Design patterns 1

IT Academy, 2014

Objektovo orientované programovanie

- základný pojem - trieda - vzor pre objekty
- proces vytvorenia objektu sa nazýva vytvorenie inštancie triedy, jej výsledok je konkrétny objekt
- proces vytvorenia inštancie zvyčajne zahŕňa spustenie konštruktora danej triedy, ktorý by mal vrátiť konkrétny okjekt tejto triedy
- konkrétny objekt == inštancia

Objektovo orientované programovanie

- OOP je postup, ktorý nám v prvom rade umožňuje vytvoriť znovupoužiteľný kód (v kontraste s generickým programovaním, ktoré vytvára obecné riešenia)
- v dobe svojho príchodu sa táto metodika označovala ako revolučná, v praxi však ide iba o rozšírenie štrukturovaného programovania a programovania pomocou abstraktných dátových typov
- objekt je abstraktný dátový typ s pridaním **polymorfizmu** a **dedičnosti**

Objektovo orientované programovanie

- Cieľom OOP je:
  - zvýšená schopnosť pochopiť kód
  - zjednodušenie údržby kódu
  - zjednodušenie evolúcie kódu

V konečnom dôsledku má OOP za cieľ používať pri popise riešení na miesto databáz a programových subrutín objekty, ktoré sú známe pre danú problematiku.

Objektovo orientované programovanie

Trocha histórie:

Na začiatku programátori písali tzv. “špagetový kód

```
10 i = 0
20 i = i + 1
30 PRINT i; " squared = "; i * i
40 IF i >= 10 THEN GOTO 60
50 GOTO 20
60 PRINT "Program Completed."
70 END
```

```
10 FOR i = 1 TO 10
20     PRINT i; " squared = "; i * i
30 NEXT i
40 PRINT "Program Completed."
50 END
```

preto vymysleli štruktúrované programovanie

Objektovo orientované programovanie

Štruktúrované programovanie prinieslo subrutiny a procedúry. Prinieslo nezávislosť kódu pomocou scope.

OOP k nemu pridáva dáta. Objekty vlastnia svoje dáta, schovávajú ich pred okolím a pridávajú k nim funkcionalitu. Aplikácia je rozdelená na objekty a každý z nich je zodpovedný za svoje vlastné správanie. Okrem toho, že vieme spraviť nejakú operáciu vieme aj zabezpečiť, že dáta sa nie len nedajú modifikovať, **neexistuje k nim ani prístup okrem toho prístupu, ktorý je predpísaný.**

Toto sa nazýva ENCAPSULATION alebo ZAPÚZDRENIE

Objektovo orientované programovanie

Accessor

```
Public Class Person
    ' We use Private here to hide the implementation of the objects
    ' fullName, which is used for the internal implementation of Person.
    Private _fullName As String = "Raymond Lewallen"

    ' This property acts as an accessor.  To the caller, it hides the
    ' implementation of fullName and where it is set and what is
    ' setting its value. It only returns the fullname state of the
    ' Person object, and nothing more. From another class, calling
    ' Person.FullName() will return "Raymond Lewallen".
    ' There are other things, such as we need to instantiate the
    ' Person class first, but thats a different discussion.
    Public ReadOnly Property FullName() As String
        Get
            Return _fullName
        End Get
    End Property

End Class
```

Objektovo orientované programovanie

```
Public Class Person
    ' We use Private here to hide the implementation of the objects
    ' fullName, which is used for the internal implementation of Person.
    Private _fullName As String = "Raymond Lewallen"

    ' This property now acts as an accessor and mutator.  We still
    ' have hidden the implementation of fullName.
    Public Property FullName() As String
        Get
            Return FullName
        End Get
        Set(ByVal value As String)
            _fullName = value
        End Set
    End Property

End Class
```

Mutator

Objektovo orientované programovanie

```
Public Class Person
    Private _fullName As String = "Raymond Lewallen"

    ' Here is another example of an accessor method,
    ' except this time we use a function.
    Public Function GetFullName() As String
        Return _fullName
    End Function

    ' Here is another example of a mutator method,
    ' except this time we use a subroutine.
    Public Sub SetFullName(ByVal newName As String)
        _fullName = newName
    End Sub

End Class
```

Accessor/Mutator

Objektovo orientované programovanie

Ďalšie slovné spojenie spojené s OOP:

open/closed principe

Modul je **uzavretý**, ak poskytuje okoliu stabilný interface pre komunikáciu, a nedovoľuje modifikovať svoje správanie.

Modul je **otvorený**, ak umožňuje okoliu modifikáciu svojho správania, ak je to potrebné.

Objektovo orientované programovanie

Objekty reprezentujú veci z reálneho sveta (Employee, BankAccount), ktoré disponujú určitým správaním. Program môže používať viacero inštancií objektov každého typu, a tie spolu interagujú.

Okrem toho, môžeme takýmto spôsobom abstrahovať aj procedurálny kód aplikácie. V programovaní existuje viacero všeobecných vzorov, ktoré umožňujú takéto štruktúrovanie.

Objektovo orientované programovanie

"An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of object and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer."

G. Booch, Object-Oriented Design With Applications, Benjamin Cummings, Menlo Park, California, 1991.

Objektovo orientované programovanie

Okrem zapúzdrenia (encapsulation), sa OOP spája aj s týmito termínmi.

Dedičnosť (to už poznáme :P)

Dátová abstrakcia (aj to už poznáme)

Polymorfizmus

- **je sprístupnenie obecného rozhrania pre entity rozdielnych typov**
- v C++ sa polymorfizmus môže implementovať pomocou preťaženia funkcií - ad hoc polymorfizmus alebo pomocou šablón - parametrický polymorfizmus
- **vytváranie podtypov** - inclusion polymorfizmus - polymorfizmus, ktorého základom je vytvorenie tried, ktoré zdieľajú rovnakého predka - stromov tried
- v C++ to znamená, že na mieste, kde môžem použiť obecný typ, napr. Animal, môžem použiť aj špecifikejší typ, napr. Cat, ak Cat dedí od Animal (Cat je potomkom Animal)

Objektovo orientované programovanie

OOP nie je samé o sebe ideálnym riešením.

OOP nie je ani ideálnym prostriedkom pre modelovanie situácii.

OOP je však:

- jednoduché na pochopenie
- extrémne produktívne, ak je správne použité

Objektovo orientované programovanie

Mutable vs. Immutable object

Immutable - inicializácii sa stav objektu už nemôže menit’.

Design patterns

Návrhový vzor

Je to všeobecne využiteľná predloha pre bežne sa vyskytujúci problém a kontexte dizajnu software. Je to predpis šablóny možného riešenia problému, ktorý je možné aplikovať vo viacerých situáciach. Vzory sú vypracované podľa best practises, aké môže programátor využiť pri riešení toho daného typu problému.

Design patterns

Návrhový vzor

Je to pojem, ktorý chápeme v kontexte modulov a prepojení medzi nimi.

Na vyššom leveli sú **architektonické vzory**, ktoré modelujú software na vyššej úrovni.

Design patterns

MVC

Podľa najnovších informácii na wiki:

Design pattern určený pre implementáciu užívateľských rozhraní.

Autor tejto prezentácie stále nepovažuje MVC za design pattern.

Design patterns

example-servant.java

analyzujte!

Design patterns

Služobník - servant

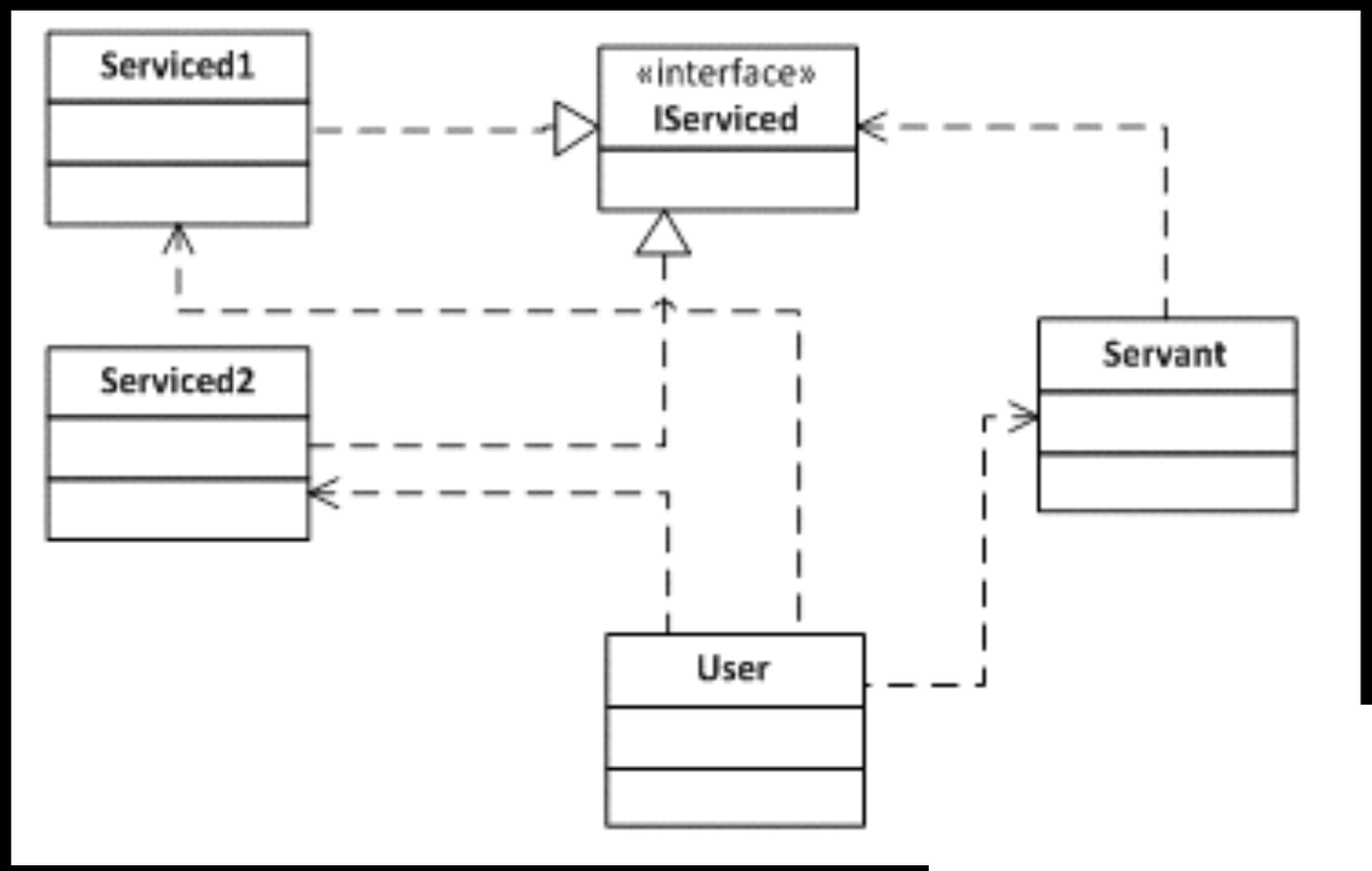
- používa sa pre pridanie nejakého správania skupine tried
- používa sa vtedy, ak nemôžem modikovať base class (napríklad aj pre to, že existuje viacero base class)
- v C++ ním vieme nahradiť niektoré dôvody implementácie viacnásobnej dedičnosti, alebo použitia friend classes

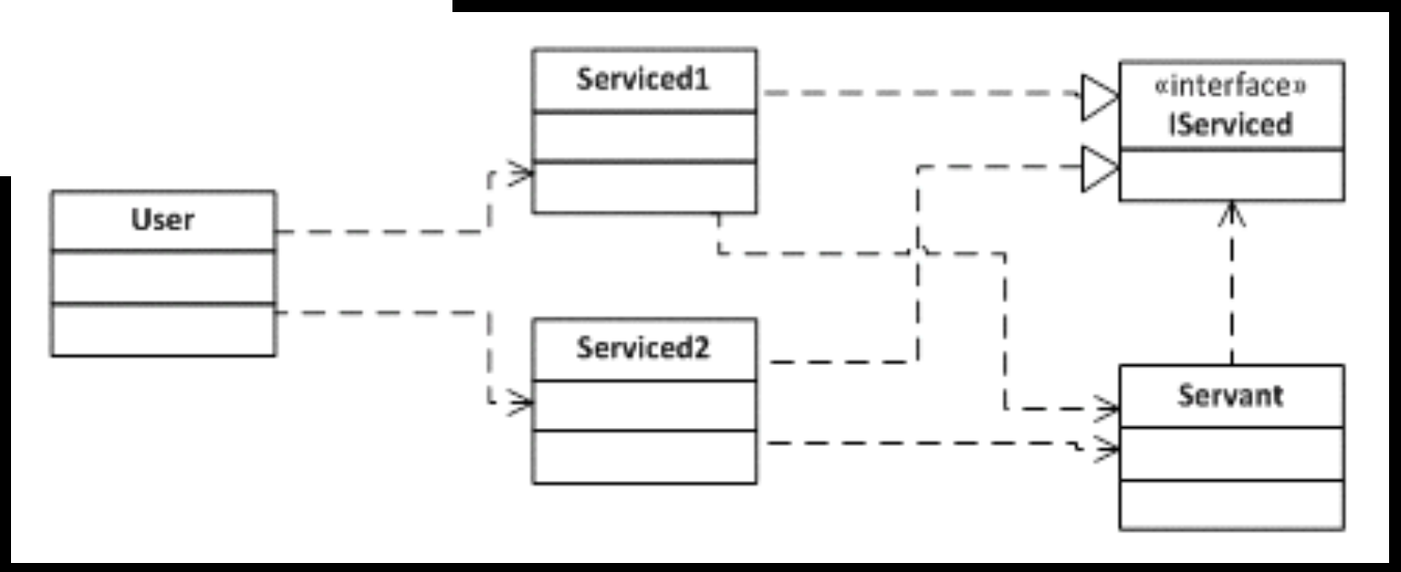
Design patterns

2 spôsoby implementácie služobníka:

- buď užívateľ pozná služobníka, a predá mu potrebné objekty ako parametre, aby s nimi mohol služobník vykonať potrebné akcie
- alebo objekty poznajú služobníka, ale užívateľ nemusí. Ak si užívateľ vyžiada akciu od objektov, tie kontaktujú služobníka a požiadajú ho o vykonanie potrebnej akcie.

Design patterns





Design patterns

Servant pattern sa veľmi podobá na

Command pattern

Rozdiel je v tom, že pri Command patterne chceme modifikovať predané objekty ->
napr. prepočítať pre ne nejaké hodnoty

Kód pre **Command** a **Servant** vyzerá takmer identicky štruktúrou, rozlišuje ho iba zámer

Design patterns

Pre úplnosť - v osnove školenia je spomenutý aj "návrhový vzor"

Prepravka

Ktorý hovorí:

Miesto 10 argumentov jednoduchého typu použi objekt!!!

Nikde inde v literatúre sa takýto štandardný postup OOP neoznačuje ako design pattern, okrem prác pána Pecinovského, autora knižky, podľa ktorej je vytvorená osnova školenia.

http://edu.pecinovsky.cz/papers/2006_ITiCSE_Design_Patterns_First.pdf

Design patterns

Okrem toho sa v osnove spomína aj

Null object design pattern

viac si o ňom môžete prečítať na wiki:
http://en.wikipedia.org/wiki/Null_Object_pattern#C.2B.2B

v skratke:
Spravím si vlastnú triedu len preto, aby som získal prázdny objekt, ktorý potom používam ako NULL object, ktorý implementuje požadované rozhrania, čiže nevyvolá chybu pri preklade

Design patterns

Library class pattern

V skratke vo väčšine jazykov:

Všetko je v jednej triede, a všetko je statické. Ani len neinštancujem (nemôžem).

```
▶ TypeError: 'undefined' is not a function (evaluating 'require.config')
> new Math();
▶ TypeError: '[object Math]' is not a constructor (evaluating 'new Math()')
> Math.abs(-3)
< 3
```

Design patterns

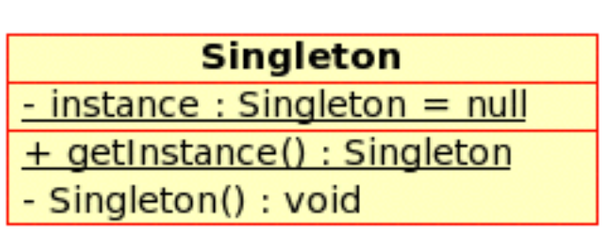
Singleton design pattern

```
public class SingletonDemo {
        private static SingletonDemo instance = null;

        private SingletonDemo() {       }

        public static synchronized SingletonDemo getInstance() {
                if (instance == null) {
                        instance = new SingletonDemo ();
                }
                return instance;
        }
}
```

Design patterns

Singleton design pattern

UML [edit]




```
public class Singleton {
    private static final Singleton INSTANCE = new Singleton();

    private Singleton() {}

    public static Singleton getInstance() {
        return INSTANCE;
    }
}
```

```
public enum Singleton {
        INSTANCE;
        public void execute (String arg) {
                // perform operation here
        }
}
```

```
Foo := Object clone
Foo clone := Foo
```

Design patterns

Singleton design pattern

- jeden z absolútne najrozšírenejších design patternov
- nesmie sa používať nadmerne, inak narušuje korektné OOP modelovanie
- najčastejšie sa používa pre situácie, kde potrebujeme objekt, ktorý je v aplikácii vždy iba jeden - napr.: **logger**
- môže výhodne centralizovať akcie - napríklad koordinovať časti systému ($app, $kernel)

Design patterns

Enum design pattern

http://www.codeproject.com/Articles/38666/Enum-Pattern

Design patterns

Object pool

Tento design pattern používa kolekciu (pool) predinštancovaných objektov, ktoré následne poskytuje pre použitie. Význam má vtedy, ak je vytvorenie inštancie drahé, alebo ak je inštancií málo.

However these benefits are mostly true for objects that are expensive with respect to time, such as database connections, socket connections, threads and large graphic objects like fonts or bitmaps. In certain situations, simple object pooling (that hold no external resources, but only occupy memory) may not be efficient and could decrease performance.

http://en.wikipedia.org/wiki/Object_pool_pattern

Design patterns

Flyweight.py - **Python**
Flyweight.cs - **C#**
Flyweight.rb - **Ruby**
Flyweight.java - **Java**

Vyber si jeden zdrojový kód a povedz, čo je špecifikom tejto implementácie, čo robí tento zdrojový kód, a prečo si myslíš, že je navrhnutý práve takto.

Design patterns

Flyweight design pattern

- Je to design pattern, ktorý redukuje množstvo použitej pamäte pri behu programu
- Dosahuje to tak, že vytvorí úložisko dát, ktoré sa medzi jednotlivými inštanciami opakujú a uloží každú z nich iba raz
- K jednotlivým hodnotám dát sa potom pristupuje pomocou indexu, ktorý je uložený v jednotlivých inštanciách
- Redukujem množstvo pamäte tak, že vyabstrahujem opakujúce sa dáta a vytvorím pre ne tabuľku
- Za akých podmienok sa mi to oplatí?

Design patterns

Proxy design pattern

./proxy/

Čo robí trieda ImageProxy a prečo je súčasťou návrhu?

Design patterns

Proxy

Návrhový vzor Proxy poskytuje obal, alebo zástupcu pre iný objekt, aby mohol k nemu kontrolovať prístup. Používa sa na reprezentácu komplexného objektu pomocou objektu jednoduchšieho. Ak je vytvorenie objektu drahé (pamäťovo, výpočtovo), môžeme jeho vytvorenie odložiť až do momentu, keď je nevyhnutné objekt vytvoriť, a medzitým môže jednoduchší objekt slúžiť ako zástupca. Tento objekt-zástupca sa nazýva Proxy objekt.

Design patterns

Proxy

Zaujímavým je využitie Proxy a Flyweight v kombinácii:

Flyweight mi vytvorí slovník dátových objektov (akože dát), pristupujem k nim ale priamo cez Proxy inštancie, ktoré sú málo pamäťovo náročné. Ak budú Proxy inštancie inštancovať vo volaných metódach pamäťovo náročný objekt, tak vždy po dokončení bloku kódu bude objekt **dealokovaný**.

Design patterns

```
1 template <class Category, class T, class Distance = ptrdiff_t,
2           class Pointer = T*, class Reference = T&>
3   struct iterator {
4     typedef T          value_type;
5     typedef Distance   difference_type;
6     typedef Pointer    pointer;
7     typedef Reference reference;
8     typedef Category   iterator_category;
9   };
```

Design patterns

Iterator.cpp

Design patterns

Design patern **Iterátor** je použitý v STL pre prechádzanie rôznych kontajnerov. Plné pochopenie návrhového vzoru Iterátor umožní programátorovi vytvárať znovupoužiteľné a ľaho pochopiteľné kontajnery dát.

Základnou myšlienkou Iterátora je, že povoľuje prechádzanie kontajnera (tak ako pointer umožňuje prechádzanie poľa). Pre prechod na ďalší element kontajnera však potrebujeme poznať vnútornú štruktúru kontajnera. Toto je úlohou Iterátora. Pomocou metód iterátora môžete prechádzať všetky prvky kontajnera vo vopred určenom poradí, od prvého prvku po posledný.

Design patterns

…a teraz pseudokód :P

——————————> pseudostate.pseudo

Design patterns

State pattern je o tom, že zapúzdrujem stav objektu v objekte

Môžem sa tak vyhnúť napríklad zbytočným switchom…

Design patterns

>>> facade.cpp

Design patterns

Facade pattern

Po slovensky **fasáda** - zakrývam to, čo je pod ňou.

Vytváram zjednotený známy interface pre viacero úkonov, ktoré potrebujem vykonať’, alebo vytváram interface pre class library.

Design patterns

Facade pattern

- software sa potom ľahšie používa, chápe a udržuje
- môžeme týmto spôsobom redukovať počet závislostí, ktoré potrebujeme používať pre dané úkony - **facade** tieto závislosti naloaduje vo svojom vnútornom fungovaní, ja uvádzam ako závislosť iba na **facade**.
- Ak spravil kolega šialené API, viem si ho prerobiť tak, aby bolo ľahšie použiteľné

Design patterns

Adapter.cs

analyzujte!

Design patterns

Adapter pattern

Jednoducho sa nazýva aj wrapper. Slúži pre vytvorenie rozhrania pre objekt, ktorý požadované rozhranie neobsahuje.

Design patterns

Composite pattern

Design patterns

Composite pattern - popisuje viacero objektov tak, akoby to bol jeden jediný objekt, pričom objekty sú usporiadané do stromovej štruktúry. Pomocou composite potom môžeme rovnako pristupovať ku individuálnym objektom, aj k ich kompozíciam (skupinám objektov)

Anti-pattern

http://en.wikipedia.org/wiki/Anti-pattern#Object-oriented_design

;)