# 1 Design Paterns

## 1.1 Ciel

Cielom prednasky je predstavit jednu z design technik - design patterns. Nie je cielom predstavit detailne jednotlive patterny, skor trendy ktore viedli ku ich vzniku.

Zaroven chceme poukazat na silu tohto nastroja a motivovat studentov aby o nich zacali uvazovat, pripadne ich pouzivat.

## 1.2 Preco Patterny?

Co je expert? Co odlisuje experta od novacika? (Je to najma skusenost)

Experti neriesia problemy tak, ze by k nim pristupovali uplne z nicoho (maju iste skusenosti), zvycajne vyuzivaju riesenie, ktore uz kedysi pouzili, a ktore sa osvedcili ako spravne...

V design patternoch su ukryte "dobre design techniky/zrucnosti". Tie mozu byt povedane uz len v samotnom mene patternu, resp. jeho obsahu.

Napr. Pattern Iterator, kde sa modeluje "sekvencny pristup ku agregovanemu objektu(kolekcii objektov) bez toho, aby sa odhalila vnutorna struktura agregacie(kolekcie)". To jest pri navrhu je dobre aby sa vnutorna struktura agregacie ukryval za nejaky vseobecnejsi intrace, co potom vedie k robustnejsiemu a lahsie spravovatelnemu systemu.

## 1.3 Co je design pattern?

Kazdy vzor reprezentuje akysi problem, ktory sa neustale opakuje (v prostredi OOD) a nasledne popisuje zaklad riesenia, ktore na ten problem mozno aplikovat.

Ludia si vsimli, ze castokrat vymyslaju te iste veci co uz boli vymyslene, rep. sa k nim tazko dopracuvaju.

Patterny su vysledkom pozorovania expertov (skusenych programatorov a designerov), ktori ku patternu dospeli zvycajne po dlhej cest zmien a refactoringu (takze to nie je easy)

Pattern na prvy pohlad moze vyzerat zlozito, zvycajne vsak plati, ze neskor je vyhodne mat viac vseobecnejsie a reusovatelnejsie riesenie.

## 1.4 Design Pattern

Vo vseobecnosti ma DP tieto zakladne prvky:

1. Meno
   - jedno alebo viacslovne pomenovanie, ktore sluzi na komunikaciu (vystizne)
2. Problem
   - Kedy aplikovat pattern
   - Kontext, popis situacie v ktorej sa pattern vyskytuje
3. Riesenia
   - Abstraktny popis tried, vztahov, zodpovednosti a kolaboracii

- Nie je to konkrtne riesenie, ale iba akysi template

4. Nasledky
   - Co sposobuje pouzitie patternu (napr. reusability)

### *1.5 Kratke zhrnutie*

Co si predstavime, ked sa povie **(design) pattern**

- **Zovseobecnenie**
- **Identifikacia klucovych prvkov (objektov, tried, vztahov,...)**
- **Znovapouzitelnost**

Tieto aspekty je potrebne dodrziavat (mat na pamati) ked definujeme novy pattern.

## 2 Priklady patternov

### *2.1 Decorator*

#### 2.1.1 Obsah

Pridat objektu dodatocnu funcionalitu (zodpovednost) dynamicky. Alternativa ku rozsitrovaniu funkcionality prostrednictvom dedenia (ktore je neflexibilne).

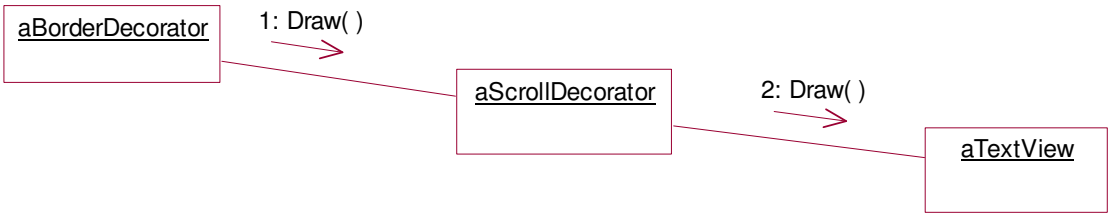
#### 2.1.2 Motivacia

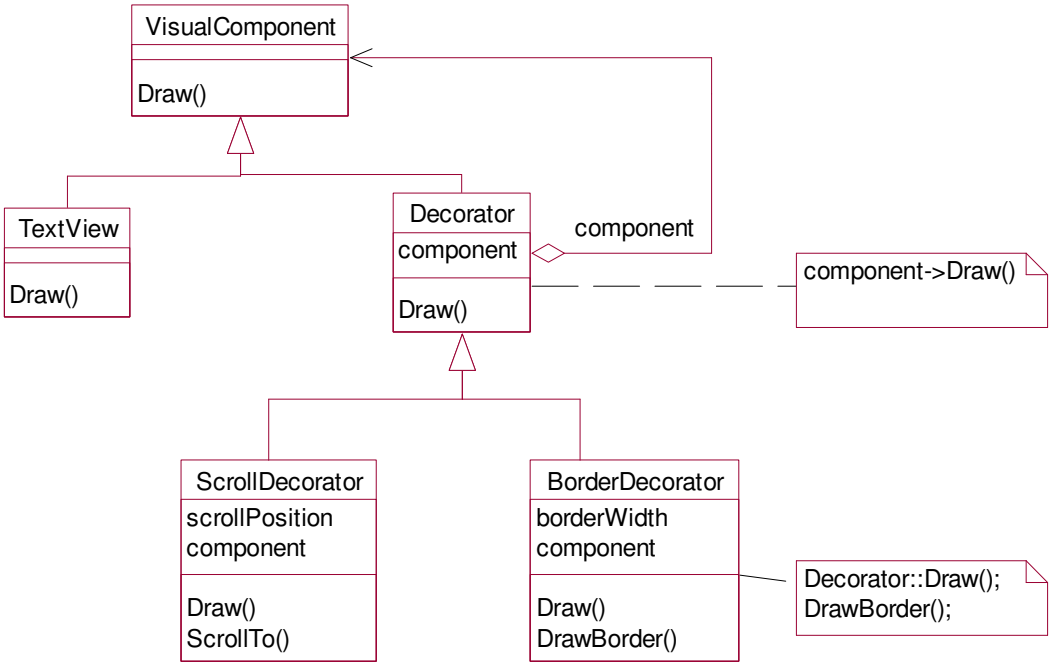
Niekedy je potrebne pridavat objektom dodatocnu funkcionalitu (resp. zodpovednost).

Napr. vezmime si GUI toolkit, v ktorom mame GUI element TextView. Tato trieda reprezentuje cisty prezeac textov, bez akejkolvek pridavnej funkcionality (pretoze nie vzdy je potrebna). Obcas potrebuje klient aby TextView obsahovalo aj scroll-bary a pripadn aby bolo cele oramovane hrubym ramom. V principe su dva mozne sposoby ako to dosiahnut, **staticky** a **dynamicky**.

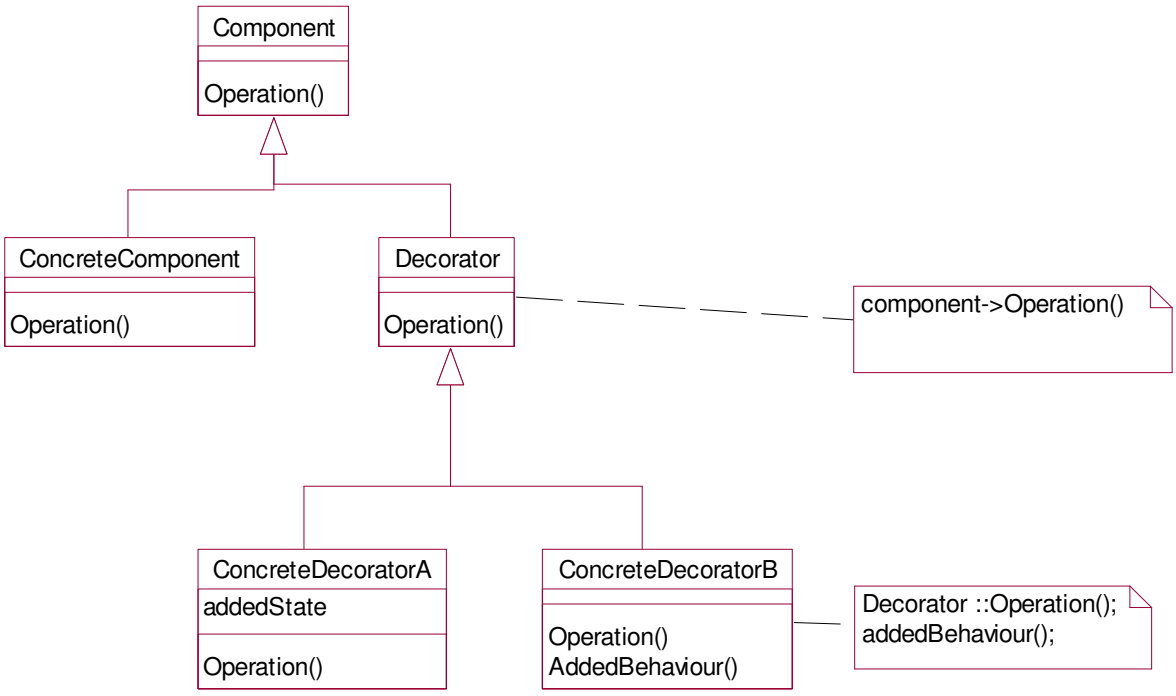
V pripade **statickeho** riesenia, sa vyuzije dedenie. Klient si zadefinuje novu triedu, ktora zdedi vlastnosti povodnej, navyse k nej prida pozadovanu funkcinalitu (scroll-bary a hruby ram). Toto riesenia je sice robustne, ale je neflexibilne, pretoze si vyzaduje prekompilovanie kodu. A klient musi dopredu vediet, aku funkcionalitu chce pridavat.

Dalsou moznostou je dynamicke riesenie. V tomto pripade sa objekt, ktoreho funkcionlitu chceme rozsirit, zabali (encapsulates) do noveho objektu, ktory prida danu funkcionalitu. Tento objekt nazyvame **decorator** a jeho interface je zhodny z interfacesom objektu, ktory rozsirujeme. Podobne mozme rekzivne vhniezdovat decoratory.


aBorderDecorator
1: Draw( )
aScrollDecorator
2: Draw( )
aTextView


VisualComponent
Draw()
TextView
Draw()
Decorator
component
Draw()
component
component->Draw()
ScrollDecorator
scrollPosition
component
Draw()
ScrollTo()
BorderDecorator
borderWidth
component
Draw()
DrawBorder()
Decorator::Draw();
DrawBorder();

### 2.1.3 Struktura




### 2.1.4 Pouzitie

Tento pattern pouzivame ak:

- chceme pridat funkcionalitu jednotlivym objektom dynamicky
- chceme dynamicky nejaku funkcnost odoberat
- ked rozsirovanie dedenim je nemozne (napr. def. triedy je skryta a neda sa dedit)

### 2.1.5 Objekty

Component

ConcreteComponent

Decorator

ConcreteDecorator

### 2.1.6 Dosledky

- Vacsia flexibilita.
- "Pay-as-you-go", aplikacia nemusi byt plna funkcionality, ktoru nepouziva.
- Decorator a objet nie su identicke, pozor pri vyuzivani identity objektu
- Pri nekontrolovanom pouzivani vznika v systeme vela malych objektov, co je neprehladne a moze sa zle debugovat.

## 2.2 Visitor

### 2.2.1 Obsah

Mame hierarchiu, ktora je dost staa a potrebujeme pridavat nove operacie pracujuce nad elementami tejto hierarchie. Nove operacie modelujeme ako triedy.

## 2.3 Motivacia

Uvazujme napriklad system na vzdialenu administraciu pocitacov s roznymi operacnymi systemami v jednej sieti.

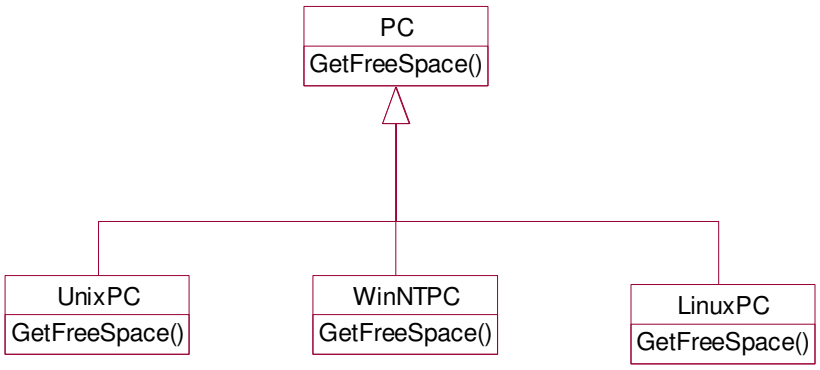
Teda nasa struktura objektov bude nejaky graf, ktory reprezentuje siet pocitacov. Kazda trieda v hierarchii modeluje pictace s rovnakym OS.

Chceme na tejto strukture implementovat nejaku udrzbu, teda nejake operacie nad pocitacmi, napr. zistit kolko je volneho miesta vo file systeme. Pre kazdu triedu (napr. UnixPC, WinNTPC, LinuxPC...) bude implementacia tejto operacie ina. Takze urobime abstraktnu triedu PC a v nej zadefinujeme operaciu GetFreeSpace(), pricom kazda z podtried ju implementuje podla potreby.

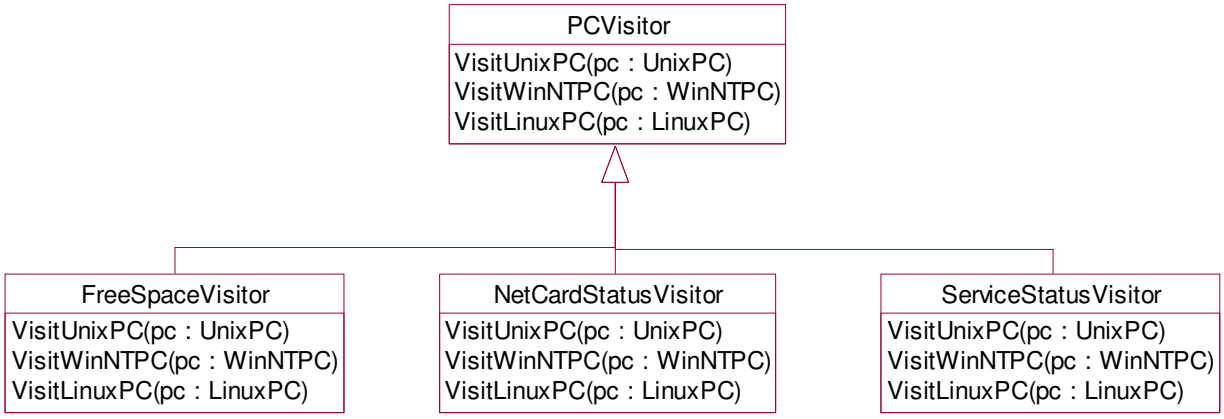
Ak vsak budeme v buducnosti potrebovat pridat novu operaciu napr. ci bezi nejaky servis, alebo zistit status sietovej karty, tak budeme musiet menit celu hierarchiu tried. Je to dost neflexibilne, a navyse taketo operacie logicky nepatria do triedy PC, su to operacie nad jej spravanim a stavom.

Preto vytiahneme tieto operacie do specialneho objektu Visitor. Pre kazdu operaciu bude samostatny Visitor a pre kazdy element hierarchie bude Visitor obsahovat specialnu metodu na spracovanie (metoda navstivenia danej triedy).

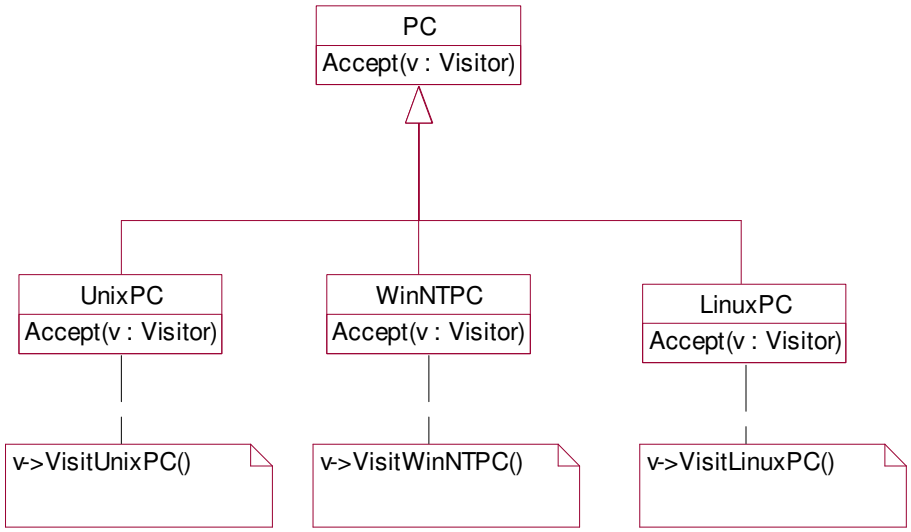
Takto by mohla vyzerat povodna hierarchia.



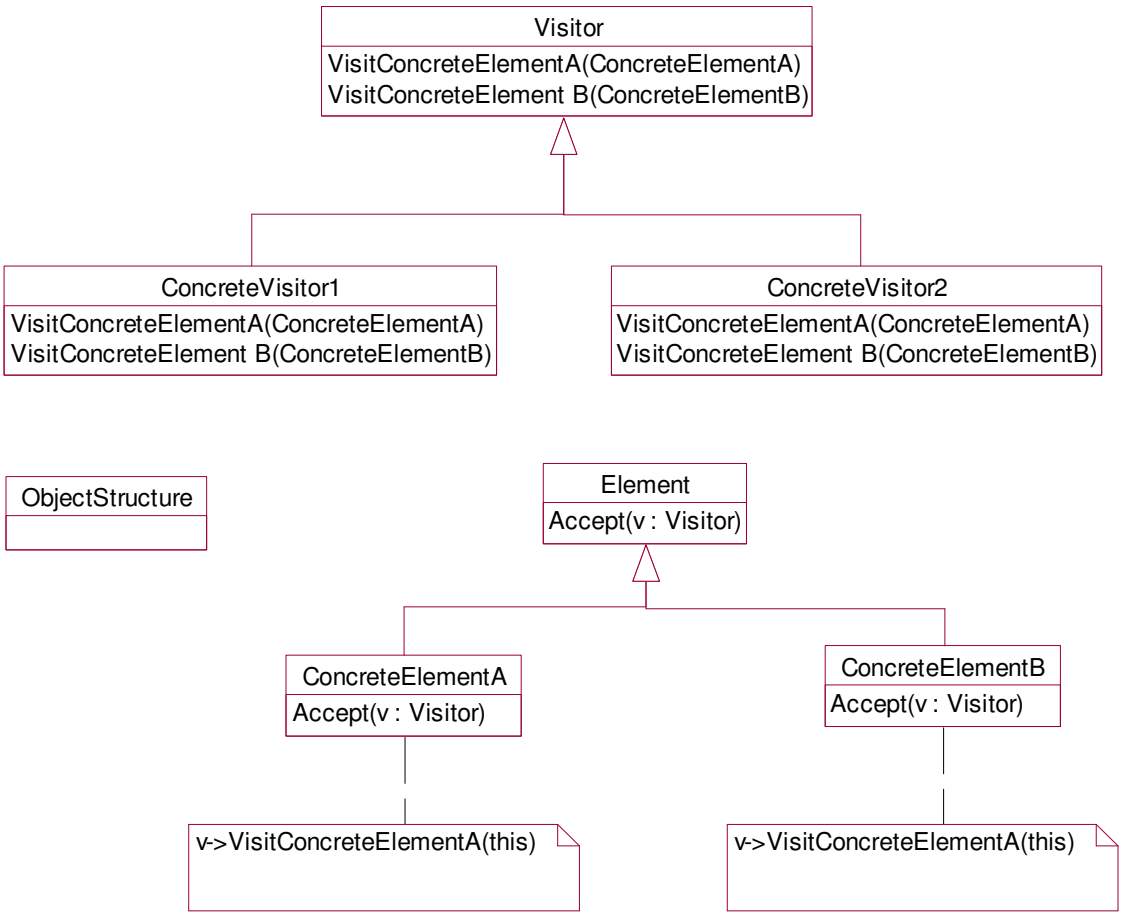

Pridanie novej operacie do tejto hierarchie je vsak dost neflexibilne a vyzaduje si zasah do kazdej triedy (je interfacesu). Preto vytiahneme tuto operaciu do samostatnj triedy Visitor.




Nasledne upravime povodnu hierarchiu.



### 2.3.1 Struktura




### 2.3.2 Pouzitie

- Ak objektova struktura obsahuje vel tried s rozdielnym interfacesom a je potrebne vykonavat operacie na objektoch tejto hierarchie, ktore zavisia od konkretnej triedy daneho objektu
- Niektore operace sluzia len na manipulaciu s objektami a nie su logickou sucastou ich tried. Ak chceme zabranit "preplnovaniu" tried tak ich modelujeme v samostatnych triedach. Zaroven nam to umoznuje customizovat, mnozinu tychto operacii (niektorym aplikaciam nemusime dat celu mnozinu visitorov).
- Ak sa hierarchia meni ib zriedka, ale castejsie sa pridavaju nove opracie na tejto strukture.

### 2.3.3 Objekty

- Visitor
- ConcreteVisitor
- Element
- ConcreteElement

- ObjectStructure

### 2.3.4 Interakcie

### 2.3.5 Dosledky

- Lahke pridanie novych operacii
- Spajanie navzajom suvisiacich operacii a oddelovani nesuvisiacich
- Nezavislost od povodnej hierarchie
- Moznost akumulacie stavu
- Pridanie noveho elemntu do povodnej hierarchie je zlozite
- Narusanie principov encapsulacie

# 3 Strucne o inych patternoch

## *3.1 Proxy*

Obsah: vytvorenie zastupcu pre nejaky objekt, ktory kontroluje pristup ku tomu zastupovanemu objektu.

## *3.2 Chain of responsibility*

Obsah: umoznit viac ako len jednemu objektu spracovat nejaku poziadavku (request). Cielove objekty su zretazene a poziadavka je postupne predavana kazdemu az kym ju niekto neosetri.

Napr. ked mame help button v dialogu, tak pri jeho zavolani sa najpr vola metoda dialogu (aby user dostal konkretnu help zavislu od dialogu). Ak dialog nereaguje na tuto metodu, tak sa request posuva vyssie napr. aplikacii ktora ho spracuje a zobrazi svoj help (ktory uz nebude az tak specificky, pretoze aplikacia nevie, ze request pochadza z dialogu).

## *3.3 Command*

Obsah: oddelit objekt ktory vydava nejaky prikaz (klient) od objektu, ktory ho spracuva. Prikaz je zabaleny do smostatneho objektu, ktory sa potom moze zaradovat do fronty, logova, pripadne inak parametrizovat. Klient nie je zavisly na konkretnej implementacii prikazu, ani na objekte ktory ho implementuje.

Umoznuje to parametrizovat klientov, tak ze miesto toho aby volali konkretnu metodu nejakeho konkretneho objektu, zavolaju standardnu metodu specialneho objektu Command, ktory potom sam rozhodne co dalej spravi.

Vyuziva sa napr. v GUI v menu. Ked sa zavola polozka menu tak ta ma priradeny nejaky Command objekt a toho metodu Execute potom zavola.

## *3.4 Interpreter*

...nerozumiem...

### 3.5 Iterator

Obsah: sekvencne prehladvanie v agregovanych objektoch (kolekciach)

Majme nejaky agregovany objekt napr. zoznam (alebo strom, mnozinu...) a iterator nad tymto objektom. Agregovany objekt definujeme pomocou operacii add(), remove, count() a pod.

Kedze chceme aby aplikacia bola nezavisla od implementacie agregovaneho objektu, potom iterator nad touto strukturou musi splnat nejaky vseobecny interface. Potom vsak mame problem aby iterator bol pripraveny na "mozno" meniacu sa strukturu agregovaneho objektu.

Teda agregovany objekt bude zodpovedny za vytvorenie vhodneho iteratora, ktory bude rozumiet jeho strukture.

### 3.6 Mediator

Obsah: modelovat inteakciu v skupine objektov prostrednictvom samostatneho objektu a tym znizovat zavislosti medzi jednotliymi objektami.

To umoznuje aby jednotlive objekty s kupine nereferncovali priamo ostatne objekty (**loose coupling**) a zaroven aby sa mohla jednoducho **menit** ich **interakcia** bez nutnosti zmeny objektov.

Napriklad dialog box v GUI. Zvycajne byvaju jednotlive komponenty (check boxy, buttony, list boxy...) nejako previazane, teda ak sa zmeni jeden komponent potom sa moze zmenit jeden alebo viac inych komponentov (napr. vyplnenie text boxu moze enablnut nejaky button).

### 3.7 Memento

Obsah: zapamatat si stav objektu (externalizovat ho) bez porusenia encapsulacie, tak aby sme mohli neskor objekt priviest do toho isteho stavu.

Pouziva sa pri implmentovani Undo operacii.

### 3.8 Observer

Obsah: niekolko objektov zavislich na obsahu jedneho objektu su automaticky notifikovane v pripade jeho zmeny.

Napr. ked mame data v Exceli tak mozme tieto data zobrazovat v tabulke, alebo v grafe a pod.

### 3.9 State

Obsah: povolit objektu zmenit spravanie ak sa zmeni jeho stav. V tomto pripade sa vlastne meni trieda objektu.

### 3.10 Strategy

Strategia vybere algoritmus z triedy navzajom ekvivalentnych algoritmov, bez toho aby do toho klient zasahoval

### 3.11 Template method

Zadefinovat kostru algoritmu, pricom niektore kroky sa mozu predefinovat v podtriedach. Klient tak moze vyspecifikovat dane kroky, ale nemoze menit charakter algoritmu.

## 4 Zhrnutie

<nedokoncene>

### 4.1 Creational Patterns

### 4.2 Structural Patterns

### 4.3 Behavioral Patterns

- Zapuzdrenie zmeny

  Ak sa nejaky aspekt programu casto meni, tak ho zabalime do objektu

- Objekty ako argumenty

  Vseobecny interface, ale pouzivame vzdy konkretny objekt ako argument, a ten je nositelom konkretnej funkcnosti (variabilita zodpovednosti)

- Distribucia interakcie(komunikacie) vs. Zapuzdrenie interakcie

  Observer distribuuje komunikaciu....mediaor centralizuje.

- Rozdeolvanie posielatelov sprav od prijimatelov

## 5 Links

Patterns (referncie)

http://www.hillside.net/patterns/EgPatterns.html

GOF Pattern Models

http://www.tml.hut.fi/~pnr/Tik-76.278/gof/html/index.html

Haifa Patterns Group

http://www.cs.technion.ac.il/cdrom-3.98/236700/dp/

Software Patterns

http://vismod.www.media.mit.edu/~tpminka/patterns/