

1 Design Paterns

1.1 Ciel

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

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

1.2 Preto Patterny?

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

Expert 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 "dobré 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 vseobecnejši intrace, co potom vedie k robustnejšiemu a ľahšie spravovateľnému systému.

1.3 Co je design pattern?

Kazdy vzor reprezentuje akysi problem, ktory sa neustale opakuje (v prostredí 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), ktorí ku patternu dospeli zvycajne po dlhej cest zmién a refactoringu (takze to nie je easy)

Pattern na prvý pohľad moze vyzerat zlozito, zvycajne vsak plati, ze neskôr je vyhodne mat viac vseobecnejšie a reusovatelnejšie riesenie.

1.4 Design Pattern

Vo vseobecnosti ma DP tieto zakladne prvky:

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

- Nie je to konkrétne riešenie, ale iba akýsi template
4. Nasledky
- Co spôsobuje použitie patternu (napr. reusability)

1.5 *Kratke zhrnutie*

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

- **Zovseobecnenie**
- **Identifikacia klucovyh prvkov (objektov, tried, vzťahov,...)**
- **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 funkcionalitu (zodpovednost) dynamicky. Alternativa ku rozsiritovaniu funkcionality prostrednictvom dedenia (ktore je neflexibilne).

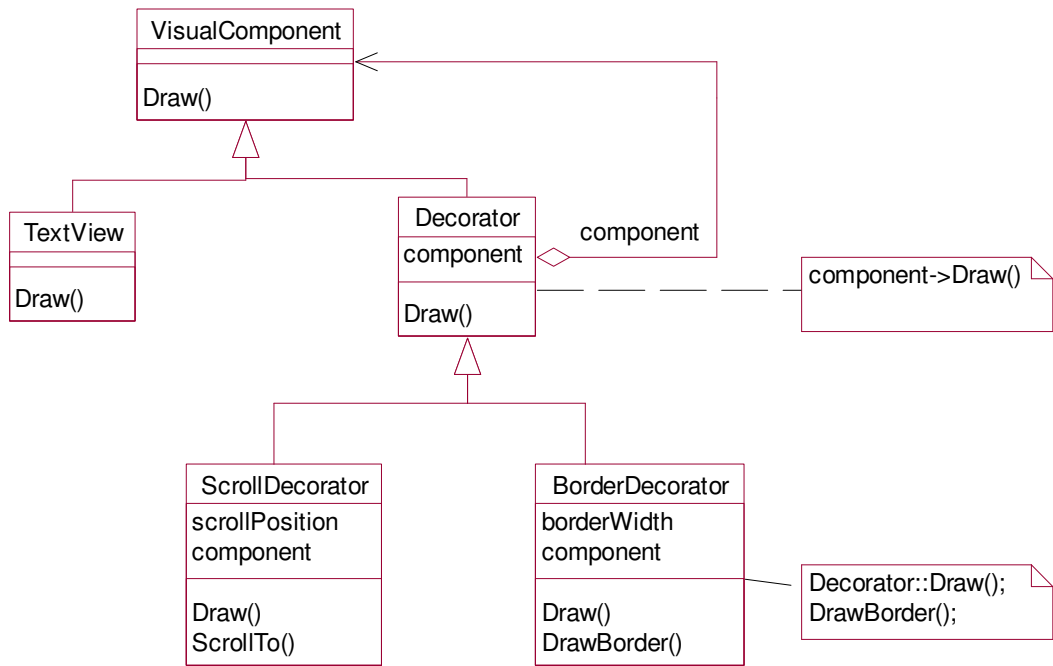
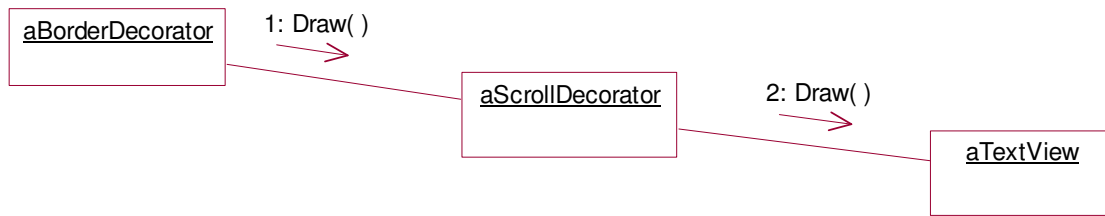
2.1.2 **Motivacia**

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

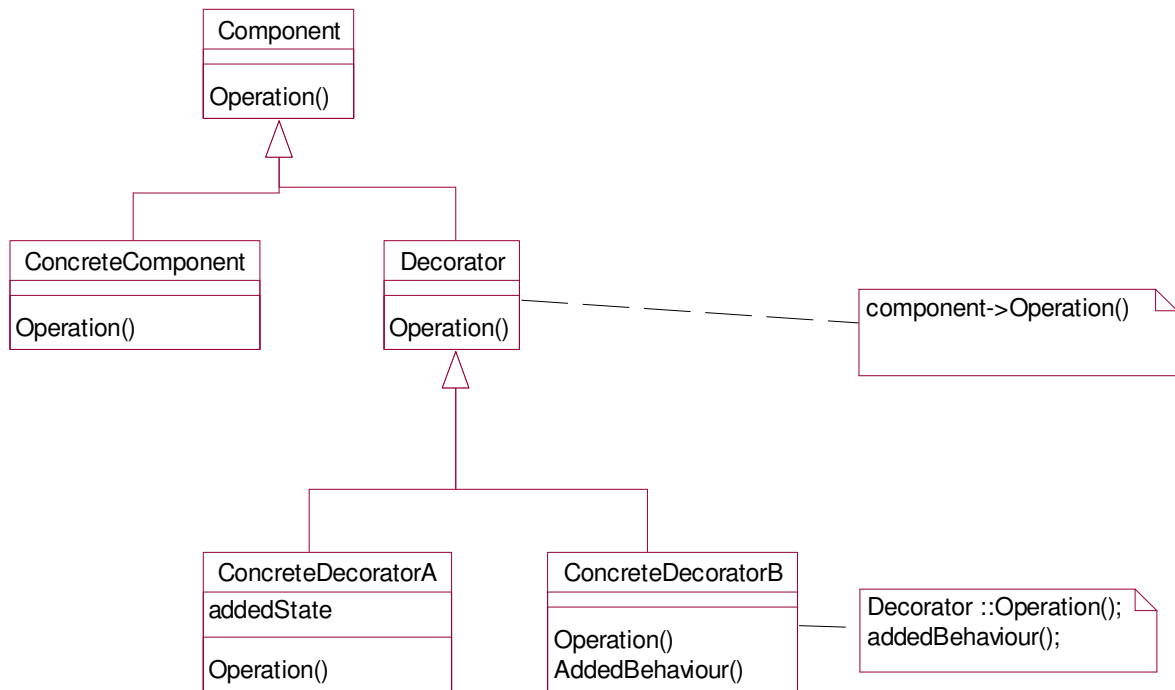
Napr. vezmime si GUI toolkit, v ktorom mame GUI element TextView. Tato trieda reprezentuje cisty prezeac textov, bez akejkoľvek pridavnej funkcionality (pretoze nie vzdy je potrebna). Obcas potrebuje klient aby TextView obsahovalo aj scroll-bary a pripadn aby bolo cele oramovane hrubym ramom. V principe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

V pripade **statickeho** riesenia, sa vyuzije dedenie. Klient si zdefinuje novu triedu, ktora zdedi vlastnosti povodnej, navyse k nej prida pozadovanu funkcionalitu (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 funkcionalitu chceme rozsirit, zabali (encapsulates) do noveho objektu, ktory prida danu funkcionalitu. Tento objekt nazývame **decorator** a jeho interface je zhodny z interfacesom objektu, ktory rozsirujeme. Podobne mozme rekurzivne vhniedzovat decoratoru.



2.1.3 Struktura



2.1.4 Použitie

Tento pattern používame ak:

- chceme pridať funkčnosť jednotlivým objektom dynamicky
- chceme dynamicky nejakú funkčnosť odobrať
- keď rozširovanie dedením je nemožné (napr. def. triedy je skrytá a neda sa dediť)

2.1.5 Objekty

Component
ConcreteComponent
Decorator
ConcreteDecorator

2.1.6 Dôsledky

- Väčšia flexibilita.
- "Pay-as-you-go", aplikácia nemusí byť plná funkčnosti, ktorú nepoužíva.
- Decorator a objekt nie sú identické, pozor pri využívaní identity objektu
- Pri nekontrolovanom používaní vzniká v systéme veľa malých objektov, čo je neprehľadné a môže sa zle debugovať.

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 napríklad system na vzdialenu administráciu počítačov s roznoými operačnými systemami v jednej sieti.

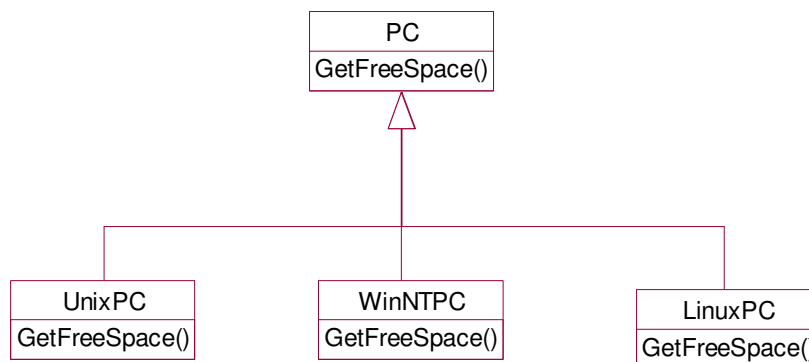
Teda naša štruktúra objektov bude nejaký graf, ktorý reprezentuje sieť počítačov. Každá trieda v hierarchii modeluje počítač s rovnakým OS.

Chceme na tejto štruktúre implementovať nejakú údržbu, teda nejaké operácie nad počítačmi, napr. zistiť koľko je voľného miesta vo file systeme. Pre každú triedu (napr. UnixPC, WinNTPC, LinuxPC...) bude implementácia tejto operácie ina. Takže urobíme abstraktnú triedu PC a v nej zadefinujeme operáciu GetFreeSpace(), pričom každá z podtried ju implementuje podľa potreby.

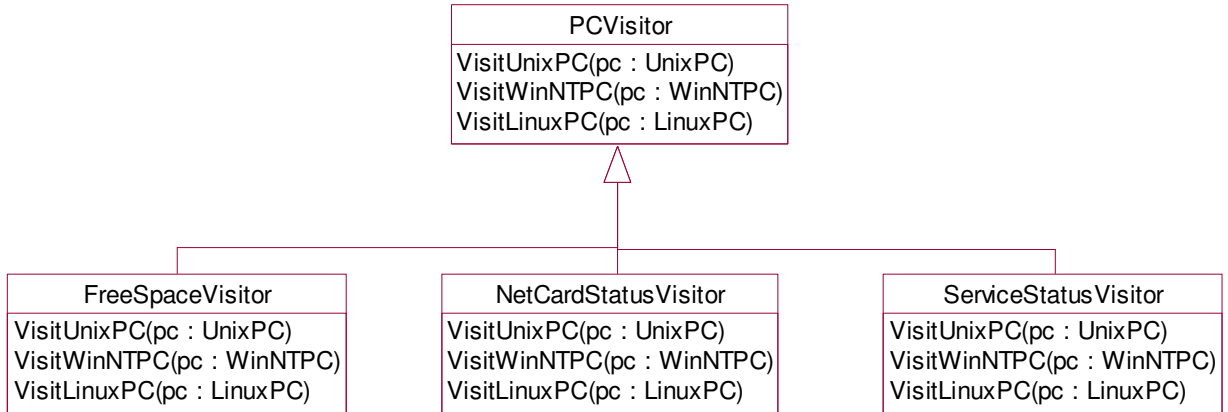
Ak však budeme v budúcnosti potrebovať pridať novú operáciu napr. či beží nejaký servis, alebo zistiť status sieťovej karty, tak budeme musieť meniť celú hierarchiu tried. Je to dost neflexibilné, a navyše takéto operácie logicky nepatria do triedy PC, sú to operácie nad jej správaním a stavom.

Preto vytiahneme tieto operácie do špeciálneho objektu Visitor. Pre každú operáciu bude samostatný Visitor a pre každý element hierarchie bude Visitor obsahovať špeciálnu metódu na spracovanie (metóda navštevovania danej triedy).

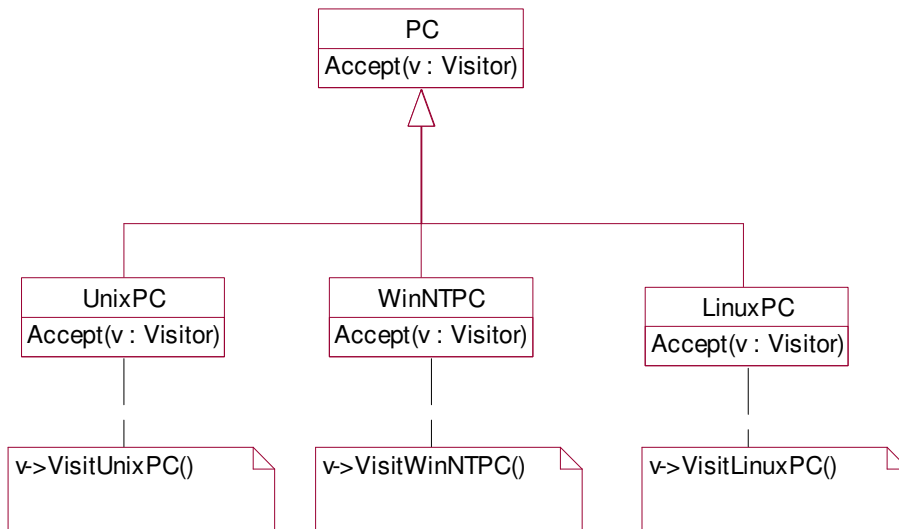
Takto by mohla vyzeráť pôvodná hierarchia.



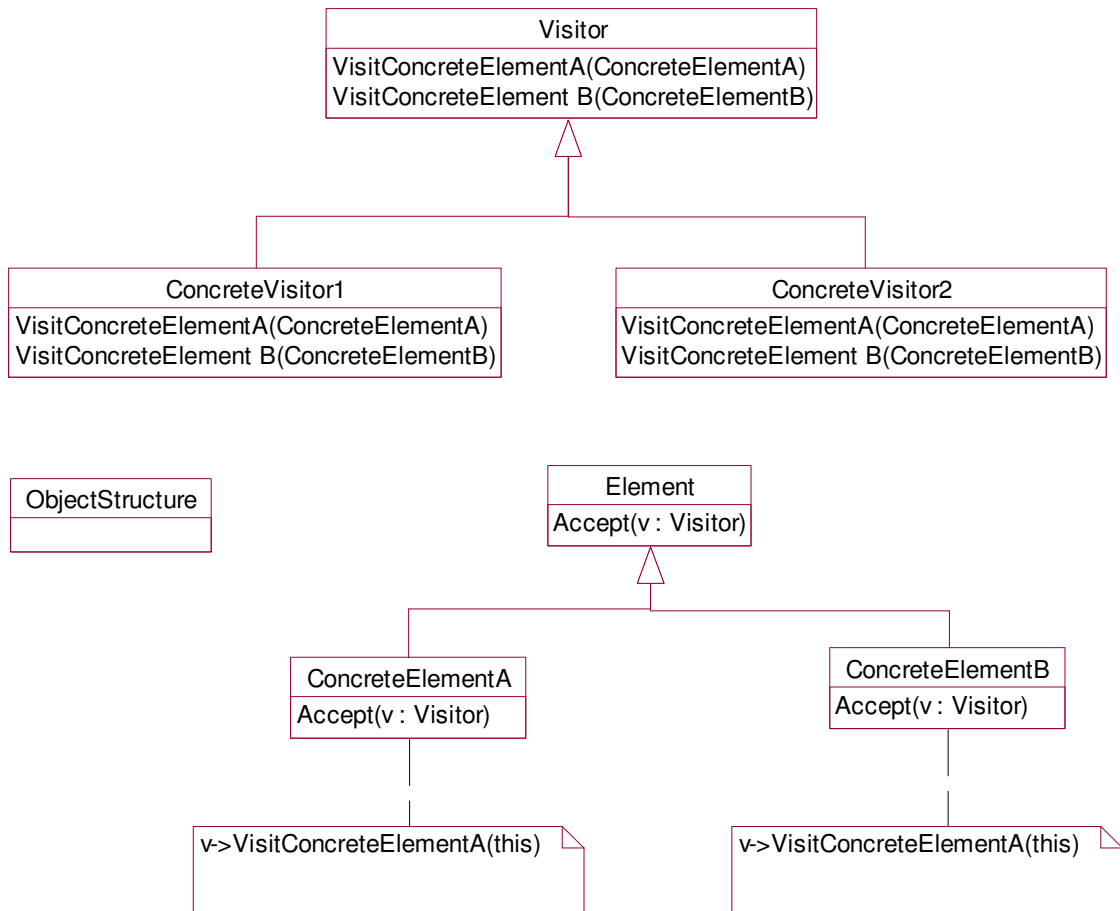
Pridanie novej operácie do tejto hierarchie je však dost neflexibilné a vyžaduje si zásah do každej triedy (je interfacesu). Preto vytiahneme túto operáciu do samostatnej triedy Visitor.



Nasledne upravime povodnu hierarchiu.



2.3.1 Struktura



2.3.2 Použitie

- 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 operacie 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 iba zriedka, ale castejsie sa pridavaju nove operacie 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, ktorý kontroluje prístup ku tomu zastupovanemu objektu.

3.2 Chain of responsibility

Obsah: umožniť viac ako len jednému objektu spracovať nejakú požiadavku (request). Cieľové objekty sú zretazované a požiadavka je postupne predávaná každému až kým ju niekto neosetri.

Napr. keď máme help button v dialogu, tak pri jeho zavolaní sa najprv volá metóda dialogu (aby user dostal konkrétnu help závislú od dialogu). Ak dialog nereaguje na túto metódu, tak sa request posúva vyššie napr. aplikácii ktorá ho spracuje a zobrazí svoj help (ktorý už nebude až tak špecifický, pretože aplikácia nevie, že request pochádza z dialogu).

3.3 Command

Obsah: oddeliť objekt ktorý vydáva nejaký príkaz (klient) od objektu, ktorý ho spracováva. Príkaz je zabalený do smostatného objektu, ktorý sa potom môže zaradovať do fronty, logovať, prípadne inak parametrizovať. Klient nie je závislý na konkrétnej implementácii príkazu, ani na objekte ktorý ho implementuje.

Umožňuje to parametrizovať klientov, tak že miesto toho aby volali konkrétnu metódu nejakého konkrétneho objektu, zavolajú štandardnú metódu špeciálneho objektu Command, ktorý potom sám rozhodne čo ďalej spraví.

Využíva sa napr. v GUI v menu. Keď sa zavola položka menu tak tá má priradený nejaký Command objekt a toho metódu Execute potom zavola.

3.4 Interpreter

...nerozumiem...

3.5 Iterator

Obsah: sekvenčne prehládvanie v agregovaných objektoch (kolekciách)

Majme nejaký agregovaný objekt napr. zoznam (alebo strom, množinu...) a iterator nad týmto objektom. Agregovaný objekt definujeme pomocou operácií add(), remove, count() a pod.

Keďže chceme aby aplikácia bola nezávislá od implementácie agregovaného objektu, potom iterator nad touto štruktúrou musí spĺňať nejaký všeobecný interface. Potom však máme problém aby iterator bol pripravený na "možno" meniacu sa štruktúru agregovaného objektu.

Teda agregovaný objekt bude zodpovedný za vytvorenie vhodného iteratora, ktorý bude rozumieť jeho štruktúre.

3.6 Mediator

Obsah: modelovať interakciu v skupine objektov prostredníctvom samostatného objektu a tým znížiť závislosti medzi jednotlivými objektami.

To umožňuje aby jednotlivé objekty v skupine nereferovali priamo ostatné objekty (**loose coupling**) a zároveň aby sa mohla jednoducho **meniť** ich **interakcia** bez nutnosti zmeny objektov.

Napríklad dialog box v GUI. Zvyčajne bývajú jednotlivé komponenty (checkboxy, buttony, list boxy...) nejakým spôsobom previazané, teda ak sa zmení jeden komponent potom sa môže zmeniť jeden alebo viac iných komponentov (napr. vyplnenie text boxu môže enablenúť nejaký button).

3.7 Memento

Obsah: zapamätat si stav objektu (externalizovať ho) bez porušenia encapsulácie, tak aby sme mohli neskôr objekt príviesť do toho istého stavu.

Používa sa pri implementovaní Undo operácií.

3.8 Observer

Obsah: niekoľko objektov závislých na obsahu jedného objektu sú automaticky notifikované v prípade jeho zmeny.

Napr. keď máme data v Exceli tak môžeme tieto data zobrazovať v tabuľke, alebo v grafe a pod.

3.9 State

Obsah: povoliť objektu zmeniť správanie ak sa zmení jeho stav. V tomto prípade sa vlastne mení trieda objektu.

3.10 Strategy

Stratégia vybere algoritmus z triedy navzájom ekvivalentných 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.

- Rozdeovanie posielateľov sprav od prijímateľov

5 Links

Patterns (referencie)

<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/>