

# SOLIDní kód

**Martin Dybal**

Microsoft MSP, MCP

[www.dotnetcollege.cz](http://www.dotnetcollege.cz)

# SOLIDní kód

- CleanCode
- Návrhové vzory
- SOLID
- IoC/DI


[www.dotnetcollege.cz](http://www.dotnetcollege.cz)



# Návrhové vzory

Návrhové vzory dohromady zapouzdřují osvědčené způsoby pro využívání možností jazyka. Umožňují programovat na vyšší úrovni a propagují dobré programátorské praktiky.


*Judith Bishop*

- Již „nepoužívané“ návrhové vzory
  - Návrhové vzory GoF
  - EEA – Enterprise návrhové vzory Martin Fowler
  - Paralelní návrhové vzory
- 

# SOLID - Hlavní účely Hlavní účely

- Testování
- Udržitelný kód
- Snadné rozšiřování

# Single Responsibility Principle

- Každá třída má jen jednu odpovědnost
    - Neznamená to, že má jen jednu metodu!
    - Spíš aby každá třída měla jen jeden důvod ke změně.
- 

# Open / Closed Principle

- Otevřenost pro rozšíření, uzavřenost pro změny
- Navrhujeme rozhraní tak, aby nebyla omezující (abychom snadno mohli přidávat a rozšiřovat), ale abychom je nemuseli již měnit

# Liskov Substitution Principle

- Instanci lze nahradit instancí poděděné třídy
  - Rozhodně ne toto:
    - Podědíme List<string>
    - Rušení metod vyhozením NotSupportedException
    - Změna chování oproti původní třídě

# Interface Segregation Principle

- De facto SRP pro rozhraní
  - Nedělat jedno velké rozhraní s 30 metodami, použít více malých rozhraní s jasně definovanými odpovědnostmi




# Dependency Inversion Principle

- Třídy mají své závislosti deklarovat navenek a nechat si je naplnit zvenčí

```
public NewsletterService()  
{  
    this.contactList = new ContactList();  
    this.MailerService = new MailerService();  
}  
  
internal class NewsletterService : INewsletterService  
{  
    private readonly ContactList contactList;  
    :  
    3 references  
    public IMailerService MailerService { get; set; }  
    :  
    0 references  
    public NewsletterService(ContactList contactList, IMailerService mailerService)  
    {  
        this.contactList = contactList;  
        this.MailerService = mailerService;  
    }  
}
```

# Dependency Inversion Principle

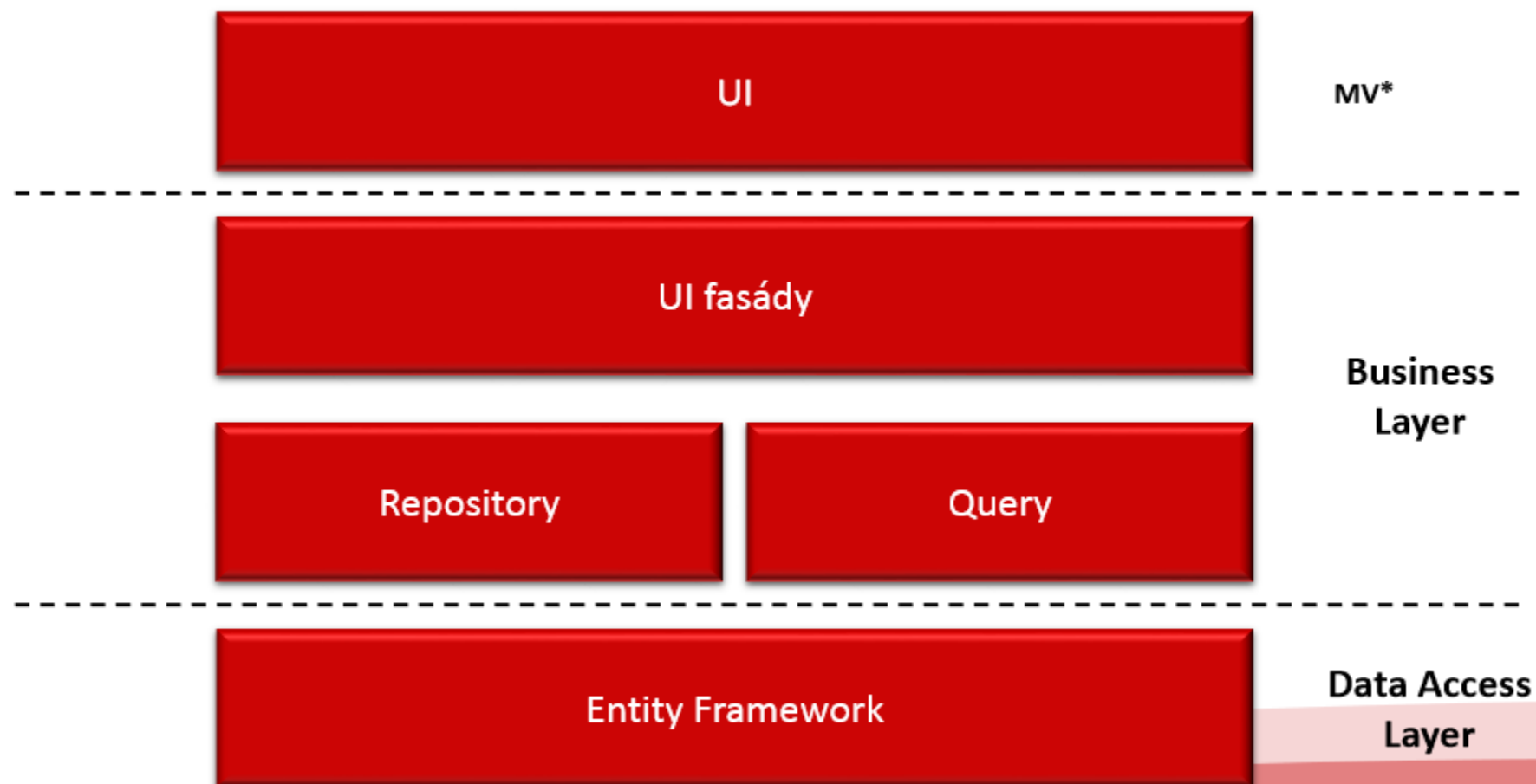
- Druhy závislostí
    - Constructor Dependency **CI**
      - Závislost je předána jako parametr konstruktoru
    - Property Dependency **DI**
      - Závislost je držena ve vlastnosti třídy
- 

# IoC/DI

- **Inversion of Control**


- Třídy o sobě neví, komunikují přes rozhraní
- Starají se jen o své věci
- Třídy, které potřebuje, si nevytváří sama
- Výhoda: krabičky s přesně danou odpovědností
- Kdykoliv je lze nahradit za jinou implementaci
  - A to i na úrovni konfigurace aplikace

# Příklad architektury



# Prezenční návrhové vzory

Prezenční návrhové vzory se zabývají rozdělením nejvyšší (UI) vrstvy .

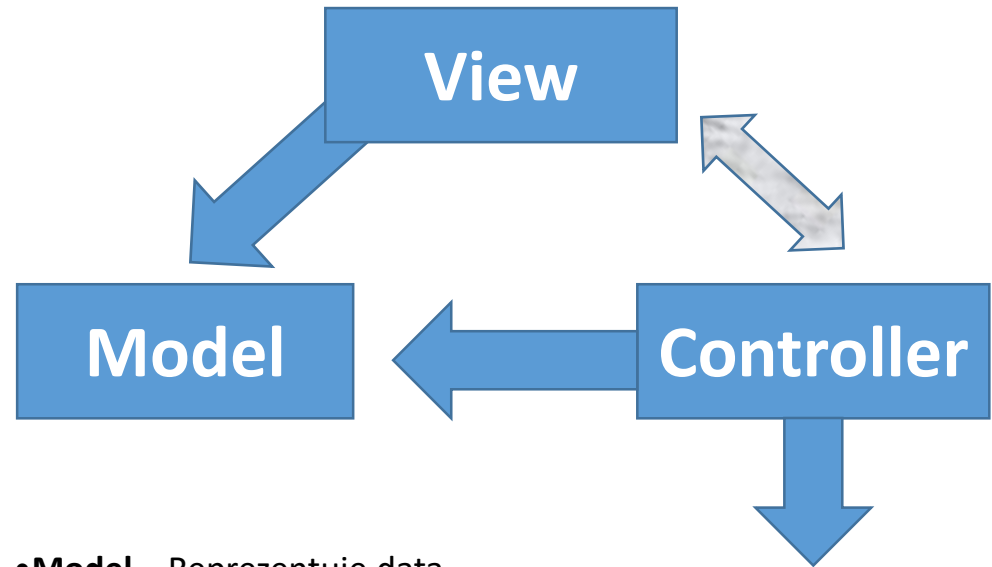
- **Hlavní účely**
  - Oddělit logiku zobrazování dat od logiky zpracování
  - Testovatelný kód
  - Znovu použitelnost kódu
- 

# Model View Controller

- V bez stavovém prostředí

## Life cycle

1. Uživatel interaguje s UI
2. Je volána příslušná metoda controlleru
  1. Ověří práva
  2. Validnost vstupních dat
  3. Vykoná požadovanou operaci
  4. Rozhodne, které view se má vrátit a předá mu potřebný model
3. Uživatel vidí nové view



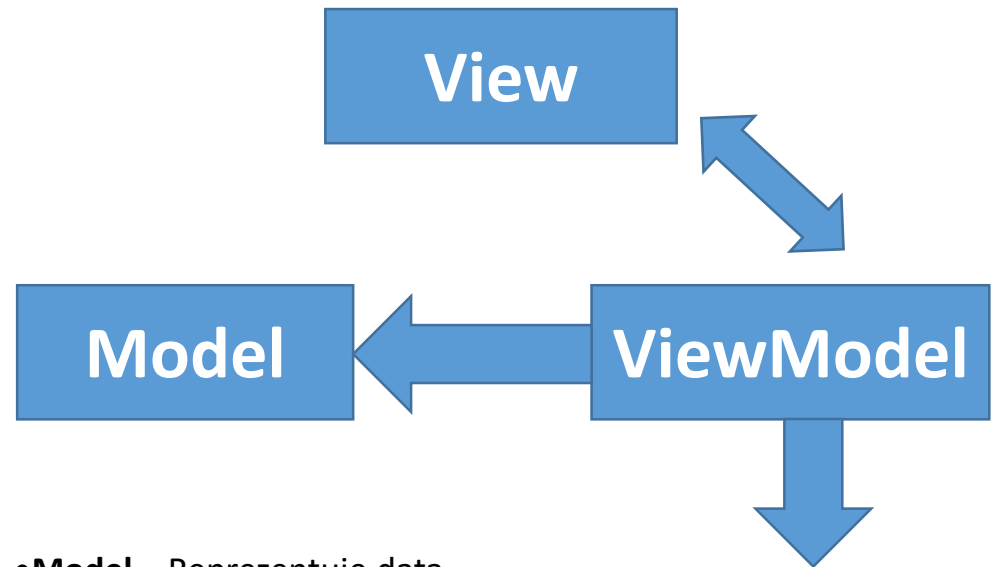
• **Model** – Reprezentuje data.

• **View** – Zobrazuje data uživateli

• **Controller** – zajišťuje požadavky od uživatele, ověřuje práva, validnost vstupních dat, „vrací view“.

# Model View ViewModel

- Ve stavovém prostředí



- **Model** – Reprezentuje data.
- **View** – Zobrazuje data uživateli a dává mu možnost ovládání programu a zadávání nových dat.
- **ViewModel** – drží si kontext aktuální (části) obrazovky.

# Návrhové vzory GoF

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides

Design Patterns: Elements of Reusable Object-Oriented Software

## Strukturální vzory:

- Decorator
- Proxy
- Bridge
- Composite
- Flyweight
- Adapter
- Façade

## Vytvářecí vzory

- Factory Method
- Abstract Factory
- Builder
- Prototype
- Singleton

## Vzory chování:

- Strategy
- State
- Template Method
- Chain of Responsibility
- Command
- Iterator
- Mediator
- Observer
- Visitor
- Interpreter
- Memento

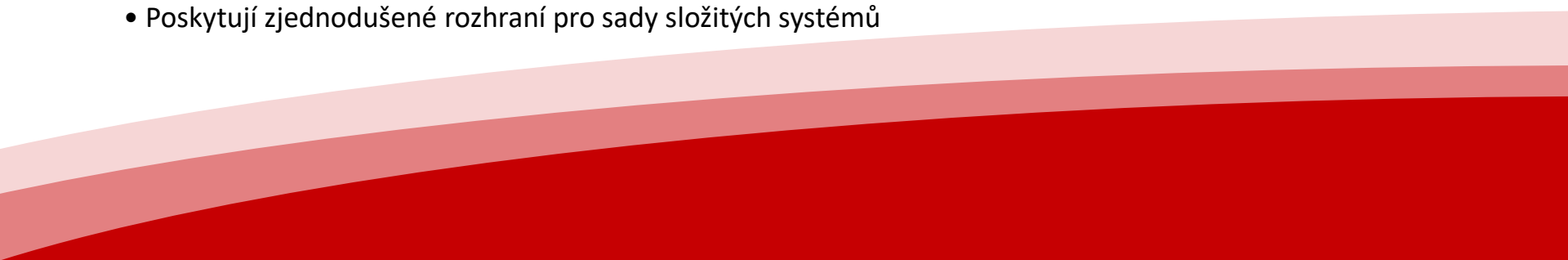




# Strukturální návrhové vzory

Strukturální návrhové vzory se zabývají skladbou tříd a objektů do větších struktur.

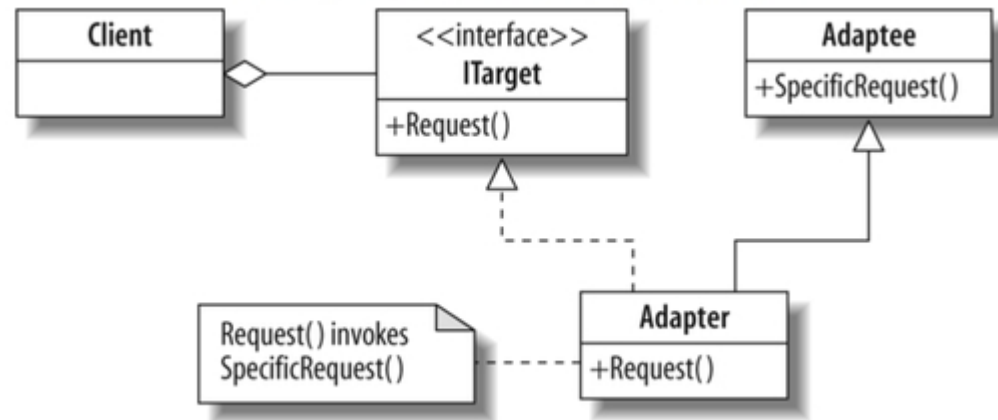
## • Hlavní účely

- Dynamicky přidávají funkcionalitu k již existujícím objektům
  - Řídí přístup k objektům
  - Na vyžádání vytváří nákladné objekty
  - Umožňují vyvíjet rozhraní a implementaci nezávisle na sobě
  - Umožňují spolupráci se vzájemně nekompatibilním rozhraním
  - Redukují náklady na práci s velkým množstvím malých objektů
  - Vybírají nebo přepínají implementace za běhu
  - Umožňují spravovat jednoduché a složené objekty stejným způsobem
  - Propojují typy, které nebyly původně navrženy pro vzájemnou spolupráci
  - Poskytují zjednodušené rozhraní pro sady složitých systémů
- 

# Adaptér/Wrapper

- **Účel**

- Přizpůsobit rozhraní
- Obalit implementace tenkou vrstvou, pokud může dojít ke změně. (API, ORM, ...)



- **ITarget** - Rozhraní, které chceme dále používat.
- **Adapter** - Třída, která upravuje adaptee na rozhraní ITarget
- **Adaptee** – Původní rozhraní/implementace

# Adaptér/Wrapper

## Příklad

Chceme pracovat s daty se sociální sítí (Facebook, Twitter, Google plus). Chceme osvobodit business vrstvu od nutnosti pracovat s různými api, za tímto účelem jsme vytvořili interface ISocialApi.

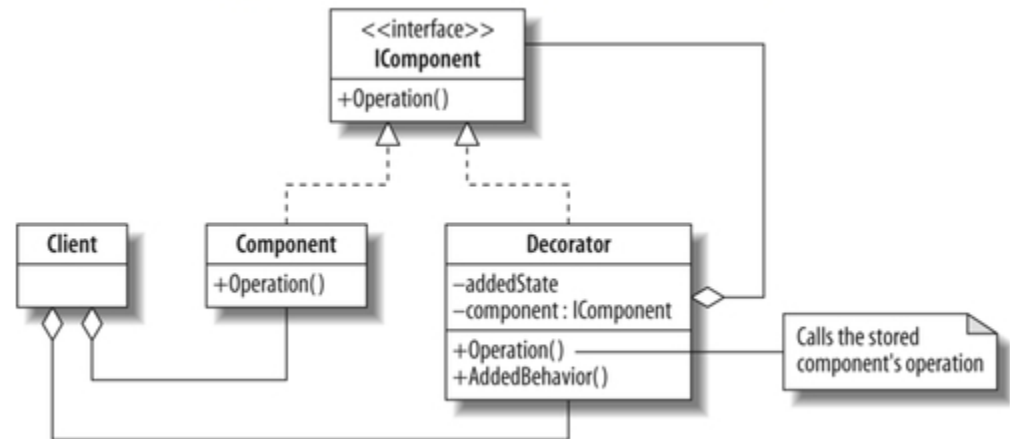
<b>1) ITarget</b>	A
<b>2) Adapter</b>	B, E
<b>3) Adaptee</b>	C, D

<b>a) ISocialApi</b>
<b>b) FacebookSocialApi</b>
<b>c) Facebook API</b>
<b>d) Twitter API</b>
<b>e) TwitterSocialApi</b>

# Decorator

- Účel

- Poskytuje způsob pro dynamické připojení nového stavu nebo chování k objektu.
- Změnit určité vlastnosti objektu bez ovlivnění dalších
- Obzvláště výhodný, pokud mohou existovat objekty s libovolnými kombinacemi nových vlastností, jelikož při použití prosté dědičnosti by došlo k enormnímu nárustu počtu tříd.



- **Icomponent** - Rozhraní identifikující rozhraní, které mohou být dekorovány
- **Component** - Původní třída objektů, jejichž operace mohou být přidávány nebo měněny
- **Operation** - Metoda objektu Component zahrnutá v rozhraní IComponent. (Metod může být n)
- **Decorator** - Třída implementující Icomponent (Tříd může být n)

# Decorator

## Příklad

Chceme vytvořit jednoduchou aplikaci pro úpravu aplikací. Aplikace má umožňovat přidávat k fotografii rámečky a popisky.

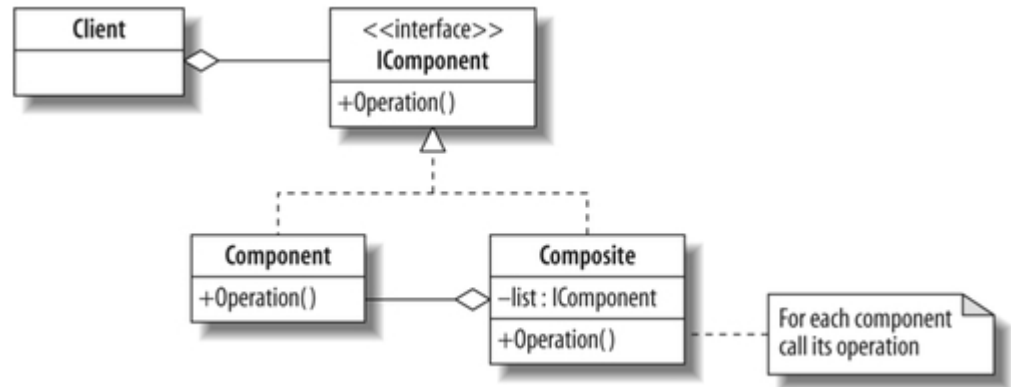
1) <b>IComponent</b>	C
2) <b>Component</b>	D
3) <b>Decorator</b>	B
4) <b>Operation</b>	A

a) Zobrazit fotku
b) Fotka s rámečkem
c) Fotka
d) Původní fotka

# Composite

- **Účel**

- Vytváří strukturované hierarchie, se kterými lze pracovat jako se skupinami komponent.
- Typické operace na komponentách jsou přidávání, upravování, vyhledávání, seskupování a zobrazování.



- **IComponent** – Společné rozhraní
- **Composite** – Skupina Icomponent, operace se vždy aplikuje na všechny prvky
- **Component** – Konečný prvek

# Composite

## Příklad

Potřebujeme reprezentovat hierarchii souborů.

1) <b>IComponent</b>	C
2) <b>Composite</b>	A
3) <b>Component</b>	B

a) Složka
b) Soubor
c) Složka nebo soubor

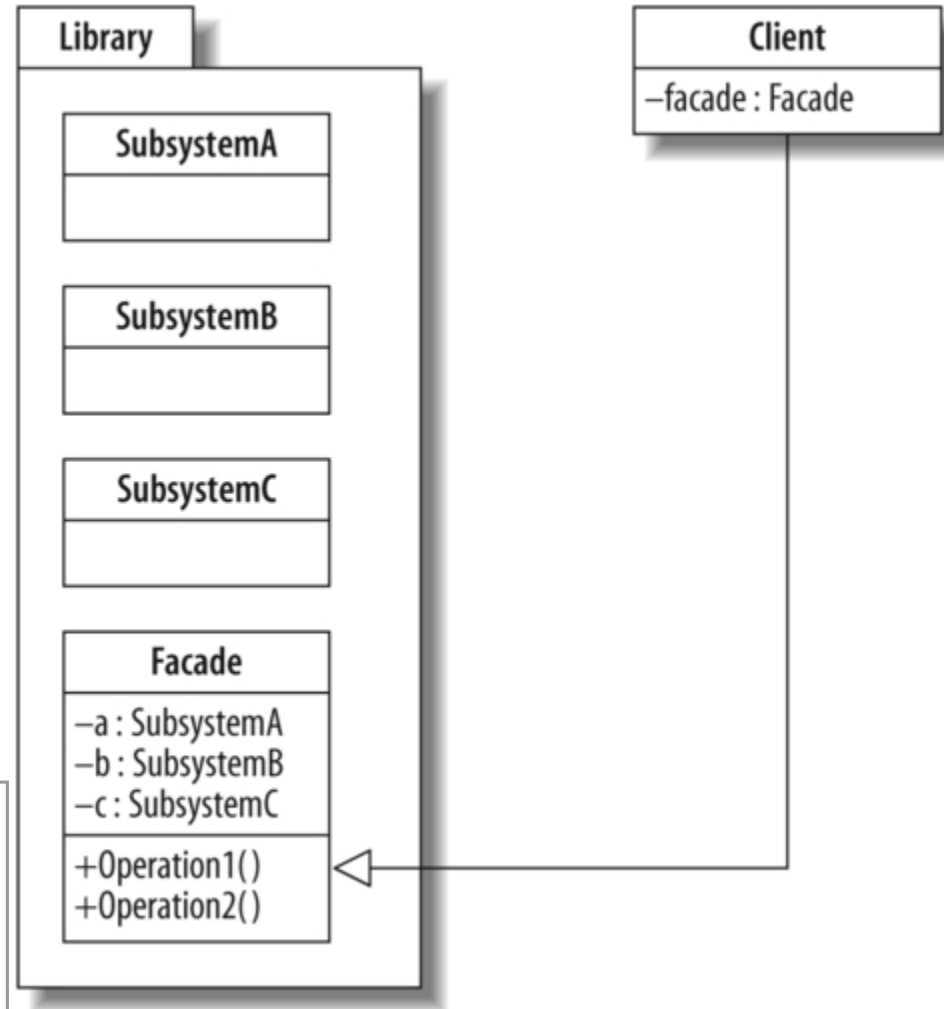
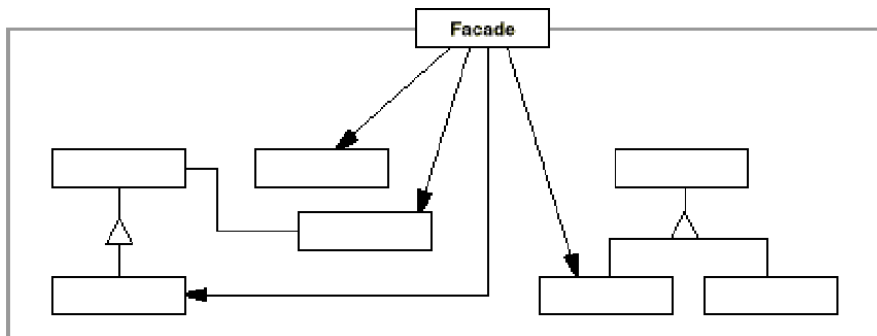
Napadá vás kdy bude **Composite** i **Component** stejný = bude existovat pouze **Composite**?

- A co třeba reprezentace **HTML dokumentu**?

# Façade

- Účel

- Zastřešit operace business vrstvy pro volání UI
- Velmi vhodné v návrhu univerzálního kódu.
- Vhodné pro propojení s prezenčními návrhovými vzory MV\*





# Singelton

- Účel

- Zajišťuje že vždy bude existovat pouze jedna instance

```
public sealed class Setting
{
    2 references
    public static Setting Instance { get; private set; }
    0 references
    public int Level { get; set; }
    0 references
    static Setting()
    {
        Instance = new Setting();
    }
    1 reference
    private Setting()
    {
    }
}

//Použití
Setting.Instance.Level = 5;
```

Často za implementaci odpovídá IoC

# Vzory chování

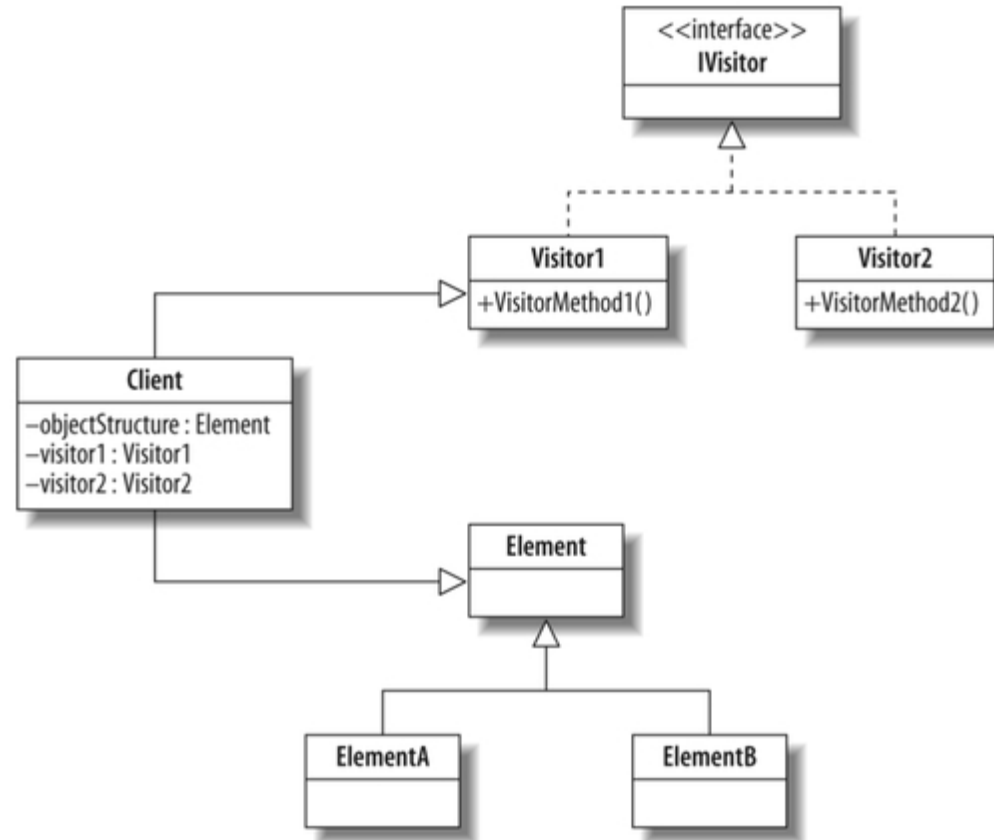


# Visitor

- Účel

- Je potřeba rozšířit možnosti hierarchie
- Velmi často užívaný v kombinaci s Composite
- **Pozor** – velmi nepraktické, pokud dochází k nárůstu tříd

```
public virtual T Visit(Decorator.Expression node)
{
    if (node is Decorator.UnaryMinus)
    {
        return this.VisitUnaryMinus((Decorator.UnaryMinus)node);
    }
    if (node is Decorator.BinaryPlus)
    {
        return this.VisitBinaryPlus((Decorator.BinaryPlus)node);
    }
    if (node is Decorator.Constant)
    {
        return this.VisitConstant((Decorator.Constant)node);
    }
    throw new NotImplementedException();
}
```



# Visitor

## Příklad

Máme HTML dokument reprezentovaný jako Composite a chceme zajistit jeho vykreslování, musíme rozlišit vykreslování podle vlastnosti **display** (block, inline block, text).

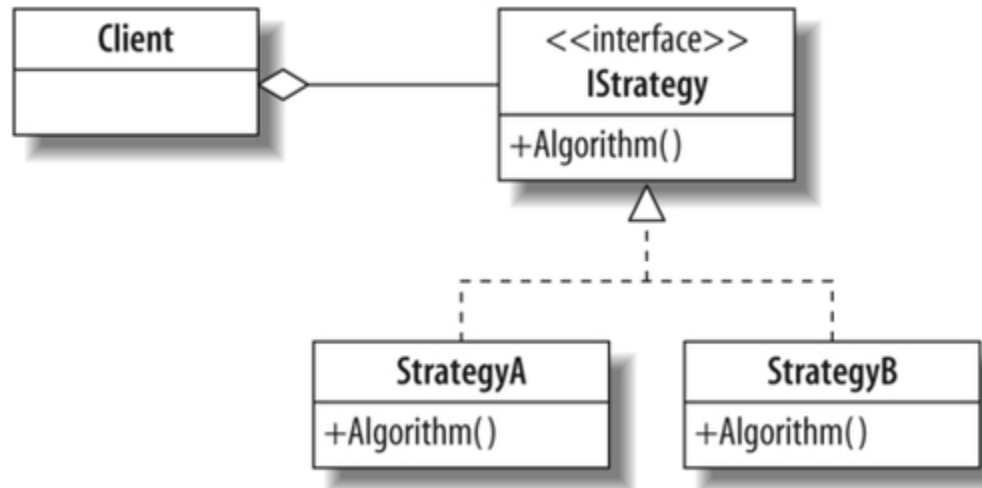
1) <b>IVisitor</b>	D
2) <b>VisitorA</b>	A
3) <b>ElementA, ElementB ...</b>	C
4) <b>Element</b>	B

a) <b>HtmlDisplayVisitor</b>
b) <b>HtmlElement</b>
c) <b>blockElement, textElement ...</b>
d) <b>HtmlVisitor</b>

# Strategy

- Účel

- Rozhoduje, který algoritmus se použije



```
internal class StrategySelector
{
    //References
    public IStrategy GetSortStrategy<T>(IList<T> list)
    {
        if (list.IsDescending())
        {
            return new ShellSort();
        }
        if (list.Count > 1000)
        {
            return new QuickSortStrategy();
        }
        if (list.Count > 1000000)
        {
            return new ParallelQuickSortStrategy();
        }
        return new MergeSortStrategy();
    }
}
```

# Strategy

## Příklad

Potřebujeme vybrat správný algoritmus pro seřazení kolekce.

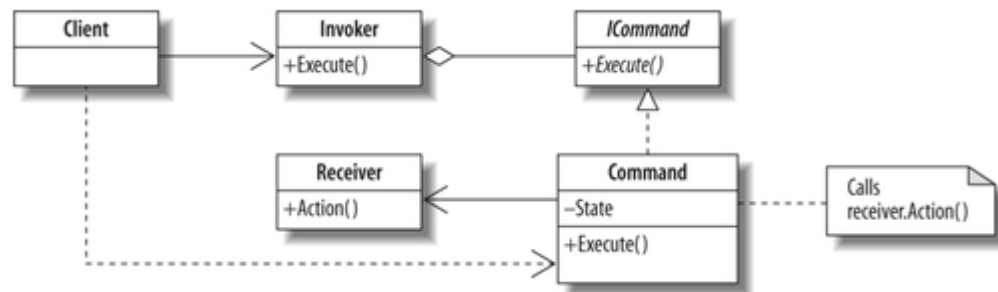
1) <b>IStrategy</b>	D
2) <b>StrategyA</b>	A    C
3) <b>StrategySelector</b>	B
4) <b>StrategyB</b>	A    C

a) QuickSort
b) GetSortStrategy
c) MergeSort
d) Řadící algoritmus

# Command

- Účel

- Vyčlenit metodu do vlastní třídy



```
internal interface ICommand
{
    bool CanExecute { get; }
    event EventHandler CanExecuteChanged;
    void Execute();
}

internal interface IUndoCommand : ICommand
{
    void ExecuteUndo();
}
```

- Výhody

- Paralelní zpracování
- Undo
- Unifikace logování

# Command

## Příklad

Textový editor má několik operací, které dokážou pracovat s textem (cut, copy, paste...). Na některé se dají aplikovat akce Undo, Redo. Operace se dají vyvolat různým způsobem (kliknutím, klávesovou zkratkou...)

1.) Client	C
2.) ICommand	F
3.) Command	E
4.) Receiver	A
5.) Invoker	D
6.) Action	B

a) Objekt, který může vykonat vyříznutí
b) Vyřezání
c) Uživatel editoru, který vybere příkaz vyřezání
d) Vykonání výběru akce „vyřezat“
e) Placeholder pro dotaz na vyřezání
f) Menu



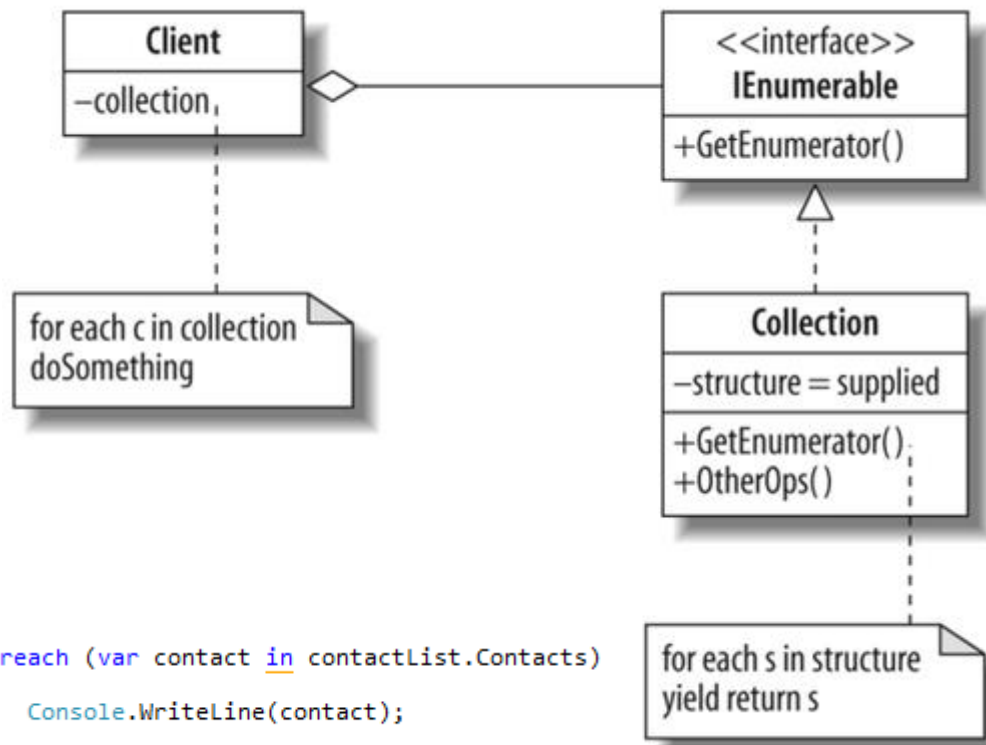
# Iterátor

- Účel

- Zefektivnění práce s velkou kolekcí
- Úspora paměti

```
private void ShowFirstFiveFibonacciNumber()
{
    int counter = 1;
    foreach (var number in this.GetFibonacciNumber())
    {
        if (counter > 5)
        {
            break;
        }
        Console.WriteLine(number);
        counter++;
    }
}
```

```
private IEnumerable<int> GetFibonacciNumber()
{
    int beforeNumber = 1;
    int actualNumber = 1;
    while (true)
    {
        int newNumber = beforeNumber + actualNumber;
        beforeNumber = actualNumber;
        actualNumber = newNumber;
        yield return newNumber;
    }
}
```



```
foreach (var contact in contactList.Contacts)
{
    Console.WriteLine(contact);
}
```

# Iterátor

## Příklad

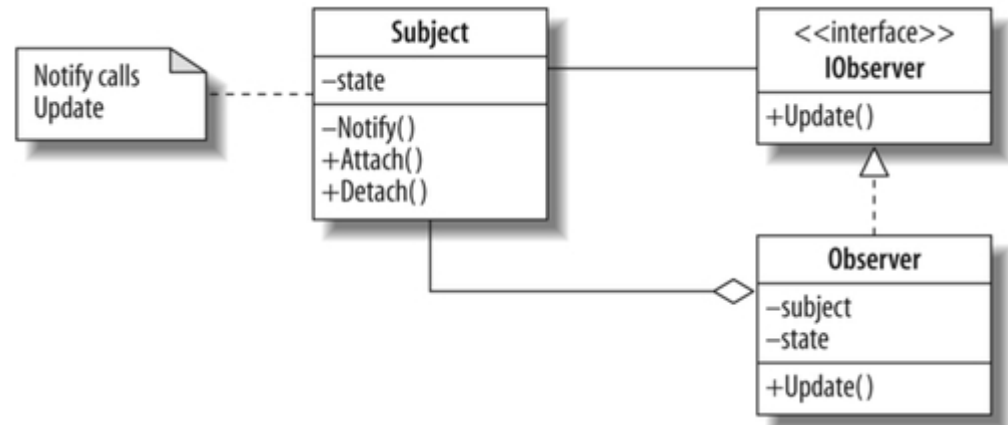
Složková struktura. Dá se vypsát hierarchicky do n-té úrovně. Dá se v ní vyhledávat dle klíče (datum vytvoření, název...), při hledání se hierarchie zanedbává. Dají se nad ní spouštět funkce jako zjištění velikosti složek, souborů...

1) <b>Collection</b>	B
2) <b>GetEnumerator</b>	C
3) <b>Client</b>	A
4) <b>OtherOps</b>	D

- a) Uživatel, který iteruje nad složkami souborů
- b) Struktura složek a souborů
- c) Vrátí 1 soubor nebo složku
- d) Vrátí 1 soubor nebo složku dle dalších kritérií

# Observer

- Účel



# Observer

## Příklad

Blogovací stránka. Blogger přidává nové články, uživatelé se můžou přihlásit na odběr článků a pak jim chodí notifikace emailem.

1.) Subject	C
2.) Observer	A
3.) IObserver	F
4.) Notify	B
5.) Update	D
6.) State	E

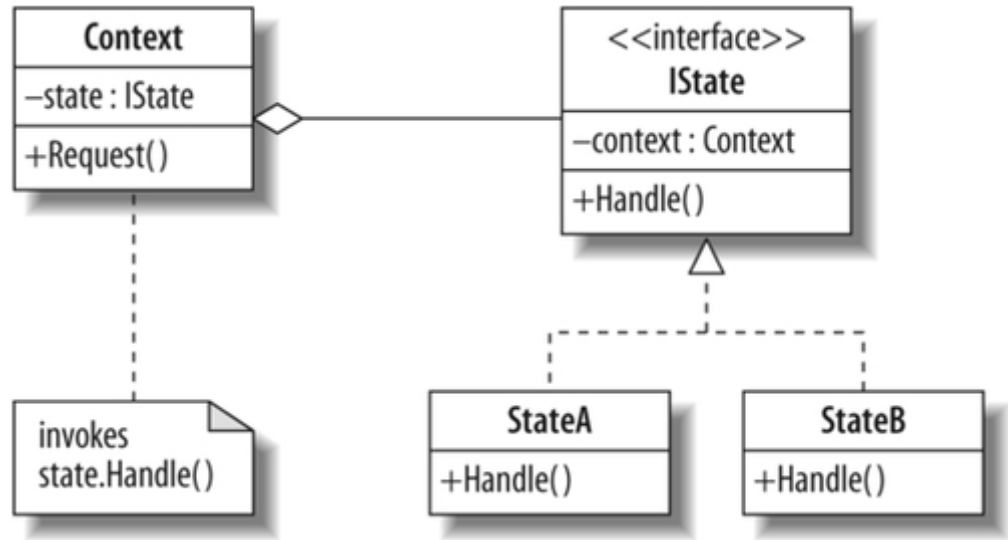
a) Programátor zajímavící se o C#
b) Email z blogovací stránky všem přihlášeným na odběr
c) Blogger, který píše o C#
d) Přijetí notifikace mailem
e) Blog
f) Emailová notifikace

# State

- **Účel**

dynamická verze Strategy

když se stav změní, může změnit  
chování přepnutím na jinou  
sadu operací



# State

## Příklad

Věrnostní program aerolinek. Klienti se dělí na různé typy účtů (stříbrný, zlatý...) dle počtu nalétaných km. Na základě typu účtu dostávají klienti výhody (km zdarma, rychlejší odbavení na letišti...)

1.) Context	B
2.) StateA, StateB, ...	E
3.) Request	C
4.) Handle	A
5.) IState	D

a) Konkrétní výhody aplikovány v jednotlivých typech účtů

b) Uživatelský účet

c) Seznam různých výhod, které se mohou měnit v závislosti na typu účtu

d) Aktivita v konkrétním typu účtu

e) Aktivita jako let přes několik km, získávání volných km

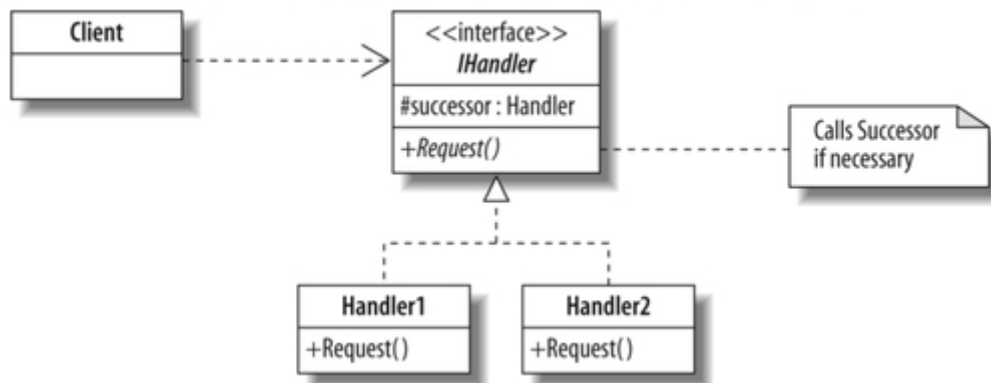
# Chain of responsibility

- Účel

handlery objektů

pokud objekt neumí zpracovat  
požadavek, pošle jej dále

pokud nikdo nezpracuje vykoná se buď defaultní chování, nebo výjimka



# Chain of responsibility

## Příklad

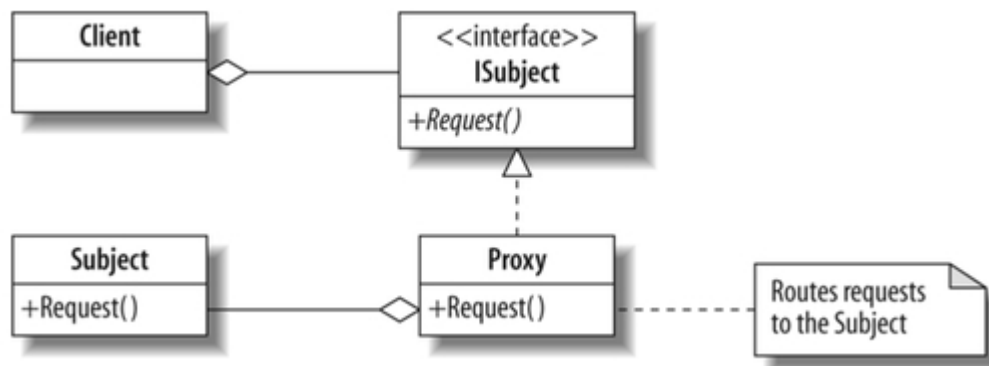
Banka. Úředníci jsou zodpovědní za jednoduché úkony, cokoli v zložitějšího posílají svým nadřazeným až po tu úroveň, která je za daný úkon zodpovědná. Na každé úrovni může být víc pracovníků, kteří ji zastávají.

1) Client	D
2) Handlers	B
3) Successor	A
4) Request	C

a) Další zaměstnanec v pořadí vedení
b) manažeři, zaměstnanci, úředníci
c) Zákazníkův požadavek
d) Zákazník



# Proxy



## Kdy použít

- Když objekty
  - Jsou náročné na vytvoření
  - Potřebují kontrolu nad přístupy
  - Komunikují po síti z dalšími PC
  - Potřebují vykonávat nějakou operaci při každém přístupu k nim
- Chci
  - Vytvořit objekt, až když je vyžadována jeho operace
  - Provádět kontrolu nebo údržbu objektů vždy, když jsou přístupné
  - Mít lokální objekt zastupující vzdálený objekt
  - Implementovat přístupová práva

## Příklady použití

- Odložení procesu vytváření složitého prostředí
- Ověření zda má odesílatel právo přístupu
- Posílání požadavků skrz síť
- Výkonání akcí na jiném "spřáteleném" webu
- Vykreslování obrázků. Proxy zabere místo kde bude obrázek vykreslen a teprve pak se začne vykreslovat obrázek

## Typy

- Proxy má několik druhů, nejsou zde uvedeny všechny
- **Virtuální proxy**
  - Předávání tvorby objektu jinému objektu (proxy)
  - Vhodné pokud je zakládání objektu pomalé a nejsou hned potřeba všechny jeho funkce nebo pokud se může jeho zakládání ukázat jako zcela zbytečné
- **Ověřovací proxy**
  - Ověřuje jestli má daný uživatel přístup k požadované operaci
- **Vzdálené proxy**
  - Kódují požadavky a posílají je po síti
- **Chytré proxy**
  - Přidávají nebo mění požadavky před jejich odesláním

# Proxy

## Příklad

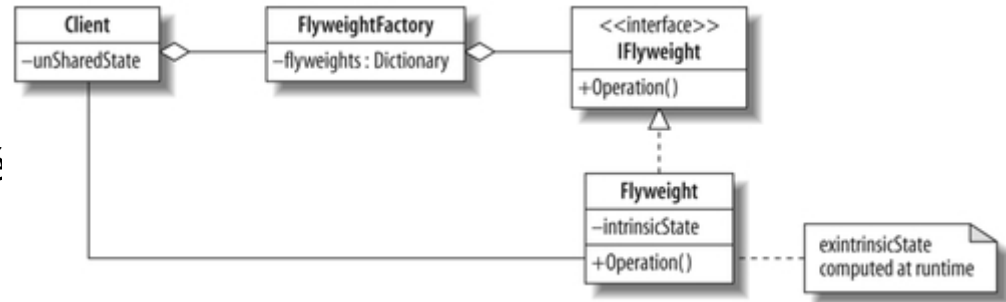
Sociální síť. Uživatelé mohou po registraci a přihlášení přistupovat k stránkám jiných uživatelů, mohou na nich vykonávat různé akce (přidání příspěvku, hodnocení, komentování...)

<b>1.) ISubject</b>	D	a) Fronted ke konkrétní stránce
<b>2.) Subject</b>	F	b) Návštěvník stránek
<b>3.) Subject.Request</b>	E	c) Vykoná nějakou akci před jejím předáním dále
<b>4.) Proxy</b>	A	d) Seznam akcí, které se dají udělat na stránce
<b>5.) Proxy.Request</b>	C	e) Změna stránky
<b>6.) Client</b>	B	f) Konkrétní stránka patřící jedné osobě

# Flyweight

- Účel

efektivní způsob sdílení společné informace pro velké množství objektů



rozděluje mezi vnitřním a vnějším stavem objektu:

vnitřní může být sdílen, minimalizuje nároky na paměť

vnější může být počítán za běhu

úspora paměti

# Flyweight

## Příklad

Fotogalerie. Fotografie se dají zobrazit v plné kvalitě samostatně nebo jako přehled skupiny se zmenšenými náhledy. Fotografie obsahují také informace o jejich zařazení do alb, které nejsou uživateli přímo přístupné.

1.) Client	C	a) Informace o skupině
2.) xFlyweight	E	b) Registrace unikátních obrázků
3.) FlyweightFactory	B	c) Aplikace skupiny fotek
4.) Flyweight	G	d) Náhled
5.) intrinsicState	D	e) Specifikace obrázku
6.) extrinsicState	F	f) Obrázek v plné kvalitě
7.) unSharedState	A	g) Tvůrce a vykreslovač náhledů

# Vytvářecí vzory

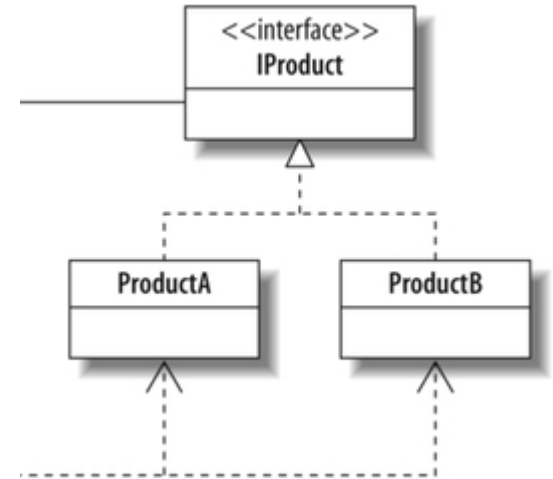
- **Hlavní účely**

- Starají se o vytvoření objektu
- Zajišťují inicializaci
- Rozhodují, která implementace se použije

# Factory method

- **Účel**

- Rozhoduje, která třída se bude instanciovat
- Také se používá pokud je potřeba větší logika k vytvoření
- Často je asynchronní

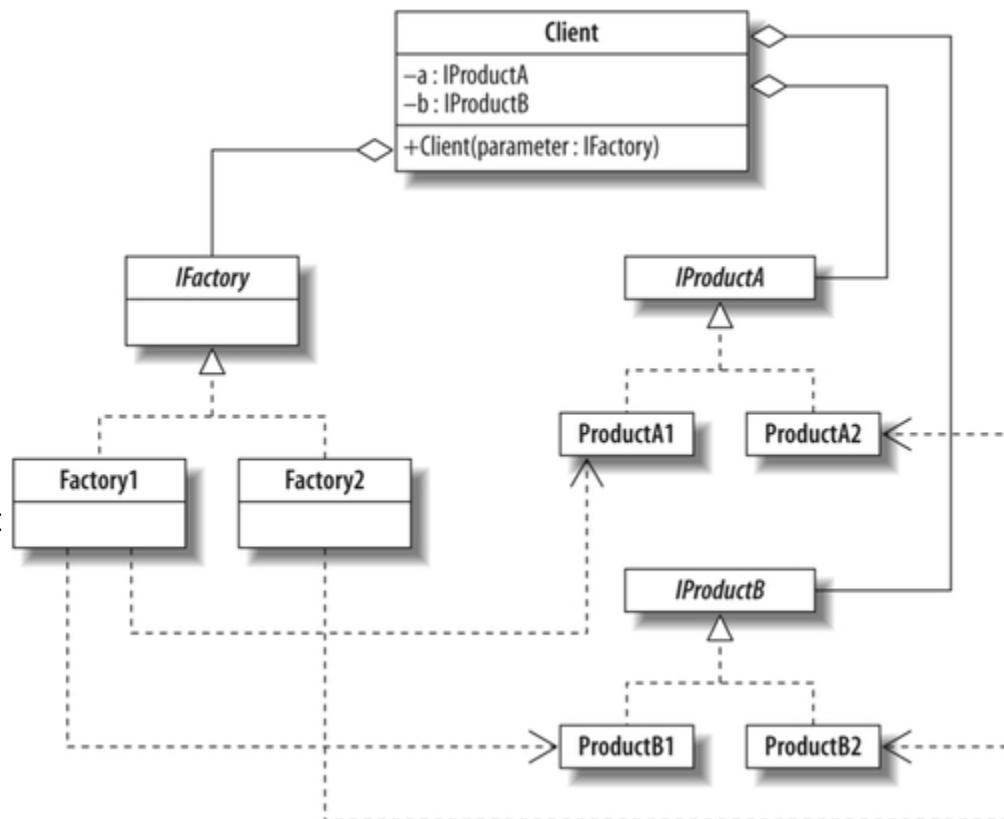


```
var id = Guid.NewGuid();
```

# Abstract factory

- Účel

- „Je to factory method na steroidech“
- Rozhoduje, která factory se bude instanciovat



# Fluent API

- Účel

- Zvýšit čitelnost kódu
  - Bohužel ne vždy

```
public HtmlElement AddElement(HtmlElement element)
{
    this.innerElements.Add(element);
    return this;
}
```

```
var silverBmwOwners = cars.OfType<BMW>().Where(car => car.Color == Color.Silver).Select(car => car.Owner)
    .OrderBy(owner => owner.Lastname).ThenBy(owner => owner.Firstname);
```

```
Div main = new Div();
main.AddAttribute(new HtmlAttribute("id", "Main")).AddClass("container")
    .AddElement(new H1("Title").AddClass("title"))
    .AddElement(
        new Div().AddClass("Menu")
            .AddElement(new Div().AddClass("menu-item").AddElement(new Span("Menu item 1")))
            .AddElement(new Div().AddClass("menu-item").AddElement(new Span("Menu item 2")))
            .AddElement(new Div().AddClass("menu-item").AddElement(new Span("Menu item 2")))
    )
;
```

```
this.Container.Register(Component.For<IMailerService>().ImplementedBy<MailerService>()
    .LifestyleTransient().Interceptors<LoggingInterceptor>());
```




# Enterprise vzory



# Domain Model

- Business data a business operace reprezentována pomocí objektů a jejich metod
- Nemusí být 1:1 se schématem databáze
  - Ani by neměl být
  - Návrh aplikace by měl začít právě návrhem domain modelu
  - Praxe je často jiná
- Alternativy
  - Data v SQL databázi, práce s nimi např. přes stored procedury, v aplikaci není nutná třída pro reprezentaci záznamu z databáze, používá se např. DataSet, DataTable
    - Dnes již překonané


# Data Mapper

- Mapování dat ze SQL databáze (případně jiné) na objekty doménového modelu
  - Typicky řeší i vazby mezi tabulkami
    - Lazy loading, eager loading
    - Dědičnost
      - Table per class
      - Table per type
      - Table per hierarchy
- 

# Identity Map

- Řeší konkurenční přístup k datům v rámci jedné transakce
  - Např. načteme řádek z tabulky Order s Id=1
  - Upravíme na něm nějakou vlastnost
  - Řádek načteme znovu (třeba jako součást jiného dotazu)
  - Která verze platí?
- Identity Map v rámci každé transakce eviduje všechny entity (podle primárního klíče)
  - Pokud již entita existuje, není materializována znovu, ale použije se existující instance


# Unit of Work

- Ohraničuje business transakci
  - Typicky propojená s Identity Map
  - Eviduje změněné objekty a umožňuje promítnout jejich změny do datového úložiště
    - Řeší i pořadí updatů
  - Často používá IDisposable a vzor Registry
    - Vytvoření instance zaregistruje Unit of Work do aktuálního vlákna
    - Podpora vnořování UOW – použijeme Stack
    - Dispose Unit of Work odregistruje
- 

# Repository

- Umožňuje CRUD operace nad kolekcí objektů
  - Např. databázovou tabulkou
- Add, Update, Delete, FindById
- Někdo umožňuje i vracení IQueryable (GetAll)
  - V praxi není úplně vhodné
    - Svádí to k psaní složitých dotazů přímo přes repozitář
    - Jednoduché dotazy se v aplikaci často opakují
  - Lepší je použít Query objekt

# Query Object

- Dotaz nad daty reprezentován pomocí třídy
  - Metoda Execute
  - Další možnosti
    - Parametrizovatelnost dotazů
    - Obecná podpora stránkování, řazení, filtrování
    - Možnost přidat např. metodu GetTotalCount • Post-processing výsledků
- 

# IoC/DI

- Otázka: Jak tyto krabičky propojit?

```
public interface INewsletterService
{
    void SendNewsletters(string customerGroup);
}
```

```
public interface IMailerService
{
    void SendMail(string to, string subject, string body);
}
```

