538.73

Рейтинг

OTUS

Цифровые навыки от ведущих экспертов

Подписаться

Lexo 29 сен 2023 в 14:43

Принципы ООП в примерах для начинающих

Простой 8 мин 103K

Блог компании OTUS, Программирование*, .NET*, C#*, ООП*

Туториал

Как создатель и руководитель курсов по C# я вижу, что часто у людей, начинающих изучать этот язык, принципы Объектно-Ориентированного Программирования вызывают затруднения в понимании. А так как один из лучших способов что-то понять, это посмотреть применение на примерах, то я решил написать статью с примерами принципов. Рекомендую найти какую-нибудь статью или книгу, где прочитать основную теорию, а в этой статье уже посмотреть примеры применения этой теории, чтобы понять её лучше.

На текущий момент есть различные точки зрения на то, сколько же в ООП всё-таки принципов и в этой статье мы будем считать, что этих принципов четыре: Инкапсуляция, Наследование,

Полиморфизм и Абстракция. Примеры будут приведены на языке C#, однако, они очень простые, да и сама суть не зависит от языка, поэтому будет полезна всем начинающим изучать ООП программистам.

Инкапсуляция

Инкапсуляция в программировании является объединением данных и кода, работающего с этими данными, в большинстве случае это сводится к тому, чтобы не давать доступа к важным данным напрямую. Вместо этого мы создаем ограниченный набор методов, с помощью которых можно работать с нашими данными. Давайте рассмотрим несколько повседневных примеров, чтобы лучше понять это.

Пример: Уровень заряда батареи смартфона

```
public class Smartphone
{
    private int _batteryLife;

    // Метод заряжает батарею, но не имеет доступа к уровню заряда
    public void Charge(int amount)
    {
        // Устанавливаем свои правила для работы с переменной
        if (amount <= 0)
        {
            throw new ArgumentException("В метод для зарядки телефона передано значение меньш
        }

        _batteryLife += amount;
    }

    // Метод получает текущее значение, но не может его изменить
    public int GetBatteryLife()
    {
        return _batteryLife;
    }
}
```

Здесь `_batteryLife` — это наши важные данные. У нас есть методы для зарядки и показа текущего значения, однако мы не даем доступ к самой переменной `_batteryLife`, поэтому, например, пользователи класса не смогут убавить значение нашей переменной.

Пример: Избранные песни

Давайте представим музыкальное приложение, в котором можно добавлять или удалять песни из своего списка избранного.

```
public class MusicApp
{
    private List<string> _favoriteSongs = new List<string>();

    public void AddToFavorites(string songName)
    {
        if(!string.IsNullOrEmpty(songName) && !_favoriteSongs.Contains(songName))
        {
            _favoriteSongs.Add(songName);
        }
    }

    public void RemoveFromFavorites(string songName)
    {
        _favoriteSongs.Remove(songName);
    }

    public List<string> GetFavorites()
    {
        return new List<string>(_favoriteSongs);
    }
}
```

В этом примере инкапсулирован, то есть спрятан от доступа извне класса, список наших избранных песен (`_favoriteSongs`). Мы предоставляем методы для управления списком, но не даем возможности работать со списком напрямую.

Пример: Переписка в соцсетях

В самом простом случае все, что мы можем сделать при общении в соцсети - отправить кому-то сообщение и прочитать сообщения, отправленные нам. В таком случае, чтобы не дать возможности другим программистам, которые будут использовать наш класс `SocialMediaPlatform` случайно перезаписать наши сообщения, лучше будет предоставить им всего два метода - `SendMessage` и `ShowMyMessages` .

```
public class SocialMediaPlatform
{
    private List<string> _privateMessages = new List<string>();

    public void SendMessage(string message)
    {
        if(!string.IsNullOrEmpty(message))
        {
            _privateMessages.Add(message);
        }
    }
```

```
    public List<string> ShowMyMessages()
    {
        return new List<string>(_privateMessages);
    }
}
```

Как мы видим, сообщения инкапсулированы в списке `_privateMessages` и код, использующий наш класс, не может делать с нашими сообщения ничего, кроме получения текущих и добавления новых.

Наследование

Наследование в какой-то степени похоже с биологическим наследованием. Вы получаете какие-то черты от своих родителей, но, в то же время, отличаетесь от них. Или представьте это как базовую модель гаджета, к которой затем добавляются улучшенные версии с дополнительными функциями. Давайте рассмотрим несколько примеров, чтобы лучше понять это.

Пример: Игры и дополнения

Предположим, вы купили игру в Стиме и через какое-то время у неё появляются два дополнения: HD version и HotA, которые основаны на оригинальной игре, но изменяют её части.

```
public class HeroesOfMightAndMagic3
{
    public void Play()
    {
        Console.WriteLine("Запускаем классическую версию игры...");
    }
}

public class HeroesOfMightAndMagic3Hd : HeroesOfMightAndMagic3
{
    public void PlayHd()
    {
        Console.WriteLine("Запускаем игру в высоком разрешении (HD)...");
    }
}

public class HeroesOfMightAndMagic3Hota : HeroesOfMightAndMagic3
{
    public void PlayHota()
    {
        Console.WriteLine("Запускаем игру с двумя новыми городами...");
    }
}
```

Классы `HeroesOfMightAndMagic3Hd` и `HeroesOfMightAndMagic3Hota` наследуют метод `Play` для запуска оригинальной версии игры, но также каждый добавляет свои уникальные методы.

Пример: Версии смартфона

Рассмотрим смартфон, у которого есть базовая модель и есть версия Pro, которая наследует все базовые функции, плюс, добавляет некоторые продвинутые.

```
public class BasicSmartphone
{
    public void Call()
    {
        Console.WriteLine("Совершаем звонок...");
    }
}

public class ProSmartphone : BasicSmartphone
{
    public void VideoCall()
    {
        Console.WriteLine("Совершаем видеозвонок...");
    }
}
```

`ProSmartphone` может звонить так же, как и `BasicSmartphone`, но также имеет дополнительную функцию видеозвонка.

Пример: Онлайн кинотеатр

Онлайн кинотеатры часто предоставляют различные подписки для своих пользователей. Рассмотрим пример, где у такого кинотеатра есть базовый тариф и премиальный тариф, который предлагает все основные функции плюс эксклюзивный контент.

```
public class BasePlan
{
    public void StreamStandardContent()
    {
        Console.WriteLine("Показываем контент базового плана...");
    }
}

public class PremiumPlan : BasePlan
{
    public void StreamExclusiveShows()
    {
        Console.WriteLine("Показываем контент премиального плана...");
```

```
    }
}
```

`PremiumPlan` предлагает все, что и `BasePlan`, но также добавляет и новый, эксклюзивный контент.

Полиморфизм

Полиморфизм немного напоминает универсальный пульт дистанционного управления, который может адаптироваться для управления различными устройствами. В программировании это означает, что один интерфейс может использоваться для управления разными методами, давая разные результаты в зависимости от контекста.

Пример: Музыкальный плеер

Представьте себе музыкальный плеер, который может воспроизводить разные аудиоформаты, такие как mp3, wav и flac. Для каждого формата требуется свой метод воспроизведения, однако, вместо создания методов `Play`, `PlayMp3`, `PlayWav`, `PlayFlac`, правильнее будет использовать общий метод `Play`.

```
public class MusicPlayer
{
    public virtual void Play()
    {
        Console.WriteLine("Воспроизводим аудио в стандартном формате...");
    }
}

public class Mp3Player : MusicPlayer
{
    public override void Play()
    {
        Console.WriteLine("Воспроизводим mp3...");
    }
}

public class WavPlayer : MusicPlayer
{
    public override void Play()
    {
        Console.WriteLine("Воспроизводим wav...");
    }
}

public class FlacPlayer : MusicPlayer
{
    public override void Play()
```

```
    {
        Console.WriteLine("Воспроизводим flac...");
    }
}
```

В этом примере независимо от аудиоформата у нас есть один постоянный метод `Play`, выполнение которого меняется в зависимости от формата.

Пример: Виртуальный ассистент

Подумайте о виртуальном ассистенте, который работает на смартфоне, смарт-часах и смарт-колонке. Вы можете попросить все эти устройства "Включить свет", но ответ может быть адаптирован в зависимости от устройства.

```
public class VirtualAssistant
{
    public virtual void ExecuteCommand(string command)
    {
        Show($"Выполняю команду {command}...");
    }
}

public class SmartwatchAssistant : VirtualAssistant
{
    public override void ExecuteCommand(string command)
    {
        ShowOnSmallScreen($"Выполняю команду {command}...");
    }
}

public class SmartSpeakerAssistant : VirtualAssistant
{
    public override void ExecuteCommand(string command)
    {
        Say($"Выполняю команду {command}...");
    }
}
```

Команда одинакова, но ее выполнение адаптируется в зависимости от контекста устройства. В базовом случае мы просто выводим сообщение о том, что команда выполняется, на экран (`Show`). У умных часов экран маленький, поэтому нам нужен особый способ вывода сообщения на экран (`ShowOnSmallScreen`), а у умной колонки вообще может не быть экрана, поэтому сообщение лучше озвучить голосом (`Say`).

Пример: Потоковое видео

Рассмотрим платформу потокового видео, которая изменяет свое качество воспроизведения в зависимости от скорости интернета пользователя: HD, SD или 4K.

```
public class VideoStreamer
{
    public virtual void Stream()
    {
        Console.WriteLine("Показываем в обычном качестве...");
    }
}
```



```
    public override void Stream()
    {
        Console.WriteLine("Показываем в HD качестве...");
    }
}

public class SDStreamer : VideoStreamer
{
    public override void Stream()
    {
        Console.WriteLine("Показываем в SD качестве...");
    }
}

public class FourKStreamer : VideoStreamer
{
    public override void Stream()
    {
        Console.WriteLine("Показываем в 4K качестве...");
    }
}
```

Независимо от качества интернета, пользователь просто запускает метод Stream, а платформа корректирует качество трансляции сама.

Абстракция

Абстракция похожа на использование умного устройства, не зная его сложной схемы. Например, чтобы переключить канал на телевизоре, мы просто нажимаем на кнопку на пульте, как кодируется пультом нажатие на кнопку, передается на телевизор и декодируется нам не важно. Важно чтобы канал переключился, а не тонкости радиотехники. Вот и в программировании абстракция означает предоставление основных функций без погружения в детали.

Пример: Автомобиль

Чтобы управлять автомобилем, нам в базовом случае достаточно знать о том, где находится руль, педаль тормоза и газа (да-да, и педаль сцепления для механики). То есть чтобы ехать нам совсем не нужно понимать тонкости работы двигателя, передачи крутящего момента, как устроен гидро или электроусилитель руля. Мы просто нажимаем на газ и машина едет, крутим руль и она поворачивает. Это и есть абстракция.

```
public abstract class Car
{
    public void Accelerate()
    {
        Console.WriteLine("Разгоняемся...");
    }

    public void Brake()
    {
        Console.WriteLine("Тормозим...");
    }

    // Абстрактный метод запуска, различающийся для разных двигателей
    public abstract void TurnOnEngine();
}

public class ElectricCar : Car
{
    public override void TurnOnEngine()
    {
        Console.WriteLine("Запускаем электрический двигатель...");
    }
}

public class DieselCar : Car
{
    public override void TurnOnEngine()
    {
        Console.WriteLine("Запускаем дизельный двигатель...");
    }
}
```

Независимо от типа автомобиля, мы запускаем двигатель нажатием на кнопку Start, не обращая внимания на то, что на самом деле процесс под капотом различается.

Пример: Погода

Что мы делаем, чтобы узнать прогноз погоды на сегодня? Просто открываем приложение на телефоне и оно показывает нам погоду. Как оно собирает для этого данные, как их обрабатывает, все это скрыто от нас.

```
public abstract class WeatherApp
{
    public void DisplayForecast()
    {
        Console.WriteLine("Показываем текущий прогноз погоды...");
    }

    // Абстрактный метод получения данных, различающийся для текущего способа связи
    public abstract void GetWeatherData();
}

public class WifiWeatherApp : WeatherApp
{
    public override void GetWeatherData()
    {
        Console.WriteLine("Запрашиваем данные по WiFi...");
    }
}

public class MobileWeatherApp : WeatherApp
{
    public override void GetWeatherData()
    {
        Console.WriteLine("Запрашиваем данные по мобильной сети...");
    }
}
```

Мы просто видим прогноз погоды на экране, хотя способ его получения может отличаться.

Пример: Кофемашина

Чтобы приготовить кофе в кофемашине мы заливаем воду, засыпаем кофейные зерна и выбираем тип кофе. Как дальше кофемашина заваривает его, скрыто от нас.

```
public abstract class CoffeeMachine
{
    public void PourWater()
    {
        Console.WriteLine("Заливаем воду...");
    }

    public void AddBeans()
    {
        Console.WriteLine("Засыпаем зерна...");
    }

    // Абстрактный метод, специфичный для каждой кофемашины
```

```
    public abstract void BrewCoffee();
}

public class EspressoMachine : CoffeeMachine
{
    public override void BrewCoffee()
    {
        Console.WriteLine("Варим с использованием пара под высоким давлением...");
    }
}

public class DripCoffeeMachine : CoffeeMachine
{
    public override void BrewCoffee()
    {
        Console.WriteLine("Пропускаем горячую воду через зерна...");
    }
}
```

В обоих случаях мы в результате получаем кофе, но метод заваривания (абстрагированный процесс) различается между машинами.

Заключение

Хоть эти концепции и могут казаться абстрактными, я очень надеюсь, что аналогии из реальной жизни и примеры кода помогают их понять. При этом, важно помнить, что ООП - это не серебрянная пуля и не высеченные в камне истины, которым всегда и везде нужно следовать. Ведь самое главное в нашей работе - это создание кода, который решает реальные проблемы, ну и желательно, чтобы его было удобно поддерживать и масштабировать.

Хочу также пригласить вас на бесплатный вебинар, где я расскажу про пять ключевых программных парадигм в C#: процедурное, объектно-ориентированное, функциональное, событийное и компонентно-ориентированное программирование. Мы рассмотрим основные характеристики каждого подхода, их преимущества и недостатки, а также примеры их применения на практике. Регистрируйтесь, будет интересно!

Теги: ООП, c#

Хабы: Блог компании OTUS, Программирование, .NET, C#, ООП