# Введение

Перед разработчиками приложения всегда встаёт вопрос, как создать систему объектов, способную смоделировать конкретную сложную систему. По мнению М. Фаулера [[1](#Фау10)], основное преимущество объектной парадигмы состоит в облегчении понимания запутанной логики. Разработка хорошей ***модели предметной области*** (Domain Model [[2](#Вер16)]) для сложной проблемы реальной задачи является фундаментом всего проекта. Однако, и это еще не все. Модель предметной области должна быть отображена в базе данных (БД). Причем, именно в этой последовательности: сначала продумывается архитектура приложения, а потом строится БД, сохраняющая все необходимые данные. Объектно-ориентированная модель предметной области часто похожа на соответствующую схему БД, однако, в отличие от схемы данных, помимо данных, она содержит и методы работы с ними.

Технологии ORM, такие как, LINQ to SQL, Entity Framework и NHibernate, позволяют отобразить реляционную модель базы данных, состоящую из таблиц, столбцов и ключей в объектную модель приложения, состоящую из классов и свойств.

## Важная терминология

POCO, Plain Old CLR Object. Альтернативный термин для класса в C#, который:

Не наследуются от классов, определенных в любом фреймворке (framework).

Не реализуют интерфейсы, определенные в любом фреймворке.

Не используют никаких атрибутов, определенных в любом фреймворке.

Entity – сущность. Класс, имеющий идентификатор, который определяет экземпляр на диаграмме сущностей. Этот класс является классом POCO.

Сущности, которые никогда не были сохранены в БД, называются transient. Сущности, которые были сохранены, называются persistent.

Persistent Class – сохраняемый класс. Любой класс, который можно сохранить в одной или нескольких таблицах.

Attribute/property – атрибут/ свойство. В случае использования в классе POCO имеется в виду геттеры и сеттеры свойства или поля класса.

## Отображение Модели (Mapping)

Наша задача – разработать модель, которая является ядром приложения. Наше приложение должно работать с БД. Мы понимаем, что существует огромная разница в работе с моделью и БД. Модель – объектно-ориентированная, база данных – реляционная. Модель работает с отдельными объектами, в то время как БД – с набором данных. Это является концептуальным несоответствием между двумя подходами часто называемым «несоответствием импеданса» ***impedance mismatch***. Отображение (Mapping) является «мостом» между этим несоответствием. Отображение определяет, как данные, которые «живут» в модели внутри объектов и их свойств находят соответствие с таблицами БД и их полями, описанными в реляционной схеме БД.

Когда мы начинаем новый проект с нуля, сначала необходимо определить модель, а БД спроектировать таким образом, чтобы она «следовала» проекту. Не нужно определять сначала схему БД и пытаться поставить ею сверху над моделью. В случае, когда БД следует требованиям модели, отображение проходит легче. Мы можем использовать соглашения для большинства частей отображения, следовательно, мы однозначно определим множество деталей. В экстренном случае мы можем даже позволить фреймворку автоматически создать отображение за нас.

М. Фаулер даёт следующую формулировку архитектурному шаблону проектирования ***отображение данных*** (Data Mapper): слой отображателей (Mapper), который осуществляет передачу данных между объектами и БД, сохраняя последние независимыми друг от друга и от самого отображателя [[1](#Фау10)].

# Пример приложения

## Создание схемы базы данных

Работу рассмотрим на примере разработки приложения «Библиотека». Для выбранной предметной области создадим схему БД. Все сущности, поля и ключи будут иметь одинаковые названия и будут построены на одинаковых доменах.

Схема БД «Библиотека» представлена на следующем рисунке (см. Рисунок 1). БД имеет следующие таблицы:

Книга (Book);

Автор (Author);

КнигаАвтор (AuthorBook);

Издательство (Publishing);

Город (City);

Серия (Series);

Комната (Room);

Полка (Shelf).



Рисунок 1 – ER-диаграмма

# Разработка приложения

Для демонстрации работы с библиотекой создадим консольное приложение. Наше приложение будет содержать несколько проектов, в которых разместим классы, разделённые по функциям.

## Создание решения и репозитория

В IDE Visual Studio 2022 выберите вкладку Создание нового проекта (Рисунок 2).

Изображение выглядит как текст, снимок экрана, программное обеспечение, веб-страница

Автоматически созданное описание

Рисунок  – Создание нового проекта

В диалоговом окне выберете тип проекта (Рисунок 3).

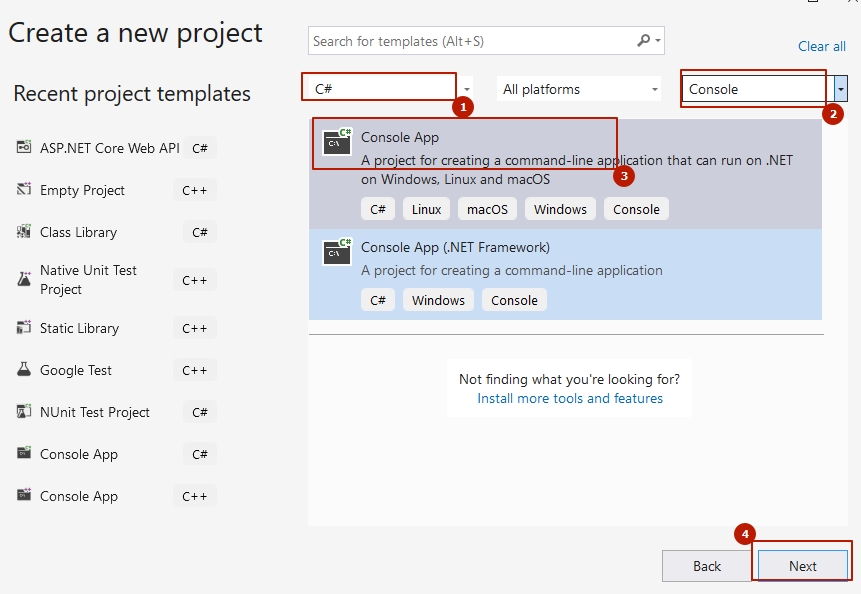


Рисунок  – Выбор типа проекта

После нажатия кнопки дайте имя проекту (Project Name) и решению (Solution Name) (Рисунок 4).

Изображение выглядит как текст, снимок экрана, Шрифт, число

Автоматически созданное описание

Рисунок  – Конфигурация проекта

Нажмите кнопку далее (Next).

Откроется окно (Рисунок 5), в котором подтвердите дополнительную информацию о проекте.

Изображение выглядит как текст, снимок экрана, дисплей, программное обеспечение

Автоматически созданное описание

Рисунок  – Дополнительная информация о проекте

Наше решение имеет один проект Demo. Для дальнейшей работы добавим еще два проекта: библиотека классов, описывающая работу предметной области, и библиотеку тестовых классов на эту предметную область.

Добавьте новый проект в решение (Рисунок 6). Выберите тип проекта (Рисунок 7). Отконфигурируйте проект (Рисунок 8) и подтвердите дополнительною информацию о проекте (Рисунок 9).

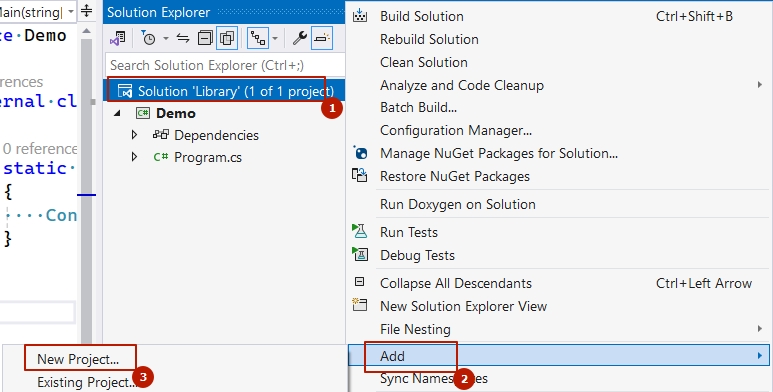


Рисунок  – Добавление нового проекта в решение

Изображение выглядит как текст, снимок экрана, программное обеспечение, число

Автоматически созданное описание

Рисунок  – Выбор типа нового проекта

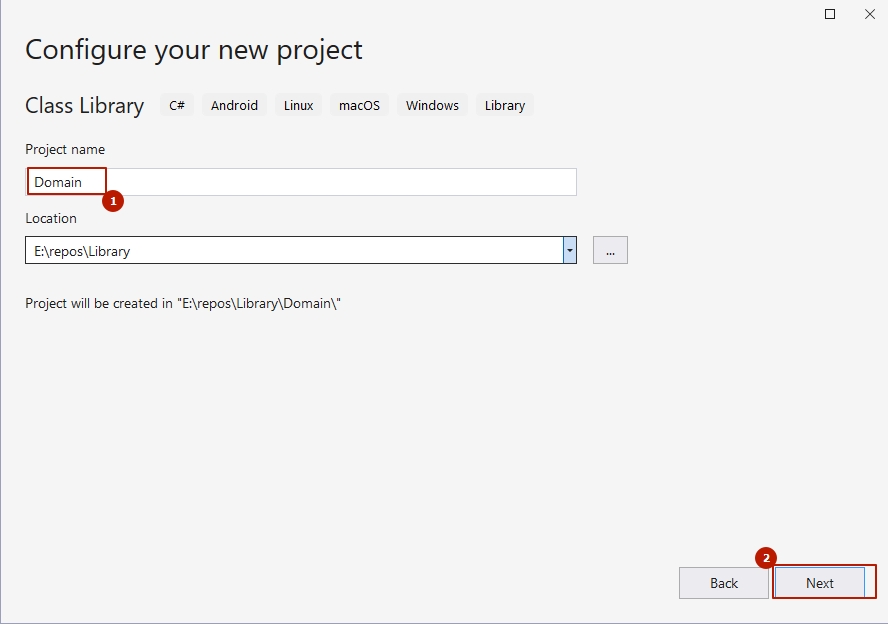


Рисунок  – Конфигурация проекта Domain

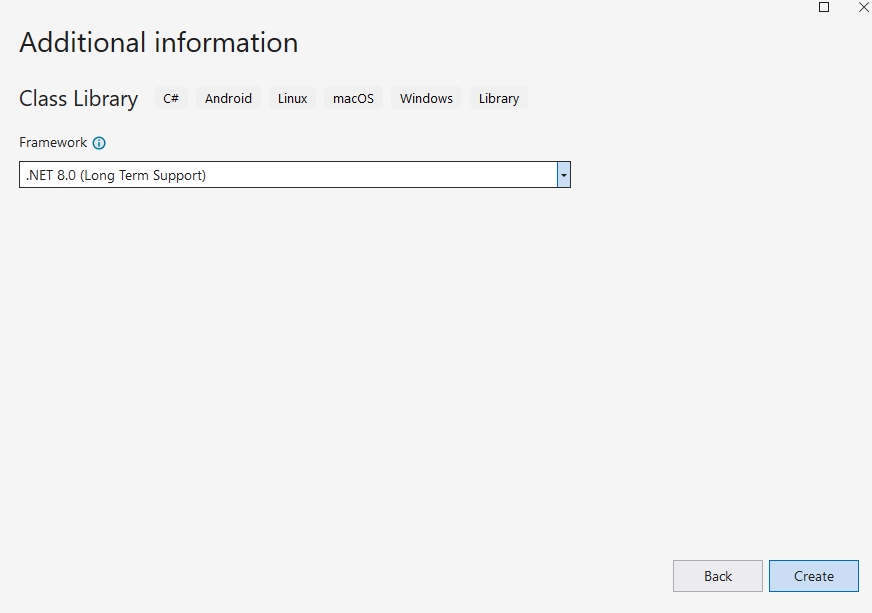


Рисунок  – Дополнительная информация для проекта Domain

Добавьте новый проект в решение. Данный проект будет тестовым (Рисунок 10). Выполните настройку проекта (Рисунок 11, Рисунок 11).

Изображение выглядит как текст, снимок экрана, программное обеспечение, число

Автоматически созданное описание

Рисунок  – Выбор тестового проекта

Изображение выглядит как текст, снимок экрана, программное обеспечение, дисплей

Автоматически созданное описание

Рисунок  – Конфигурация проекта тестов

### StyleCop

StyleCop следит за соблюдением заданных стилистических правил в коде. Несоответствия отображаются в редакторе Visual Studio и в виде сообщений на этапе компиляции. Можно использовать как готовый набор правил, так и создать свой, в котором для каждого правила определить тип сообщения (ошибка, предупреждение и т. д.).

Для правил создадим новую папку (Рисунок 12).

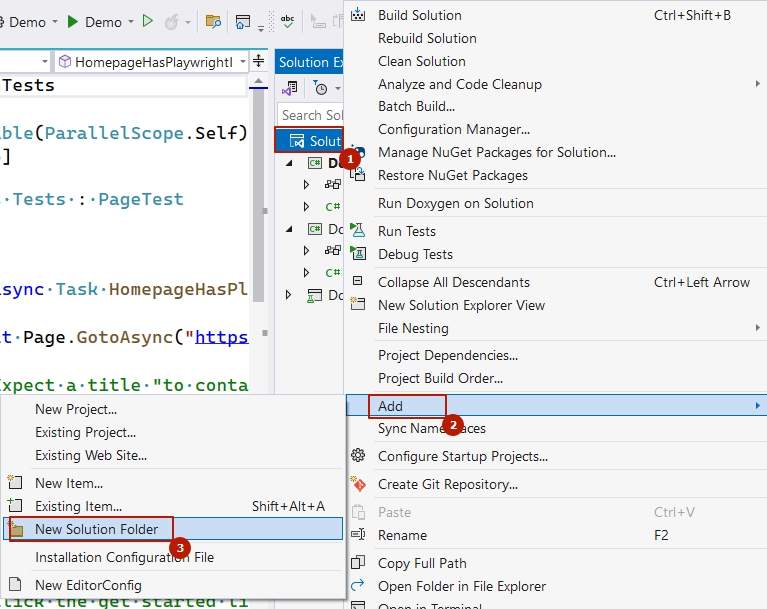


Рисунок  – Добавление новой папки к проекту

Добавим в наше решение зависимость от StyleCop с помощью NuGet пакета (Рисунок 13).

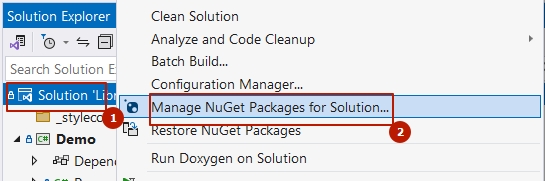


Рисунок  – Управление пакетами

В открытом окне менеджера пакетов выберите указанные ниже опции (Рисунок 14). Установите StyleCop (Рисунок 15, Рисунок 16).

Изображение выглядит как текст, снимок экрана, Шрифт, программное обеспечение

Автоматически созданное описание

Рисунок  – Выбор StyleCop

Изображение выглядит как текст, снимок экрана, дисплей, программное обеспечение

Автоматически созданное описание

Рисунок  – Выбор проектов для StyleCop

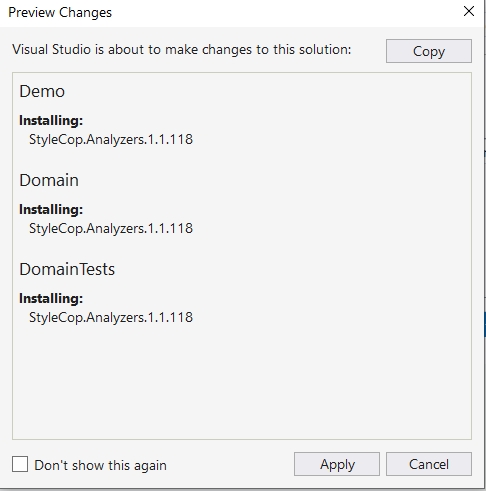


Рисунок  – Подтверждение установки зависимостей

После правильного подключения StyleCop в проектах выявлены ошибки (Рисунок 17).

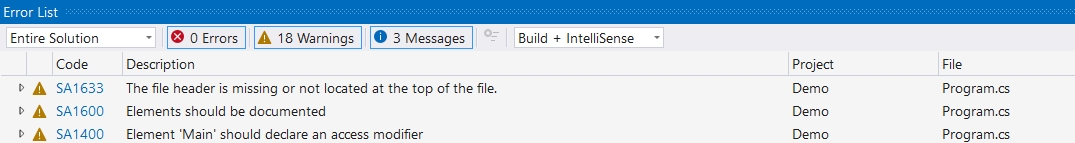


Рисунок  – Ошибки, выявленные StyleCop

В примерах, данных на лекциях, имеются отредактированные правила для проектов типа develop и tests. их можно скачать из соответствующих репозиториев (<https://github.com/MarinaPaley/Library-2024>) (Рисунок 18, Рисунок 19, Рисунок 20) в папку -stylecop.

Изображение выглядит как текст, снимок экрана, число, Шрифт

Автоматически созданное описание

Рисунок  – Имеющиеся правила для StyleCop

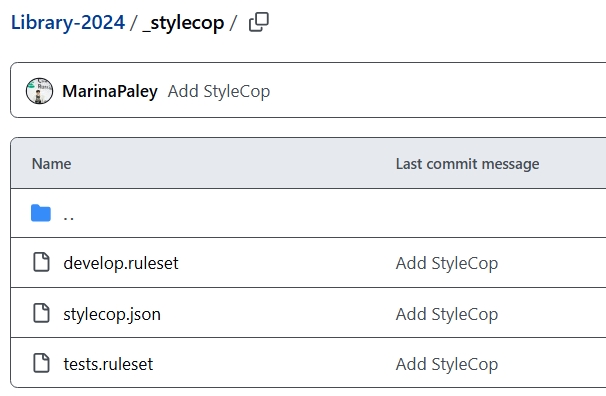


Рисунок  – Правила для StyleCop

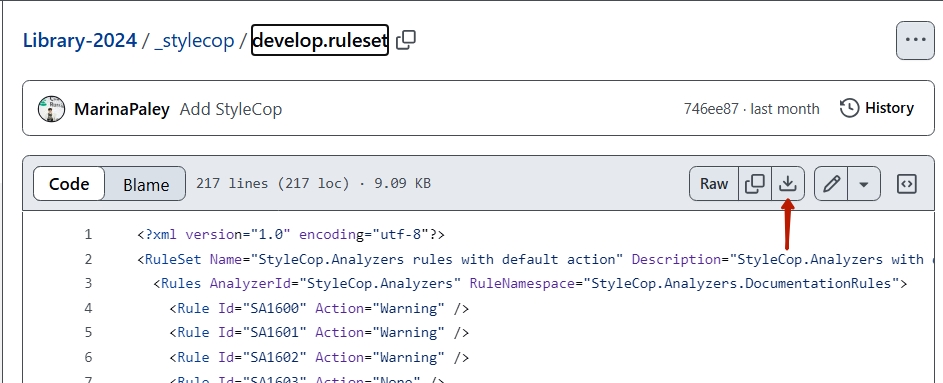


Рисунок  – Скачивание файлов

Для того, чтобы скачанные правила были применены, добавьте в файлы проектов, соответственно, правила для разработки и тестов, следующие строки:

для разработки

<PropertyGroup>  
 <CodeAnalysisRuleSet>$(SolutionDir)\_stylecop\  
develop.ruleset</CodeAnalysisRuleSet>

</PropertyGroup>  
 <ItemGroup>  
 <AdditionalFiles Include="$(SolutionDir)\_stylecop\stylecop.json" Link="\_stylecop\stylecop.json" />  
 <AdditionalFiles Include="$(SolutionDir)\_stylecop\develop.ruleset" Link="\_stylecop\develop.ruleset" />  
 </ItemGroup>

для тестов

<PropertyGroup>  
 <CodeAnalysisRuleSet>$(SolutionDir)\_stylecop\  
tests.ruleset</CodeAnalysisRuleSet>  
 </PropertyGroup>  
 <ItemGroup>  
 <AdditionalFiles Include="$(SolutionDir)\_stylecop\stylecop.json" Link="\_stylecop\stylecop.json" />  
 <AdditionalFiles Include="$(SolutionDir)\_stylecop\tests.ruleset" Link="\_stylecop\develop.ruleset" />  
 </ItemGroup>

Откройте файл stylecop.json в любом текстовом редакторе для того, чтобы исправить имя разработчика и название компании.

Наше решение показано ниже (Рисунок 21).

Изображение выглядит как текст, программное обеспечение, Значок на компьютере, веб-страница

Автоматически созданное описание

Рисунок  – Решение со связями на правила StyleCop

После того, как выполнили вышеуказанные действия, исправьте все ошибки и предупреждения, выносимые StyleCop.

### Добавление решения в систему контроля версий

Внизу студии выберете значок Добавление в систему контроля версий (Add to Source Control) (Рисунок 22).

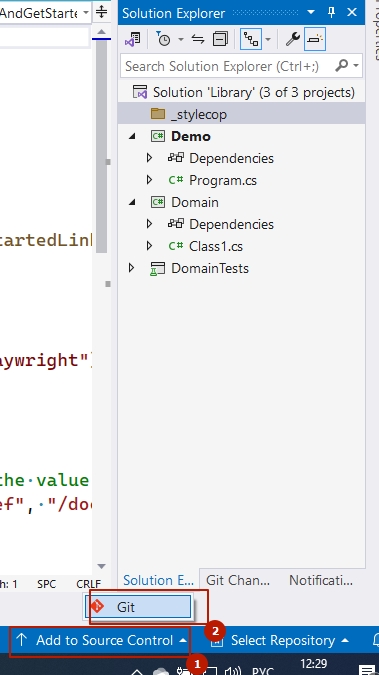


Рисунок  – Добавление в систему контроля версий

Далее настройте параметры удаленного репозитория (Рисунок 23).

Изображение выглядит как текст, снимок экрана, программное обеспечение, число

Автоматически созданное описание

Рисунок  – Выбор параметров репозитория

После нажатия на кнопку «Создать и Отправить» (Create and Push) (Рисунок 23), в веб-хостинге GitHub будет создан новый репозиторий (Рисунок 24).

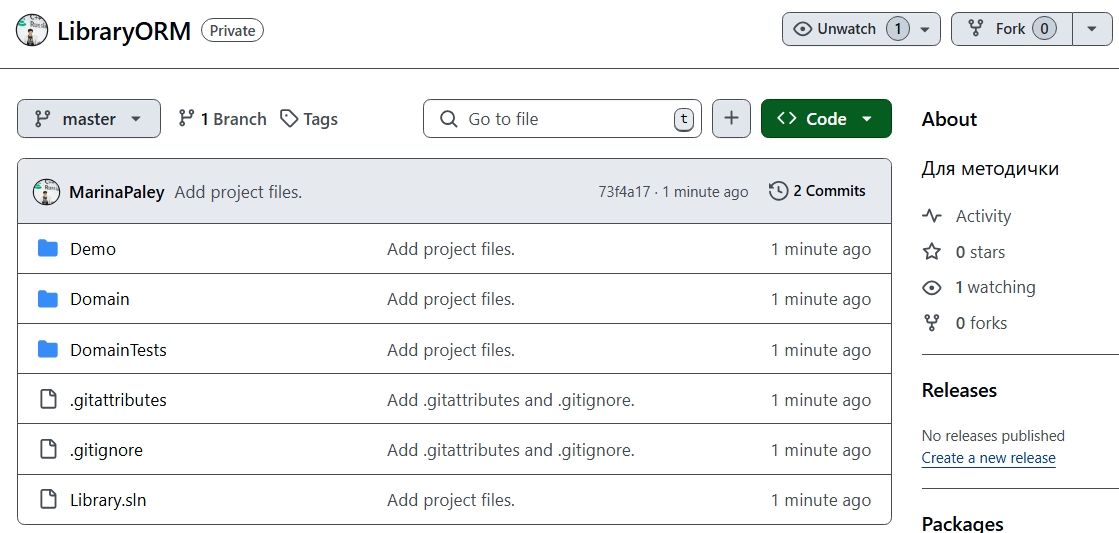


Рисунок  – Созданный репозиторий

В новом репозитории создайте issue, над которым собираетесь работать (Рисунок 25).

Изображение выглядит как текст, снимок экрана, Шрифт, линия

Автоматически созданное описание

Рисунок  – Создание задания

После этого вернитесь в локальный репозиторий. В IDE Visual Studio создайте ветку для работы над заданием (Рисунок 26, Рисунок 27).

Изображение выглядит как текст, снимок экрана, программное обеспечение, веб-страница

Автоматически созданное описание

Рисунок  – Создание новой ветки

Изображение выглядит как текст, снимок экрана, программное обеспечение, линия

Автоматически созданное описание

Рисунок  – Создание новой ветки

## Разработка библиотеки классов (Domain)

### Выделение основных классов

Для выполнения минимального функционала наше приложение должно иметь три класса: Автор, Полка и Книга. На одной полке может находиться несколько книг. Каждая книга может быть написана в соавторстве. Каждый автор может написать несколько книг.

Для каждой сущности необходимо написать соответствующий ей класс (для сущностей, созданных только для того, чтобы разорвать связь «многие-ко-многим» класс создавать не нужно).

### Создание сущностного класса Автор

Класс Автор должен содержать сведения о фамилии, имени и отчеству автора. Отчество может отсутствовать. Может содержаться информация о годах жизни. Сведения о ФИО решено не редактировать.

Автор должен уникально содержаться в библиотеке. Исходя из этого описания принято решение о том, что класс будет реализовывать интерфейс IEquatable<T>.

Ниже (листинг) представлена заготовка для класса Автор.

namespace Domain  
{  
 using System;  
  
 public class Author : IEquatable<Author>  
 {  
 public Guid Id { get; }  
 public string FamilyName { get; }  
 public string FirstName { get; }  
 public string? PatronicName { get; }  
 public DateOnly? DateBirth { get; set; }  
  
 public bool Equals(Author? other)  
 {  
 throw new NotImplementedException();  
 }  
  
 public bool Equals(Author? other)  
 {  
 throw new NotImplementedException();  
 }  
  
 public override bool Equals(object obj)  
 {  
 return Equals(obj as Author);  
 }  
  
 public override int GetHashCode()  
 {  
 throw new NotImplementedException();}  
 }  
}

Создадим конструктор. Для этого можно выбрать правой кнопкой мыши действие Quick Actions and Refactoring и указать какие свойства необходимо инициализировать в конструкторе и нужно ли проверять на null.

Полученный код для конструктора показан ниже.

public Author(  
 string familyName,  
 string firstName,  
 string? patronicName = null,  
 DateOnly? dateBirth = null,  
 DateOnly? dateDeath = null)  
{  
 this.Id = Guid.Empty;  
 this.FamilyName = familyName ?? throw new  
 ArgumentNullException(nameof(familyName));  
 this.FirstName = firstName ?? throw new  
 ArgumentNullException(nameof(firstName));  
 this.PatronicName = patronicName;  
 this.DateBirth = dateBirth;  
 this.DateDeath = dateDeath;  
}

Для проверки строки на то, что она не null и не пустая можно использовать метод string.IsNullOrEmpty(value), который возвращает тип bool. Затем проверить данное выражение и, если строка валидная, присвоить соответствующее значение полю. А можно написать метод расширение для строкового типа. Так как этот метод не относится к предметной области, а является вспомогательным, создадим для таких методов отдельный проект – библиотека классов и назовем его Staff.

Ниже представлен код для класса StringExtensions.

public static class StringExtensions  
{  
 public static bool IsNullOrEmpty(this string? value)  
 => string.IsNullOrEmpty(value);  
 public static string? TrimOrNull(this string value)  
 {  
 var trimmed = value?.Trim();  
 return trimmed.IsNullOrEmpty()  
 ? null  
 : trimmed;  
 }  
}

Для того, чтобы использовать в проекте Domain созданный метод, добавим зависимостей в решение.

Проект Demo создан для того, чтобы взаимодействовать с пользователем. В проекте Domain будут находиться классы, ответственные за бизнес-логику. Проект Staff содержит вспомогательные классы. Проект DomainTests будет содержать тесты на логику. Следовательно, все проекты, кроме Staff, зависят от Domain. А зависит от Staff.

#### Добавление зависимостей

Выберите проект в Обозревателе Решений (Solution Explorer), правой кнопкой мыши нажмите на опцию Зависимости (Dependencies) (Рисунок 28, Рисунок 29, Рисунок  30, Рисунок 31).

Изображение выглядит как текст, снимок экрана, программное обеспечение, Значок на компьютере

Автоматически созданное описание

Рисунок  – Добавление зависимостей

Изображение выглядит как текст, снимок экрана, программное обеспечение, дисплей

Автоматически созданное описание

Рисунок  – Зависимости проекта Demo

Изображение выглядит как текст, снимок экрана, дисплей, программное обеспечение

Автоматически созданное описание

Рисунок   – Зависимости проекта DomainTests

Изображение выглядит как текст, снимок экрана, дисплей, программное обеспечение

Автоматически созданное описание

Рисунок 31 – Зависимости проекта Domain

Перепишем код для конструктора, используя созданный метод расширения.

public Author(  
 string familyName,  
 string firstName,  
 string? patronicName = null,  
 DateOnly? dateBirth = null,  
 DateOnly? dateDeath = null)  
{  
 this.FamilyName = familyName .TrimOrNull() ?? throw new  
 ArgumentNullException(nameof(familyName));  
 this.FirstName = firstName .TrimOrNull() ?? throw new  
 ArgumentNullException(nameof(firstName));  
 this.PatronicName = patronicName;  
 this.DateBirth = dateBirth;  
 this.DateDeath = dateDeath;  
}

#### Переопределение метода Equals()

Переопределим метод Equals()

public override bool Equals(object? obj)  
{  
 return this.Equals(obj as Author);  
}

public bool Equals(Author? other)  
{  
 return other is not null  
 && this.FamilyName == other.FamilyName  
 && this.FirstName == other.FirstName  
 && this.PatronicName == other.PatronicName  
 && this.DateBirth == other.DateBirth  
 && this.DateDeath == other.DateDeath;  
}

#### Переопределение метода GetHashCode()

Переопределение метода Equals() влечет за собой и переопределение метода GetHashCode()

public override int GetHashCode() =>  
 HashCode.Combine(this.FamilyName, this.FirstName, this.PatronicName,  
 this.DateBirth, this.DateDeath);

#### Переопределение метода ToString()

public override string ToString()  
{  
 return this.PatronicName is null  
 ? $"{this.FamilyName} {this.FirstName}"  
 : $"{this.FamilyName} {this.FirstName} {this.PatronicName}";  
}

### Тесты для класса Автор

Тестирование метода Equals().Тест проверяет утверждение. Что два разных автора не эквивалентны.

[Test]  
public void Equals\_DifferentAuthors\_False()  
 {  
 // Arrange  
 var author1 = new Author("Толстой", "Лев", "Николаевич");  
 var author2 = new Author("Пушкин", "Александр", "Сергеевич");  
 // Act & Assert  
 Assert. That(author1, Is.Not.EqualTo(author2));  
 }

Если требуется проверить, выбрасывает ли метод исключение, то в качестве примера приведем тест на конструктор с неопределенными параметрами. В данной предметной области валидными параметрами приняты имя и фамилия с заданными значениями. Если одно из них null, то ожидается исключение. Для проверки в одном случае двух вариантов использован атрибут TestCase.

Так как ожидается (обычно, когда нет ошибок), что конструктор создает объект, а созданный объект не будет использоваться, то в качестве подсказки компилятору вместо имени объекта в лямбда-функции пишем discard (нижнее подчеркивание).

[TestCase(null, "")]  
[TestCase("", null)]  
public void Ctor\_WrongData\_ExpectedException(string? familyName, string? firstName)  
{  
 Assert.Throws<ArgumentNullException>(  
 () => \_ = new Author(familyName!, firstName!));  
}

Так как параметры конструктора принимают не nullable значения, но для теста передаем null, то для отсутствия предупреждений (warnings) в конце каждого параметра ставим восклицательный знак (точно значение не null).

Для проверки того, что авторы с точки зрения бизнес-логики, у которых отличаются даты смерти и рождения, разные, напишем следующий тест.

[TestCaseSource(nameof(ValidNullDates))]  
public void Equals\_SimilarAuthorsDifferentDates\_False(  
    DateOnly? dateBirth1,  
    DateOnly? dateBirth2,  
    DateOnly? dateDeath1,  
    DateOnly? dateDeath2)  
{  
    // Arrange  
    var author1 = new Author("Толстой", "Лев", "Николаевич",  
  dateBirth1, dateDeath1);  
    var author2 = new Author("Толстой", "Лев", "Николаевич",  
  dateBirth2, dateDeath2);  
  
    // Act & Assert  
    Assert.That(author1, Is.Not.EqualTo(author2));  
}

private static IEnumerable<TestCaseData> ValidDateData()  
{  
   yield return new TestCaseData(new DateOnly(1828, 09, 28), null);  
   yield return new TestCaseData(null, new DateOnly(1910, 10, 20));  
   yield return new TestCaseData(null, null);  
}

Тестовые данные сформированы в статическом методе ValidDateData() в виде последовательности типа IEnumerable<TestCaseData>. В тестовый метод эти данные поступают в качестве параметров, взятых из [TestCaseSource(nameof(ValidNullDates))].

На каждый публичный метод следует написать тесты с различными исходами и отладить бизнес-логику.

### Создание остальных сущностных классов

После того, как написана логика класса Автор, и все тесты на этот класс проходят, объединим рабочую и главную ветки. Для нового класса создадим свою рабочую ветку и issue.

### Создание сущностного класса Полка (Shelf)

public sealed class Shelf : IEquatable<Shelf>  
{  
    public Shelf(string name)  
    {  
        this.Id = Guid.Empty;  
        this.Name = name.TrimOrNull() ?? throw   
 new ArgumentNullException(nameof(name));  
 }  
public Guid Id { get; }  
public string Name { get; }  
}

Для класса Полка переопределим методы Equals(), GetHashCode() ToString(). Напишем тесты на публичный API.

### Создание сущностного класса Книга (Book)

public sealed class Book : IEquatable<Book>  
    {  
    public Book(string title, int pages, string ibsn)  
        {  
            this.Title = title.TrimOrNull() ?? throw   
            new ArgumentNullException(nameof(title));  
            ArgumentOutOfRangeException.ThrowIfNegativeOrZero(pages);  
  
            this.Pages = pages;  
            this.IBSN = ibsn.TrimOrNull() ?? throw   
            new ArgumentNullException(nameof(ibsn));  
            this.Id = Guid.Empty;  
        }  
public Guid Id { get; }  
public string Title { get; }  
public int Pages { get; }  
public string IBSN { get; }  
}

Для класса Книга переопределим методы Equals(), GetHashCode() ToString(). Напишем тесты на публичный API.

Мы хотим реализовать основной класс для всех сущностей

### Определение связей между сущностями

#### Владеет или содержит

Значимые объекты (value object) никогда не существуют в одиночестве. Они становятся значимыми в соединении с сущностью. Сущность может владеть или содержать нуль или много значимых объектов. В нашем примере (*Библиотека*) имеется отношение *Автор* (Author), с полями *Имя* (FirstName), *Фамилия* (LastName) и *Отчество* (PatronicName). Создадим значимый класс Name с указанными полями:

public Name(  
 string familyName,  
 string firstName,  
 string? patronicName = null)  
{  
 this.FamilyName = familyName .TrimOrNull() ?? throw new  
 ArgumentNullException(nameof(familyName));  
 this.FirstName = firstName .TrimOrNull() ?? throw new  
 ArgumentNullException(nameof(firstName));  
 this.PatronicName = patronicName;  
}

Для удобного вывода переопределим метод ToString():

public override string ToString()  
{  
    return this.PatronicName is null  
          ? $"{this.FamilyName} {this.FirstName}"  
          : $"{this.FamilyName} {this.FirstName} {this.PatronicName}";  
}

Напишем тесты для класса Name.

Далее исправим класс Author. Создадим открытое свойство FullName типа Name:

public Name FullName { get; }  
public Author(  
 string fullName,   
 DateOnly? dateBirth = null,  
 DateOnly? dateDeath = null)  
{  
 this.FullName = fullName ?? throw new  
 ArgumentNullException(nameof(familyName));  
 this.DateBirth = dateBirth;  
 this.DateDeath = dateDeath;  
}

В нашем случае значимый объект Name содержится в сущности Author. Однако значимый объект не знает, кто является его владельцем.

#### Связь один-ко-многим

Давайте посмотри на две сущности: *Полка* (Shelf) и *Книга* (Book). Как они связаны друг с другом?

Каждая книга принадлежит точно одной полке. В дальнейшем мы сможем определить свойство Shelf типа Shelf в классе Book. Это свойство является ссылкой от книги к полке. Оно может применяться для навигации от полки к ассоциированной книге. Эта связь между книгой и полкой может быть представлена следующим кодом:

public class Book  
{

...  
 public Shelf Shelf {get; set;}  
}

Каждая полка может содержать много книг. Следовательно, мы можем определить свойство Books в классе Shelf, которое является коллекцией или набором книг. Код может выглядеть следующим образом:

public sealed class Shelf: IEquatable<Shelf>  
{...  
 public ISet<Book> Books {get; set;}  
 => new HashSet<Book>();  
}

#### Связь многие-ко-многим

Рассмотрим связь многие-ко-многим на примере: связь между автором и книгой. Каждый автор может написать несколько книг, также, книга может быть написана в соавторстве. В нашем случае код будет следующим:

public sealed class Book: IEquatable<Book>  
{   
 public ISet<Author> Authors {get; set;}  
 => new HashSet<Author>();  
}  
public sealed class Author: IEquatable< Author >  
{  
 public ISet<Book> Books {get; set;}  
 => new HashSet<Book>();  
}

Т. к. в БД может существовать автор, который не написал ни одной книги, но книга, у которой нет автора, существовать не может, то главной таблицей будет таблица *Автор* (Author). Процедура добавления новой книги в класс *Автор* описана ниже:

public bool AddBook(Book book)  
{  
 if (book is null)  
 {  
 return false;  
 }  
  
 if (this.Books.Add(book))  
 {  
 \_ = book.Authors.Add(this);  
 return true;  
 }  
  
 return false;  
}

Процедура удаления книги из класса *Автор* описана ниже:

public bool RemoveBook(Book book)  
 {  
 if (book is null)  
 {  
 return false;  
 }  
  
 if (this.Books.Remove(book))  
 {  
 book.Authors.Remove(this);  
 return true;  
 }  
  
 return false;  
 }

Аналогично запишем удаление и добавление Книги на Полку.

Для Книги изменим конструктор. Добавим параметры Полка и Список Авторов.

public Book(  
 string title,  
  int pages,  
  string ibsn,  
  ISet<Author> authors,  
  Shelf? shelf = null)  
       {  
  
 foreach (var author in authors)  
            {  
                author.AddBook(this);  
            }  
  
            this.Shelf = shelf;  
            if (this.Shelf is not null)  
            {  
                \_ = this.Shelf.AddBook(this);  
            }  
        }

Для удобства использования добавим конструктор с params:

public Book(  
 string title,  
 int pages,  
 string ibsn,  
 Shelf? shelf = null,  
 params Author[] authors)  
        : this(title, pages, ibsn, new HashSet<Author>(authors), shelf)  
        {  
        }

Также перепишем метод ToString() у классов Книга и Полка, чтобы отображались Авторы у Книги и Книги у Полки. Для этого напишем класс-расширение для IEnumerable<T> и поместим его в проект Staff:

public static class EnumerableExtensions  
{  
    public static string Join<T>(  
 this IEnumerable<T> values, string separator = ", ")  
       => string.Join(separator, values);  
}

public override string ToString()  
{  
    return this.Authors.Count > 0  
    ? $"{this.Title} {this.Authors.Join()}"  
    : this.Title;

}

### Выделение базового класса Person

Класс Автор может быть наследником класса Человек (Person). В данной предметной области у книги может быть переводчик, редактор, рецензент и все эти сущности наследуются от базового класса Человек.

public abstract class Person : IEqualityComparer<Person>  
{  
protected Person(  
 string familyName,  
 string firstName,  
 string? patronicName = null,  
 DateOnly? dateBirth = null,  
 DateOnly? dateDeath = null)  
{  
 this.Id = Guid.Empty;  
 this.FamilyName = familyName ?? throw new  
 ArgumentNullException(nameof(familyName));  
 this.FirstName = firstName ?? throw new  
 ArgumentNullException(nameof(firstName));  
 this.PatronicName = patronicName;  
 this.DateBirth = dateBirth;  
 this.DateDeath = dateDeath;  
}

public bool Equals(Person? lha, Person? rha)  
        {  
            if (lha is null || rha is null)  
            {  
                return false;  
            }  
            return lha.Equals(rha);  
        }  
public int GetHashCode([DisallowNull] Person obj) => obj.GetHashCode();

В этот класс перенесем из класса Автор реализованные методы Equals(), GetHashCode(), ToString().

Класс Автор теперь наследуется от Человека:

public sealed class Author : Person  
{  
 public Author(  
            Name fullName,  
            DateOnly? dateBirth = null,  
            DateOnly? dateDeath = null)  
            : base(fullName, dateBirth, dateDeath)  
        {  
        }  
}

## Data Access Layer

Изображение выглядит как текст, программное обеспечение, Значок на компьютере, Операционная система

Автоматически созданное описание

Рисунок

Изображение выглядит как текст, снимок экрана, программное обеспечение, веб-страница

Автоматически созданное описание

Рисунок

Изображение выглядит как текст, снимок экрана, программное обеспечение, Значок на компьютере

Автоматически созданное описание

Рисунок

Изображение выглядит как текст, снимок экрана, программное обеспечение, веб-страница

Автоматически созданное описание

Рисунок

Изображение выглядит как текст, снимок экрана, программное обеспечение, веб-страница

Автоматически созданное описание

Рисунок

Изображение выглядит как текст, снимок экрана, программное обеспечение, веб-страница

Автоматически созданное описание

Рисунок

Изображение выглядит как текст, снимок экрана, Шрифт, программное обеспечение

Автоматически созданное описание

Рисунок

Изображение выглядит как текст, снимок экрана, программное обеспечение, веб-страница

Автоматически созданное описание

Рисунок

Изображение выглядит как текст, снимок экрана, программное обеспечение, веб-страница

Автоматически созданное описание

Рисунок

Изображение выглядит как текст, снимок экрана, программное обеспечение, Значок на компьютере

Автоматически созданное описание

Рисунок

Изображение выглядит как текст, снимок экрана, программное обеспечение, число

Автоматически созданное описание

Рисунок

Изображение выглядит как текст, снимок экрана, Шрифт, программное обеспечение

Автоматически созданное описание

Рисунок

Изображение выглядит как текст, снимок экрана, программное обеспечение, Значок на компьютере

Автоматически созданное описание

Рисунок

### Отображение (Mapping)

### Транзакции

Транзакция – это концепция, которая позволяет нам работать с множеством заданий как с одним блоком. Результат выполнения транзакции такой же, если бы все составные шаги были бы выполнены успешно, а, если хотя бы один шаг не выполнился, то вся система возвратилась в исходное состояние, как будто ничего и не происходило.

Транзакция обладает следующими характеристиками (ACID – atomic, consistent, isolated, durable):

*Атомарность* – операция должна быть выполнена как одно целое и не может быть разделена на меньшие выполняемые части.

*Согласованность* – результат выполнения транзакции должен оставлять систему в согласованном состоянии.

*Изолированность* – операция должна проходить изолированно, т.е. незаконченные транзакции не должны быть видны извне.

*Долговечность* – результат операции является долговечным.

Способ организации бизнес-логики по процедурам, каждая из которых обслуживает один запрос, инициируемый слоем представления, назван М. Фаулером (**сценарий транзакции** Transaction Script[[1](#Фау10)]). Сценарий транзакций организует логику вычислительного процесса в виде единой процедуры, которая обращается к БД напрямую или при помощи кода тонкой оболочки. Каждой транзакции ставится в соответствие собственный сценарий транзакции [[1](#Фау10)].

### CRUD

CRUD (create, read, update, delete) ‑ акроним для четырех основных функций БД (создание, чтение, обновление и удаление).

## Запросы

Объект запроса (Query Object) – типовое решение объектно-реляционного отображения - объект, представляющий запрос к БД [[1](#Фау10)]. Применение SQL ‑ выражений для выполнения запросов к БД в приложении ведёт не только к неудобству работы с таким приложением, так как не все разработчики хорошо владеют языком запросом. Также приложение с применением «сырого SQL» подвержено SQL инъекциям. Особенностью объекта запроса является возможность представления запросов в терминах объектов приложения, а не базы данных [[1](#Фау10)].

### LINQ

Внедрение LINQ в .NET 3.5 изменило способ взаимодействия разработчиков с набором данных. До LINQ цикл foreach устанавливал правила игры, когда речь шла о взаимодействии с наборами или списками данных. Сейчас эту роль играет LINQ. Центральным для LINQ to Object является интерфейс IEnumerable<T>. LINQ to Object манипулирует набором объектов, находящихся в памяти. LINQ to NHibernate схожим образом взаимодействует с БД. Центральным интерфейсом для LINQ to NHibernate является интерфейс IQuerable<T>.

## Создание Хранилища (Repository)

Data Access Object (DAO) ‑ широко распространенный паттерн для сохранения объектов бизнес-области в базе данных. В самом широком смысле, DAO ‑ это класс, содержащий CRUD методы для конкретной сущности. Когда к функциям приложения относится, помимо основных операций, выполнение множества запросов к БД, то лучше создать ещё один слой абстракции. В этот слой помещается весь код, написанный на SQL, который необходимо осуществить в терминах объектов. Типовое решение объектно-реляционного отображения Хранилище (Repository) выступает в роли посредника между слоем домена и слоем отображения данных, предоставляя интерфейс в виде коллекции для доступа к объектам домена [[1](#Фау10)].

Для создания Хранилища определим интерфейс:

public interface IRepository<T>  
 {  
 IList<T> GetAll();  
 T GetById(GUID id);  
 void Save(T entity);  
 void Delete(T entity);  
 }

# Список литературы

x

|  |  |
| --- | --- |
| 1. | Фаулер М. . Шаблоны корпоративных приложений. Москва: Издательский дом "Вильямс", 2014. |
| 2. | Вернон В. . Реализация методов предметно-ориентированного проектирования. Москва: Вильямс, 2016. |
| 3. | Dentler J. . NHibernate 3.0 Cookbook. Birmingham: Packt Publishing Ltd., 2010. |
| 4. | Троелсен Э. , Джепикс Ф. . Язык программирования C# 6.0 и платформа.NET 4.6. Москва: Издательсткй дом "Вильямс", 2016. |

x