

Návrhové vzory I

Návrhové vzory pro vytváření instancí,
rozšiřování funkcionality

Obecná definice

- Představují šablony pro řešení jednoho nebo více konkrétních problémů
- Jejich účel spočívá ve zjednodušení problémů svou aplikací
- Díky ověřené funkčnosti snižují možnosti potenciálních chyb
- Lze se na ně dívat podobně jako na vzorečky v matematice a fyzice

Druhy návrhových vzorů

- Obecně se návrhové vzory dělí do 3 typů podle toho, čemu jejich aplikace má pomoci a jaké problémy řeší:
 - **Vzory chování** (behavioral patterns)
 - Mění nebo vytváří chování objektů, jak vnitřně, tak mezi různými objekty
 - Patří sem např. **MVC, interpreter, observer, iterátor**, ...
 - **Strukturální vzory** (structural patterns)
 - Řeší uspořádání (strukturu) systému a jeho komponent
 - Patří sem např. **zástupce, adaptér, prázdný objekt, neměnný objekt**, ..
 - **Vzory pro vytváření** (creational patterns)
 - Zabývají se problémy v souvislosti s tvorbou objektů, většinou počet
 - Patří sem např. **Singleton, Pool, Knihovna, Tovární metoda**, ..

Příklady návrhových vzorů

Vzory pro vytváření

- **TOVÁRNÍ METODA**
 - Slouží k vytvoření nové instance třídy na základě nějakých parametrů a vnitřní logiky, vytvořenou instanci může i vrátet
 - Využívá se tam, kde vytvářený objekt/objekty jsou odvozeny od stejné třídy
 - Často se využívá u neměnných (immutable) objektů, jejichž stav po vytvoření nelze změnit.

Příklad z úlohy „Návrhové vzory I“, třída Osoba

```
public static Osoba getInstance(int age, GenderEnum gender, string name)
{
    return age switch
    {
        _ when age < 0 => null,
        _ when age < 8 => new Předškolák(age, gender, name),
        _ when age < 20 => new Školák(age, gender, name),
        _ when age < 66 => new Pracující(age, gender, name),
        _ => new Důchodce(age, gender, name),
    };
}
```

- **SINGLETON**
 - Singleton se využívá v situaci, kdy může existovat maximálně jedna instance dané třídy
 - Zároveň by měla samotná instance být centralizovaně dostupná (= z více míst/bodů)
 - Musí být zajištěno několik kritérií, aby byl Singleton plně funkční a bezpečný
 - Nelze vytvářet instance mimo samotný Singleton
 - Musí být vláknově bezpečný a měl by být serializovatelný

- Implementací existuje několik, každá přináší nějaké bonusy a nevýhody - viz. Příklad, který nemá zajištěnou vláknovou stabilitu

Příklad z úlohy „Návrhové vzory II“, třída Předškolák

```
private static Předškolák existingInstance = null;

public Předškolák(int age, GenderEnum gender, string name)
    : base(age, gender, name)
{
    existingInstance = this;
}
■ }
```

• FOND (POOL)

- Využitím fondu umožňujeme znovu použít existující instance třídy, pokud už nejsou někde potřeba, mohou se využít jinde
- Řeší se tím paměťové náročné třídy, nebo třídy s velkým množstvím instancí během běhu programu
- Implementace spočívá ve vytvoření nějaké kolekce, která bude obsahovat veškeré instance
- Dále se vytvoří metoda pro získání volné instance a pro návrat nepoužívané instance
- Musí se také rozhodnout, co se stane při nedostatku dostupných instancí (vytvořit novou, počkat na dostupnou, vyhodit výjimku, ...)

```
private const int ModulesCount = 5;
public static Module[] Modules = new Module[ModulesCount]
{
    new Module(Actions.BROUSENI),
    new Module(Actions.BROUSENI),
    new Module(Actions.REZANI),
    new Module(Actions.SVAROVANI),
    new Module(Actions.VRTANI)
};

public Module takeModule(Actions a)
{
    if (Modules.Where(x => !x.InUse && x.Action == a).Count() > 0) // InUse
        představuje dostupnost - jestli je modul používán
        return Modules.Where(x => !x.InUse && x.Action == a).ToArray()[0];
    else
        throw new Exception("Žádné dostupné moduly");
}
■ }
```

• KNIHOVNA

- Knihovna nebo Utility představuje nějakou skupinu metod, které spolu nějak souvisí a jsou využívány společně, uloženou v jedné třídě
- Tato třída by neměla být schopná dědit a vytvářet instance - v C# toto splňuje modifikátor **static**
- Řešení pomocí knihoven zlepšuje přehlednost a přístupnost, popřípadě je i znovu použitelná (pokud je navržena dostatečně obecně)

```

namespace MT_DP
{
    public static class UtilityExample // modifikátory public a static
    zpřístupní třídu a zakážou jí dědit a instancovat
    {
        public static float GetArithmeticAverage(float[] numbers)
        {
            return numbers.Sum() / numbers.Length;
        }

        public static float GetHarmonicAverage(float[] numbers)
        {
            Array.ForEach(numbers, x => x = 1 / x);
            return (float)Math.Pow(GetArithmeticAverage(numbers), -1);
        }
    }
}

```

- **ORIGINÁL**

- Originál představuje třídu, jejíž instance jsou svými parametry unikátní a neexistuje víc jak jedna instance se stejnými parametry
- S tímto řešením může být spojená identifikace přes nějaký klíč nebo seed vygenerovaný na základě parametrů instance pro jednoduchý přístup

```

public class Color
{
    private static Dictionary<int, Color> colorStorage { get; set; }
    private readonly int r, g, b;
    private Color(int r, int g, int b)
    {
        this.r = r;
        this.g = g;
        this.b = b;
    }
    public static Color GetColor(int r, int g, int b)
    {
        if ((r < 0) || (r > 255) || (g < 0) || (g > 255) || (b < 0) || (b > 255))
        {
            throw new Exception("Špatný rozsah barev");
        }
        int key = r + g * 256 + b * 65536;
        if (!colorStorage.ContainsKey(key))
        {
            colorStorage.Add(key, new Color(r, g, b));
        }
        return colorStorage.GetValueOrDefault(key);
    }
}

```

Strukturální vzory

- **MUŠÍ VÁHA (FLYWEIGHT)**

- Toto řešení spočívá ve sdílení prostředků, které využívá více instancí místo toho, aby se prostředky vytvářely pro každou instanci zvlášť (např. textury)

- Podobá se to trochu vzoru fond, ale místo instancí se to týká prostředků, s kterými instance pracují
- Primární účelem řešení je ušetření paměťové náročnosti programu v případech, kdy prostředky jsou paměťově náročné, nebo s nimi pracuje velké množství instancí

```
public class FlyWeightExample
{
    public string CompanyName { get; set; }
    public string CompanyLocation { get; set; }
    public string CompanyWebSite { get; set; }
    // Náročná/velká data
    public byte[] CompanyLogo { get; set; }
}

public static class FlyWeightPointer
{
    public static readonly FlyWeightExample Company = new FlyWeightExample
    {
        CompanyName = "Abc",
        CompanyLocation = "XYZ",
        CompanyWebSite = "www.abc.com"
        // načteme CompanyLogo zde
    };
}

public class MyObject
{
    public string Name { get; set; }
    public string Company
    {
        get
        {
            return FlyWeightPointer.Company.CompanyName;
        }
    }
}
```

- **NEMĚNNÉ (Immutable) objekty**

- Neměnný objekt je hodnotový objekt, u kterého nejde změnit jeho hodnota
- Používání proměnných objektů výrazně snižuje bezpečnost programu
- Se změnou hodnoty se mění i její hash-code
- Příkladem neměnného datového typu je například string.
- Pokaždé, když měníme/modifikujeme jeho hodnotu, se vytváří nový objekt a odkaz se přemísťuje na něj
- Starý objekt zůstává bez odkazu a musí ho odstranit garbage collector.
- Proměnným ekvivalentem stringu je StringBuilder



```

class MyClass
{
    private readonly string myString;
    public string GetStr { get { return myString; } }
    public MyClass(string str)
    {
        myString = str;
    }
}
  
```

- **PŘEPRAVKA (Crate, DTO)**

- Přepravka se využívá na předávání několika samostatných informací
- Přeppravku řadíme mezi „kontejnery“
- Atributy přepravky se mohou definovat jako konstantní, aby byla zajištěna jejich neměnnost
- Dobrým příkladem jsou souřadnice
- Místo toho, abychom si říkali o každou hodnotu zvlášť (getX() + getY() + getZ()), požádáme o celou přeppravku.

```

class Souradnice
{
    public int x;
    public int y;
    public int z;
    public Souradnice(int x, int y, int z)
    {
        this.x = x;
        this.y = y;
        this.z = z;
    }
}

class pocitame
{
    public Souradnice getSouradnice()
    {
        return new Souradnice(12, 24, 10);
    }
}
  
```

Vzory chování

- **SLUŽEBNÍK (Servant)**

- Služebníka používáme v případě, že potřebujeme aby instance více tříd, které nemohou mít společného předchůdce, obsahovali společnou funkčnost
- Služebník obsahuje funkce, které potřebujeme
- Objekty, které mu dodáváme jako parametry, musí splňovat určité rozhraní, které vyžaduje služebník.
- Dvě možná použití:
 - Klient zná služebníka
 - Klient nezná služebníka

```

class Servant
{
    public void skok(Iskokan obj, int delka_skoku)
    {
        obj.x += delka_skoku;
    }
}

class Atlet : Iskokan
{
    public int x = 0;
}

class Klient
{
    public static void main(String[] args)
    {
        public Servant sluzebnik = new Servant();
        sluzebnik.skok(new Atlet, 5);
    }
}

```

- **PRÁZDNÝ (Null) objekt**

- V případě, kdy by nám použití klasické hodnoty null mohlo přinést problémy, použijeme Null Object, který je v podstatě plnohodnotný objekt, ovšem relativně prázdný
- Díky použití tohoto nám odpadá neustálé testování návratových hodnot zda nejsou null a tedy s nimi můžeme jednodušeji pracovat

```

class Animal : IAnimal
{
    public static readonly IAnimal Null = new NullAnimal();
    private class NullAnimal : Animal
    {
        public override void Sound() { } //prázdné
    }
    public abstract void Sound();
}

class Dog : IAnimal
{
    public void Sound()
    {
        Console.WriteLine("Haf");
    }
}

class Program
{
    public static void main(String[] args)
    {
        IAnimal dog = new Dog();
        dog.Sound(); //vypíše "Haf"
        IAnimal unknown = Animal.Null; //zde přiřazujeme Null Objekt
        dog.Sound(); //nic se nestane
    }
}

```