Занятие 2. Принципы SOLID

Теоретическая часть

Принципы SOLID - это набор принципов, которые помогают создавать более качественный и поддерживаемый код.

Сами принципы следующие:

- Single Responsibility Principle (Принцип единственной ответственности) - каждый класс должен иметь только одну ответственность
- Open/Closed Principle (Принцип открытости/закрытости) - классы должны быть открыты для расширения, но закрыты для модификации
- Liskov Substitution Principle (Принцип подстановки Барбары Лисков) - объекты должны быть заменяемыми на объекты своих подтипов без изменения корректности программы
- Interface Segregation Principle (Принцип разделения интерфейсов) - клиенты не должны зависеть от интерфейсов, которые они не используют
- Dependency Inversion Principle (Принцип инверсии зависимостей) - модули верхнего уровня не должны зависеть от модулей нижнего уровня, оба должны зависеть от абстракций

На практике соблюдение принципов SOLID не всегда возможно, но всегда стоит стремиться к их соблюдению.

Практическая часть

Для рассмотрения принципов SOLID рассмотрим задачу написания сервиса для работы с отчетами. Что он должен уметь:

- Генерировать отчеты
- Сохранять отчеты
- Отправлять отчеты по email

Вариант без соблюдения принципов SOLID

Создание проекта

Для создания проекта выполним следующие команды:

```
dotnet new solution -n 02-solid
dotnet sln migrate
rm 02-solid.sln
dotnet new console -n AppWithoutSolid -o ./AppWithoutSolid
dotnet sln add ./AppWithoutSolid/AppWithoutSolid.csproj
```

Рассмотрим каждую команду:

- `dotnet new solution -n 02-solid` - создаем решение с именем `02-solid`

- `dotnet sln migrate` - мигрируем решение с формата SLN на формат SLNX, добавленный в последних версиях .NET
- `rm 02-solid.sln` - удаляем старый файл решения
- `dotnet new console -n AppWithoutSolid -o ./AppWithoutSolid` - создаем консольное приложение с именем `AppWithoutSolid` в папке `./AppWithoutSolid`
- `dotnet sln add ./AppWithoutSolid/AppWithoutSolid.csproj` - добавляем созданный проект в решение

Создание класса `ReportService`

Создадим класс `ReportService` в корне проекта:

```
using System.Text;

namespace AppWithoutSolid;

public class ReportService
{
    public string GenerateReport()
    {
        var builder = new StringBuilder();

        builder
            .AppendLine("Отчет")
            .AppendLine($"Дата: {DateTime.Now.ToString("dd.MM.yyyy")}")
            .AppendLine($"Время: {DateTime.Now.ToString("HH:mm:ss")}")
            .AppendLine("--------------------------------")
            .AppendLine($"Продано автомобилей: {100} шт.")
            .AppendLine($"Продано мотоциклов: {50} шт.")
            .AppendLine("--------------------------------");

        return builder.ToString();
    }

    public void SaveReport(string report, string fileName)
    {
        File.WriteAllText(fileName, report);
    }

    public void SendReport(string report, string email)
    {
        Console.WriteLine($"Отправка отчета на email: {email}");
    }
}
```

Добавление использования класса `ReportService` в `Program.cs`

Добавим использование класса `ReportService` в `Program.cs`:

```
using AppWithoutSolid;

var reportService = new ReportService();
var report = reportService.GenerateReport();
reportService.SaveReport(report, "report.txt");
reportService.SendReport(report, "example@example.com");
```

Запуск приложения

Запустим приложение:

```
dotnet run --project ./AppWithoutSolid/AppWithoutSolid.csproj
```

В результате мы получим следующий вывод:

```
Отправка отчета на email: example@example.com
```

При этом в текущем каталоге будет создан файл `report.txt` с содержимым:

```
Отчет
Дата: 21.09.2025
Время: 14:52:59
--------------------------------
Продано автомобилей: 100 шт.
Продано мотоциклов: 50 шт.
--------------------------------
```

Анализ кода

Рассмотрим подробно код класса `ReportService`. Положительным моментом в данном коде является то, что код простой для понимания.

Однако, допустим, что нам стало недостаточно сохранять только текстовые файлы, но и нужно сохранять отчеты, например, в PDF, либо в файл JSON для последующей отправки. В этом случае у нас есть два варианта:

- Меняем код метода `SaveReport` - в этом случае весь код, который использует данный метод, также нужно будет изменить. В простых приложениях, вроде нашего, это не страшно, но в больших проектах это может привести к затратам как по времени, так и по деньгам.
- Добавляем новые методы в класс `ReportService` - в моменте нам это, дейсвительно, поможет, но затем, если мы захотим дополнить форматы сохранения отчетов, то нам придется менять код, который завязался на новые методы, так как мы явно захотим, чтобы всё наше приложение поддерживало новые форматы.

Также мы здесь не учитываем, что при сохранении в различные форматы нам нужно по-разному представлять структуру отчета - сейчас же это всегда текст в одном и том же формате, что не совсем корректно.

Абсолютно ту же ситуацию можно заметить при отправке отчетов. Если в дальнейшем мы захотим добавить отправку отчетов, например, в Telegram, то мы так же столкнемся с трудностями.

Вариант с соблюдением принципов SOLID

Теперь посмотрим, как могло бы выглядеть то же самое приложение, но с соблюдением принципов SOLID.

Создание проекта

Решение у нас уже создано - поэтому создадим только проект:

```
dotnet new console -n AppWithSolid -o ./AppWithSolid
dotnet sln add ./AppWithSolid/AppWithSolid.csproj
```

Добавление класса для представления отчета

Создадим класс `Report` в папке `Models` проекта:

```
using System.Text;

namespace AppWithSolid.Models;

public sealed record Report(
    string Title,
    string Date,
    string Time,
    int CarsSold,
    int MotorcyclesSold
) {
    public override string ToString()
    {
        var builder = new StringBuilder();

        builder
            .AppendLine(Title)
            .AppendLine($"Дата: {Date:dd.MM.yyyy}")
            .AppendLine($"Время: {Time:HH:mm:ss}")
            .AppendLine("--------------------------------")
            .AppendLine($"Продано автомобилей: {CarsSold} шт.")
            .AppendLine($"Продано мотоциклов: {MotorcyclesSold} шт.")
            .AppendLine("--------------------------------");

        return builder.ToString();
```

```
    }
}
```

Здесь мы использовали синтаксис для создания классов-записей. Класс-запись является неизменяемым классом, который может использоваться для представления данных.

Такой способ представления отчета является болеи гибким и позволяет:

- Легко адаптировать формат представления отчета в зависимости от ситуации - например, для каждого формата хранения отчета можно выбирать свой способ представления того или иного поля
- Легко добавлять новые поля в отчет без изменения существующего кода - старый кодпросто не будет выводить новые поля, но при этом продолжит работать корректно

Добавление интерфейса для сохранения отчета и реализацию для сохранения в текстовый файл

Создадим интерфейс `IReportSaver` в папке `Services` проекта:

```
using AppWithSolid.Models;

namespace AppWithSolid.Services;

public interface IReportSaver
{
    void SaveReport(Report report);
}
```

Добавим реализацию данного интерфейса в папке `Services/Implementations` проекта:

```
using AppWithSolid.Models;

namespace AppWithSolid.Services.Implementations;

internal sealed class TextReportSaver(string fileName) : IReportSaver
{
    public void SaveReport(Report report)
    {
        File.WriteAllText(fileName, report.ToString());
    }
}
```

Такой подход позволяет легко добавлять новые способы сохранения отчетов - при этом для кода, который использует сохранение отчетов, ничего не меняется - он может просто использовать интерфейс `IReportSaver` и не знать, какая конкретно реализация будет использоваться.

Добавление интерфейса для отправки отчета и реализацию для отправки по email

Создадим интерфейс `IReportSender` в папке `Services` проекта:

```
using AppWithSolid.Models;

namespace AppWithSolid.Services;

public interface IReportSender
{
    void SendReport(Report report);
}
```

Добавим реализацию данного интерфейса в папке `Services/Implementations` проекта:

```
using AppWithSolid.Models;

namespace AppWithSolid.Services.Implementations;

internal sealed class EmailReportSender(string email) : IReportSender
{
    public void SendReport(Report report)
    {
        Console.WriteLine($"Отправка отчета на email: {email}");
    }
}
```

Такой подход позволяет легко добавлять новые способы отправки отчетов - при этом для кода, который использует отправку отчетов, ничего не меняется - он может просто использовать интерфейс `IReportSender` и не знать, какая конкретно реализация будет использоваться.

Добавление фасада для работы с отчетами

Создадим класс `ReportService` в папке `Services` проекта:

```
using AppWithSolid.Models;

namespace AppWithSolid.Services;

public sealed class ReportService(IReportSaver reportSaver, IReportSender
reportSender)
{
    public void ProcessReport(Report report)
    {
        reportSaver.SaveReport(report);
        reportSender.SendReport(report);
    }
}
```

Мы выделили ответственность за различные этапы работы с отчетом в отдельные сервисы. Теперь если мы хотим поменять этапы, либо как-то по-другому реализовать один из этапов, то это делается довольно просто и затрагивает минимальное количество кода.

Также мы получили возможность тестировать данный код в дальнейшем.

Добавление использования класса `ReportService` в `Program.cs`

Добавим использование класса `ReportService` в `Program.cs`:

```
using AppWithSolid.Models;
using AppWithSolid.Services;
using AppWithSolid.Services.Implementations;

var textReportSaver = new TextReportSaver("report.txt");
var emailReportSender = new EmailReportSender("example@example.com");

var reportService = new ReportService(textReportSaver, emailReportSender);

var report = new Report(
    Title: "Отчет",
    Date: DateTime.Now.ToString("dd.MM.yyyy"),
    Time: DateTime.Now.ToString("HH:mm:ss"),
    CarsSold: 100,
    MotorcyclesSold: 50
);

reportService.ProcessReport(report);
```

Запуск приложения

Запустим приложение:

```
dotnet run --project ./AppWithSolid/AppWithSolid.csproj
```

Как мы видим, результат работы приложения не изменился, но при этом оно стало более простым в поддержке, а также более простым в расширении.

Анализ примененных принципов SOLID

Теперь посмотрим, какие принципы SOLID мы применили в данном приложении:

- Single Responsibility Principle - мы выделили ответственность за различные этапы работы с отчетом в сущности:
    - Report - представляет содержимое отчета
    - IReportSaver и его реализации - умеют только сохранять отчеты
    - IReportSender и его реализации - умеют только отправлять отчеты

    - ReportService - фасад для работы с отчетами

- Open/Closed Principle - мы можем легко добавлять новые способы сохранения отчетов и отправки отчетов, при этом не меняя ни сами отчеты, ни смежный функционал
- Liskov Substitution Principle - мы можем заменить реализацию интерфейса `IReportSaver` или `IReportSender` на другую реализацию, при этом не меняя код, который использует эти интерфейсы
- Interface Segregation Principle - мы выделили сохранение и отправку отчетов в отдельные интерфейсы - так различный код может использовать только то, что ему нужно
- Dependency Inversion Principle - наш `ReportService` зависит от абстракций `IReportSaver` и `IReportSender` и только вызывающий код решает, как именно он хочет сохранить или отправить отчет