VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

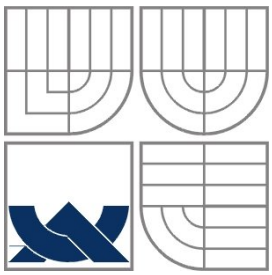
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

VYUŽITÍ NÁVRHOVÝCH VZORŮ V PROSTŘEDÍ .NET

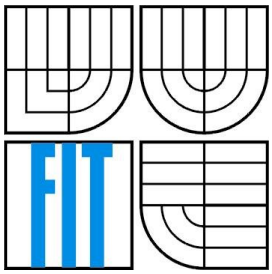
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. Stanislav Miško
AUTHOR

BRNO 2008



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

VYUŽITÍ NÁVRHOVÝCH VZORŮ V PROSTŘEDÍ .NET

APPLICATION OF DESIGN PATTERNS IN .NET ENVIRONMENT

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. Stanislav Miško

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. Vladimír Bartík, Ph.D.

BRNO 2008

Abstrakt

Práca rozoberá problematiku návrhových vzorov v prostredí .NET. Teoretická časť sa skladá zo štyroch kapitol. V prvej časti sa venujem kódu veľkých aplikácií z hľadiska prehľadnosti, robustnosti, znovupoužiteľnosti. V druhej časti stručne popisujem štruktúru 3-vrstvovej aplikácie. Vo zvyšných kapitolách sa venujem návrhovým vzorom, poukazujem na přínos pri vytváraní rozsiahlych aplikácií. Súčasťou práce je popis implementácie vybraných návrhových vzorov.

Klíčová slova

.NET, C#, návrhové vzory.

Abstract

The thesis deals with the design patterns in .NET environment. The theoretical part consists of four chapters. The first one is about the code of enterprise applications focusing on readability, robustness and reusability. The second chapter briefly describes the structure of 3-tier applications. The rest of work is about the design patterns pointing out its help in designing large applications. The thesis also describes the implementation of chosen design patterns.

Keywords

.NET, C#, design patterns.

Citace

Miško Stanislav: Využitie návrhových vzorov v prostredí .NET. Brno, 2008, diplomová práce, FIT VUT v Brně.

Využitie návrhových vzorov v prostredí .NET

Prehlásenie

Prehlasujem, že som túto diplomovú prácu vypracoval samostatne pod vedením Ing. Vladimíra Bartíka, Ph.D. Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

……
Stanislav Miško
15.5.2008

Poďakovanie

Na tomto mieste by som sa rád poďakoval najmä môjmu vedúcemu Ing. Vladimírovi Bartíkovi, Ph.D za ochotu, s ktorou zodpovedal moje časté dotazy a pomáhal mi pri vzniku tejto práce. Ďakujem svojím blízkym za zhovievavosť a trpezlivosť , ktorú so mnou v období vzniku tejto práce mali.

© Stanislav Miško,2008.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

Obsah 1
1 Úvod 4
2 Objektovo orientovaný návrh 6
2.1 Správny návrh 6
2.2 Zlý návrh 6
3 UML 8
4 3-vrstvová architektúra 9
4.1 Definícia 9
4.2 Dátová vrstva 10
4.3 Logická vrstva 10
4.4 Prezenčná vrstva 10
5 Návrhové vzory 11
5.1 Creational Patterns (vytvárajúce) 11
5.1.1 Singleton 11
5.1.2 Builder 12
5.1.3 Prototype 12
5.1.4 Factory Method 13
5.1.5 Abstract Factory 13
5.2 Structural Patterns (štrukturálne) 14
5.2.1 Decorator 14
5.2.2 Composite 15
5.2.3 Adapter 16
5.2.4 Facade 17
5.2.5 Proxy 17
5.3 Behavioral Patterns (správanie) 18
5.3.1 Template Method 18
5.3.2 Command 19
5.3.3 Iterator 19
5.3.4 Observer 20
5.3.5 State 20
5.3.6 Strategy 21
6 Enterprise návrhové vzory 22
6.1 Domain model 22
6.2 Identity field 23

6.3 Lazy load 23
6.3.1 Lazy initialization 23
6.3.2 Virtual proxy 23
6.3.3 Value holder 23
6.3.4 Ghost 24
6.4 Service layer 24
6.4.1 Domain facade 24
6.4.2 Operation script 24
6.5 Transaction script 24
6.6 Transform view 25
7 Popis platformy .NET 26
7.1 .NET Framework 27
7.2 Common Language Runtime 27
7.3 MSIL 29
7.4 ASP.NET 30
7.5 C# 31
7.6 Attributes 32
8 Vzorová aplikácia 33
8.1.1 Dátový zdroj 34
8.1.2 Organizácia projektov 34
9 Implementované návrhové vzory 36
9.1 Singleton 36
9.2 Abstract Factory 38
9.2.1 Data Access Object 38
9.3 Composite 39
9.4 Facade 40
9.5 Proxy 41
10 Enterprise návrhové vzory 43
10.1 Domain Model 43
10.2 Identity field 44
10.3 Lazy load 44
10.4 Domain facade 46
10.5 Transaction script 46
10.6 Transform view 46
11 Poznatky z implementácie 48
11.1 Atribúty 48
11.2 Master pages 48

11.3 Kľúčové slovo using 49
12 Záver 50
Literatúra 51
Zoznam príloh 52

1 Úvod

Práca predstavuje náhľad do tvorby viac vrstvových aplikácií. Pohybujem sa istý čas v praxi, žiaľ so správnym implementovaním rozshiahlej aplikácie som sa ešte nestretol. Rád by som vo svojej práci predstavil spôsoby ako vytvárať prehľadné, robustné, rozšíriteľné a znovupoužiteľné aplikácie.

Domnievam sa, že úsilie ktoré musí byť vynaložené na návrh aplíkacie pomocou návrhových vzorov, sa niekoľko násobne vráti neskôr pri údržbe prípadne rozširovaní systému. V realite sa však návrhu informačného systému nevenuje dostatočný čas, aplikácie sa tak neskôr stávajú neohybné a náročné na údržbu.

V druhej kapitole sa venujem objektovo orientovanému návrhu, poukazujem na správne a špatné koncepty.

Nasledovná kapitola zoznamuje čitateľa s UML, nakoľko v nasledujúcich kapitolách bude použité pri popise návrhových vzorov.

Štvrtá kapitola má za cieľ stručne popísať princíp 3-vrstvovej aplikácie, zmysel jednotlivých vrstiev.

V nasledovnej kapitole sa už venujem návrhovým vzorom. Popisujem vybrané návrhové vzory, ktoré považujem za najčastejšie používané. Sú zo skupiny Gang of Four, bez týchto základov nie je možné vytvárať viacvrstvové aplikácie, kde by vrstvy spolu efektívne komunikovali. Kapitola popisuje návrhové vzory v teoretickej rovine, ku každému vzoru je priložený UML diagram pre názornosť.

Piata kapitola sa venuje tzv. Enterprise návrhovým vzorom, ktoré boli navrhnuté s cieľom riešiť typické problémy rozsiahlych systémov. Opäť je to teoretická kapitola s cieľom objasniť zmysel jednotlivých vzorov. Pri tvorbe veľkých systémov sa programátor stretne s problémami, ktoré sa opakujú v každom systéme. Prístup k dátam, spôsoby predávania dát, logovanie a pod. Tieto momenty sa snažia zachytiť Enterprise návrhové vzory. Opäť som vybral len niektoré, ktoré budú použité v ukážkovej aplikácií.

Siedma kapitola popisuje platformu, ktorú som si zvolil na implementáciu ukážkovej aplikácií. Venujem sa základným častiam platformy.

V nasledovnej kapitole sa venujem bližšie implementovanej ukážkovej aplikácií, v ktorej implementujem niektoré návrhové vzory popisané v predošlých kapitolách. Predstavuje úvod k aplikácií, stručne zoznamuje čitateľa s implementovaným riešením.

Deviata kapitola popisuje implementáciu návrhových vzorov, s ktorými sa čitateľ bližšie zoznámil v piatej kapitole. Obsahuje hlbší popis a často je riešeniu návrhového vzoru priložený obrázok.

Kapitola desať popisuje enterprise návhové vzory implementované v aplikácií. Naväzuje na kapitolu s označením šesť, ktorá rozoberá problematiku v teoretickej rovine.

Na záver ešte prikladám poznatky, ktoré som nabral počas implementácie. Momenty, ktoré považujem za užitočné.

Nasledovnou kapitolou je záver, ktorý popisuje dosiahnuté výsledky a prípadné pokračovanie projektu.

2 Objektovo orientovaný návrh

Objektovo orientovaný návrh (OON) zvyšuje možnosť pridávať novú funkcionalitu v budúcnosti.

Obchodná činnosť potrebuje časté zmeny, čo znamená ťažkú úlohu pre návrhára vytvoriť aplikáciu, ktorá je schopná sa rýchlo adaptovať. Dobre navrhnuté aplikácie pomáhajú zákazníkom predbehnúť ich konkurenciu, špatne navrhnuté aplikácie vytvárajú proces neohybný a stojí to príliš veľa, aby sme ho mohli znova použiť.

2.1 Správny návrh

Správne návrhnuté aplikácie ponúkajú komponenty, ktoré sú oveľa robustnejšie, udržovateľnejšie a viac znovupoužiteľné. Také aplikácie by sa mali byť schopné adaptovať na zmeny v obchodnej činnosti bez dopadu na návrh. Napr. aplikácia pre banku by mala byť schopná podporovať nové typy účtov bez zmeny existujúceho návrhu. Hlavné črty dobrého návrhu sú:

- Udržovateľnosť tzn. ľahkosť akou je systém alebo komponenta modifikovaná na zmenu prostredia, zvýšenia výkonu, opravy chýb atď. Správne navrhnuté aplikácie vyžadujú menej zdrojov na udržovanie a zmeny systému.
- Znovupoužiteľnosť tzn. stupeň použiteľnosti komponenty vo viac ako jednom výpočetnom programe alebo systéme.
- Robustnosť tzn. stabilita systému na extrémne situácie (napr. chybné vstupy). Robustné systémy sú menej mimo prevádzku a redukujú cenu udržovania.

2.2 Zlý návrh

Nikto neplánuje vytvoriť zle navrhnutú aplikáciu. Často sa to ale stane v dôsledku nedostatku skúsenosti alebo aplikácia bola navrhnutá príliš rýchlo, aby sa stihol termín. Slabo navrhnuté aplikácie majú obvykle spoločné nasledovné problémy:

- Sú neohybné. Návrh je neohybný lebo nemôže byť jednoducho zmenený. Napr. jediná zmena úzko vnútorne závislej komponenty neohybného systému spôsobuje zmeny v závislých komponentách. Keď taký program narastíe do veľkosti, ľudia zodpovední za návrh alebo údržbu nedokážu odhadnúť rozsah kaskádovej zmeny tzn. nie sme schopní odhadnúť dopad na systém a jeho cenu.
- Sú krehké. Po jedinej zmene slabo navrhnuté systémy majú tendenciu zlyhať na mnohých miestach. Jednoduchá zmena v jednej časti aplikácie vedie k chybám v iných častiach, ktoré sa javia byť úplne nesúvisiace. Oprava spôsobených chýb vedie ešte

k viacerým podobným problémom. Krehkosť systému znemožňuje manažérom odhad budúcich kvalít produktu.

- Nie sú znovupoužitelné. Návrh je náročné opäť použiť, ak sú jeho časti vysoko závislé na iných detailoch, ktorú pre nás nie sú zaujímavé. Ak je návrh vysoko vnútorne závislý, ostatný návrhári budú odradení od množstva práce potrebnej na vyňatie žiadaného kusu návrhu od častí, ktoré sú nežiadané. Vo väčšine prípadov je cena oddelenia komponenty vyššia ako cena opätovného vývoja.

3 UML

UML (Unified model language) je štandardizovaný jazyk na špecifikovanie, vizualizáciu, konštrukciu a dokumentáciu častí softwarevého systému. UML predstavuje zbierku často používaných praktík, ktoré boli úspešné vo veľkých a komplexných systémoch. UML je dôležitou časťou vo vývoji objektovo-orientovaného softvéru a procesu vývoja. Používa najmä grafickú reprezentáciu na popis softvérového projektu. Pomáha skúmať potenciále návrhy na systém, validuje architektonický návrh softvéru.

Dôvody, prečo som si zvolil UML:

1. Je nezávislý na programovacom jazyku a procese vývoja.
2. Umožňuje zobraziť požiadavky vo vizuálnej, ľahko čitelnej forme.

UML definuje notáciu a sémantiku nasledovných domén:

- Use Case Model (diagram prípadov použitia) – popisuje hranice a vzájomné pôsobenie medzi systémom a užívateľom. Korešponduje v niektorých ohľadoch s modelom požiadaviek.
- Interaktívny alebo komunikatívny model - popisuje ako objekty v systéme budú na seba vzájomne pôsobiť aby fungovali správne
- Stavový alebo dynamický model – popisuje stavy a podmienky, ktoré triedy predpokladajú.
- Model tried alebo logický model – popisuje triedy a objekty, ktoré vytvárajú systém.
- Komponentový model – popisuje softvér a niekedy hardverové kompenenty, ktoré vytvárajú systém

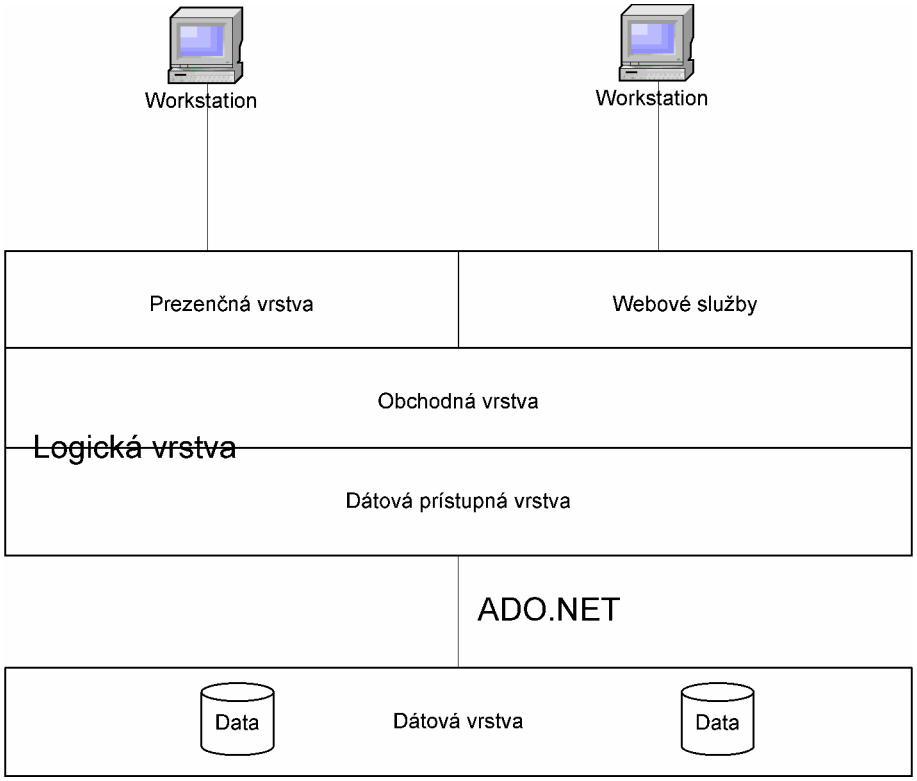
4 3-vrstvová architektúra

4.1 Definícia

3-vrstvová aplikácia je program, ktorý je organizovaný do 3 hlavných disjunktných vrstiev. Sú to

- Prezenčná vrstva (Presentation Tier)
- Logická vrstva (Logical Tier)
- Dátová vrstva (Data tier)

Každá vrsvtva môže byť nasadená na geograficky oddelených počítačoch v sieti. Niektorí architekti rozdeľujú logickú vrstvu do dvoch podtried Obchodná a Dátová prístupná vrstva za účeľom zvýšenia rožšíriteľnosti a transparentnosti. Vrstvy môžu byť nasadené na fyzicky odlišné počítače. Charakteristický znak aplikácie založenej na vrstvách je, že jednotlivé vrstvy komunikujú len so susednými vrstvami. Napr. prezenčná vrstva komunikuje priamo s biznis vrstvou, ale nikdy by nemala komunikovať s vrstvou dátovou.



Obr. 1. Schéma 3-vrstvovej aplikácie

4.2 Dátová vrstva

Vrstva je zodovedná za doťahovanie, ukladanie a aktualizovanie informácií, preto je táto vrstva najčastejšie reprezentovaná ako databáza. Súčasťou vrstvy sú aj uložené procedúry, zvyšujú výkon a transparentnosť aplikácie.

4.3 Logická vrstva

Je mozog 3-vrstvovej aplikácie. Niektorí architekti nerozlišujú biznis vrstvu a dátovú prístupovú vrstvu. Hlavným argumentom je, že pridaná vrstva znižuje výkon. Rozdelením vrsvty na dve však získame aj výhody, ktoré podľa mojho názoru prevyšujú nad nevýhodami. Sú to napr.

- Zvýšenie transparentnosti kódu
- Podporuje zmeny v dátovej vrstve. Je možné zmeniť databázu bez toho, aby sme sa dotkli biznis vrstvy resp. prípadné zmeny by boli minimálne.

4.4 Prezenčná vrstva

Vrstva je zodpovedná za komunikáciu s používateľom, prípadne s webovými službami a používa objekty s biznis vrstvy ako odpoveď na udalosti z užívateľského rozhrania.

5 Návrhové vzory

Návrhové vzory pomájú architektom vytvárať aplikácie, v ktorých použijú výhody Gang of Four (GoF) návrhových vzorov. Objektovo orientovaná aplikácia sa tak stáva flexibilnejšia, znovupoužiteľnejšia a jednoduchá na údržbu. Programovanie pomocou návrhových vzorov vyžaduje viac úsilia pri programovaní na začiatku, ale po istom čase sa údržba aplikácie stáva oveľa jednoduchšia, spotrebuje menej času na údržbu, je menej náchylná na chyby. Ak sa programátor stará o kód, ktorý používa návrhové vzory, znižujú sa prípady, že je v kríze, naopak sa zvyšujú možnosti ako aplikáciu rozšíriť na základe pôvodného návrhu.

GoF vzory reprezentujú jadro návrhových vzorov. Je ich celkovo 23, sú organizované do troch skupín.

5.1 Creational Patterns (vytvárajúce)

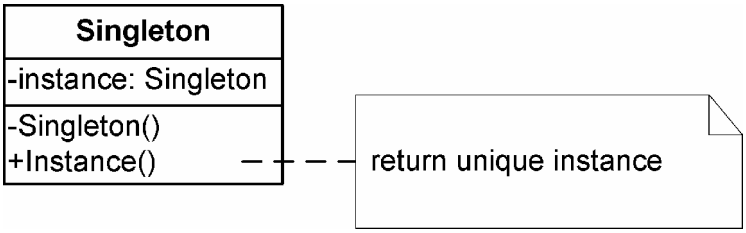
Všetky vytvárajúce návrhové vzory sa zaoberajú spôsobmi ako vytvárať inštancie objektov. Je to dôležité, pretože program by nemal závisieť na tom, ako sú objekty vytvárané. Prirodzená cesta ako vytvoriť objekt je pomocou konštruktora, avšak týmto spôsobom je informácia o vytvorení objektu zaznamená na tvrdo v kóde. Môže to viezť k problémom v návrhu alebo pridávaniu komplexnosti návrhu.

V mnohých prípadoch konkrétny typ objektu, ktorý je vytvorený, závisí od rôznych atribútov, ktoré sa môžu meniť. Z toho dôvodu je vhodné vyabstrahovať proces tvorby objektu do samostatnej jednotky, aby sa stal program všeobecnejší, flexibilnejší.

5.1.1 Singleton

Úlohou vzoru je zaistenie existencie maximálne jednej inštancie triedy a umožňuje globálny prístup k nej.

Zobereme triedu a necháme ju, nech sa stará o svoju jedinú inštanciu. Zabraňujeme akejkoľvek inej triede aby si sama vytvorila ďaľšiu inštanciu. Ak chceme vytvoriť inštanciu, musíme pristúpiť k samotnej triede. Takisto vytvárame globálny prístupový bod k inštancií. Kedykoľvek potrebujeme inštanciu, musíme požiadať triedu a ona vráti späť tú jedinú existujúc. Je možné singleton implementovať spôsobom, že je objekt vytvorený až v momente, keď k nemu prvý krát pristúpime, čo môže byť dôležité pri objektoch, ktoré sú náročné na zdroje.

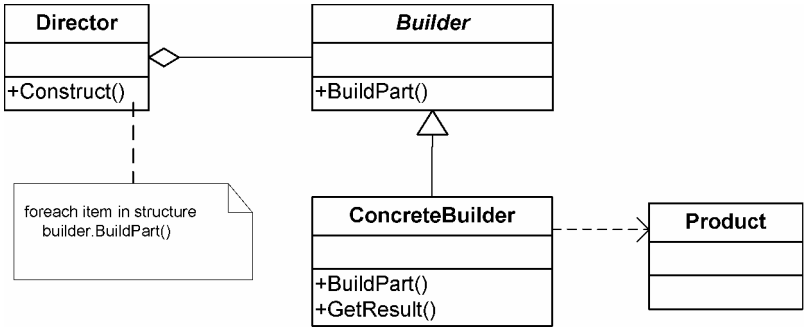


Obr. 2. Singleton

5.1.2 Builder

Návrhový vzor oddeľuje vytváranie komplexného objektu od jeho reprezentácie tak, aby na základe potrieb programu mohli byť vytvorené rôzne reprezentácie.

Návrhový vzor použijeme, ak chceme zapúzdriť vytváranie objektu a umožniť mu, aby bol vytváraný po krokoch. Používame ho ak chceme zapúzdiť spôsob ako je objekt vytváraný, umožniť objektu aby bol vytváraný vo viacerých krokoch a rôznymi procesmi, skrýva internú reprezentáciu produktu od klienta, implementácia výsledného produktu môže byť zamenená nakoľko klient vidí len abstraktné rozhranie. Často sa používa na vytváranie zmiešaných štruktúr. Nedostatkom vzoru je, že vytváranie objektu vyžaduje viac doménovej znalosti klienta ako pri použití napr. vzoru Factory.

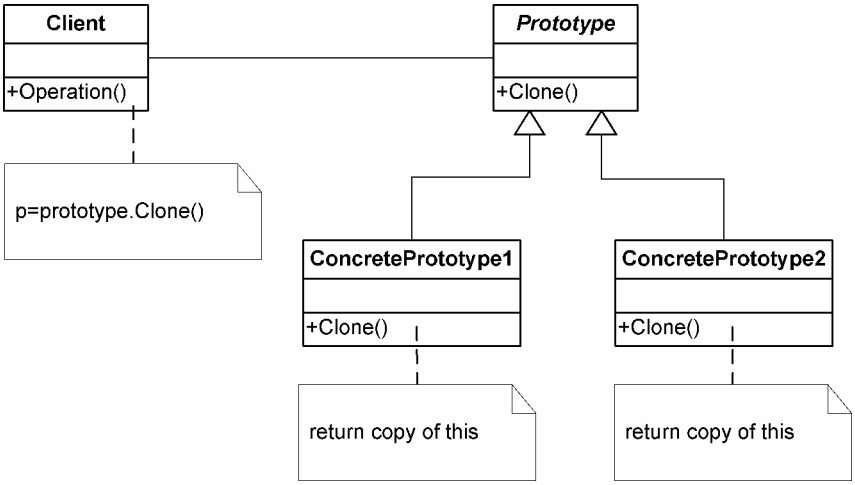


Obr. 3. Builder

5.1.3 Prototype

Návrhový vzor Prototype začína s inštanciou triedy a kopíruje ju alebo klonuje s cieľom vytvoriť novú inštanciu. Nové inštancie možu byť ďalej prispôsobované použitím ich verejných metód.

Používame ho ak je vytváranie danej triedy buď náročné alebo komplikované. Výhodami vzoru sú že, skrýva komplexnosť vytvárania nových inštancií od klienta, umožňuje klientovi generovať objekty, u ktorých nepoznáme typ. V niektorých prípadoch môže byť kopírovanie objektu výkonnejšie ako vytváranie nového objektu. Prototype by sme mali zvážiť keď systém musí vytvárať objekty rôznych typov v zložitej hierarchií tried. Nevýhodou je, že niekedy vytvorenie kópie objektu môže byť príliš komplikované.

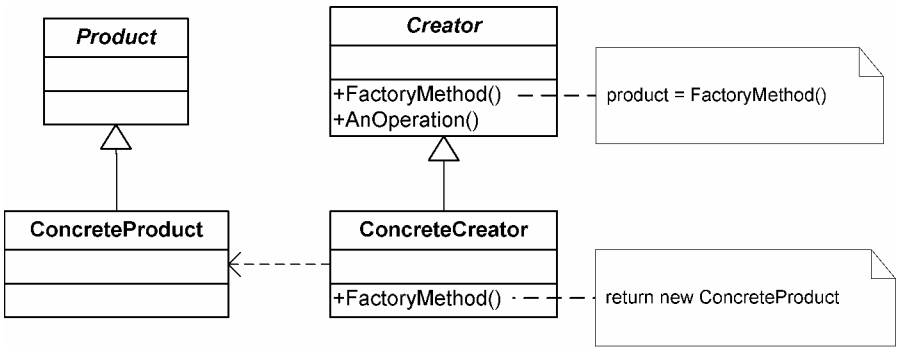


Obr. 4. Prototype

5.1.4 Factory Method

Predstavuje jednoduchú rozhodovaciu triedu, ktorá vracia jednu z viacerých podtried abstraktného základného typu na základe dostupných dát.

Tento návrhový vzor umožňuje podtriedam rozhodovať, z ktorej triedy urobí inštanciu. Návrhový vzor umožňuje podtriedam rozhodovať počas behu, ktorú inštanciu vytvoria.Vytváracia trieda je napísaná bez konkrétnej znalosti, ktorá trieda bude inštanciovaná a to zámerne. Rozhodnutie necháva na podtriedy.

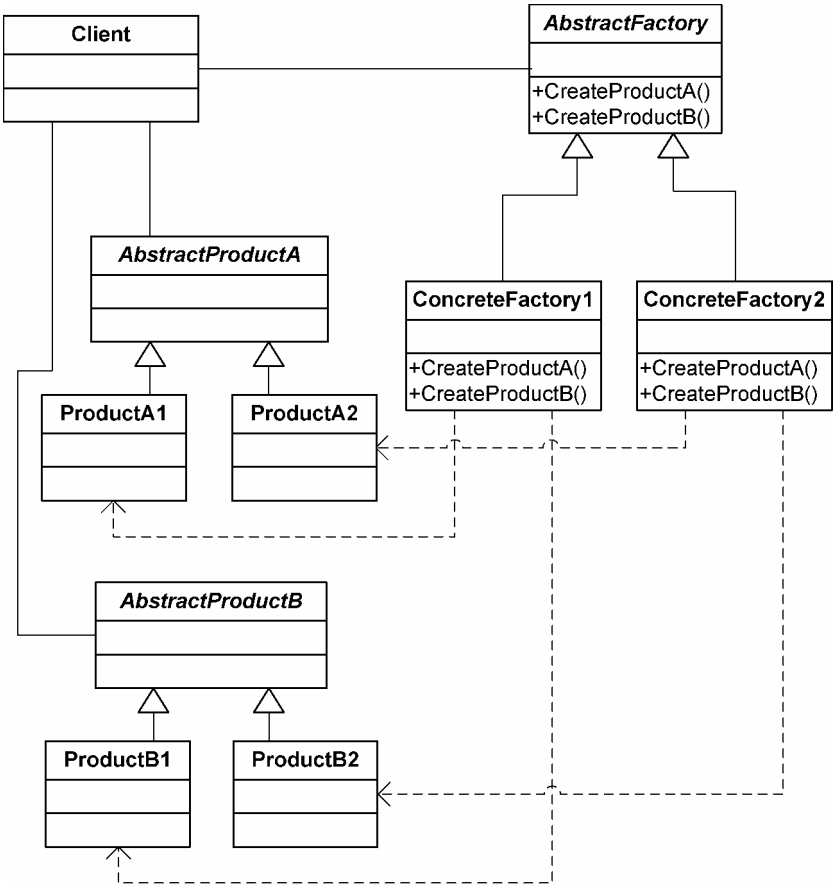


Obr. 5. Factory method

5.1.5 Abstract Factory

Zabezpečuje rozhranie na vytváranie rodín súvisiacich alebo závislých objektov bez špecifikovania konkrétnej triedy.

Umožňuje vytvárať skupinu objektov, dovoľuje klientovi použiť abstraktné rozhranie na vytvorenie množiny súvisiacich produktov bez vedomosti o konkrétnom produkte, ktorý vlastne v skutočnosti produkujeme.V tomto prípade je klient oddelený od všetkých špecifík konkrétnych produktov.



Obr. 6. Abstract factory

5.2 Structural Patterns (štrukturálne)

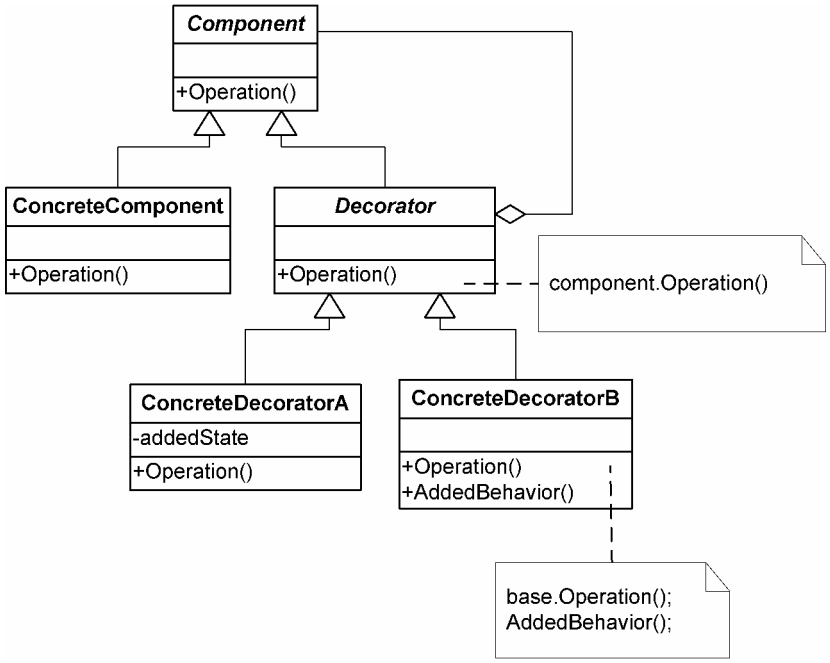
Štrukturálne vzory popisujú ako môžu byť triedy a objekty kombinované, aby vytvárali väčšie štruktúry resp. znázorňujú väzby medzi entitami. Ich cieľom je zprehľadniť kód prípadne vytvoriť novú funkčnosť skladaním objektov.

5.2.1 Decorator

Umožňuje modifikovať spŕavanie objektu bez vytvorenia zdedenej triedy.

Dekorátor môže byť použitý v prípade, že chceme umožniť rozšírenie (dekoráciu) funkčnosti nejakej triedy počas jej behu. Funguje to pridaním novej triedy, ktorá obaľuje pôvodnú triedu. Toto obaľovanie je obvykle dosiahnuté pomocou predávania pôvodného originálneho objektu ako parameter konštruktora dekorátoru. Dekorátor implementuje novú funkčnosť, ale funkčnosť ktorá nie je nová, ostáva pôvodná. Trieda dekorátoru musí mať rovnaké rozhranie ako rozhranie pôvodnej triedy.

Je to alternatíva k vytváraniu podtried. Podtriedy pridávajú správanie počas kompilácie, dekorátor dokáže vytvoriť nové chovanie počas behu.



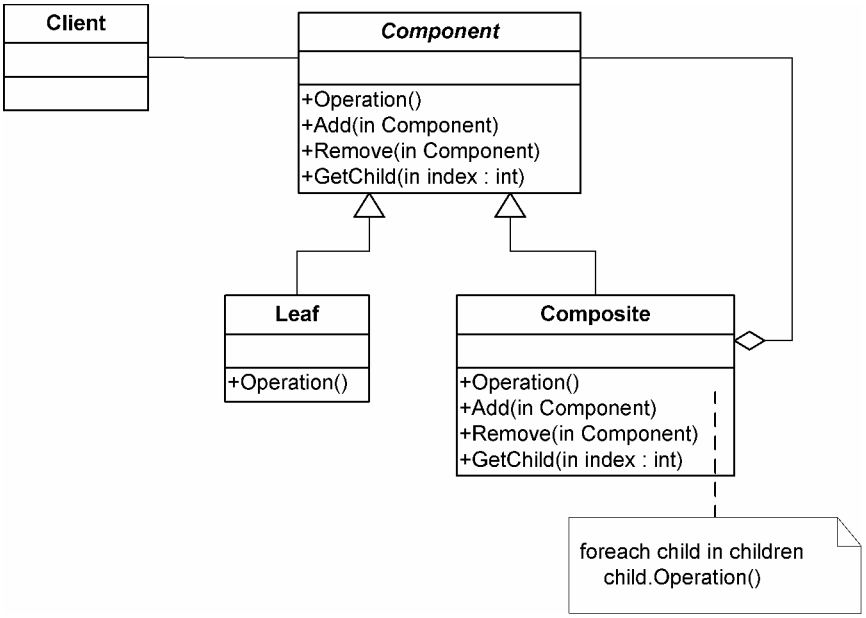
Obr. 7. Decorator

5.2.2 Composite

Composite predstavuje možnosť ako usporiadať jednoduché a kompozitné objekty. Cieľom je, aby sme pristupovali k jednoduchým aj zloženým objektom jednotným spôsobom.

Keď programujeme stromovo štruktúrované dáta, vždy musíme rozlišovať či sa jedná o koreňový list alebo o syna. Robí to kód komplexným a preto náchylný k chybám. Riešením je rozhranie, ktoré umožňuje jednotné zaobchádzanie s komplexnými aj primitívnymi objektami. V objektovo orientovaných jazykoch v tomto kontexte rozumieme pod komplexnými objektami zloženie jedného alebo viacerých podobných objektov, ktoré majú spoločnú funkcionalitu. Kľúčovým koneptom je, že umožňuje manipulovať s inštanciou jedného objektu rovnako ako keby to bola skupina objektov.

Tento návrhový vzor by sme mali použiť v prípadoch keď klienti by mali ignorovať rozdiel medzi zložením viacerých objektov a individuálneho objektu. Ak programátor zistí, že používajú hromadné objekty rovnakým spôsobom akokeby to bol jeden objekt a často má takmer identický kód na jeho obhospodárenie, potom tento návrhový vzor je dobrá voľba. Je menej náročné pristupovať k primitívnym a komplexným objektom ako k homogénnym.



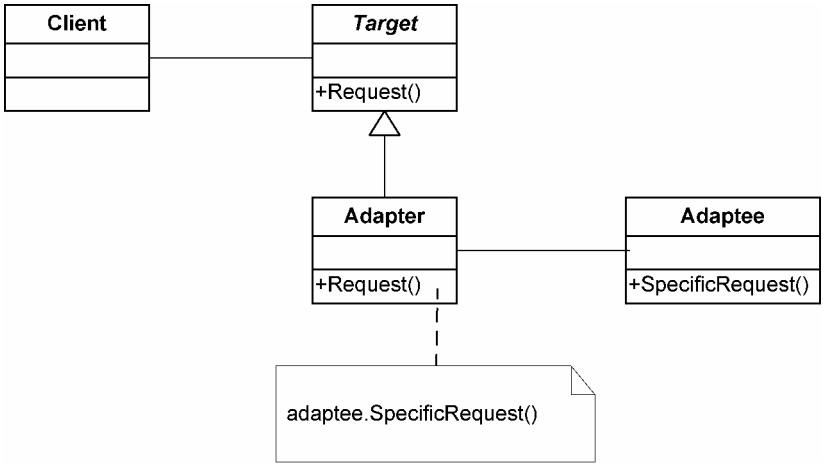
Obr. 8. Composite

5.2.3 Adapter

Adapter sa používa na zmenu rozhrania triedy na rozhranie inej triedy, aby ju bolo možné používať iným požadovaným spôsobom.

Tento vzor nám umožňuje použiť klienta s nekompatibilnými rozhraniami a to tak, že vytvoríme adaptér, ktorý urobí konverziu. Je to z dôvodu oddeliť klienta od implementovaného rozhrania a ak očakávame, že sa po čase rozhranie zmení, adaptér zapúzdruje zmenu, takže nie je nutné aby bol klient modifikovaný vždy keď potrebuje pracovať oproti inému rozhraniu.

Adaptér teda konvertuje rozhranie triedy na iné rozhranie, ktoré klient očakáva. Umožňuje aby triedy spolupracovali i napriek nekompatibilným rozhraniam.



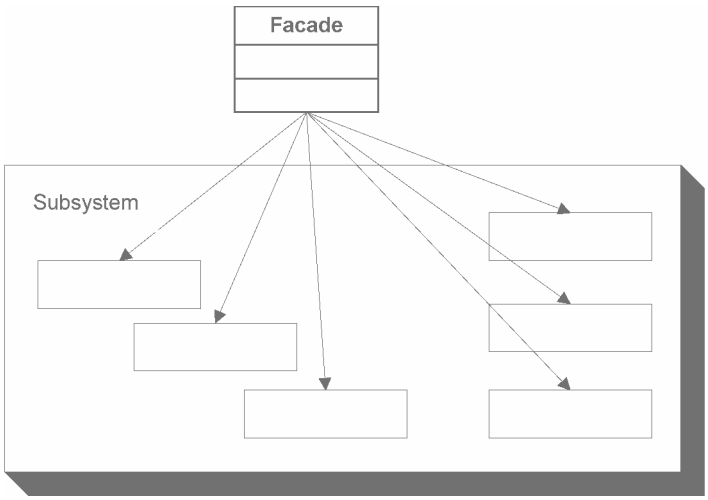
Obr. 9. Adapter

5.2.4 Facade

Zoskupuje komplexnú množinu objektov a vytvára nové jednoduchšie rozhranie na prístup k dátam.

Na implementáciu je nutné vytoriť triedu, ktorá zjednodušuje a unifikuje množinu viacerých komplexných tried, ktoré patria nejakému subsystém. Na rozdiel od mnohých vzorov, fasáda je úplne priamočiara, neexistujé žiadne rozhrania ani abstraktné triedy. Fasáda nám umožňuje sa vyhnúť úzkemu spojeniu medzi klientom a podsystémom a pomáha nám dodržiavať nový objektovo orientovaný princíp.

Fasáda poskytuje unifikované rozhranie množine rozhraní v subsystem. Fasáda definuje vyšší stupeň rozhrania, ktorý robí systém jednoduchší na použitie.

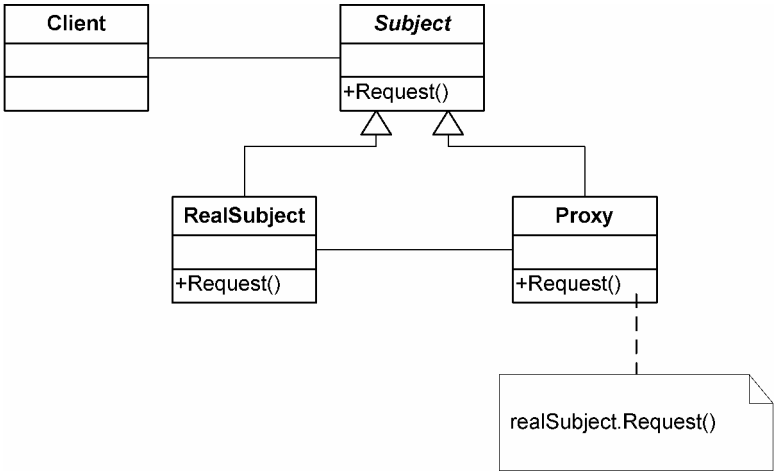


Obr. 10. Facade

5.2.5 Proxy

Predstavuje miesto, kam "odložíme" objekt, ktorý chceme neskôr použiť a z nejakého dôvodu je jeho opätovné inštanciovanie nevhodné.

Existuje mnoho variant tohto návrhového vzoru, čo majú však spoločné je, že zachytávajú volanie metódy, ktorú volá klient na predmet. Táto úroveň zmeny smerovania nám umožňuje robiť mnohé veci včetne poslania žiadosti vzdialenému objektu, zabezpečenie zástupcu pre nákladný objekt keď sa vytvára, alebo vytvoriť istú úroveň ochrany.



Obr. 11. Proxy

5.3 Behavioral Patterns (správanie)

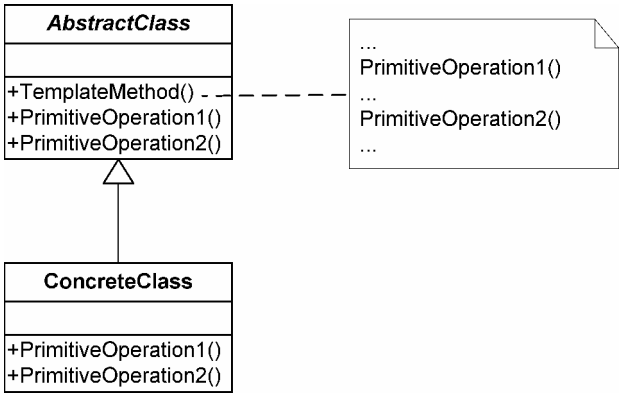
Behavioral Pattern sa zaoberá komunikáciou medzi objektami. Snaží sa o vyňatie komunikácie z objektov za účeľom zprehľadnenia kódu a zvýšenia flexibility.

5.3.1 Template Method

Predstavuje abstraktnú definíciu algoritmu. Zdedené triedy možu ovplyvniť chod algoritmu, ktorý je implementovaný v triede predka.

Definuje kostru algoritmu v metóde a prikláňa sa k podtriedam. Návrhový vzor umožňuje podtriedam predefinovať niektoré kroky algoritmu bez zmeny štruktúry algoritmu.

Vzor je celý o vytváraní šablóny pre algoritmus. Čo je šablóna? Je to len metóda, ktorá definuje algoritmus ako množinu krokov. Toto zabezpečuje, že štruktúra algoritmu ostáva nezmenená, pri čom podtriedy poskytujú časť implementácie.

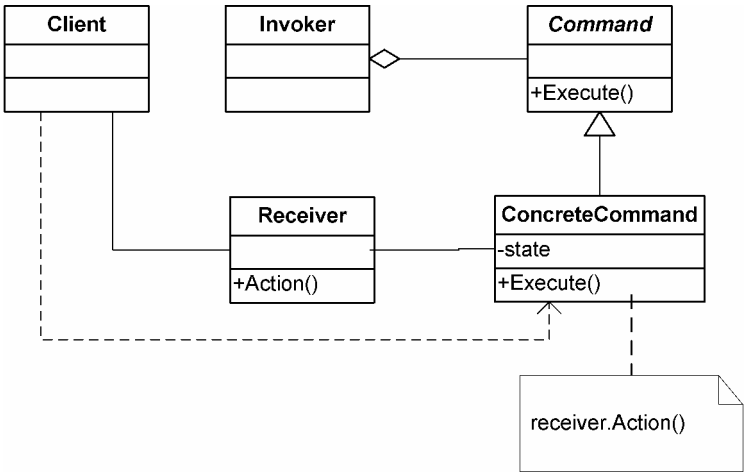


Obr. 12. Template Method

5.3.2 Command

Využitie jednoduchých objektov tak, aby reprezentovali spúšťanie príkazov a umožňuje jednoduchú podporu napr. logovania.

Návrhový vzor zapúzdruje žiadosť ako objekt, teda umožňuje možnosť parametrizácie iných objektov s rôznymi žiadosťami. Podporuje navrátanie operácií.

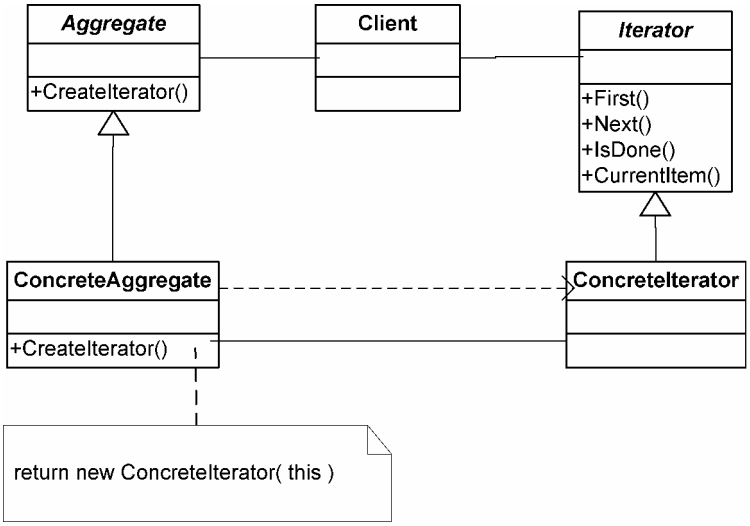


Obr. 13. Command

5.3.3 Iterator

Formalizuje spôsob akým prechádzame cez zoznam dát v triede.

Používa sa na sekvenčný prístup k elementom agregovaného objektu bez odhalenia jej vnútornej reprezentácie.



Obr. 14. Iterator

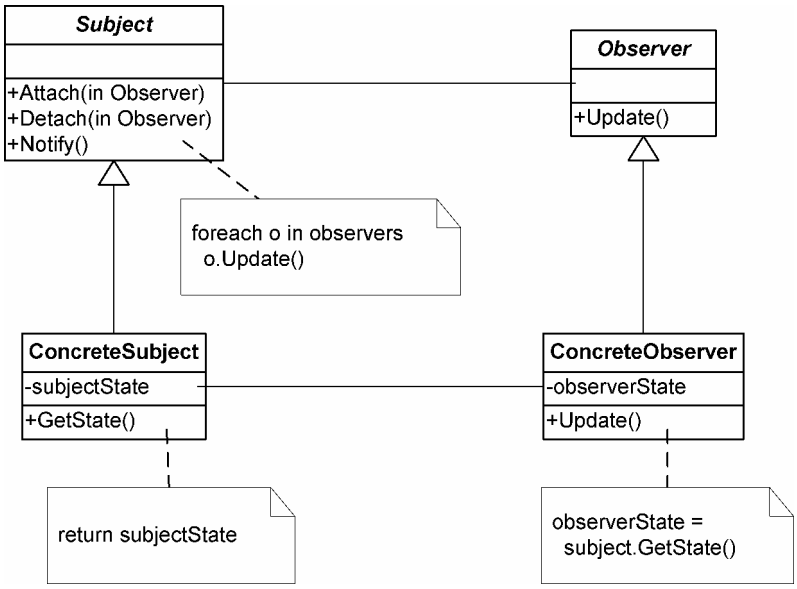
5.3.4 Observer

Umožňuje upovedomiť veľké množstvo objektov a zmene stavu iného objektu.

Návrhový vzor definuje 1:M závislosť medzi objektami tak, že ak jeden z nich zmení stav, všetci kto na ňom závisia sú upovedomení a aktualizovaní automaticky.

Predmetom je objekt, ktorý obsahuje stav a kontroluje ho. Takže máme jeden objekt so stavom. Na druhej strane, pozorovatelia používajú stav aj v prípade, že ho nevlastnia. Existuje mnoho pozorovateľov a spolihajú sa na predmet, že im povie, že sa zmenil stav.

Predmet je jediný, kto udržiava dáta, pozorovatelia sú naňom závislí aby aktualizovali svoje dáta. Toto vedie k čistejšiemu objektovo orientovanému návrhu ako umožnenie mnohým objektom aby kontrolovali rovnáke dáta.



Obr. 15. Observer

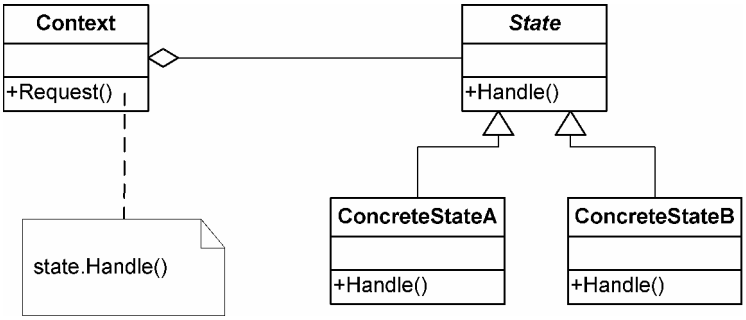
5.3.5 State

Umožňuje objektu modifikovať svoje správanie, ak sa zmení jeho stav.

Návrhový vzor zapúzdruje stav do oddelených tried a deleguje objektu reprezuntujúcemu aktuálny stav. Vieme, že správanie sa mení počas interného stavu. Ak objekt, ktorý používame môže úplne zmeniť svoje správanie, potom sa nám zdá, že objekt je vlastne inštanciovaný z inej triedy. V skutočnosti však používame kompozíciu, aby sme dali vzhľad triedy jednoduchým referencovaním iného stavu objektov.

Vzor teda umožňuje objektu aby pozmenil svoje správanie keď sa zmení jeho vnútorný stav. Zjaví sa objekt aby zmenil jeho stav.

Všeobecne sa dá pozerať na tento vzor ako na flexibilnú alternatívu k podtriedam. Ak použijeme dedičnosť na definovanie správania triedy, potom už nie je možné správanie zmeniť.

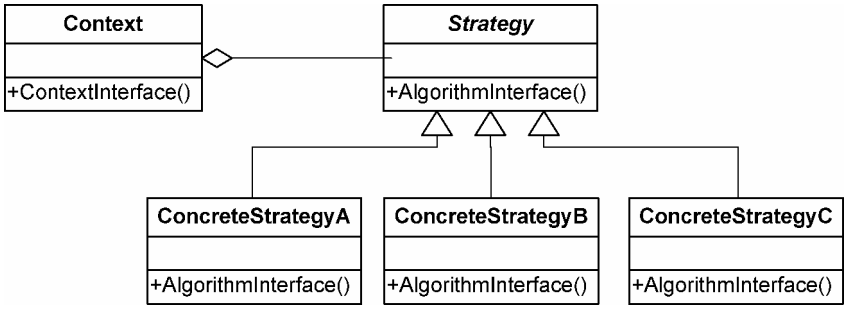


Obr. 16. State

5.3.6 Strategy

Zapúzdruje algoritmus vnútri triedy.

Návrhový vzor definuje rodinu algoritmov, každý z nich zapúzdruje a robí ich vzájomne zamenitelné. Umožňuje, aby boli algoritmy nezávislé od klienta, ktorý ich používa.



Obr. 17. Strategy

6 Enterprise návrhové vzory

Tieto vzory môžeme považovať za najlepšie praktiky ako vytvárať rozsiahle aplikácie pomocou návrhových vzorov. Často si však v praxi s návrhovými vzormi GoF nevystačíme, enterprise návrhové vzory vnikajú často ich skladaním. Sú zamerané na 3-vrstvové aplikácie.

Niektoré navrhové vzory z tejto kategórie však sú irelevantné vzhľadom na platformu .NET. Dôvodom je, že .NET má mnohé enterprise návrhové vzory už v sebe vstavané vo svojom frameworku. Napr. Enterprise návrhový vzor Record Set je v platforme .NET implementovaný ako DataTable. Iným príkladom je Iterator, kde stačí implementovať dostupné rozhranie.

Zámerom enterprise návrhových vzorov je špecializácia na typické problémy pri tvorbe rozsiahlych 3-vrstvových aplikácií.

V nasledujúcich podkapitolách spomeniem niektoré návrhové enterprise vzory, ktoré budú implementované v ukážkovej aplikácií.

6.1 Domain model

Trieda uchováva spoločne dáta aj správanie.

V mnohých prípadoch môže byť biznis logika veľmi komplexná. Pravidlá a logika popisuje veľké množstvo prípadov použitia a správania objektu a trieda sa takýmto spôsobom stáva rozsiahla. Domain model vytvára sieť prepojených objektov, kde každý objekt reprezentuje nejakú zmysluplnú individualitu.

Príkladom môže byť trieda Objednavka, ktorá bude mať určite mnohé dôležité metódy na obsluhu. Budeme však potrebovať v prezenčnej vrstve urobiť niečo špeciálne, avšak ak to umiestnime to triedy Objednavka, môže sa čoskoro táto trieda stať neprehľadnou, najmä ak sa funkcionalita použije iba v jednom špecifickom prípade.

Pri návrhu sa treba rozhodnúť či je funkcionalita všeobecná a teda sa má nachádzať v triede Objednavka, alebo je špecifická a mala by byť implementovaná napr. pomocou Transaction script, resp. sa nachádzať v prezenčnej vrstve.

Problém je ak oddelíme správanie triedy, že to môže viezť k duplicite. Správanie oddelené od Objednávky je ťažké nájsť, programátory ho nevidia a často ho nevedomky zduplikujú. Duplicita može viezť k vačšej komplexnosti a nekonzistencii. Avšak keď sa to vyskytne, tak to nie je problém opraviť.

Princíp tohto návrhového vzoru spočíva v uchovaní všetkých metód, ktoré sa týkajú logicky nejakej triedy, v nej. V prípade, že sa stane neprehľadná, začneme uvažovať o presúvaní.

6.2 Identity field

Ukladáme databázové ID záznamu v objekte aby sme uchovali identitu medzi databázovým záznamom a objektu v pamäti.

V podstate to je veľmi jednoduché. Napr. v prípade, že používame doménový model, potrebujeme vedieť, ktorý záznam v databázi odpovedá ktorému objektu v pamäti. Riešením je uchovanie primárneho kľúča záznamu v atribúte objektu.

6.3 Lazy load

Objekt neobsahuje všetky dáta, ktoré potrebujeme, ale vie, ako ich získať.

Keď nahrávame dáta z objektu do pamäte, tak je vhodné to navrhnúť tak, že sa naplnia atribúty objektu spolu s úzko súvisiacimi dátami. Vytvára to nahrávanie jednoduchším, programátor má vždy všetky dáta týkajúce sa objektu v pamäti, nepotrebuje explicitne pristupovať k databáze.

Avšak v tomto prípade nahrávame vždy veľké množstvo dát, ktoré ani nebudeme potrebovať, lebo nás zaújmajú dva či tri objekty. Má to dopad na výkon aplikácie.

Lazy load prerušuje proces nahrávania. Pamatá si, ktorý objekt už je v pamäti. Ak pristúpime k dátam, ktoré v pamäti ešte nie sú, automaticky ich nahrá z databáze, ale len v prípade, že ich skutočne potrebujeme.

6.3.1 Lazy initialization

Je to najjednoduchší spôsob. Základná myšlienka je, že vždy keď pristupujeme k polu, skontrolujeme či nemá hodnotu NULL. Ak áno, vypočíta hodnotu pred tým ako ju vráti. Aby to fungovalo, musíme sa uistiť, že je pole implementované ako atribút a prístup k nemu je len pomocou tohoto atribútu.

Hodnota NULL funguje ako dobrý signalizátor, či dáta sú resp. nie sú nahrané, dokým nie je NULL jedna z očakávaných hodnôt z databázy. V tom prípade potrebujeme nejaký explicitný signalizátor či dáta boli dotiahnuté.

6.3.2 Virtual proxy

Tento návrhový vzor je prevzatý z GoF. Je to objekt, ktorý vyzerá, že by mal byť v políčku, ale v skutočnosti nič neobsahuje. Nahrá sa v prípade, že je volaná jedna z jeho metód.

6.3.3 Value holder

Princípom je objekt, ktorý obaľuje iný objekt. Aby sme získali základný objekt, musíme požiadať Value holder pre jeho hodnotu, ale len pri prvom prístupe doťahuje dáta z databáze.

Nevýhodou je, že trieda potrebuje vedieť, že je prítomná a že stratíme explicitnosť silného typovania.

6.3.4 Ghost

Je skutočný objekt v neúplnom stave. Keď ho nahráme z databáze, obsahuje len svoje ID. Pri prvom pokuse o dotiahnutie hodnoty nejakého poľa sa celý nahrá z databáze.

6.4 Service layer

Definuje aplikačné hranice medzi vrstvami tak, že stanovuje množinu dostupných operácií a koordinuje odpovede aplikácie na každú operáciu.

Enterprise aplikácie obykle vyžadujú rôzne druhy rozhraní k dátam, ktoré ukladajú a k logike, ktorú implementujú. I napriek tomu, že sú tieto rozhrania značne odlišné, potrebujú spoločnú interakciu s aplikáciou na prístup a manipulovanie s dátami a volaniu jej obchodnej logiky. Interekcie možu byť komplexné, možu vyžadovať transakciu cez niekoľko zdrojov a koordináciu niekoľkých odpovedí na akciu. Ak píšeme kód oddelene v každom rozhraní, spôsobuje to veľa duplicity.

Service layer zapúzdruje business logiku aplikácie, kontroluje tranzakcie a koordinuje odpovede v implementáciách operácií.

Može byť implementovaná rôznymi spôsobmi, líšia sa v zopodvednosti, ktorú ukrývajú jednotlivé rozhrania.

6.4.1 Domain facade

Je implementovaná ako množina tenkých fasád cez doménový model. Triedy implementujúce fasády neimplementujú žiadnu biznis logiku, ale doménový model implementuje všetku biznis logiku. Tenké fasády stanovujú hranice a množiny operácií cez ktoré klientské vrstvy komunikujú s aplikáciou.

6.4.2 Operation script

Service layer je implementovaná ako množina tlstejších tried, ktoré priamo implementujú aplikačnú logiku, ale delegujú zapúzdreným doménovým objektom triedy pre doménovú logiku. Operácie dostupné klientom sú implementované ako skripty.

6.5 Transaction script

Organizuje biznis logiku do procedúr, kde sa každá procedúra zaoberá jednou úlohou z prezentačnej vrstvy.

Na väčšinu veľkých aplikácií sa môžeme pozerať ako na zoskupenie tranzakcií. Tranzakcia môže zobraziť nejaký pohľad, iná ho može zmeniť. Každá interakcia medzi klientom a serverom obsahuje isté množstvo logiky. Občas to môže byť len zobrazenie dát z databáze, inokedy môže vyžadovať rôzne validácie a výpočty.

Transcation script organizuje všetku logiku primárne v jednej procedúre, ktorá volá priamo databázu alebo tenkú databázovú obálku. Každá transakcia má svoj vlastný skript, avšak spoločné podúlohy môžu byť rozložené do podprocedúr.

6.6 Transform view

Pohľad, ktorý spracováva doménové dáta krok po kroku a transformuje ho do HTML

Dotazom do doménového modelu získame dáta z databáze, ktoré však musíme preformátovať do HTML ak ich chceme zobraziť ako webovú stránku. Transform view znamená transformácia dát, kde je vstupom dáta modelu a výstupom HTML.

7 Popis platformy .NET

Pod pojmom .NET môžeme rozumieť niekoľko vecí. Na jednej strane je takto označovaná iniciatíva, ktorej cieľom je vybudovať novú generáciu operačních systémov, rôznych typov aplikačných serverov patriacich do rodiny Microsoft a v neposlednej rade vývojovýh nástrojov. Všetky uvedené produkty sú alebo budú závislé na jadre platformy nazvonej .NET Framework. A práve .NET Framework predstavuje to, čo sa mnohým vybaví, keď vyslovíme termín plarforma .NET.

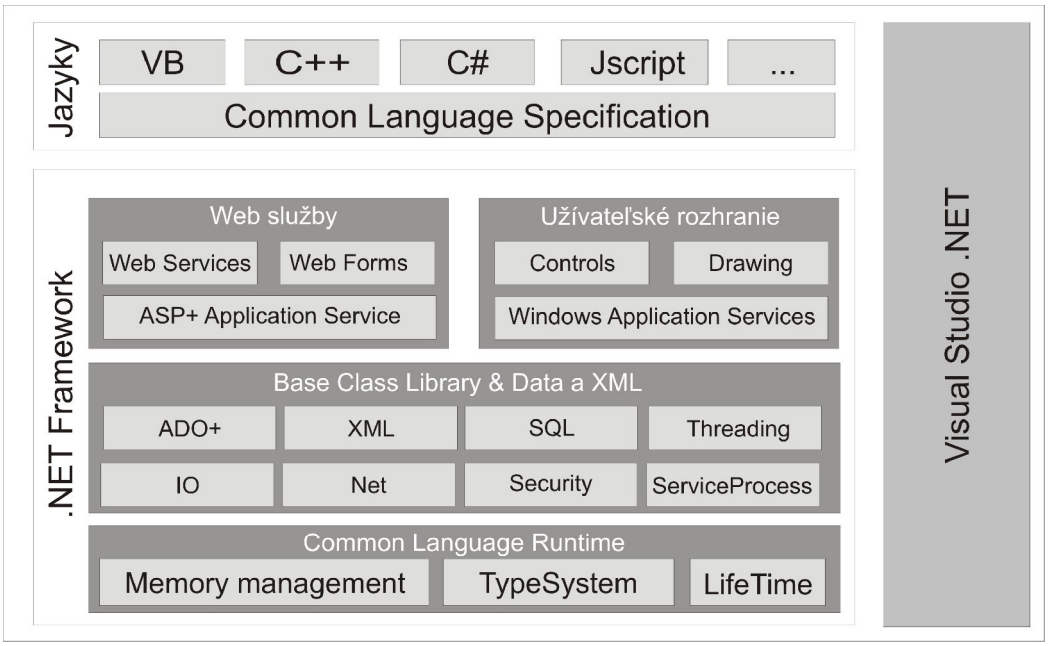
Platforma .NET prináša výrazne pozitívne zmeny do architektúry a spôsobu tvorby aplikácií. Sú nimi najmä:

- *Common Language Runtime*, ktorý sa stáva prostredím pre spustenie .NET aplikácií.
- Objektovo orientované knižnice dostupné vo všetkých programovacích jazykoch, výrazne zjednodušujú vývoj internetových aj desktopových aplikácií.
- *Common Language Specification*, sada jazykov vyvinutých firmou Microsoft a ďalšími stranami, podporujúcich platformu .NET.

Jej nasadenie je veľmi široké, počnúc aplikáciami na systémovej úrovni, cez jednoužívateľské desktopové aplikácie a končiac pri aplikáciách s úzkou väzbou na internet. Srdcom platformy .NET je jadro nazvané .NET Framework, ktoré plní niekoľko základných úloh:

- Je run-time prostredím pre beh aplikácií.
- Obsahuje množinu jazykovo nezávislých, objektovo orientovaných knižníc podporujúcich prácu s databázami, vstupom/výstupom, komunikačnými protokolmi, procesmi a vláknami, bezpečnosť atď.
- Dáva k dispozícií unifikovanú architektúru pre tvorbu aplikácií s užívateľským rozhraním.

7.1 .NET Framework



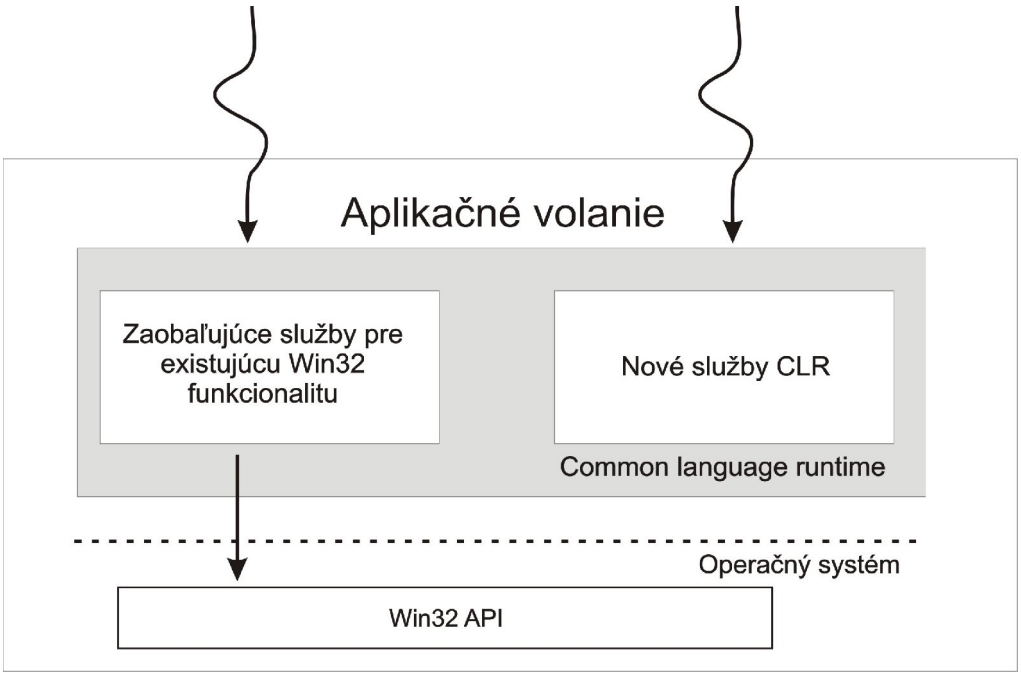
Obr. 18. Štruktúra .NET Framework a naväzujúce jazyky s vývojovým prostredím

Je to prostredie, ktoré umožňuje vytvárať programátorom oveľa jednoduchšie dobré, robustné aplikácie v krátkej časovej dobe a starať sa, nasadzovať a revizovať existujúci kód. Programy a komponenty, ktoré píšeme sa púšťa vo vnútri prostredia. To programátorovi uľahčuje prácu napr. zhladiska automatickej správy pamäte (zber odpadu) a jednoduchší prístup ku všetkým systémovým službám. Pridáva mnoho užitočných vlastností ako jednoduchý prístup na internet alebo k databázi. Zabezpečuje rovnako aj nový mechanizmus na znovupoužitie kódu, je jednoduchšie ho použiť a rovnako je silnejší a flexibilnejší ako COM. .NET framework je jednoduchší na nasadzovanie pretože nepotrebuje žiadne nastavenie registrov. Poskytuje aj štandardizované verzovanie na úrovni systému.

7.2 Common Language Runtime

Koncepcia *run-time* prostredia a knižníc nie je nová. Každý programátor sa určite stretol s istou formou tohto nástroja. V jazyku Visual Basic máme runtime knižnicu *VBRUNxxx.dll*, ktorá musí byť prítomná na každom počítači spúšťajúcom jeho aplikácie. Jazyky C a C++ majú tiež svoju *run-time* knižnicu funkcií. Dostupné sú aj objektovo orientované knižnice nazvané *MFC* a *ATL*. Výnimku netvorí ani jazyk Java vyžadujúci k svojej funkcionalite tzv. Virtuálny stroj. Obdobne si môžeme predstaviť funkcionalitu CLR. Dôvodov pre vytvorenie nového *run-time* prostredia bolo hneď niekoľko:

- výrazne zjednodušiť vývoj aplikácií
- pripraviť robustné a bezpečné prostredie pre beh aplikácií
- podporiť veľké množstvo programovacích jazykov
- zjednodušiť nasadenie a administráciu aplikácií



Obr. 19. Skupiny služieb CLR

Výsledkom sa stala sada *run-time* služieb, programovo dostupných vo forme tried a štruktúr implementujúcich rôzne verejné rozhrania. Množinu služieb môžeme rozdeliť do dvoch kategorií. Prvá zaobaluje existujúce *Win32* funkcie a služby a druhá ich doplňuje o služby vyžadované modernými vývojovými postupmi (viz obrázok 19).

.Net programy sú kompilované. Každý jazyk ktorý smeruje k .NET kompiluje zdrojové kódy do metadát a MSIL kódu. Metadata obsahujú úplnú špecifikáciu pre program vrátane všetkých typov, okrem implementácie každej funkcie. Implementácie sú uložené ako MSIL, čo je nezávislý kód, ktorý popisuje inštrukcie programu. CLR používa „blueprint“ aby priniesol .NET program k životu počas behu, poskytuje služby, ktoré tradičné kompilované programy neumožňujú.

Kľučové vlastnosti CLR sú:

- Interaktivita počas behu – programy môžu bohate komunikovať jeden z druhým počas behu použitím ich metadát. Program môže počas behu vyhľadať nový typ, vytvoriť inštanciu a vyvolať metódy na danom type.
- Mobilita – programy môžu bežat bez rekompilácie na akomkoľvek operačnom systéme a procesore, ktorý podporuje CLR. Kľúčovým elementom tejto platformovej

nezávislosti je JIT (Just-In_Time) kompilátor, ktorý kompiluje MSIL kód do natívneho kódu, ktorý beží na danej platforme.

- Bezpečnosť – CLR má schopnosť analyzovať MSIL inštrukcie ako bezpečné alebo nebezpečné.
- Zjednodušené nasadzovanie – Assembly je úplne samopopisujúci balík, ktorý obsahuje všetky metadáta a MSIL programu. Nasadzovanie môže byť jednoduché do takej miery, že stačí skopírovať dané assembly na klientský počítač.
- Verzovanie – Assembly môže fungovať správne s novými verziami knižníc, na ktorých závisí bez rekompilácie. Je to možné kvôli schopnosti rozlíšiť všetky typové referencie cez metadáta.
- Zjednodušený vývoj – CLR poskytuje mnoho výhod, ktoré vysoko zjednodušujú vývoj, včetne služieb ako zber odpadu, odchytávanie výnimiek, debugovanie.
- Integrácia cez viaceré jazyky – CTS definuje typy, ktoré sa môžu vyskytovať v metadatách a v MSIL a možné operácie, ktoré môžu byť vykonávané s týmito typmi. CTS je dostatočne rozsiahla aby podporovala mnoho rôznych jazykov včetne jazykov od Microsoftu ako C#, VB.NET a C++ a jazykov tretích strán ako napr. COBOL, Python, Smalltalk. CLS definuje podmnožinu CTS, ktorá predstavuje spoločný štandard, ktorý umožňuje .NET jazykom zdieľať a rozširovať vzájomné knižnice. Napr. je možné v C++ zdediť z triedy napísanej v C# a preťažiť jej metódu.
- Interoperabilita – CLR zabezpečuje interoperabilitu so širokou základňou existujúceho software napísaného v COM a C. .NET typy môžu byť exponované ako COM typy a COM typy môžu byť importované ako .NET typy. CLR obsahuje Pinvoke čo je mechanizmus, ktorý umožňuje aby C funkcie, štruktúry a spatné volania boli použité v rámci .NET.

7.3 MSIL

Pred tým ako začneme hovoriť o *Common Language Runtime*, musíme si vysvetliť termín *Intermediate Lagnuage* a riadený kód (*Managed code*). Pod termínom riadený kód si predstavme kód, kde sa o prevedenie stará CLR. Ak spustíme aplikáciu, ktorá nie je napísaná pre platformu .NET, alebo sa explicitne zriekne výhod a služieb ponúknutým CLR, potom taký kód nazývame Neriadený (*Unmanaged*).

Výstupom kompilátoru každého z jazykov schopného generovať riadený kód je *Microsoft Intermediate Language* (MSIL alebo len IL). MSIL je precesorovo nezávislý jazyk veľmi podobný assembleru. Oproti nemu je oveľa vyspelejši a v súčasnej dobe neexistuje žiadny procesor, ktorý by mu rozumel.

Dôvodom jeho zavedenia je snaha o jednoduché prenášanie existujúceho kódu medzi rôznymi hardwerovými platformami, na ktorých je rozšírený operačný systém Windows.
Keďže sa kód v jazyku MSIL nedá vykonávať na žiadnom procesore, musí byť pred spustením prevedený na skutočné inštrukcie konkrétneho čipu. Na tento účel slúžia tzv. *Just-In-Time* Kompilátory (JITTER), ktoré sa delia na tri druhy:

- preklad v dobe inštalácie.
- *Just-In-Time* prekladač
- ekonomický *Just-In-Time* kompilátor

7.4 ASP.NET

Nový progamovací model na vytváranie HTML stránok s názvom ASP.NET. V dnešnej dobe sú internetové aplikácie na vzostupe, väčšina z nich používa na zobrazenie prezenčnej vrstvy všeobecný prehliadač. To vyžaduje mať server, ktorý vytvára stránky použitím HTML jazyka, ktorému prehliadač rozumie a môže zobraziť požadovaný obsah používateľovi. ASP.NET (nová verzia ASP) je nové prostredie, ktoré beží na webovom serveri napr. IIS a uľahčuje programátorom vytvárať kód, ktorý zostavuje HTML webové stránky, ktoré sa potom zobrazujú užívateľom v priehliadači. ASP.NET má novú vlastnosť, ktorá umožňuje písať jazykovo nezávislý kód a zviazať ho s požiadavkami webových stránok. Sú to .NET Web Forms, udalosťami organizovaný progamovací model, kde sa prvky medzi sebou navzákom ovplyvňujú. Vytvára to dojem akokeby sme programovali formuláre pre okienkovú aplikáciu. ASP.NET je oveľa robustnejší, obsahuje mnoho pozitívnych zmien oprati pôvodnému ASP.

Prehľad vlastností:

- ASP.NET oddeľuje HTML výstup od programovej logiky použitím tzv. kódu na pozad. Namiesto toho aby sme miešali HTML a kód, kód ktorý píšeme sa nachádza v samostatnom súbore, ku ktorému ASP stránka obsahuje referenciu. Oddelením HTML tagov a aktívneho kódu sa stáva výsledná aplikácia čitelnejšia, udržovatelnejšia.
- ASP.NET podporuje webové formuláre, je to technológia, ktorá robí programovanie webových stránok veľmi podobné programovaniu formulárov desktopových aplikácií. Programovanie pozostáva z pridávania kontroliek na stránku a písania metód, ktoré obsluhujú udalosti.

7.5 C#

C# je nový jazyk navrhnutý na optimálnu zmes jednoduchosti, expresívnosti a výkonu. Väčšina vlastností jazyka bola navrhnutá ako odpoveď na silné a slabé stránky iných jazykov. Kľúčovými vlastnosťami jazyka C# sú

- Komponentová orientácia – Výborný spôsob ako sa vyrovnať so zložitosťou programu je rozdelenie na niekoľko komponent, ktoré spolu komunikujú, niektoré z nich môžu byť použité vo viacerých scenároch. C# bol navrhnutý tak, aby umožnil vytváranie kompenent jednoducho a zabezpečil komponentovo orientované konštrukcie ako vlastnosti, udalosti a deklaratívne koštrukcie nazvané atribúty.
- Kód na jednom mieste – všetko čo patrí do deklarácie v C# je umiestnené v samotnej deklarácií, namiesto toho aby bolo rozložené cez niekoľko zdrojových súborov alebo niekoľko miest v rámci jedného zdrojového súboru. Typy nepotrebujú dodatočnú deklaráciu v zvláštnych hlavičkových súboroch alebo v IDL súboroch, vlastnosti sú logicky zoskupené, dokumentácia je vložená priamo v deklarácií atď. Vzhľadom na to, že poradie deklarácií je irelevantné v objektovo orientovaných jazykoch, typy nepotrebujú extra deklaráciu aby mohli byť použité v rámci iných typov
- Verzovanie – C# podporuje vlastnosti ako explicitné implementácie rozhraní, skrývanie zdedených vlastností a modifikátor určených len na čítanie, čo pomáha novým verziám komponent spolupracovať so staršími komponentami, ktoré na nich závisia.
- Typová bezpečnosť a unifikovaný systém typov – C# je typovo bezpečný, čo zabezpečuje že premenná môže be prístupná jedine cez typ pričlenený s danou premennou. Toto zapúzdrenie zabezpečuje dobré programové návrhy a eliminuje možné chyby alebo bezpečnostné incidenty, pretože nie je možné úmyselne alebo neúmyselne prepísať hodnotu premennej. Všetky typy (včetne primitívnych typov) dedia z jedného bázového typu, čím vytvára unifikovaný typový systém. To znamená všetky typy, štruktúry, rozhrania, delegáti, polia zdieľajú rovnakú základnú funkcionalitu, ako napríklad možnosť byť prekonvertovaný na reťazec, serializovaný alebo byť uložený v kolekcii.
- Automatická a manuálna správa pamäti – C# sa spolieha na framework, ktorý sa automaticky stará o správu pamäti. Toto oslobodzuje programátora od uvoľnovania objektov, čo eliminuje problémy typu neprístupné ukazatele, pretečenie pamäti a kopírovanie s kruhovými referenciami. Avšak C# neeliminuje ukazatele. Je možné použiť kľúčové slovo unsafe a pracovať s ukazateľmi na nižšej úrovni. Odporúča sa to v prípadoch ak chceme vytvárať vysoko výkonné aplikácie. Sú nutné vysoké bezpečnostné práva na vykonávanie takýchto blokov.

- Vplyv CLR – veľkou výhodou C# oproti iným jazykom, najmä tradičným kompilovaným jazykom ako napr. C++, je jeho blízke spolunažívanie s .NET CLR. Mnohé z vlastností C# sú vlastne vlastnosti CLR ako typový systém, správa pamäti, odchytávanie výnimiek.

7.6 Attributes

Atribúty sú deklaratívne tagy, ktoré môžu byť použité na anotáciu typov alebo členov triedy. Pomocou nich je možné modifikovať význam alebo meniť správanie. Táto popisná informácia získaná od atribútu je uložená ako metadáta v .NET assembly a môže byť použitá buď počas návrhu alebo počas behu.

Ako položka metadát atribúty popisujú programový elemnt, na ktorý sa aplikujú a je k dispozícií počas návrhu, v čase kompilácie aj v čase behu.

Atribúty je vlastnosť .NETu, ktorá pri správnom použití zprehľadňuje kód a jeho znovupoužitelnosť.

8 Vzorová aplikácia

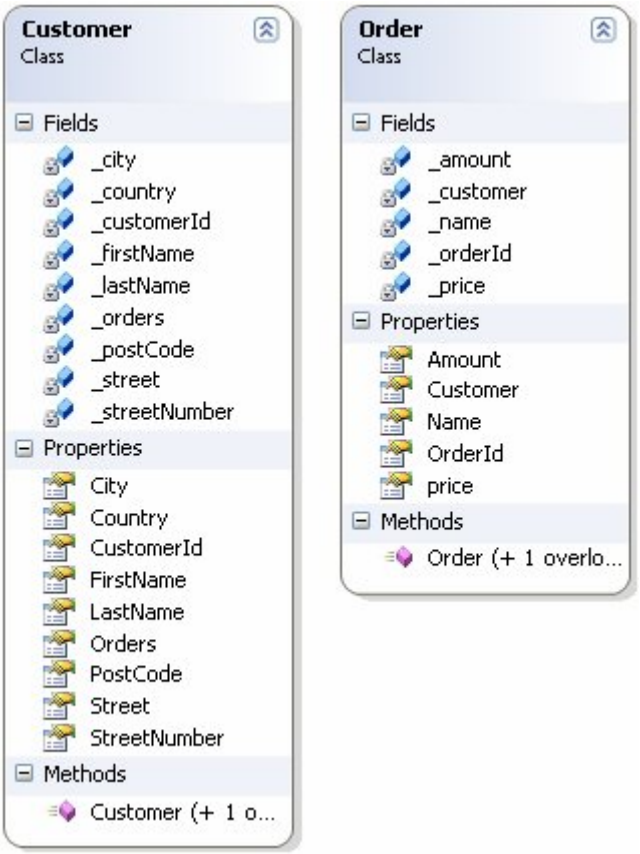
Na ukážku návrhových vzorov bola vytvorená webová aplikácia, kde sa na rôznych vrstvách aplikácie snažím o implementáciu návrhových vzorov. Ku každému vzoru sa snažím podať vysvetlenie, prečo považujem za vhodné použitie návrhového vzoru.

Aplikácia neobsahuje žiadnu zložitú biznis logiku. Aplikácia nemá za cieľ byť rozsiahlym projektom, skôr na príklade jednduchého modelu poukazujem na použitie návrhových vzorov. Podávam odvôvodnenie použitia, prípadne hovorím o výhodách/nevýhodách použitia.

V nasledovných kapitolách sa odvolávam na použitie dvoch základných entít a to zákaznik (*Customer*) a objednávka (*Order*).

Nemalo význam implementovať rozsiahlu aplikáciu s biznis logikou, nakoľko to nie je cieľom tejto práce. Podľa môjho názoru stačí na príklade dvoch entít ukázať zaobchádzanie na jednotlivých úrovniach, na ich základe je možné odvodiť použitie pre ľubovolné množstvo entít.

Na obrázku sú znázornené použité biznis objekty.



Obr. 20. Entity customer a order

8.1.1 Dátový zdroj

Na implementáciu perzistentného ložiska používam Microst Office Access 2007. Aplikácia je pripravená aj na použitie iných dátových zdrojov viz. vzory nižšie.

Pre prehľadnosť sú použité len dve tabule so vzťahom 1:M. customer a order. Tabula order obsahuje cudzí kľúč na na tabulu customer.

Order

Field Name	Data Type
OrderId	AutoNumber
CustomerId	Number
Name	Text
Amount	Number
Price	Currency

Customer

Field Name	Data Type
FirstName	Text
Street	Text
StreetNumber	Text
PostCode	Text
LastName	Text
City	Text
Country	Text
CustomerId	AutoNumber

Obr. 21. Reprezentácia tabuliek v aplikácií

Stĺpce zvolené v tabuľkách majú skôr informatívny charakter, len aby bolo vidieť ako data cestujú jednotlivými vrstvami.

8.1.2 Organizácia projektov

Pri vytvárani projektov v prostredí .NET je vhodné projekty organizovať podľa vrstiev. Aplikácie sa obvykle skladajú z troch základných vrstiev, ktorými sú dátová vrstva, biznis vrstva a prezentačná vrstva.

Biznis vrstva obsahuje dva projekty: *BusinessObjects* a *Facade*. BusinessObjects obsahuje jednotlivé biznis objekt, s ktorými aplikácia pracuje. Sú to v podstate dátove schránky. V prípade, že potrebujeme využiť ich funkcionalitu, nepristupujeme k nim priamo, ale využívame projekt *Facade*, ktorý obsahuje triedy na prístup k biznis objektom. V ďaľších kapitolách bude problematika bližšie popísaná.

Dátová vrstva obsahuje jeden projekt s názvom *DataObjects*, ktorý obsahuje triedy na prácu s biznis objektami, blízko databázi tzn. Prenášajú hodnoty ktoré nám vráti dátový zdroj.

Okrem týchto základných vrstiev mám v návrhu riešenia založený ešte jeden projekt, ktorý obsahuje projekty, ktoré svojou logikou vypomáha trom základným vrstvám. Má nazov *Framework*, obsahuje kód týkajúci sa napr. Logovania. Do tejto zložky ukladám projekty v dvoch prípadoch :

a) Ak musím vytvoriť niečo k čomu majú mať prístup všetky vrstvy
b) Ak je niečo príliš zložité resp. rieši nejakú zložitú implementáciu

9 Implementované návrhové vzory

V prvej časti sa budem venovať základným návrhovým vzorom zo skupiny Gang of Four (GoF). Tie sú základom pre Enterprise návrhové vzory, ktoré sú popísané v nasledovnej kapitole. Kapitola obsahuje popis použitých návrhových vzorov, ich implementáciu v ukážkovej aplikácií.

Popisujem len vzory, ktoré boli skutočne implementované a ich použitie považujem za zmysluplné pri budovaní viacvrstvových aplikácií. Snažil som sa o implementáciu takých situácií, ktoré sa budú opakovať pri budovaní rozsiahlych projektov.

9.1 Singleton

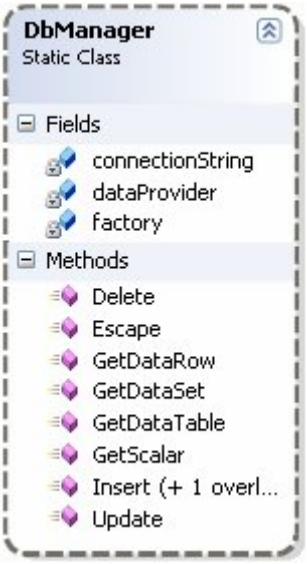
Singleton je typický príklad ako vyriešiť prístup k objektu v rámci celej aplikácie. Tzn. vytvoriť taký objekt, ktorého v rámci behu existuje práve jedna inštancia a všetky objekty v systému pristupujú práve k tomuto jednému objektu.

9.1.1.1 Objekt na prístup k databázi

Príkladom je prístup k objektu, ktorý sa stará o select/insert/update/delete operácie nad databázou. Je vhodné mať túto logiku implementovanú v jednom objekte a všetky DataObjects z dátovej vrstvy pristupujú k tomuto objektu.

Pri prístupe k databázi sa vytvárajú tzv. *connection pool*, ktoré sú nutné pri ukončení uzatvoriť, aby neostávali zbytočne otvorené a nikto ich nepoužíva. Výsledkom by totiž mohlo byť, že ostanú otvorené všetky a databáza nám odopre prístup. Preto je vhodné pristupovať k databázi z jedného miesta a vždy uzatvoriť spojenie. Ak sa to implementuje v jednej triede, ktorá sa následne dôkladne odladí a všetky objekty v systéme budú k databázi pristupovať cez toto rozhranie, tak minimalizujeme riziko otvorených databázových spojení.

V mojom konkrétnom príklade je objekt *DbManager* realizovaný ako statická trieda. Ak je trieda implementovaná ako statická, vždy sa vyskytuje v systéme iba raz, z toho dôvodu ju môžeme považovať za návrhový vzor singleton.



Obr. 22. Statická trieda DbManager

9.1.1.2 Objekt na logovanie

Každá enterprise aplikácia musí riešiť problematiku logovania. Opäť je to vhodný príklad na návrhový vzor Singleton. Potrebujeme objekt, ktorý zabezpečí zápis do databáze v prípade neočakaváneho stavu. Prístup k tomuto objektu vyžadujeme na všetkých vrstvách nakoľko chyby sa môžu vyskytnúť na každej úrovni.

Implentácia je podľa obr. 2. tzn. konštruktor triedy je privátny. Jediným spôsobom ako pristúpiť k objektu je pomocou verejnej vlastnosti Instance, ktorá pri prvom prístupe k objektu, objekt vytvorí. Od toho momentu je objekt statický a vždy keď k nemu pristúpime, tak pristupujeme k tej istej inštancií objektu.

```
namespace DP.Framework.Log
{
    public sealed class SingletonLogger
    {
        // prave jedna instancia logera
        private static readonly SingletonLogger instance
            = new SingletonLogger();

        // privatny konstruktor
        private SingletonLogger()...

        // vrati instanciu objektu
        public static SingletonLogger Instance...

        dalsi kod...
    }
}
```

Obr. 23. Implementácia singletonu ako logovacia trieda

9.2 Abstract Factory

Tento návrhový vzor sa často vyskytuje implementovaný v 3 vrstvovej architektúre na najnižších miestach tj. na dátovej vrstve konkrétne ako implementácia Data Access Object enterprise návrhového vzoru.

9.2.1 Data Access Object

Je ho možné nájsť aj pod skratkou DAO. Kód, ktorý závisí na špecifikách dátového zdroja zväzuje dokopy biznis a dátovú vrstvu. Je potom náročné vymeniť/modifikovať dátové zdroje aplikácie. Vzor odddeľuje klientské dátové rozhranie od jeho mechanizmu prístupu k dátam, adaptuje špecifické API aplikácie na generické klientské rozhranie.

DAO model umožňuje meniť mechanizmus prístupu k dátam nezávisle od kódu, ktorý data používa.V ukážkovej aplikácii je pre každý typ dátového zdroja založený vlastný adresár. Ten obsahuje triedu pre každý biznis objekt aplikácie. Tzn. ak má aplikácia desať biznis objektov, potom musí obsahovat desať DAO objektov v dátovej vrstve. Každý DAO objekt predstavuje akúsi bránu na prístup k dátam z dátového zdroja. Dátovým zdrojom nemusí byť nevyhnutne databáza. Môže ním byť akýkoľvek perzistentný dátový zdroj napr. XML súbor.

DAO trieda obsahuje metódy na prístup, manipuláciu s dátami na najnižšej úrovni. Výsledkom métody select je obvykle kolekcia biznis objektov. Teda ak máme v úmysle pracovať s biznis objektom Customer, tak biznis objekt volá metódu z dátovej vrstvy s názvom napr. GetCustomers() výsledkom je kolekcia objektov typu Customer.

```
namespace DP.DataLayer.DataObjects
{
    public interface ICustomerDao
    {
        // vrati kolekciu
        IList<Customer> GetCustomers();

        // vrati objekt zakaznik podla jeho id
        Customer GetCustomer(int customerId);

        // vlozenie zakaznika
        void InsertCustomer(Customer customer);

        // modifikacia dat zakaznika
        int UpdateCustomer(Customer customer);

        // zmazanie zakaznika
        int DeleteCustomer(Customer customer);
    }
}
```

Obr. 24. Ukážka DAO objektu

Návrhový vzor je implementovaný v statickej triede DataAccess. Má privátny statický atribút na čítanie s názvom factory, ktorý volá metódu GetFactories statickej triedy DaoFactories. Tá rozhoduje o výbere správnej factory na základe dátového poskytovateľa. Čiže keď v biznis vrstve chcem dáta o zákazníkovi, dostávajú sa tam dáta zo zdroja, ktorý je aktuálne zapnutý. Dôležité je, že biznis vrstva nevie aký je dátový zdroj (ani to nepotrebuje vedieť), rovnako dátová vrstva nevie ako bude použitá. Môže byť volaná napr. z webovej služby. Jednotlivé časti systému sú na sebe nezávislé, zvyšuje sa možnosť znovupoužitia existujúceho kódu napr. v inej aplikácií.

```
namespace DP.DataLayer.DataObjects
{
    public static class DataAccess
    {
        // datovy provider, rozhoduje o pouziti datoveho zdroja
        private static readonly string dataProvider
            = ConfigurationManager.AppSettings.Get("DataProvider");

        // na zaklade datoveho zdroja vytvorim tovaren
        private static readonly DaoFactory factory
            = DaoFactories.GetFactory(dataProvider);

        public static ICustomerDao CustomerDao
        {
            get { return factory.CustomerDao; }
        }

        public static IOrderDao OrderDao
        {
            get { return factory.OrderDao; }
        }
    }
}
```

Obr. 25. Použitie vzoru Factory

9.3 Composite

Composite návrhový vzor má za cieľ zakryť zložitú implementáciu a snaží sa, aby bol prístup k zložitému objektu rovnako intuitívny akokeby to bol jeden objekt. Návrhový vzor som použil pri implementácií Menu vzorovej aplikácie.

V hornej časti aplikácie sa vyskytujú položky menu. Logicky to je kus kódu, ktorý sa opakuje. Menu sa skladá z položiek, kde každá obsahuje vlastnosti Item, ktorý obsahuje text – položku menu, vlastnosť Link, ktorý obsahuje URL, na ktoré sa stránka presmeruje a vlastnosť Children, ktorá ukazuje na kolekciu objektov samého seba. Takýmto spôsobom je možné menu logicky usporiadať a rozhodovať, ktorá položka patrí k čomu. V ukážkovej aplikácií to nevyužívam nakoľko mám len

dve entity, s ktorými pracujem. V reálnej situácií by však bolo možné napr. farebne rozlíšiť prvky menu, podľa toho, k čomu daná položka logicky patrí.

Jadro implementácie je v použití rekúrzívneho volania metódy, ktorá renderuje požadované menu v HTML.

```
MenuCompositeItem root
  = new MenuCompositeItem("Main page",ResolveUrl("~/MyWeb/Default.aspx"));
MenuCompositeItem customers
  = new MenuCompositeItem("Customers",ResolveUrl("~/MyWeb/Customers.aspx"));

root.Children.Add(customers);

TheMenuComposite.MenuItems = root;
```

Obr. 26. Vytvorenie menu aplikácie pomocou vzoru composite

9.4 Facade

Fasáda ako názov napovedá znamená povrch budovy. Nevieme nič o jej vnútre, o zložitosti akú skrýva. Skrýva ju a zobrazuje navonok len to zaújmavé.

A takto v podstate návrhový vzor fasáda funguje. Skrýva zložité časti systému a zabezpečuje rozhranie klientovi odkiaľ k nemu klient môže pristupovať.

Fašada je typický predstaviteľ vzoru, ktorý ma použitie najmä v biznis vrstve. Biznis objekt Customer sa nachádza v strednej vrstve, ale keď k nemu chceme pristúpiť, tak nemáme tu možnosť prístupu priamo. Na jeho využitie musíme pristúpiť k objektu CustomerFacade, ktorý sprostredkováva jeho funkcionalitu okolnému svetu. Tzn. keby sme chceli v prezentačnej vrstve zobraziť tabuľku zákazníkov, tak zavoláme metódu GetCustomers() objektu CustomerFacade. Vráti nám kolekciu objektov typu Customer, ktorý následne môžeme použiť ako dátový zdroj pre serverové elementy webovej stránky.

Toto oddelenie logiky od dátovej schránky zprehľadňuje prácu s biznis vrstvou. Vždy je všetka logika umiestnená na jednom mieste. Môže sa stať, že nám objekt poskytuje príliš veľké množstvo logiky, v tom prípade nám nič nebráni vytvárať viac fasád pre daný biznis objekt. Ak dané fasády nejak rozumne pomenujeme, zprehľadníme prácu s biznis vrstvou. Biznis objekt sa tak stáva prehľadnejší, udržovateľnejší.

```
namespace DP.BusinessLayer.Facade
{
    public class CustomerFacade
    {
        private ICustomerDao customerDao = DataAccess.CustomerDao;
        private IOrderDao orderDao = DataAccess.OrderDao;

        public void AddCustomer(Customer customer)...

        public int DeleteCustomer(int customerId)...

        public IList<Customer> GetCustomers()...

        public Customer GetCustomer(int customerId)...

        public Order GetOrder(int orderId)...

        public void UpdateCustomer(Customer customer)...
    }
}
```

Obr. 27. Fasáda pre triedu Customer

9.5 Proxy

Proxy vzor používame keď potrebujeme reprezentovať komplexný objekt jednoducho. Ak je z hľadiska zdrojov nákladné vytvorenie objektu, môžeme jeho vytvorenie odložiť dokým ho nebudeme naozaj potrebovať. Dovtedy ho môže reprezentovať jednoduchý objekt. Tento jednoduchý objekt nazývame proxy objekt komplexného objektu.

Aplikácie často obsahujú biznis objekty, ktoré sú uzko späté s inými biznis objektami. Objektami, ktoré logicky náležia inému biznis objektu a bez neho nemajú význam. Preto sa takýto biznis objekt nevyskytuje samostatne, ale ako vlastnosť nejakého iného objektu. V ukážkovej aplikácií som zámerne zvolil takýto objekt, aby som demonštroval prácu s týmto typom objektov.

Biznis objekt Customer má jednu zo svojich vlastností kolekciu objektov typu Order. Teda ku každému zákazníkovi máme priradenú kolekciu objednávok, ktorá náleží danému zákazníkovi. Lenže táto informácia nemusí byť vždy potrebná, na druhú stranu logicky patrí do triedy Customer. Preto je implementovaná ako Proxy objekt kolekcie Orders. Až v momente keď skutočne budeme chcieť prístupiť k tejto informácií bude kolekcia dotiahnutá z dátového zdroja.

Proxy objekt znamená, že navonok sa javí, že je objekt inicializovaný, ale v skutočnosti sa načíta do pamäte až v momente keď k nemu skutočne pristúpime.

```
public Customer GetCustomer(int customerId)
{
    Customer customer = customerDao.GetCustomer(customerId);
    customer.Orders = new ProxyForOrders<Order>(customer);

    return customer;
}
```

Obr. 28. Metóda zobrazuje použitie Proxy triedy

10 Enterprise návrhové vzory

10.1 Domain Model

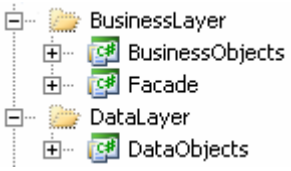
Na doménový model sa môžeme pozerať ako na konceptuálny model systému, ktorý popisuje rôzne entity týkajúce sa systému a jeho vzťahov. Doménový model je vytvorený aby dokumentoval kľúčové koncepty a slovník systému. Model identifikuje vzťahy medzi všetkými hlavnými entitami v systéme a obvykle identifikuje jeho dôležité metódy a atribúty. To znamená, že model poskytuje štrukturálny pohľad na systém, ktorý je za normálnych okolností doplnený dynamickými pohľadmi v modele prípadov použitia. Dôležité plus doménového modelu je popis a obmedzenia systému.

Doménový model môže byť efektívne použitý na verifikáciu a validáciu problému medzi rôznymi skupinami ľudí. Pomáha ako komunikačný nástroj a zameriava sa na technické týmy a biznis týmy.

V našom konkrétnom prípade je doménový model implementovaný nasledovne. Máme vyhradenú vrstvu, v ktorej sa nachádzajú všetky biznis objekty s názvom BusinessLayer. Obsahuje dva základné projekty:

- BusinessObjects – obsahuje definíciu biznis objektu z hľadiska dát. Môžeme sa naň pozerať ako na dátovú schránku, ktorá obsahuje len dáta. Je to výhodné mať oddelenú logiku od dát z viacerých dôvodov. Keď máme len dáta, tak môžeme kdekoľvek v systéme používať daný biznis objekt ako dátový zdroj, prenášame len dáta. Všetka logika týkajúca sa objektu je uložená na inom mieste, v mojom prípade je to projekt v BusinessLayer s názvom Facade.
- Facade – udržuje všetku funkcionalitu biznis objektov. Pre každý biznis objekt existuje objekt v tomto projekte. Názov sa skladá z názvu biznis objektu a sufixu Facade. Programátor tak nájde ľahko všetku logiku, ktorú daný objekt poskytuje. Je organizovaná na jednom mieste. Rovnako to je jediný spôsob ako dotazovať dátovú vrstvu, žiadny iný spôsob neexistuje – bezpečnosť.

Takže pre náš objekt Customer sídliaci v projekte BusinessObjects, existuje objekt CustomerFacade sídliaci v projekte Facade. Poskytuje funkcionalitu, ktorú programátor využije z prezenčnej vrstvy resp. z iných vrstiev nad biznis vrstvami napr. ak by sme chceli zverejniť funkcionalitu širšej verejnosti, núka sa možnosť použiť webové služby, ktoré by prirodzene volali funkcionalitu našej biznis vrstvy resp. fasádu daného objektu.



Obr. 29. Vrstvy v aplikácií

10.2 Identity field

Pri programovaní sa stretávame s problémom ako uchovať identitu záznamu z dátovej vrstvy na úrovni biznis vrstvy. Riešenie je jednoduché a to tým spôsobom, že každý objekt biznis vrstvy má atribút ID, ktorý odpovedá primárnemu kľúču daného záznamu v databázi.

Teda povedzme, že máme dátový zdroj Microsoft SQL Server 2005. Umožňuje použiť ako primárny kľúč hodnotu GUID, čo je náhodne vygenerovaný celosvetovo unikátny identifikátor umiestnený na 128 bitoch. Nevýhodou tejto hodnoty je, že je značne nečitelná a v ľudskej komunikácií ťažko použitelná. Napr. databázový server Oracle používa sekvencie, kde je primárny kľúč číselná hodnota.

Pri akejkoľvek manipulácií so záznamami identifikujeme objekt na základe primárneho kľúča. Biznis objekt má atribút ID, ktorý uchováva túto hodnotu. Ak chceme napr. modifikovať nejaký objekt, modifikujeme ho samozrejme v prvom kroku v prezentačnej vrstve. Zavolaním modifikujúcej metódy z príslušnej fasády daného objektu modifikujeme jeho dáta. Na úrovni databázi sa vykoná Update, kde bude záznam identifikovaný na základe primárneho kľúča, čím zabezpečíme, aby sme vždy referencovali správny objekt/záznam.

Pri zobrazovaní hodnôt v prezentačnej vrstve používame biznis objekty. Snažíme sa zachovať identitu objektov, pri napĺňaní polí používame celé objekty a komponentám priradzujeme atribút objektu, ktorý sa má v danej komponentu zobraziť. Hlavné však je, že pri modifikácií hodnoty sa vždy vieme vrátiť k pôvodnému objektu, vieme ktorý to je záznam, poznáme jeho ID. Ľahko teda môžeme implementovať modifikáciu, mazanie týchto záznamov.

Na implementáciu tohto vzoru sa v objektovo orientovaných jazykoch ponúka využitie dedičnosti, kde každý biznis objekt dedí z abstraktného predka, ktorý má atribút ID.

Ja som tento návrhový vzor implementoval ako atribút, ktorého názov pozostáva s názvu objekt a sufixu ID viz obr. 20.

10.3 Lazy load

V rozsiahlych aplikáciach sa často stretneme s prípadmi, keď objekt alebo kolekcia objektov logicky patrí nejakému inému objektu. Napr. objednávky budú vždy patriť zákazníkovi. Vždy sa k nim bude pristupovať cez ID zákazníka, nikdy nebude existovať samostatne. Z toho dôvodu musí byť objednávka na úrovni biznis vrstvy súčasťou iného objektu, konkrétne objektu zákazníka.

Ak chceme v prezentačnej vrstve zobraziť zoznam zákazníkov, je nutné dotiahnuť z biznis vrstvy kolekciu zákazníkov. Nakoľko objednávky sú tiež súčasťou objektu zákazník, v prezentačnej

vrstve máme možnosť v objekte zákazník pristúpiť ku kolekcií objednávky. Tie však v tomto konkrétnom prípade nepotrebujeme, ale keďže logicky patria do zákazníka, tak máme možnosť prístupu.

Riešením je, že budú objednávky v rámci zákazníka implementované ako Proxy objekt. V rámci inicializácie bude hodnota kolekcie objednávky nastavená na hodnotu NULL. V našom prípade skutočne objednávky nepotrebujeme, chceme zobrazovať len informácie týkajúce sa zákazníka. Môžeme zobraziť webovú stránku s detailami zákazníka bez toho aby sme dotiahli informáciu o objednávkach. V tomto prípade je nám táto informácia zbytočná.

Keby sme chceli zobraziť detail o objednávkach nejakého klienta, vytvoríme objekt typu Customer. V tejto chvíli je hodnota kolekcie Orders nastavená na hodnotu NULL a až pri prístupe k tejto hodnote je kolekcia inicializovaná. Na obrázku je vidieť implementáciu. Ak chceme vložiť záznam do kolekcie, kontrolujeme či je kolekcia skutočne inicializovaná. Nasledovný obrázok ukazuje inicializáciu daného zoznamu.

```
private void LazyLoad()
{
    if (!_isLoaded)
        LoadList();
}

public void Insert(int index, T item)
{
    LazyLoad();
    _list.Insert(index, item);
}
```

Obr. 30. Vloženie záznamu do kolekcie

```
protected override int LoadList()
{
    base.List = (IList<Order>)orderDao.GetOrders(_customer);
    return Count;
}
```

Obr. 31. Inicializácia zoznamu v zdedenej triede

Toto chovanie šetrí miesto v pamäti ako aj množstvo prenesených dát z dátovej vrstvy. Má to pozitívny vplyv na výkon systému. Je to však nutné používať s rozvahou. Ak sme si istí, že hodnotu kolekcie budeme s veľkou pravdepodobnosťou potrebovať, je lepšie ju inicializovať už na začiatku, pretože nepristupujeme zbytočne dva krát k databázi.

10.4 Domain facade

Doménová fasáda znamená rozdelenie aplikácie na tenké vrstvy, ktoré medzi sebou komunikujú a zachovávajú isté poradie. Doménovou fasádou rozumieme vrstvu v biznis vrstve, ktorá obaľuje biznis objekty a ku každému biznis objektu vytvára objekt vo fasáde. Dáta sú uložené v biznis objekte, jeho logika sa vyskytuje vo fasáde, ktorá je jediným miestom ako pristupovať k funkcionalite, ktorú daný objekt poskytuje.

10.5 Transaction script

Podstatou tohto modelu je sprístupniť logiku biznis objektu volaním jedenj procedúry alebo funkcie. Takto presne fungujú fasády, pomocou ktorých je stredná vrstva implementovaná. V prezentačnej vrstve je prakticky možné zavolať logiku biznis objektu volaním jednej funkcie. Robí to kód čitelnejší, prehľadnejší.

Volanie takto organizovanej biznis logiky je pre programátora intuitívnejšie. Vie, že na nič potrebné pri volaní nezabudol, pretože na začatie celého procesu stačí zavolať jednu funkciu. V podstate sa na volanie takejto procedúry môžeme pozerať ako na ucelenú transakciu.

Na obrázku vidieť priradenie kolekcie zákaznikov do serverovej komponenty, ktorá danú kolekciu zobrazí vo formáte tabuľky.

```
private void Bind()
{
    CustomerFacade facade = new CustomerFacade();
    GridViewCustomers.DataSource = facade.GetCustomers();
    GridViewCustomers.DataBind();
}
```

Obr. 32. Ukážka tranzakčného skriptu

10.6 Transform view

Keď vyvýjame webovú aplikáciu, dáta sú uložené v dátovom zdroji, prejdú dátovou vrstvou, ale na konci sa musia transformovať na formát, ktorému užívateľ bude rozumieť. Myslíme najmä na prehliadač a webové stránky.

Je nutné teda vytvárať stránky pozostávajúce z HTML tagov, pričom do nich chceme zakomponovať naše biznis dáta. Na tento účel používam transform view.

Nakoľko dáta vo vzorovej aplikácií majú obvykle formu tabuľky, stači jednoducho použiť ASP.NET serverovú komponentu GridView, ktorá dostane dátový zdroj a interne HTML vyrenderuje. Z toho dôvodu som použil tento vzor na generovanie menu.

Návrhový vzor úzko súvisí s návrhovým vzorom Composite, ktorý je popísaný v deviatej kapitole. Za jeho pomoci vytvoríme dátovú štruktúru, ktorá je následne použitá ako vstup pre teno návrhový vzor. Z danej stromovej štruktúry vygenerujeme menu aplikácie.

Vytvoril som si vlastnú serverovú komponentu, ktorá má ako vstup položky menu. Názov položky, ktorá sa zobrazí a url, na ktorú je stránka presmerovaná pri kliku na danú položku. Takýmto spôsobom je možné pridávať položky do menu. Každá položka obsahuje vlastnosť Children, ktorá ukazuje na ďalšiu položku, takže vieme definovať vzťah jednotlivých položiek. To je možné využiť pri renderovaní menu, ak chceme rozlíšiť jednotlivé položky.

Vzhľadom na malý počet biznis objektov, je menu v aplikácií organizované horizontálne.

```
protected override void RenderContents(HtmlTextWriter output)
{
    MenuCompositeItem root = this.MenuItems;

    output.Write(@"<table class=""menu"">");
    output.Write(@"<tr>");

    RecursiveRender(output,root);

    output.Write(@"</tr>");
    output.Write(@"</table>");
}

private void RecursiveRender(HtmlTextWriter output, MenuCompositeItem item)
{
    output.Write(@"<td><span onmouseover=""this.style.backgroundColor='black';
                    this.style.color='white'"" onmouseout=""
                    this.style.backgroundColor='white';
                    this.style.color='black'; ""
                    onclick=""location.href='" + item.Link + @"'"">
                    " + item.Item + @"</span></td>");

    for(int i = 0; i < item.Children.Count; i++)
    {
        RecursiveRender(output,item.Children[i]);
    }
}
```

Obr. 33. Implementácia renderovania menu

11 Poznatky z implementácie

Diplomová práca pojednáva o návrhových vzoroch, ich cieľom by malo byť zprehľadnenie a čiteľnosť kódu. Z toho dôvodu si dovolím spomenúť niektoré momenty, ktoré síce nepatria medzi návrhové vzory, ale podľa môjho názoru prispievajú k prehľadnosti aplikácie.

11.1 Atribúty

V ukážkovej aplikácií som síce atribúty nepoužil, i napriek tomu sa domnievam, že použitie metadát môže prispieť k čitelnosti kódu, k jeho údržbe, znovupoužitelnosti. V trojvrstvovej architektúre vidím priestor na realizáciu pri definícií CRUD operácií.

Ku každému biznis objektu vytvárame fasádu, ktorá udržuje logiku a prístup k metódam daného objektu. Veľká časť z nich sa týka operácií typu načítaj z dátového zdroja, ulož a pod. Z toho dôvodu vidím priestor na označenie týchto metód pomocou atribútu. Atribút sa štandardne vyskytuje v .NET platforme. Metóda by potom mohla vyzerať nasledovne.

Takto označené metódy je potom možné rôzne využiť či už v čase návrhu, kompiláciu alebo behu.

```
[DataObjectMethod(DataObjectMethodType.Select)]
public IList<Customer> GetCustomers()
{
    return customerDao.GetCustomers();
}
```

Obr. 34. Teoretické použitie atribútu

11.2 Master pages

Profesionálna webový portál musí mať štandardizovaný vzhľad na všetkých stránkach. Navigačné menu, hlavičku, pätičku a obsah v strede. Je nepraktické tieto statické informácie dávať do každej stránky. Pri zmene musíme prechádzať každú z nich, údržba sa stáva nákladnou. V ASP.NET 2.0 bola uvoľnená koncepcia Master page. Spoločné črty stránky naprogramujeme len raz, máster stránka slúži ako vzor pre ostatné stránky. Tie obsahujú už len unikátny obsah, ktorý sa vloži do šablóny, ktorú zdieľajú všetky stránky. Tým pádom majú všetky rovnaký vzhľad a spoločný kód sa vyskytuje len na jednom mieste.

V ukážkovej aplikácií som šablónu použil, stránky týkajúce sa biznis objektov obsahujú skutočne len unikátny obsah reprezentujúci daný biznis objekt.

11.3 Kľúčové slovo using

Programovací jazyk C# obsahuje kľúčové slovo *using* (nemýliť si s direktívou using).

Kľúčové slovo umožňuje programátorovi špecifikovať, kedy by mali objekty uvoľniť zdroje, ktoré používajú. Objekt musí implementovať rozhranie *IDisposable*. Obsahuje metódu *Dispose* , ktorá sa stará o uvoľnenie zdrojov.

Za normálnych okolností sa o uvoľňovanie pamati stará CLR interne bez vedomosti programátora. Týmto spôsobom si môžeme explicitne výnutiť, aby boli zdroje po skončení prác s objektom uvoľnené.

S obľubou to využívam pri prístupe do databázi. Zaisťujem si, že všetky zdroje budú uvoľnené, tým pádom neostávajú otvorené konexie k databázi.

12 Záver

Práca pojednáva o použití návrhových vzorov v prostredí .NET. V teoretickej časti popisujem jednotlivé návrhové vzory. Pri ich štúdiu som prenikol do ich podstaty. Zoznámil som sa s tvorbou 3-vrstvových aplikácií, skúmal som jednotlivé vrstvy a hľadal miesta vhodné na použitie návrhových vzorov. Získal som teoretické základy enterprise vzorov, ktoré sa používajú pri tvorbe rozsiahlych aplikácií.

Poukazujem na správne praktiky tvorby veľkých aplikácií vzhľadom na čiteľnosť, flexibilitu a znovupoužiteľnosť návrhu.

Cieľom teoretickej časti bol popis jednotlivých návrhových vzorov s prihliadnutím na prostredie .NET či už jednoduchých zo skupiny GoF alebo tzv. enterprise vzorov.

Po naštudovaní teoretickej časti bola naprogramovaná ukážková aplikácia, v ktorej získané dovednosti implementujem. Na vzore 3-vrstvovej webovej aplikácie poukazujem na implementáciu návrhových vzorov, či už základných GoF alebo enterprise návrhových vzorov. Snažím sa o ich efektívne využitie, aby bol výsledný kód aplikácie čitelnejší, prehľadnejší, robustnejší.

Pri programovaní som nadobudol skúsenosti s využívaním návrhových vzorov, viac sa zamýšľam nad implementáciou, obohatilo sa moje analytické myslenie, ktoré môžem zúročiť pri návrhu rozsiahlejších systémov.

Do budúcna by bolo vhodné takto navrhnutú aplikáciu otestovať v heterogénnejšom prostredí, napríklad vytvoriť viac prezentačných vrstiev, webové služby, desktopová aplikácia a doladiť systém tak, aby bolo vždy možné volať tú istú biznis logiku. Skúsiť meniť dátové zdroje, zmeny by sa mali týkať len dátovej vrstvy. Teda zamerať sa na flexibilitu systému, rozvýjať systém smerom, ktorý bol stanovený a to aby bol systém flexibilný, prehľadný, znovupoužiteľný.

Literatúra

[1] Kačmář, D., *Programujeme .NET aplikace.* 1.vyd. Praha, Computer Press 2001.

[2] Joshi, B., *Profesional ADO.NET.* 1.vyd. Chicago, Wrox Press Limited 2003.

[3] Freeman, E., Sierra, K., Bates, B., *Head First Design Patterns*. O'Reilly, 2004.

[4] Kerievsky, J., *Refactoring To Patterns*. Addison-Wesley, 2004.

[5] Fowler, M., *Patterns of enterprise application architecture*. Addison-Wesley, 2005.

Zoznam príloh

Príloha 1. CD obsahujúce diplomovú prácu v elektronickej podobe včetne zdrojových textov vzorovej aplikácie

/src - zdrojové texty vzorovej aplikácie

/text - práca v elektronickej podobe