VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

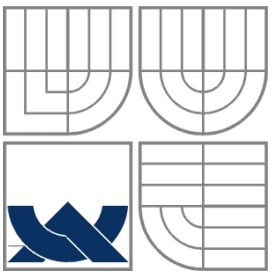
VYUŽITÍ NÁVRHOVÝCH VZORŮ PŘI TVORBĚ SOFTWARU

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

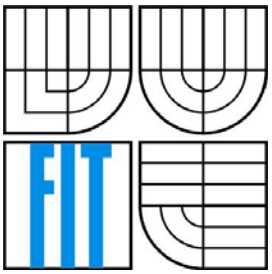
AUTOR PRÁCE
AUTHOR

BC. JIŘÍ VOLF

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 PŘI TVORBĚ SOFTWARU

USAGE OF DESIGN PATTERNS IN SOFTWARE DEVELOPMENT

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

BC. JIŘÍ VOLF

VEDOUCÍ PRÁCE
SUPERVISOR

RNDR. JITKA KRESLÍKOVÁ, CSC.

BRNO 2008

Abstrakt

Cílem mé diplomové práce je zmapování důvodů použití návrhových vzorů při vývoji moderních informačních systémů a dalších aplikací. Zabývá se základními aspekty vývoje objektově-orientovaných systémů, vysvětluje význam a strukturu návrhových vzorů. Dále uvádí rozřazení návrhových vzorů do kategorií a podrobněji představuje nejpoužívanější návrhové vzory včetně přínosů a případných komplikací, které při jejich použití mohou nastat. Na konkrétním příkladu databázové aplikace je prezentováno využití výhod návrhových vzorů.

Klíčová slova

Návrhové vzory, Objektově orientované programování, Vývoj aplikací, Informační systém, SQL, .NET, LINQ

Abstract

The objective of my diploma thesis was to map over reasons, why are design patterns so popular in the modern information systems and other applications development. It is considering with the basic aspects of object-oriented systems, expounds the importance and structure of design patterns. The explication is followed by categorization of design patterns and in more detail is focused on the most frequently used design patterns including their contribution or complications, which could eventually occur. The advantages of design with design patterns are demonstrated on practical example of database application.

Keywords

Design patterns, Object-oriented programming, Application development, Information system, SQL, .NET, LINQ

Citace

Volf Jiří: *Využití návrhových vzorů při tvorbě softwaru,* [diplomová práce]. Brno , 2008. FIT VUT v Brně.

Využití návrhových vzorů při tvorbě softwaru

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením
RNDr. Jitky Kreslíkové, CSc.
Další informace mi poskytl zejména konzultant Ing. Stanislav Mikulecký, z firmy Unicorn a.s.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Jiří Volf
30. 7. 2008

Poděkování

Děkuji panu Stanislavu Mikuleckému za poskytnuté konzultace.

Obsah

1 Úvod

Informační technologie představují jeden z nejrychleji rozvíjejících se oborů současnosti. Požadované systémy jsou stále komplexnější a tak nároky na jejich řízení a vývoj rychle narůstají. Architektura celého systému musí být navržena zcela jednoznačně, aby nedocházelo k nedorozumění jak v komunikaci s uživatelem tak v rámci samotného vývojového týmu. Trendem posledních let je použití objektově orientovaného programování a s ním se dostává do popředí i problematika návrhových vzorů.

Jedním z důvodů, proč obliba návrhových vzorů roste, je stále stoupající konkurence na trhu IT a tlak na co největší efektivitu programování. Snaha o dosažení maximální znovupoužitelnosti jednotlivých částí kódů i celých programových celků vzrůstá. Vhodné použití návrhových vzorů může redukcí nutně vynakládané práce výrazně snížit náklady na vývoj informačního systému ve firmě a celkově pozvednout kvalitativní úroveň návrhu.

V následujících odstavcích nejprve stručně nastíním historii a souvislosti návrhových vzorů, následovanou popisem principů objektově orientovaných systémů. Třetí kapitola obsahuje důležité společné vlastnosti spolu s terminologií, jež se jako červená linie promítají ve všech návrhových vzorech.

V čtvrté části nejprve uvedu syntaxi používanou k popisu/zápisu návrhových vzorů a jejich rozdělení do kategorií. Podkapitoly pak obsahují analýzu a popisy vybraných návrhových vzorů dle uvedené syntaxe i s možnostmi jejich využití a u každého vzoru je uveden jeho přínos designu systému.

Pátou kapitolou stručně shrnu společné vlastnosti a doporučení při použití návrhových vzorů.

Číslo šest nese část teoreticky rozebírající možné metriky pro hodnocení kvality software.

Další statě přinášejí popis praktického využití návrhových vzorů demonstrovaný na příkladu informačního systému finanční instituce, důsledky použití a výsledky softwarových metrik.

Práce navazuje na můj semestrální projekt, s nímž má společná témata prvních čtyř kapitol. Pro diplomovou práci jsem ovšem jejich obsah výrazně revidoval. Zadání práce pochází od firmy Unicorn a.s.

2 Objektově orientované programování a historie návrhových vzorů

Svět okolo nás se skládá z objektů. Při návrhu a tvorbě systémů programováním se snažíme vytvořit model těchto objektů tak, aby co nejvěrněji napodoboval skutečný stav, ale aby na druhé straně nebyl enormně složitý. Před příchodem objektově orientovaného programování probíhal vývoj převedením objektů z problémové domény reálného světa na „virtuální“ jednotky daného vývojového prostředí. Se stoupající komplexností vyvíjených systémů již přestal tento způsob postačovat a vedl často k problémům s interpretací návrhu systému.

Na rozdíl od jazyků modulárních tedy nepřistupuje objektově orientované programování ve své podstatě k dekompozici shora-dolů, ale zdola nahoru – tedy z jednodušších prvků skládá komplexnější funkce a části systému. Už to je příčinou vyšší znuvupoužitelnosti prvků objektově orientované programování, jelikož určité objekty obsahuje většina systémů.

2.1 Objektově orientované programování

Objektově orientované programování (dále OOP) se snaží nahlížet na objekty tak, jak je známe z běžného života. Prvky modelované reality jsou seskupeny do tzv. objektů.

2.1.1 Objekty

Objekt bychom měli chápat jako určitou součást modelované reality. Netýká se tedy pouze objektového programování, ale také objektově orientované analýzy a objektově orientovaného návrhu.

Definice uvedená v kapitole Teorie objektů v [3] říká: „Základním pojmem je objekt. Objekt si „pamatuje“ svůj stav (v podobě dat čili atributů) a poskytuje rozhraní operací, aby s ním bylo možné pracovat.“ Vlastnosti objektů definujeme jako atributy ve třídách.

Vztah objektu k třídě je takový, že objekt nazýváme instancí třídy. V rámci jedné třídy jsou všechny objekty stejné a liší se pouze obsahem svých atributů. Nicméně samotná teorie OOP nevyžaduje existenci třídy a umožňuje definovat samostatný objekt, pokud zadáme i jeho vlastnosti. Třída je tedy šablonou objektů – abstraktním vyjádřením jejich struktury. Při vytváření objektu (resp. instance třídy) dochází k zavolání konstruktoru – speciální funkce (metoda, viz dále), jež přidělí úložný prostor pro vnitřní data objektu a s těmito daty asociuje operace.

Abychom mohli s objekty provádět nějaké operace, zavádí OOP pojem metoda.

Operace, metody, zapouzdření

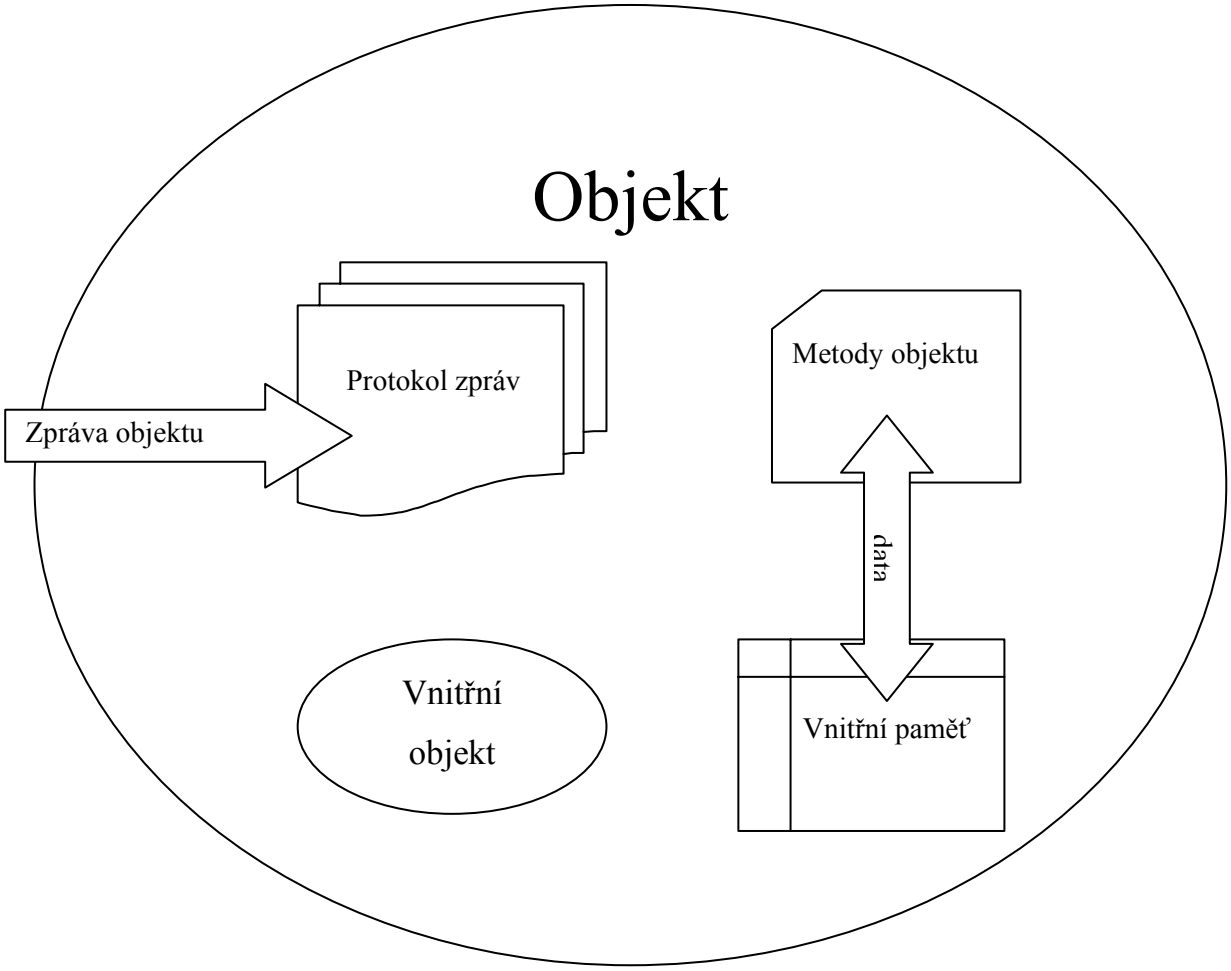
V OOP není vhodné s atributy objektů manipulovat přímo. Zpočátku to může vypadat jako komplikace, ale tento přístup má mnoho výhod. K práci s objekty a jejich atributy slouží pouze *operace*. Článek [8] o operaci uvádí: „Operace specifikuje transformaci stavu cílového objektu (a potenciálně stavu zbytku systému dosažitelného z cílového objektu) nebo dotaz (query), který vrací hodnotu volajícímu objektu.“ a dále pak „Operace specifikuje pouze výsledek chování, nikoliv vlastní chování.“ Způsob provedení operace je dán metodou. Je to typ funkce definované ve třídě a představuje jeden z nástrojů tzv. principu zapouzdření.

Zapouzdření je jedním ze základních kamenů OOP a zajišťuje, že objekt nemůže samovolně přistupovat k datům jiného objektu, ale musí k tomu využit rozhraní objektu představované metodami.

V praxi je možné zapouzdření obejít tím, že atributům nastavíme modifikátor na hodnotu *public* (viz dále bod 2.1.2. – oddíl dědičnost). Takový postup se však nedoporučuje, protože se ztrácí celistvost informace objektu – důležitá to vlastnost OOP. Celistvost zaručuje, že při přístupu k samotnému objektu máme k dispozici veškerá data s objektem související. Pokud bychom povolili přístup k objektu z vnějšku prostřednictvím modifikátoru public, procesy, které jej takto ovlivňují, nebudeme mít k dispozici jednoduše tím, že zavedeme objekt jako instanci patřičné třídy.

Protokol zpráv

Objekt je schopen přijmout a zpracovat informaci z vnějšku pomocí tzv. protokolu zpráv (viz obr. 2.1). Je to nástroj pro převod mezi příchozími zprávami a metodami objektu. Pokud objekt přijme nějakou zprávu, musí v protokolu zpráv vyhledat odpovídající metodu, jež danou zprávu zpracuje a spustit ji. Po ukončení vykonávání kódu metodou jsou zprávě zpátky poslány výstupní parametry volané metody.



Obr. 2.1 Vnitřní struktura objektu

2.1.2 Další principy objektově orientovaného programování

Dědičnost

Mezi odvozené vlastnosti (zdůvodnění viz [5], strana 7) OOP patři princip dědičnosti umožňující vznik tříd objektů připomínající hierarchickou uspořádanou strukturu. Objekty mohou dědit vlastnosti svých nadřazených tříd a navíc k nim přidávat své vlastní atributy nebo metody. Opakující se/společné vlastnosti tak nemusí být definovány mnohokrát, stačí je nadefinovat pouze jednou v nadřazené třídě a ostatní třídy je zdědí.

Vzniká při tom vztah předek – následník resp. generalizace-specializace, kde předek zastupuje původní obecnější třídu. Z ní je vytvořena třída speciální, zastoupená následníkem. Pokud tento postup opakujeme vícekrát, z třídy A zděděním vlastností odvodíme specializovanou třídu B

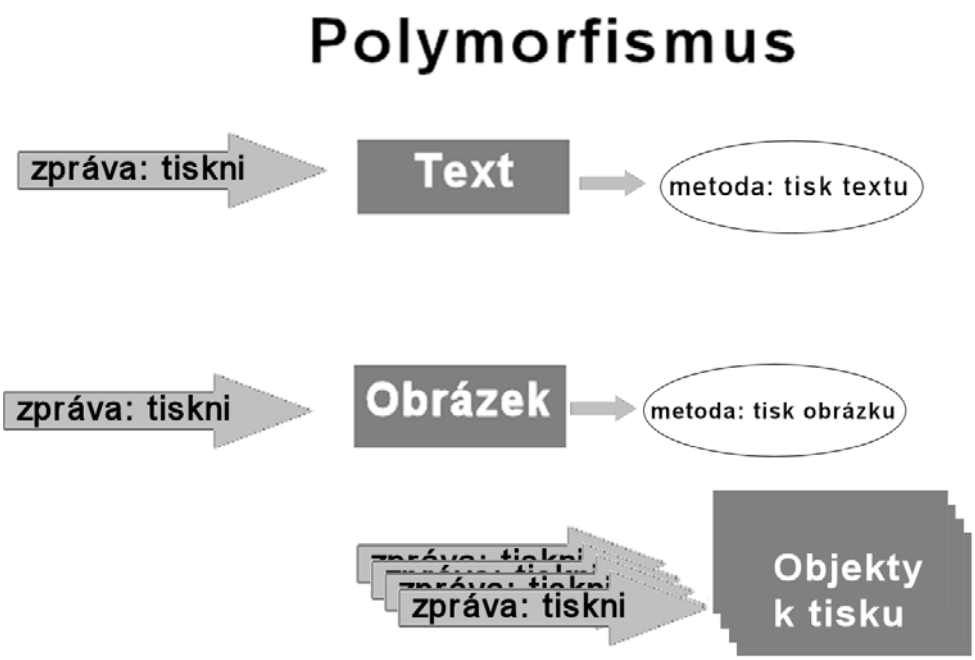
(následníka) a z ní další specializovanou třídu C (následníka), pak o třídě B mluvíme jako o přímém následníkovi. Analogicky je třída B pro třídu C přímým předchůdcem.

Základním typem dědičnosti je *jednoduchá dědičnost*. Ta dovoluje každému následníkovi mít jediného předka a hierarchie struktur pak má tvar stromu. V závislosti na použitém implementačním jazyku je možná i *vícenásobná dědičnost*. Ta v praxi znamená, že atributy i metody určené pro manipulaci s nimi dokážeme zdědit z více tříd. Počet předků v hierarchické struktuře není pro jednoho následníka omezen, a pokud vytvoříme grafickou reprezentaci takového návrhu, vznikne obecný acyklický graf.

Jaké atributy (resp. operace) budou třídou poskytnuty ostatním třídám, ovlivňujeme modifikátory. Pokud má vlastnost (operace) modifikátor *public*, je viditelná i pro všechny ostatní třídy. Jestliže má nastavenu hodnotu *private*, viditelnost je omezena jen na samotnou instanci dané třídy. Poslední možností je modifikátor *protected* - zde kromě instance třídy je informace k dispozici i následníkům v třídní hierarchii.

Polymorfismus

Posledním důležitým odvozeným nosníkem objektově orientovaného přístupu programování je polymorfismus. Ten nám umožňuje pojmenovat podobné metody pro zpracování různých typů objektů (případně více druhů zpracování stejného objektu) stejně. Objektový systém pak sám vybere, jakou metodu pro zpracování použije.



Obr. 2.2 Systém fungování polymorfismu

Mějme například dva objekty různých tříd (int, Long). Zašleme oběma stejnou zprávu (např. abs) a systém provede automaticky výběr vhodné metody ke zpracování podle typu objektu. Je tomu tak proto, že v protokolu zpráv obou objektů bude zpráva s odkazem na jinou metodu a dojde tedy k zavolání jiné metody s parametrem volaného objektu a nakonec k navrácení výsledné hodnoty zprávě. Polymorfismus je tedy důsledkem implementace objektových systémů.

Rozhraní objektu, typ objektu

Jak jsme si řekli výše v bodu 2.1.1. (oddíl operace, metody, zapouzdření), s objektem je kvůli zapouzdření možné komunikovat pomocí operací, představovanými metodami, prostřednictvím zasílání zpráv. Pojem operace bývá s metodou často zaměňován. Pokud však máme příklad předka definujícího operaci a jeho následníci tuto operaci přepíší svou implementací, existuje v systému jediná operace implementovanou několika metodami.

Každá operace definuje název operace, typy všech předávaných parametrů a typ návratové hodnoty operace pod souhrnným názvem *signatura* operace. Signatura musí být v rámci objektu jedinečná, aby systém věděl, jakou metodu má zavolat na jakou příchozí zprávu. *Rozhraní objektu* je tvořeno sadou signatur objektu, tedy seznamem všech zpráv, jež můžeme objektu poslat. Určitou část tohoto rozhraní můžeme označit jako *typ*. Objekt je daného typu tehdy, pokud má ve své tabulce protokolu zpráv implementovány všechny operace definované v rozhraní tohoto typu. Z toho vyplývá, že objekt může být současně více typů, ale na druhou stranu i objekty různých tříd mohou být stejného typu.

Rozhraní objektu je jedinou možností komunikace s ním. Rozhraní nesouvisí přímo s implementací a tak různé objekty s různými implementacemi mohou mít shodné rozhraní a vůči okolí se mohou shodně chovat. Tato vlastnost zapouzdření je pro rozsáhlé projekty velmi výhodná.

Rozhraní objektu je také důvod, proč může fungovat princip polymorfismu (viz bod 2.1.2. – oddíl polymorfismus). Využívá se u něj faktu, že při komunikaci s objektem se objektu pošle zpráva a chování aplikace závisí i na způsobu, jakým objekt zprávu zpracuje. Vzniká zde termín *dynamická vazba* - za běhu aplikace dochází k přiřazení zprávy přicházející objektu s jednou metodou z protokolu zpráv objektu (V jazyce C++ dochází k dynamické vazbě, jen pokud vytvoříme tzv. virtuální metody a k názvu metody přidáme klíčové slovo *virtual*. V opačném případě dojde při volání metody k brzké/statické vazbě již v době překladu.). Můžeme tedy zaměnit objekty se shodnými rozhraními a měnit vztahy objektů v průběhu chodu programu.

V praxi je nutné si uvědomit, že kromě *dědičnosti třídní*, při níž dochází k implementaci objektu pomocí jiného objektu (a o níž jsem se zmiňoval výše), existuje také *dědičnost rozhraní*. Je to způsob, jak místo určitého typu, použít typ jiný.

2.2 Historie návrhových vzorů

Název oboru návrhových vzorů vznikl překladem anglických slov design patterns. Jejich původ pochází ze stavebnictví, kde si lidé (podobně jako později v informačních systémech) všimli, že řešené problémy se stále opakují.

Christopher Alexander definoval v [4] návrhové vzory takto: „Každý návrhový vzor popisuje problém, který se vyskytuje znovu a znovu v našem prostředí a popisuje jádro řešení tohoto problému takovým způsobem, že můžeme použít toto řešení mnohokrát, aniž bychom dělali stejnou věc dvakrát.“

Poprvé se pojem návrhové vzory v kontextu informační technologií objevil v roce 1987 na konferenci Object-Orienter Programming, Systems, Languages and Applications. Sloužily tehdy jako podpora začínajících programátorů v systému Smalltalk.

Později - kolem roku 1991 představil Erich Gamma několik prvních návrhových vzorů, o nichž bude řeč v této práci. Již v roce 1989 sestavil Jim Coplien několik specifických vzorů v podobě úseků programovacích jazyka C++.

Současné rozdělení návrhových vzorů se nezměnilo od přelomové publikace vydané Erichem Gammou, Richardem Helmem, Ralphem Johnsonem a Johnem Vlissidesem v roce 1994 s názvem Design Patterns: Element sof Reusable Object-Orienter Software. Tato kniha byla v roce 2003 vydána i v Češtině pod názvem „Návrh programů pomocí vzorů. Stavební kameny objektově orientovaných programů.“ a je také mým zdrojem informací ([1]). V roce 2005 byla oceněna součástí SIGPLAN (Special Interet Group on Programming Languages) organizace ACM (Association for Computing Machinery, www.acm.org) za „významný a trvalý přínos na poli programovacích jazyků“ [11]. Nutno ovšem podotknout, že toto ocenění vyvolalo značně rozporuplné reakce mezi odbornou veřejností.

Říjen roku 1993 dal základy nevýdělečné skupině Hillside Group zaměřující se na zlepšení návrhu SW systémů a pořádající konference k tomuto tématu. Jsou na nich mimo jiné analyzovány nově vzniklé návrhové vzory.

3 Úvod do návrhových vzorů

Návrhové vzory se snaží zefektivnit vývoj informačních systémů objektově orientovaným způsobem tak, že kladou důraz na maximální znovupoužití už vyvinutého úsilí. Jedná se vlastně o obecný popis řešených problémů. Ačkoli se často může zdát, že řešíme stále nové problémy, většina vznikajících problémů pochází z použitých nástrojů a má tedy společné jádro. Lze je vyjádřit obecně a k nim definovat obecná pravidla, jak dané problémy řešit.

Prvotně můžeme návrhové vzory rozdělit na implicitní a explicitní. Implicitní návrhové vzory vznikají sbíráním zkušeností každého programátora a nosíme si je každý ve své hlavě jako postupy urychlující a usnadňující řešení klasických problémů.

Daleko menší skupinu tvoří vzory explicitní. Jedná se o podrobně popsané a formálně zdokumentované postupy užité při vývoji objektově orientovaným způsobem. Základními vlastnostmi návrhových vzorů je řešení netriviálních a opakujících se problémů.

Návrhové vzory souvisejí s designovou stránkou návrhu systému a jejich základním smyslem je usnadnit design. Ovlivňují tedy až fáze návrhu systému po provedení analýzy požadavků zákazníka.

3.1 Základní prvky návrhových vzorů

K posílení efektivity komunikace mezi návrháři systémů, je nutné, aby používali společný systém zápisu návrhových vzorů. Vzory nepopisují konkrétní řešení jednoho problému, ale snaží se o obecný popis obecně se vyskytujícího problému. Zde však vzniká mírná kolize, jelikož pochopení něčeho čistě abstraktně popsaného je pro člověka obvykle obtížné. Proto je každý vzor doprovázen příkladem konkrétního použití (tzv. motivace vzoru), aby si jej programátoři byli schopni lépe osvojit. Fakticky je motiv prvním použitím vzoru, jež vedl autory k zavedení popisovaného vzoru.

Nyní přistoupím k vyjmenování základních prvků návrhových vzorů – použiji strukturu definovanou v [1]:

- Jméno vzoru
- Klasifikace vzoru - zařazení do kategorie (viz kapitola Rozdělení vzorů v bodu 4)
- Účel (intent) – stručný popis účelu návrhového vzoru o délce jedné až dvou vět
- Alias – alternativně se vyskytující označení návrhového vzoru
- Motivace – příklad určený k vysvětlení konkrétního použití návrhového vzoru
- Struktura vzoru – uvádí popis nutný k aplikaci vzoru tak, aby výsledek odpovídal záměru návrhového vzoru. Pro snazší pochopení postupu se často využívá diagramů (obvykle využívajících notaci UML – Unified Modeling Language)
- Součásti – účastníci návrhového vzoru a jejich role ve vzoru

- Spolupráce – koexistence prvků z kapitoly Součásti
- Důsledky použití – jak dopadla aplikace návrhového vzoru na řešený problém
- Implementace – popis rizik, jež při použití daného vzoru ohrožují výsledný záměr
- Související vzory – pro konstrukci značného počtu složitějších řešení se používá kombinace několik vzorů. Seznam vzorů je uveden pod tímto bodem. Pokud je možné nějaký vzor použít jako alternativu k vysvětlovanému vzoru, je zde tato skutečnost také uvedena.
- Známá použití – uvádí seznam konkrétních případů použití využívajících popisovaný vzor

V některých dalších materiálech se popisovaná struktura mírně liší. Protože však většina publikací, prací i článků ve svých definicích logicky vychází z knihy autorů vzorů [1], bude lepší se držet jejich definice. Nicméně pro pořádek zde uvedu i tabulku „mapující“ pojmenování struktury použité v často citované diplomové práci [2], jelikož vychází z výrazů a struktury často se vyskytující na Internetu.

GoF: Design patterns [1]	**Dvořák M.: [2]**
Jméno a klasifikace	Název
Účel Jiné názvy Motivace Aplikovatelnost	Problém Alias Podmínky
Struktura Součásti Spolupráce	Řešení
Důsledky Implementace	Odůvodnění a souvislosti Výsledek
Ukázkový kód	Příklady
Známá použití	Reference
Související vzory	Související vzory

Tabulka 3.1 Struktura popisu návrhových vzorů [10]

3.2 Společné principy návrhových vzorů

Návrhové vzory využívají některé společné principy, které je obecně vhodné využít při programování a popíši je v této kapitole.

3.2.1 Upřednostnění rozhraní

Pokud využijeme dědičnosti tříd, podaří se nám ušetřit značné množství úsilí i kódu. Na využití dědičnosti pro definování objektů se společnými rozhraními závisí funkce polymorfismu. Jestliže z určité abstraktní třídy budeme dědit operace pouze přidáváním, nebo překrýváním, vzniknou podtypy abstraktní třídy, které jsou schopny zpracovat zprávy ve „formátu abstraktní třídy“. Pokud objekty dodrží rozhraní definované abstraktní třídou, klienti (abstraktní třídu využívající) nemusí znát vnitřní strukturu objektů/tříd a postačuje jim znalost rozhraní.

Čtveřice autorů vzorů v [1] tento princip označuje „Programujte pro rozhraní, nikoliv pro implementaci“ a dále pak „Nedeklarujme proměnné, aby byly instancemi určitých konkrétních tříd. Místo toho se zavážeme pouze pro rozhraní definované abstraktní třídou. Dříve, či později zjistíme, že se jedná o společné téma návrhových vzorů.“

3.2.2 Priorita objektové skladby

Prvním ze dvou postupů pro opětovné využití kódu je dědičnost. Umožní nám využít už jednou naprogramovaný kód a pro třídu následníka použít principy třídy předka. Protože jsou procesy probíhající v předkovi viditelné i třídě následníka, používá se pro tento fakt pojem *znovupoužití bílé skříňky*. Prvoplánově to může vypadat jako výhoda, ale skrývá se zde více problémů, nežli užitku. Její použití je jednodušší, protože nám vývojové nástroje při chybě zahlásí vzniklý problém už v době překladu a navíc znovupoužití existující implementace urychluje vývoj.

Implementace zděděné třídy však zůstává konstantní a nemůžeme ji po sestavení jen tak změnit. Další problém vzniká při potřebě změny třídy na generalizované straně. Vlivem dědičnosti musíme skoro vždy měnit i třídu zděděnou, aby vše fungovalo, jak má. Charakteristická vlastnost dědičnosti, tedy svázanost s třídou předka vedla ke vzniku výroku „dědičnost porušuje zapouzdření“ (Snyder A. Encapsulation and inheritance in object-oriented languages, strany 38-45, Portland, 1986). Doporučuje se tedy předchozí pravidlo, abychom se snažili dědit od abstraktních tříd – tedy pouze rozhraní. Sice se možná některý kus kódu vyskytne duplicitně, ale ušetříme si jiné problémy.

Druhou nejpoužívanější možností zvýšení efektivity kódu je tzv. *objektová skladba*. Nové funkce tentokrát vznikají za běhu aplikace skládáním existujících dílčích objektů. Protože objekty vidí vzájemně pouze svá rozhraní, vžil se také pojem *znovupoužití černé skříňky*. Je třeba, aby byly

objekty navzájem kompatibilní svými rozhraními (resp. byly shodného typu ve smyslu definovaném v bodu 2.1.2 – oddíl rozhraní objektu, typ objektu). Výsledkem preferování objektové skladby je návrh obsahující méně tříd a více vzájemně kompatibilních objektů. Výhodou může být v případě větších projektů zjednodušení návrhu třídní hierarchie.

3.2.3 Princip delegování

Umožňuje použít objektovou skladbu v praxi pro vykonání operací, jež původní objekt nezpracuje, a jinak bychom řešili dědičností. Princip delegování zajišťují nejméně dva objekty – první požadavek (resp. zprávu do protokolu zpráv) jen přijímá a přeposílá jinému objektu. Přijímající objekt si tedy musí držet instanci třídy delegáta, aby přes rozhraní mohl zprávu přeposlat. Jelikož používáme objektovou skladbu, tak pozorný čtenář jistě očekává výhody nastíněné v bodu 3.2.2 tedy zaměnitelnost za chodu. Při korektně definovaných typech můžeme ve spuštěné aplikaci vyměnit delegáta a zprávy přeposílat na jiný objekt - využijeme tedy vlastnosti jiného objektu. Nutnou podmínkou toho, abychom mohli prohlásit, že se jedná o delegování, je fakt, že zpráva odesílaná delegátovi musí obsahovat odkaz na přijímající objekt, aby mohla být navrácena výsledná hodnota.

4 Rozdělení návrhových vzorů a jejich popis

Návrhové vzory rozdělila už skupina Gang of Four (jak jsou často nazýváni autoři prvních třiadvaceti vzorů – [1]) na tři skupiny dle jejich funkce. Dodnes se toto rozdělení nezměnilo:

- tvořivé vzory (Creational patterns), v některých českých publikacích označované jako „tvořící“, případně „vytvářecí“
- vzory strukturální (Structural patterns)
- vzory chování (Behavioral patterns)

Každý návrhový vzor je začleněn do některé z těchto tří skupin. Dále bychom mohli rozdělit vzory podle kritéria, zda vzory organizují třídní strukturu, nebo samotné objekty. Objekty na rozdíl od napevno deklarovaných tříd lze měnit za běhu aplikace, takže objektové vzory poskytují lepší možnosti dynamických změn.

Hlavním nástrojem třídních návrhových vzorů je dědičnost, zatímco objektové návrhové vzory využívají pro své fungovaní mechanismu skládání objektů, tak jak jsem je popsal v bodu 3.2.2.

Není mým úmyslem popsat v následujících odstavcích všechny vzory. Jednak je jich velké množství a další stále vznikají. Uvedu zde popis těch pravděpodobně nejčastěji používaných, jak jej sestavili ve své práci autoři [1] v části „Předmluva experta“. Tyto vzory pomohou začátečníkovi zorientovat se v problematice návrhových vzorů a udělat si zběžnou představu o nich. Pokud bude chtít čtenář pátrat dále, bude lepší jej odkázat na odbornou literaturu – zejména [1]. V případě problematického chápání použití vzorů se mi osvědčila publikace Freeman Eric, Freeman Elisabeth: Head First Design Patterns a také webová stránka [6] znázorňující netechnickým způsobem principy návrhových vzorů na příkladech z běžného života.

V popisu jednotlivých vzorů jsem bod klasifikace vzoru neuváděl znovu - rozřazení je dáno kapitolou, ve které jsou vzory uvedeny. Část důsledky návrhových vzorů jsem převedl na body výhody a nevýhody, neboť obvykle mapuje právě (ne)výhodné vlastnosti návrhových vzorů a lépe to také odpovídá třetímu bodu zadání diplomové práce.

4.1 Tvořivé vzory – Creational Patterns

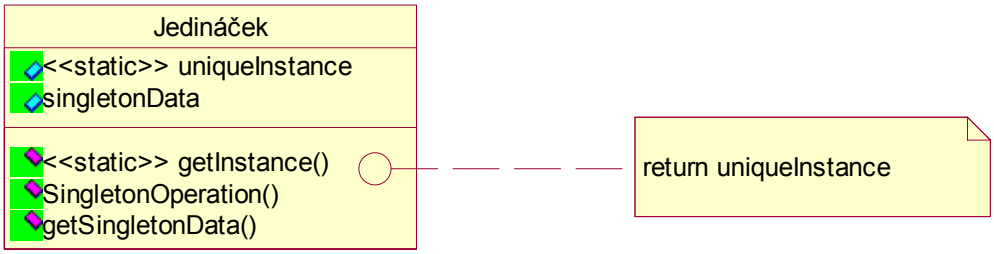
Tyto vzory se zaobírají tvorbou objektů v systému. Popisují postup výběru třídy objektu (často za běhu programu).

Společným hlediskem tvořivých vzorů, jak uvádí [1] na straně 95 je zapouzdření znalostí, které konkrétní systém používá a navíc způsob, jak jsou instance tříd vytvářeny a spojovány dohromady. Systém zná pouze interface definovaný abstraktními třídami.

4.1.1 Jedináček – Singleton

- Účel – zajištění existence jediné instance určité třídy a poskytnutí přístupu k ní
- Motivace

Existují třídy, u nichž je žádoucí, aby mohly existovat jen jednou, resp. aby mohla být instanciována pouze jediná jejich instance. Představme si například Taskmanager. Pokud by se v systému vyskytovalo více instancí Taskmanageru, těžko by mohl takový systém pracovat, protože by pravděpodobně docházelo k „soupeření“ o systémové prostředky. V případě navrhovaného programu takovým příkladem může být hlavní okno aplikace. Je nutné zajistit, aby mohlo být vytvořeno pouze jeden krát.



Obr. 4.1 Diagram vzoru Jedináček

- Struktura vzoru a spolupráce

Podle návrhových vzorů je nejsnadnější cestou, jak jedinečnosti dosáhnout, použití vzoru Singleton. Třída, již označíme jako Singleton, musí sama zajišťovat, aby nemohla vytvořit více instancí než li právě jednu. Zároveň by měla být snadno dostupná z celého systému.

- Součásti

Jedinou součástí Jedináčka (obr. 4.1) je třída obsahující operaci getInstance() typu static/final (v prostředích C/Java).

- Implementace

Více než u jiných vzorů je implementace Jedináčka platformově závislá. Obvykle je Singleton vytvářen „pozdně“ (tak jsem přeložil termín „lazily“) – neexistuje, dokud není poprvé zapotřebí.

Jelikož k implementaci použiji prostředí C#, další postup se bude týkat tohoto jazyka (problém s jazykem Java – viz odkaz v bodu Nevýhody použití). Konstruktor je bez parametrů a je typu private, což zabraňuje vytvoření podtříd. Statická proměnná uniqueInstance pak drží odkaz na jedinou instanci třídy. Je třeba dávat pozor na fakt, že na Internetu koluje celá řada nesprávných implementací popsaných například v [13].

- Výhody použití

  Při správné implementaci máme jistotu existence jediné instance třídy v systému, nad níž aplikujeme vzor Jedináček. Na druhou stranu je možné Jedináčka snadno modifikovat tak, aby kontroloval i jiný počet instancí a změnit tak jeho funkcionalitu.

- Nevýhody použití

  Problematická implementace u multithreadových aplikací běžících na vícejádrových (víceprocesorových) systémech. O problémech implementace v jazyce Java, se lze dočíst na webové stránce [12].

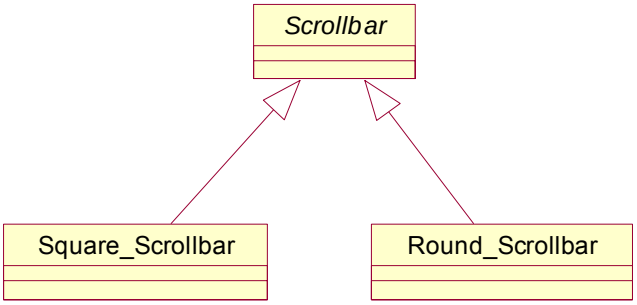
- Související vzory

  Často jej využívají vzory Abstraktní továrna a Stavitel.

4.1.2 Abstraktní továrna – Abstract Factory Method Pattern

- Alias – KIT
- Účel – vytvoření rozhraní pro přepínání skupin objektů a oddělení klienta od volání tříd těchto objektů
- Motivace

Dnešní aplikace umožňují často volbu grafického rozhraní mezi „jednoduchým-základním“ a „rozšířeným“ grafickým rozhraním určeným pro pokročilejší uživatele aplikace. Pokud budeme vytvářet aplikaci s přepínatelným uživatelským rozhraním, budeme potřebovat, aby aplikace uměla pracovat s oběma typy objektů a ještě by je měla být schopna přepínat za běhu. Aplikace tedy bude potřebovat zavést například objekt square_scrollbar (posuvník) pro první grafické rozhraní a objekt round_scrollbar pro druhé rozhraní. Abychom mohli rozhraní přepínat, přidáme do aplikace abstraktní třídu scrollbar (jak ukazuje obr. 4.2). Její podtřídy square_scrollbar a round_scrollbar pak implementují samotný vzhled objektu posuvník.

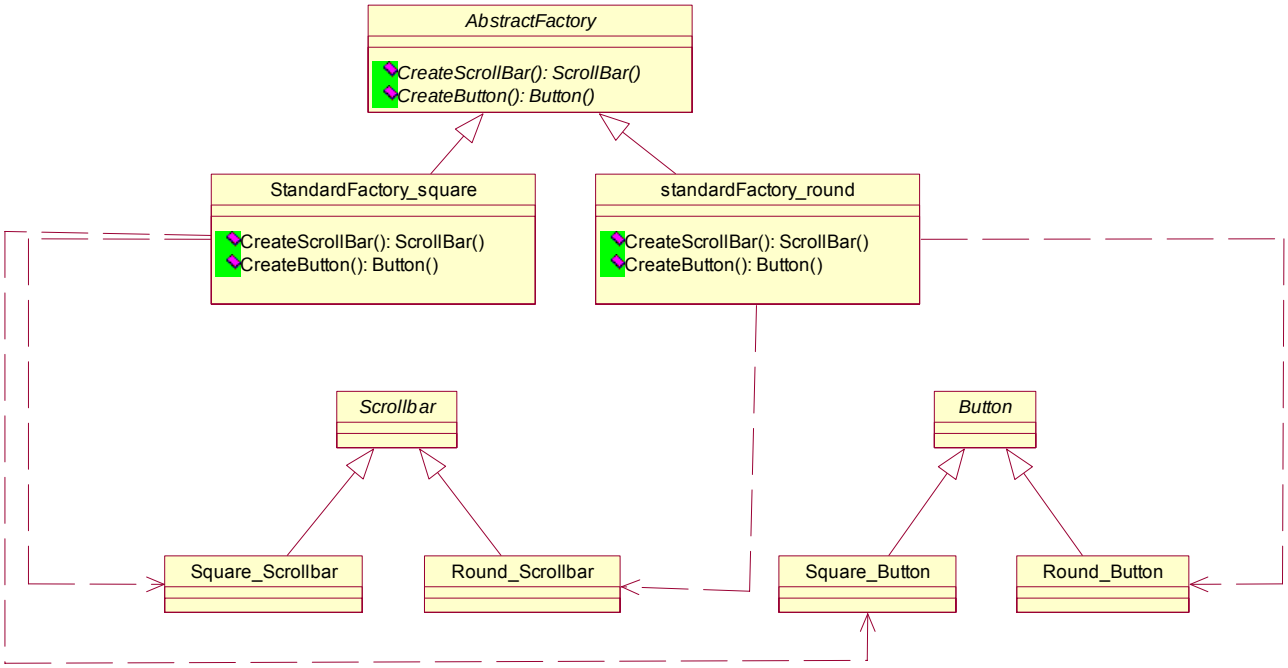


Obr. 4.2 Abstraktní třída scrollbar

Při práci s posuvníkem je nutné zvolit správnou třídu. Aplikaci je možné řešit pomocí skupin globálních proměnných se stavem výběru a globálními funkcemi pro přepínání. Navíc každá část kódu přistupující k posuvníku by musela volat funkci mujScrollbar=FCreateScrollbar().

Protože prvků jako scrollbar obsahuje grafické rozhraní obvykle velké množství a pro každou třídu GUI bychom museli vytvářet nejen abstraktní třídu, ale i funkce vytvoření (CreateScrollBar(), CreateButton()), je jednodušší, když budeme přepínat GUI pomocí polymorfního volání interface objektu Factory. Vytvoříme tedy abstraktní třídu AbstractFactory a dvě odvozené třídy StandardFactory_square a StandardFactory_round (obr. 4.3). Pro vznik nových objektů scrollbar a button má třída AbstractFactory abstraktní operace CreateScrollBar() a CreateButton() vracející typy ScrollBar a Button (a podobně by měla další operace pro jakýkoliv další prvek GUI). Její odvozené třídy StandardFactory_square a StandardFactory_round tyto operace přepisují a implementují.

Zde se poprvé setkáváme výhodnou s kombinací více návrhových vzorů. Jednotlivé konkrétní továrny, tedy třídy vytvářející volitelné skupiny objektů (v našem případě grafických motivů) jsou nejčastěji Jedináčky. Jejich existence je vyžadována pouze jednou a vícenásobná přítomnost v programu by způsobila jeho nefunkčnost.



Obr. 4.3 Abstraktní třída AbstractFactory

Při porovnání s původním návrhem řešení jsme namísto vytváření zrodových funkcí pro každý ovládací prvek zavedli skupinu operací na interface třídy Factory. Klienti vytvářejí prvky uživatelského rozhraní pomocí interface AbstractFactory a nevědí nic o konkrétních podtřídách. Jak je uvedeno v [1] na straně 101: „Klienti se pouze musí zavázat k rozhraní definovanému abstraktní třídou, nikoli k určité konkrétní třídě.“

- Struktura vzoru

Pro proměnná prostředí zavedeme nejprve třídu AbstractFactory a pro třídy představující jednotlivé možnosti volby zavedeme třídy ConcreteFactory1 až ConcreteFactoryX. Pro každou třídu proměnných prvků jsme zavedli abstraktní třídu Product. AbstractFactory zavádí interface pro tvorbu objektů. Odvozené třídy implementují režim možné volby a klienti pro konstrukci objektů nevolají jejich vlastní konstruktory, nýbrž spouští metody interfacu AbstractFactory().

- Součásti

AbstractFactory – deklaruje rozhraní pro metody vytvářející abstraktní skupiny objektů

ConcreteFactory (StandardFactory_square, StandardFactory_round) – implementují operace AbstractFactory pro vytváření skupin objektů

AbstractProduct (Scrollbar, Button) – deklaruje rozhraní pro typ objektu

ConcreteProduct (Square_Scrollbar…) – implementuje rozhraní deklarované v AbstractProduct

- Výhody použití

Pokud potřebujeme za běhu volit mezi více skupinami tříd, zjednodušuje a zpřehledňuje tento návrhový vzor strukturu kódu. Zamezíme tak zmatku při práci se skupinami produktů.

- Nevýhody použití

  Problémem tohoto vzoru jsou komplikace vznikající, pokud je třeba přidat novou skupinu tříd – nový produkt. Do všech odvozených tříd Factory je třeba přidat novou operaci. Tento vzor také nemá smysl použít, pokud se přepínané skupiny tříd neodlišují dost podstatně.

- Související vzory

  Factory method – třídy ConcreteFactory jsou implementací továrních metod.

  Jedináček (Singleton viz 4.1.2) – bývá využit pro reprezentaci ConcreteFactory

4.2 Vzory strukturální – Structural Patterns

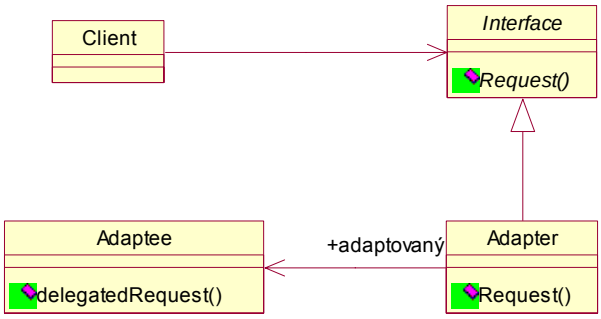
Vzory struktur se zabývají, jak již název napovídá, uspořádáním tříd a komponent programu tak, aby byl systém co nejpřehlednější a nedocházelo v něm ke zbytečnému opakování. Podobně jako u vzorů tvořivých jsou nástroji třídních vzorů dědičnost (např. sloučením více tříd do jedné případně jako v případě třídního adaptéru privátním zděděním od adaptované třídy) a u objektových vzorů skládání objektů.

4.2.1 Adaptér - Adapter Pattern

- Alias – Wrapper
- Účel – Vytváří rozhraní, jež pro svou komunikaci očekává klient. Umožňuje komunikaci tříd s různými interfacy.
- Motivace

  Typicky situace pro použití návrhového vzoru Adaptér nastane, pokud potřebujeme do existujícího programu/systému napojit nějaké třídy, jež s původní aplikací nesouvisely. To nastává například při zajišťování zpětné kompatibility během vývoje zcela nových verzí existujících systémů. Tento návrhový vzor umožňuje vytvořit mezičlánek, jímž se obě nekompatibilní části „slepí“ dohromady.

- Struktura vzoru



Obr. 4.4 Návrhový vzor Adaptér – objektová verze

Klient zde využívá jemu známé rozhraní definované v abstraktní třídě Interface. Diagram 4.4 znázorňuje převod požadavku metody Request() třídy Interface na požadavek delegatedRequst() třídy Adaptee. Adaptér zde funguje jako spojka mezi metodou Request() a přesměrovanou metodou objektu Adaptee a přizpůsobuje Adaptee na rozhraní Interface.

- Součásti

  Client – pracuje s objekty komunikujícími pomocí rozhraní Target

  Target (Interface) – definováno rozhraní známé klientovi

  Adaptee – určuje rozhraní, jež je nutno přizpůsobit

  Adapter – modifikuje interface Adaptee na cílový interface Target

- Implementace

Existují dvě varianty adaptéru – objektový a třídní adaptér. Na obrázku 4. 4 je znázorněna objektová varianta adaptéru. Třídní varianta by naproti tomu dědila nejen veřejně rozhraní Interface, ale také privátně implementaci třídy Adaptee.

- Výhody použití

Výhodou zde uvedené varianty objektového adaptéru je, že dokáže kromě samotné třídy Adaptee pracovat i se všemi jejími potomky a navíc může přidat funkcionalitu všem těmto třídám najednou.

- Nevýhody použití

Nastává problém s potlačením vlastností třídy Adaptee – řešíme třídou dceřinou od Adaptee a navíc napojením Adaptéru na tuto dceru – ztrácíme ovšem výhodu vyjmenovanou o odstavec výše.

- Související vzory

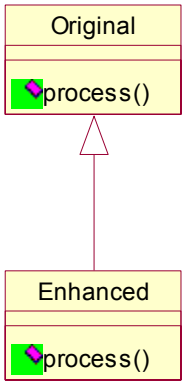
  Dekorátor – viz 4.2.2

  Most – podobá se strukturou adaptéru, ale účelem je oddělení interface od jeho implementace. Na rozdíl od adaptéru je využíván převážné už při počátečním návrhu systému.

4.2.2 Dekorátor - Decorator Pattern

- Účel – nabízí dynamické přidávání funkcionality instancím třídy
- Motivace

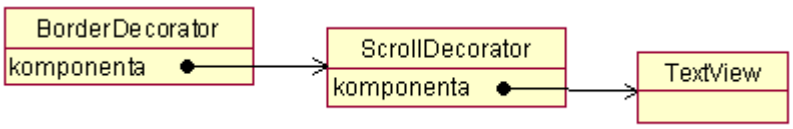
Mějme třídu Original s metodou process(). Nyní potřebujeme soubor funkcí metody process rozšířit o další akce. Pokud bychom to provedli použitím dědění, jak je uvedeno na obrázku 4.5, odvozená třída Enhanced by sice obsahovala vlastnosti, které Original neposkytovala, ale toto přiřazení by bylo pevné bez možnosti ovlivnění (natož změny za běhu aplikace).



Obr. 4.5 Rozšíření funkčnosti třídy Original

Potřebujeme tedy použít jiný postup. Pokud místo dědění uzavřeme objekt do jiného objektu - Dekorátoru, můžeme využít flexibilní skládání. Pokud je rozhraní propojení mezi prvky shodné, obsahují prvky jedinou metodu process() a lze je různě vyměňovat i za běhu aplikace. Dekorátor předá žádost o volání metody process() původní komponentě a před či po této akci může provádět další akce, o něž rozšiřuje funkcionalitu původního objektu. Dokážeme tak ovlivnit vlastnosti jednotlivých objektů, aniž bychom museli rozšiřovat vlastnosti třídy daného objektu.

Jak uvádí kniha Návrh programů pomocí vzorů na straně 177 [1], lze princip Decoratoru snadno demonstrovat na rozšiřování možností GUI. Mějme Objekt TextView zobrazující text v okně.

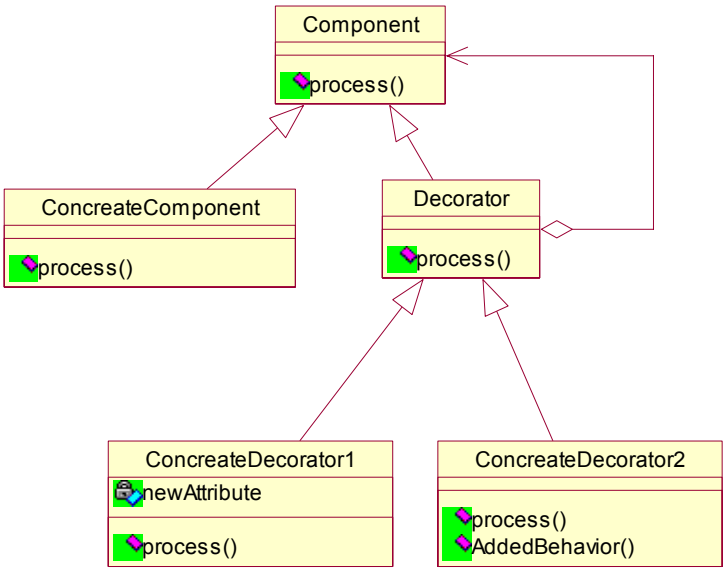


Obr. 4.6 Aplikace návrhového vzoru Dekorátor

K tomuto objektu chceme přidat posuvníky pomocí objektu ScrollDecorator a dále chceme kolem objektu TextView vytvořit rámeček využitím objektu BorderDecorator. Obrázek 4.6 ukazuje propojení objektů, aby výsledkem bylo ohraničené textové pole s posuvníkem.

- Struktura vzoru

Všechny prvky vytvářeného seznamu musí podporovat rozhraní abstraktní třídy Component. Pak je možné je za běhu přeskupovat. Vzor díky tomu rozšiřuje možnosti třídy ConcreateComponent, jejíž prvky tvoří vždy konce řetězce. Třída Decorator za sebe navazuje další prvky a to buď koncový objekt třídy ConcreateComponent, nebo další prvky třídy Decorator, případně jeho dceřiných tříd. Ty jsou opatřeny přidávanými vlastnostmi. Obrázek 4.7 ukazuje, jak ConcreateDecorator1 rozšiřuje množinu atributů o newAttribute a ConcreateDecodator2 přidává metodu AddedBehavior().



Obr. 4.7 Návrhový vzor Dekorátor rozšiřující funkčnost třídy Original

- Součásti

  ConcreteComponent – objekt, jehož možnosti rozšiřujeme

  Component – abstraktní třída s definicí rozhraní objektů

  Decorator – odkazuje na objekt Component

  ConcreteDecorator – prvky obsahující definice přidávaných vlastností

- Výhody použití

  Použití vzoru Dekorátor má jednoznačné výhody, pokud potřebujeme rozšiřovat možnosti prvků aplikace za běhu. Jako typické použití je uváděno přidávání (samozřejmě jiným seskupením docílíme i odebírání) prvků v grafickém uživatelském rozhraní. Navíc došlo k rozšíření možností systému bez nutnosti vytváření mnoha nových odvozených tříd.

- Nevýhody použití

  Tím, že pokaždé potřebujeme přidat nové instance třídy, abychom rozšířili funkcionalitu, může vznikat zmatek vinou nadměrného počtu použitých prvků.

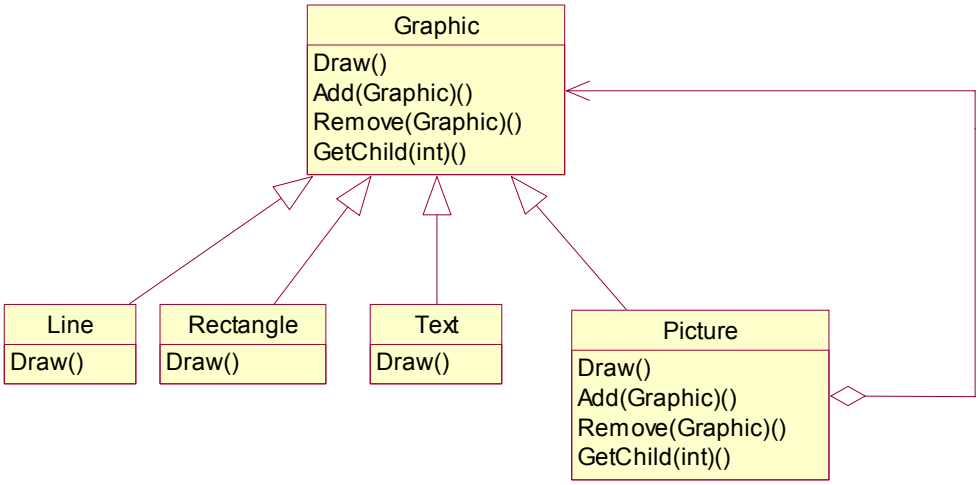
- Související vzory

  Adaptér – na rozdíl od Dekoratéru nemění vlastnosti objektu, ale jeho rozhraní

4.2.3 Skladba - Composite Pattern

- Účel – Vytváří stromové struktury a to tak, že klient nemusí rozlišovat, zda se jedná o jediný objekt nebo o skupinu objektů
- Motivace

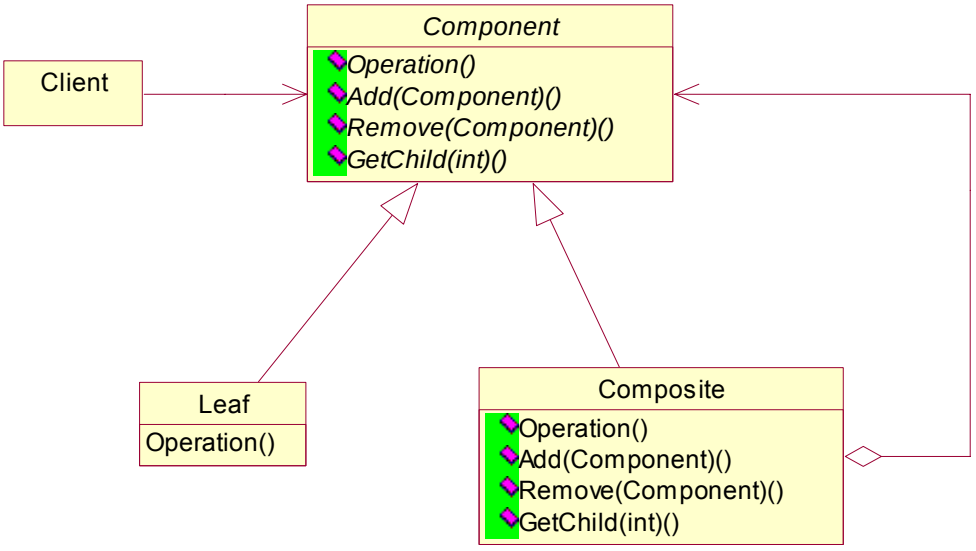
Pokud potřebujeme implementovat strukturu, v níž se mohou nějaké prvky-primitiva seskupovat a to i mnohonásobně, stále do nekonečna, narazíme na problém potřeby s prvky-primitivy zacházet odlišně od prvků seskupujících tato primitiva. Abychom nemuseli stavět aplikaci takto složitě, můžeme vytvořit abstraktní třídu spojující operace všech prvků.



Obr. 4.8 Příklad aplikace návrhového vzoru Skladba

K vysvětlení mi nejlépe pomohl příklad uvedený v [1]. Obrázek 4.8 ukazuje příklad software pro tvorbu diagramů nebo nějakou grafickou aplikaci. Abstraktní třída Graphic deklaruje jak operace grafických primitiv, tak operace složených objektů pro manipulaci s primitivy – přístup k potomkům a jejich správu. Třída Picture umožňuje seskupovat objekty třídy Graphic a její metoda Draw je delegována na její potomky. Třídy Line, Rectangle, Text představují grafická primitiva sloužící k sestavení složitější objektů/obrazců a implementující metodu Draw() abstraktní třídy Graphic.

- Struktura vzoru



Obr. 4.9 Struktura návrhového vzoru Skladba

Abstraktní třída Component (obr 4.9) slouží všem prvkům. Metoda Operation() je volána rekurzivně na prvky tvořící objekt třídy Composite.

- Součásti

Component (Graphic) – definuje interface objektů (operace pro zacházení s potomky), umožňuje přístup k rodičům objektu

Composite (Picture) – implementuje operace potomků interface Component

Leaf (Rectangle, Line, Text) – implementuje metody primitivních prvků

- Výhody použití

Aplikace využívající tento vzor může jednotně přistupovat jak k samostatným prvkům, tak k jejich seskupení v konstruované hierarchii. Klienti nepotřebují řešit, zda právě pracují se seskupením, nebo samostatným prvkem a pro nové třídy Component je není třeba nijak upravovat.

- Nevýhody použití

Pokud potřebujeme hierarchii omezit jen na některé komponenty, musíme toto vyřešit dodatečnou kontrolou v systému, jelikož na nové komponenty neklade žádná omezení.

- Související vzory

Návrhový vzor Dekorátor (4.2.2) je někdy využíván společně se vzorem skladba.

4.3 Vzory chování – Behavioural Patterns

Vzory chování se snaží řešit chování objektů a tříd systému, slouží k vylepšení spolupráce a komunikace mezi objekty a dědičnosti mezi třídami. V této skupině existují opět vzory třídní i objektové. Dědičnost slouží třídním vzorům tentokrát k rozdělení chování, objektové k řízení chování používají, jako u zbývajících skupin, objektovou skladbu.

4.3.1 Pozorovatel - Observer Pattern

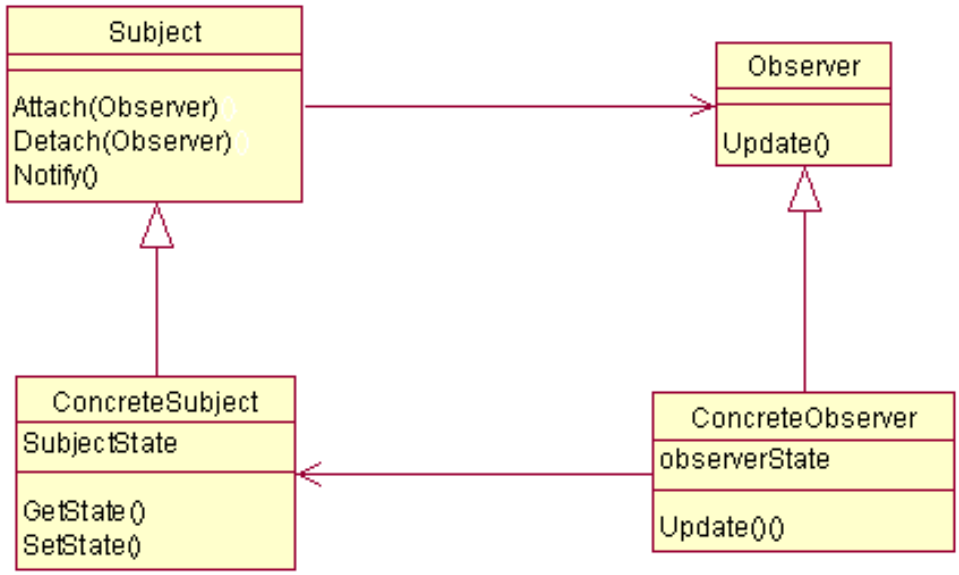
- Účel – poskytuje závislost 1:N mezi objekty včetně obdoby aktualizace hodnoty při změně z databází
- Alias: Dependents, Publish – Subscribe (interakce vydávat – odebírat)
- Motivace

Pro zvýšení znuvupoužitelnosti aplikací je často přistupováno k rozdělení aplikací na více vrstev, jež jsou na sobě do určité míry nezávislé a je možné jejich prostředky použít i jinak. Představme si více prvků grafického rozhraní zobrazující stejná data (tabulka a graf). Samotná data jsou uložena v objektu aplikační logiky, zatímco zobrazování provádí objekty prezentační vrstvy. Je pak přirozeným požadavkem vzájemná synchronizace údajů mezi různými způsoby grafické interpretace údajů.

Je tedy třeba, aby všechny prvky – pozorovatelé – byly upozorněny na změnu stavu pozorovaného prvku – subjektu. Pozorovatelé se u subjektu přihlásí k odebírání informací o změně subjektu. Ten navíc vůbec nemusí o pozorovatelích nic vědět. Pokud dojde ke změně stavu subjektu, informuje subjekt o této skutečnosti všechny své pozorovatele a ti následně začnou komunikovat se subjektem dotazem na jeho stav.

- Struktura vzoru a spolupráce

Seznam pozorovatelů umožňuje udržovat abstraktní třída Subject (viz obr 4.10) operacemi Attach(Observer) a Detach(Observer), které zajistí přidání objektu Observer mezi pozorovatele. Třída Observer obsahuje operaci Update(), již dědí každý konkrétní objekt třídy ConcreteObserver a přepisuje dle potřeby.



Obr. 4.10 Struktura návrhového vzoru Pozorovatel

Pokud nastane u konkrétního subjektu událost, operací update() dojde k přeposlání informace o změně subjektu všem pozorovatelům. Protože objekt ConcreteObserver má přístup k subjektu, může pomocí operace Update() zjistit změnu sledovaného subjektu.

- Součásti

Subject – má seznam svých pozorovatelů a poskytuje rozhraní pro jeho udržování

Observer – objekt, jehož možnosti rozšiřujeme

ConcreteSubject – pokud nastane změna stavu, zasílá informaci o změně pozorovatelům

ConcreteObserver – implementuje aktualizační rozhraní Observeru

- Výhody použití

Díky oddělení pozorovatele a subjektu je možné zajistit synchronizaci různých vrstev programované aplikace. Subjekt se kvůli synchronizaci nemusí nijak zajímat o to, jestli jej někdo pozoruje nebo ne. Stačí, když udržuje seznam pozorovatelů.

- Nevýhody použití

Problémem vyplývajícím z výše uvedeného ovšem je, že jednotliví pozorovatelé se taktéž nijak nezajímají o ostatní pozorovatele a tak i jen minimální změna na subjektu má za následek aktualizaci všech ostatních pozorovatelů (v případě špatného návrhu kaskádovitě vyvolanou rekaci).

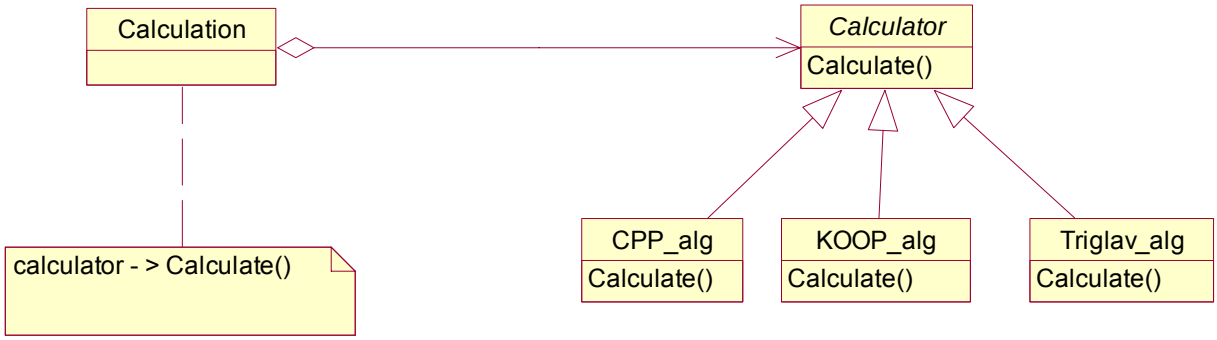
4.3.2 Strategie - Strategy Pattern

- Účel – Definuje řadu vzájemně vyměnitelných algoritmů
- Alias: Policy

- Motivace

Firma, ve které jsem pracoval, pochází z oblasti finančnictví a spolupracuje s řadou pojišťoven. Každá z nich má jiné podmínky výpočtů potřebných algoritmů a navíc existují rozdíly i v rámci jediné pojišťovny mezi produktovými řadami. Aplikace má být například schopna zobrazit provizi obchodního zástupce v době vytváření smlouvy. Je tedy zapotřebí, aby se algoritmus výpočtu měnil za běhu aplikace.

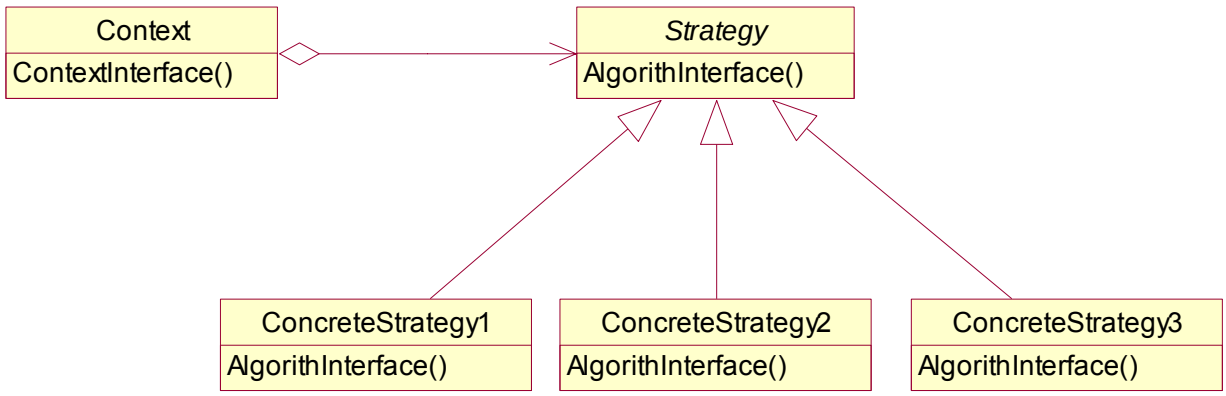
Samozřejmě je možné požadavek implementovat tak, že v systému bude existovat datová proměnna udávající aktuálně používaný algoritmus (výběr bude záviset na zvoleném produktu/pojišťovně). Při zavolání výpočtu provize se podle hodnoty přepínače vybere konkrétní algoritmus a provize se vypočítá. Takovéto pevné naprogramování algoritmů do tříd není vhodné, protože takto roste velikost klientů, jež chtějí algoritmy použít (a s ní i nároky na údržbu zdrojového kódu).



Obr. 4.11 Příklad struktury návrhového vzoru Strategie

Místo toho bude vhodnější nadefinovat třídy zapouzdřující různé formy algoritmů. Zapouzdřené algoritmy se nazývají strategií. Na obrázku 4.11 vidíme, že pro výpočet provize byla zavedena abstraktní třída Calculator zavádějící interface s operací Calculate(). Ten je pak přepsán jejími dceřinými třídami CPP_alg, KOOP_alg a Triglav_alg představující jednotlivé algoritmy výpočtu provize. Výměna objektu do téže pozice se shodným interfacem se rovná záměně algoritmu výpočtu. Klient třídy Calculation specifikuje, jaký Calculator se má použít pro nainstalování příslušného objektu třídy Calculator do Calculation.

- Struktura vzoru a spolupráce



Obr. 4.12 Struktura návrhového vzoru Strategie

Na obrázku 4.12 můžeme vidět, že Context zavádí interface pro volání požadavků klienta a udržuje instanci Strategy. Změna algoritmu je provedena záměnou objektu z dceřinné třídy Strategy.

- Součásti

Strategy (Calculator) – deklaruje rozhraní podporovaným algoritmům, jež využívá Context k jejich zavolání

ConcreteStrategy (CPP_alg, KOOP_alg, Triglav_alg) – implementuje samotný algoritmus přepisem metody AlgorithmInterface() abstraktní třídy Strategy

Context – udržuje kontakt na objekt Strategy

- Výhody použití

Vzor Strategie zvyšuje znovupoužitelnost vyčleněním společných funkcí z algoritmů. Na rozdíl od řešení pevným zděděním algoritmů není naprogramování třídy Context statické a těžkopádné vlivem příliš velkého množství algoritmů, jež ale v dané chvíli nejsou aktuálně potřeba. Pokud použijeme popisované řešení, budeme schopni měnit využívaný algoritmus dynamicky za běhu aplikace. Navíc jeho výměna je usnadněna zapouzdřením algoritmu do speciální třídy Strategy.

- Nevýhody použití

Jako některé další vzory Strategie zvyšuje množství objektu v aplikaci. Navíc zde vzniká určitá komunikační režie mezi třídami Context a Strategy. Protože složitost implementovaných algoritmů může být samozřejmě různá, může se stát i to, že ne všechny třídy ConcreteStrategy využijí všechny informace, jež jim dodá rozhraní. Context tedy někdy zinicializuje atributy, jež se nikdy nepoužijí.

5 Shrnutí výhod a nevýhod použití návrhových vzorů

Návrhové vzory umožňují díky lepšímu pochopení fungování objektově orientovaných systémů dosáhnout podstatně lepšího návrhu. Objevuje se i názor, že jejich vznik byl nutný, kvůli velmi nekonkrétní specifikaci pravidel objektově orientovaného programování. K vymezení přesnějšího způsobu, jako využít OOP jsou vzory zcela jistě vhodné. Obsahují spoustu dobrých nápadů, jak řešit konkrétní problematické situace. Dodržování jejich zásad umožní vylepšit kvalitu návrhu i v případě, že se při vývoji nebudeme držet konkrétního vzoru.

V poslední době dochází ke vzniku nových vzorů a to nejen na nejnižší úrovni, představované návrhovými vzory, ale i na vyšších úrovních náhledu na systém. Jako první jmenujme vzory analytické. Mezi ně řadíme například vzory Accountability, Enterprise Segment, Accounting, využívané při specifikování požadavků a vytváření analytického návrhu. Následují vzory BPMN Workflow (Business Process Modeling Notation) zaznamenávající ustálená řešení modelování firemních procesů – např. vzory Parallel Split, Exclusive Choice, Simple marge a další (viz kapitola Příloha - přednáška Kamila Svobody). Další a určitě ne poslední skupinou jsou vzory pro modelování organizací.

Často se objevují upozornění na to, že ne všechny vzory (nyní již opět mluvíme o klasických návrhových vzorech) můžeme skutečně považovat za návrhové vzory v jejich původním významu. Zopakujme si tedy, že vzor popisuje netriviální, opakovaně se objevující problém. Vznikly na základě reálných situací, které motivovaly jejich tvůrce abstrahovat od tohoto problému obecná řešení.

Musíme si ovšem uvědomit, že návrhové vzory nejsou žádnou panaceou (všelékem). Je zejména třeba dát pozor na to, abychom vzory nepoužívali samoúčelně. Jak píše Joshua Kerievsky v [9]: „Kvalitní softwarový návrhář používá refaktorizaci v mnoha směrech, aby dosáhl cíle lepšího návrhu. Mnoho součinností refaktoringu nevyužívá vzory. Ale pokud jsou vzory součástí mnou aplikovaného refaktoringu, provádím změny nejen ve smyslu jejich použití, ale dokonce i jejich vynechání z aplikace.“ (volně přeloženo).

Autor také v úvodu knihy popisuje příklad, že mnoho programátorů (včetně jeho samotného) je vzory často tak nadšeno, že se jimi začíná zabývat přespříliš a používat je i v situacích, ve kterých sice je jejich místo dle popisu vzoru, ale postup klasickým způsobem (např. sekvence if, else if nahrazovaná vzorem Command) je jednodušší. Jelikož smyslem návrhových vzorů je návrh vylepšit a to jak po stránce možností výsledného produktu, nebo

vlastnosti označované jako podporovatelnost (supportability), neznamená použití vzoru automaticky lepší návrh. Zmiňovaná publikace obsahuje kapitoly pojednávající také o reimplementaci na kód nevyužívající návrhové vzory – s typickými příklady nevhodného použití.

5.1 Možné přínosy použití vzorů

Abych nevrhl na návrhové vzory zbytečně negativní pohled, uvádím zde soupis základní přínosů návrhových vzorů jednoho z autorů a člena „klanu“ GangOfFour John Vlissidese z [15]:

- Zachycují expertní znalosti a řešení a zpřístupňují je méně zkušeným vývojářům
- Pojmenovávají formu těchto řešení a zakládají nový slovníček termínů, což společně pomáhá zlepšit kvalitu komunikace programátorů
- Celý projekt je snadněji a rychleji pochopitelný, jestliže k jeho návrhu, vývoji i dokumentaci byly použity vzory.
- Ulehčují restrukturalizaci systému, ať již platí nebo neplatí, že při jeho vývoji byly vzory použity.

Vlissides původně sepsal tyto přínosy v pořadí dle jejich důležitosti, jak ale poznamenává, časem zjistil, že druhý bod – tedy zlepšení komunikace mezi programátory je možná nejdůležitějším aspektem. Je to logické – jakoukoliv práci se snažíme vykonávat v co nejmenším počtu lidí ne jen kvůli mzdovým nákladům, ale i proto, že zvětšení týmů nemusí zdaleka znamenat zvýšení produktivity práce (jak by určitě potvrdily firmy zaměstnávající programátory bez praxe). Největší překážkou (kromě základní znalosti problematiky vývoje) je vzájemné porozumění a pochopení řešeného úkolu. Problém narůstá s velikostí projektu a množství komunikace nutné pro řešení projektu je značné.

Jakékoliv zvýšení efektivity komunikace je tedy velkým přínosem. Jak jsem podotkl v kapitole 2.2, výrazné vyzdvihování významu návrhových vzorů je některým odborníkům proti srsti a popularita autorů vzorů i obrovský úspěch jimi vydaného katalogu jim připadá absurdní. Zejména argumentují tím, že návrhové vzory nepřinášejí nic nového a jejich myšlenky už dávno používali. V kontextu předchozího odstavce pravděpodobně chápete, proč jsou (minimálně zde uvedené) výtky skeptiků plané. To, že znalost měli a používali ji, přinášelo užitek jen jim samotným. Pokud se expertní znalosti pokusili nějakým způsobem popsat a vydat, byl to jistě pozitivní krok, ale dokud nebyla zavedena všeobecně přijímaná terminologie, význam onoho počinu stále nebyl dostačující. Avšak návrhové vzory jsou obecně známé a přijímané (i když v zahraničí podle všeho výrazně více, než v České republice) a tak přínos v podobě zlepšení kvality komunikace může být velmi patrný.

V případě, že se vývojář dostává k novému projektu a musí se s ním seznamovat od základu, pomáhají vzory podobně jako v předchozím případě pochopení návrhu. Někteří vývojáři však zde upozorňují na problém nazývaný faktor rozložení.

5.2 Komplikující faktory

Návrhové vzory jsou vhodné pro programování, otázkou další ovšem je, nakolik jsou programovací jazyky, nástroje a modelovací techniky vhodné pro návrhové vzory. Vzor samotný je kus relativně snadno pochopitelného kódu. Avšak v praktickém projektu jsou vzory rozmělněny do mnoha tříd a velkého množství kódu. Tomuto rozkladu se říká faktor rozložení. Velmi výrazně znesnadňuje teoreticky snadno pochopitelný problém. Ve chvíli, kdy se v projektu nachází jen několik málo vzorů, tento problém ještě není tak zřejmý. Ale při růstu projektu se setkáme i s komplikací, jak návrhové vzory alespoň znázornit.

Protože je vhodné návrhové vzory navzájem využívat, případně do sebe vnořovat, není často možné diagramy prostě rozdělit. V případě použití stávajících diagramů na komplexnější systémy využívající vzory však dojdeme ke grafům nepřesným nebo neúměrně složitým.

Někteří odborníci si myslí, že aplikace vzorů na již existující objektově orientovaný systém je úspěšnější, než použití přímo při jeho vývoji. Zejména pokud musel být provozován delší dobu a musela být na něm provedena dlouhá řada změn vynucených okolnostmi, bývá kód „zanesen“ prvky, jež bychom tam mít rozhodně nechtěli. Ale protože je obvykle nutné na změněné podmínky reagovat maximálně pružně, neměli jsme čas kód přetvořit tak, abychom se vyhnuli případným nešvarům. V takové chvíli bývá daleko zřetelnější, jestli je vhodné návrhový vzor použít.

6 Softwarové metriky a jejich aplikace

K poměřování kvality softwaru je nutná existence softwarových metrik. Jejich význam ještě vzrostl poté, co za vzrůstající investice do IT mnoho firem nedostalo jako protipól očekávanou hodnotu. Značná část zmíněného faktu má původ už jen v tom, že stále větší kvantum dat podléhá elektronickému zpracování. Obecně řečeno jsou důvody pro zavedení metrik následující:

- dosažení kvality výsledného SW produktu a kontrola kvality
- možnost odhadovat a ovlivňovat produktivitu
- získání vstupních údajů pro plánování projektu
- zdokonalení práce zpětnou vazbou

6.1 Oblasti analyzované metrikami

K zajištění základního smyslu softwarových metrik musíme standardizovat systém měření. Mezinárodní organizace, zabývající se standardy, ISO se dohodla na vytvoření normem pro kvalitu softwarového produktu ISO 9126 a ISO 14598. V současnosti se připravuje už také nástupce (ISO 25000). Druhá jmenovaná se věnuje zdrojům potřebným pro hodnocení softwaru a samotným procesem, jež software produkuje. Proto je pro nás důležitější první norma kontrolující atributy SW produktu a výsledným efektem jeho použití.

Při použití metrik k zjišťování a analýze kvality vztahujeme výsledek k hodnotě nějakého atributu. Norma ISO 9126 jej definuje jako fyzickou nebo abstraktní vlastností entity. Dokonce i pojem kvalita použití má svou definici – Kompatibilita softwarového produktu umožňující specifikovaným uživatelem dosáhnout specifikovaných cílů efektivně, produktivně a bezpečně.

Pro účel měření parametrů existuje jejich základní rozdělení a to na přímé a nepřímé měření. Přímé měření zaznamenává atributy, jež nejsou závislé na jiných metrikách, zatímco nepřímé metriky jsou odvozené z metrik přímých - jednoho i více atributů.

Pokud zjednodušíme definici uvedenou o odstavec výše, zjistíme, že kvalita je definovaná jako míra splnění požadavků uživatele. Oblasti, v nichž sledujeme kvalitu atributů, můžeme dle normy rozdělit na:

- spolehlivost
- funkčnost
- použitelnost
- udržovatelnost
- efektivnost
- přenositelnost

6.1.1 Proces zlepšování softwaru

V případě metodiky se zkratkou SSPI je již z jejího názvu - Statistical software process improvement - zřejmé, o co v ní běží. Pramen [17] k ní uvádí:

1. chyby a defekty jsou kategorizovány podle původu (chyba ve specifikaci, v logice..)
2. je stanovena cena za opravu chyby
3. sečte se počet chyb podle kategorie
4. je stanovena celková cena chyb podle kategorie
5. analyzují se kategorie s nejdražšími chybami
6. snaha modifikovat proces, aby se eliminovala frekvence výskytu chyb v této kategorii

6.1.2 Metriky orientované na velikost

Na základě praxe z již existujících projektů můžeme odvodit po normalizaci faktorem kvality a produktivity:

- Cenu jednoho řádku kódu
- Počet řádků/chyb produkovaných realizátorem za měsíc
- Počet chyb na tisíc řádků
- Počet stran a jejich cenu na tisíc řádků kódu (KLOC)

6.1.3 Funkčně orientované metriky

Funkčně orientované metriky využívají empirický vztah mezi spočitatelným měřením informační domény a odhadem složitosti softwaru pomoci:

- Počet různých vstupů
- Počet výstupů
- Počet dotazů
- Počet vnitřních logických souborů
- Počet souborů na rozhraních

7 Praktická aplikace návrhových vzorů

Čtvrtým bodem zadaní této diplomové práce je vyzkoušet praktické využití návrhových vzorů. Nejprve bylo potřeba vybrat aplikaci, při jejímž vývoji budu návrhové vzory aplikovat. Jelikož mám za sebou větší projekt databázové aplikace pro firmu z oblasti finančnictví, demonstruji využití návrhových vzorů na něm. Zmiňovaný projekt byl součástí mé bakalářské práce.

Bohužel se nejedná o objektově orientovanou aplikaci, takže nebude možné provést přímé porovnání kvalit objektově orientované aplikace bez použití vzorů a po jejich aplikaci, jež by bylo patrně nejnázornější. Vzhledem k faktu, že návrhové vzory je možné aplikovat jen po důkladném seznámení s problémovou doménou, tak pro mne neexistuje vhodnější volba (resp. žádný objektově orientovaný systém jsem, s výjimkou malých školních projektů, doposud neprogramoval).

Uvádím zde návrh implementace těch částí aplikace, ve kterých jsem nalezl místo pro použití některého návrhového vzoru. Výsledný produkt nebude sloužit místo původní aplikace a neexistuje tedy důvod implementovat plnou funkčnost systému. Slouží jako demonstrativní příklad použití vzorů na reálné aplikaci.

Experti zabývající se návrhovými vzory často zdůrazňují, že dobrý návrh vzniká až mnoha revizemi a vytvořit jej napoprvé je možné až získáním velkého množství zkušeností jak z hlediska dokonalé znalosti programovacího jazyka/prostředí, tak návrhu objektově orientovaných systémů. Je nutné vzít v potaz, že všechna navrhovaná řešení nemusejí být ve skutečnosti zcela ideální.

7.1 Seznámení se systémem

Systém byl původně vytvořen pro potřeby malé firmy o několika zaměstnancích na základě databázového systému MS Access tehdy ve verzi 97 a běžel na jediném počítači. Jakmile vznikla potřeba práce na více počítačích, byla realizována rozdělením databáze na programovou a datovou část, kdy datový soubor byl umístěn na „serveru“, nicméně práci s ním zajišťovaly (kromě samotných souborových operací) stále klientské počítače s MS Access.

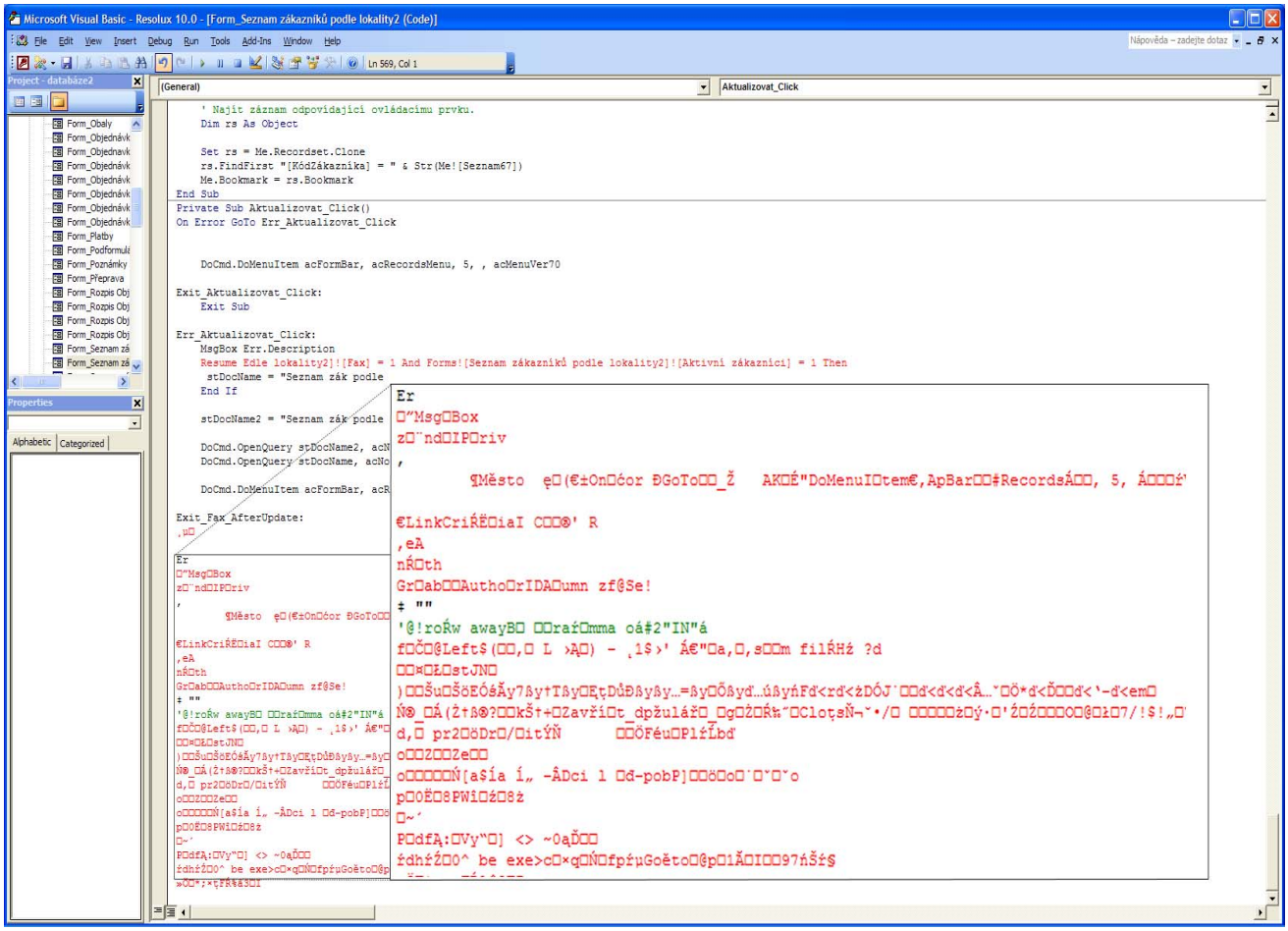
S růstem firmy shledali majitelé tento systém nevyhovující a rozhodli se přejít na serverovou verzi systému. Jeden z důvodů byl i ten, že firma vlastní více poboček a přenos dat mezi jednotlivými pobočkami byl do té doby řešen dosti katastrofálním způsobem - synchronizací samostatných databázových souborů (umístěných na počítačích poboček) s centrální databází firmy. Při tom vznikaly situace, že klienti byli na různých pobočkách evidováni pod různými identifikátory. Jednou za čas se tedy musel provést import dat do centrální databáze a musely být vytvořeny nové verze databázových souborů pro pobočky.

Přechod byl po přihlédnutí především k ceně, komplexnosti a budoucím možnostem řešení uskutečněn na server PostgreSQL (skoro jediný SQL server, jež je skutečně zdarma a to i pro

komerční využití). Jeho výhodou bylo kromě nezávislosti na operačním systému také velký počet volně dostupných utilit pro správu, jež výrazně rozšiřují možnosti jeho využití. Majitelé nebyli ochotni investovat do nákupu licence MS SQL serveru, i když takové spojení by bylo z hlediska výkonu i možností pravděpodobně nejoptimálnější. MS Access by mohl s MS SQL serverem komunikovat přes nativní ovladače přímo a nemusela by využívat překlad ODBC (Open Database Connectivity – tedy neproprietární komunikační protokol).

7.2 Problémy původního systému

Hlavní nevýhodou formulářů aplikace MS Access obecně je právě jejich „absolutní zapouzdření“. Vývojář nemůže vůbec zasahovat do systému fungování formulářů a k dispozici má pouze modul kódu náležející k jednoduché obsluze procedurálním způsobem. V důsledku to pak znamená faktickou nemožnost využít dědičnosti, i když samotný Access podporuje takzvané Class Modules – tedy moduly kódu psané objektově. V samotných formulářích je ale jejich využití omezené a nedovedu si jej v praxi představit.



Obr. 7.1 Nestabilita – poškození kódu

Dalším velkým problémem souvisejícím s výše zmiňovanou „absolutní zapouzdřeností“ je nespolehlivost. Není možné kód formuláře zobrazit, exportovat ani uložit. Manipulovat lze jen a pouze s modulem procedur k formuláři přiložených. Při ukládání objektů v Accessu systém vytvoří nový objekt a nakopíruje do něj data starého formuláře spolu s provedenými změnami. Bohužel tato akce někdy selže a výsledkem jsou buď „visící“ verze starých objektů nebo v horším případě zcela poškozené objekty jednotlivých komponent (obr. 7.2). Dokud je aplikace relativně malá, s podobnými problémy se patrně nikdy nesetkáte, ale pokud se souborem databáze Access pracujete několik let, jeho začne se projevovat nestabilita, kterou bohužel nelze odstranit ani zabudovanými funkcemi pro opravu ani importem objektů do nové databáze – objekty si své chyby přenesou s sebou.

7.3 Použitá technologie

Pro reimplementaci jsem si vybral vývojové prostředí Microsoft Visual Studio 2008, jež máme díky škole k dispozici v licenci MSDN AA. Samotná implementace probíhala v programovacím jazyce C# verze 3.0 běžícím na frameworku .NET 3.5. Jak uvádí encyklopedie Wikipedie: „Framework je softwarová struktura sloužící pro podporu při programování a vývoji a organizaci jiných softwarových projektů.“ Framework .NET zastřešuje systém komponent používajících společné nástroje pro vývoj v jazycích Visual Basic, C++, C# a J#. Možné je použít i další jazyky, ale ty již nepodporuje Microsoft přímo.

Jazyk C# vznikl jako odpověď Microsoftu na vzrůstající oblibu řešení na bázi jazyku Java. Oproti jazyku C++ usnadňuje spoustu bolestivých aspektů programování – jmenujme například práci s pamětí, vlákny nebo řešení rozvrstvení aplikací. Microsoft sám o jazyku C# prohlašuje, že s jednoduchostí Visual Basicu nabízí možnosti C++. Podobně jako Java obsahuje nástroj pro správu paměti.

Za všechny novinky v jazyku C# uvedu alespoň partial class. Je to atribut přidávaný před modifikátor třídy umožňující definici třídy na více místech kódu. Ke sjednocení vlastností dojde automaticky při překladu kódu. Je tak například možné, aby více členů vývojového týmu pracovalo na různých metodách stejné třídy, a přitom synchronizace práce je nutná jen při tvorbě výsledné aplikace.

Jako datový zdroj mi sloužil SQL Server Compact dodávaný s Visual Studiem ve verzi Professional edition. Jeho možnosti jsou sice velmi omezené, ale pro můj účel postačující. SQL Server Compact je aplikace simulující klasický databázový server. Data jsou uložena na disku v jediném souboru podobně jako u aplikace MS Access.

Pro přístup k databázi používám novinku .NET 3.5 systém LINQ. Je to prostředek umožňující pracovat s mnoha zdroji dat jednotně. Pomocí klíčových slov ne nepodobných jazyku SQL nám umožňuje pokročilé dotazování včetně podpory množin a transformací. Výhodou je také integrace

dotazování přímo do programovacího jazyka, což přináší možnost typové kontroly již při překladu aplikace. U datových zdrojů, které dosud nejsou podporovány (například MS do systému LINQ zahrnul podporu pouze vlastního MS SQL serveru), je možné kompatibilitu zajistit implementací rozhraní pro daný datový zdroj. LINQ standardně doposud obsahuje nástroje pro práci s MS SQL serverem (LINQ to SQL), daty ve formátu XML (LINQ to XML), kolekcemi (LINQ to Objects) a datasety ADO.NET (LINQ to Dataset).

Třídám, jež jsou pomocí LINQ to SQL namapovány do databáze, říkáme entitní třídy. Chovají se jako jiné třídy, ale mají již specifikovány některé operace určené pro manipulaci s daty. Vytvořit je můžeme vytvořit buď ručně (ať již prostým psaním kódu nebo pomocí grafického nástroje Object Relational Designer), nebo využít nástroj s názvem SQL Metal - konzolová aplikace pro vygenerování struktury entitních tříd dle existující relační databáze. Protože soubor entitních tříd definuje třídy jako partial (viz výše), dodefinování vlastností k vygenerovanému kódu je poměrně snadné. Nemusíme se vůbec starat o standardní operace v souboru o několika tisících řádků, jen doplníme kód do nějakého jiného souboru (což je obecně doporučováno, abychom si vygenerovaný soubor neúmyslně nepoškodili).

7.4 Požadavky aplikace

Aplikace slouží k evidenci smluv pojištění, leasingů, stavebních spoření a mnoha dalších produktů finančního poradenství. Základním smyslem není nahradit jejich papírové verze, ty musí existovat tak jako tak, ale shrnout informace, jež jsou pro firmu užitečné. Aplikace tedy neeviduje žádné podrobné informace smluv, ale pouze vybrané atributy. Dále obsahuje stručnou evidenci pojišťoven, produktů, obchodních zástupců a jejich provizí. Umožňuje do systému zadat novou smlouvu a podle parametrů smlouvy vyhledávat. Ručně je spouštěna funkce, jež z obsažených dat vypočítá provize obchodním zástupcům.

7.5 Analýza problému

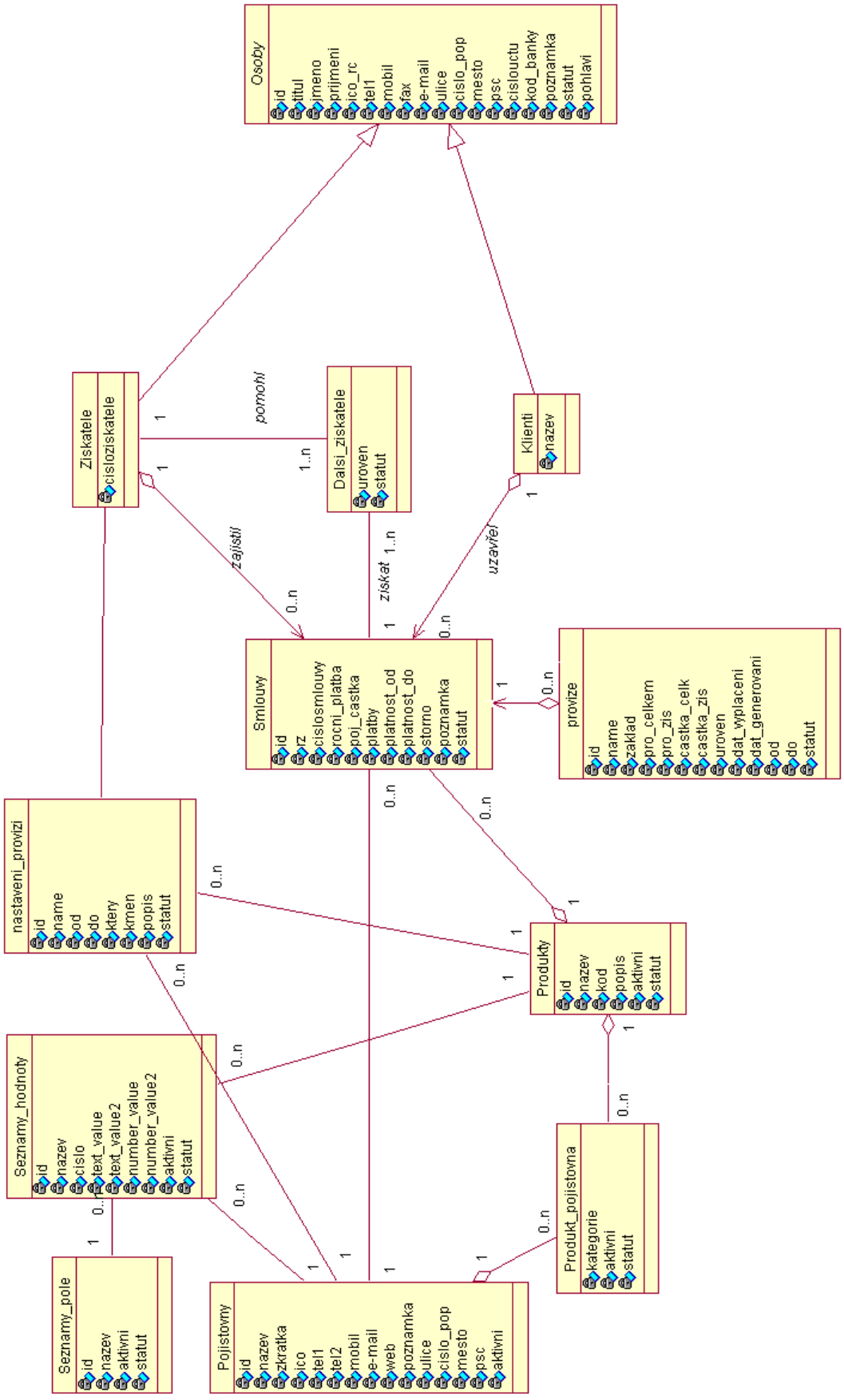
Problémovou doménu aplikace znázorňuje diagram tříd uvedený na obrázku 7.2. Jak napovídá název třídy Pojistovny, její objekty si nesou informace o pojišťovnách, jejichž produkty může firma nabízet. Třída Produkty obsahuje obecně zadané produkty a mezi ní a pojišťovnami zprostředkovává vztah M:N asociační třída Produkt_pojistovna.

Abstraktní třída Osoby deklaruje atributy společné pro odvozené třídy Ziskatele (jiný název pro obchodní zástupce) a Klienti. Ziskatele přidávají pouze parametr cisloziskatele, vlastnost identifikující obchodníka ve firemní struktuře. Jelikož je však zadávána ručně uživatelem, není vhodné ji použít jako klíč (i když by se tak na první pohled mohlo zdát).

Třída Klienti pak přidává pouze atribut nazev – v případě, že klientem je právnická osoba, je v něm uloženo jméno právnické osoby.

Klíčovou třídou, kolem níž se vše točí, je třída Smlouvy. Mezi ní a třídou Ziskatele existuje další asociační třída Dalsi_ziskatele vyjadřující možnost podílu více obchodních zástupců na jedné smlouvě/kontraktu. Poslední z tříd jsou Klienti, jejichž vztah ke smlouvám je 1:N.

Ve všech třídách je přítomen atribut statut. Jelikož se jedná o podnikovou databázi, nastává často případ mylného smazání a tak je nutné smazaná data opět obnovit. Nejefektivnějším způsobem je aplikovat mazání pouze změnou atributu – slouží pro to atribut statut s výchozí hodnotou 1 (hodnota 0 odpovídá stavu smazáno). Stará verze aplikace také obsahovala v každé tabulce pole indikující, jaký uživatel pole vytvořil, naposledy změnil a kdy k oběma akcím došlo. Nyní tento detail zanedbáme.



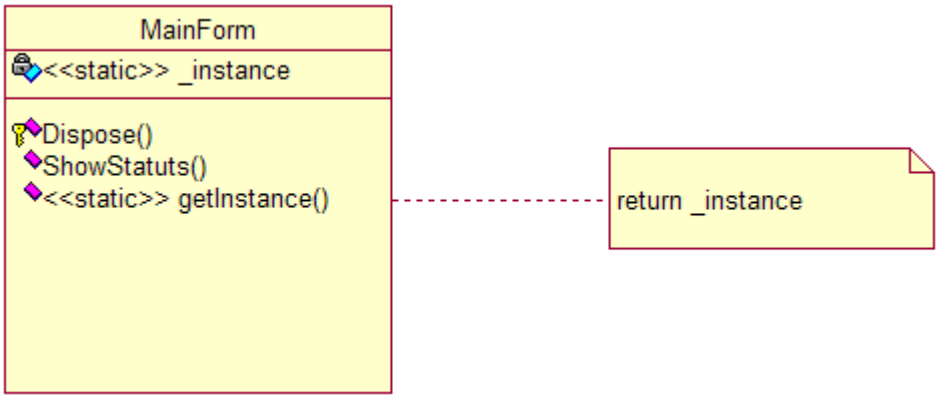
Obr. 7.2 Diagram tříd navržené aplikace

7.6 Použité vzory

V následujících podkapitolách popíši, které vzory jsem navrhl v aplikaci použít.

7.6.1 Jedináček - Singleton

Jak jsem uvedl v kapitole 4.1.1. Jedináček slouží pro implementaci jedinečné instance třídy, jež bude sama zajišťovat neexistenci dalších kopií, ale na druhou stranu poskytne také globální přístupový bod k chráněnému objektu. V případě mé aplikace se pro úlohu jedináčka jevilo vhodné formulářové okno hlavního menu, ale především formulář hledání smlouvy – představě dopomáhá diagram 6.3.



Obr. 7.3 Diagram použití vzoru Jedináček

Hlavní okno aplikace – formulář Program.cs si udržuje statický odkaz na sebe sama:

```
private static MainForm _instance = new MainForm();
```

Tato implementace sice neumožňuje změnu Jedináčka, aby kontroloval jiný počet běžících instancí, než pouze jediný, jak je uvedeno v bodu výhody popisu vzoru, ale vynahrazuje to stabilitou v případě běhu na vícejádrových systémech.

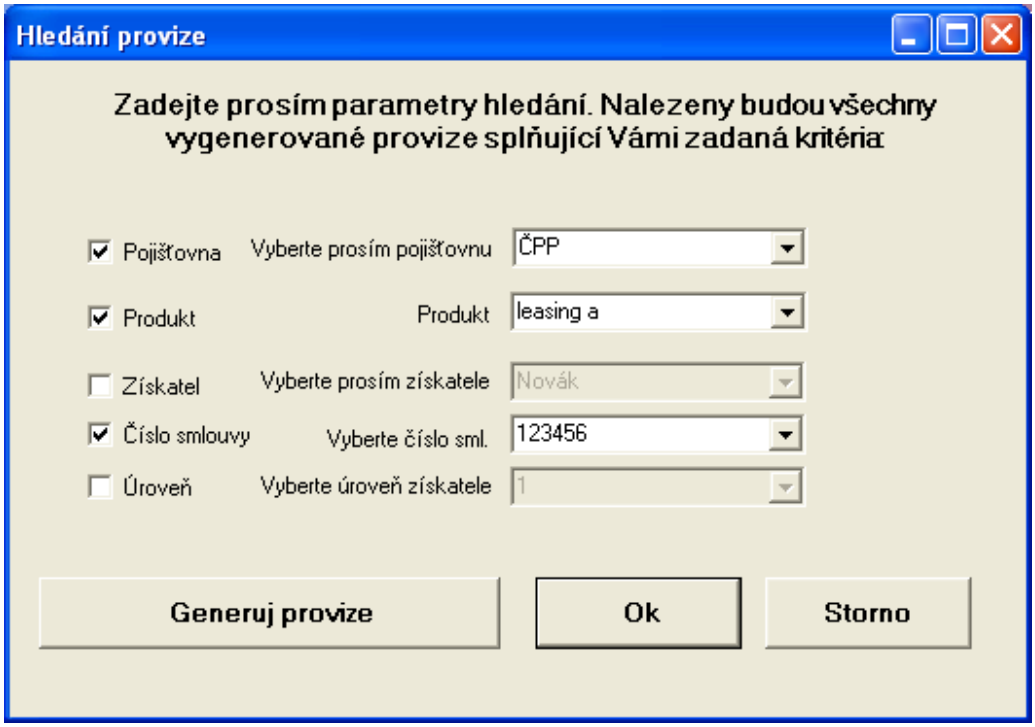
Přístup k instanci i pro další členy aplikace zajišťuje veřejný metoda:

```
public static MainForm Instance
  {
      get
      {
          return _instance;
      }
  }
```

Zmíněná implementace je stabilnější a přehlednější, než mnoho jiných způsobů, vyskytujících se v mnoha popisech návrhových vzorů.

7.6.2 Příkazový objekt - Command

Příkazový objekt slouží k zapouzdření požadavku do objektu a umožňuje měnit parametrizaci klientů. Klienti pouze definují požadavky na zpracování a mohou určovat i zpracovatele, jež žádosti provede. Způsob resp. algoritmus vykonávání je však od klienta odstíněn.



Obr. 7.4 Okno vyhledávání provizí dle zadaných atributů

Vzor je s výhodou používán ke spouštění akcí, jež je možné spustit více způsoby. Dnes je v grafickém uživatelském rozhraní zcela běžné, že mnoho příkazů můžeme provádět mnoha různými způsoby. K dispozici jich máme celou sadu – menu programu, klávesové zkratky, tlačítka, ikony a kontextová menu aplikace. Zpracování daného příkazu by ale nemělo mít nic společného s těmito ovládacími prvky a tak je vhodné použit návrhový vzor Příkazový objekt.

Můj návrh jej využívá v případě požadavku hledání smlouvy nebo hledání provize a spouštění generování provizí. Při této akci je nutné sestavit relativně komplikovaný dotaz a najít hledaný záznam podle mnoha různorodých parametrů. Ve třídě Command.cs definuji abstraktní třídu Command:

```
abstract class Command
    {
        //protected Receiver receiver;
               public Command(/*Receiver receiver*/)
        {
            /*this.receiver = receiver;*/
        }
        public abstract void Execute();
    }
```

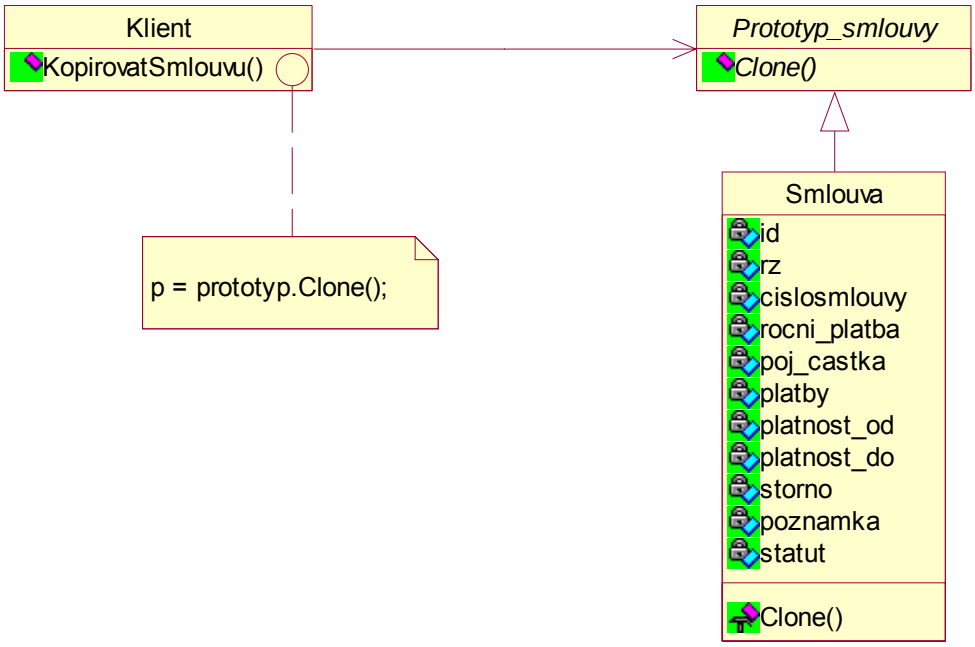
V zakomentovaném kódu vidíte, na jakém místě by byla implementace případné volby příjemce. Na formuláři hledání (obr. 7.4) tlačítko „OK“ spustí akci, jež vytvoří novou instanci spouštěče (neboli instanci třídy `Invoker`) a příkazového objektu:

```
this.command = new DIP1.Com_hledani_sml(getPojistovna(),getProdukt(),
getSmlouva());
this.invoker = new DIP1.Invoker();
invoker.SetCommand(command);
invoker.ExecuteCommand();
```

Poslední řádek kódu spustí akci samotného příkazového objektu.

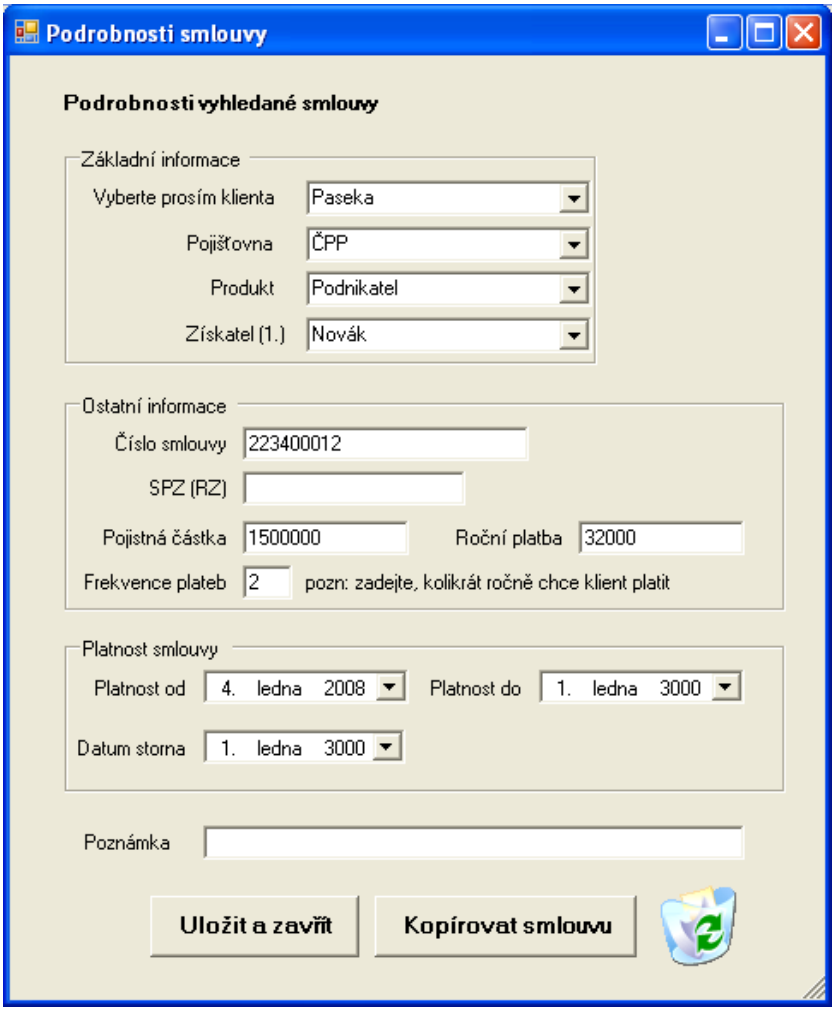
7.6.3 Prototyp - Prototype

Prototyp patří do skupiny tvořivých vzorů a jeho účelem je vytváření objektů kopírováním určitého prototypu a to dokonce bez znalosti třídy tohoto objektu. Druh vytvářeného objektu je specifikován prototypickou instancí.



Obr. 7.5 Diagram použití vzoru Prototyp

Tento vzor jsem navrhl využít pro akci kopírování nastavení. V případě formuláře se smlouvami (obr. 7.6) je po jejich uložení možné smlouvy kopírovat do smlouvy nové, aby uživatel nemusel většinu položek vyplňovat znovu (diagram 7.5). Při zadávání dat do databáze má obsluhující většinou smlouvy seřazeny podle produktu, pojišťovny, případně dalších kritérií a tak může s výhodou využít funkce kopírování smlouvy.



Obr. 7.6 Formulář s informacemi o podrobnosti smlouvy

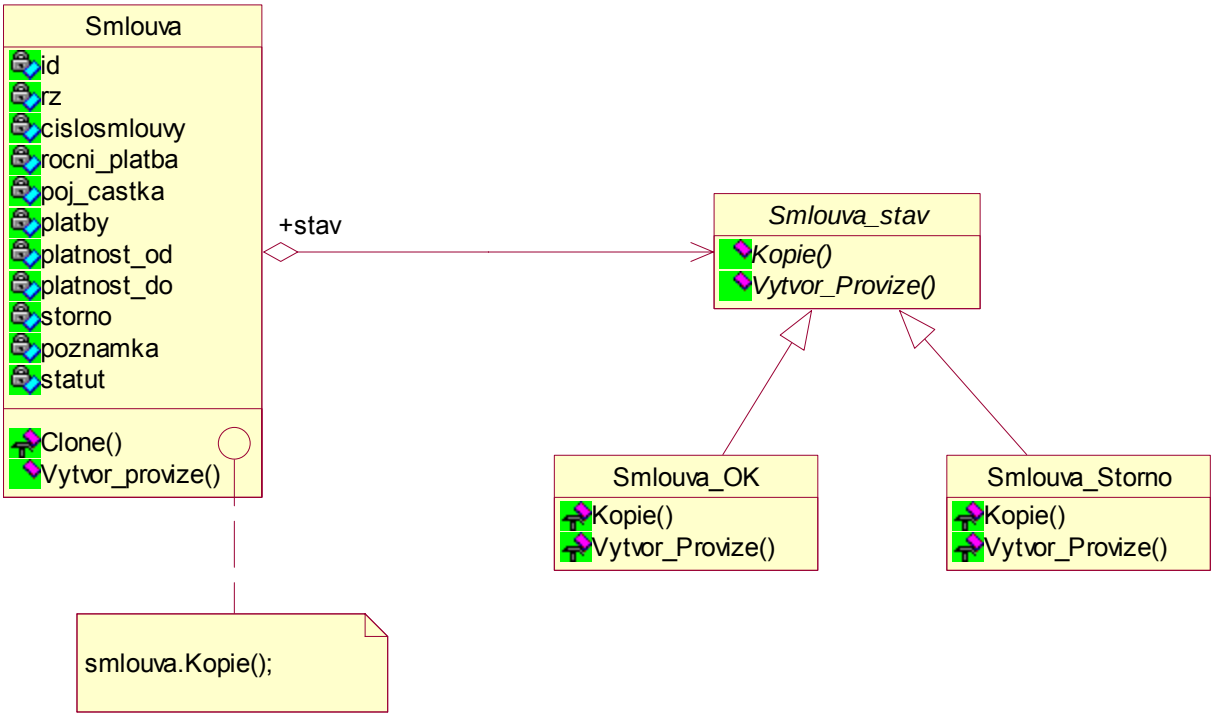
Další použití tohoto vzoru jsem našel v případě nastavení provizí obchodních zástupců. Protože toto nastavení je obzvláště pracné, resp. náročné na pozornost uživatele (v případě chyby je obchodnímu zástupci vypočítána špatná provize a dostane zaplacenou jinou částku, než by měl dostat). Podobně jako v případě smlouvy i při kopírování nastavení provizí získatelů je cílem vytvoření kopie původního objektu.

Cílem použití vzoru prototyp je obvykle zjednodušení vytváření objektu. Co se týče schopnosti klienta spustit tvorbu kopie, aniž by klient znal přesnou třídu objektu, vychází z vlastnosti polymorfismu a není až tak překvapivá. Zajímavější je pak použití k redukci režie. Můj příklad používá variantu deep copy – předaný objekt (prototyp) tedy serializuje, zapíše do paměti a posléze znovu načte z paměti a vrátí tak nový objekt. V případě zadávání kopie smlouvy je ušetřena komunikace se serverem, což je obzvláště zajímavé, pokud je konektivita nedostačující. Dochází k ní až při ukládání upraveného objektu do databáze tlačítkem „Uložit“, zatímco jinak by klient zasílal serveru nejprve dotaz na přístupnost sekvence pro vytvoření primárního klíče, následovalo by samotné vytvoření záznamu a až poté by byl záznam poslán klientovi v nějakém druhu recordsetu.

Vzhledem k všeobecné potřebě škálovatelnosti systémů mi použití prototypu připadá jako vhodné řešení.

7.6.4 Stav - State

Návrhový vzor state je členem skupiny vzorů chování, jak je popisuje kapitola 4.3. Pokud nastane definovaná změna stavu (atributů) objektu, mění jeho chování, jako by se stával objektem jiné třídy.



Obr. 7.7 Diagram použití vzoru Stav

Tento vzor bývá používán, jestliže je třeba chování objektu změnit při určitém stavu takovým způsobem, že bychom v kódu museli aplikovat enormní množství podmíněných příkazů rozhodující stejný problém.

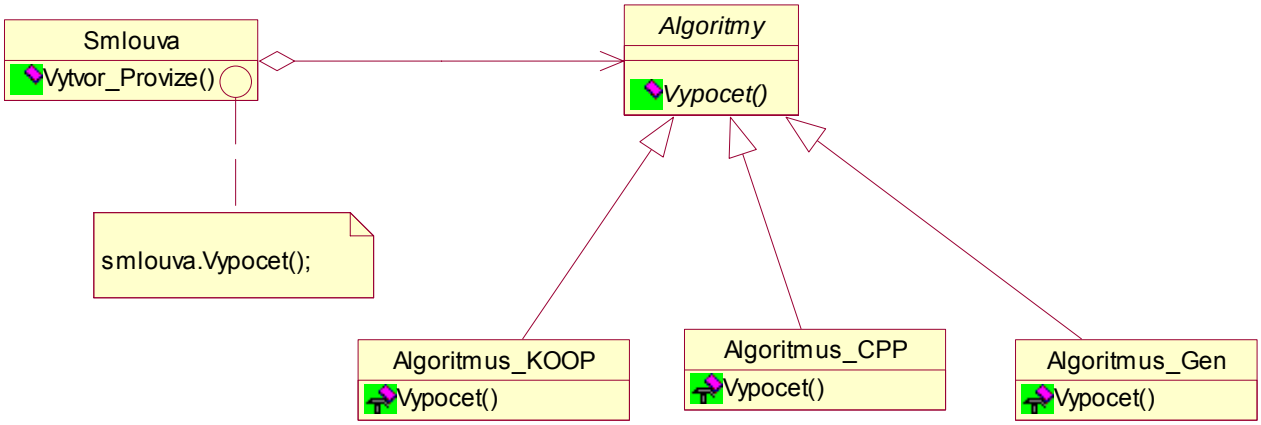
V aplikaci databáze smluv tato situace nastane například tehdy, když dojde k ukončení smlouvy před datem její konečné platnosti - stornování smlouvy (diagram 7.7). Místo toho, aby uživateli byly zakazovány operace se smlouvou formou mnoha podmíněných příkazů, je vhodnější využít návrhový vzor state.

Objekt třídy Smlouva si udržuje instanci stavového objektu reprezentující aktuální stav smlouvy. Všechny zádosti o akce Clone() případně Vytvor_provizi() deleguje na podtřídy Smlouva_OK a Smlouva_Storno udržované instance abstraktní třídy Smlouva_stav. Ty již samy implementují operace.

Toto řešení se jeví výhodné v případě velkého a složitého větvení kódu, na druhé straně násobí množství tříd a tak je nutné zvážit pozitiva i negativa jeho implementace.

7.6.5 Strategie - Strategy

Tento vzor patří také mezi vzory popisující vzájemné chování objektů. Umožňuje zapouzdřit sadu algoritmů použitých například pro nějaký výpočet a to bez závislosti na klientech používajících tyto algoritmy. K jejímu spouštění používám vzor příkazový objekt.



Obr. 7.8 Diagram použití vzoru Strategie

Jeden z procesů spouštěných v navrhované aplikaci je algoritmus výpočtu provizí (resp. výplaty obchodních zástupců z jednotlivých kontraktů – diagram 7.8). Ten je ve své plné podobě mírně řečeno velmi složitý a jeví se tak jako ideální příklad použití návrhového vzoru strategie. Každá pojišťovna používá jiný systém výpočtu provizí. I v případech, kdy se nám na první pohled může zdát, že by výpočet mohl probíhat obdobně, zjistíme, že opak je pravdou.

8 Důsledky použití vzorů

8.1 Přínosy implementace

Kapitola 7.2 osvětlovala hlavní problematické části předchozí implementace. Ty by pravděpodobně byly odstraněny už jen implementací v jiném jazyce/nástroji i za předpokladu procedurálního přístupu.

Nyní se pokusím vysvětlit, jaký přínos mělo použití vzorů v databázové aplikaci evidence smluv. Z kapitoly sedm je patrné, že jsem využíval především vzory z kategorie vzorů chování. Je to dáno především tím, že aplikace nevyžaduje složitou strukturu tříd pro uložení samotných dat. Nemodeloval jsem grafický editor, editor hudebních partitur ani jinou podobnou aplikaci, na které se obvykle velmi názorně (i když opět často účelově) demonstrují výhody tvořivých vzorů. Jedná se o databázovou aplikaci, a tak například vytváření tříd představujících i konkrétní řady produktů nemá žádný smysl, protože jejich množství je enormně vysoké (v mnoha pramenech o vzorech jsou však takovéto příklady uváděny). Využití návrhových vzorů jsem hledal především v oblasti řízení procesů probíhajících v aplikaci.

Návrhový vzor Singleton zajišťuje jedinečnost prvků, jako například nástroje vyhledávání, nebo také hlavního okna aplikace. Ve srovnání s řešením pomocí globálních proměnných je kontrola jedinečnosti zapouzdřena do třídy MainForm. V původním projektu byla tato vlastnost definována přímo vlastností formuláře aplikace Access. Formulář je objektovou částí aplikace Microsoft Access, ale není přístupný, takže nemohu porovnat, jakým způsobem byla tato funkce implementována původně. Každopádně použití tohoto vzoru je jednoduché, účelné.

Více návrhových vzorů se pak týká třídy smlouva. Vzor Command účinně oddělil vykonávání kódu aplikační logiky v prezentační vrstvě převedením kódu akce na žádost zapouzdřenou v objektu. Vzor Prototyp pomáhá snižovat síťovou zátěž vytvářením kopírovaného objektu z existujícího.

V komplexněji navrženém systému by bylo pravděpodobně i místo pro vzor Mediátor – prostředník. Aplikace má ke svému fungování ve firmě k dispozici zvláštní modul pro import dat z různých zdrojů (převážně pojišťoven). To by byl vhodný námět pro využití vzoru Adaptér.

Zcela jednoznačně se zvýšila přehlednost a rozšiřitelnost kódu oproti původní implementaci. Jako mnoho kvalitativních měřítek je kvalitní návrh veličina, jež není dost dobře možné změřit.
Metriky pro porovnávání objektově-orientovaných systémů nemohu vztáhnout na původní systém – neobsahuje žádné objekty. Metriky pro poměřování kvality procedurálně programovaných systémů pro změnu vyznívají výrazně ve prospěch starého řešení, protože prostředí MS Access je silně optimalizováno pro jednoduchou práci s databázemi a vzhledem k přítomnosti jazyka Visual Basic (ve verzi for Applications) je kód krátký a jednoduchý. Přesto však v kapitole devět vybrané použitelné metriky aplikuji pro získání měřitelných dat.

Neoddiskutovatelně se ovšem zlepšila struktura kódu a možnosti re-use. Příklad, na kterém je to patrné je zadávání nové smlouvy do systému. V předešlé verzi byla potřeba existence dvou formulářů pro jeden účel – přidávání smlouvy a zobrazení podrobností o smlouvě. Nyní stejnou práci vykoná jediný formulář. Počet objektů v aplikaci tak zbytečně neroste.

8.2 Obecný pohled na použití návrhových vzorů

Při studiu problematiky návrhových vzorů jsem došel k názoru, že jejich využití je nutné obezřetně hlídat. Mnohdy totiž není náš problém tak složitý, abychom ho museli řešit návrhovými vzory a jejich použití může aplikaci jen zkomplikovat. Například tabulka 8.1 uvádí délky kódu v bytech při použití různých druhů implementace návrhového vzoru Command.

Typ programu	**Velikost kódu v Bytech**
Bez vzoru	1719
Pojmenované inner clases	4450
Nepojmenované inner clases	3683
Externí třídy	3838

Tabulka 8.1 Porovnání délky kódu různých implementací vzoru Command [14]

Jak je z tabulky patrné, použití tohoto vzoru v případně řešení jednoduchého problému s několika málo podmíněnými příkazy výsledný kód prodlouží. Odměnou nám ale je oddělení řešení aplikační logiky od ovládacího rozhraní a ta je tím patrnější, čím více ovládacích prvků aplikace obsahuje.

Je třeba zvažovat, zda nám vzor více přinese, než ubere. Příkladem budiž vzor Abstract Factory, jež je ve většině informačních zdrojů zmiňujících existenci návrhových vzorů pokládán za typický příklad. Využití autoři vidí například v přepínání různých grafických režimů za běhu aplikace nebo například v schopnosti aplikace přizpůsobit se různým databázovým serverům. První případ je jistě zcela správný, ale o druhém mne například kniha [9] donutila silně pochybovat. Je totiž nutné naprogramovat celou továrnu pro každý server, který chceme podporovat. Ale přepínání mezi různými servery není, na rozdíl od přepínání režimu grafického zobrazení, každodenní záležitostí. To, na jakém databázovém serveru bude postavena databáze firmy, je dle mého názoru, záležitostí dlouhodobého strategického rozhodnutí firmy – prostě servery neměníme jako ponožky!

Vzhledem k faktu, že každý server funguje vnitřně trochu jinak a rozdíly nejsou jen v syntaxi SQL, ale i dalším chování serveru, čas strávený na programování kompatibility s jiným serverem, jen pro případ že to možná jednou nebude špatná funkce, je ve skutečnosti promrhaný. Podobné tendence

ukazuje i obecně známý fakt, že 60% času programátora je promrháno prací na složitých funkcích, jež jsou ve skutečnosti využity zcela minimálně, a jen 40% je věnováno programování nejpoužívanějších částí systému – ověřeno vlastní zkušeností.

Místo toho by bylo daleko užitečnější v budovaném systému pozornost věnovat čemukoliv jinému. Samozřejmě existuje určitá malá skupina aplikací, které se snaží podporovat všechny existující standardy, aby spektrum potenciálních zákazníků bylo co nejširší. Ale frekvence, s jakou jsem se setkal v popisech vzorů s tímto využitím, mi vnucuje myšlenku, že se jedná o typický příklad aplikace vzoru jen proto, abychom mohli říci, že náš systém využívá myšlenky návrhových vzorů.

Dále musím podotknout, že použití některých vzorů již není tak aktuální, jako tomu bylo v minulosti. Návrhové vzory GoF byly sepsány v roce 1994 a to je třeba mít na mysli při jejich aplikaci. Například vzor Iterator je nahrazen zlepšenými schopnostmi vývojových nástrojů resp. prostředí. Tento vzor sice umožňoval definovat více možností procházení kolekcemi dat, než jen jednoduché operace typu první prvek, další prvek a podobně, ale v nejnovějších prostředcích jsou i tyto možnosti součástí jazyka (obsahuje je například systém LINQ zmiňovaný v kapitole 7.2).

Nicméně s ohledem na stáří původních návrhových vzorů a navzdory předchozímu odstavci je s podivem, nakolik jsou jimi řešené problémy stále aktuální. Využívá je rostoucí počet aplikací a jejich úloha při vývoji bezesporu poroste. Navíc postupně vznikají vzory nové reagující na aktuální potřeby a problémy.

9 Softwarové metriky a jejich aplikace

Výraznou nevýhodou původního řešení - tedy velká duplikace kódu byla do značné části vynahrazena tím, že kód byl jednodušší. Kvality nově navrženého systému jsou však výraznější s přibývající velikostí projektu. Protože kód formulářů byl s nimi napevno svázán, mnoho se jej opakovalo. Vezmeme-li v úvahu metriku LOC (Lines-of-code), vzniká například problém, které řádky bychom měli započítat (kód formulářů Accessu je z valné většiny skrytý a nelze jej tedy započítat kompletně). Množství kódu nového systému je tedy výrazně vyšší, čím méně funkčních částí je implementováno

Nyní aplikujeme metriku LOC na původní a nově implementovaný projekt. Porovnávat budu vybrané srovnatelné části s podobnou funkcionalitou.

Existuje několik variant metriky LOC [18]:

- Součet řádků jako fyzických řádků na výstupu obrazovky
- Součet řádků jako řádků ukončených fyzickými oddělovači
- Součet skutečně proveditelných řádků
- Součet proveditelných řádků s definicí datových typů
- Součet proveditelných řádků s definicí datových typů a komentáři

Jako vhodnou jsem vybral poslední verzi, protože komentáře obvykle píši na stejný řádek jako komentovaný kód (výsledek by měl být tedy víceméně shodný s předposlední variantou).

	Projekt Access	Projekt C#
Grafická část	189	178
Část vyhledávací	405	257
Celkem	594	435

Tab. 9.1 Akce/formulář vyhledávání smlouvy

	Projekt Access	Projekt C#
Grafická část	89	66
Část vyhledávací	101	61
Celkem	190	127

Tab. 9.2 Akce/formulář vyhledané smlouvy

	Projekt Access	Projekt C#
Grafická část	289	308
Část vyhledávací	136	408
Celkem	425	716

Tab. 9.3 Akce/formulář Detail smlouvy

	Projekt Access	Projekt C#
Grafická část	103	328
Část vyhledávací	261	433
Celkem	364	761

Tab. 9.4 Akce/formulář vyhledávání provizí

U obou prostředí jsem nezapočítával řádky generované automaticky do dalších souborů. V případě nové aplikace se jedná o součet řádků potřebných k vykonání potřebné operace. Tedy postupně funkci formuláře vyhledávání smlouvy, zobrazení vyhledaných záznamů, zobrazení detailu z vyhledaných záznamů a nakonec formuláře obsahující detailní informace o smlouvách.

V případě aplikace Access je přístupná pouze část kódu, zejména tedy grafická část by ukazovala výrazně vyšší čísla, kdybychom se mohli na kód podívat. I tak, ale není rozdíl ve prospěch MS Access tak pozitivní, jak bychom mohli očekávat a je znát, že pro větší nasazení se určitě vyplatí přejít na novější technologie.

10 Další vývoj

Návrhové vzory a další typy vzorů představují značný pokrok ve vývoji softwarového produktu.

Pokud se podíváme do analogické domény – například hudebního průmyslu, uvidíme však úspěšnost použití vzorů výrazně větší. Nástroji magnetické rezonance a pozitronové emisní tomografie se podařilo analyzovat reakce mozků lidí na jednotlivé typy, artefakty, části a tóny hudby takovým způsobem, že byly sestaveny relativně přesné matematické vzorce produkující hudbu, jež se většině masové populace líbí. Současná produkce populární hudby je toho přímým důkazem. Jediným rozdílem mezi hudebními a softwarovými vzory tedy zůstává primární cíl – v hudební oblasti je to zvýšení zisku bez ohledu na kvalitu, zatímco v informačních technologiích je kvalita produktu důležitou prioritou.

Přesto se objevují mnohé pokusy, jak vývoj software automatizovat. Jedním z nich je například MDA (Model Driven Architecture). Důvod, proč MDA prozatím nevypadá jako cestou k cíli, je pravděpodobně jeho přílišná obecnost. Nyní však jeden z hlavních tahounů vývoje - Microsoft představil koncept softwarových továren. Označil současný specifikační jazyk UML jako nedostačující (nepřesný, nekonkrétní) a doporučil jej k použití zejména pro: skicování, malování na tabuli, dokumentaci a konceptuální schémata, která mají přímý vztah ke kódu.

Vize vypadá tak, že softwarové továrny budou o znovupoužívání vzorů, frameworků, nástrojů, šablon, požadavků, testů, instrumentací, procesních návodů a mnoha dalších aktiv používaných v životním cyklu softwarového vývoje, v systematickém spíše než v ad-hoc smyslu, a nikoliv o vytváření bezpočtu kopií stále té samé věci.

Z dnešního pohledu nejsou tedy návrhové vzory cílem, ale pouze milníkem na cestě k co nejvyšší automatizaci programování.

11 Závěr

Aby v dnešní náročné době firma podnikající na trhu informačních technologií mohla uspět, je třeba dohlédnout na využívání efektivních vývojových metod. K těm jistě patří objektově orientované programování. Nejen začátečníci v oblasti objektově orientovaného programování se však potýkají s problémem rozvržení tříd a objektů a správného návrhu systému. Pod tíhou problému pak často zanevřou na celé objektově orientované programování a dají přednost klasickým procedurálním nástrojům, na něž byli celou dobu zvyklí.

Návrhové vzory se nezabývají celou problematikou vývoje a návrhu systému, nýbrž usnadňují designovou část návrhu. Snaží se na základě už známých faktů a analyzovaných problémů uspořit programátorovi čas i prostředky a zvyšují kvalitu návrhu vyvíjených systémů.

V této práci jsem se snažil seskupit všechny znalosti, jež jsou potřeba pro pochopení principu fungování návrhových vzorů a jejich využití na praktickém příkladě. Postupoval jsem tak, aby se čtenář nejenom naučil používat konkrétní návrhové vzory, ale také jejich společné pozitivní vlastnosti a mohl je využít při budování svého systému.

Jejich aplikace na projekt databázové aplikace finanční instituce přinesla výrazné zlepšení vlastností a to jak po stránce stability systému, tak nesrovnatelně větší znuvupoužitelnosti a kvalitnější struktury kódu.

Pokud se podíváme na možnosti dalšího pokračování tohoto projektu, tak je zde mnoho prostoru ke zvyšování funkcionality navrhované aplikace vedoucí k nově vzniklým problémům a tedy i novému prostoru pro využití návrhových vzorů.

Literatura

[1] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: *Návrh programů pomocí vzorů. Stavební kameny objektově orientovaných programů.* Praha, Grada 2003.

[2] Dvořák, M.: *Návrhové vzory (Design patterns)*, [diplomová práce]. Dokument dostupný na URL: http://objekty.vse.cz/Objekty/Vzory-uvod (leden 2008).

[3] Internetová encyklopedie Wikipedia, Článek *Objektově orientované programování*. Dokument dostupný na URL: http://encyklopedie.seznam.cz/heslo/141232-objektove-orientovane-programovani (2.1. 2008).

[4] Christopher Alexander: *Notes on the Synthesis of Form*. Harvard University Press, 1964. Kniha dostupná v aplikaci Google Book Search (14. 5. 2008).

[5] Kraval, Ilja: *Úvodní pojmy objektového programování*. Vydáno 1999. Dokument dostupný na URL http://www.objects.cz po kontaktování autora (3. 1. 2008).

[6] Duell, Michael: *Non-Software Examples of Software Design Patterns*. Vydáno 2002. Dokument dostupný na URL http://www2.ing.puc.cl/~jnavon/IIC2142/patexamples.htm (13. 5. 2008).

[7] Kraval, Ilja: *COM a DCOM v praxi*. Vydáno 2000. Dokument dostupný na URL http://www.objects.cz po kontaktování autora (3. 1. 2008).

[8] OO, UML, analýza, metodologie, Článek *Class diagram - diagram tříd*. Dokument dostupný na URL: http://mpavus.wz.cz/uml/uml-s-class-3-3-1.php (11. 5. 2008).

[9] Kerievsky, Joshua: *Refactoring to Patterns*. Courier Westford in Westford, Massachusetts, Pearson Education, 2007, 5. vydání.

[10] Bernard, B.: *Vývojové prostředí pro návrhové vzory v C#*, [diplomová práce]. Vysoká škola ekonomická v Praze, nám W. Churchilla 4, 130 67, Praha 3.

[11] Hailpern, B.: Design Patterns' Receives ACM SIGPLAN Award. Dokument dostupný na URL http://science.slashdot.org/article.pl?sid=05/07/31/0046223 (16. 5. 2008).

[12] Bacon, D. aj.: *The „Double-Checked Locking is Broken" Declaration*. Dokument dostupný na URL: http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html (17. 5. 2008).

[13] Skeet, J.: *Implementing the Singleton Pattern in C#*. Dokument dostupný na URL: http://www.yoda.arachsys.com/csharp/singleton.html (17. 5. 2008).

[14] Cooper, J.W.: *D*esign patterns. *Java Companion*. Vydáno 1998.

[15] Vlissides, John: *Pattern Hatching – Design Patterns Applied*. Reading, Massachusetts 01867, Addison Wesley Longman, Inc. 1998. První vydání.

[16] Matějka, Petr: *Analýza a praktická implementace softwarových metrik pro oblast adaptability SW produktu*. Bratislava. 2005. Dokument dostupný na URL: http://www.dcs.fmph.uniba.sk/diplomovky/obhajene/getfile.php/diplomovka.pdf?id=76&fid=132&type=application%2Fpdf (27. 7. 2008).

[17] Vosátka, Karel: *Studijní podklady pro předmět SI2: Řízení softwarových projektů*, kapitola Metriky softwarového projektu. FEL ČVUT Praha. Dokument dostupný na URL: http://cs.felk.cvut.cz/~richta/www-si2/WS3.HTML (27. 7. 2008).

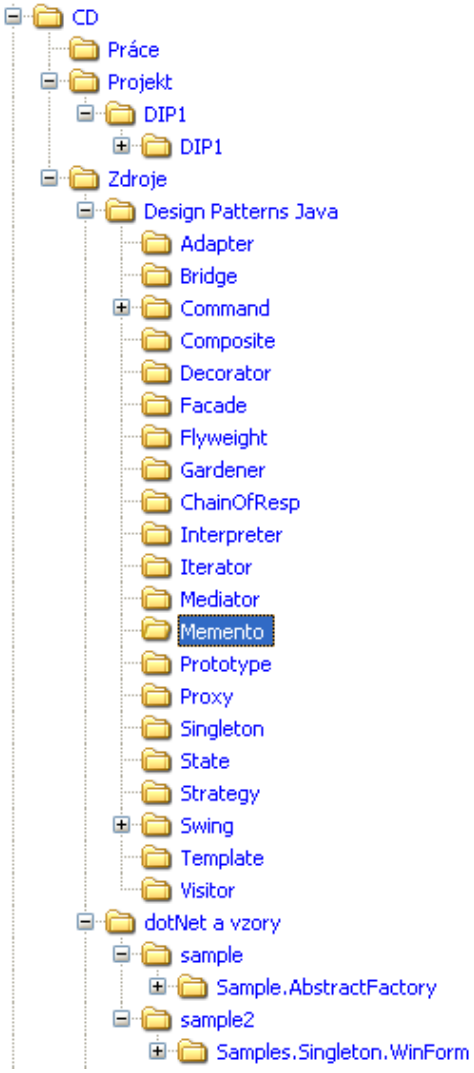
[18] Kan S. H.: *Metrics and Models in Software Quality Engineering*. Reading, Massachusetts 01867, Addison Wesley Longman, Inc. 1995.

Příloha 1 – obsah přiloženého CD

Adresář práce obsahuje soubory s textem diplomové práce ve formátu Microsoft Word 1997-2003 a PDF. V projektu se skrývají soubory potřebné pro spuštění a překlad programované práce. K uspokojivému běhu je vyžadováno prostředí MS Visual Studio 2008 se serverem MS SQL Compact.

Adresář „Zdroje“ pak obsahuje zdrojové soubory, které mi napomohly k řešení projektu. V souboru BPMN_workflow_patterns.pdf můžete nalézt materiály k přednášce Kamila Svobody na téma workflow patterns a materiály Rudolfa Pecinovského k problematice efektivní výuky OOP.

Ve složce „Design Patterns Java“ vám může pomoci elektronická publikace rozebírající použití návrhových vzorů v Jave i s příklady zdrojových kódů k jednotlivým vzorům. Nakonec adresář „dotNet a vzory“ obsahuje dva příklady použití návrhových vzorů v jazyku C# (konkrétně mnou kritizované použití Abstract faktory k rozlišení připojovaných serverů a využití Jedináčka)



Obr. 12. 1 Struktura souborů na přiložené CD nosiči