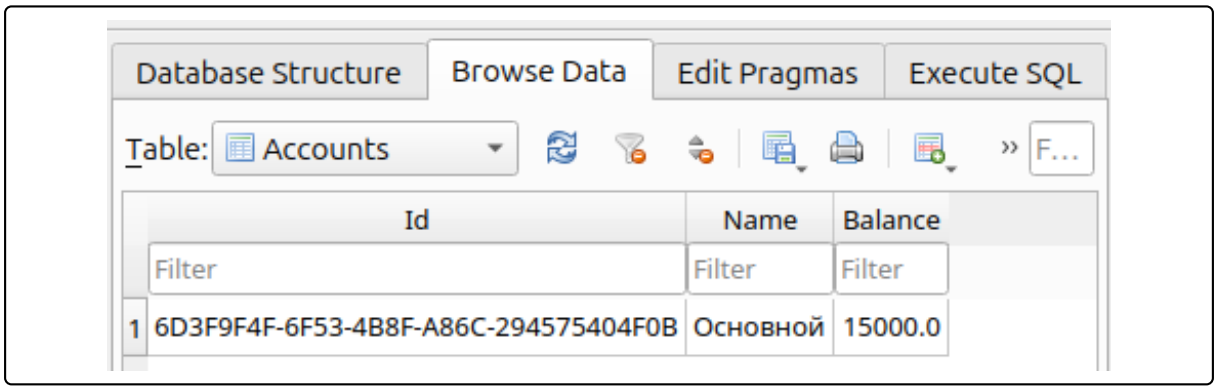

Рис. 6: Баланс уже 15000, а не 10000

Мы можем отредактировать операцию (пункт 8) и изменение тоже сразу отобразится в базе данных:

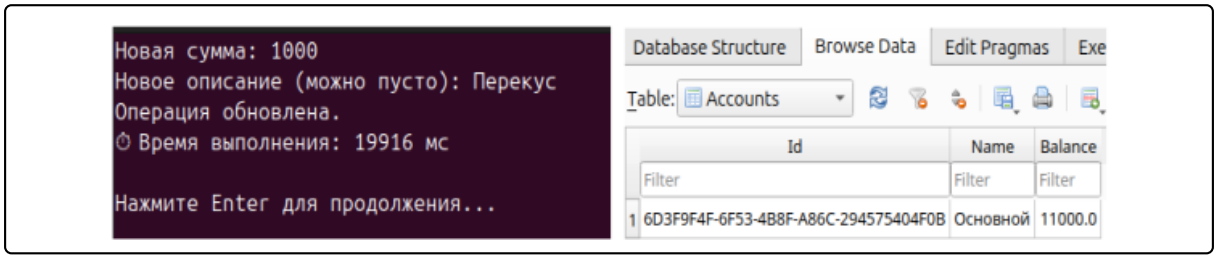

Рис. 7: Редактирование операции

Ниже показана реализация паттернов проектирования

a. Общая идея решения

Разработан модуль учёта финансов, состоящий из доменной модели, фасадов предметной логики и инфраструктуры хранения данных (SQLite + Dapper + репозитории), а также консольного пользовательского интерфейса на основе паттерна «Команда» и системы импорта/экспорта данных.

Область	**Возможности**
Счета	Создание, переименование, удаление, пересчёт баланса, просмотр
Категории	Создание, переименование, удаление, просмотр
Операции	Создание, редактирование, удаление с автоматической корректировкой баланса
Аналитика	Расчёт доходов/расходов за период, группировка по категориям
Экспорт	JSON и CSV (Visitor)
Импорт	JSON, YAML, CSV (Template Method)
UI	Меню на паттерне «Команда» + измерение времени выполнения (Декоратор)

Важная особенность: баланс счёта корректируется при изменении операции и может быть повторно вычислен на основе всех операций.

b. Принципы SOLID

Принцип	Где реализован	Суть реализации	Почему важно
S	Доменные сущности	Только бизнес-логика, без UI/БД	Тестируемость и простота
O	Импорт/экспорт через наследование	Новые форматы добавляются без изменения существующего кода	Расширяемость
L	IExportVisitor / ImporterBase	Любая реализация взаимозаменяема	Ослабление связности
I	Разделённые репозитории	Интерфейсы не перегружены	Минимизация зависимостей
D	Фасады зависят от интерфейсов	Хранилище заменяемо (SQLite → PostgreSQL)	Модульность

Пояснение реализации принципов SOLID

S — Single Responsibility (Принцип единственной ответственности)

Каждая доменная сущность отвечает только за свою собственную предметную роль и не содержит логики, относящейся к хранению данных, пользовательскому интерфейсу или аналитике.

Пример — класс `BankAccount`:

Файл: `Finance.Domain/Entities/BankAccount.cs`

```
public sealed class BankAccount
{
    public Guid Id { get; }
    public string Name { get; private set; }
    public decimal Balance { get; private set; }

    public void Rename(string newName) => Name = newName;

    public void Apply(Operation op)
        => Balance += op.Type == OpType.Income ? op.Amount : -op.Amount;

    public void Revert(Operation op)
        => Balance -= op.Type == OpType.Income ? op.Amount : -op.Amount;

    public void SetBalance(decimal value) => Balance = value;
}
```

Почему SRP соблюдён:

- `BankAccount` хранит состояние счёта и знает правила его изменения.
- Он *не* зависит от БД, UI или сервисов экспорта.
- Это упрощает тестирование и делает модель устойчивой к изменениям.

То же верно и для других сущностей:

Класс	**Ответственность**
`Category`	Хранит имя и тип категории (доход/расход)
`Operation`	Описывает параметры финансовой транзакции
`AccountFacade`	Организует сценарии работы со счётом, не содержит бизнес-логики счёта

O — Open/Closed (Открыт для расширения, закрыт для изменения)

При добавлении нового формата импорта или экспорта существующий код изменять не нужно.

Файл: `Finance.Infrastructure/Import/ImporterBase.cs`

```
public abstract class ImporterBase : IImporter
{
    public void Import(string path)
    {
        var dto = Load(path); // точка расширения
        Upsert(dto);          // общая логика
    }

    protected abstract RootDto Load(string path);
}
```

Добавление нового формата:

```
public sealed class ExcelImporter : ImporterBase
{
    protected override RootDto Load(string path) { ... }
}
```

Таким образом система развивается через наследование, не нарушая существующий код.

L — Liskov Substitution (Подстановка Лисков)

Все импортеры реализуют одинаковый контракт, поэтому их можно взаимозаменять:

```
_import.Import(_json, path);
_import.Import(_yaml, path);
_import.Import(_csv, path);
```

Замена одного импорта на другой не нарушает работу системы — это и есть соблюдение принципа LSP.

I — Interface Segregation (Разделение интерфейсов)

Вместо одного большого репозитория используются специализированные:

```
public interface IAccountRepo
{
    void Add(BankAccount acc);
    BankAccount Get(Guid id);
    IEnumerable<BankAccount> List();
    void Update(BankAccount acc);
    void Remove(Guid id);
}
```

```
public interface ICategoryRepo
{
    void Add(Category cat);
    Category Get(Guid id);
    IEnumerable<Category> List();
    void Rename(Guid id, string newName);
    void Remove(Guid id);
}
```

```
public interface IOperationRepo
{
    Operation Get(Guid id);
    IEnumerable<Operation> List();
    IEnumerable<Operation> ListByAccount(Guid accountId);
    void AddAndApplyBalance(Operation op, BankAccount account);
    void RemoveAndRevertBalance(Operation op, BankAccount account);
}
```

Каждый интерфейс описывает только одну предметную роль → код чище и проще тестировать.

D — Dependency Inversion (Инверсия зависимостей)

Высокоуровневые компоненты зависят от абстракций, а не от конкретных реализаций.

```
services.AddSingleton<IAccountRepo>(sp =>
    new CachedAccountRepo(
        new DbAccountRepo(sp.GetRequiredService<DbConnectionFactory>())
    ));
```

Фасады получают интерфейсы:

```
public OperationFacade(IOperationRepo ops, IAccountRepo accounts) { ... }
```

Это позволяет заменить SQLite → PostgreSQL → InMemory без изменения доменной логики.

c. GRASP

Принцип	**Реализация**	**Польза**
Information Expert	BankAccount.Apply/Revert	Логика находится у владельца данных
Controller	Фасады предметной логики	UI остаётся «тонким»
Low Coupling	Домен не знает о БД	Слои независимы
High Cohesion	Каждая сущность выполняет одну задачу	Удобство сопровождения

Пояснение реализации принципов GRASP

Controller (Контроллер сценариев)

Каждый пользовательский сценарий оформлен как отдельная команда, реализующая `ICommand`. Команда получает ввод, вызывает фасады и не взаимодействует с БД напрямую:

```
public sealed class AddOperationCommand : ICommand
{
    private readonly AccountFacade  _accounts;
    private readonly OperationFacade _ops;

    public void Execute()
    {
        var acc = ConsoleSelector.Select(
            _accounts.List().ToList(),
            a => $"{a.Name} ({a.Balance□})"
        );

        Console.Write("Тип (income/expense): ");
        var t = Console.ReadLine()!.Trim().ToLower() == "income"
            ? OpType.Income : OpType.Expense;

        Console.Write("Сумма: ");
        var amount = decimal.Parse(Console.ReadLine()!);

        _ops.Create(t, acc.Id, amount, DateOnly.FromDateTime(DateTime.
   Today));
    }
}
```

Команда описывает *сценарий*, а не доменную логику → UI остаётся «тонким» и нечувствительным к изменениям модели.

Low Coupling (Слабая связанность)

Фасады не зависят от конкретного хранилища — только от интерфейсов:

```
public sealed class OperationFacade
{
    private readonly IOperationRepo _ops;
    private readonly IAccountRepo   _accounts;

    public OperationFacade(IOperationRepo ops, IAccountRepo accounts)
    {
        _ops      = ops;
        _accounts = accounts;
    }
}
```

Конкретные реализации (`DbOperationRepo`, `CachedAccountRepo`, SQLite, Dapper...) подключаются в `Program.cs`. Это позволяет заменить хранилище (например, на PostgreSQL) без изменения доменной логики.

High Cohesion (Высокая связность)

Каждый фасад обслуживает одну сущность и не смешивает обязанности:

```
public sealed class AccountFacade
{
    public BankAccount Create(string name, decimal openingBalance = 0m) {
    … }
    public void Rename(Guid id, string newName) { … }
    public void Delete(Guid id) { … }
    public decimal RecalcBalance(Guid accountId) { … }
    public IEnumerable<BankAccount> List() { … }
}
```

Фасад не содержит логики категорий, аналитики или операций → область ответственности остаётся чёткой.

Information Expert (Эксперт по данным)

Изменение баланса выполняет именно владелец данных — `BankAccount`:

```
public void Apply(Operation op)
{
    Balance += op.Type == OpType.Income ? op.Amount : -op.Amount;
}

public void Revert(Operation op)
{
    Balance -= op.Type == OpType.Income ? op.Amount : -op.Amount;
}
```

Логика живёт рядом с данными → код проще, корректнее и устойчивее.

Indirection (Промежуточное звено)

Взаимодействие слоя UI и БД проходит через фасады:

UI → Команда → Фасад → Репозиторий → БД

Это защищает UI и команды от изменений инфраструктуры.

Pure Fabrication (Искусственный объект)

Кэширующие репозитории (`CachedAccountRepo`, `CachedCategoryRepo`) не являются частью предметной модели. Они созданы для оптимизации производительности, не нарушая бизнес-логику.

d. Паттерны GOF

Паттерн	**Реализация**	**Назначение**
Facade	Все фасады доменной логики	Скрытие сложных сценариев
Command	Finance.App/Commands/*	Каждое действие пользователя — объект
Decorator	TimedCommand	Добавление измерения времени без изменения команд
Template Method	ImporterBase	Повторное использование алгоритма импорта
Visitor	IExportVisitor + Exporters	Новые форматы без изменения домена
Factory	DomainFactory	Создание валидных сущностей
Proxy (Cache)	CachedCategoryRepo и др.	Ускорение доступа к данным

Пояснение реализации принципов GoF

Фасад (Facade)

Пользователь не работает напрямую с доменными сущностями. Для взаимодействия используются фасады уровня предметной логики:

- `AccountFacade` — создание, переименование, пересчёт и получение списка счетов;
- `CategoryFacade` — управление категориями;

- `OperationFacade` — создание, обновление и удаление операций с автоматической корректировкой баланса;
- `AnalyticsFacade` — вычисление доходов, расходов и группировок;
- `ExportFacade` — подготовка данных к экспорту;
- `ImportFacade` — загрузка данных из внешних источников.

Фасады инкапсулируют использование репозиториев и скрывают внутреннюю структуру домена. Это делает систему устойчивой к изменению способа хранения и форматов данных.

Команда (Command)

Каждое действие в пользовательском интерфейсе оформлено как отдельная команда в модуле `Finance.App.Commands`. Примеры:

- `CreateAccountCommand`
- `AddOperationCommand`
- `EditOperationCommand`
- `ExportCommand`
- `ImportCommand`
- `GroupByCategoryCommand`

Команда определяет *что* должно быть сделано, но не определяет, *как* устроено хранилище или интерфейс. Это:

- упрощает тестирование (команды вызываются напрямую),
- позволяет добавлять действия без изменения `Program.cs`,
- делает сценарии пользователя явными и изолированными.

Декоратор (Decorator)

Для добавления измерения времени выполнения команд используется обёртка `TimedCommand`:

```
public sealed class TimedCommand : ICommand
{
    private readonly ICommand _inner;
    public string Name => _inner.Name;

    public void Execute()
    {
        var sw = Stopwatch.StartNew();
        _inner.Execute();
        sw.Stop();
        Console.WriteLine(
            $"Выполнено( за {sw.ElapsedMilliseconds} мс)"
        );
    }
}
```

Поведение команды расширяется без изменения её кода — это характерный признак паттерна «Декоратор».

Шаблонный метод (Template Method)

Общий алгоритм импорта данных вынесен в `ImporterBase`:

```
Import()
 ├─ Load()    // определяется в наследниках (JSON/YAML/CSV)
 └─ Upsert()  // обновление домена: общее для всех форматов
```

Конкретные импортеры переопределяют только загрузку формата, не дублируя логику обновления. Это обеспечивает расширяемость и соблюдает принцип DRY.

Посетитель (Visitor)

Для экспорта используется паттерн «Посетитель». Фасад вызывает экспортёр через единый интерфейс:

```
public interface IExportVisitor
{
    void Visit(
        IEnumerable<BankAccount> acc,
        IEnumerable<Category>   cat,
        IEnumerable<Operation>  ops);
}
```

Добавление нового формата (например, XML или Excel) не требует изменения доменных сущностей или фасада.

Прокси с кэшированием (Proxy)

Для ускорения чтения используются кэширующие репозитории:

- `CachedAccountRepo`
- `CachedCategoryRepo`

Кэш хранит часто запрашиваемые данные, чтение выполняется из памяти, а запись обновляет и БД, и кэш. Интерфейс хранилища остаётся прежним → никаких изменений в остальном коде.