Rastislav Lencses

3. 5. 2005

Architektúra informačných systémov

Štrukturálne programovanie

- programy pred érou OOP: štrukturálne (procedurálne) programovanie, funkčná dekompozícia
- problém sa rozdelil na menšie celky atď, pre ktoré boli napísané funkcie
- problém s veľkou zodpovednosťou hlavného programu
- problematické zapracovanie zmien
- požiadavky typicky sú: neúplné, chybné, zavádzajúce
- zmena: buď sa rieši doplnením pôvodnej funkcie, alebo oddelením zmeny do novej funkcie volanej pomocou if, switch
- problémy so súdržnosťou (cohesion): ako blízko súvisia operácie v jednom bloku (rutine)
- problémy s väzbou (coupling): sila spojenia medzi dvoma blokmi (rutinami)
- problémy s bočnými efektami (side effect): v čase údržby (maintenance) sa väčšina času strávi odhaľovaním bočných efektov funkcií; oprava chyby je relatívne ľahká
- cieľ je vytvoriť bloky s veľkou internou integritou (silnou súdržnosťou) a malou, viditeľnou a flexibilnou väzbou na iné bloky (slabá väzba)

Pojmy OO terminológie

- abstraktná trieda (čo môžu robiť skupiny súvisiacich tried)
- trieda (zodpovednosť za správanie a/alebo stav, čo je riešené metódami a atribútmi)
- konkrétna trieda (implementuje konkrétne nemenné správanie odvodené od abstraktnej triedy)
- zapuzdrenie (dát, dizajnu, implementácie)
- dedičnosť
- inštancia (objekt; konkrétny príklad triedy)
- interface (špecifikácia bez implementácie)
- polymorfizmus (rôzne správanie sa odvodených triedy, pričom sa naň dá u nadradenej triedy odkazovať jednotným spôsobom)
- atribút, metóda, člen, konštruktor, deštruktor, nadtrieda, rodič

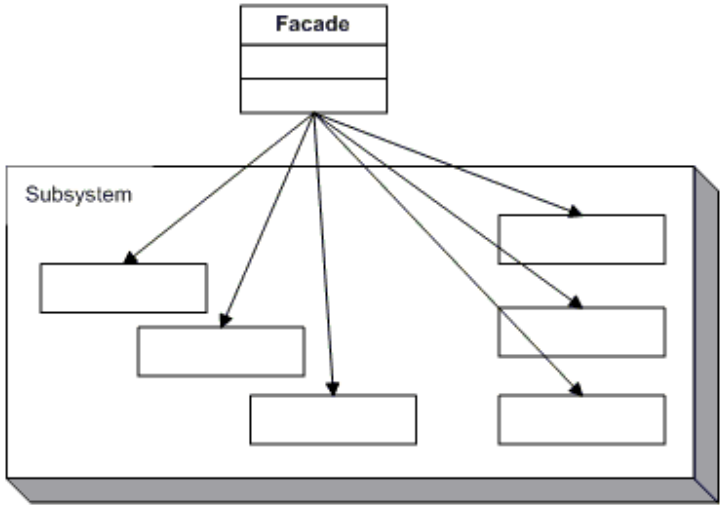
Návrhový vzor

- rieši opakujúce sa problémy vytvorením špecifického riešenia, ktoré môže byť po identifikovaní situácie použité s minimálnym úsilím
- skladá sa z: meno, cieľ, problém, riešenie, podieľajúce a spolupracujúce entity, dôsledky, implementácia, všeobecná štruktúra
- umožňuje: znovupoužitie myšlienky, zavedenie všeobecnej terminológie

- dobrý objektovo-orientovaný dizajn (podľa GoF): návrh využívajúci rozhrania (interface), uprednostňovať agregáciu pred dedičnosťou, nájsť variabilitu a obaliť ju
- tri druhy vzorov: tvorba, štruktúra, správanie

Façade (priečelie)

- poskytuje jednotné rozhranie množine rozhraní subsystému
- definuje rozhranie vyššej úrovne, ktoré vytvorí abstrakciu subsystému
- štrukturálny vzor



- cieľ: zjednodušiť použitie existujúceho systému, prípadne potrebujeme definovať vlastné rozhranie
- problém: chceme použiť len časť komplexného systému, prípadne interafovať so systémom vlastným spôsobom
- riešenie: nové rozhranie pre klienta využívajúceho systém
- podieľajúce sa entity: *façade* a *subsystém* so svojimi triedami
- *façade*: vie, ktorý objekt *subsystému* je zodpovedný za splnenie požiadavky a deleguje naň požiadavky *klienta*, môže byť stavový (vie rozlíšiť medzi dvoma rovnakými požiadavkami *klienta*), môže byť bezstavový (vie obsluhovať viac *klientov*)
- *subsystém*: implementuje svoju funkcionalitu a obsluhuje požiadavky *façade* objektu, pričom o ňom nevie (a nemá naň žiaden odkaz)
- následky: façade objekt zjednodušuje použitie subsystému, pričom však nemusí byť použiteľná celá funkcionalita subsystému
- implementácia: definovať triedu (alebo viac), ktorá požadované rozhranie
- façade objekt môže byť použitý ku zapuzdreniu systému – potom je možné monitorovať použitie subsystému, prípadne vymeniť subsystém bez problémov na strane klienta tohto subsystému

```
// Facade pattern -- Structural example
using System;
// "SubSystem"
```

```
class SubSystemOne
{
  public void MethodOne()
  {
    Console.WriteLine("SubSystemOne Method");
  }
}

class SubSystemTwo
{
  public void MethodTwo()
  {
    Console.WriteLine("SubSystemTwo Method");
  }
}

class SubSystemThree
{
  public void MethodThree()
  {
    Console.WriteLine("SubSystemThree Method");
  }
}

class SubSystemFour
{
  public void MethodFour()
  {
    Console.WriteLine("SubSystemFour Method");
  }
}

// "Facade"
class Facade
{
  // Fields
  SubSystemOne one;
  SubSystemTwo two;
  SubSystemThree three;
  SubSystemFour four;

  // Constructors
  public Facade()
  {
    one = new SubSystemOne();
    two = new SubSystemTwo();
    three = new SubSystemThree();
    four = new SubSystemFour();
  }

  // Methods
  public void MethodA()
  {
    Console.WriteLine( "\nMethodA() ---- " );
    one.MethodOne();
    two.MethodTwo();
    four.MethodFour();
```

```
  }

  public void MethodB()
  {
    Console.WriteLine( "\nMethodB() ---- " );
    two.MethodTwo();
    three.MethodThree();
  }
}

/// <summary>
/// Client test
/// </summary>
public class Client
{
  public static void Main(string[] args)
  {
    // Create and call Facade
    Facade f = new Facade();
    f.MethodA();
    f.MethodB();
  }
}
```

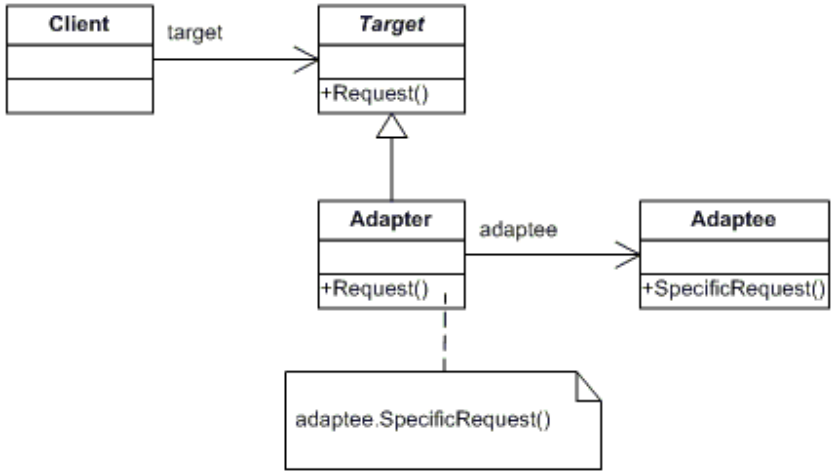
```
// Facade pattern -- Real World example

using System;

// "SubSystem ClassA"

class Bank
{
  // Methods
  public bool SufficientSavings( Customer c )
  {
    Console.WriteLine("Check bank for {0}", c.Name );
    return true;
  }
}

// "SubSystem ClassB"

class Credit
{
  // Methods
  public bool GoodCredit( int amount, Customer c )
  {
    Console.WriteLine( "Check credit for {0}", c.Name );
    return true;
  }
}

// "SubSystem ClassC"

class Loan
```

```
{
  // Methods
  public bool GoodLoan( Customer c )
  {
    Console.WriteLine( "Check loan for {0}", c.Name );
    return true;
  }
}

class Customer
{
  // Fields
  private string name;

  // Constructors
  public Customer( string name )
  {
    this.name = name;
  }

  // Properties
  public string Name
  {
    get{ return name; }
  }
}

// "Facade"

class MortgageApplication
{
  // Fields
  int amount;
  private Bank bank = new Bank();
  private Loan loan = new Loan();
  private Credit credit = new Credit();

  // Constructors
  public MortgageApplication( int amount )
  {
    this.amount = amount;
  }

  // Methods
  public bool IsEligible( Customer c )
  {
    // Check creditworthyness of applicant
    if( !bank.SufficientSavings( c ) )
      return false;
    if( !loan.GoodLoan( c ) )
      return false;
    if( !credit.GoodCredit( amount, c ))
      return false;

    return true;
  }
}
```

```
/// <summary>
/// Facade Client
/// </summary>
public class FacadeApp
{
  public static void Main(string[] args)
  {
    // Create Facade
    MortgageApplication mortgage =
                new MortgageApplication( 125000 );
    // Call subsystem through Facade
    mortgage.IsEligible(
              new Customer( "Gabrielle McKinsey" ) );

  }
}
```

Adaptér

- konvertuje rozhranie triedy na iné rozhranie, ktoré očakáva klient
- umožňuje spolupracovať takým triedam, pre ktoré by to nebolo inak možné kvôli nekompatibilným rozhraniam
- štrukturálny vzor



- cieľ: napasovať objekt mimo našej kontroly na špecifické rozhranie
- problém: systém má správne dáta a správanie avšak nevhodné rozhranie
- riešenie: adaptér je wrapper s požadovaným rozhraním
- podieľajúce sa entity: adapter, adaptee, target, client
- *adapter*: mení rozhranie *adaptee* na rozhranie *target*; potom môže *client* používať *adaptee* tak akoby to bol typu *target*
- *adaptee*: definuje rozhranie, ktoré je treba zmeniť
- *target*: definuje rozhranie, ktoré môže používať *client*
- *client*: spolupracuje s objektami, ktoré majú rozhranie podľa *target*
- následky: adaptér umožňuje existujúcim objektom zapojiť sa do štruktúry iných tried bez obmedzenia ich rozhraní
- implementácia: existujúcu triedu vložiť do inej (do adaptéru). Pre adaptér treba definovať požadované rozhranie a v metódach tohto rozhrania volať metódu existujúcej triedy

```
// Adapter pattern -- Structural example
using System;
// "Target"
class Target
{
  // Methods
  virtual public void Request()
  {
```

```
    // Normal implementation goes here
  }
}

// "Adapter"
class Adapter : Target
{
  // Fields
  private Adaptee adaptee = new Adaptee();

  // Methods
  override public void Request()
  {
    // Possibly do some data manipulation
    // and then call SpecificRequest
    adaptee.SpecificRequest();
  }
}

// "Adaptee"
class Adaptee
{
  // Methods
  public void SpecificRequest()
  {
    Console.WriteLine("Called SpecificRequest()" );
  }
}

/// <summary>
/// Client test
/// </summary>
public class Client
{
  public static void Main(string[] args)
  {
    // Create adapter and place a request
    Target t = new Adapter();
    t.Request();

  }
}
```

```
// Adapter pattern -- Real World example

using System;

// "Target"

class ChemicalCompound
{
  // Fields
  protected string name;
  protected float boilingPoint;
  protected float meltingPoint;
```

```
  protected double molecularWeight;
  protected string molecularFormula;

  // Constructors
  public ChemicalCompound( string name )
  {
    this.name = name;
  }

  // Properties
  public float BoilingPoint
  {
    get{ return boilingPoint; }
  }

  public float MeltingPoint
  {
    get{ return meltingPoint; }
  }

  public double MolecularWeight
  {
    get{ return molecularWeight; }
  }

  public string MolecularFormula
  {
    get{ return molecularFormula; }
  }
}

// "Adapter"

class Compound : ChemicalCompound
{
  // Fields
  private ChemicalDatabank bank;

  // Constructors
  public Compound( string name ) : base( name )
  {
    // Adaptee
    bank = new ChemicalDatabank();
    // Adaptee request methods
    boilingPoint = bank.GetCriticalPoint( name, "B" );
    meltingPoint = bank.GetCriticalPoint( name, "M" );
    molecularWeight = bank.GetMolecularWeight( name );
    molecularFormula = bank.GetMolecularStructure( name );
  }

  // Methods
  public void Display()
  {
    Console.WriteLine("\nCompound: {0} ------ ",name );
    Console.WriteLine(" Formula: {0}",MolecularFormula);
    Console.WriteLine(" Weight : {0}",MolecularWeight );
    Console.WriteLine(" Melting Pt: {0}",MeltingPoint );
```

```
    Console.WriteLine(" Boiling Pt: {0}",BoilingPoint );
  }
}

// "Adaptee"

class ChemicalDatabank
{
  // Methods -- the Databank 'legacy API'
  public float GetCriticalPoint( string compound, string point )
  {
    float temperature = 0.0F;
    // Melting Point
    if( point == "M" )
    {
      switch( compound.ToLower() )
      {
        case "water": temperature = 0.0F; break;
        case "benzene" : temperature = 5.5F; break;
        case "alcohol": temperature = -114.1F; break;
      }
    }
    // Boiling Point
    else
    {
      switch( compound.ToLower() )
      {
        case "water": temperature = 100.0F;break;
        case "benzene" : temperature = 80.1F; break;
        case "alcohol": temperature = 78.3F; break;
      }
    }
    return temperature;
  }

  public string GetMolecularStructure( string compound )
  {
    string structure = "";
    switch( compound.ToLower() )
    {
      case "water": structure = "H20"; break;
      case "benzene" : structure = "C6H6"; break;
      case "alcohol": structure = "C2H6O2"; break;
    }
    return structure;
  }

  public double GetMolecularWeight( string compound )
  {
    double weight = 0.0;
    switch( compound.ToLower() )
    {
      case "water": weight = 18.015; break;
      case "benzene" : weight = 78.1134; break;
      case "alcohol": weight = 46.0688; break;
    }
    return weight;
```

```
  }
}

/// <summary>
/// AdapterApp test application
/// </summary>
public class AdapterApp
{
  public static void Main(string[] args)
  {
    // Retrieve and display water characteristics
    Compound water = new Compound( "Water" );
    water.Display();

    // Retrieve and display benzene characteristics
    Compound benzene = new Compound( "Benzene" );
    benzene.Display();

    // Retrieve and display alcohol characteristics
    Compound alcohol = new Compound( "Alcohol" );
    alcohol.Display();

  }
}
```

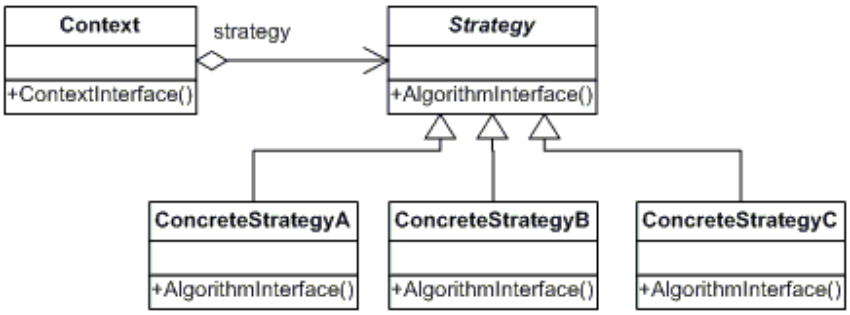
Adaptér versus façade

- oba sú wrappery (obaľovače)

	Adapter	**Façade**
Sú tam existujúce triedy ?	Áno	Áno
Musíme navrhnúť rozhranie ?	Áno	Nie
Musí sa objekt správať polymorfne ?	Pravdepodobne	Nie
Je potrebné jednoduchšie rozhranie ?	Nie	Áno
Jednou vetou	Konvertuje rozhranie	zjednodušuje rozhranie

Strategy

- umožní použiť rôzne biznis pravidlá alebo algoritmy (patriace do jednej rodiny) v závislosti na kontexte v ktorom sa vyskytnú
- je to odlišné od vytvárania potomkov kontextu
- vzor týkajúci sa správania



- cieľ: umožniť využiť rôzne biznis pravidlá alebo algoritmy v závislosti na kontexte
- problém: výber algoritmu, ktorý treba vykonať, závisí na klientovi, zasiela požiadavku. Tento algoritmus podlieha zmene.
- riešenie: oddeliť výber algoritmu od jeho implementácie a umožniť jeho výber v závislosti na kontexte
- podieľajúce sa entity: strategy, concreteStrategies, context
- *strategy*: deklaruje rozhranie spoločné všetkým algoritmom
- *concreteStrategies*: implementujú tieto rôzne algoritmy
- *context*: používa špecifickú konkrétnu stratégiu, udržiava si na ňu odkaz, môže mať rozhranie, ktoré bude využívať konkrétna stratégia
- následky: vzor stratégia definuje rodinu algoritmov, vyhne sa použitiu switch/if, vyvolanie všetkých algoritmov je nutné urobiť rovnakým spôsobom
- implementácia: trieda context bude obsahovať abstraktnú stratégiu, pre ktoré budú existovať jej potomkovia. Za výber konkrétnej stratégie zodpovedá klient, ktorý ju pre context zvolí

```
// Strategy pattern -- Structural example

using System;

// "Strategy"

abstract class Strategy
{
  // Methods
  abstract public void AlgorithmInterface();
}
```

```
// "ConcreteStrategyA"

class ConcreteStrategyA : Strategy
{
  // Methods
  override public void AlgorithmInterface()
  {
  Console.WriteLine(
      "Called ConcreteStrategyA.AlgorithmInterface()");
  }
}

// "ConcreteStrategyB"

class ConcreteStrategyB : Strategy
{
  // Methods
  override public void AlgorithmInterface()
  {
    Console.WriteLine(
        "Called ConcreteStrategyB.AlgorithmInterface()");
  }
}

// "ConcreteStrategyC"

class ConcreteStrategyC : Strategy
{
  // Methods
  override public void AlgorithmInterface()
  {
    Console.WriteLine(
        "Called ConcreteStrategyC.AlgorithmInterface()");
  }
}

// "Context"

class Context
{
  // Fields
  Strategy strategy;

  // Constructors
  public Context( Strategy strategy )
  {
    this.strategy = strategy;
  }

  // Methods
  public void ContextInterface()
  {
    strategy.AlgorithmInterface();
  }
}

/// <summary>
```

```
/// Client test
/// </summary>
public class Client
{
  public static void Main( string[] args )
  {
    // Three contexts following different strategies
    Context c = new Context( new ConcreteStrategyA() );
    c.ContextInterface();

    Context d = new Context( new ConcreteStrategyB() );
    d.ContextInterface();

    Context e = new Context( new ConcreteStrategyC() );
     e.ContextInterface();
  }
}
```

```
// Strategy pattern -- Real World example

using System;
using System.Collections;

// "Strategy"

abstract class SortStrategy
{
  // Methods
  abstract public void Sort( ArrayList list );
}

// "ConcreteStrategy"

class QuickSort : SortStrategy
{
  // Methods
  public override void Sort(ArrayList list )
  {
    list.Sort(); // Default is Quicksort
    Console.WriteLine("QuickSorted list ");
  }
}

// "ConcreteStrategy"

class ShellSort : SortStrategy
{
  // Methods
  public override void Sort(ArrayList list )
  {
    //list.ShellSort();
    Console.WriteLine("ShellSorted list ");
  }
}
```

```
// "ConcreteStrategy"

class MergeSort : SortStrategy
{
  // Methods
  public override void Sort( ArrayList list )
  {
    //list.MergeSort();
    Console.WriteLine("MergeSorted list ");
  }
}

// "Context"

class SortedList
{
  // Fields
  private ArrayList list = new ArrayList();
  private SortStrategy sortstrategy;

  // Constructors
  public void SetSortStrategy( SortStrategy sortstrategy )
  {
    this.sortstrategy = sortstrategy;
  }

  // Methods
  public void Sort()
  {
    sortstrategy.Sort( list );
  }

  public void Add( string name )
  {
    list.Add( name );
  }

  public void Display()
  {
    foreach( string name in list )
      Console.WriteLine( " " + name );
  }
}

/// <summary>
/// StrategyApp test
/// </summary>
public class StrategyApp
{
  public static void Main( string[] args )
  {
    // Two contexts following different strategies
    SortedList studentRecords = new SortedList( );
    studentRecords.Add( "Samual" );
    studentRecords.Add( "Jimmy" );
    studentRecords.Add( "Sandra" );
    studentRecords.Add( "Anna" );
```

```
    studentRecords.Add( "Vivek" );

    studentRecords.SetSortStrategy( new QuickSort() );
    studentRecords.Sort();
    studentRecords.Display();
  }
}
```

O OO návrhu

objekty (triedy)

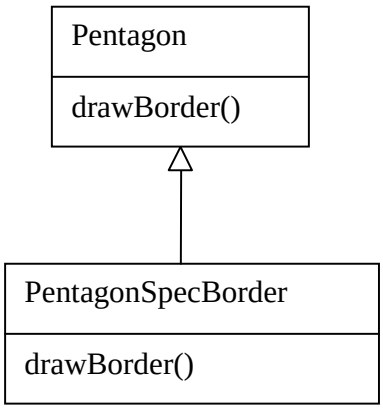
- tradičný pohľad na objekty: dáta (atribúty) popisujúce doménu a metódy umožňujúce s nimi manipulovať
- nový pohľad: objekty sú entity majúce zodpovednosť (za znalosti uchovávané v atribútoch a za správanie definované v metódach)
- objekt Shape: chceme vedieť, kde je, chceme, aby bol schopný sa nakresliť na nejaké plátno a vymazať sa
- riešenie: nadefinujeme rozhranie (interface) zložené z metód: getLocation(...), draw(...) a hide (...), pričom sa nestaráme, ako to je vyriešené vovnútri (čím si uchováme možnosť pružnej zmeny)

zapuzdrenie

- tradičný pohľad je, že ide o ukrytie (encapsulation) atribútov, prípadne metód (klientska trieda o nich nevie a teda ich možno pružne meniť bez ďalších následkov na klientskej triede)
- ukrývať však možno aj agregované objekty a typy (rodič nevie, že existujú jeho potomkovia), čím dosiahneme pružnosť pri prípadnej zmene

dedičnosť versus „nájdi variabilitu a obaľ (zapuzdri) ju“

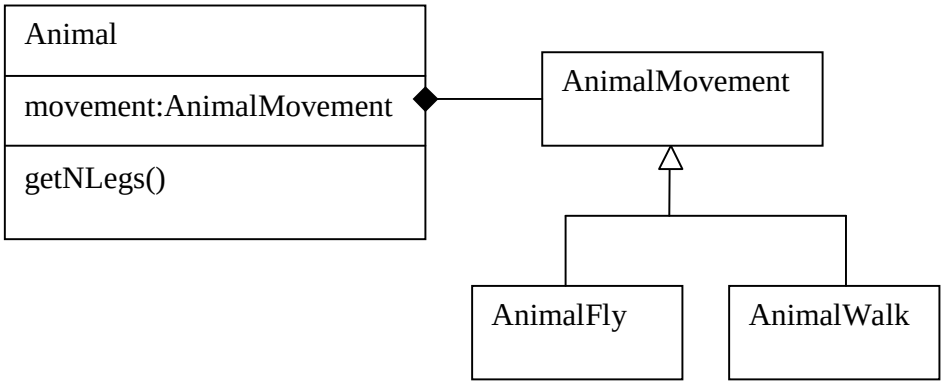
- tradičný pohľad je, že dedičnosťou možno zabezpečiť opätovné znovupoužitie tried (s tým, že odvodíme novú triedu, ktorá bude mať pozmenené potrebné metódy)



- nevýhody:
  - slabá súdržnosť: čo v prípade viacerých typov hraníc – potomkovia triedy Pentagon sa už nestarajú iba o kreslenie nejakého Pentagonu, ale aj o jeho hranicu
  - zníženie znovupoužiteľnosti: potomkovia (s špec. hranicou) sa už len ťažko dajú použiť v inom kontexte

- slabá škálovateľnosť: čo ak treba meniť ešte niečo iné (tieňovanie, ...) – vtedy treba písať celú hierarchiu obsahujúcu všetky kombinácie potomkov

- nový pohľad: nájsť, čo sa mení a nahradiť to novou (abstraktnou) triedou, ktorá bude agregovaná ku doménovej triede
- úloha: Zvieratá môžu mať rôzny počet nôh a trieda si to musí vedieť uchovať a sprístupniť. Zvieratá môžu mať rôzny (alebo rôzne) spôsob pohybu a trieda musí vedieť vypočítať čas, za ktorý zviera prekoná danú vzdialenosť na danom povrchu



agile coding, eXtreme Programming

- techniky programovania (a návrhu) založené na vývoji pomocou malých kontrolovaných krokoch, podstatný je kód
- neredundantnosť: jedno pravidlo na jednom mieste (súvislosť s väzbou - coupling)
- čitateľnosť: triedy sú napísané tak, že ich možno ľahko pochopiť (súvislosť so súdržnosťou - cohesion)
- testovateľnosť

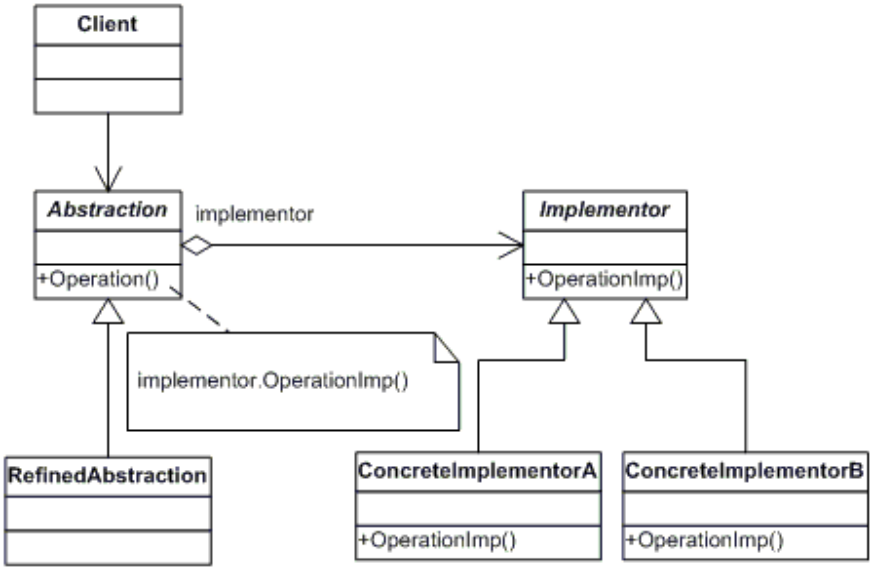
```
// Tests -- Real World example

public class Mathematics {
  public static double square(double x) {
     return x*x;
  }
}

class TestMathematics {
  public static void testSquare() throws TestException {
     for (int i=0; i<100; i++) {
        double testNumber = Math.random()*1000;
        if (Mathematics.square(testNumber)!= testNumber*testNumber) {
            // error
            throw new TestException(testNumber);
        }
     }
  }
```

```
}

class TestException extends Exception { …
}
```

- jeden zo základných princípov XP je napísať test metódy (triedy) pred jej implementáciou
- dosiahne sa tým automatizácia testov, návrh je prinútený byť orientovaný na rozhrania, rozdelenie funkcionality na testovateľné kusy kódu (dobrá súdržnosť a väzba)
- súdržný kód sa testuje ľahšie (testuje sa len jedna vec)
- slabo viazaný kód sa testuje ľahšie kvôli minimálnym interakciám, ktoré je treba uvažovať
- redundantný kód sa testuje ťažšie (treba napísať viac testov)
- čitateľný kód sa testuje ľahšie (mená metód, atribútov a parametrov sú popisné a teda vieme, čo majú vykonávať)
- test driven development

Bridge

- oddelí abstrakciu od jej implementácie, aby sa obe mohli nezávisle meniť
- štrukturálny vzor



- cieľ: oddelenie množiny implementácií od množiny objektov, ktoré ichpoužívajú
- problém: potomkovia abstraktnej triedy musia používať viaceré implementácie bez zbytočného nárastu počtu tried
- riešenie: definovať rozhranie pre všetky implementácie a mať potomkov abstraktnej triedy, ktoré ho využívajú
- podieľajúce sa entity: Abstraction, Implementor, redefinedAbstraction, ConcreteImplementor
- *abstraction*: definuje rozhranie pre implementované objekty
- *implementor*: definuje rozhranie pre špecifickú implementáciu. Zvyčajne je úplne odlišné od rozhrania *abstraction* (typicky *implementor* definuje nejaké nízkoúrovňové operácie a *abstraction* ich definuje na vyššej úrovni, pričom využíva nízkoúrovňové).
- *redefinedAbstraction*: potomok *abstraction*, ktorý používa nejakého potomka triedy *implementor*, pričom predbežne nevie (nie je podstatné, určí sa to neskôr), ktorého
- následky: oddelenie implementácie od objektov, ktoré ich používajú zvýši rozširovateľnosť návrhu. Klientske triedy sa nezaujímajú o implementačné problémy.
- implementácia: (abstraktná) implementácia sa zapuzdrí do abstraktnej triedy, ktorá obsahuje metódu závislú na metóde z implementácie. Potomkovia abstraktnej triedy tak využívajú potomkov implementačnej triedy

```
// Bridge pattern -- Structural example

using System;

// "Abstraction"

class Abstraction
{
  // Fields
  protected Implementor implementor;

  // Properties
  public Implementor Implementor
  {
    set{ implementor = value; }
  }

  // Methods
  virtual public void Operation()
  {
    implementor.Operation();
  }
}

// "Implementor"

abstract class Implementor
{
  // Methods
  abstract public void Operation();
}

// "RefinedAbstraction"

class RefinedAbstraction : Abstraction
{
  // Methods
  override public void Operation()
  {
    implementor.Operation();
  }
}

// "ConcreteImplementorA"

class ConcreteImplementorA : Implementor
{
  // Methods
  override public void Operation()
  {
    Console.WriteLine("ConcreteImplementorA Operation");
  }
```

```
}

// "ConcreteImplementorB"

class ConcreteImplementorB : Implementor
{
  // Methods
  override public void Operation()
  {
    Console.WriteLine("ConcreteImplementorB Operation");
  }
}

/// <summary>
/// Client test
/// </summary>
public class Client
{
  public static void Main( string[] args )
  {
    Abstraction abstraction = new RefinedAbstraction();

    // Set implementation and call
    abstraction.Implementor = new ConcreteImplementorA();
    abstraction.Operation();

    // Change implemention and call
    abstraction.Implementor = new ConcreteImplementorB();
    abstraction.Operation();
  }
}
```

```
// Bridge pattern -- Real World example

using System;
using System.Collections;

// "Abstraction"

class BusinessObject
{
  // Fields
  private DataObject dataObject;
  protected string group;

  // Constructors
  public BusinessObject( string group )
  {
    this.group = group;
  }
```

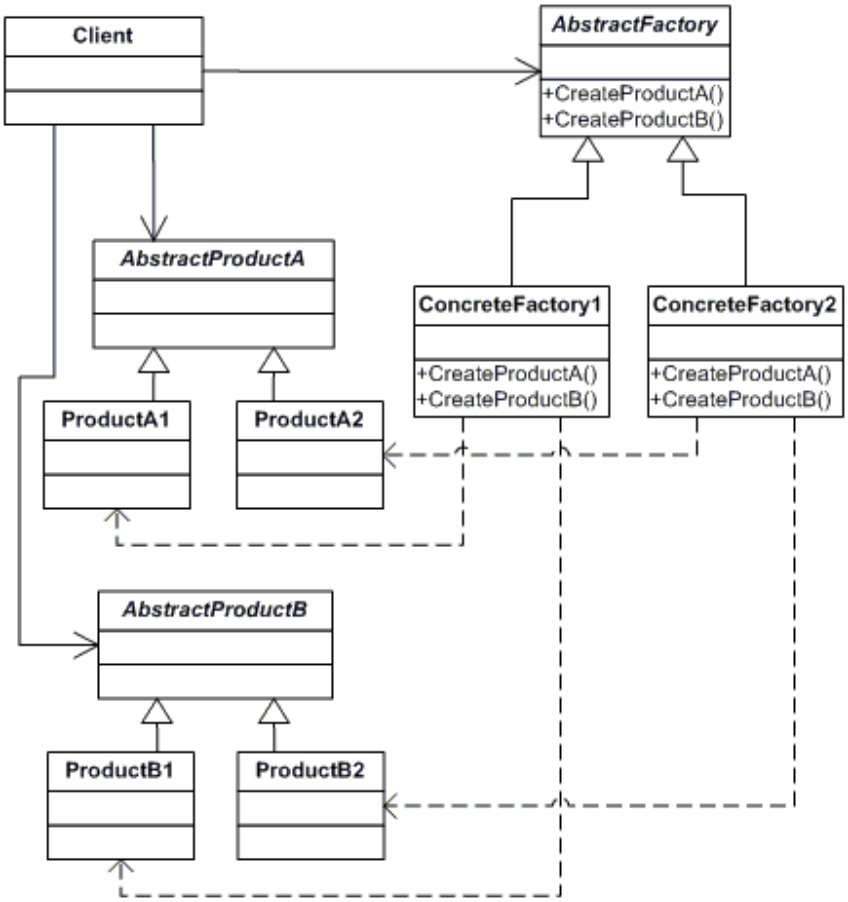
```
  // Properties
  public DataObject DataObject
  {
    set{ dataObject = value; }
    get{ return dataObject; }
  }

  // Methods
  virtual public void Next()
  {
    dataObject.NextRecord();
  }

  virtual public void Prior()
  {
    dataObject.PriorRecord();
  }

  virtual public void New( string name )
  {
    dataObject.NewRecord( name );
  }

  virtual public void Delete( string name )
  {
    dataObject.DeleteRecord( name );
  }

  virtual public void Show()
  {
    dataObject.ShowRecord();
  }

  virtual public void ShowAll()
  {
    Console.WriteLine( "Customer Group: {0}", group );
    dataObject.ShowAllRecords();
  }
}

// "RefinedAbstraction"

class CustomersBusinessObject : BusinessObject
{
  // Constructors
  public CustomersBusinessObject( string group )
      : base( group ){}

  // Methods
  override public void ShowAll()
  {
    // Add separator lines
    Console.WriteLine();
    Console.WriteLine( "------------------------" );
    base.ShowAll();
    Console.WriteLine( "------------------------" );
  }
```

```
}

// "Implementor"

abstract class DataObject
{
  // Methods
  abstract public void NextRecord();
  abstract public void PriorRecord();
  abstract public void NewRecord( string name );
  abstract public void DeleteRecord( string name );
  abstract public void ShowRecord();
  abstract public void ShowAllRecords();
}

// "ConcreteImplementor"

class CustomersDataObject : DataObject
{
  // Fields
  private ArrayList customers = new ArrayList();
  private int current = 0;

  // Constructors
  public CustomersDataObject()
  {
    // Loaded from a database
    customers.Add( "Jim Jones" );
    customers.Add( "Samual Jackson" );
    customers.Add( "Allen Good" );
    customers.Add( "Ann Stills" );
    customers.Add( "Lisa Giolani" );
  }

  // Methods
  public override void NextRecord()
  {
    if( current <= customers.Count - 1 )
    current++;
  }

  public override void PriorRecord()
  {
    if( current > 0 )
      current--;
  }

  public override void NewRecord( string name )
  {
    customers.Add( name );
  }

  public override void DeleteRecord( string name )
  {
    customers.Remove( name );
  }
```

```
  public override void ShowRecord()
  {
    Console.WriteLine( customers[ current ] );
  }

  public override void ShowAllRecords()
  {
    foreach( string name in customers )
      Console.WriteLine( " " + name );
  }
}

/// <summary>
/// Client test
/// </summary>
public class BusinessApp
{
  public static void Main( string[] args )
  {
    // Create RefinedAbstraction
    CustomersBusinessObject customers =
    new CustomersBusinessObject(" Chicago ");

    // Set ConcreteImplementor
    customers.DataObject = new CustomersDataObject();

    // Exercise the bridge
    customers.Show();
    customers.Next();
    customers.Show();
    customers.Next();
    customers.Show();
    customers.New( "Henry Velasquez" );

    customers.ShowAll();
  }
}
```

Abstract Factory

- poskytuje rozhranie pre vytvorenie rodiny súvisiacich alebo závislých objektov bez špecifikovania ich konkrétnych tried
- vzor pre tvorbu



- cieľ: chceme mať množinu objektov pre špeciálnych klientov (alebo prípady)
- problém: vytvorenie (inštancovanie) rodiny súvisiacich objektov
- riešenie: koordinovať vytváranie rodiny objektov. Umožniť zaviesť pravidlá špecifikujúce ako vytvoriť objekty mimo klienta, ktorý ich chce používať.
- podieľajúce sa entity: AbstractFactory, concreteFactory, AbstractProduct, Product, Client
- *abstractFactory*: definuje rozhranie určujúce, ako vytvárať každého požadovaného člena rodiny. Typicky je každá rodina vytvorená s vlastnou jedinečnou *concreteFactory*, ktorá implementuje operácie nutné na vytvorenie *Product*u
- *abstractProduct*: definuje rozhranie pre *Product*

- následky: vzor oddeľuje to, aký objekt sa použije, od aplikačnej logiky (ako sa použije daný objekt)
- implementácia: definuje sa abstraktná trieda, ktorá špecifikuje, aké objekty majú byť vytvárané. Pre každú rodinu objektov sa vytvorí konkrétna trieda (potomok)

```
// Abstract Factory pattern -- Structural example

using System;

// "AbstractFactory"

abstract class AbstractFactory
{
  // Methods
  abstract public AbstractProductA CreateProductA();
  abstract public AbstractProductB CreateProductB();
}

// "ConcreteFactory1"

class ConcreteFactory1 : AbstractFactory
{
  // Methods
  override public AbstractProductA CreateProductA()
  {
    return new ProductA1();
  }
  override public AbstractProductB CreateProductB()
  {
    return new ProductB1();
  }
}

// "ConcreteFactory2"

class ConcreteFactory2 : AbstractFactory
{
  // Methods
  override public AbstractProductA CreateProductA()
  {
    return new ProductA2();
  }

  override public AbstractProductB CreateProductB()
  {
    return new ProductB2();
  }
}

// "AbstractProductA"

abstract class AbstractProductA
{
}
```

```
// "AbstractProductB"

abstract class AbstractProductB
{
  // Methods
  abstract public void Interact( AbstractProductA a );
}

// "ProductA1"

class ProductA1 : AbstractProductA
{
}

// "ProductB1"

class ProductB1 : AbstractProductB
{
  // Methods
  override public void Interact( AbstractProductA a )
  {
    Console.WriteLine( this + " interacts with " + a );
  }
}

// "ProductA2"

class ProductA2 : AbstractProductA
{
}

// "ProductB2"

class ProductB2 : AbstractProductB
{
  // Methods
  override public void Interact( AbstractProductA a )
  {
    Console.WriteLine( this + " interacts with " + a );
  }
}

// "Client" - the interaction environment of the products

class Environment
{
  // Fields
  private AbstractProductA AbstractProductA;
  private AbstractProductB AbstractProductB;

  // Constructors
  public Environment( AbstractFactory factory )
  {
    AbstractProductB = factory.CreateProductB();
    AbstractProductA = factory.CreateProductA();
```

```
  }

  // Methods
  public void Run()
  {
    AbstractProductB.Interact( AbstractProductA );
  }
}

/// <summary>
/// ClientApp test environment
/// </summary>
class ClientApp
{
  public static void Main(string[] args)
  {
    AbstractFactory factory1 = new ConcreteFactory1();
    Environment e1 = new Environment( factory1 );
    e1.Run();

    AbstractFactory factory2 = new ConcreteFactory2();
    Environment e2 = new Environment( factory2 );
    e2.Run();
  }
}
```

```
// Abstract Factory pattern -- Real World example

using System;

// "AbstractFactory"

abstract class ContinentFactory
{
  // Methods
  abstract public Herbivore CreateHerbivore();
  abstract public Carnivore CreateCarnivore();
}

// "ConcreteFactory1"

class AfricaFactory : ContinentFactory
{
  // Methods
  override public Herbivore CreateHerbivore()
  {
    return new Wildebeest();
  }
  override public Carnivore CreateCarnivore()
  {
    return new Lion();
  }
}

// "ConcreteFactory2"
```

```
class AmericaFactory : ContinentFactory
{
  // Methods
  override public Herbivore CreateHerbivore()
  {
    return new Bison();
  }

  override public Carnivore CreateCarnivore()
  {
    return new Wolf();
  }
}

// "AbstractProductA"

abstract class Herbivore
{
}

// "AbstractProductB"

abstract class Carnivore
{
  // Methods
  abstract public void Eat( Herbivore h );
}

// "ProductA1"

class Wildebeest : Herbivore
{
}

// "ProductB1"

class Lion : Carnivore
{
  // Methods
  override public void Eat( Herbivore h )
  {
    // eat wildebeest
    Console.WriteLine( this + " eats " + h );
  }
}

// "ProductA2"

class Bison : Herbivore
{
}

// "ProductB2"

class Wolf : Carnivore
```

```
{
  // Methods
  override public void Eat( Herbivore h )
  {
    // Eat bison
    Console.WriteLine( this + " eats " + h );
  }
}

// "Client"

class AnimalWorld
{
  // Fields
  private Herbivore herbivore;
  private Carnivore carnivore;

  // Constructors
  public AnimalWorld( ContinentFactory factory )
  {
    carnivore = factory.CreateCarnivore();
    herbivore = factory.CreateHerbivore();
  }

  // Methods
  public void RunFoodChain()
  {
    carnivore.Eat( herbivore );
  }
}

/// <summary>
///  GameApp test class
/// </summary>
class GameApp
{
  public static void Main( string[] args )
  {
    // Create and run the Africa animal world
    ContinentFactory africa = new AfricaFactory();
    AnimalWorld world = new AnimalWorld( africa );
    world.RunFoodChain();

    // Create and run the America animal world
    ContinentFactory america = new AmericaFactory();
    world = new AnimalWorld( america );
    world.RunFoodChain();
  }
}
```

Princípy návrhových vzorov

open-close princíp

- Bertrand Meyer (1997)
- software by mal byť otvorený pre rozširovanie, uzavretý pre modifikáciu
- zväčšenie možností sw bez jeho zmeny

dependency inversion princíp

- existuje tiež dependency inversion of controll (používaná trieda/entita by nemala nič vedieť/predpokladať/závisieť od klientskej triedy, ktorá ju používa)
- dependency inversion: vysokoúrovňové entity by nemali priamo závisieť od nízkoúrovňových, ale oba druhy by mali závisieť od svojich abstrakcií (a tie už navzájom môžu závisieť)

Liskovej princíp

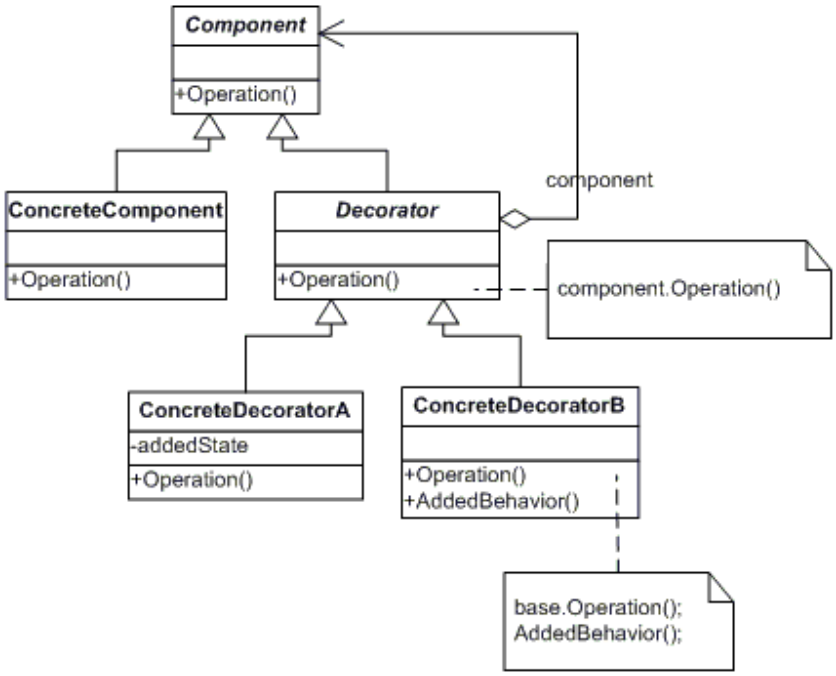
- Barbara Liskov (1988)
- trieda odvodená od rodičovskej triedy by mala podporovať všetko jej správanie (a nič navyše)
- potomok by teda nemal mať nové verejné metódy
- výhoda v zámennosti

Abstraktné triedy vs. Interface

- abstr. triedy umožňujú definovať všeobecný stav (cez atribúty)
- rozhrania umožňujú definovať len všeobecné správanie (to môžu aj triedy)
- rozhrania umožňujú viacnásobnú dedičnosť (v Jave a C# sa iným spôsobom nedá dediť od viacerých tried, v C++ je viacnásobné dedenie približne podobné implementovaniu rozhraní)

Decorator

- dynamicky pridá zodpovednosť(i) nejakému objektu pomocou reťaze volaní konštruktora tohto objektu
- štrukturálny vzor



- cieľ: dynamicky pridať objektu ďalšiu zodpovednosť (alebo viac)
- problém: používaný objekt má základnú funkcionalitu, ku ktorej je niekedy nutné pridať ďalšiu funkcionalitu (volanú pred alebo po základnej funcionalite)
- riešenie: pridať novú funkčnosť bez ovplyvnenia základnej či iných odvodených tried
- podieľajúce sa entity: Component, concreteComponent, Decorator, ConcreteDecorator
- *concreteComponent*: je trieda, ku ktorej je pridaná dodatočná funkčnosť *Decoratorom* (ktorý naň udržuje odkaz). Týmto elementom končí reťaz volaní.
- *Component*: definuje rozhranie pre ostatné všetky triedy vzoru
- *concreteDecorator*: implementuje zodpovednosť
- následky: pridaná funčnosť je v malých objektoch. Výhoda je v možnosti dynamicky pridávať túto funkčnosť pred/po volaní funčnosti v *ConcreteComponent*
- implementácia: vytvorí sa abstraktná trieda reprezentujúca originálnu triedu novú funkčnosť pridanú do triedy. V dekorátoroch sa umiestnia nové volania funkcií pred/po volaní funkcie nadradenej triedy.

```
// Decorator pattern -- Structural example

using System;

// "Component"

abstract class Component
{
  // Methods
  abstract public void Operation();
}

// "ConcreteComponent"

class ConcreteComponent : Component
{
  // Methods
  override public void Operation()
  {
    Console.WriteLine("ConcreteComponent.Operation()");
  }
}

// "Decorator"

abstract class Decorator : Component
{
  // Fields
  protected Component component;

  // Methods
  public void SetComponent( Component component )
  {
    this.component = component;
  }

  override public void Operation()
  {
    if( component != null )
      component.Operation();
  }
}

// "ConcreteDecoratorA"

class ConcreteDecoratorA : Decorator
{
  // Fields
  private string addedState;

  // Methods
  override public void Operation()
  {
    base.Operation();
    addedState = "new state";
```

```
    Console.WriteLine("ConcreteDecoratorA.Operation()");
  }
}

// "ConcreteDecoratorB"

class ConcreteDecoratorB : Decorator
{
  // Methods
  override public void Operation()
  {
    base.Operation();
    AddedBehavior();
    Console.WriteLine("ConcreteDecoratorB.Operation()");
  }

  void AddedBehavior()
  {
  }
}

/// <summary>
/// Client test
/// </summary>
public class Client
{
  public static void Main( string[] args )
  {
    // Create ConcreteComponent and two Decorators
    ConcreteComponent c = new ConcreteComponent();
    ConcreteDecoratorA d1 = new ConcreteDecoratorA();
    ConcreteDecoratorB d2 = new ConcreteDecoratorB();

    // Link decorators
    d1.SetComponent( c );
    d2.SetComponent( d1 );

    d2.Operation();
  }
}
```

```
// Decorator pattern -- Real World example

using System;
using System.Collections;

// "Component"

abstract class LibraryItem
{
  // Fields
  private int numCopies;
  // Properties

  public int NumCopies
  {
```

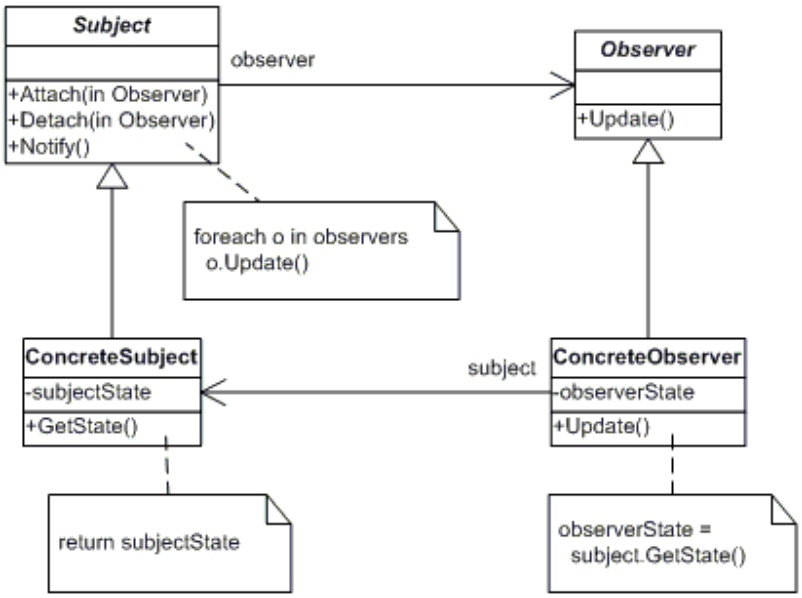
```
    get{ return numCopies; }
    set{ numCopies = value; }
  }

  // Methods
  public abstract void Display();
}

// "ConcreteComponent"

class Book : LibraryItem
{
  // Fields
  private string author;
  private string title;

  // Constructors
  public Book(string author,string title,int numCopies)
  {
    this.author = author;
    this.title = title;
    this.NumCopies = numCopies;
  }

  // Methods
  public override void Display()
  {
    Console.WriteLine( "\nBook ------ " );
    Console.WriteLine( " Author: {0}", author );
    Console.WriteLine( " Title: {0}", title );
    Console.WriteLine( " # Copies: {0}", NumCopies );
  }
}

// "ConcreteComponent"

class Video : LibraryItem
{
  // Fields
  private string director;
  private string title;
  private int playTime;

  // Constructor
  public Video( string director, string title,
                      int numCopies, int playTime )
  {
    this.director = director;
    this.title = title;
    this.NumCopies = numCopies;
    this.playTime = playTime;
  }

  // Methods
  public override void Display()
  {
    Console.WriteLine( "\nVideo ----- " );
```

```
    Console.WriteLine( " Director: {0}", director );
    Console.WriteLine( " Title: {0}", title );
    Console.WriteLine( " # Copies: {0}", NumCopies );
    Console.WriteLine( " Playtime: {0}", playTime );
  }
}

// "Decorator"

abstract class Decorator : LibraryItem
{
  // Fields
  protected LibraryItem libraryItem;

  // Constructors
  public Decorator ( LibraryItem libraryItem )
  {
    this.libraryItem = libraryItem;
  }

  // Methods
  public override void Display()
  {
    libraryItem.Display();
  }
}

// "ConcreteDecorator"

class Borrowable : Decorator
{
  // Fields
  protected ArrayList borrowers = new ArrayList();

  // Constructors
  public Borrowable( LibraryItem libraryItem )
                           : base( libraryItem ) {}

  // Methods
  public void BorrowItem( string name )
  {
    borrowers.Add( name );
    libraryItem.NumCopies--;
  }

  public void ReturnItem( string name )
  {
    borrowers.Remove( name );
     libraryItem.NumCopies++;
  }

  public override void Display()
  {
    base.Display();
    foreach( string borrower in borrowers )
      Console.WriteLine( " borrower: {0}", borrower );
  }
```

```
}

/// <summary>
///  DecoratorApp test
/// </summary>
public class DecoratorApp
{
  public static void Main( string[] args )
  {
    // Create book and video and display
    Book book = new Book( "Schnell", "My Home", 10 );
    Video video = new Video( "Spielberg",
                          "Schindler's list", 23, 60 );
    book.Display();
    video.Display();

    // Make video borrowable, then borrow and display
    Console.WriteLine( "\nVideo made borrowable:" );
    Borrowable borrowvideo = new Borrowable( video );
    borrowvideo.BorrowItem( "Cindy Lopez" );
    borrowvideo.BorrowItem( "Samuel King" );

    borrowvideo.Display();

  }
}
```

Observer

- definuje vzťah medzi jedným objektom a jeho pozorovateľmi; ak sa prvý objekt zmení, pozorovatelia sú o tom automaticky oboznámení
- vzor správania sa




```
// Observer pattern -- Structural example

using System;
using System.Collections;

// "Subject"

abstract class Subject
{
  // Fields
  private ArrayList observers = new ArrayList();

  // Methods
  public void Attach( Observer observer )
  {
    observers.Add( observer );
  }

  public void Detach( Observer observer )
  {
    observers.Remove( observer );
  }
```

```
  public void Notify()
  {
    foreach( Observer o in observers )
      o.Update();
  }
}

// "ConcreteSubject"

class ConcreteSubject : Subject
{
  // Fields
  private string subjectState;

  // Properties
  public string SubjectState
  {
    get{ return subjectState; }
    set{ subjectState = value; }
  }
}

// "Observer"

abstract class Observer
{
  // Methods
  abstract public void Update();
}

// "ConcreteObserver"

class ConcreteObserver : Observer
{
  // Fields
  private string name;
  private string observerState;
  private ConcreteSubject subject;

  // Constructors
  public ConcreteObserver( ConcreteSubject subject,
                                    string name )
  {
    this.subject = subject;
    this.name = name;
  }

  // Methods
  override public void Update()
  {
    observerState = subject.SubjectState;
    Console.WriteLine( "Observer {0}'s new state is {1}",
                                    name, observerState );
  }

  // Properties
  public ConcreteSubject Subject
```

```
  {
    get { return subject; }
    set { subject = value; }
  }
}

/// <summary>
/// Client test
/// </summary>
public class Client
{
  public static void Main( string[] args )
  {
    // Configure Observer structure
    ConcreteSubject s = new ConcreteSubject();
    s.Attach( new ConcreteObserver( s, "X" ) );
    s.Attach( new ConcreteObserver( s, "Y" ) );
    s.Attach( new ConcreteObserver( s, "Z" ) );

    // Change subject and notify observers
    s.SubjectState = "ABC";
    s.Notify();
  }
}
```

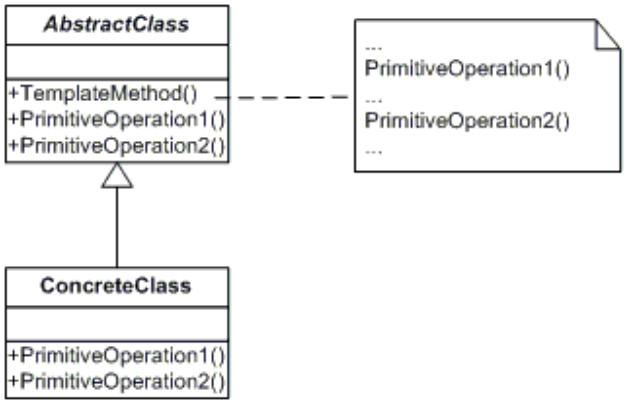
```
// Observer pattern -- Real World example

using System;
using System.Collections;

// "Subject"

abstract class Stock
{
  // Fields
  protected string symbol;
  protected double price;
  private ArrayList investors = new ArrayList();

  // Constructor
  public Stock( string symbol, double price )
  {
    this.symbol = symbol;
    this.price = price;
  }

  // Methods
  public void Attach( Investor investor )
  {
    investors.Add( investor );
  }

  public void Detach( Investor investor )
  {
    investors.Remove( investor );
  }
```

```
  public void Notify()
  {
    foreach( Investor i in investors )
      i.Update( this );
  }

  // Properties
  public double Price
  {
    get{ return price; }
    set{ price = value;
         Notify(); }
  }

  public string Symbol
  {
    get{ return symbol; }
    set{ symbol = value; }
  }
}

// "ConcreteSubject"

class IBM : Stock
{
  // Constructor
  public IBM( string symbol, double price )
                  : base( symbol, price ) {}
}

// "Observer"

interface IInvestor
{
  // Methods
  void Update( Stock stock );
}

// "ConcreteObserver"

class Investor : IInvestor
{
  // Fields
  private string name;
  private string observerState;
  private Stock stock;

  // Constructors
  public Investor( string name )
  {
    this.name = name;
  }

  // Methods
  public void Update( Stock stock )
  {
```

```
    Console.WriteLine( "Notified investor {0} of {1}'s " +
      change to {2:C}", name, stock.Symbol, stock.Price );
  }

  // Properties
  public Stock Stock
  {
    get{ return stock; }
    set{ stock = value; }
  }
}

/// <summary>
/// ObserverApp test
/// </summary>
public class ObserverApp
{
  public static void Main( string[] args )
  {
    // Create investors
    Investor s = new Investor( "Sorros" );
    Investor b = new Investor( "Berkshire" );

    // Create IBM stock and attach investors
    IBM ibm = new IBM( "IBM", 120.00 );
    ibm.Attach( s );
    ibm.Attach( b );

    // Change price, which notifies investors
    ibm.Price = 120.10;
    ibm.Price = 121.00;
    ibm.Price = 120.50;
    ibm.Price = 120.75;
  }
}
```

Template

- definuje kostru algoritmu pomocou operácií implementovaných v podtriedach
- vzor správania sa




```
// Template Method pattern -- Structural example

using System;

// "AbstractClass"

abstract class AbstractClass
{
  // Methods
  abstract public void PrimitiveOperation1();
  abstract public void PrimitiveOperation2();

  // The Template method
  public void TemplateMethod()
  {
    Console.WriteLine(
                "In AbstractClass.TemplateMethod()");
    PrimitiveOperation1();
    PrimitiveOperation2();
  }
}

// "ConcreteClass"

class ConcreteClass : AbstractClass
{
  // Methods
  public override void PrimitiveOperation1()
  {
```

```
    Console.WriteLine(
        "Called ConcreteClass.PrimitiveOperation1()");
  }

  public override void PrimitiveOperation2()
  {
    Console.WriteLine(
        "Called ConcreteClass.PrimitiveOperation2()");
  }
}

/// <summary>
/// Client test
/// </summary>
public class Client
{
  public static void Main( string[] args )
  {
    // Create instance and call template method
    ConcreteClass c = new ConcreteClass();
    c.TemplateMethod();

  }
}
```

```
// Template Method pattern -- Real World example

using System;
using System.Data;
using System.Data.OleDb;

// "AbstractClass"

abstract class DataObject
{
  // Methods
  abstract public void Connect();
  abstract public void Select();
  abstract public void Process();
  abstract public void Disconnect();

  // The "Template Method"
  public void Run()
  {
    Connect();
    Select();
    Process();
    Disconnect();
  }
}

// "ConcreteClass"

class CustomerDataObject : DataObject
{
  private string connectionString =
```

```
          "provider=Microsoft.JET.OLEDB.4.0; "
          + "data source=c:\\nwind.mdb";
  private string commandString;
  private DataSet dataSet;

  // Methods
  public override void Connect( )
  {
    // Nothing to do
  }

  public override void Select( )
  {
    commandString = "select CompanyName from Customers";
    OleDbDataAdapter dataAdapter = new OleDbDataAdapter(
               commandString, connectionString );
    dataSet = new DataSet();
    dataAdapter.Fill( dataSet, "Customers" );
  }

  public override void Process()
  {
    DataTable dataTable = dataSet.Tables["Customers"];
    foreach( DataRow dataRow in dataTable.Rows )
      Console.WriteLine( dataRow[ "CompanyName" ] );
  }

  public override void Disconnect()
  {
    // Nothing to do
  }
}

/// <summary>
///  TemplateMethodApp test
/// </summary>
public class TemplateMethodApp
{
  public static void Main( string[] args )
  {
    CustomerDataObject c = new CustomerDataObject( );
    c.Run();

  }
}
```

O továrňach

manažment vytvárania objektov

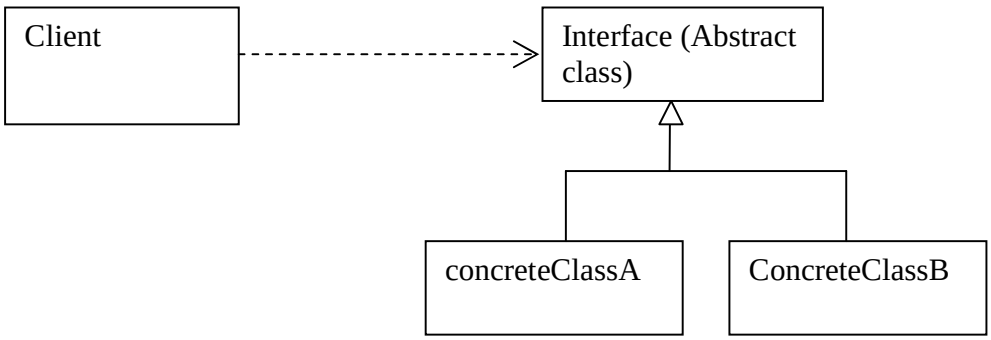
- tradičný (predošlý) názor: kto objekt používa, ten nech si ho vytvorí
- nový pohľad: vytváranie objektov je zložité, je nutné rozhodnúť, kedy ich vytvoriť, s akými parametrami, ako zabezpečiť znovupoužitie (pool), často sa zbytočne príliš skoro vytvorí väzba medzi používaným a klientskym objektom
- z dôvodu lepšej súdržnosti, väzby a testovateľnosti sa odporúča presúnúť zodpovednosť za správu objektov na nejakú továreň

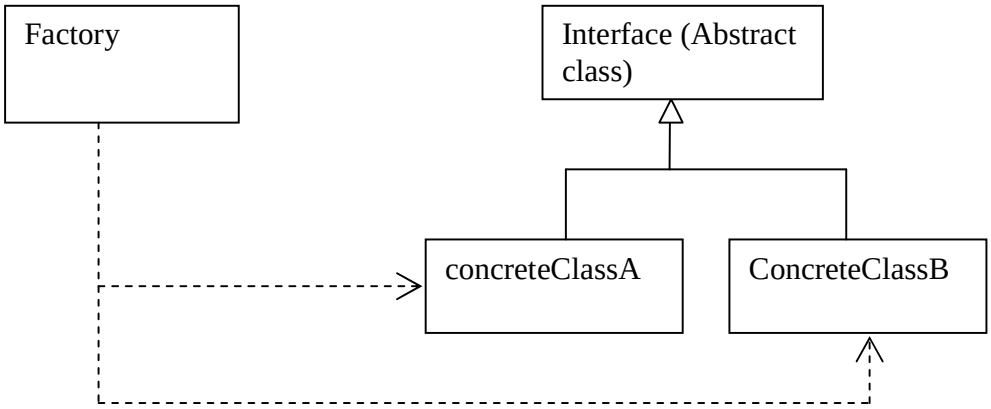
továrne

- metóda, objekt, ... používaný na vytváranie objektov
- konkrétne:
  - abstract factory pattern
  - builder pattern
  - factory method pattern
  - prototype
  - singleton

vývoj

- kroky:
  - definovať triedy a spôsob ich spolupráce
  - definovať továrne, ktoré budú vytvárať správne inštancie pre danú situáciu a spravovať existujúce objekty pre prípad ich zdieľania
- vyššia súdržnosť (rozdelenie funkcionality), nižšia väzba (každý krok robí niečo iné a nezávisí od druhého)





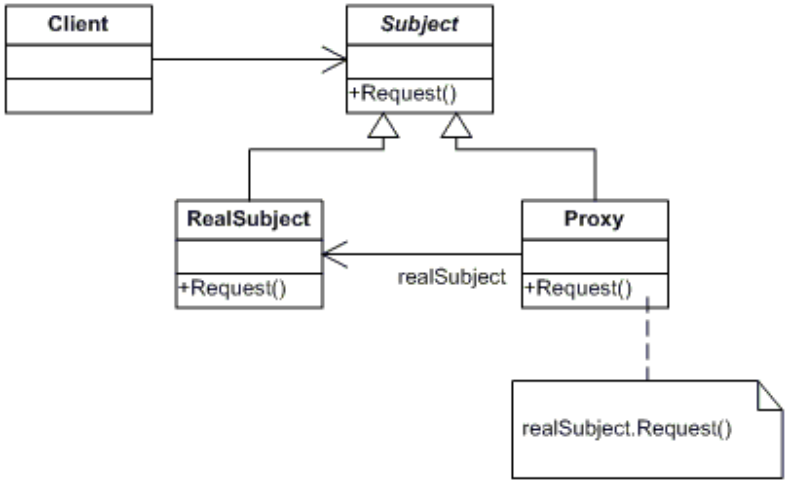
obmedzenie dosahu zmien

- síce je zvyčajne veľa klientov používajúcich objekty, je len jedno miesto, kde sa objekty vytvárajú
- zmena sa zyčajne týka toho, ktorý objekt je požadovaný alebo spôsobu, akým je objekt používaný
- vytváranie objektov sa presunie na neskorší čas
- vytváranie nového kódu je zvyčajne jednoduchšie, než integrovanie ďalšieho

- OO vzory zvyčajne vytvoria zložitejší kód (než napr. procedurálnym spôsobom), keďže vytvoria vrstvu medzi klientom a používanými objektami
- továrne ukryjú komplikovanosť týkajúcu sa zložitých znalostí o vytváraní týchto objektov

Proxy

- poskytuje obal pre iný objekt, ktorý treba riadiť
- štrukturálny vzor
- typy: remote (pre subjekt v inom adresnom priestore), virtual (cache), protection (dodatočná ochrana)




```
// Proxy pattern -- Structural example

using System;

// "Subject"

abstract class Subject
{
  // Methods
  abstract public void Request();
}

// "RealSubject"

class RealSubject : Subject
{
  // Methods
  override public void Request()
  {
    Console.WriteLine("Called RealSubject.Request()");
  }
}

// "Proxy"
```

```
class Proxy : Subject
{
  // Fields
  RealSubject realSubject;

  // Methods
  override public void Request()
  {
    // Uses "lazy initialization"
    if( realSubject == null )
      realSubject = new RealSubject();

    realSubject.Request();
  }
}

/// <summary>
/// Client test
/// </summary>
public class Client
{
  public static void Main( string[] args )
  {
    // Create proxy and request a service
    Proxy p = new Proxy();
    p.Request();

  }
}
```

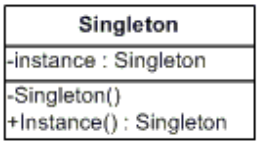
```
// Proxy pattern -- Real World example

using System;
using System.Runtime.Remoting;

// "Subject"

public interface IMath
{
  // Methods
  double Add( double x, double y );
  double Sub( double x, double y );
  double Mul( double x, double y );
  double Div( double x, double y );
}

// "RealSubject"

class Math : MarshalByRefObject, IMath
{
  // Methods
  public double Add( double x, double y ){ return x + y; }
  public double Sub( double x, double y ){ return x - y; }
  public double Mul( double x, double y ){ return x * y; }
  public double Div( double x, double y ){ return x / y; }
}
```

```
// Remote "Proxy Object"

class MathProxy : IMath
{
  // Fields
  Math math;
  // Constructors
  public MathProxy()
  {
    // Create Math instance in a different AppDomain
    AppDomain ad = System.AppDomain.CreateDomain(
                                  "MathDomain",null, null );
    ObjectHandle o =
      ad.CreateInstance("Proxy_RealWorld", "Math", false,
      System.Reflection.BindingFlags.CreateInstance,
      null, null, null,null,null );
    math = (Math) o.Unwrap();
  }

  // Methods
  public double Add( double x, double y )
  {
    return math.Add(x,y);
  }

  public double Sub( double x, double y )
  {
    return math.Sub(x,y);
  }

  public double Mul( double x, double y )
  {
    return math.Mul(x,y);
  }

  public double Div( double x, double y )
  {
    return math.Div(x,y);
  }
}

/// <summary>
/// ProxyApp test
/// </summary>
public class ProxyApp
{
  public static void Main( string[] args )
  {
    // Create math proxy
    MathProxy p = new MathProxy();

    // Do the math
    Console.WriteLine( "4 + 2 = {0}", p.Add( 4, 2 ) );
    Console.WriteLine( "4 - 2 = {0}", p.Sub( 4, 2 ) );
    Console.WriteLine( "4 * 2 = {0}", p.Mul( 4, 2 ) );
    Console.WriteLine( "4 / 2 = {0}", p.Div( 4, 2 ) );
```

```
  }
}
```

Singleton

- zaistí, že trieda má iba jednu inštanciu a poskytne pre ňu globálny prístup
- Singleton má špeciálnu metódu, ktorá vytvára inštanciu v prípade potreby (lazy instantiantion) a poskytuje ju na požiadanie



- Singleton pre multithreading (viacvláknové) aplikácie môže byť problematický, v závislosti od stavovosti
- preto sa tu používa dvojúrovňová kontrola (double-checked locking): dvojitá kontrola na nulitu inštancie, pričom medzi kontrolami sa kód synchronizuje prostriedkami jazyka (v Jave pomocou synchronized)

```
// Singleton pattern -- Structural example

using System;

// "Singleton"

class Singleton
{
  // Fields
  private static Singleton instance;

  // Constructor
  protected Singleton() {}

  // Methods
  public static Singleton Instance()
  {
    // Uses "Lazy initialization"
    if( instance == null )
      instance = new Singleton();

    return instance;
  }
}

/// <summary>
/// Client test
/// </summary>
public class Client
{
  public static void Main()
  {
```

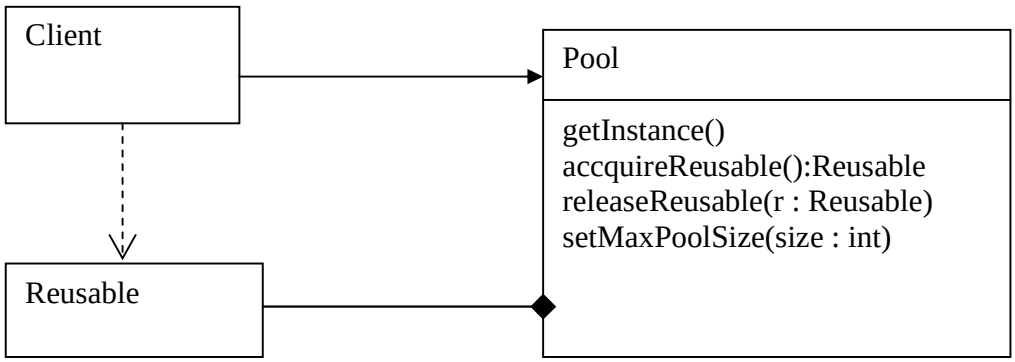
```
    // Constructor is protected -- cannot use new
    Singleton s1 = Singleton.Instance();
    Singleton s2 = Singleton.Instance();

    if( s1 == s2 )
      Console.WriteLine( "The same instance" );
  }
}
```

```
// Singleton pattern -- Real World example

using System;
using System.Collections;
using System.Threading;

// "Singleton"

class LoadBalancer
{
  // Fields
  private static LoadBalancer balancer;
  private ArrayList servers = new ArrayList();
  private Random random = new Random();

  // Constructors (protected)
  protected LoadBalancer()
  {
    // List of available servers
    servers.Add( "ServerI" );
    servers.Add( "ServerII" );
    servers.Add( "ServerIII" );
    servers.Add( "ServerIV" );
    servers.Add( "ServerV" );
  }

  // Methods
  public static LoadBalancer GetLoadBalancer()
  {
    // Support multithreaded applications through
    // "Double checked locking" pattern which avoids
    // locking every time the method is invoked
    if( balancer == null )
    {
      // Only one thread can obtain a mutex
      Mutex mutex = new Mutex();
      mutex.WaitOne();

      if( balancer == null )
        balancer = new LoadBalancer();

      mutex.Close();
    }
    return balancer;
  }

  // Properties
```

```
  public string Server
  {
    get
    {
      // Simple, but effective random load balancer
      int r = random.Next( servers.Count );
      return servers[ r ].ToString();
    }
  }
}

/// <summary>
/// SingletonApp test
/// </summary>
///
public class SingletonApp
{
  public static void Main( string[] args )
  {
    LoadBalancer b1 = LoadBalancer.GetLoadBalancer();
    LoadBalancer b2 = LoadBalancer.GetLoadBalancer();
    LoadBalancer b3 = LoadBalancer.GetLoadBalancer();
    LoadBalancer b4 = LoadBalancer.GetLoadBalancer();

    // Same instance?
    if( (b1 == b2) && (b2 == b3) && (b3 == b4) )
      Console.WriteLine( "Same instance" );

    // Do the load balancing
    Console.WriteLine( b1.Server );
    Console.WriteLine( b2.Server );
    Console.WriteLine( b3.Server );
    Console.WriteLine( b4.Server );
  }
}
```

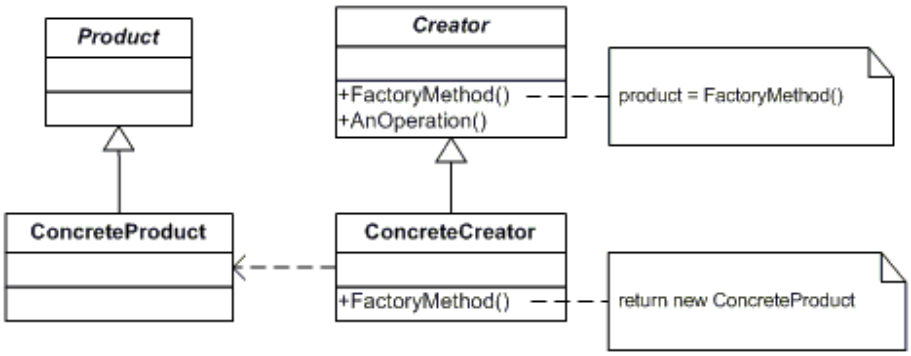
Object Pool

- riadenie znovupoužiteľnosti objektov v prípade, že drahé tieto objekty vytvárať alebo ich môže byť len obmedzený počet
- Client zavolá metódu Pool.acquireObject, ktorá buď vráti nejaký voľný objekt, alebo ho vytvorí (ak voľný nie je) alebo čaká, kým sa nejaký objekt neuvoľní
- Client po použití objektu zavolá metódu Pool.releaseObject, čím ho uvoľní pre ďalšie použitie (taktiež je možné nechať objekt, nech sa sám uvoľní)
- tento vzor je výhodný v prípade rovnomernej záťaže
- implementácia je pomocou poľa (pevného alebo dynamického) objektov, pre Pool sa využije Singleton



Factory method

- definuje rozhranie pre vytváranie objektu, avšak až podtriedy rozhodujú o tom, ktorý objekt sa inštancuje
- vzor pre tvorbu




```
// Factory Method pattern -- Structural example

using System;
using System.Collections;

// "Product"

abstract class Product
{
}

// "ConcreteProductA"

class ConcreteProductA : Product
{
}

// "ConcreteProductB"

class ConcreteProductB : Product
{
}

// "Creator"

abstract class Creator
{
  // Methods
  abstract public Product FactoryMethod();
}
```

```
// "ConcreteCreatorA"

class ConcreteCreatorA : Creator
{
  // Methods
  override public Product FactoryMethod()
  {
    return new ConcreteProductA();
  }
}

// "ConcreteCreatorB"

class ConcreteCreatorB : Creator
{
  // Methods
  override public Product FactoryMethod()
  {
    return new ConcreteProductB();
  }
}

/// <summary>
/// Client test
/// </summary>
class Client
{
  public static void Main( string[] args )
  {

    // FactoryMethod returns ProductA
    Creator c = new ConcreteCreatorA();
    Product p = c.FactoryMethod();
    Console.WriteLine( "Created {0}", p );

    // FactoryMethod returns ProductB
    c = new ConcreteCreatorB();
    p = c.FactoryMethod();
    Console.WriteLine( "Created {0}", p );

  }
}
```

```
// Factory Method pattern -- Real World example

using System;
using System.Collections;

// "Product"

abstract class Page
{
}
```

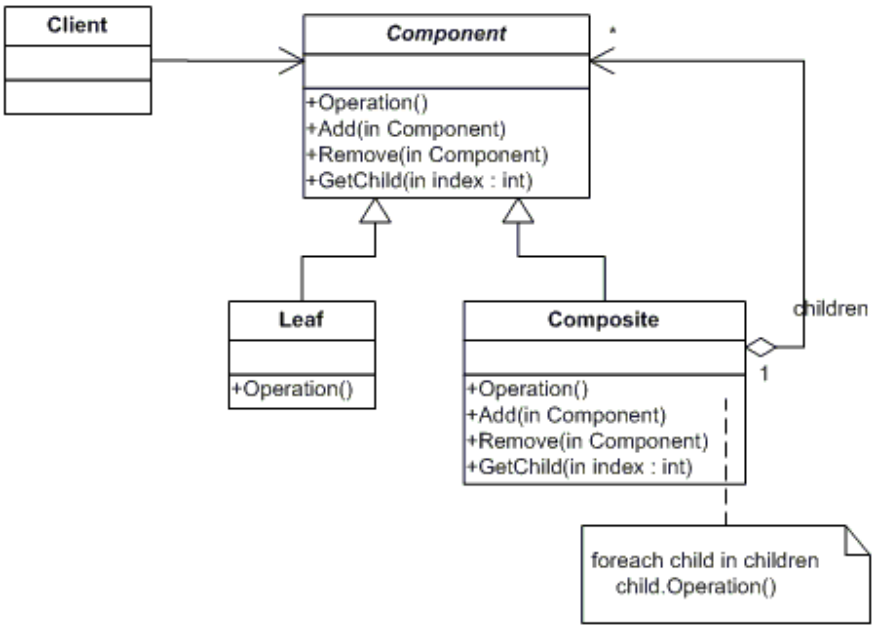
```
// "ConcreteProduct"

class SkillsPage : Page
{
}

// "ConcreteProduct"

class EducationPage : Page
{
}

// "ConcreteProduct"

class ExperiencePage : Page
{
}

// "ConcreteProduct"

class IntroductionPage : Page
{
}

// "ConcreteProduct"

class ResultsPage : Page
{
}

// "ConcreteProduct"

class ConclusionPage : Page
{
}

// "ConcreteProduct"

class SummaryPage : Page
{
}

// "ConcreteProduct"

class BibliographyPage : Page
{
}

// "Creator"

abstract class Document
{
  // Fields
  protected ArrayList pages = new ArrayList();

  // Constructor
  public Document()
```

```
  {
    this.CreatePages();
  }

  // Properties
  public ArrayList Pages
  {
    get{ return pages; }
  }

  // Factory Method
  abstract public void CreatePages();
}

// "ConcreteCreator"

class Resume : Document
{
  // Factory Method implementation
  override public void CreatePages()
  {
    pages.Add( new SkillsPage() );
    pages.Add( new EducationPage() );
    pages.Add( new ExperiencePage() );
  }
}

// "ConcreteCreator"

class Report : Document
{
  // Factory Method implementation
  override public void CreatePages()
  {
    pages.Add( new IntroductionPage() );
    pages.Add( new ResultsPage() );
    pages.Add( new ConclusionPage() );
    pages.Add( new SummaryPage() );
    pages.Add( new BibliographyPage() );
  }
}

/// <summary>
///  FactoryMethodApp test
/// </summary>
class FactoryMethodApp
{
  public static void Main( string[] args )
  {
    Document[] docs = new Document[ 2 ];

    // Note: constructors call Factory Method
    docs[0] = new Resume();
    docs[1] = new Report();

    // Display document pages
    foreach( Document document in docs )
```

```
    {
      Console.WriteLine( "\n" + document + " ------- " );
      foreach( Page page in document.Pages )
        Console.WriteLine( " " + page );
    }
  }
}
```

Composite

- vytvorenie stromovej štruktúry obsahujúcej rôzne druhy objektov
- klient pristupuje ku všetkým objektom rovnakým spôsobom
- vzor pre tvorbu




```
// Composite pattern -- Structural example

using System;
using System.Text;
using System.Collections;

// "Component"

abstract class Component
{
  // Fields
  protected string name;

  // Constructors
  public Component( string name )
  {
    this.name = name;
  }

  // Methods
  abstract public void Add(Component c);
  abstract public void Remove( Component c );
```

```
  abstract public void Display( int depth );
}

// "Composite"

class Composite : Component
{
  // Fields
  private ArrayList children = new ArrayList();

  // Constructors
  public Composite( string name ) : base( name ) {}

  // Methods
  public override void Add( Component component )
  {
    children.Add( component );
  }
  public override void Remove( Component component )
  {
    children.Remove( component );
  }
  public override void Display( int depth )
  {
    Console.WriteLine( new String( '-', depth ) + name );

    // Display each of the node's children
    foreach( Component component in children )
      component.Display( depth + 2 );
  }
}

// "Leaf"

class Leaf : Component
{
  // Constructors
  public Leaf( string name ) : base( name ) {}

  // Methods
  public override void Add( Component c )
  {
    Console.WriteLine("Cannot add to a leaf");
  }

  public override void Remove( Component c )
  {
    Console.WriteLine("Cannot remove from a leaf");
  }

  public override void Display( int depth )
  {
    Console.WriteLine( new String( '-', depth ) + name );
  }
}

/// <summary>
```

```
/// Client test
/// </summary>
public class Client
{
  public static void Main( string[] args )
  {
    // Create a tree structure
    Composite root = new Composite( "root" );
    root.Add( new Leaf( "Leaf A" ));
    root.Add( new Leaf( "Leaf B" ));
    Composite comp = new Composite( "Composite X" );

    comp.Add( new Leaf( "Leaf XA" ) );
    comp.Add( new Leaf( "Leaf XB" ) );
    root.Add( comp );

    root.Add( new Leaf( "Leaf C" ));

    // Add and remove a leaf
    Leaf l = new Leaf( "Leaf D" );
    root.Add( l );
    root.Remove( l );

    // Recursively display nodes
    root.Display( 1 );
  }
}
```

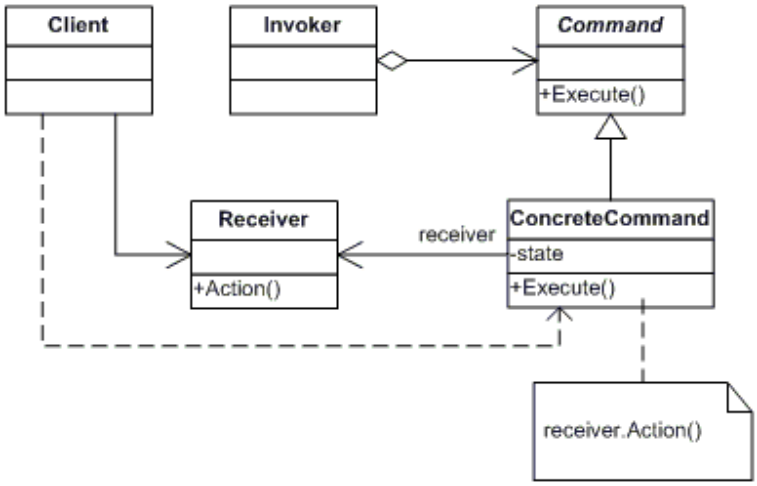
```
// Composite pattern -- Real World example

using System;
using System.Collections;

// "Component"

abstract class DrawingElement
{
  // Fields
  protected string name;

  // Constructors
  public DrawingElement( string name )
  {
    this.name = name;
  }

  // Methods
  abstract public void Add( DrawingElement d );
  abstract public void Remove( DrawingElement d );
  abstract public void Display( int indent );
}

// "Leaf"

class PrimitiveElement : DrawingElement
```

```
{
  // Constructors
  public PrimitiveElement( string name ) : base( name ) {}

  // Methods
  public override void Add( DrawingElement c )
  {
    Console.WriteLine("Cannot Add");
  }

  public override void Remove( DrawingElement c )
  {
    Console.WriteLine("Cannot Remove");
  }

  public override void Display( int indent )
  {
    Console.WriteLine( new String( '-', indent ) +
                                  " draw a {0}", name );
  }
}

// "Composite"

class CompositeElement : DrawingElement
{
  // Fields
  private ArrayList elements = new ArrayList();

  // Constructors
  public CompositeElement( string name )
                                     : base( name ) {}

  // Methods
  public override void Add( DrawingElement d )
  {
    elements.Add( d );
  }

  public override void Remove( DrawingElement d )
  {
    elements.Remove( d );
  }

  public override void Display( int indent )
  {
    Console.WriteLine( new String( '-', indent ) +
                                  "+ " + name );

    // Display each child element on this node
    foreach( DrawingElement c in elements )
      c.Display( indent + 2 );
  }
}

/// <summary>
///  CompositeApp test
```

```
/// </summary>
public class CompositeApp
{
  public static void Main( string[] args )
  {
    // Create a tree structure
    CompositeElement root = new
                    CompositeElement( "Picture" );
    root.Add( new PrimitiveElement( "Red Line" ));
    root.Add( new PrimitiveElement( "Blue Circle" ));
    root.Add( new PrimitiveElement( "Green Box" ));

    CompositeElement comp = new
                    CompositeElement( "Two Circles" );
    comp.Add( new PrimitiveElement( "Black Circle" ) );
    comp.Add( new PrimitiveElement( "White Circle" ) );
    root.Add( comp );

    // Add and remove a PrimitiveElement
    PrimitiveElement l = new
                    PrimitiveElement( "Yellow Line" );
    root.Add( l );
    root.Remove( l );

    // Recursively display nodes
    root.Display( 1 );
  }
}
```

Command

- požiadavka (správa) zasielaná medzi objektmi je tiež objekt
- umožní nám to rozlišovať medzi klientami pomocou správ, správy môžeme radiť, ukladať do logu, podporovať undo/redo
- vzor správania sa




```
// Command pattern -- Structural example

using System;

// "Command"

abstract class Command
{
  // Fields
  protected Receiver receiver;

  // Constructors
  public Command( Receiver receiver )
  {
    this.receiver = receiver;
  }

  // Methods
  abstract public void Execute();
}

// "ConcreteCommand"

class ConcreteCommand : Command
{
```

```
  // Constructors
  public ConcreteCommand( Receiver receiver ) :
                              base ( receiver ) {}

  // Methods
  public override void Execute()
  {
    receiver.Action();
  }
}

// "Receiver"

class Receiver
{
  // Methods
  public void Action()
  {
    Console.WriteLine("Called Receiver.Action()");
  }
}

// "Invoker"

class Invoker
{
  // Fields
  private Command command;

  // Methods
  public void SetCommand( Command command )
  {
    this.command = command;
  }

  public void ExecuteCommand()
  {
    command.Execute();
  }
}

/// <summary>
///  Client test
/// </summary>
public class Client
{
  public static void Main( string[] args )
  {
    // Create receiver, command, and invoker
    Receiver r = new Receiver();
    Command c = new ConcreteCommand( r );
    Invoker i = new Invoker();

    // Set and execute command
    i.SetCommand(c);
    i.ExecuteCommand();
  }
```

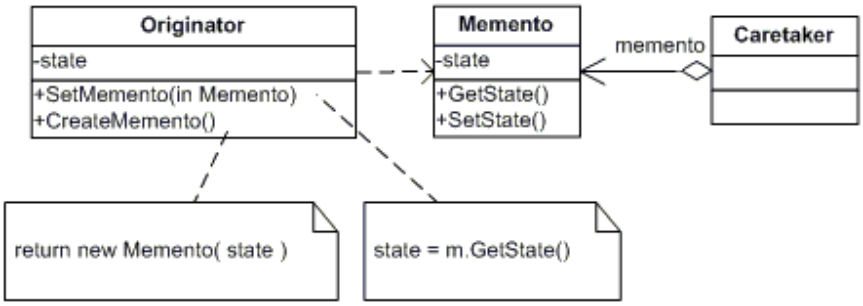
```
}

// Command pattern -- Real World example

using System;
using System.Collections;

// "Command"

abstract class Command
{
  // Methods
  abstract public void Execute();
  abstract public void UnExecute();
}

// "ConcreteCommand"

class CalculatorCommand : Command
{
  // Fields
  char @operator;
  int operand;
  Calculator calculator;

  // Constructor
  public CalculatorCommand( Calculator calculator,
                        char @operator, int operand )
  {
    this.calculator = calculator;
    this.@operator = @operator;
    this.operand = operand;
  }

  // Properties
  public char Operator
  {
    set{ @operator = value; }
  }

  public int Operand
  {
    set{ operand = value; }
  }

  // Methods
  override public void Execute()
  {
    calculator.Operation( @operator, operand );
  }

  override public void UnExecute()
  {
     calculator.Operation( Undo( @operator ), operand );
  }
```

```
  // Private helper function
  private char Undo( char @operator )
  {
    char undo = ' ';
    switch( @operator )
    {
      case '+': undo = '-'; break;
      case '-': undo = '+'; break;
      case '*': undo = '/'; break;
      case '/': undo = '*'; break;
    }
    return undo;
  }
}

// "Receiver"

class Calculator
{
  // Fields
  private int total = 0;

  // Methods
  public void Operation( char @operator, int operand )
  {
    switch( @operator )
    {
      case '+': total += operand; break;
      case '-': total -= operand; break;
      case '*': total *= operand; break;
      case '/': total /= operand; break;
    }
    Console.WriteLine( "Total = {0} (following {1} {2})",
                    total, @operator, operand );
  }
}

// "Invoker"

class User
{
  // Fields
  private Calculator calculator = new Calculator();
  private ArrayList commands = new ArrayList();
  private int current = 0;

  // Methods
  public void Redo( int levels )
  {
    Console.WriteLine( "---- Redo {0} levels ", levels );
    // Perform redo operations
    for( int i = 0; i < levels; i++ )
      if( current < commands.Count - 1 )
        ((Command)commands[ current++ ]).Execute();
  }

  public void Undo( int levels )
```

```
  {
    Console.WriteLine( "---- Undo {0} levels ", levels );
    // Perform undo operations
    for( int i = 0; i < levels; i++ )
      if( current > 0 )
        ((Command)commands[ --current ]).UnExecute();
  }

  public void Compute( char @operator, int operand )
  {
    // Create command operation and execute it
    Command command = new CalculatorCommand(
                calculator, @operator, operand );
    command.Execute();

    // Add command to undo list
    commands.Add( command );
    current++;
  }
}

/// <summary>
/// CommandApp test
/// </summary>
public class CommandApp
{
  public static void Main( string[] args )
  {
    // Create user and let her compute
    User user = new User();

    user.Compute( '+', 100 );
    user.Compute( '-', 50 );
    user.Compute( '*', 10 );
    user.Compute( '/', 2 );

    // Undo and then redo some commands
    user.Undo( 4 );
    user.Redo( 3 );
  }
}
```

Memento

- zachytenie (a uloženie) interného stavu objektu do mementa a opätovné obnovenie tohto stavu
- vzor správania sa




```
// Memento pattern -- Structural example

using System;

// "Originator"

class Originator
{
  // Fields
  private string state = "OFF";

  // Properties
  public string State
  {
    get{ return state; }
    set{ state = value; }
  }

  // Methods
  public Memento CreateMemento()
  {
    return (new Memento( state ));
  }

  public void SetMemento( Memento memento )
  {
    state = memento.State;
    Console.WriteLine( "Restored to state: {0}", state );
  }
}

// "Memento"
```

```
class Memento
{
  // Fields
  private string state;

  // Constructors
  public Memento( string state )
  {
    this.state = state;
  }

  // Properties
  public string State
  {
    get{ return state; }
  }
}

// "Caretaker"

class Caretaker
{
  // Fields
  private Memento memento;

  // Properties
  public Memento Memento
  {
    set{ memento = value; }
    get{ return memento; }
  }
}

/// <summary>
/// Client test
/// </summary>
public class Client
{
  public static void Main( string[] args )
  {
    Originator o = new Originator();
    o.State = "On";

    // Store internal state
    Caretaker c = new Caretaker();
    c.Memento = o.CreateMemento();

    // Continue changing originator
    o.State = "Off";

    // Restore saved state
    o.SetMemento( c.Memento );
  }
}

// Memento pattern -- Real World example
```

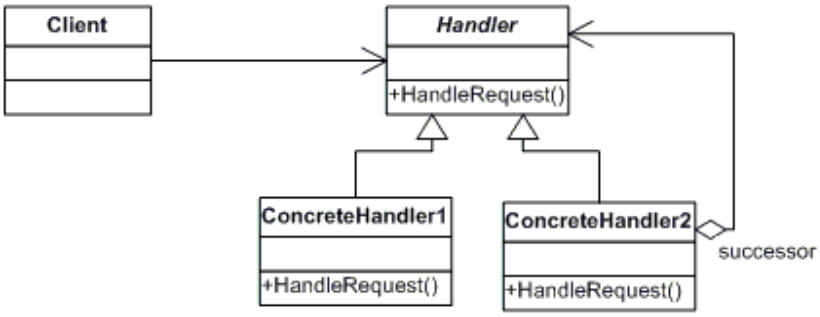
```
using System;

// "Originator"

class SalesProspect
{
  // Fields
  private string name;
  private string phone;
  private double budget;

  // Properties
  public string Name
  {
    get{ return name; }
    set{ name = value; }
  }

  public string Phone
  {
    get{ return phone; }
    set{ phone = value; }
  }

  public double Budget
  {
    get{ return budget; }
    set{ budget = value; }
  }

  // Methods
  public Memento SaveMemento()
  {
    return (new Memento( name, phone, budget ));
  }

  public void RestoreMemento( Memento memento )
  {
    this.name = memento.Name;
    this.phone = memento.Phone;
    this.budget = memento.Budget;
  }

  public void Show()
  {
    Console.WriteLine( "\nSales prospect ---- " );
    Console.WriteLine( "Name: {0}", this.name );
    Console.WriteLine( "Phone: {0}", this.phone );
    Console.WriteLine( "Budget: {0:C}", this.budget );
  }
}

// "Memento"

class Memento
{
```

```
  // Fields
  private string name;
  private string phone;
  private double budget;

  // Constructors
  public Memento(string name,string phone,double budget)
  {
    this.name = name;
    this.phone = phone;
    this.budget = budget;
  }

  // Properties
  public string Name
  {
    get{ return name; }
    set{ name = value; }
  }

  public string Phone
  {
    get{ return phone; }
    set{ phone = value; }
  }

  public double Budget
  {
    get{ return budget; }
    set{ budget = value; }
  }
}

// "Caretaker"

class ProspectMemory
{
  // Fields
  private Memento memento;

  // Properties
  public Memento Memento
  {
    set{ memento = value; }
    get{ return memento; }
  }
}

/// <summary>
/// MementoApp test
/// </summary>
public class MementoApp
{
  public static void Main( string[] args )
  {
    SalesProspect s = new SalesProspect();
    s.Name = "Noel van Halen";
```

```
    s.Phone = "(412) 256-0990";
    s.Budget = 25000.0;
    s.Show();

    // Store internal state
    ProspectMemory m = new ProspectMemory();
    m.Memento = s.SaveMemento();

    // Continue changing originator
    s.Name = "Leo Welch";
    s.Phone = "(310) 209-7111";
    s.Budget = 1000000.0;
    s.Show();

    // Restore saved state
    s.RestoreMemento( m.Memento );
    s.Show();
  }
}
```

Chain of responsibility

- zasielajúci objekt posiela správu objektu, ktorý na ňu buď reaguje alebo prenesie zodpovednosť za jej vybavenie na ďalší objekt v reťazi
- vzor správania sa




```
// Chain of Responsibility pattern -- Structural example

using System;

// "Handler"

abstract class Handler
{
  // Fields
  protected Handler successor;

  // Methods
  public void SetSuccessor( Handler successor )
  {
    this.successor = successor;
  }
  abstract public void HandleRequest( int request );
}

// "ConcreteHandler1"

class ConcreteHandler1 : Handler
{
  // Methods
  override public void HandleRequest( int request )
  {
    if( request >= 0 && request < 10 )
      Console.WriteLine("{0} handled request {1}",
        this, request );
    else
      if( successor != null )
        successor.HandleRequest( request );
  }
```

```
}

// "ConcreteHandler2"

class ConcreteHandler2 : Handler
{
  // Methods
  override public void HandleRequest( int request )
  {
    if( request >= 10 && request < 20 )
      Console.WriteLine("{0} handled request {1}",
        this, request );
    else
      if( successor != null )
        successor.HandleRequest( request );
  }
}

// "ConcreteHandler3"

class ConcreteHandler3 : Handler
{
  // Methods
  override public void HandleRequest( int request )
  {
    if( request >= 20 && request < 30 )
      Console.WriteLine("{0} handled request {1}",
        this, request );
    else
      if( successor != null )
        successor.HandleRequest( request );
  }
}

// "Request"

class Request
{
  // Fields
  private int iRequestType;
  private string strRequestParameters;

  // Constructors
  public Request(int requestType, string requestParameters)
  {
    iRequestType = requestType;
    strRequestParameters = requestParameters;
  }

  // Properties
  public int RequestType
  {
    get{ return iRequestType; }
    set{iRequestType = value; }
  }
}
```

```
/// <summary>
/// Client test
/// </summary>
public class Client
{
  public static void Main( string[] args )
  {
    // Setup Chain of Responsibility
    Handler h1 = new ConcreteHandler1();
    Handler h2 = new ConcreteHandler2();
    Handler h3 = new ConcreteHandler3();
    h1.SetSuccessor(h2);
    h2.SetSuccessor(h3);

    // Generate and process request
    int[] requests = { 2, 5, 14, 22, 18, 3, 27, 20 };

    foreach( int request in requests )
      h1.HandleRequest( request );

    }
}
```

```
// Chain of Responsibility pattern -- Real World example

using System;

// "Handler"

abstract class Approver
{
  // Fields
  protected string name;
  protected Approver successor;

  // Constructors
  public Approver( string name )
  {
    this.name = name;
  }

  // Methods
  public void SetSuccessor( Approver successor )
  {
    this.successor = successor;
  }

  abstract public void ProcessRequest(
                           PurchaseRequest request );
}

// "ConcreteHandler"

class Director : Approver
{
  // Constructors
```

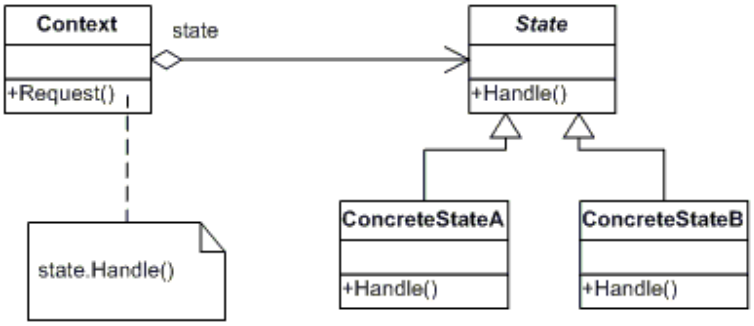
```
  public Director ( string name ) : base( name ) {}

  // Methods
  override public void ProcessRequest(
                          PurchaseRequest request )
  {
    if( request.Amount < 10000.0 )
      Console.WriteLine( "{0} {1} approved request# {2}",
        this, name, request.Number);
    else
      if( successor != null )
        successor.ProcessRequest( request );
  }
}

// "ConcreteHandler"

class VicePresident : Approver
{
  // Constructors
  public VicePresident ( string name ) : base( name ) {}

  // Methods
  override public void ProcessRequest(
                          PurchaseRequest request )
  {
    if( request.Amount < 25000.0 )
      Console.WriteLine( "{0} {1} approved request# {2}",
        this, name, request.Number);
    else
      if( successor != null )
        successor.ProcessRequest( request );
  }
}

// "ConcreteHandler"

class President : Approver
{
  // Constructors
  public President ( string name ) : base( name ) {}
  // Methods
  override public void ProcessRequest(
                            PurchaseRequest request )
  {
    if( request.Amount < 100000.0 )
      Console.WriteLine( "{0} {1} approved request# {2}",
        this, name, request.Number);
    else
      Console.WriteLine( "Request# {0} requires " +
        "an executive meeting!", request.Number );
  }
}

// Request details

class PurchaseRequest
```

```
{
  // Member Fields
  private int number;
  private double amount;
  private string purpose;

  // Constructors
  public PurchaseRequest( int number,
            double amount, string purpose )
  {
    this.number = number;
    this.amount = amount;
    this.purpose = purpose;
  }

  // Properties
  public double Amount
  {
    get{ return amount; }
    set{ amount = value; }
  }

  public string Purpose
  {
    get{ return purpose; }
    set{ purpose = value; }
  }

  public int Number
  {
    get{ return number; }
    set{ number = value; }
  }
}

/// <summary>
///  ChainApp Application
/// </summary>
public class ChainApp
{
  public static void Main( string[] args )
  {
    // Setup Chain of Responsibility
    Director Larry = new Director( "Larry" );
    VicePresident Sam = new VicePresident( "Sam" );
    President Tammy = new President( "Tammy" );
    Larry.SetSuccessor( Sam );
    Sam.SetSuccessor( Tammy );

    // Generate and process different requests
    PurchaseRequest rs = new PurchaseRequest(
                      2034, 350.00, "Supplies" );
    Larry.ProcessRequest( rs );

    PurchaseRequest rx = new PurchaseRequest(
                      2035, 32590.10, "Project X" );
    Larry.ProcessRequest( rx );
```

```
    PurchaseRequest ry = new PurchaseRequest(
                        2036, 122100.00, "Project Y" );
    Larry.ProcessRequest( ry );

  }
}
```

State

- umožní objektu zmeniť svoje správanie v závislosti od jeho vnútorného stavu
- vzor správania sa




```
// State pattern -- Structural example

using System;

// "State"

abstract class State
{
  // Methods
  abstract public void Handle( Context context );
}

// "ConcreteStateA"

class ConcreteStateA : State
{
  // Methods
  override public void Handle( Context context )
  {
    context.State = new ConcreteStateB();
  }
}

// "ConcreteStateB"

class ConcreteStateB : State
{
  // Methods
  override public void Handle( Context context )
  {
    context.State = new ConcreteStateA();
  }
}
```

```
// "Context"

class Context
{
  // Fields
  private State state;

  // Constructors
  public Context( State state )
  {
    this.state = state;
  }

  // Properties
  public State State
  {
    get{ return state; }
    set{ state = value; }
  }

  // Methods
  public void Request()
  {
    state.Handle( this );
  }

  public void Show()
  {
    Console.WriteLine( "State: " + state );
  }
}

/// <summary>
/// Client test
/// </summary>
public class Client
{
  public static void Main( string[] args )
  {
    // Setup context in a state
    Context c = new Context( new ConcreteStateA() );
    c.Show();

    // Issue request, which toggles state
    c.Request();
    c.Show();

    // Issue request, which toggles state
    c.Request();
    c.Show();
  }
}
```

```
// State pattern -- Real World example

using System;
```

```
// "State"

abstract class State
{
  // Fields
  protected Account account;
  protected double balance;
  protected double interest;
  protected double lowerLimit;
  protected double upperLimit;

  // Properties
  public Account Account
  {
    get{ return account; }
    set{ account = value; }
  }
  public double Balance
  {
    get{ return balance; }
    set{ balance = value; }
  }

  // Methods
  abstract public void Initialize();
  abstract public void Deposit( double amount );
  abstract public void Withdraw( double amount );
  abstract public void PayInterest();
  abstract public void StateChangeCheck();
}

// "ConcreteState"
// Account is overdrawn

class RedState : State
{
  // Fields
  double serviceFee;

  // Constructors
  public RedState( State state )
  {
    this.balance = state.Balance;
    this.account = state.Account;
    Initialize();
  }

  // Methods
  override public void Initialize()
  {
    // Should come from a database
    interest = 0.0;
    lowerLimit = -100.0;
    upperLimit = 0.0;
    serviceFee = 15.00;
  }
```

```
  override public void Deposit( double amount )
  {
    balance += amount;
    StateChangeCheck();
  }

  override public void Withdraw( double amount )
  {
    amount = amount - serviceFee;
    Console.WriteLine(
              "No funds available to withdraw!" );
  }

  override public void PayInterest()
  {
    // No interest is paid
  }

  override public void StateChangeCheck()
  {
    if( balance > upperLimit )
      account.State = new SilverState( this );
  }
}

// "ConcreteState"
// Silver is non-interest bearing state

class SilverState : State
{
  // Constructors
  public SilverState(
            double balance, Account account )
  {
    this.balance = balance;
    this.account = account;
    Initialize();
  }

  public SilverState( State state )
  {
    this.balance = state.Balance;
    this.account = state.Account;
    Initialize();
  }

  // Methods
  override public void Initialize()
  {
    // Should come from a database
    interest = 0.0;
    lowerLimit = 0.0;
    upperLimit = 1000.0;
  }

  override public void Deposit( double amount )
```

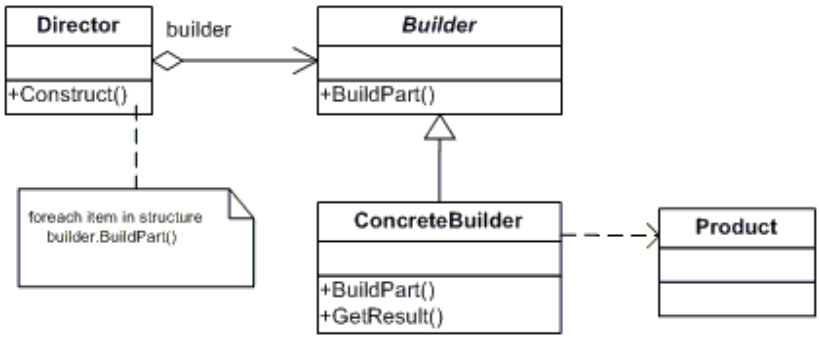
```
  {
    balance += amount;
    StateChangeCheck();
  }

  override public void Withdraw( double amount )
  {
    balance -= amount;
    StateChangeCheck();
  }

  override public void PayInterest()
  {
    balance += interest * balance;
    StateChangeCheck();
  }

  override public void StateChangeCheck()
  {
    if( balance < lowerLimit )
      account.State = new RedState( this );
    else if( balance > upperLimit )
      account.State = new GoldState( this );
  }
}

// "ConcreteState"
// Interest bearing state

class GoldState : State
{
  // Constructors
  public GoldState(
          double balance, Account account )
  {
    this.balance = balance;
    this.account = account;
    Initialize();
  }

  public GoldState( State state )
  {
    this.balance = state.Balance;
    this.account = state.Account;
    Initialize();
  }

  // Methods
  override public void Initialize()
  {
    // Should come from a database
    interest = 0.05;
    lowerLimit = 1000.0;
    upperLimit = 10000000.0;
  }

  override public void Deposit( double amount )
```

```
  {
    balance += amount;
    StateChangeCheck();
  }

  override public void Withdraw( double amount )
  {
    balance -= amount;
    StateChangeCheck();
  }

  override public void PayInterest()
  {
    balance += interest * balance;
    StateChangeCheck();
  }

  override public void StateChangeCheck()
  {
    if( balance < 0.0 )
      account.State = new RedState( this );
    else if( balance < lowerLimit )
      account.State = new SilverState( this );
  }
}

// "Context"

class Account
{
  // Fields
  private State state;
  private string owner;

  // Constructors
  public Account( string owner )
  {
    // New accounts are 'Silver' by default
    this.owner = owner;
    state = new SilverState( 0.0, this );
  }

  // Properties
  public double Balance
  {
    get{ return state.Balance; }
  }
  public State State
  {
    get{ return state; }
    set{ state = value; }
  }

  // Methods
  public void Deposit( double amount )
  {
    state.Deposit( amount );
```

```
    Console.WriteLine( "Deposited {0:C} --- ",
                                    amount);
    Console.WriteLine( " Balance = {0:C}",
                                    this.Balance );
    Console.WriteLine( " Status = {0}" ,
                                    this.State );
    Console.WriteLine( "" );
  }

  public void Withdraw( double amount )
  {
    state.Withdraw( amount );
    Console.WriteLine( "Withdrew {0:C} --- ",
                                    amount);
    Console.WriteLine( " Balance = {0:C}",
                                    this.Balance );
    Console.WriteLine( " Status = {0}" ,
                                    this.State );
    Console.WriteLine( "" );
  }

  public void PayInterest()
  {
    state.PayInterest();
    Console.WriteLine( "Interest Paid --- ");
    Console.WriteLine( " Balance = {0:C}",
                                    this.Balance );
    Console.WriteLine( " Status = {0}" ,
                                    this.State );
    Console.WriteLine( "" );
  }
}

/// <summary>
/// StateApp test
/// </summary>
public class StateApp
{
  public static void Main( string[] args )
  {
    // Open a new account
    Account account = new Account( "Ana Micola" );

    // Apply financial transactions
    account.Deposit( 500.0 );
    account.Deposit( 300.0 );
    account.Deposit( 550.0 );
    account.PayInterest();
    account.Withdraw( 2000.00 );
    account.Withdraw( 1100.00 );

  }
}
```

Builder

- oddelí konštrukciu zložitého objektu od jeho reprezentácie, takže rovnaký konštrukčný proces môže vytvoriť rôzne produkty
- vzor pre tvorbu




```
// Builder pattern -- Structural example

using System;
using System.Collections;

// "Director"

class Director
{
  // Methods
  public void Construct( Builder builder )
  {
    builder.BuildPartA();
    builder.BuildPartB();
  }
}

// "Builder"

abstract class Builder
{
  // Methods
  abstract public void BuildPartA();
  abstract public void BuildPartB();
  abstract public Product GetResult();
}

// "ConcreteBuilder1"

class ConcreteBuilder1 : Builder
{
  // Fields
  private Product product;
```

```
  // Methods
  override public void BuildPartA()
  {
    product = new Product();
    product.Add( "PartA" );
  }

  override public void BuildPartB()
  {
    product.Add( "PartB" );
  }

  override public Product GetResult()
  {
    return product;
  }
}

// "ConcreteBuilder2"

class ConcreteBuilder2 : Builder
{
  // Fields
  private Product product;

  // Methods
  override public void BuildPartA()
  {
    product = new Product();
    product.Add( "PartX" );
  }

  override public void BuildPartB()
  {
    product.Add( "PartY" );
  }

  override public Product GetResult()
  {
    return product;
  }
}

// "Product"

class Product
{
  // Fields
  ArrayList parts = new ArrayList();

  // Methods
  public void Add( string part )
  {
    parts.Add( part );
  }

  public void Show()
```

```
  {
    Console.WriteLine( "\nProduct Parts -------" );
    foreach( string part in parts )
      Console.WriteLine( part );
  }
}
/// <summary>
/// Client test
/// </summary>
public class Client
{
  public static void Main( string[] args )
  {
    // Create director and builders
    Director director = new Director( );

    Builder b1 = new ConcreteBuilder1();
    Builder b2 = new ConcreteBuilder2();

    // Construct two products
    director.Construct( b1 );
    Product p1 = b1.GetResult();
    p1.Show();

    director.Construct( b2 );
    Product p2 = b2.GetResult();
    p2.Show();
  }
}
```

```
// Builder pattern -- Real World example

using System;
using System.Collections;

// "Director"

class Shop
{
  // Methods
  public void Construct( VehicleBuilder vehicleBuilder )
  {
    vehicleBuilder.BuildFrame();
    vehicleBuilder.BuildEngine();
    vehicleBuilder.BuildWheels();
    vehicleBuilder.BuildDoors();
  }
}

// "Builder"

abstract class VehicleBuilder
{
  // Fields
  protected Vehicle vehicle;
```

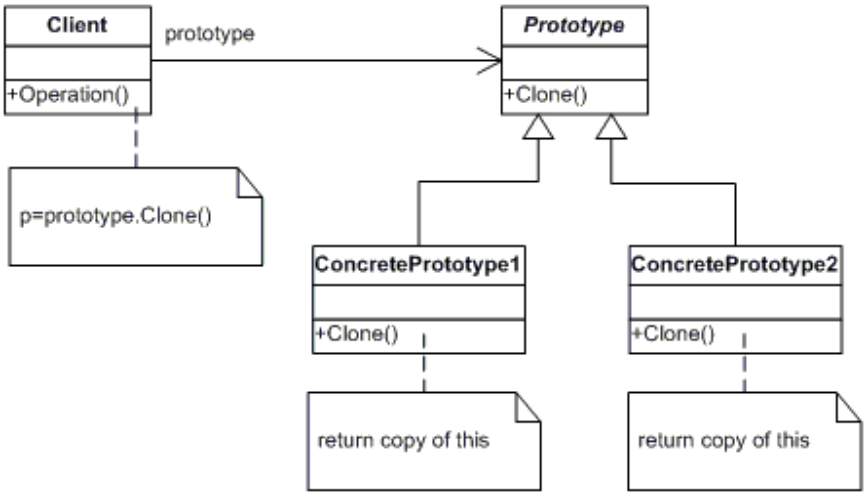
```
  // Properties
  public Vehicle Vehicle
  {
    get{ return vehicle; }
  }

  // Methods
  abstract public void BuildFrame();
  abstract public void BuildEngine();
  abstract public void BuildWheels();
  abstract public void BuildDoors();
}

// "ConcreteBuilder1"

class MotorCycleBuilder : VehicleBuilder
{
  // Methods
  override public void BuildFrame()
  {
    vehicle = new Vehicle( "MotorCycle" );
    vehicle[ "frame" ] = "MotorCycle Frame";
  }

  override public void BuildEngine()
  {
    vehicle[ "engine" ] = "500 cc";
  }

  override public void BuildWheels()
  {
    vehicle[ "wheels" ] = "2";
  }

  override public void BuildDoors()
  {
    vehicle[ "doors" ] = "0";
  }
}

// "ConcreteBuilder2"

class CarBuilder : VehicleBuilder
{
  // Methods
  override public void BuildFrame()
  {
    vehicle = new Vehicle( "Car" );
    vehicle[ "frame" ] = "Car Frame";
  }

  override public void BuildEngine()
  {
    vehicle[ "engine" ] = "2500 cc";
  }
```

```
  override public void BuildWheels()
  {
    vehicle[ "wheels" ] = "4";
  }

  override public void BuildDoors()
  {
    vehicle[ "doors" ] = "4";
  }
}

// "ConcreteBuilder3"

class ScooterBuilder : VehicleBuilder
{
  // Methods
  override public void BuildFrame()
  {
    vehicle = new Vehicle( "Scooter" );
    vehicle[ "frame" ] = "Scooter Frame";
  }

  override public void BuildEngine()
  {
    vehicle[ "engine" ] = "none";
  }

  override public void BuildWheels()
  {
    vehicle[ "wheels" ] = "2";
  }

  override public void BuildDoors()
  {
    vehicle[ "doors" ] = "0";
  }
}

// "Product"

class Vehicle
{
  // Fields
  private string type;
  private Hashtable parts = new Hashtable();

  // Constructors
  public Vehicle( string type )
  {
    this.type = type;
  }

  // Indexers
  public object this[ string key ]
  {
    get{ return parts[ key ]; }
```

```
    set{ parts[ key ] = value; }
  }

  // Methods
  public void Show()
  {
    Console.WriteLine( "\n---------------------------");
    Console.WriteLine( "Vehicle Type: "+ type );
    Console.WriteLine( " Frame : " + parts[ "frame" ] );
    Console.WriteLine( " Engine : "+ parts[ "engine"] );
    Console.WriteLine( " #Wheels: "+ parts[ "wheels"] );
    Console.WriteLine( " #Doors : "+ parts[ "doors" ] );
  }
}

/// <summary>
/// BuilderApp test
/// </summary>
public class BuilderApp
{
  public static void Main( string[] args )
  {
    // Create shop and vehicle builders
    Shop shop = new Shop();
    VehicleBuilder b1 = new ScooterBuilder();
    VehicleBuilder b2 = new CarBuilder();
    VehicleBuilder b3 = new MotorCycleBuilder();

    // Construct and display vehicles
    shop.Construct( b1 );
    b1.Vehicle.Show();

    shop.Construct( b2 );
    b2.Vehicle.Show();

    shop.Construct( b3 );
    b3.Vehicle.Show();
  }
}
```

Prototype

- definuje druh objektu, ktorý treba vytvárať, pomocou prototypickej inštancie. Potom na požiadanie vytvorí objekt klonovaním tejto inštancie.
- vzor pre tvorbu




```
// Prototype pattern -- Structural example

using System;

// "Prototype"

abstract class Prototype
{
  // Fields
  private string id;

  // Constructors
  public Prototype( string id )
  {
    this.id = id;
  }

  public string Id
  {
    get{ return id; }
  }

  // Methods
  abstract public Prototype Clone();
}

// "ConcretePrototype1"

class ConcretePrototype1 : Prototype
```

```
{
  // Constructors
  public ConcretePrototype1( string id ) : base ( id ) {}

  // Methods
  override public Prototype Clone()
  {
    // Shallow copy
    return (Prototype)this.MemberwiseClone();
  }
}

// "ConcretePrototype2"

class ConcretePrototype2 : Prototype
{
  // Constructors
  public ConcretePrototype2( string id ) : base ( id ) {}

  // Methods
  override public Prototype Clone()
  {
    // Shallow copy
    return (Prototype)this.MemberwiseClone();
  }
}

/// <summary>
/// Client test
/// </summary>
class Client
{
  public static void Main( string[] args )
  {
   // Create two instances and clone each
   ConcretePrototype1 p1 = new ConcretePrototype1( "I" );
   ConcretePrototype1 c1 = (ConcretePrototype1)p1.Clone();
   Console.WriteLine( "Cloned: {0}", c1.Id );

   ConcretePrototype2 p2 = new ConcretePrototype2( "II" );
   ConcretePrototype2 c2 = (ConcretePrototype2)p2.Clone();
   Console.WriteLine( "Cloned: {0}", c2.Id );
  }
}
```

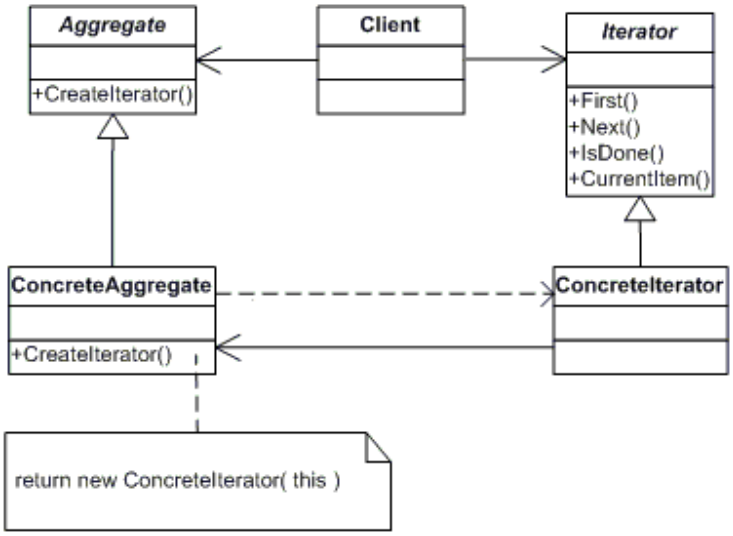
```
// Prototype pattern -- Real World example

using System;
using System.Collections;

// "Prototype"

abstract class ColorPrototype
{
```

```
  // Methods
  public abstract ColorPrototype Clone();
}

// "ConcretePrototype"

class Color : ColorPrototype
{
  // Fields
  private int red, green, blue;

  // Constructors
  public Color( int red, int green, int blue)
  {
    this.red = red;
    this.green = green;
    this.blue = blue;
  }

  // Methods
  public override ColorPrototype Clone()
  {
    // Creates a 'shallow copy'
    return (ColorPrototype) this.MemberwiseClone();
  }

  public void Display()
  {
    Console.WriteLine( "RGB values are: {0},{1},{2}",
                                    red, green, blue );
  }
}

// Prototype manager

class ColorManager
{
  // Fields
  Hashtable colors = new Hashtable();

  // Indexers
  public ColorPrototype this[ string name ]
  {
    get{ return (ColorPrototype)colors[ name ]; }
    set{ colors.Add( name, value ); }
  }
}

/// <summary>
///  PrototypeApp test
/// </summary>
class PrototypeApp
{
  public static void Main( string[] args )
  {
     ColorManager colormanager = new ColorManager();
```

```
    // Initialize with standard colors
    colormanager[ "red" ] = new Color( 255, 0, 0 );
    colormanager[ "green" ] = new Color( 0, 255, 0 );
    colormanager[ "blue" ] = new Color( 0, 0, 255 );

    // User adds personalized colors
    colormanager[ "angry" ] = new Color( 255, 54, 0 );
    colormanager[ "peace" ] = new Color( 128, 211, 128 );
    colormanager[ "flame" ] = new Color( 211, 34, 20 );

    // User uses selected colors
    string colorName = "red";
    Color c1 = (Color)colormanager[ colorName ].Clone();
    c1.Display();

    colorName = "peace";
    Color c2 = (Color)colormanager[ colorName ].Clone();
    c2.Display();

    colorName = "flame";
    Color c3 = (Color)colormanager[ colorName ].Clone();
    c3.Display();
  }
}
```

Iterator

- poskytuje spôsob, ako pristupovať ku elementom agregujúceho objektu bez toho, aby sme museli poznať (a boli závislí na) vnútornú štruktúru
- vzor správania sa




```
// Iterator pattern -- Structural example

using System;
using System.Collections;

// "Aggregate"

abstract class Aggregate
{
  // Methods
  public abstract Iterator CreateIterator();
}

// "ConcreteAggregate"

class ConcreteAggregate : Aggregate
{
  // Fields
  private ArrayList items = new ArrayList();

  // Methods
  public override Iterator CreateIterator()
  {
    return new ConcreteIterator( this );
```

```
  }

  // Properties
  public int Count
  {
    get{ return items.Count; }
  }

  // Indexers
  public object this[ int index ]
  {
    get{ return items[ index ]; }
    set{ items.Insert( index, value ); }
  }
}

// "Iterator"

abstract class Iterator
{
  // Methods
  public abstract object First();
  public abstract object Next();
  public abstract bool IsDone();
  public abstract object CurrentItem();
}

// "ConcreteIterator"

class ConcreteIterator : Iterator
{
  // Fields
  private ConcreteAggregate aggregate;
  private int current = 0;

  // Constructor
  public ConcreteIterator( ConcreteAggregate aggregate )
  {
    this.aggregate = aggregate;
  }

  // Methods
  override public object First()
  {
    return aggregate[ 0 ];
  }

  override public object Next()
  {
    if( current < aggregate.Count-1 )
      return aggregate[ ++current ];
    else
      return null;
  }

  override public object CurrentItem()
  {
```

```
    return aggregate[ current ];
  }

  override public bool IsDone()
  {
    return current >= aggregate.Count ? true : false ;
  }
}

/// <summary>
///  Client test
/// </summary>
public class Client
{
  public static void Main(string[] args)
  {
    ConcreteAggregate a = new ConcreteAggregate();
    a[0] = "Item A";
    a[1] = "Item B";
    a[2] = "Item C";
    a[3] = "Item D";

    // Create Iterator and provide aggregate
    ConcreteIterator i = new ConcreteIterator(a);

    // Iterate over collection
    object item = i.First();

    while( item != null )
    {
      Console.WriteLine( item );
      item = i.Next();
    }
  }
}
```

```
// Iterator pattern -- Real World example

using System;
using System.Collections;
class Item
{
  // Fields
  string name;

  // Constructors
  public Item( string name )
  {
    this.name = name;
  }

  // Properties
  public string Name
  {
    get{ return name; }
```

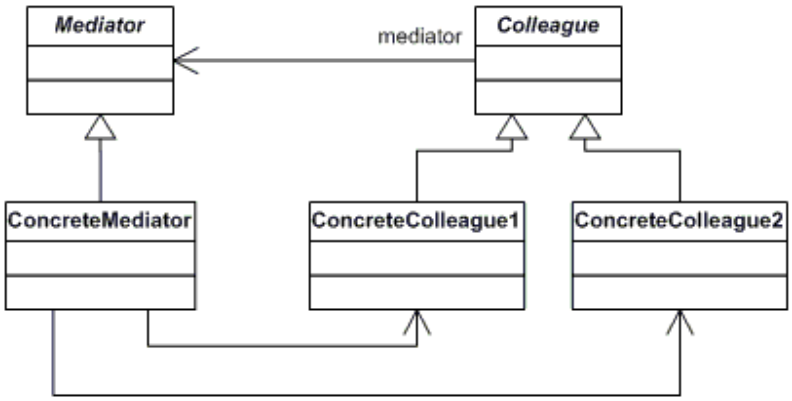
```
  }
}

// "Aggregate"

abstract class AbstractCollection
{
  abstract public Iterator CreateIterator();
}

// "ConcreteAggregate"

class Collection : AbstractCollection
{
  // Fields
  private ArrayList items = new ArrayList();

  // Methods
  public override Iterator CreateIterator()
  {
    return new Iterator( this );
  }

  // Properties
  public int Count
  {
    get{ return items.Count; }
  }

  // Indexers
  public object this[ int index ]
  {
    get{ return items[ index ]; }
    set{ items.Add( value ); }
  }
}

// "Iterator"

abstract class AbstractIterator
{
  // Methods
  abstract public Item First();
  abstract public Item Next();
  abstract public bool IsDone();
  abstract public Item CurrentItem();
}

// "ConcreteIterator"

class Iterator : AbstractIterator
{
  // Fields
  private Collection collection;
  private int current = 0;
  private int step = 1;
```

```
  // Constructor
  public Iterator( Collection collection )
  {
    this.collection = collection;
  }

  // Properties
  public int Step
  {
    get{ return step; }
    set{ step = value; }
  }

  // Methods
  override public Item First()
  {
    current = 0;
    return (Item)collection[ current ];
  }

  override public Item Next()
  {
    current += step;
    if( !IsDone() )
      return (Item)collection[ current ];
    else
      return null;
  }

  override public Item CurrentItem()
  {
    return (Item)collection[ current ];
  }

  override public bool IsDone()
  {
    return current >= collection.Count ? true : false ;
  }
}

/// <summary>
/// IteratorApp test
/// </summary>
public class IteratorApp
{
  public static void Main(string[] args)
  {
    // Build a collection
    Collection collection = new Collection();
    collection[0] = new Item( "Item 0" );
    collection[1] = new Item( "Item 1" );
    collection[2] = new Item( "Item 2" );
    collection[3] = new Item( "Item 3" );
    collection[4] = new Item( "Item 4" );
    collection[5] = new Item( "Item 5" );
    collection[6] = new Item( "Item 6" );
    collection[7] = new Item( "Item 7" );
```

```
    collection[8] = new Item( "Item 8" );

    // Create iterator
    Iterator iterator = new Iterator( collection );

    // Skip every other item
    iterator.Step = 2;

    // For loop using iterator
    for( Item item = iterator.First();
         !iterator.IsDone(); item = iterator.Next() )
    {
      Console.WriteLine( item.Name );
    }
  }
}
```

Mediator

- definuje objekt, ktorý v sebe ukrýva spôsob komunikácie medzi množinou iných objektov. Zabezpečuje slabú väzbu tým, že komunikujúce objekty sa navzájom priamo neodkazujú.
- vzor správania sa




```
// Mediator pattern -- Structural example

using System;
using System.Collections;

// "Mediator"

abstract class Mediator
{
  // Methods
  abstract public void Send( string message,
  Colleague colleague );
}

// "ConcreteMediator"

class ConcreteMediator : Mediator
{
  // Fields
  private ConcreteColleague1 colleague1;
  private ConcreteColleague2 colleague2;

  // Properties
  public ConcreteColleague1 Colleague1
  {
    set{ colleague1 = value; }
  }
```

```
  public ConcreteColleague2 Colleague2
  {
    set{ colleague2 = value; }
  }

  // Methods
  override public void Send( string message,
                             Colleague colleague )
  {
    if( colleague == colleague1 )
      colleague2.Notify( message );
    else
      colleague1.Notify( message );
  }
}

// "Colleague"

abstract class Colleague
{
  // Fields
  protected Mediator mediator;

  // Constructors
  public Colleague( Mediator mediator )
  {
    this.mediator = mediator;
  }
}

// "ConcreteColleague1"

class ConcreteColleague1 : Colleague
{
  // "Constructors"
  public ConcreteColleague1( Mediator mediator )
                                    : base ( mediator ) { }

  // Methods
  public void Send( string message )
  {
    mediator.Send( message, this );
  }

  public void Notify( string message )
  {
    Console.WriteLine( "Colleague1 gets message: "
                                             + message );
  }
}

// "ConcreteColleague2"

class ConcreteColleague2 : Colleague
{
  // Constructors
```

```
  public ConcreteColleague2( Mediator mediator )
                                  : base ( mediator ) { }

  // Methods
  public void Send( string message )
  {
    mediator.Send( message, this );
  }
  public void Notify( string message )
  {
    Console.WriteLine( "Colleague2 gets message: "
                                           + message );
  }
}

/// <summary>
/// Client test
/// </summary>
public class Client
{
  public static void Main(string[] args)
  {
    ConcreteMediator m = new ConcreteMediator();
    ConcreteColleague1 c1 = new ConcreteColleague1( m );
    ConcreteColleague2 c2 = new ConcreteColleague2( m );

    m.Colleague1 = c1;
    m.Colleague2 = c2;

    c1.Send( "How are you?" );
    c2.Send( "Fine, thanks" );
  }
}
```

```
// Mediator pattern -- Real World example

using System;
using System.Collections;

// "Mediator"

interface IChatroom
{
  // Methods
  void Register( Participant participant );
  void Send( string from, string to, string message );
}

// "ConcreteMediator"

class Chatroom : IChatroom
{
  // Fields
  private Hashtable participants = new Hashtable();

  // Methods
```

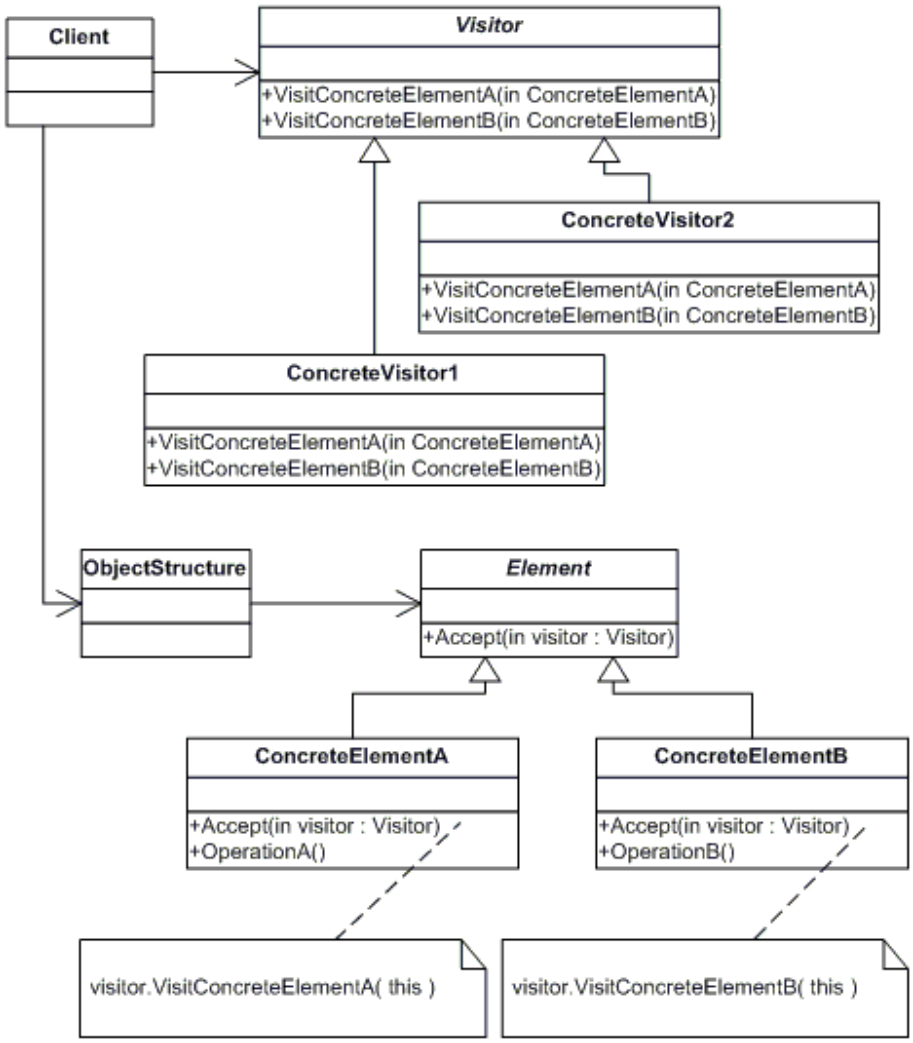
```
  public void Register( Participant participant )
  {
    if( participants[ participant.Name ] == null )
      participants[ participant.Name ] = participant;

    participant.Chatroom = this;
  }

  public void Send( string from, string to, string message )
  {
    Participant pto = (Participant)participants[ to ];
    if( pto != null )
      pto.Receive( from, message );
  }
}

// "AbstractColleague"

class Participant
{
  // Fields
  private Chatroom chatroom;
  private string name;

  // Constructors
  public Participant( string name )
  {
    this.name = name;
  }

  // Properties
  public string Name
  {
    get{ return name; }
  }

  public Chatroom Chatroom
  {
    set{ chatroom = value; }
    get{ return chatroom; }
  }

  // Methods
  public void Send( string to, string message )
  {
     chatroom.Send( name, to, message );
  }

  virtual public void Receive(
                        string from, string message )
  {
            Console.WriteLine( "{0} to {1}: '{2}'",
                           from, this.name, message );
  }
}

//" ConcreteColleague1"
```

```
class BeatleParticipant : Participant
{
  // Constructors
  public BeatleParticipant( string name )
                                    : base ( name ) { }

  override public void Receive(
                           string from, string message )
  {
    Console.Write( "To a Beatle: " );
    base.Receive( from, message );
  }
}

//" ConcreteColleague2"

class NonBeatleParticipant : Participant
{
  // Constructors
  public NonBeatleParticipant( string name )
                                    : base ( name ) { }

  override public void Receive(
                           string from, string message )
  {
    Console.Write( "To a non-Beatle: " );
    base.Receive( from, message );
  }
}

/// <summary>
/// MediatorApp test
/// </summary>
public class MediatorApp
{
  public static void Main(string[] args)
  {
    // Create chatroom
    Chatroom c = new Chatroom();

    // Create 'chatters' and register them
    Participant George = new BeatleParticipant("George");
    Participant Paul = new BeatleParticipant("Paul");
    Participant Ringo = new BeatleParticipant("Ringo");
    Participant John = new BeatleParticipant("John") ;
    Participant Yoko = new NonBeatleParticipant("Yoko");

    c.Register( George );
    c.Register( Paul );
    c.Register( Ringo );
    c.Register( John );
    c.Register( Yoko );

    // Chatting participants
    Yoko.Send( "John", "Hi John!" );
    Paul.Send( "Ringo", "All you need is love" );
```

```
    Ringo.Send( "George", "My sweet Lord" );
    Paul.Send( "John", "Can't buy me love" );
    John.Send( "Yoko", "My sweet love" ) ;
  }
}
```

Visitor

- reprezentuje operáciu, ktorá má byť vykonaná na elementoch nejakej štruktúry. Visitor umožní definovať túto operáciu bez zmeny elementov, na ktorých je vykonávaná jeho operácia.
- vzor správania sa




```
// Visitor pattern -- Structural example

using System;
using System.Collections;

// "Visitor"
```

```
abstract class Visitor
{
  // Methods
  abstract public void VisitConcreteElementA(
                  ConcreteElementA concreteElementA );
  abstract public void VisitConcreteElementB(
                   ConcreteElementB concreteElementB );
}

// "ConcreteVisitor1"

class ConcreteVisitor1 : Visitor
{
  // Methods
  override public void VisitConcreteElementA(
                   ConcreteElementA concreteElementA )
  {
    Console.WriteLine( "{0} visited by {1}",
                             concreteElementA, this );
  }

  override public void VisitConcreteElementB(
                   ConcreteElementB concreteElementB )
  {
    Console.WriteLine( "{0} visited by {1}",
                             concreteElementB, this );
  }
}

// "ConcreteVisitor2"

class ConcreteVisitor2 : Visitor
{
  // Methods
  override public void VisitConcreteElementA(
                  ConcreteElementA concreteElementA )
  {
    Console.WriteLine( "{0} visited by {1}",
                             concreteElementA, this );
  }
  override public void VisitConcreteElementB(
                  ConcreteElementB concreteElementB )
  {
    Console.WriteLine( "{0} visited by {1}",
                             concreteElementB, this );
  }
}

// "Element"

abstract class Element
{
  // Methods
  abstract public void Accept( Visitor visitor );
}

// "ConcreteElementA"
```

```
class ConcreteElementA : Element
{
  // Methods
  override public void Accept( Visitor visitor )
  {
    visitor.VisitConcreteElementA( this );
  }

  public void OperationA()
  {
  }
}

// "ConcreteElementB"

class ConcreteElementB : Element
{
  // Methods
  override public void Accept( Visitor visitor )
  {
    visitor.VisitConcreteElementB( this );
  }

  public void OperationB()
  {
  }
}

// "ObjectStructure"

class ObjectStructure
{
  // Fields
  private ArrayList elements = new ArrayList();

  // Methods
  public void Attach( Element element )
  {
    elements.Add( element );
  }

  public void Detach( Element element )
  {
    elements.Remove( element );
  }

  public void Accept( Visitor visitor )
  {
    foreach( Element e in elements )
      e.Accept( visitor );
  }
}

/// <summary>
/// Client test
/// </summary>
public class Client
```

```
{
  public static void Main( string[] args )
  {
    // Setup structure
    ObjectStructure o = new ObjectStructure();
    o.Attach( new ConcreteElementA() );
    o.Attach( new ConcreteElementB() );

    // Create visitor objects
    ConcreteVisitor1 v1 = new ConcreteVisitor1();
    ConcreteVisitor2 v2 = new ConcreteVisitor2();

    // Structure accepting visitors
    o.Accept( v1 );
    o.Accept( v2 );
  }
}
```

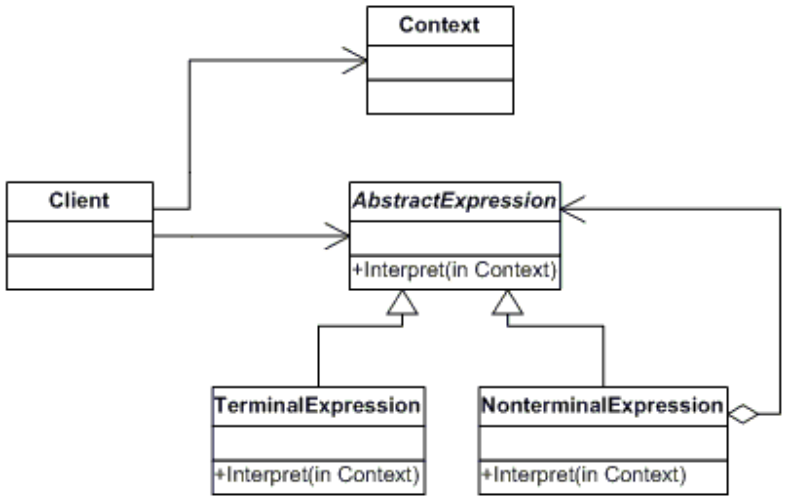
```
// Visitor pattern -- Real World example

using System;
using System.Collections;

// "Visitor"

abstract class Visitor
{
  // Methods
  abstract public void Visit( Element element );
}

// "ConcreteVisitor1"

class IncomeVisitor : Visitor
{
  // Methods
  public override void Visit( Element element )
  {
    Employee employee = ((Employee)element);

    // Provide 10% pay raise
    employee.Income *= 1.10;
    Console.WriteLine( "{0}'s new income: {1:C}",
                employee.Name, employee.Income );
  }
}

// "ConcreteVisitor2"

class VacationVisitor : Visitor
{
  public override void Visit( Element element )
  {
    Employee employee = ((Employee)element);

    // Provide 3 extra vacation days
```

```
    employee.VacationDays += 3;
    Console.WriteLine( "{0}'s new vacation days: {1}",
              employee.Name, employee.VacationDays );
  }
}

// "Element"

abstract class Element
{
  // Methods
  abstract public void Accept( Visitor visitor );
}

// "ConcreteElement"

class Employee : Element
{
  // Fields
  string name;
  double income;
  int vacationDays;

  // Constructors
  public Employee( string name, double income,
                                    int vacationDays )
  {
    this.name = name;
    this.income = income;
    this.vacationDays = vacationDays;
  }

  // Properties
  public string Name
  {
    get{ return name; }
    set{ name = value; }
  }

  public double Income
  {
    get{ return income; }
    set{ income = value; }
  }

  public int VacationDays
  {
    get{ return vacationDays; }
    set{ vacationDays = value; }
  }

  // Methods
  public override void Accept( Visitor visitor )
  {
    visitor.Visit( this );
  }
}
```

```
// "ObjectStructure"

class Employees
{
  // Fields
  private ArrayList employees = new ArrayList();

  // Methods
  public void Attach( Employee employee )
  {
    employees.Add( employee );
  }

  public void Detach( Employee employee )
  {
    employees.Remove( employee );
  }

  public void Accept( Visitor visitor )
  {
    foreach( Employee e in employees )
      e.Accept( visitor );
  }
}

/// <summary>
/// VisitorApp test
/// </summary>
public class VisitorApp
{
  public static void Main( string[] args )
  {
    // Setup employee collection
    Employees e = new Employees();
     e.Attach( new Employee( "Hank", 25000.0, 14 ) );
    e.Attach( new Employee( "Elly", 35000.0, 16 ) );
    e.Attach( new Employee( "Dick", 45000.0, 21 ) );

    // Create two visitors
    IncomeVisitor v1 = new IncomeVisitor();
    VacationVisitor v2 = new VacationVisitor();

    // Employees are visited
    e.Accept( v1 );
    e.Accept( v2 );
  }
}
```

Interpreter

- pre daný jazyk definuje reprezentáciu pre jeho gramatiku
- tiež definuje interpreter, ktorý používa túto reprezentáciu na interpetáciu slov v jazyku
- vzor správania sa




```
// Interpreter pattern -- Structural example

using System;
using System.Collections;

// "Context"

class Context
{
}

// "AbstractExpression"

abstract class AbstractExpression
{
  public abstract void Interpret( Context context );
}

// "TerminalExpression"

class TerminalExpression : AbstractExpression
{
  public override void Interpret( Context context )
  {
    Console.WriteLine( "Called Terminal.Interpret()" );
  }
}
```

```
// "NonterminalExpression"

class NonterminalExpression : AbstractExpression
{
  public override void Interpret( Context context )
  {
    Console.WriteLine( "Called Nonterminal.Interpret()" );
  }
}

/// <summary>
/// Client test
/// </summary>
public class Client
{
  public static void Main( string[] args )
  {
    Context c = new Context();

    // Usually a tree
    ArrayList l = new ArrayList();

    // Populate 'abstract syntax tree'
    l.Add(new TerminalExpression());
    l.Add(new NonterminalExpression());
    l.Add(new TerminalExpression());
    l.Add(new TerminalExpression());

    // Interpret
    foreach( AbstractExpression exp in l )
      exp.Interpret(c);
  }
}
```

```
// Interpreter pattern -- Real World example

using System;
using System.Text;
using System.Collections;

// "Context"

class Context
{
  // Fields
  private string input;
  private int output;

  // Constructors
  public Context( string input )
  {
    this.input = input;
  }
```

```
  // Properties
  public string Input
  {
    get{ return input; }
    set{ input = value; }
  }

  public int Output
  {
    get{ return output; }
    set{ output = value; }
  }
}

// "AbstractExpression"

abstract class Expression
{
  // Methods
  public void Interpret( Context context )
  {
    if( context.Input.Length == 0 ) return;
    if( context.Input.StartsWith( Nine() ) )
    {
      context.Output += 9 * Multiplier();
      context.Input = context.Input.Substring(2);
    }
    else if( context.Input.StartsWith( Four() ) )
    {
      context.Output += 4 * Multiplier();
      context.Input = context.Input.Substring( 2);
    }
    else if( context.Input.StartsWith( Five() ) )
    {
      context.Output += 5 * Multiplier();
      context.Input = context.Input.Substring( 1);
    }
    while( context.Input.StartsWith( One() ) )
    {
      context.Output += 1 * Multiplier();
      context.Input = context.Input.Substring( 1 );
    }
  }

  public abstract string One();
  public abstract string Four();
  public abstract string Five();
  public abstract string Nine();
  public abstract int Multiplier();
}

// Thousand checks for the Roman Numeral M
// "TerminalExpression"

class ThousandExpression : Expression
{
```

```
  // Methods
  public override string One() { return "M"; }
  public override string Four(){ return " "; }
  public override string Five(){ return " "; }
  public override string Nine(){ return " "; }
  public override int Multiplier() { return 1000; }
}

// Hundred checks C, CD, D or CM
// "TerminalExpression"

class HundredExpression : Expression
{
  // Methods
  public override string One() { return "C"; }
  public override string Four(){ return "CD"; }
  public override string Five(){ return "D"; }
  public override string Nine(){ return "CM"; }
  public override int Multiplier() { return 100; }
}

// Ten checks for X, XL, L and XC
// "TerminalExpression"

class TenExpression : Expression
{
  // Methods
  public override string One() { return "X"; }
  public override string Four(){ return "XL"; }
  public override string Five(){ return "L"; }
  public override string Nine(){ return "XC"; }
  public override int Multiplier() { return 10; }
}

// One checks for I, II, III, IV, V, VI, VII, VIII, IX
// "TerminalExpression"

class OneExpression : Expression
{
  // Methods
  public override string One() { return "I"; }
  public override string Four(){ return "IV"; }
  public override string Five(){ return "V"; }
  public override string Nine(){ return "IX"; }
  public override int Multiplier() { return 1; }
}

/// <summary>
///  InterpreterApp Test
/// </summary>
public class InterpreterApp
{
  public static void Main( string[] args )
  {
    string roman = "MCMXXVIII";

    Context context = new Context( roman );
```

```
    // Build the 'parse tree'
    ArrayList parse = new ArrayList();
    parse.Add(new ThousandExpression());
    parse.Add(new HundredExpression());
    parse.Add(new TenExpression());
    parse.Add(new OneExpression());

    // Interpret
    foreach( Expression exp in parse )
      exp.Interpret( context );

    Console.WriteLine( "{0} = {1}",
                          roman, context.Output );
  }
}
```

IRS

- Information Retrieval System
- popis problematiky
- definícia booleovského a vektorového modelu, výpočet dopytovania, matica term-dokument
- indexy: invertovaný zoznam (napr. v SQL), suffix trees, signatúry
- hľadanie v texte: Brute Force, Knuth Morris Pratt, Boyer Moore

(http://www-igm.univ-mlv.fr/~lecroq/string/index.html)

Literatúra a zdroje:

http://www.dofactory.com

http://www.netobjectives.com/dpexplained

Shalloway A., Trott, J.R.: Design patterns explained. Addison-Wesley, 2005, second edition

Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object oriented software. Addison-Wesley, 1995, preklad vyšiel tiež v češtine vo vydavateľstve Grada