Занятие 6. Поведенческие паттерны

Цель занятия

- Научиться использовать поведенческие паттерны

Описание

Поведенические паттерны предназначены для улучшения взаимодействия между классами.

В данном занятии мы рассмотрим следующие паттерны:

- Команда
- Шаблонный метод

Для рассмотрения данных паттернов рассмотрим следующую задачу.

Студенты ВШЭ, занимающиеся продажей автомобилей, заметили, что часто из-за спешки и невнимательности допускаются ошибки при учете автомобилей и покупателей в системе. Для решения данной проблемы они просят добавить сеансы работы с системой учета. Сеанс должен позволять следующее:

- Добавлять новые автомобили и покупателей;
- Отменить последнее действие;
- Просмотреть список внесенных изменений;
- Сохранить внесенные изменения.

При этом все производимые действия должны попадать в отчет о работе системы.

Должна быть возможность выгрузить отчет в следующих форматах:

- JSON;
- Markdown.

Структура проекта

Так как наш проект уже достаточно большой, для начала разобьем его на несколько каталогов. Существуют различные стратегии организации файлов в проектах:

- По типу - модели, сервисы, контроллеры и т.д.
- По функционалу - авторизация, регистрация, учет, отчеты и т.д.
- Смешанный подход.

Так как наш проект еще недостаточно большой, нам хватит разделения по функционалу.

Создадим следующие каталоги:

- `Engines` - двигатели;
- `Cars` - автомобили;
- `Customers` - покупатели;
- `Reports` - отчеты;
- `Sales` - продажи.

В `Engines` будут находиться следующие типы:

- `IEngine`;
- `HandEngine`;
- `PedalEngine`;
- `EmptyEngineParams`;
- `PedalEngineParams`.

В `Cars` будут находиться следующие типы:

- `Car`;
- `ICarProvider`;
- `ICarFactory`;
- `PedalCarFactory`;
- `HandCarFactory`;
- `CarService`;

В `Customers` будут находиться следующие типы:

- `Customer`;
- `ICustomerProvider`;
- `CustomerStorage`;

В `Reports` будут находиться следующие типы:

- `Report`;
- `ReportBuilder`.

В `Sales` будут находиться следующие типы:

- `HseCarService`.

После перенесения файлов корректируем пространства имен соответствующим образом.

После структуризации нашего проекта можно приступать к внесению остальных изменений.

Сеанс работы с системой учета

Для решения подобных задач, где необходимо вести история изменений с возможностью отмены, обычно используется паттерн "Команда".

Команда — это поведенческий паттерн, который превращает запросы в объекты, позволяя передавать их как аргументы при вызове методов, ставить запросы в очередь, логировать их, а также поддерживать отмену операций.

Решение

Начем с того, что создадим каталог `Accounting` для группировки типов, относящихся непосредственно к системе учета.

Далее определим интерфейс для представления команд в нашей системе. Для этого объявим интерфейс `IAccountingSessionCommand` и добавим в него метод для применения изменений.

```
public interface IAccountingSessionCommand
{
    void Apply();
}
```

Далее создадим класс `AccountingSession`, который будет использовать команду для применения изменений. Данный класс будет поддерживать следующие действия:

- Добавление команды в сеанс;
- Отмена последней команды;
- Просмотр списка несохраненных изменений;
- Сохранение внесенных изменений.

Для реализации данного класса нам потребуется следующие поля:

- Список непримененных команд;
- Список отменных команд.

Также нам потребуются методы:

- Добавление команды в сеанс;
- Отмена последней команды;
- Повтора отмененной команды;
- Просмотр списка несохраненных изменений;
- Сохранение внесенных изменений.

Начнем с определения класса `AccountingSession`.

```
public sealed class AccountingSession
{
    private readonly LinkedList<IAccountingSessionCommand> _unappliedCommands =
new();
    private readonly Stack<IAccountingSessionCommand> _undoneCommands = new();

    private readonly ReportBuilder _reportBuilder;

    public AccountingSession(ReportBuilder reportBuilder)
    {
        _reportBuilder = reportBuilder;
    }
}
```

Далее определим методы класса. Метод добавления команды в сеанс должен добавлять команду в список неприменённых команд, а также очищать список отменённых команд, так как они становятся ненужными.

```
public void AddCommand(IAccountingSessionCommand command)
{
    _unappliedCommands.AddLast(command);
    _undoneCommands.Clear();
}
```

Далее определим методы для отмены последней команды и повтора отменённой команды.

```
public void UndoLastCommand()
{
    if (_unappliedCommands.Count > 0)
    {
        var command = _unappliedCommands.Last.Value;
        _unappliedCommands.RemoveLast();
        _undoneCommands.Push(command);
    }
}

public void RedoUndoneCommand()
{
    if (_undoneCommands.Count > 0)
    {
        var command = _undoneCommands.Pop();
        _unappliedCommands.AddLast(command);
    }
}
```

Далее определим свойство для просмотра списка несохраненных изменений.

```
public IReadOnlyCollection<IAccountingSessionCommand> UnappliedCommands =>
_unappliedCommands;
```

Далее определим метод для сохранения внесенных изменений.

```
public void SaveChanges()
{
    foreach (var command in _unappliedCommands)
    {
        command.Apply();
```

```
            var operation = command.ToString();

            if (operation is not null)
            {
                _reportBuilder.AddOperation(operation);
            }
        }
    }
```

Мы определили класс, представляющий сеанс работы с системой учета. Далее реализуем необходимые команды.

Команда добавления автомобиля

Команда добавления автомобиля должна добавлять автомобиль в систему учета. Так как у нас есть два типа автомобилей, предлагается создать два класса-команды.

Начнем с класса `AddPedalCarCommand`.

```
public sealed class AddPedalCarCommand : IAccountingSessionCommand
{
    private readonly PedalCarFactory _pedalCarFactory;
    private readonly CarService _carService;
    private readonly int _size;

    public AddPedalCarCommand(PedalCarFactory pedalCarFactory, CarService
carService, int size)
    {
        _pedalCarFactory = pedalCarFactory;
        _carService = carService;
        _size = size;
    }

    public void Apply()
    {
        _carService.AddCar(_pedalCarFactory, new PedalEngineParams(_size));
    }

    public override string ToString()
    {
        return $"Добавлен педальный автомобиль с размером педалей {_size}.";
    }
}
```

Далее определим класс `AddHandCarCommand`.

```
public sealed class AddHandCarCommand : IAccountingSessionCommand
{
    private readonly HandCarFactory _handCarFactory;
```

```
    private readonly CarService _carService;

    public AddHandCarCommand(HandCarFactory handCarFactory, CarService carService)
    {
        _handCarFactory = handCarFactory;
        _carService = carService;
    }

    public void Apply()
    {
        _carService.AddCar(_handCarFactory, EmptyEngineParams.DEFAULT);
    }

    public override string ToString()
    {
        return "Добавлен автомобиль с ручным приводом.";
    }
}
```

Далее определим класс `AddCustomerCommand`.

```
public sealed class AddCustomerCommand : IAccountingSessionCommand
{
    private readonly CustomersStorage _customersStorage;
    private readonly string _name;
    private readonly int _legPower;
    private readonly int _handPower;

    public AddCustomerCommand(CustomersStorage customersStorage, string name, int
legPower, int handPower)
    {
        _customersStorage = customersStorage;
        _name = name;
        _legPower = legPower;
        _handPower = handPower;
    }

    public void Apply()
    {
        _customersStorage.AddCustomer(new Customer(_name, _legPower, _handPower));
    }

    public override string ToString() => $"Добавлен покупатель {_name}. Сила ног:
{_legPower}. Сила рук: {_handPower}.";
}
```

Модифицируем наш класс `Program` для использования нашей системы учета.

Для начала создадим сеанс работы с системой учета.

```
var report = new ReportBuilder();
var accountingSession = new AccountingSession(report);
```

Модифицируем код добавления покупателей.

```
accountingSession.AddCommand(new AddCustomerCommand(
    customersStorage: customers,
    name: "Ваня",
    legPower: 6,
    handPower: 4
));

accountingSession.AddCommand(new AddCustomerCommand(
    customersStorage: customers,
    name: "Света",
    legPower: 4,
    handPower: 6
));

accountingSession.AddCommand(new AddCustomerCommand(
    customersStorage: customers,
    name: "Сергей",
    legPower: 6,
    handPower: 6
));

accountingSession.AddCommand(new AddCustomerCommand(
    customersStorage: customers,
    name: "Алексей",
    legPower: 4,
    handPower: 4
));
```

Для проверки отмены последней команды добавим лишнего покупателя.

```
// Добавим лишнего покупателя
accountingSession.AddCommand(new AddCustomerCommand(
    customersStorage: customers,
    name: "Маша",
    legPower: 4,
    handPower: 4
));
```

Отменяем последнюю команду.

```
accountingSession.UndoLastCommand();
```

Модифицируем код добавления автомобилей.

```
accountingSession.AddCommand(new AddPedalCarCommand(pedalCarFactory, cars, 2)); //
добавляем педальный автомобиль
accountingSession.AddCommand(new AddPedalCarCommand(pedalCarFactory, cars, 3)); //
добавляем педальный автомобиль
accountingSession.AddCommand(new AddHandCarCommand(handCarFactory, cars)); //
добавляем автомобиль с ручным приводом
accountingSession.AddCommand(new AddHandCarCommand(handCarFactory, cars)); //
добавляем автомобиль с ручным приводом
```

Добавим лишний автомобиль.

```
accountingSession.AddCommand(new AddPedalCarCommand(pedalCarFactory, cars, 4)); //
добавляем педальный автомобиль
```

Отменяем последнюю команду.

```
accountingSession.UndoLastCommand();
```

Сохраняем внесенные изменения.

```
accountingSession.SaveChanges();
```

Немного изменим содержимое отчета.

```
report
    .AddOperation("Вывод списка покупателей")
    .AddCustomers(customers.GetCustomers());
```

Проверим, что результат выполнения программы не изменился, но в отчете появились новые действия, соответствующие нашим командам. Как можем видеть, лишние покупатели не были добавлены в систему - значит, наши команды работают корректно.

Несколько форматов выгрузки отчета

В нашем приложении мы можем использовать несколько форматов выгрузки отчета.

- JSON;
- Markdown.

Для реализации данного функционала можно использовать паттерн "Шаблонный метод".

Шаблонный метод — это поведенческий паттерн, который определяет скелет алгоритма, перекладывая ответственность за некоторые его шаги на подклассы. Паттерн позволяет поддерживать алгоритм неизменным, изменяя только некоторые конкретные шаги.

Решение

Создадим абстрактный класс `ReportExporter`.

```
public abstract class ReportExporter
{
    public abstract void Export(Report report, TextWriter writer);
}
```

В качестве параметра метода `Export` используется `TextWriter`, который позволяет записывать данные в любой поток вывода. Он реализует для нас паттерн "Стратегия".

Паттерн "Стратегия" позволяет выбрать алгоритм, который будет использоваться в зависимости от контекста.

Далее определим класс, реализующий экспорт в формате JSON.

```
public sealed class JsonReportExporter : ReportExporter
{
    public override void Export(Report report, TextWriter writer)
    {
        writer.Write(JsonSerializer.Serialize(report));
    }
}
```

Здесь мы использовали метод `Serialize` класса `JsonSerializer` из пространства имен `System.Text.Json`. Это встроенный в .NET метод сериализации в JSON.

Далее определим класс, реализующий экспорт в формате Markdown.

```
public sealed class MarkdownReportExporter : ReportExporter
{
    public override void Export(Report report, TextWriter writer)
    {
        writer.WriteLine($"# {report.Title}");
        writer.WriteLine();
        writer.WriteLine(report.Content);
    }
}
```

Теперь определим перечисление `ReportFormat` для обозначения формата экспорта.

```
public enum ReportFormat
{
    Json,
    Markdown
}
```

Далее определим класс `ReportExporterFactory` для создания экспортера в зависимости от формата.

```
public sealed class ReportExporterFactory
{
    public ReportExporter Create(ReportFormat format)
    {
        return format switch
        {
            ReportFormat.Json => new JsonReportExporter(),
            ReportFormat.Markdown => new MarkdownReportExporter(),
            _ => throw new ArgumentException($"Неизвестный формат экспорта:
{format}")
        };
    }
}
```

Теперь зарегистрируем нашу фабрику в DI-контейнере. Для этого добавляем строчку в методе `CreateServiceProvider` в классе `CompositionRoot`.

```
services.AddSingleton<ReportExporterFactory>();
```

Теперь мы можем использовать нашу фабрику для создания экспортеров. Для этого после окончания работы с системой создадим экспортер в Markdown и выведем отчет на консоль в данном формате.

```
var markdownExporter = services
    .GetRequiredService<ReportExporterFactory>()
    .Create(ReportFormat.Markdown);

markdownExporter.Export(report.Build(), Console.Out);
```

Теперь запишем в файл `report.json` наш отчет в формате JSON.

```
var jsonExporter = services
    .GetRequiredService<ReportExporterFactory>()
    .Create(ReportFormat.Json);

using var reportFile = new StreamWriter("report.json");
```

```
jsonExporter.Export(report.Build(), reportFile);
```

Запустим программу и проверим, что отчеты были сохранены корректно.

Как мы можем видеть, использование паттерна "Шаблонный метод" позволило нам легко добавлять новые форматы экспорта, не меняя код работы экспортером. Также использование паттерна "Стратегия" в виде встроенного в .NET класса `TextWriter` позволило нам легко переключать способ сохранения отчета, не меняя код самого экспортера.