ФГАОУ ВО «КФУ им. В.И. Вернадского»
Физико-технический институт (структурное подразделение)

Кафедра компьютерной инженерии и моделирования

Скибинский Дмитрий Константинович

отчет по практической работе №7
по дисциплине **«ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ»**

Направление подготовки:
09.03.04 "Программная инженерия"

Оценка 90

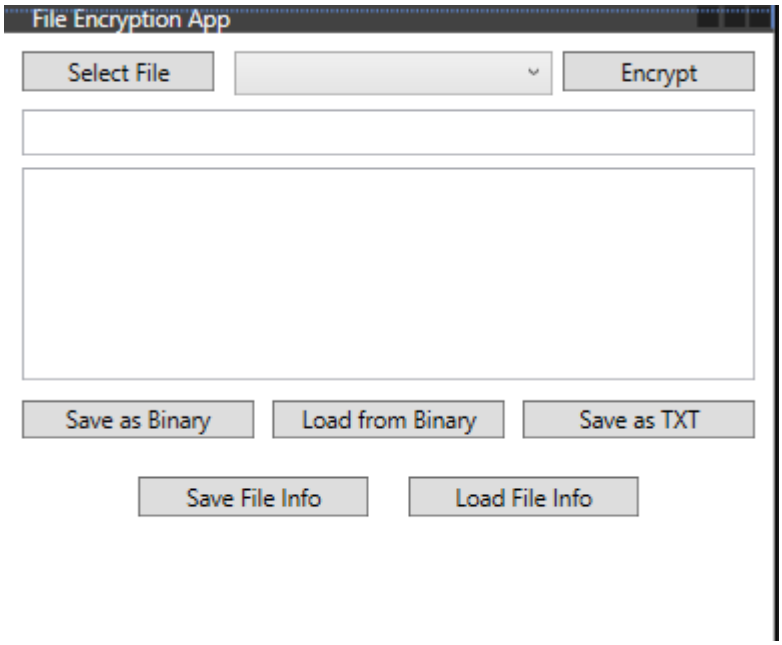
Симферополь, 2024

Практическая работа №7
Тема: Практическое программирование. Работа с файлами. Паттерны проектирования ПО на C#

Цель работы: Изучить приемы практического программирования с использованием файлов. Изучить основные паттерны проектирования по книге Эриха Гаммы, Ричарда Хелма, Ральфа Джонсона, Джона Влиссидеса ("Банда четырёх"). "Приемы объектно-ориентированного проектирования. Паттерны проектирования", разобраться с вопросом целесообразности использования паттернов в крупных проектах, научиться реализации паттернов на языке программирования C#.

Задание 1. Реализовать приложение, использующее классы File, FileInfo и FileStream, StreamReader и StreamWriter, BinaryWriter и BinaryReader.

Для выполнения этого задания сделаем приложение на WPF, которое будет принимать файл txt и шифровать его с помощью методов MD5 и SHA256. Для начала сделаем пользовательский интерфейс



Сделаем функцию, которая позволяет нам выбрать файл на нашем компьютере. По кнопки мы сможем выбрать такой файл, который мы хотим зашифровать. При помощи FileInfo мы можем определить такие параметры файла, как имя, длина, когда создали файл и когда было последнее редактирование файла

```
Ссылок: 1
private void btnSelectFile_Click(object sender, RoutedEventArgs e)
{
    var openFileDialog = new Microsoft.Win32.OpenFileDialog
    {
        Filter = "Text files (*.txt)|*.txt|All files (*.*)|*.*"
    };

    if (openFileDialog.ShowDialog() == true)
    {
        txtFilePath.Text = openFileDialog.FileName;

        // Используем FileInfo для получения информации о файле
        FileInfo fileInfo = new FileInfo(openFileDialog.FileName);
        string fileDetails = $"File Name: {fileInfo.Name}\n" +
                             $"File Size: {fileInfo.Length} bytes\n" +
                             $"Creation Time: {fileInfo.CreationTime}\n" +
                             $"Last Write Time: {fileInfo.LastWriteTime}";

        MessageBox.Show(fileDetails, "File Information");
    }
}
```

Добавим метод, который будем проверять выбран ли файл и выбран ли тип шифрования. Он будет считывать содержимое файла при помощи File.ReadAllText.

```
Ссылок: 1
private void btnEncrypt_Click(object sender, RoutedEventArgs e)
{
    if (string.IsNullOrEmpty(txtFilePath.Text))
    {
        MessageBox.Show("Please select a file first.");
        return;
    }

    string filePath = txtFilePath.Text;

    string encryptionMethod = string.Empty;
    if (cbEncryptionMethod != null && cbEncryptionMethod.SelectedItem != null)
    {
        encryptionMethod = ((ComboBoxItem)cbEncryptionMethod.SelectedItem).Content.ToString();
    }
    else
    {
        MessageBox.Show("Please select an encryption method.");
        return;
    }

    string fileContent = File.ReadAllText(filePath);
    string encryptedContent = Encrypt(fileContent, encryptionMethod);

    txtResult.Text = encryptedContent;
}
```

Добавим метод, который сможет работать с бинарными файлами. Используя класс BinaryReader

```
private void btnLoadBinary_Click(object sender, RoutedEventArgs e)
{
    var openFileDialog = new Microsoft.Win32.OpenFileDialog
    {
        Filter = "Binary files (*.bin)|*.bin|All files (*.*)|*.*"
    };

    if (openFileDialog.ShowDialog() == true)
    {
        using (var binaryReader = new BinaryReader(File.Open(openFileDialog.FileName, FileMode.Open)))
        {
            string encryptedContent = binaryReader.ReadString();
            txtResult.Text = encryptedContent;
        }

        MessageBox.Show("Encrypted data loaded from binary file.");
    }
}
```

Для сохранения полученного текста будем также будет использоваться как бинарный оператор, так и обычный оператор File.WriteAllText

```
Ссылок: 1
private void btnSaveTxt_Click(object sender, RoutedEventArgs e)
{
    if (string.IsNullOrEmpty(txtResult.Text))
    {
        MessageBox.Show("No encrypted data to save.");
        return;
    }

    var saveFileDialog = new Microsoft.Win32.SaveFileDialog
    {
        Filter = "Text files (*.txt)|*.txt|All files (*.*)|*.*",
        DefaultExt = "txt"
    };

    if (saveFileDialog.ShowDialog() == true)
    {
        File.WriteAllText(saveFileDialog.FileName, txtResult.Text);
        MessageBox.Show("Encrypted data saved as text file.");
    }
}
```

```
private void btnSaveBinary_Click(object sender, RoutedEventArgs e)
{
    if (string.IsNullOrEmpty(txtResult.Text))
    {
        MessageBox.Show("No encrypted data to save.");
        return;
    }

    var saveFileDialog = new Microsoft.Win32.SaveFileDialog
    {
        Filter = "Binary files (*.bin)|*.bin|All files (*.*)|*.*"
    };

    if (saveFileDialog.ShowDialog() == true)
    {
        using (var binaryWriter = new BinaryWriter(File.Open(saveFileDialog.FileName, FileMode.Create)))
        {
            binaryWriter.Write(txtResult.Text);
        }

        MessageBox.Show("Encrypted data saved as binary file.");
    }
}
```

Сделаем сохранения данных FileInfo в отдельный файл уже используя более гибкие способы StreamReader и StreamWriter

```
ссылок: 1
private void btnSaveFileInfo_Click(object sender, RoutedEventArgs e)
{
    if (string.IsNullOrEmpty(txtFilePath.Text))
    {
        MessageBox.Show("Please select a file first.");
        return;
    }

    string filePath = txtFilePath.Text;

    FileInfo fileInfo = new FileInfo(filePath);
    if (!fileInfo.Exists)
    {
        MessageBox.Show("The selected file does not exist.");
        return;
    }

    var saveFileDialog = new Microsoft.Win32.SaveFileDialog
    {
        Filter = "Text files (*.txt)|*.txt|All files (*.*)|*.*",
        DefaultExt = "txt"
    };

    if (saveFileDialog.ShowDialog() == true)
    {
        using (StreamWriter writer = new StreamWriter(saveFileDialog.FileName))
        {
            writer.WriteLine($"File Name: {fileInfo.Name}");
            writer.WriteLine($"File Path: {fileInfo.FullName}");
            writer.WriteLine($"File Size: {fileInfo.Length} bytes");
            writer.WriteLine($"Creation Time: {fileInfo.CreationTime}");
            writer.WriteLine($"Last Write Time: {fileInfo.LastWriteTime}");
            writer.WriteLine($"Last Access Time: {fileInfo.LastAccessTime}");
            writer.WriteLine($"Attributes: {fileInfo.Attributes}");
        }

        MessageBox.Show("File information saved successfully.");
    }
}
```

```
ссылок: 1
private void btnLoadFileInfo_Click(object sender, RoutedEventArgs e)
{
    var openFileDialog = new Microsoft.Win32.OpenFileDialog
    {
        Filter = "Text files (*.txt)|*.txt|All files (*.*)|*.*"
    };

    if (openFileDialog.ShowDialog() == true)
    {
        using (StreamReader reader = new StreamReader(openFileDialog.FileName))
        {
            string fileInfoContent = reader.ReadToEnd();
            txtResult.Text = fileInfoContent;
        }

        MessageBox.Show("File information loaded successfully.");
    }
}
```

Задание 2. Написать в отчет о практической работе небольшой авторский реферат о паттернах проектирования:

Порождающие паттерны

Порождающие паттерны (Creational Patterns) в программировании предназначены для управления созданием объектов. Они упрощают процесс инстанцирования объектов, делая код более гибким и понятным. Рассмотрим ключевые порождающие паттерны:

I. Абстрактная фабрика (Abstract Factory)

Абстрактная фабрика — это порождающий паттерн проектирования, который предоставляет интерфейс для создания семейств взаимосвязанных объектов без указания их конкретных классов.

Идея паттерна заключается в том, чтобы вынести процесс создания объектов в отдельные фабрики, что позволяет легко заменять или добавлять новые реализации.

Основные элементы паттерна:

1. **Абстрактная фабрика**: определяет интерфейс для создания семейств связанных объектов.
2. **Конкретная фабрика**: реализует интерфейс абстрактной фабрики и создает конкретные объекты.
3. **Абстрактный продукт**: задает интерфейс для классов, объекты которых будут создаваться.
4. **Конкретный продукт**: реализация интерфейса абстрактного продукта.
5. **Клиент**: использует фабрику для создания объектов, но остается независимым от их конкретных классов.

Пример применения: GUI с разными темами

Проблема:

Вы разрабатываете приложение с графическим интерфейсом, который должен поддерживать несколько тем: светлую и темную. В каждой теме кнопки, текстовые поля и другие элементы интерфейса должны выглядеть по-разному.

Решение:

Используйте абстрактную фабрику, которая создаёт семейства элементов интерфейса для каждой темы.

II. Строитель (Builder)

Строитель — это порождающий паттерн проектирования, который позволяет пошагово создавать сложные объекты. Вместо того чтобы вынуждать клиента описывать каждый компонент объекта вручную, **Строитель** инкапсулирует логику создания внутри отдельных классов.

Этот паттерн особенно полезен, если:

- Объект имеет множество параметров.
- Создание объекта требует сложной пошаговой сборки.
- Существует необходимость в нескольких вариациях объекта.

Основные элементы паттерна:

1. **Builder (Строитель):**
   Интерфейс, определяющий этапы сборки объекта.
2. **Concrete Builder (Конкретный строитель):**
   Реализация интерфейса строителя, которая создаёт конкретный объект.
3. **Director (Директор):**
   Указывает, как именно создавать объект, используя определённого строителя.
4. **Product (Продукт):**
   Финальный сложный объект, который создаётся.

III. Фабричный метод (Factory Method)

Фабричный метод — это порождающий паттерн проектирования, который определяет интерфейс для создания объектов в суперклассе, но позволяет подклассам изменять тип создаваемых объектов.
Этот паттерн позволяет:

- Делегировать создание объектов подклассам.
- Скрыть логику создания объектов от клиента.
- Легко добавлять новые типы объектов, не нарушая существующий код.

Основные элементы паттерна:

1. **Product (Продукт):**
   Общий интерфейс или абстрактный класс для объектов, которые будут создаваться.
2. **Concrete Product (Конкретный продукт):**
   Конкретная реализация интерфейса продукта.
3. **Creator (Создатель):**
   Абстрактный класс или интерфейс, содержащий метод фабрики.
4. **Concrete Creator (Конкретный создатель):**
   Класс, реализующий фабричный метод для создания конкретных продуктов.

Пример использования: Документы
Представьте текстовый редактор, который может работать с разными типами документов, такими как **PDF** и **Word**. Вместо того чтобы привязываться к конкретным типам, редактор использует фабричный метод для создания документов.

IV. Прототип (Prototype)

Прототип — это порождающий паттерн проектирования, который позволяет копировать уже существующие объекты вместо их создания с нуля, что иногда бывает полезно для оптимизации производительности или упрощения создания сложных объектов.

Ключевая идея состоит в том, чтобы клонировать объект, а не создавать его заново. Для этого объект предоставляет метод клонирования, который возвращает копию самого себя.

Основные элементы паттерна:

1. **Прототип (Prototype):**
   Интерфейс или базовый класс, который объявляет метод для клонирования объекта (clone).
2. **Конкретный прототип (Concrete Prototype):**
   Класс, который реализует метод клонирования.
3. **Клиент (Client):**
   Использует метод клонирования для создания копий объекта.

Пример использования: Клонирование геометрических фигур
Представьте графический редактор, где есть различные фигуры (круги, прямоугольники и т. д.). Вместо создания каждой фигуры с нуля, редактор может копировать существующие.

V. Одиночка (Singleton)

Одиночка (Singleton) — это порождающий паттерн проектирования, который гарантирует, что у класса есть только один экземпляр, и предоставляет глобальную точку доступа к этому экземпляру. Этот паттерн полезен, когда требуется ограничить создание объектов определённого типа до одного, например, для управления доступом к общему ресурсу.

Основные характеристики паттерна:

1. **Единственный экземпляр:** Класс создаёт только один объект на протяжении работы программы.
2. **Глобальная точка доступа:** Экземпляр доступен из любого места в коде.

Пример использования:

- **Логгирование:** Чтобы все сообщения логировались в один файл через общий экземпляр логгера.
- **Конфигурации приложения:** Хранение настроек, которые используются во всей программе.
- **Пул соединений с базой данных:** Управление ограниченным количеством соединений.

Структурные паттерны

Структурные паттерны проектирования относятся к категории шаблонов, которые помогают организовывать классы и объекты в более сложные структуры. Они обеспечивают удобство и гибкость в управлении связями между компонентами системы.

I. Адаптер (Adapter)

Цель: Паттерн *Адаптер* используется для "перевода" интерфейса одного класса в интерфейс, понятный другому. Это позволяет объектам с несовместимыми интерфейсами работать вместе.

Применение

- Когда необходимо использовать существующий класс, интерфейс которого не соответствует вашим требованиям.
- Чтобы обеспечить взаимодействие между двумя независимыми компонентами или системами с несовместимыми интерфейсами.

Примеры из жизни

Адаптер — это как переходник для розетки: он преобразует форму вилки устройства в форму розетки, к которой она должна быть подключена. Например:

- Зарядка с разъемом USB, используемая через адаптер к розетке с другой формой вилки.
- Конвертер, который позволяет старым системам работать с новыми технологиями.

II. Мост (Bridge)

Паттерн "Мост" (Bridge) — это структурный паттерн проектирования, который разделяет абстракцию и реализацию так, чтобы их можно было изменять независимо друг от друга.

Цель: Основная задача *Моста* — избежать жесткой привязки абстракции к ее реализации, обеспечивая гибкость в разработке и расширении системы. Этот паттерн особенно полезен, если:

- Вы хотите избежать множества подклассов для сочетания различных абстракций и реализаций.
- Требуется менять реализацию без изменения клиентского кода.
- У абстракции и реализации есть свои независимые иерархии.

Пример из реального мира

Представьте, что вы строите мосты через реки. Сам мост (конструкция) — это абстракция. Тип материала (бетон, сталь) или способ строительства — это реализация. Они могут развиваться независимо:

- Конструкция: пешеходный мост, автомобильный мост.
- Материал: бетон, сталь, дерево.

Вы можете строить разные типы мостов (абстракция) с использованием разных материалов (реализация).

Структура

1. **Абстракция (Abstraction)** — интерфейс, который содержит высокоуровневую логику.
2. **Расширенная Абстракция (RefinedAbstraction)** — уточненная версия абстракции, которая использует базовый интерфейс.
3. **Реализация (Implementation)** — интерфейс, определяющий низкоуровневую реализацию.
4. **Конкретная Реализация (ConcreteImplementation)** — конкретные классы, которые реализуют интерфейс реализации.

III. Компоновщик (Composite)

Паттерн "Компоновщик" (Composite) — это структурный паттерн проектирования, который позволяет объединять объекты в древовидные структуры для представления иерархий "часть-целое". Этот паттерн дает клиенту возможность единообразно работать как с отдельными объектами, так и с их группами.

Цель: Основная задача *Компоновщика* — упростить работу с объектами и их группами. Это достигается за счет предоставления общего интерфейса для всех объектов в структуре, независимо от того, являются ли они "листами" (конечными объектами) или "композитами" (группами объектов).

Пример из реальной жизни

Дерево файловой системы:

- Файл — это "лист", который не содержит других элементов.
- Папка — это "композит", который может содержать как файлы, так и другие папки.

Клиент может запросить размер как у файла, так и у папки (включая все вложенные элементы), используя единый интерфейс.

Структура

1. **Компонент (Component)** — общий интерфейс для всех элементов структуры.
2. **Лист (Leaf)** — конечный объект, который не имеет вложенных элементов.
3. **Композит (Composite)** — объект, который может содержать другие компоненты.
4. **Клиент (Client)** — объект, который взаимодействует с компонентами через общий интерфейс.

IV. Декоратор (Decorator)

Паттерн "Декоратор" (Decorator) — это структурный паттерн, который позволяет динамически добавлять объектам новую функциональность без изменения их структуры. Декоратор предоставляет гибкий способ расширения функциональности объектов через композицию, а не наследование.

Цель: Основная цель *Декоратора* — добавление нового поведения объектам без необходимости изменять их код. Вместо того чтобы создавать многочисленные подклассы для каждого варианта поведения, вы можете использовать декораторы, которые оборачивают оригинальные объекты, добавляя новые возможности.

Пример из реальной жизни
Представьте кафе, где вы заказываете кофе:

- Базовое кофе — это простой напиток.
- Добавление молока или сахара — это декораторы, которые изменяют (расширяют) поведение основного напитка.

Идея в том, что вы можете добавить различные компоненты (молоко, сахар, сироп) к базовому кофе, не изменяя класс самого кофе.

Структура

1. **Компонент (Component)** — общий интерфейс для всех объектов, которые могут быть декорированы.
2. **Конкретный компонент (ConcreteComponent)** — объект, который будет декорироваться.
3. **Декоратор (Decorator)** — абстракция, которая реализует интерфейс компонента и содержит ссылку на объект, который декорируется.
4. **Конкретный декоратор (ConcreteDecorator)** — добавляет конкретную функциональность объекту.

V. Паттерн Фасад

Паттерн "Фасад" (Facade) — это структурный паттерн проектирования, который предоставляет упрощенный интерфейс для сложной системы или набора подсистем. Фасад скрывает сложность системы и предоставляет клиенту простой способ взаимодействовать с ней.

Цель: паттерна *Фасад* — сделать систему проще и удобнее для использования, скрывая от пользователя детали реализации и предлагая только необходимые для него методы.
Фасад не изменяет работу системы, но представляет собой интерфейс, который упрощает взаимодействие с системой или её компонентами.

Пример из реальной жизни

Представьте, что вы хотите заказать услугу через туроператора. Вместо того, чтобы иметь дело с различными агентами (перевозка, размещение, экскурсии), вы взаимодействуете с одним менеджером, который решает все вопросы за вас. Менеджер и есть фасад, который скрывает всю сложность процесса.

Структура

1. **Фасад (Facade)** — предоставляет упрощенный интерфейс для клиентского кода.
2. **Подсистемы (Subsystems)** — компоненты, которые выполняют основную работу, но их детали скрыты за фасадом.
3. **Клиент (Client)** — взаимодействует только с фасадом, не зная о внутренней сложности системы.

VI. Приспособленец (Flyweight)

Паттерн "Приспособленец" (Flyweight) — это структурный паттерн проектирования, который позволяет уменьшить количество объектов, создаваемых системой, за счет использования общего состояния для объектов с одинаковыми свойствами. Приспособленец позволяет экономить память, объединяя общие данные, а также уменьшить нагрузку на систему, если объектов много и они имеют схожие характеристики.

Цель: *Приспособленца* — минимизация затрат на создание и хранение большого числа однотипных объектов, используя общие части состояния (вместо хранения одинаковых данных в каждом объекте).
Паттерн помогает:

1. Экономить память.
2. Повторно использовать объекты, которые имеют одинаковое состояние.
3. Разделять неизменяемые и изменяемые части состояния объекта.

Пример из реальной жизни

Представьте систему, которая рисует множество одинаковых деревьев на карте. Каждое дерево имеет координаты (которые могут различаться), но другие характеристики, такие как тип дерева (например, дуб или сосна), могут быть одинаковыми для множества деревьев. Вместо того, чтобы создавать отдельный объект для каждого дерева с одинаковым типом, можно использовать один общий объект для всех одинаковых деревьев, а индивидуальные различия (например, координаты) сохранять отдельно.

Структура

1. **Приспособленец (Flyweight)** — объект, который разделяет общие данные между множеством объектов.

2. **Контекст (Context)** — хранит изменяющиеся данные, которые могут быть различными для каждого экземпляра.
3. **Фабрика (FlyweightFactory)** — управляет созданием и кэшированием приспособленцев, чтобы избежать создания дубликатов.

VII. Заместитель (Proxy)

Паттерн "Заместитель" (Proxy) — это структурный паттерн проектирования, который предоставляет объект, который контролирует доступ к другому объекту. Заместитель действует как "посредник" между клиентом и реальным объектом, часто выполняя дополнительные действия, такие как проверка прав доступа, кеширование, ленивое создание или логирование.

Цель: Основная цель *Заместителя* — предоставить интерфейс, который контролирует доступ к другому объекту. Это может включать:

- Контроль доступа (например, проверка прав пользователя).
- Оптимизацию работы (например, ленивое создание объектов).
- Логирование операций.
- Кеширование или защиту от чрезмерных операций.

Типы заместителей

1. **Виртуальный заместитель (Virtual Proxy)** — используется для управления ленивой инициализацией объектов, например, когда объект тяжелый или ресурсоемкий, и нужно отложить его создание до того, как он будет нужен.
2. **Защитный заместитель (Protection Proxy)** — используется для контроля доступа к объектам, предоставляя доступ только определенным пользователям или группам.
3. **Удаленный заместитель (Remote Proxy)** — используется для управления доступом к объектам, которые находятся в другой адресной области, например, удаленные объекты в распределенных системах.
4. **Умный заместитель (Smart Proxy)** — добавляет дополнительную функциональность, такую как подсчет ссылок на объект или управление его жизненным циклом.

Структура

1. **Предмет (Subject)** — интерфейс, который реализуют как реальный объект, так и его заместитель.
2. **Реальный объект (RealSubject)** — объект, с которым клиент на самом деле будет работать.
3. **Заместитель (Proxy)** — объект, который управляет доступом к реальному объекту.

4. **Клиент (Client)** — использует объект через интерфейс, предоставляемый заместителем.

Поведенческие паттерны

I. Цепочка обязанностей

Цепочка обязанностей (Chain of Responsibility) — это поведенческий паттерн проектирования, который позволяет передавать запрос последовательно по цепочке обработчиков. Каждый обработчик решает, может ли он обработать запрос, либо передаёт его следующему обработчику в цепочке.

Основная идея

Цепочка обязанностей используется для разделения обязанностей между объектами и обеспечения гибкости добавления новых обработчиков. Запрос отправляется первому объекту в цепочке, а тот либо обрабатывает его, либо передаёт следующему.

Структура

1. **Источник запроса** (Client) — отправляет запрос.
2. **Обработчик** (Handler) — интерфейс или абстрактный класс, который определяет метод обработки запроса и ссылку на следующий обработчик.
3. **Конкретные обработчики** (Concrete Handlers) — классы, реализующие обработку запроса. Если конкретный обработчик не может обработать запрос, он передаёт его дальше по цепочке.

II. Команда

Команда (Command) — это поведенческий паттерн проектирования, который превращает запросы в объекты, позволяя сохранять, передавать и выполнять их позднее. Основная идея заключается в том, чтобы отделить отправителя запроса от его получателя, инкапсулируя запрос как объект.

Основная идея

Паттерн **Команда** используется, когда нужно:

1. Разделить обязанности отправителя и получателя.
2. Сохранить историю выполненных операций.
3. Реализовать отмену и повтор операций.
4. Упрощать динамическое создание и исполнение запросов.

Структура

1. **Command (Команда)** — интерфейс или абстрактный класс, определяющий метод для выполнения команды.
2. **ConcreteCommand (Конкретная команда)** — реализует метод выполнения, связывая действие с получателем.
3. **Receiver (Получатель)** — объект, выполняющий действие, указанное в запросе.

4. **Invoker (Отправитель)** — инициатор выполнения команд. Хранит команды и может выполнять их.
5. **Client (Клиент)** — создает команды и настраивает их для отправителя.

III. Интерпретатор

Интерпретатор (Interpreter) — это поведенческий паттерн проектирования, который определяет способ представления грамматики для заданного языка, а также интерпретирует предложения, написанные на этом языке.

Паттерн **Интерпретатор** применяется в тех случаях, когда у нас есть формальный язык или выражения, которые нужно обработать, и мы хотим создать гибкое и легко расширяемое решение для их анализа и выполнения.

Основная идея

Интерпретатор позволяет:

- Описывать грамматику языка с помощью классов.
- Создавать интерпретатор для выполнения или вычисления выражений, заданных в этой грамматике.

Структура

1. **AbstractExpression (Абстрактное выражение)**
   Определяет интерфейс для всех узлов грамматического дерева. Обычно включает метод interpret(context).
2. **TerminalExpression (Терминальное выражение)**
   Представляет конкретные выражения, которые не разбиваются на подвыражения (например, числа или константы).
3. **NonTerminalExpression (Нетерминальное выражение)**
   Определяет правила грамматики для построения сложных выражений из других выражений.
4. **Context (Контекст)**
   Хранит данные, необходимые для интерпретации выражений (например, таблицу переменных и их значений).
5. **Client (Клиент)**
   Строит дерево синтаксиса (грамматическое дерево) и передаёт его интерпретатору.

IV. Итератор

Итератор (Iterator) — это поведенческий паттерн проектирования, который предоставляет способ последовательного доступа ко всем элементам коллекции без раскрытия её внутреннего представления.

Основная идея

Паттерн **Итератор** позволяет работать с элементами коллекции (массивов, списков, деревьев и т.д.) без необходимости знать детали её реализации. Он изолирует логику обхода коллекции в отдельный объект — итератор.

Структура

1. **Iterator (Итератор)**
   Определяет интерфейс для обхода элементов коллекции (например, методы next() и has_next()).
2. **ConcreteIterator (Конкретный итератор)**
   Реализует методы интерфейса итератора. Хранит текущее положение в обходе.
3. **Aggregate (Коллекция)**
   Определяет интерфейс для создания итератора.
4. **ConcreteAggregate (Конкретная коллекция)**
   Реализует метод создания итератора и предоставляет данные для итерации.
5. **Client (Клиент)**
   Использует итератор для обхода коллекции.

V. Посредник)

Посредник (Mediator) — это поведенческий паттерн проектирования, который позволяет уменьшить связанность множества объектов между собой, централизуя их взаимодействие через один объект — посредника.

Основная идея

Вместо того чтобы объекты напрямую взаимодействовали друг с другом, они обмениваются данными через посредника. Посредник управляет взаимодействием, что упрощает поддержку и расширение системы.

Структура

1. **Mediator (Посредник)**
   Определяет интерфейс для взаимодействия с коллегами.
2. **ConcreteMediator (Конкретный посредник)**
   Реализует интерфейс посредника, координируя взаимодействие между коллегами.
3. **Colleague (Коллега)**
   Базовый класс или интерфейс для объектов, взаимодействующих через посредника.
4. **ConcreteColleague (Конкретный коллега)**
   Конкретные классы, которые взаимодействуют с другими объектами через посредника.

VI. Хранитель

Хранитель (Memento) — это поведенческий паттерн проектирования, который позволяет сохранять и восстанавливать состояние объекта, не раскрывая его внутренней структуры. Этот паттерн используется для реализации функции отмены (Undo) или восстановления (Redo).

Основная идея

Хранитель предоставляет механизм, с помощью которого объект может сохранять своё состояние, чтобы в будущем восстановить его. Это достигается благодаря созданию специального объекта, который хранит внутреннее состояние исходного объекта.

Структура

1. **Originator (Создатель)**
   - Объект, состояние которого нужно сохранить.
   - Создаёт объект-хранитель для сохранения своего состояния.
   - Может восстановить своё состояние из объекта-хранителя.
2. **Memento (Хранитель)**
   - Объект, который хранит состояние "Создателя".
   - Обычно это простой объект с данными, которые нужно сохранить.
3. **Caretaker (Опекун)**
   - Управляет хранителями и обеспечивает сохранение/восстановление состояния "Создателя".
   - Не изменяет и не анализирует содержимое объекта-хранителя.

VII. Наблюдатель

Наблюдатель (Observer) — это поведенческий паттерн проектирования, который позволяет объектам подписываться на события другого объекта и реагировать на их изменения. Этот паттерн устанавливает отношение "один ко многим": при изменении состояния одного объекта все его наблюдатели уведомляются об этом.

Основная идея

Паттерн **Наблюдатель** используется, когда одно событие или изменение состояния объекта должно автоматически вызывать обновления в других объектах. Это позволяет избавиться от жёсткой связанности между объектами.

Структура

1. **Subject (Субъект)**
   - Отвечает за управление подписчиками (наблюдателями).
   - Имеет методы для добавления, удаления и уведомления наблюдателей.
2. **Observer (Наблюдатель)**

- Интерфейс для подписчиков, который определяет метод обновления.
- Реализует реакцию на изменения в субъекте.

3. **ConcreteSubject (Конкретный субъект)**
   - Содержит данные и состояние, которые интересуют наблюдателей.
   - Уведомляет наблюдателей при изменении своего состояния.
4. **ConcreteObserver (Конкретный наблюдатель)**
   - Реализует реакцию на уведомления от субъекта.

VIII. Состояние

Состояние (State) — это поведенческий паттерн проектирования, который позволяет объекту изменять свое поведение в зависимости от его внутреннего состояния. Внешне это выглядит так, будто объект изменяет свой класс.

Основная идея

Паттерн **Состояние** помогает избежать большого количества условных конструкций, таких как if или switch, которые проверяют текущее состояние объекта. Вместо этого для каждого состояния создается отдельный класс, а сам объект делегирует выполнение поведения этим классам.

Структура

1. **State (Состояние)**
   - Определяет интерфейс для всех конкретных состояний.
   - Каждый метод отражает одно из возможных действий объекта.
2. **ConcreteState (Конкретное состояние)**
   - Реализует поведение, характерное для конкретного состояния.
3. **Context (Контекст)**
   - Хранит текущее состояние объекта.
   - Должен предоставлять интерфейс клиентам, делегируя выполнение действий текущему объекту состояния.

IX. Состояние

Состояние (State) — это поведенческий паттерн проектирования, который позволяет объекту изменять свое поведение при изменении его внутреннего состояния. Этот паттерн структурирует логику объекта в отдельные классы, каждый из которых соответствует одному состоянию.

Основная идея

Когда объект может находиться в нескольких состояниях, и его поведение изменяется в зависимости от текущего состояния, вместо множества

условных операторов (if, switch), логика поведения делегируется объектам-состояниям. Каждый из них знает, как обработать запрос в своем состоянии.

Структура

1. **State (Состояние)**
   - Интерфейс или базовый класс для всех конкретных состояний.
   - Определяет общие методы, которые должны реализовать состояния.
2. **ConcreteState (Конкретное состояние)**
   - Реализует поведение, связанное с определённым состоянием.
3. **Context (Контекст)**
   - Хранит текущее состояние объекта.
   - Делегирует выполнение задач текущему объекту состояния.
   - Может изменять свое текущее состояние.

X. Стратегия

Стратегия (Strategy) — это поведенческий паттерн проектирования, который позволяет определять семейство алгоритмов, инкапсулировать их и делать их взаимозаменяемыми. Этот паттерн позволяет изменять поведение объекта в зависимости от алгоритма, который используется для выполнения операции, не изменяя сам объект.

Основная идея

Паттерн **Стратегия** позволяет выбирать алгоритм на основе контекста. Вместо того чтобы иметь множество условных операторов, которые определяют поведение, мы можем использовать разные стратегии (алгоритмы), инкапсулированные в отдельных классах. Стратегии можно менять в процессе выполнения программы.

Структура

1. **Strategy (Стратегия)**
   - Интерфейс или абстрактный класс, который определяет общий метод для всех конкретных стратегий.
2. **ConcreteStrategy (Конкретная стратегия)**
   - Реализация конкретного алгоритма, который будет использоваться объектом.
3. **Context (Контекст)**
   - Использует объект стратегии для делегирования выполнения алгоритма. Контекст не зависит от конкретных алгоритмов и позволяет их динамическую замену.

XI. Шаблонный метод

Шаблонный метод (Template Method) — это поведенческий паттерн проектирования, который определяет скелет алгоритма в методе, оставляя

реализацию некоторых шагов на усмотрение подклассов. Таким образом, паттерн позволяет подклассам изменять определённые шаги алгоритма, не изменяя его структуру.

Основная идея

Паттерн **Шаблонный метод** позволяет создать общий алгоритм с некоторыми фиксированными шагами, которые не изменяются, и оставить возможность для подклассов переопределить отдельные шаги этого алгоритма. Это дает возможность сохранить консистентность алгоритма, но при этом гибко изменять его детали в подклассах.

Структура

1. **AbstractClass (Абстрактный класс)**
    - Определяет шаблон метода (обычно публичного), который содержит основные шаги алгоритма.
    - Может содержать конкретные методы, которые не подлежат изменению, а также абстрактные методы, которые должны быть реализованы в подклассах.
2. **ConcreteClass (Конкретный класс)**
    - Реализует шаги, которые могут изменяться в алгоритме, используя абстрактные методы.

XII. Посетитель

Посетитель (Visitor) — это поведенческий паттерн проектирования, который позволяет добавлять новые операции к объектам, не изменяя их классы. Этот паттерн используется, когда необходимо выполнять операции с объектами, принадлежащими к различным классам, но при этом необходимо избежать изменения этих классов.

Основная идея

Паттерн **Посетитель** позволяет определять новые операции для элементов структуры объектов, не изменяя сами объекты. Вместо того чтобы добавлять методы напрямую в классы, создается отдельный объект — "посетитель", который выполняет нужные действия.

Этот паттерн полезен, когда:

1. Нужно выполнить множество различных операций над объектами, но сами объекты не должны изменяться.
2. Есть необходимость часто добавлять новые операции, которые должны применяться ко всем объектам без изменения их исходного кода.

Структура

1. **Visitor (Посетитель)**
    - Интерфейс для всех конкретных посетителей, который определяет метод для каждого типа элемента (конкретного объекта).

2. **ConcreteVisitor (Конкретный посетитель)**
    - Реализует операции для каждого типа элемента, делая операции специфичными для конкретных объектов.
3. **Element (Элемент)**
    - Интерфейс для элементов, которые могут быть посещены. Каждый элемент должен реализовать метод accept(), который позволяет посетителю выполнить операцию.
4. **ConcreteElement (Конкретный элемент)**
    - Реализация элемента, который может быть посещен. Каждый элемент реализует метод accept(), который передает себя в метод посещения посетителя.
5. **ObjectStructure (Структура объектов)**
    - Хранит объекты, которые могут быть посещены.

Порождающие паттерны являются мощным инструментом для управления процессом создания объектов, что повышает гибкость и уменьшает жесткую зависимость от конкретных классов.

Например, использование паттерна **Фабричный метод** позволяет скрыть логику создания объектов за интерфейсами, предоставляя возможность легко изменять тип создаваемых объектов, не затрагивая остальную систему. Это позволяет добавлять новые виды объектов без изменения существующего кода, что является существенным плюсом в крупных и изменяющихся проектах.

Однако, внедрение таких паттернов может увеличить количество классов, что делает систему более сложной для понимания и сопровождения. Также возможна избыточная абстракция, когда простой процесс создания объекта превращается в излишне сложную структуру, что делает код трудным для поддержки.

Структурные паттерны помогают эффективно управлять взаимодействием между большими и сложными объектами. Например, с помощью паттерна **Адаптер** можно интегрировать компоненты, использующие различные интерфейсы, что снижает связанность системы и упрощает взаимодействие между объектами.

Однако, когда структура системы не требует такой гибкости, применение структурных паттернов может привести к излишней сложности, что может затруднить понимание кода. Например, использование **Декоратора** может вызвать лишнюю нагрузку на систему, когда добавление функциональности через множество небольших классов не оправдывает затрат на их создание и поддержку.

Поведенческие паттерны обеспечивают гибкость в изменении поведения объектов. Паттерн **Стратегия** позволяет изменять алгоритм в зависимости от условий, не затрагивая сам объект, что делает систему более адаптивной и

легко расширяемой. Это особенно полезно в динамичных приложениях, где поведение объектов часто меняется.

Однако, использование этих паттернов может привести к чрезмерному усложнению системы, когда множество объектов взаимодействуют через сложные цепочки, как, например, в **Цепочке обязанностей**. Это усложняет отладку и диагностику системы, так как необходимо отслеживать и контролировать каждый этап цепочки.

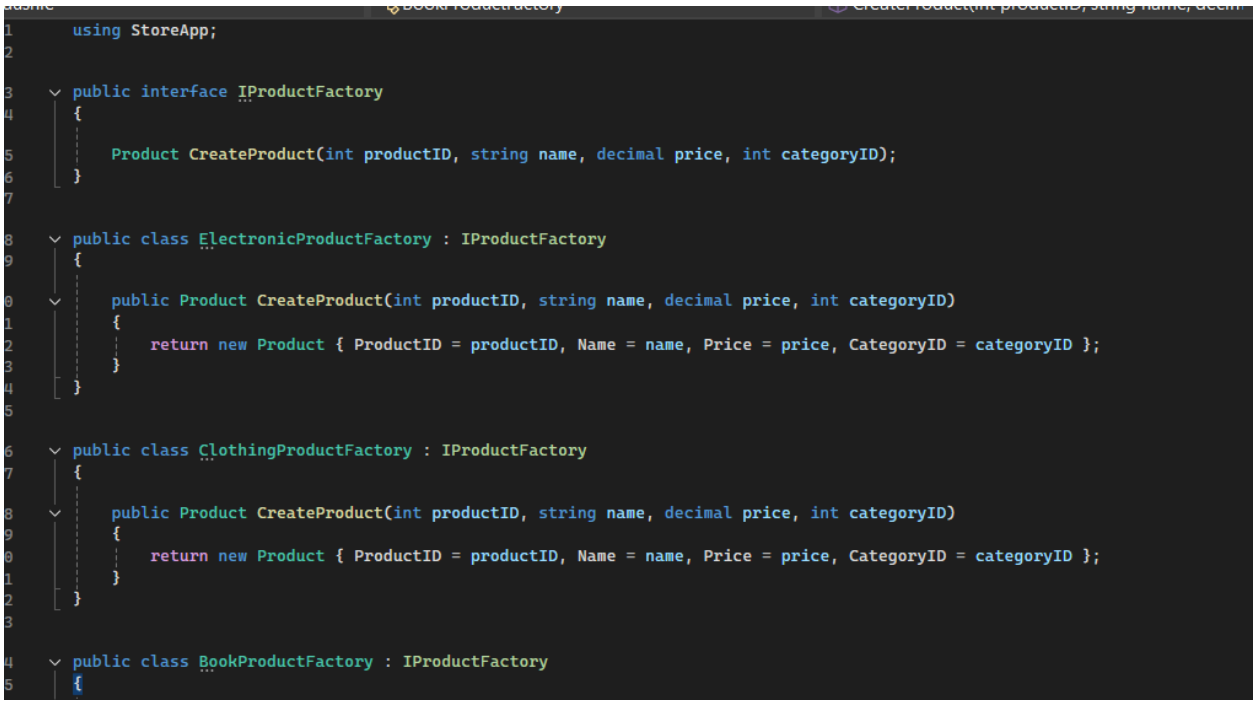
Паттерн **Посетитель**, с другой стороны, при добавлении нового типа объекта требует изменений в коде всех существующих посетителей, что затрудняет масштабирование и добавление новых возможностей в систему. В общем, хотя паттерны проектирования могут значительно упростить решение сложных задач, важно учитывать их влияние на общую структуру системы и необходимость их использования в контексте конкретной задачи.

Задание 3. Реализовать 6 паттернов проектирования не менее 3 типов на языке программирования C#.

Для выполнения задание разобьем применения паттерном на три отдельных проекта. Будем использовать два пораждающих паттерна в первом проекте, два структурных во втором и два поведенческих в третьем. Идея будет единная: Написать прототип магазина на WPF, используя базу данных SQLite

Abstract Factory — это порождающий паттерн проектирования, который предоставляет интерфейс для создания семейств связанных или зависимых объектов без указания их конкретных классов. Этот паттерн позволяет инкапсулировать логику создания объектов и скрыть её от клиентского кода. Зачем использовать?

- Упрощение добавления новых типов продуктов: Если вам нужно добавить новый тип продукта (например, FoodProduct), вам не нужно менять клиентский код. Вы просто создаете новую фабрику, реализующую интерфейс IProductFactory.
- Создание связанных объектов: Abstract Factory позволяет создавать объекты, которые логически связаны между собой (например, продукты одной категории).
- Гибкость: Клиентский код работает с интерфейсом фабрики, а не с конкретными реализациями, что делает систему более гибкой и расширяемой.



```
using StoreApp;


public interface IProductFactory
{

    Product CreateProduct(int productID, string name, decimal price, int categoryID);
}


public class ElectronicProductFactory : IProductFactory
{

    public Product CreateProduct(int productID, string name, decimal price, int categoryID)
    {
        return new Product { ProductID = productID, Name = name, Price = price, CategoryID = categoryID };
    }
}


public class ClothingProductFactory : IProductFactory
{

    public Product CreateProduct(int productID, string name, decimal price, int categoryID)
    {
        return new Product { ProductID = productID, Name = name, Price = price, CategoryID = categoryID };
    }
}


public class BookProductFactory : IProductFactory
{
```

Создадим интерфейс IProductFactory, который определяет метод CreateProduct. Затем реализуем этот интерфейс в нескольких классах: ElectronicProductFactory, ClothingProductFactory и BookProductFactory. Каждая фабрика создает объекты типа Product, но с разными характеристиками (электроника, одежда, книги).

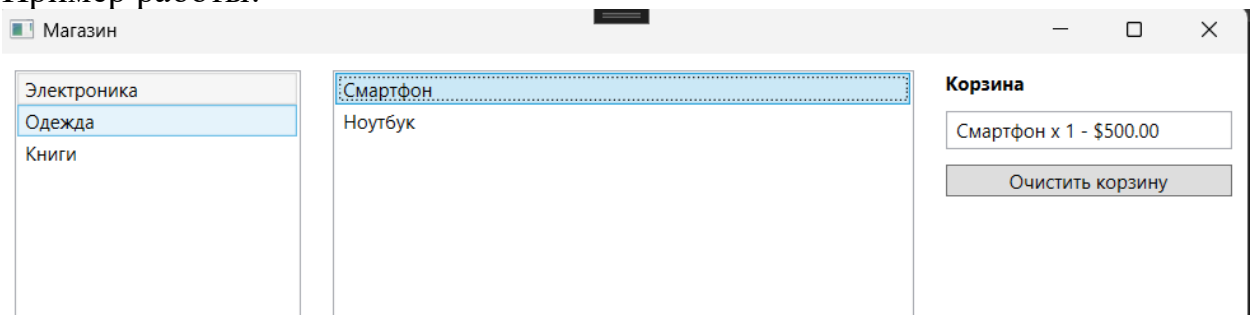
Singleton — это порождающий паттерн проектирования, который гарантирует, что класс имеет только один экземпляр, и предоставляет глобальную точку доступа к этому экземпляру.
Зачем использовать?

- Управление единственным экземпляром: Singleton гарантирует, что в приложении существует только один экземпляр класса. Это полезно, например, для управления общими ресурсами, такими как корзина покупок, логгер или конфигурация приложения.
- Глобальный доступ: Singleton предоставляет глобальную точку доступа к этому экземпляру, что упрощает использование объекта в разных частях приложения.
- Экономия ресурсов: Использование Singleton позволяет избежать создания множества экземпляров одного и того же объекта, что экономит память и другие ресурсы.

```
namespace StoreApp
{

    public class ShoppingCart
    {
        private static ShoppingCart _instance;
        private List<CartItem> _cartItems;


        private ShoppingCart()
        {
            _cartItems = new List<CartItem>();
        }


        public static ShoppingCart Instance
        {
            get
            {
                if (_instance == null)
                {
                    _instance = new ShoppingCart();
                }
                return _instance;
            }
        }
```

Создадим класс ShoppingCart, который реализует паттерн Singleton. Конструктор класса ShoppingCart объявлен как private, что предотвращает создание экземпляров класса извне. Единственный экземпляр класса создается через свойство Instance. Методы AddToCart и ClearCart позволяют управлять содержимым корзины.

Пример работы:



Перейдем к структурным паттернам:

1. Адаптер (Adapter)

- **SQLiteAdapter** выступает в роли адаптера. Он адаптирует работу с базой данных SQLite к интерфейсу, который ожидает **ProductService**. Это позволяет **ProductService** работать с базой данных, не зная о конкретной реализации работы с SQLite.
- **SQLiteAdapter** реализует методы для работы с базой данных (например, GetProducts), которые затем используются **ProductService**.

```
using System.Collections.Generic;
using System.Data.SQLite;


public class SQLiteAdapter
{
    private readonly string _connectionString;


    public SQLiteAdapter(string connectionString)
    {
        _connectionString = connectionString;
    }


    public List<Product> GetProducts()
    {
        var products = new List<Product>();
        using (var connection = new SQLiteConnection(_connectionString))
        {
            connection.Open();
            var command = new SQLiteCommand("SELECT * FROM Products", connection);
            using (var reader = command.ExecuteReader())
            {
                while (reader.Read())
                {
                    products.Add(new Product
                    {
                        Id = reader.GetInt32(0),
                        Name = reader.GetString(1),
                        Price = reader.GetDecimal(2),
                        Description = reader.GetString(3)
```

2. Мост (Bridge)

- Мост используется для разделения абстракции (ProductService) и её реализации (SQLiteAdapter). Это позволяет изменять их независимо друг от друга.
- В данном случае ProductService представляет собой абстракцию, которая не зависит от конкретной реализации работы с базой данных. Вместо этого она использует SQLiteAdapter как мост для доступа к данным.

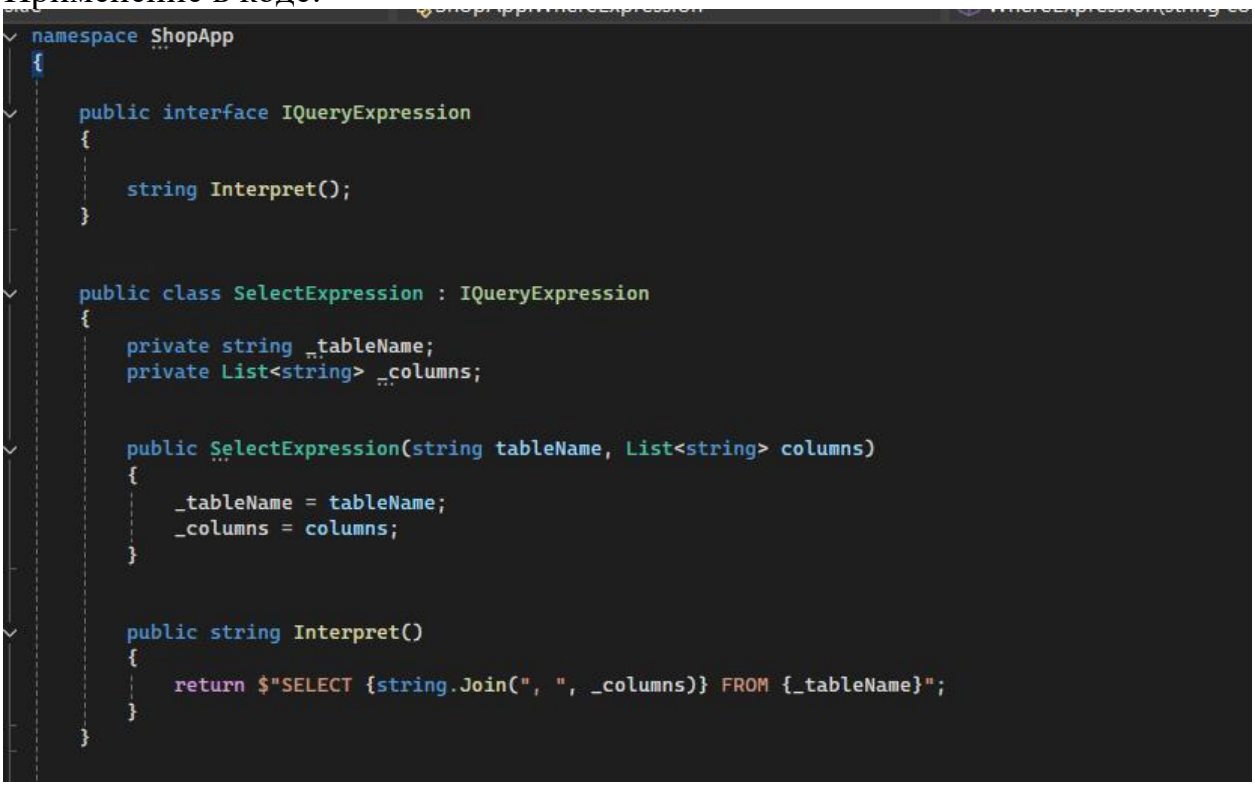
```
public class ProductService
{
    private readonly SQLiteAdapter _adapter;


    public ProductService(SQLiteAdapter adapter)
    {
        _adapter = adapter;
    }


    public List<Product> GetAllProducts()
    {
        return _adapter.GetProducts();
    }
}
```

Теперь перейдем к поведенческим паттернам:

1. Интерпретатор (Interpreter)

Паттерн **Интерпретатор** используется для определения грамматики языка и интерпретации выражений в этом языке. Он позволяет создавать сложные структуры (например, SQL-запросы) путем комбинирования простых выражений.

Применение в коде:



```
namespace ShopApp
{

    public interface IQueryExpression
    {

        string Interpret();
    }


    public class SelectExpression : IQueryExpression
    {
        private string _tableName;
        private List<string> _columns;


        public SelectExpression(string tableName, List<string> columns)
        {
            _tableName = tableName;
            _columns = columns;
        }


        public string Interpret()
        {
            return $"SELECT {string.Join(", ", _columns)} FROM {_tableName}";
        }
    }
```

В коде паттерн **Интерпретатор** используется для создания SQL-запросов. Это достигается с помощью интерфейса IQueryExpression и его реализаций: SelectExpression и WhereExpression.

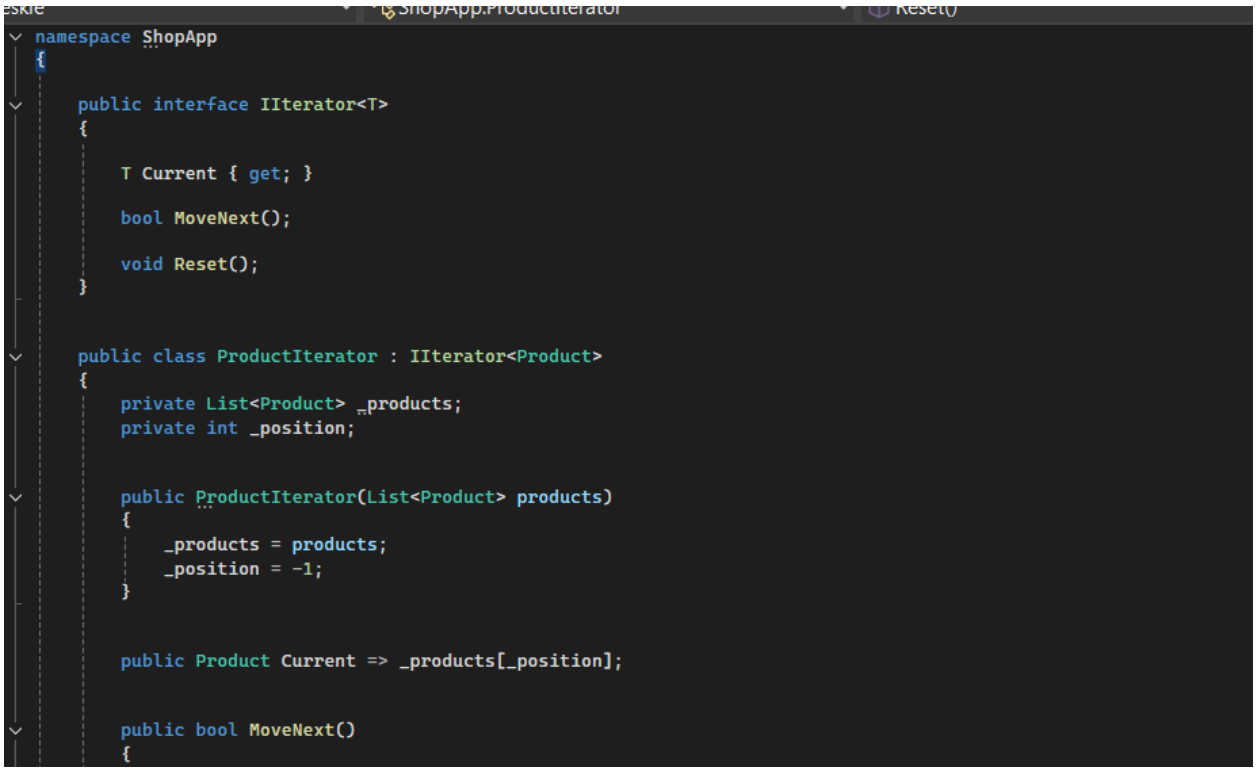
IQueryExpression - Этот интерфейс определяет метод Interpret(), который должен возвращать строку (в данном случае SQL-запрос).

SelectExpression - Этот класс представляет собой выражение для создания SQL-запроса SELECT. Он принимает имя таблицы и список столбцов, которые нужно выбрать, и возвращает соответствующую строку SQL.

WhereExpression - Этот класс представляет собой выражение для добавления условия в SQL-запрос. Он принимает строку условия и возвращает соответствующую часть SQL-запроса.

2. Итератор (Iterator)

Паттерн Итератор предоставляет способ последовательного доступа к элементам коллекции без раскрытия её внутреннего представления. Это позволяет работать с коллекцией независимо от её структуры.
Применение в коде:



```
namespace ShopApp
{

    public interface IIterator<T>
    {

        T Current { get; }

        bool MoveNext();

        void Reset();
    }


    public class ProductIterator : IIterator<Product>
    {
        private List<Product> _products;
        private int _position;


        public ProductIterator(List<Product> products)
        {
            _products = products;
            _position = -1;
        }


        public Product Current => _products[_position];


        public bool MoveNext()
        {
```

В коде паттерн Итератор используется для последовательного доступа к элементам списка продуктов. Это достигается с помощью интерфейса IIterator<T> и его реализации ProductIterator.

Интерфейс IIterator<T>
Этот интерфейс определяет методы для работы с коллекцией:

- Current: Возвращает текущий элемент.
- MoveNext(): Перемещает указатель на следующий элемент.
- Reset(): Сбрасывает указатель на начальное положение.

ProductIterator - Этот класс реализует интерфейс IIterator<Product>. Он принимает список продуктов и предоставляет методы для последовательного доступа к его элементам.

Ответы на вопросы:

1. Как во время работы программы получить путь к каталогу, откуда запущена программа?

Для получения пути к каталогу, откуда запущена программа, можно использовать статический метод System.IO.Directory.GetCurrentDirectory(). Этот метод возвращает текущий рабочий каталог, в котором находится исполняемый файл программы.

2. Как получить актуальный разделитель текста Windows?

В Windows разделитель текста (или разделитель строк) обычно представляет собой комбинацию символов \r\n (возврат каретки и перевод строки). Для получения этого разделителя в C# можно использовать статическое свойство Environment.NewLine.

3. Какие методы и классы для работы с файлами вы знаете?

В C# для работы с файлами используется пространство имен System.IO. Основные классы и методы для работы с файлами:

Классы:

- **File**: Статический класс для работы с файлами (чтение, запись, копирование, удаление и т.д.).
- **FileInfo**: Представляет файл и предоставляет методы для работы с ним.
- **Directory**: Статический класс для работы с каталогами (создание, удаление, получение списка файлов и т.д.).
- **DirectoryInfo**: Представляет каталог и предоставляет методы для работы с ним.
- **Path**: Статический класс для работы с путями к файлам и каталогам.
- **StreamReader и StreamWriter**: Используются для чтения и записи текстовых файлов.
- **BinaryReader и BinaryWriter**: Используются для работы с двоичными файлами.

Методы:

- **File.ReadAllText(string path)**: Читает весь текст из файла.
- **File.WriteAllText(string path, string content)**: Записывает текст в файл.
- **File.Copy(string source, string destination)**: Копирует файл.
- **File.Delete(string path)**: Удаляет файл.
- **Directory.CreateDirectory(string path)**: Создает каталог.
- **Directory.GetFiles(string path)**: Возвращает список файлов в каталоге.
- **Path.Combine(string path1, string path2)**: Объединяет пути.
- **Path.GetFileName(string path)**: Возвращает имя файла из пути.

4. Основные типы паттернов проектирования.

1) Пораждающие паттерны
2) Структурные паттерны
3) Поведенческие паттерны

5. На каких абстрактных типах строятся поведенческие паттерны?

Поведенческие паттерны строятся на следующих абстрактных типах:

1. Интерфейсы (Interfaces): Интерфейсы определяют контракт, который должны реализовать классы.
2. Абстрактные классы (Abstract Classes): Абстрактные классы предоставляют базовую реализацию, которую могут расширять производные классы.
3. Делегаты (Delegates) и события (Events): Используются для реализации паттернов, связанных с обработкой событий.
4. Абстрактные методы (Abstract Methods): Используются для определения обязательных действий, которые должны быть реализованы в производных классах.

Представлены 2 проекта, реализованных в Visual Studio Community 2022 и доклад по паттернам программирования. Проекты представлены преподавателю в электронной форме, продемонстрирована их работоспособность, разъяснены детали программного кода.