Design Patterns v OOP

se zaměřením na JAVU, C# a Delphi

Motto knihy: Tvořit objektově kvalitní orientovaný návrh není vůbec jednoduché, tím spíše aby byl tento návrh opětovně použitelný a flexibilní.

© **Ilja Kraval**, vydáno září roku 2002,

kontakt: http://www.objects.cz/ , mailto:objects@objects.cz

Tato e-kniha byla vytvořena pomocí nástrojů WORD a VISIO od firmy Microsoft.

Úvod

Úvodní slovo autora

Dostává se vám do rukou další e-kniha z oblasti OOP od autora Ilji Kravala. Toto dílo navazuje na předešlé knihy ze *Serveru objektových technologií* (viz http://www.objects.cz).

Kniha je věnována u nás poměrně dost zanedbávané oblasti v tvorbě SW zvané *Design Patterns* (překládá se standardně jako *návrhové vzory*). Je třeba podotknout, že v zahraničí je naopak této problematice věnována poměrně dost velká pozornost, a to hlavně v komerční oblasti výroby SW.

Poznámka: O rozdílném zájmu o tuto oblast u nás a v zahraničí svědčí například skutečnost, že o in-house školení Design Patterns, které poskytujeme, projevily zájem v mnohem větší míře pobočky zahraničních SW firem působících v ČR než původní české firmy.

Důvod zájmu o Design Patterns je velmi prostý: Použití Design Patterns vede k velmi silnému snížení nákladů ve firmě a výrazně redukuje pracnost řešení. Je to hlavně díky dalšímu možnému způsobu využití re-use, tj. využití opětovné použitelnosti. Řekněme, že se jedná o ekonomický faktor využití Design Patterns, který je velmi důležitý z hlediska snížení nákladů ve firmě.

Existuje ale i druhé hledisko výhody použití Design Patterns, než je hledisko firemní. Jedná se o hledisko čistě osobní, tj. otázka vašeho osobního studia. Mohu z vlastní zkušenosti potvrdit, že studium Design Patterns přináší pro jednotlivce dva velmi pozitivní důsledky:

Za prvé čtenáři poskytuje ucelený náhled na problematiku Design Patterns v OOP a seznámení se s již obecně zavedenými opětovně použitelnými vzory v OOP. Dá se říci, že se díky této knize naučíte určitým „obecně používaným postupům“ v návrhu v OOP, a tak se rozšíří váš obzor v použitelnosti objektově orientovaného návrhu. Důsledkem toho je, že se zvýší vaše kvalifikace jako tvůrce návrhu v OOP. Díky této knize budete seznámeni s katalogem všech známých používaných vzorů, což je devíza vhodná pro vaše další uplatnění. To je však pouze jedna strana mince.

Na druhé straně existuje ještě druhé hledisko, nazvěme jej „osobní hledisko“, které považuji za stejně důležité. Studium Design Patterns vám může poskytnout také hlubší a mnohdy až překvapivé proniknutí do OOP. Mohu potvrdit, že studium Design Patterns bylo pro mne opravdovým zážitkem. Na jeho základě se mi ještě více ukázalo, co vše je v OOP možné, jaké opravdu logicky elegantní konstrukce lze v OOP vytvořit. Jinak řečeno, studium jednotlivých vzorů v Design Patterns je také dobrým proniknutím do tajů OOP. Pro toho, kdo si chce prohloubit chápání OOP, se Design Patterns stává oblastí velmi vypovídající, navíc poučnou, a tím přínosnou.

Také v tomto směru je kniha zaměřena velmi prakticky. Snahou autora bylo nejenom vyjmenovat všechny možné známé vzory s jejich možným použitím, ale také vysvětlit základní myšlenky těchto vzorů, vysvětlit podstatu těchto vzorů, jejich nosnou ideu, a to i s vysvětlujícími odbočujícími úvahami. Proto jsou pro různá vysvětlení velmi často používána jednoduchá a názorná přirovnání. Autor se také z toho důvodu snaží důsledně rozebrat všechny možné myšlenky související s tímto vzorem. Cílem je poskytnout čtenáři nejenom všeobecný přehled o vzorech jako jejich výčet, ale také umožnit proniknutí do podstaty vzoru při plném pochopení jeho principů. Výsledkem této snahy je mimo jiné i to, že čtenář bude schopen při hlubším chápání vzoru nejenom vzor použít (tj. převzít jej jako šablonu řešení), ale také jej případně modifikovat, rozvíjet, případně tvořit další možné klony vzorů. Tím může rozšířit použití vzorů a obohatit katalogy vzorů o další nové vzory. Takovýto přístup však bez celkového pochopení podstaty vzoru není možný.

Nutno podotknout, že v angličtině existuje několik velmi dobrých publikací (přehled všech publikací viz konec knihy). Některé z nich jsou velmi kvalitní a většinou podávají ucelený a vyčerpávající přehled přes problematiku Design Patterns. Avšak mnohdy je studium těchto publikací díky jejich hutnosti a díky množství informací na malém prostoru bez velkých vysvětlování opravdu poměrně dost obtížné a

někdy zabírají dost času. Například studium celé problematiky Design Patterns na takovou úroveň, abych byl schopen přednášet tuto problematiku v rámci dvoudenního školení (v rozsahu všech 23 vzorů), mi trvalo podle těchto zdrojů okolo 6 měsíců. Rád bych předložil čtenáři knihu, která by nejenom tuto dobu studia zkrátila, ale také napomohla vzory lépe pochopit a poznat je do hloubky, a to nejenom povrchně.

Na konec úvodu mi nezbývá než jediné: Popřát čtenáři, aby při čtení této knihy poznal stejné pocity okouzlení nad elegancí a krásou logiky, jako jsem měl možnost poznat i já. Doufám, že při čtení této knihy získáte stejně pěkný zážitek, jako když jsem s elegantními konstrukcemi v Design Patterns seznamoval poprvé.

Komu je určena tato kniha

Tato kniha je určena především všem pracovníkům činným v oblasti tvorby SW, tj. analytikům, vedoucím projektů, programátorům, testerům atd., ale i uživatelům SW v podnicích, které spolupracují s firmami tvořící software objektově orientovaným přístupem.

Zvládnutí této knihy podá čtenáři ucelený obraz o použití Design Patterns v OOP. Kniha je navíc orientovaná na důkladné a názorné vysvětlení jak celé problematiky Design Patterns, tak základních myšlenek OOP a přístupu k řešení v této oblasti. Z toho důvodu se stává dost dobrou učebnicí nejenom pro danou oblast, ale napomáhá také zvládnout poměrně dost obtížnou problematiku OOP.

Nutné znalosti čtenáře

K dobrému pochopení knihy je třeba, aby čtenář měl určité znalosti. Jejich seznam lze shrnout do následujících bodů:

- základní znalosti OOP
- základní znalosti UML (Unified Modeling Language), zejména modelu tříd a sekvenčního modelu
- základní znalosti z problematiky fázování vývoje tvorby SW

Poznámka: Všechny tyto znalosti lze získat studiem knih vydaných na serveru http://www.objects.cz

Pro tuto knihu jsou některé ze znalostí z OOP obzvlášť důležité a proto se jim musíme hned v úvodu blíže věnovat.

O čem pojednává Design Patterns

Design Patterns a úrovně abstrakce

Jak známo, na vyvíjený software lze nazírat z různých úrovní abstrakce, a to od nejvyšší úrovně abstrakce (analytické modelování) až po úroveň abstrakce implementační (nasazení souborů, zdrojů apod.). Jeden a tentýž systém je tedy viděn na různých úrovních abstrakce jinak: na nejvyšší úrovni je viděn pouze jako „funkcionalita systému“ (co systém umí a co dělá) v nejvyšší pojmové abstrakci, na nejnižší úrovni se jedná o skupiny bajtů v souborech na strojích. Někde mezi tím existuje abstraktní úroveň tvorby kódu. Zjednodušeně řečeno, na nejvyšším stupni abstrakce je systém viděn a popisován (modelován) pouze ve funkcionalitě a je tak viděn všemi účastníky projektu, a to i těmi, kteří systém neprogramují (zákazník, analytik, obchodní oddělení, atd.) .

Poznámka: Úrovně abstrakce jsou známy i v normálním „občanském životě“. Například když potápějící loď vysílá morseovkou signál SOS žádající o pomoc, můžeme tento signál vidět v těchto abstrakcích:

- *Zjistili jsme volání o pomoc (nejvyšší abstrakce).*
- *Tři tečky, tři čárky, tři tečky (nižší abstrakce).*
- *Naměřili jsme modulované elektromagnetické vlnění (nejnižší abstrakce).*

Je třeba si uvědomit, že všechny tři abstrakce dohromady popisují totéž volání. Nejnižší „implementační“ úroveň je „čistě fyzikální“ a nutná pro samotný přenos a jeho realizaci, nejvyšší abstraktní úroveň vyjadřuje „podstatu věci“ a může být implementována i jinou technologií.

Z čistě pragmatického a praktického hlediska se ukazuje jako dostatečné dodržet při tvorbě softwaru minimálně tři úrovně abstrakce:

- analytické modelování
- modelování designu (neboli návrh)
- kódování a implementace („napsání kódu“ a nasazení)

Pohled z úrovně analytického modelování vystihuje základní otázka „co?“, tedy otázka, co daný software vlastně řeší. Design oproti tomu vystihuje otázka „jak?“, tj. jak bude toto zmíněné „co?“ realizováno v daném prostředí pomocí určených vývojových prostředků. Tento průchod abstrakcemi se při tvorbě softwaru provádí vždy a bez výjimky. Největší chybou SW firem bývá podle mých zkušeností to, že se abstraktní úroveň analytického modelování prostě nedokumentuje. To je mnohdy matoucí skutečnost a vede k mylnému závěru, že „analýza jakoby neexistovala“. Neznamená to však, že software neprošel fází analýzy! Není totiž pochyb o tom, že u každého softwaru se vždy musí určit „co má daný software řešit“, proto nutně existuje modelování v analýze, někdy však toto modelování existuje pouze v hlavách pracovníků bez následného výstupu do modelů „na papír“.

Stejně tak se musí navrhnout poté i „jak“ konkrétně bude toto analytické „co“ řešeno. To je náplní designu. Poté je třeba tento návrh zrealizovat (poslední fáze abstrakce – implementace, kódování).

Blíže viz literatura na serveru http://www.objects.cz

Poznámka: Z vlastní dlouholeté praxe konzultanta mohu potvrdit, že existuje jedna velmi často se opakující situace ve firmách, která dokládá nutnost zavedení těchto úvah o úrovních abstrakce a nutnosti zavést analytické modelování. Touto často se opakující situací je stav ve firmě, kdy se požaduje přechod z jedné technologie na jinou při zachování (takřka) stejné analytické podstaty softwaru bez zásadních změn v analýze (systém bude řešit „takřka“ totéž, ale na jiné technologii). Většinou je tento požadavek spojován s přechodem z klasické „2-vrstvové“ technologie (reprezentována architekturou klient databáze / server) na Web technologie. Ukazuje se, že i při přechodu z jedné technologie na jinou odlišnou technologii je třeba vždy provádět reverzně analýzu systému. Jinak řečeno, nelze prostě přepsat systém z jedné technologie do druhé („jakoby automatem“). Přechod na jinou technologii vždy usnadňuje existence zdokumentované analýzy.

Jak sám název napovídá, Design Patterns (návrhové vzory) se úzce týkají té fáze tvorby, která se nazývá návrhem (design). V podstatě to znamená, že návrhové vzory se nezabývají analýzou

systému, tj. při užití Design Patterns se již nezajímáme o to, „co“ má systém řešit, ale „jak“ se ono „co“ má konkrétně řešit v objektově orientovaném prostředí.

Z toho plyne závěr, že modely v Design Patterns jsou přímo modely programu napsaného objektově orientovaným způsobem a nejsou to modely pojmové abstrakce z analýzy. Jinak a stručně řečeno modely UML, které budeme v této knize zavádět, jsou modely návrhu a nikoliv modely analytickými. Znamená to, že pokud v této knize zavedeme třídu v modelu UML, potom tato třída je opravdu již přímým obrazem vůči třídám v programovacím jazyce a také instance tříd v modelech mají přímý obraz v instancích (objektech) v programovacím jazyce.

Poznámka: Obecně nemusí platit, že „co instance informace v analýze, to instance objektu žijící v paměti“. Jak bude vypadat tento vztah, je výsledkem přechodu od analýzy do designu. Důsledkem této úvahy může být například řešení situace, kdy analytik v analýze vysloví větu: „Bude se evidovat cca asi 100 000 subjektů – firem“. Designér nenavrhne tuto konstrukci v OOP ve smyslu „100 000 objektů žijících v paměti“, ale nějak jinak. Předešlá úvaha vyslovuje důležitý fakt, že tento proces přechodu od analýzy do designu již máme za sebou a modely tříd a vztahy objektů ve vzorech popisují již návrh samotného programu jako takového a jsou tedy jeho přímým „grafickým obrazem“.

Základním posláním Design Patterns je napomoci a usnadnit návrh (design) systému. Jinak řečeno smyslem návrhových vzorů je usnadnit práci designérovi při tvorbě dobrého a kvalitního návrhu v OOP.

Z předešlé věty vyplývají dvě základní otázky:

- co se považuje za dobrý a kvalitní návrh v OOP?
- jak návrhové vzory napomáhají vytvořit tento zmíněný dobrý a kvalitní návrh v OOP?

Na první otázku si odpovíme metodou „vyvarování se negace“. V dalších kapitolách si vyjmenujeme známé problémy, se kterými se můžeme při návrhu v OOP setkat a které, pokud je neřešíme, vedou k nepříliš šťastnému návrhu.

Na druhou otázku si odpovíme výčtem všech dosud známých vzorů a vysvětlením těchto vzorů, tj. vysvětlením skladby vzorů, vysvětlením smyslu a principů fungování.

I když oblast návrhových vzorů spadá do oblasti designu v OOP, přesto se jejich zvládnutí a použití může nepřímo promítnout i do kvality analýzy. Jeden a tentýž požadavek může být totiž realizován různým způsobem i v analýze. Pokud je analytik dobře obeznámen s konstrukcemi v Design Patterns, potom může po dohodě se zákazníkem přímo v analýze navrhnout takové analytické konstrukce, které vystihují a řeší daný problém stejně dobře, a přitom jsou z hlediska dalšího vývoje kvalitnější.

K této otázce se vrátíme konkrétně u některých vzorů (například COMPOSITE apod.).

Co je Design Pattern a jaké má elementy?

Abstrakce vzoru, motiv vzoru a smysl vzoru

Obecně lze vzor považovat za opětovně použitelný obecný návod k řešení problému, který se neustále opakuje v různých obdobách a v různých situacích.

Na chápání a na použití vzorů je obtížná právě jejich abstrakce. Samotný vzor totiž nepopisuje nějakou konkrétní situaci k řešení, ale popisuje zobecnění všech takovýchto situací. Toto zobecnění opět vychází ze základního principu dichotomie a vztahu meta-úrovně a její instance (viz například e-kniha [1]). Úvaha zní sice složitě, ale je ve své podstatě jednoduchá:

Vzor je chápán jako obdoba vzorce v matematice. Neexistují v něm konkrétní třídy ani konkrétní objekty (příp. jiné prvky modelů), ale vyskytují se v nich jejich proměnné (proměnné třídy, proměnné objekty atd.). Tyto proměnné jsou chápány jako role, za které lze dosadit konkrétní třídy, konkrétní objekty resp. jiné konkrétní prvky našeho problému, a tak dostáváme řešení naší situace. Tyto proměnné, neboli role (tj. prvky modelu vzoru), se nazývají obvykle „účastníci vzoru“ (angl. „participants“).

Podotkněme, že samo UML umožňuje práci s prvky modelu chápanými podobně jako template (tj. jako prvky šablony), které se stávají konkrétními prvky konkrétního modelu po dosazení konkrétních hodnot (viz například [2], kapitola 3.3.1.1.).

Vlastnost abstrakce vzoru je mnohdy překážkou pro jeho správné pochopení. Obecně totiž platí, že člověk lépe chápe problematiku na konkrétních instancích a nikoliv na jejich zobecnění. Úvahy na úrovni „meta“ jsou vždy obtížnější a méně pochopitelné.

Poznámka: Například úvahy v „meta meta“ úrovni jsou již takřka neschůdné.

Tato skutečnost obtížných úvah v meta úrovni se například projevuje v doporučení objektového modelování, aby se složité modely tříd nejprve navrhovaly na úrovních instancí (jako příklad instancí tříd) a teprve v druhém kroku se zobecňovalo do meta modelu (viz například [1]). Nejprve se tedy hovoří ve stylu: „Představme si nějakou konkrétní fakturu“ a teprve v druhém kroku se uvažuje o zobecňujícím pojmu faktury jako třídy. Tento postup výrazně urychluje analytické práce při tvorbě modelu tříd.

Podobně je tomu i ve vzoru. Vzor se vždy vysvětluje na konkrétních použitích. Dokonce jedno z těchto použití, pokud možno co nejvíce názorné, je zavedeno jako povinná součást vzoru a nazývá se „motiv vzoru“ (angl. „motivation“). Motiv se uvádí jako povinná součást vzoru. Nejedná se o nic jiného, než o určitou vybranou situaci s konkrétním problémem, který se podaří vyřešit a poté se toto řešení zobecní do vzoru. Motiv je tedy chápán jako „první“ známé použití vzoru, které vyústí v jeho zavedení. Samotný obecnější vzor se na něm velmi dobře pochopí. Motiv se tak stává nejen nezbytnou součástí vzoru, ale také částí velmi prospěšnou pro jeho pochopení.

Doporučuje se (a také se k tomuto doporučení připojuji) zahajovat podrobné studium vzoru na jeho motivu. Postup studia a použití předkládaného vzoru je tedy následující: Prostudujeme si motiv vzoru. Jedná se o konkrétní situaci k řešení. Na motivu pochopíme konkrétní problém a jak byl konkrétně řešen. Prostudujeme si další části vzoru, účastníky vzoru apod. Na něm pochopíme obecné použití vzoru.

To nejobtížnější v práci se vzory nás však teprve čeká: Představme si, že tvoříme software a máme před sebou nějakou konkrétní situaci. Musíme odhalit, zda a který vzor se této situace týká. Dokonce se může stát, že si ani neuvědomíme, že vůbec nějaký vzor s naší situací souvisí. Jedinou pomůckou v tomto hledání je společný problém dané situace a problém již vyřešený pomocí některého ze vzorů. Proto bychom měli znát použitelné vzory také ve tomto pohledu jako „velmi stručně vystižený problém a velmi stručně vystižené řešení tohoto problému“, pokud možno jednou větou. Z tohoto důvodu se zavádí další povinná část vzoru zvaná „smysl“ vzoru (angl. „intent“). Jedná se o velmi stručné vystižení „problému a řešení“ pomocí vzoru. Většinou se jedná o velmi krátké věty vystihující daný vzor. Doporučuji tyto stručné popisy vzoru znát takříkajíc „zpaměti“. Na základě tohoto stručného popisu můžeme velmi rychle posoudit, zda naše situace odpovídá některému ze vzorů a zda se dá zlepšit použitím daného vzoru. Je pochopitelné, že tato část vzoru neslouží k vysvětlení vzoru, ale k velmi stručnému vyjádření smyslu vzoru. Část vzoru „smysl“ je určena k tomu, aby se vzor buď dobře pamatoval (ve své zkratce), anebo při jeho rychlém přečtení se rychle vybavil. Znamená to, že smysl vzoru slouží k rychlé orientaci mezi vzory, tedy k jeho rychlé identifikaci.

Navíc běžně nastává taková situace, kdy řešení spočívá v použití celé kombinace několika vzorů současně, což poskytuje další netušené možnosti, ale vede nutně k rychlé orientaci mezi vzory.

Katalog a klasifikace vzorů

Pro knihovnu vzorů, tj. množinu vzorů, které má firma k dispozici, se ustálil název „katalog vzorů“. Je zřejmé, že se vyžaduje, abychom mohli s tímto katalogem efektivně pracovat a vzory rychle vyhledávat. Jak bylo řečeno v předešlé kapitole, slouží k tomu mimo jiné část vzoru zvaná „smysl vzoru“ stručně vystihující podstatu vzoru.

Kromě toho, že se můžeme rychle orientovat podle části vzoru nazvané „smysl vzoru“, existuje ještě druhé hledisko, které nám může usnadnit práci s katalogem. Vzory lze rozdělit i do klasifikace, tj. katalog lze pomyslně rozdělit do skupin vzorů, tedy lze jej členit podle nějakého kritéria. Podle tohoto kritéria se můžeme ve vzorech rychle orientovat. Tomuto členění budeme říkat klasifikace vzoru (angl. „classification“).

Existují dvě základní kritéria klasifikace:

- klasifikace podle účelu (angl. „purpose“)
- klasifikace podle rozsahu platnosti (angl. „scope“)

Klasifikace podle účelu (purpose) se člení na tyto tři hodnoty:

- vzory pro tvorbu objektů (angl. „creational patterns“). Tyto vzory jsou zaměřeny na tvorbu objektů a s tím související problematiku (odstínění tříd apod.).
- vzory pro struktury (angl „structural patterns“). Tyto vzory jsou zaměřeny na řešení opakujících se struktur objektů.
- vzory řešící chování („behavioral patterns“). Tyto vzory řeší problematiku chování objektů (spolupráce, odstínění chování apod.).

Každý vzor se začlení do některé z těchto skupin. Teoreticky vzato mohou existovat vzory, které by bylo možné zařadit do více než jedné skupiny.

Klasifikace podle rozsahu platnosti (scope) se člení na tyto dvě hodnoty:

- vzory s rozsahem platnosti tříd („class patterns“). Vzory s tímto rozsahem popisují interakce mezi třídami (asociace, agregace, dědění).
- vzory s rozsahem platnosti objektů („object patterns“). Vzory s tímto rozsahem popisují vztahy mezi objekty. Jsou mnohem flexibilnější než vzory typu class a většinou řeší problémy typu flexibilních změn „něčeho v runtime“.

Z praktického hlediska je důležité rozdělení vzorů podle účelu, protože podle tohoto rozdělení se v katalogu orientujeme velmi dobře. Rozdělení podle rozsahu platnosti (scope) udává spíše způsob, jak je vzor tvořen a na jakých základech je postaven.

Skladba vzoru

Design Pattern obsahuje (podle základní literatury [5]) tyto elementy:

Název vzoru a jeho klasifikace

Název vzoru stručně vystihuje jeho poslání. Pojem klasifikace byl vysvětlen v předešlé kapitole.

Smysl (Intent)

Jedná se o stručné vystižení vzoru, většinou jednou větou. (vysvětlení viz předešlé kapitoly).

Alias (Also Known As)

V mnoha případech se daný vzor vyskytuje v literatuře pod několika názvy a proto je možné, že se čtenář již s tímto vzorem setkal, ale pod jiným názvem. V této části vzoru je snaha sepsat všechny jeho známé názvy, tj. „aliasy“.

Motiv

Jedno použití vzoru je vybráno jako „počáteční motiv“. Na jednom příkladu se demonstruje vzniklý problém a jeho konkrétní řešení. V další části vzoru se toto řešení zobecňuje. Doporučuji, abyste studium vzoru zahajovali vždy od motivu.

Aplikovatelnost

V této části vzoru se popisují podmínky, které vedou k použití vzoru, tj. vypíše se stručně, kdy se vzor používá. Výčet je však orientační, popisuje dosud známé situace a nemusí být proto úplný.

Struktura vzoru

Tato část vzoru popisuje vzor jako takový. Vystupují v něm role vzoru – účastníci vzoru.

Účastníci vzoru

V této části vzoru nalezneme všechny role ve vzoru s popisem, tj. účastníky vzoru. Snahou je vystihnout jejich postavení a účast ve vzoru.

Spolupráce

Zde nalezneme princip spolupráce mezi účastníky vzoru. Jsou zde v podstatě explicitně popsány interakce mezi účastníky vzoru. Nemusí se však vždy jednat o spolupráci ve smyslu dynamiky vzoru (tj. pouze změny v čase), jako jsou například zasílání zpráv mezi objekty, scénáře apod., i když tomu tak většinou bývá. Někdy se popisuje i statická interakce mezi třídami (například „třída předek přenechává kompetenci třídě - svému potomkovi“ apod.).

Důsledky

Snahou je vystihnout všechny možné důsledky použití, a to jak kladné, tak i záporné. Upozorňuje se jak na výhody použití, tak na možná úskalí vzoru v různých prostředích.

Implementace

V této části bychom měli nalézt poznámky k různým implementacím vzoru. Z historických důvodů nalezneme v literatuře poznámky zejména k implementaci v C++ , Smalltalku apod. (například [5]). V poslední době se objevují také Design Patterns v implementaci zaměřené na jazyky JAVA a rodinu jazyků pod DOT.NET (viz například [3] + [4]). Zde v této knize budeme používat pseudo-jazyk spojující srozumitelně a logicky syntaxi jazyků C# a JAVA (s vysvětlením do Delphi). Tato syntaxe bude velmi dobře srozumitelná okruhům čtenářů, jak pro JAVU, tak pro C#. Tento jazyk budeme nadále nazývat JAVAC# a jeho syntaxe bude dále upřesněna. Současně si ukážeme mapování z tohoto jazyka na jazyk Object Pascal (Delphi).

Příklad vzoru

V této části se uvádí nějaký konkrétní příklad. Vzhledem k zaměření na pseudojazyk JAVAC# bude příklad vysvětlen již v části implementace.

Známé použití

Pokud je známo, vyjmenují se případy použití. V této knize nejsou případy v ČR uvedeny (není známo), v mezinárodním měřítku odkazujeme na knihu [5].

Související vzory

Většinou se pro řešení situace nevolí jeden vzor, ale kombinace použití vzorů. V tom případě se zde tato možná kombinace použití uvede. Současně se v této části uvádějí vzory zástupné (zaměnitelné, konkurenční) s diskusí, který je v které situaci lepší a s jakými výhodami a nevýhodami.

V této knize vzhledem k dost podrobného vysvětlení použijeme modifikovanou a zkrácenou strukturu vzoru, která vychází z předešlého výčtu. U každého vzoru použijeme následující podkapitoly (vysvětlivky viz předešlý výčet):

Struktura vzoru použitá v této knize:

- Název a klasifikace (odpovídá předešlému výčtu)
- Alias (odpovídá předešlému výčtu)
- Motiv (odpovídá předešlému výčtu)
- Struktura a popis vzoru (obsahuje z předešlého výčtu strukturu, účastníky a spolupráci)
- Poznámky k použití (obsahuje z předešlého výčtu důsledky, implementaci, příklady)
- Související vzory (odpovídá předešlému výčtu)

Modifikace vzoru do této podoby poskytuje větší prostor pro vysvětlení podrobností vzoru, přičemž „důležité“ kapitoly pro identifikaci a souvislosti zůstanou zachovány.

Dobrý návrh a flexibilita návrhu

Pokud provádíme návrh v OOP, můžeme se setkat s určitými riskantními problémy, které souvisejí s flexibilitou. Z nich si nyní uvedeme ty, které se vyskytují nejčastěji. Tyto problémy nazýváme riskantními proto, že v případě, že je nehodláme zohlednit (ať už z neznalosti anebo z pohodlnosti), narůstá v budoucnu riziko, že se tyto problémy projeví negativně na kvalitě softwaru resp. se projeví ve vyšší pracnosti jeho dalšího vývoje a úprav. Jinak řečeno, opomenutí těchto rizik sice vede k řešení a vyhotovení softwaru, avšak toto řešení nemusí být ideální. Může v sobě skrývat nepříjemné důsledky zejména pro budoucí úpravy, skrývá v sobě do budoucna technologické problémy apod.

Poznámka: V této souvislosti se většinou hovoří o tzv. flexibilitě softwaru. Tuto flexibilitu můžeme jednoduše vyjádřit slovy: „Co se změní, když...“ Tuto jednoduchou až primitivní větu mějme vždy na paměti při návrhu informačních systémů.

Nutno podotknout, že následující seznam problémů rizik jednak není konečný a krom toho nám nepřikazuje se vždy podle nich řídit. Nesnažíme se tedy „za každou cenu“ se těmto rizikům vyhnout. Může se stát, že řešíme nějaký software a v následujícím výčtu nenalezneme ani jeden z případů, který by nějak ohrožoval vývoj našeho produktu. To je sice pravda, ale neznamená to, že máme tento seznam ignorovat, naopak. Je vždy bezpodmínečně nutné projít tento seznam (případně ho rozšířit) a u každého bodu tohoto seznamu si položit jednoduchou otázku: Má pro nás toto riziko smysl anebo můžeme s jistotou říci – tato situace u nás nikdy nenastane a můžeme tedy toto riziko nekompromisně opominout?

Poznámka: Každý vývojář má neblahou zkušenost s těmito jistotami, že „toto nikdy nemůže nastat“, zejména pokud to tvrdí zákazník. Murphyho zákony jsou v tomto případě neúprosné:

„Pravděpodobnost, že určitá situace v softwaru nenastane je přímo úměrná počtu ujištění od zákazníka, že tato situace nikdy nenastane...“

Seznam dosud známých rizik (který lze pochopitelně dále rozšířit) můžeme vypsat do následujícího seznamu:

Závislost objektů na třídě

Je zřejmé, že pokud chceme použít nějaký objekt v OOP, musíme zavolat konstruktor třídy a tedy specifikovat třídu, ze které má tento objekt pocházet. Tato věc se jeví jako triviální, avšak vede jednomu nepříjemnému důsledku: Pokud specifikujeme třídu, potom píšeme „natvrdo“, že daný objekt pochází z této třídy a s tím mohou být problémy. Toto „oslovení a žádost o nový objekt“ přes „přímé“ volání konstruktoru není totiž flexibilní a dynamické. Na první pohled by se mohlo zdát, že v toto riziko „závislosti objektu na třídě“ je nějaký protimluv. Vždyť přece musíme vždy specifikovat třídu a volat konstruktor! Tento problém je řešen hned několika vzory pro tvorbu objektů („creational patterns“).

Závislost na specifické operaci

Zde je pod operací myšlena tzv. operace objektu definovaná ve třídě (zpráva objektu), tj. „to, co se dá objektu volat“. Pokud navrhneme program tak, že existuje v určitém bodě závislost na přímém volání objektu, může se tato skutečnost projevit jako nepříznivá a může vést k pracnému řešení například při rozšíření programu. Tento problém řeší například vzory chování.

Závislost na SW a HW

Problém závislosti na HW ve smyslu operačního systému je obecně znám a je v Javě a DOT.NET řešen již na úrovni vývojového prostředí „odstíněním“ operačního systému pomocí vrstvy tohoto prostředí. Podobně si můžeme představit závislost na vloženém „cizím prvku“ (vložená „cizí“ komponenta), na externím systému apod. Opět vyvstává otázka, jak efektivně tyto prvky „odstínit“, a tak získat systém, který bude nezávislý na výměnách těchto prvků.

Závislost na objektové implementaci

Hlavním cílem těchto vzorů je efektivně a flexibilně (vyměnitelně) ukrýt objektovou implementaci. Znamená to, že při zachování principu zapouzdření je možné například změnit implementaci objektů, například způsob uložení, způsob grafického zobrazení apod., aniž by se zasahovalo do klientů těchto objektů.

Závislost na algoritmu

Je zřejmé, že uvnitř programů jsou uchovány algoritmy určitých zpracování, které pocházejí z analýzy. Protože se algoritmy vyvíjejí, je žádoucí, aby klienti těchto algoritmů nebyli na těchto algoritmech závislí a byli od nich „odstíněni“. V takovém případě lze například algoritmus měnit v runtime. Při dobrém návrhu lze například bez zásahu do již existujícího programu přidat jiné modifikované algoritmy (pouze přidáním komponenty apod.) .

Těsné vazby objektů

Objekty musejí obecně mezi sebou spolupracovat, avšak těsné vazby mezi objekty v nesprávných bodech vedou ke ztrátě flexibility programu. Nastává přímé volání mezi objekty, které nejsou od sebe oddělitelné. Vznikají tak nežádoucí monolity.

Přeceňování dědění - subclassingu

V některých případech časté používání dědění (subclassing) a přeceňování tohoto způsobu re-use vede k velkým problémům hlavně s nárůstem počtu tříd, vzniku kombinace tříd, anebo k problémům

neměnitelných statických vazeb. Mnohdy je řešením nahrazení této interakce jiným typem interakce – běžnou asociací anebo agregací.

Změny bez přístupu ke kódu třídy

Pokud používáme externí třídy (resp. interfacy) tvořené mimo náš systém a jejich kód je mimo naši kontrolu, mnohdy vyvstane otázka, jak změnit chování objektů z třídy, kde je kód již daný, ale přitom je nepřístupný. V tom případě se používají vzory řešící tuto situaci.

Základní nosná myšlenka všech vzorů

Pokud studujete návrhové vzory, časem zjistíte, že tyto vzory mají něco podobného. Návrhové vzory mají určitou stejnou nosnou myšlenku, která se opakuje a prolíná se všemi vzory. Je dobré si tuto myšlenku uvést dopředu, upozornit na ni a naučit se ji používat.

Prvním cílem této kapitoly je tedy naučit se tomuto náhledu. Mimochodem, jedná se o důsledné a názorné použití OOP, takže vysvětlení tohoto axiomu je názorné i z hlediska učení se OOP.

Současně s touto myšlenkou, kterou si vysvětlíme na příkladu, si také v této kapitole uvedeme dohody syntaxe jak modelů, tak pseudo-jazyka JAVAC# a mapování do jiných jazyků. Využijeme této příležitosti, abychom si zavedli syntaxi implementace vzorů v této knize.

Interfacový přístup k problému

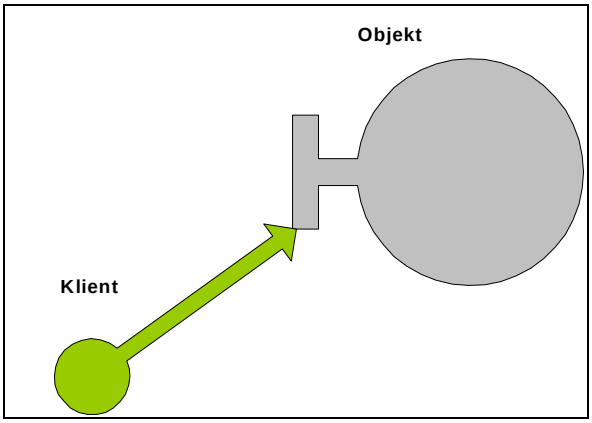
Jak je uvedeno v literatuře pojednávající o OOP (viz například [1]), na existující objekt je třeba se dívat tak říkajíc „schizofrenním“ pohledem jakoby ze dvou stran, tj. ze dvou odlišných pohledů.

Objekt je na jedné straně viděn zvenku klientem, který objekt používá. Klientem je jiná část SW systému než objekt a tato část systému používá daný objekt. Klient objektu má zpřístupněn interface objektu, tj. množinu zpráv objektu. Díky tomuto interfacu může klient volat objekt, tedy používat interface. Použít interface znamená poslat zprávu objektu a vyvolat tak jeho metodu (viz [1]). Pro zprávu objektu se používá také synonymum „operace“. Poslat zprávu znamená vyvolat operaci a tím se vyvolá odpovídající implementovaná metoda. Znamená to, že obecně mezi volatelnou operací a implementovanou metodou existuje „převodník operace-metoda“, který se nazývá protokol zpráv. Díky zapouzdření objektu neexistuje jiná možnost, jak spolupracovat s objektem, než vyvolat operaci objektu (poslat mu zprávu).

Klientovi není zpřístupněn vnitřek objektu a objekt je pro něj reprezentován pouze interfacem, tj. množinou zpráv, neboli množinou operací. Tento pohled je pohled na objekt jako na používaný interface (vnější pohled).

Existuje druhý pohled, z opačné strany, zevnitř objektu. Poslání zprávy znamená vyvolat metodu a ta něco provede. Metoda je nějakým způsobem vytvořena, je implementována, neboli konkrétně naprogramována. Tento druhý pohled (za interfacem), budeme nazývat implementační pohled. Základní myšlenka spočívá v tom, že tento pohled je klientovi objektu zásadně skryt a díky tomu není na tomto implementačním pohledu závislý. To je přímý důsledek zapouzdření objektů.

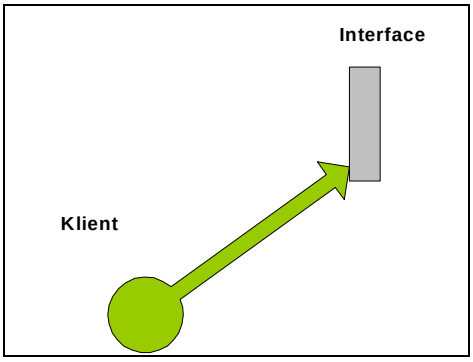
Většinou se objekt využívaný klientem objektu maluje následujícím ilustračním obrázkem:



obrázek 1 : Klient objektu, interface a objekt

Na obrázku je zřetelně znázorněno, že klient má přístupný interface objektu (znázorněno jako svislá šedá čára), klient má nějak tento interface zpřístupněn (například pomocí ukazatele). Tím je myšleno, že klient může volat operace objektu daného interfacu, a tím vyvolávat funkcionalitu daného objektu.

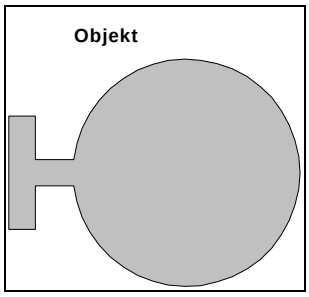
Tento obrázek je samozřejmě pravdivý, avšak je lepší jej vidět názorněji rozdělen na **dva** obrázky (se dvěma větami):



obrázek 2 Klient vidí interface (a nic jiného)

V tomto případě jsme znázornili situaci tak, jak ji vidí a chápe klient objektu: Klient vidí a používá interface a nic jiného z objektu nevidí .

Přitom však platí i druhý obrázek:



obrázek 3 Objekt implementuje operace (které nabízí pomocí interfacu) do metod konkrétně, vnitřek objektu (klient není viděn).

V tomto případě se popisuje vnitřek objektu, tj. popisuje se implementace objektu. Obrázky 2 a 3 platí „dohromady“ a pochopitelně dávají obrázek 1. V tom je však určitá záludnost. Obrázek 1 totiž popisuje najednou dvě od sebe oddělené situace, tj. to, co vidíme na obrázku 2, a to, co vidíme na obrázku 3.

Uvedené rozdělení má totiž velmi zajímavý důsledek pro návrh programu: Pokud klientovi nabídneme jiný objekt se stejným interfacem (a tedy jinou implementaci), klient tuto záměnu přijímá jako kompatibilní záměnu.

Poznámka: V podstatě jsme v předešlých větách popsali jeden z přímých důsledků použití polymorfismu v OOP (viz např. [1]).

Co je však důležité pro studium a zavádění Design Patterns, je doporučení, abychom se na každou situaci k řešení dívali přístupem „klient – interface“ a nikoliv „interface – implementace“. Znamená to, že klíčovým prvkem pro chápání použití Design Patterns se jeví obrázek 2 a nikoliv 1 nebo 3. Zaměření se na interface a jeho vidění klientem je totiž dobrou pomůckou pro získání flexibility.

Poznámka: Je to podobné, jako bychom se u vzájemně propojovaných strojů zaměřili na kompatibilitu zástrček mezi sebou a přitom vnitřek strojů (tj. prostor za zástrčkami) považovali za vedlejší.

Doporučení autorů návrhových vzorů zní ve zkratce:

„Zaměřte se na přístup přes spolupráci pomocí interfaců a nikoliv na implementaci objektů“.

Polymorfismus jako hlavní nástroj návrhových vzorů

Dá se říci, že nosným pilířem návrhových vzorů je využití polymorfismu v OOP. Protože v tomto případě opravdu jde o základní myšlenku návrhových vzorů, je třeba ji dobře a do všech důsledků pochopit. Z toho důvodu si nyní uvedeme na příkladu jednak způsob použití polymorfismu a také jeho výhody. Současně si vysvětlíme jednoduchou syntaxi našeho pseudojazyka spolu s ukázkou modelování v UML.

Aplikace „Vystřihovánky“

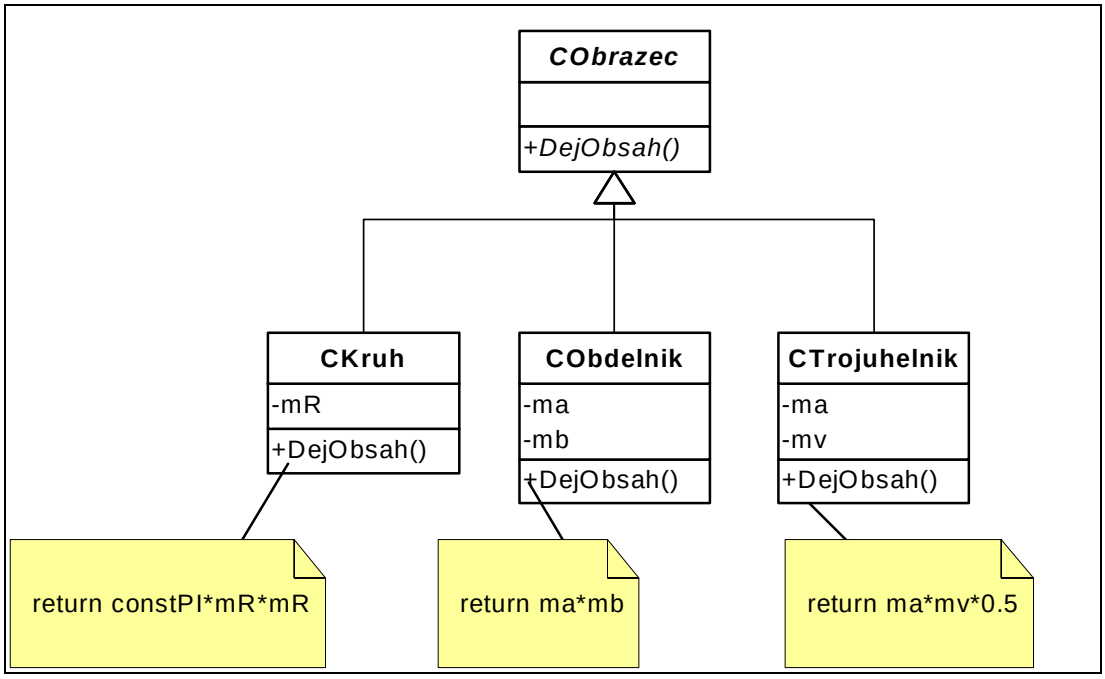
Představme si, že budeme řešit velmi jednoduchou aplikaci určenou pro práci s obrazci a jejich obsahy. Například si můžeme představit, že se jedná o výpočet obsahu obrazců při spotřebě materiálu (kůže, látka apod.). Mohli bychom tuto aplikaci nazvat například „Vystřihovánky“. Princip práce s aplikací je velmi prostý: Obsluha zvolí nový obrazec a zadá mu parametry přes formulář. Obrazec se přidá k již existujícím obrazcům. Obsluha může kdykoliv požádat systém, aby vydal obsah všech již zadaných obrazců, případně obrazec vymazat, změnit mu parametry apod.

Pro řešení scénáře vydání obsahu zvolíme polymorfismus. Každý obrazec bude mít operaci (zprávu), nazvěme ji například `DejObsah()`. Její návratová hodnota je například reálné číslo, tj. číslo typu `real`. Základní myšlenkou je to, že každý typ obrazce umí vracet `DejObsah()`, ale každý z nich tak činí jinak (má jinou implementaci metody). Pro vyjádření této skutečnosti využijeme konstrukce přepisování metod tak, jak je tento způsob polymorfismu zaveden v Javě, C# a také v Delphi.

Zaveďme třídu jako společného předka všem uvažovaným obrazcům a nazvěme ji `CObrazec`. Tato třída bude mít dědice `CKruh`, `CObdelnik` a `CTrojuhelnik`. Každá z těchto tříd (potomci) zavádí své vnitřní atributy podle povahy obrazce. Tyto atributy jsou nutné pro výpočet obsahu: Kruh má svůj vnitřní atribut - poloměr označený jako `mR` (předpona značí member), Obdélník má své vnitřní parametry – dvě strany `ma` a `mb`, Trojúhelník má své vnitřní atributy - základna a výška, tj. `ma` a `mv`. Všechny tyto vnitřní atributy jsou (jak jinak) privátní a zvně objektu neviditelné (mimo dosah viditelnosti). Pro jejich nastavení resp. čtení je třeba zavolat operace interfacu, například nějak takto `MujKruh.SetR(real a_R)`. Případně se použijí operace označené jako property u jazyků, které property podporují, například takto: `MujKruh.R = a_R`. Tato část řešení nastavování a čtení vlastností obrazců jsou pro nás nezajímavé a nebudeme se jimi dále zabývat.

Zajímavější je řešení vydání obsahu `DejObsah()`. Tato operace se definuje jak v předkovi, tak v potomkovi (programátorsky řečeno „v předkovi a v potomkovi se opakují hlavičky operací“) . Pomocí rezervovaných slov (jak bude uvedeno dále) se vyjádří skutečnost, že potomek metodu nedědí, ale přepíše její obsah (změní její implementaci). Každý obrazec má tedy zavedenu operaci `DejObsah()`, ale každý z nich na ni reaguje svým obsahem (implementovanou metodou) jinak.

Pokud zapátráme ve středoškolské matematice, můžeme psát „vnitřky“ metod, tj. implementace a tuto situaci můžeme znázornit takto:



obrázek 4 Zavedení přepisující metody DejObsah

Poznámka pro objektový Pascal: V Pascalu zde zmíněný `return` nahrazuje proměnná `Result`, do které se vkládá návratová hodnota.

Předešlý diagram vyjadřuje následující skutečnosti:

1. Obrazce Kruh, Obdélník a Trojúhelník jsou dědici obecného Obrazce.
2. Každý z těchto dědiců přepisuje metodu `DejObsah()`. Pokud vytvoříme objekt z třídy Kruh, potom `DejObsah` vrací π R na druhou, pokud vytvoříme objekt ze třídy Obdélník, tento vrací na `DejObsah` jako součin stran a a b, pokud vytvoříme objekt ze třídy Trojúhelník, metoda `DejObsah` vrací polovinu součinu základny a výšky.
3. Díky bodu 1 platí tzv. kompatibilita „zespodu nahoru“ mezi obrazci. Pokud někde deklarujeme proměnnou typu `CObrazec`, můžeme do této proměnné přiřadit proměnnou typu dědice (obdélník, trojúhelník anebo kruh). Dědicové „znají“ a mohou nabízet interface `CObrazec`, ale každý z nich na volání přes tento interface reaguje jinak.

Práce s obrazci by tedy mohla být navržena v programu takto:

Zavedeme objekt jako kolekci pomocí služební třídy kolekce v daném jazyce (`Arraylist`, `TList`, `Collection` apod.). Tato kolekce bude mít klasické metody pro práci s itemy typu obrazec jako metody pro přidání prvku `Add(CObrazec a_Obrazec)`, pro získání prvku, například

`GetItem(integer i)` vracející proměnnou typu `CObrazec` (z pozice indexu `i`), metodu `Remove(integer i)` pro odstranění prvku apod.

Poznámka: V Delphi se typ proměnné v deklaraci udává za proměnnou a odděluje se dvojtečkou. V našem pseudojazce se udává těsně před proměnnou. Tedy zde napsaná hlavička metody

```
Add(CObrazec a_Obrazec)
```

by byla do Delphi mapována nějak takto:

```
Add(a_Obrazec : CObrazec)
```

Všimněme si, že naše kolekce pracuje s itemy typu `CObrazec`, které „obsahuje“ (drží na ně reference), a nikoliv s obecnými i objekty. Protože funguje kompatibilita mezi typy „zespodu nahoru“, můžeme jako vstupní parametr metody zvolit objekt kruh, obdélník nebo trojúhelník. Znamená to, že v kolekci se může vyskytovat (na přeskáčku) několik (a bůhví kolik) trojúhelníků, obdélníků a kruhů.

Tato kolekce bude mít navíc také metodu pro „vysoučtování“ všech obrazců v ní obsažených (tj. obrazců přidaných metodou `Add` a nevymazaných metodou `Remove`). Implementace této metody bude vypadat v našem pseudojazyce nějak takto:

```
...
Suma = 0;
For each Item in MojeKolekceObrazcu
 {
   Suma  = Suma + Item.DejObsah();
 }
return Suma;
```

Tento zápis předpokládá, že v našem jazyce kolekce znají syntaxi „`for each`“ (syntaxe pro průchod prvky kolekce). Pokud daný jazyk tuto syntaxi nezná, lze si pod „`for each`“ představit jednoduchý průchod kolekcí přes index (záleží na počáteční hodnotě indexu, od kterého se v kolekci počítá, zde od nuly):

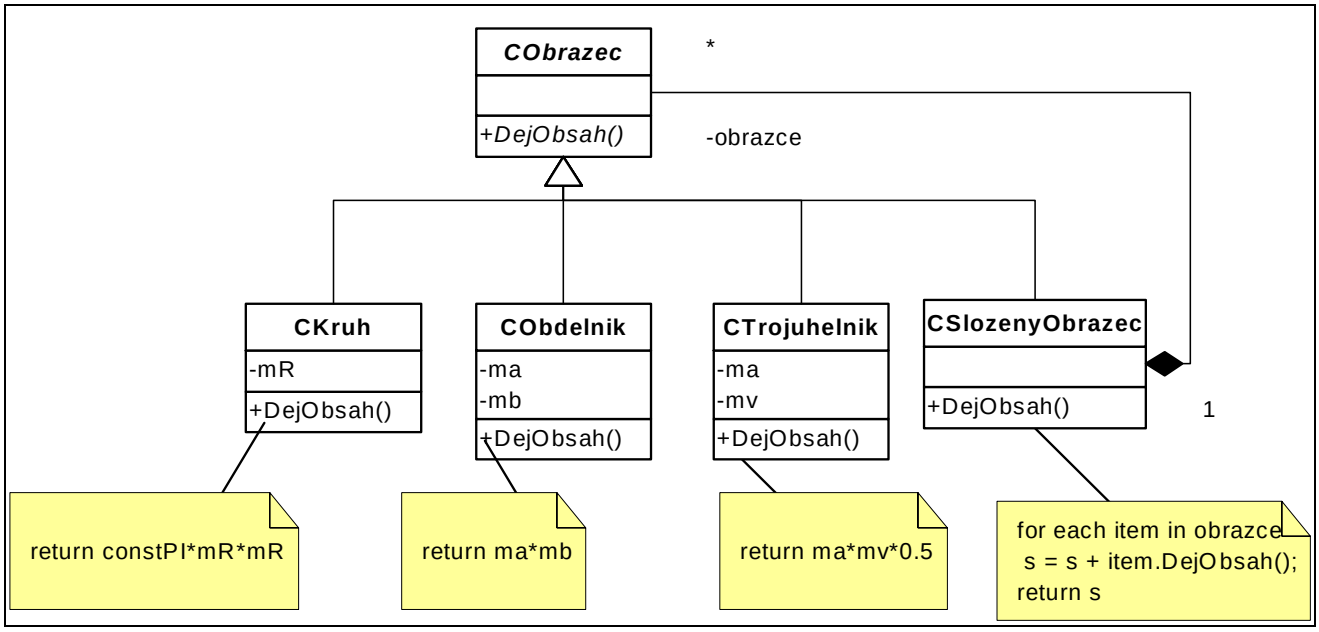
```
...
Suma = 0;
For I = 0 to MojeKolekceObrazcu.Count - 1
{
   Suma  = Suma + MojeKolekceObrazcu.GetItem(I).DejObsah;
}
return Suma;
```

Všimněme si, že všechny obrazce v kolekci jsou požádány o tutéž funkcionalitu, tj. `DejObsah()` a všechny obrazce to umějí (i když jsou z jiných tříd), ale každý obrazec na ni reaguje vnitřně jinak. V této „součtovací metodě“ se tyto odlišnosti vnitřků metod vůbec nijak neprojeví, protože kolekce své prvky vidí jako obrazce (přesněji „vidí interfacy obrazců“).

Postup práce by mohl být zvolen jako následující: Obsluha nejprve zvolí „nový obrazec“ určeného typu, objeví se formulář pro vyplnění hodnot atributů obrazce a obsluha vyplní odpovídající hodnoty. Tyto hodnoty se z formuláře „přesypou“ do obrazce a nový vyplněný obrazec se přidá do kolekce obrazců (přes metodu `Add`). Obsluha může kdykoliv požádat kolekci o vydání součtu obsahů a to tak, že se zavolá metoda, jejíž implementaci jsme uvedli o pár řádků výše (průchod kolekcí a posbírání obsahů do součtu).

Poznámka: Zmíněná kompatabilita zespodu nahoru samozřejmě znamená, že každý objekt v kolekci podporuje interface `CObrazec`, a proto může být do této kolekce zařazen. V rámci této kompatibility nemůžeme vyžadovat po prvku něco, co souvisí s jeho specializací a není obsaženo v interfacu `CObrazec`. Například představme si, že pro trojúhelník navrhneme jeho speciální metodu s názvem `JeRovnoramenný`, která vrací boolean hodnotu podle toho, zda se alespoň dvě strany shodují. Mimochodem, pro naši aplikaci je tato metoda absolutně nepotřebná. Tuto operaci znají pouze „trojúhelníci“ a nikdo jiný (není tedy součástí interfacu `CObrazec`) . Pokud chceme zjistit, že trojúhelník je rovnoramenný (pozor - pochází jako obrazec z kolekce!), nemůžeme se ptát přímo přes interface CObrazec. Musíme provést tzv. downcasting, tj. „přetypování dolů“, které, pokud se jedná opravdu o trojúhelník, nespadne a provede se. Pokud se provede downcasting nesprávně, například tak, že objekt je zrozen jako obdélník, je přidán do kolekce jako obrazec a my jej vyzvedneme jako obrazec a přetypujeme na trojúhelník, tak aplikace „spadne“. S uvedeným downcastingem pracujme velmi opatrně a ošetřujme jej, protože chyba se nemůže projevit jako chyba typu ve statickém módu (při psaní programu, editor a kompilátor „nic nehlásí“), ale až v běhu programu. Nejedná se totiž o chybu statických vztahů, ale jedná se o chybu v nesprávných pohybech s instancemi, tj. projeví se až v dynamice programu.

Předešlé úvahy jsou sice správné, ale mají v sobě jeden malý háček: V jednom bodě jsme nedotáhli správný „interfacový“ pohled do konce a přešli jsme na nedoporučovaný implementační pohled, a to u metody vracející sumu obsahů. Dotáhněme proto úvahy dále: Jak bude vypadat interface u zavedené kolekce obrazců, po které žádáme „vysoučtování obsahů“? Nabízí se pěkné a elegantní řešení: Uvedená operace vracející sumu součtů se bude také jmenovat `DejObsah()` a navíc kolekci zařadíme do rodiny obrazců. Kolekce bude „umět“ také `DejObsah()`stejně jako ostatní obrazce, tj. přepisovat její obsah jako každý jiný obrazec. Kolekce může být tedy zařazena do rodiny obrazců. Nazvěme tuto třídu jako `CSlozenyObrazec`. Předešlý obrázek modelu tříd v UML se tedy změní na následující:



Obrázek 5: Složený obrazec je umístěn v rodině obrazců

Tím, že jsme i složený obrazec vložili do rodiny obrazců (jako dědice třídy `CObrazec`), a tím, že jsme jej vybavili polymorfní operací `DejObsah()`, umožnili jsme vzniku rekurze ve skládání obrazců.

V kolekci obrazců uvnitř složeného obrazce se může vyskytovat libovolný obrazec, a to včetně dalšího složeného obrazce. Tuto situaci řeší jeden ze vzorů zvaný COMPOSITE, který dále a systematicky rozvíjí myšlenky skládání do ještě dalších elegantních vylepšení, jak si uvedeme v příslušné kapitole. Jenom upozorníme na skutečnost, že lze vytvořit zvláštní „složeniny“ jako například lichoběžník (dva trojúhelníky a jeden obdélník), aniž bychom zaváděli další třídu. Dále stojí za úvahu, kam umístit operace kolekce (pro práci s itemy), zda do interfacu předka obrazce anebo až dolů do složeného obrazce. Pokud totiž umístíme metody Add, Remove aj. do interfacu složeného obrazce a nikoliv do vyšší třídy obrazce, nastanou „složitosti“ při práci s prvky pro přidávání. Musíme totiž provádět zmíněný *downcast*, pokud získáme prvek ze stromu, protože obecný prvek ze stromu pak neumí Add a musí být přetypován. O tomto problému a řešení viz vzor COMPOSITE. Další úvahy by se týkaly problematiky zrodu nových obrazců. Pokud bychom zrod obrazců navrhli přímo a „natvrdo“, nastanou problémy s flexibilitou (pracností rozšíření). Představme si, že obsluha zvolí nějaký item GUI, například Menuitem. Do metody na událost kliknutí umístíme „natvrdo“ zrod daného typu obrazce.

Metoda na klik Menuitemu s textem „Nový trojúhelník“ bude definována takto:

```
Public void OnMenuItemNovyTrojuhelnikClick();

...

CTrojuhelnik MujTrojuhelnik = new CTrojuhelnik();

...
```

Poznámka: Volání `new CTrojuhelnik()` značí volání konstruktoru, který se jmenuje v tomto případě stejně jako třída, pouze se jedná o operaci konstrukce (viz závorky na konci). V Pascalu by se volal konstruktor pomocí odpovídajícího přiřazení při volání třídy, například nějak takto:

```
...

var MujTrojuhelnik : CTrojuhelnik;
```

```
MujTrojuhelnik := CTrojuhelnik.Create();
...
```

V tomto případě jsme si přímým voláním konstrukce a oslovením třídy přímo zadělali na drobný problém: Při přidání dalšího obrazce musíme „vstoupit“ do formuláře, otevřít jeho kód, přidat nový MenuItem a do něj přidat odpovídající kód pro zrod nového obrazce. Tímto problémem a jeho řešením se zabývají vzory pro zrod objektu a také vzory chování (viz dále).

Abstraktní třída, abstraktní operace, interface

Na předešlém příkladu se složenými obrazci jsme si vysvětlili polymorfismus. Kromě toho je třeba si vysvětlit ještě další důležité pojmy, které se v souvislosti s touto problematikou vyskytují.

Všimněme si, že v předešlých modelech UML jsme třídu `CObrazec` označili proloženým písmem. V syntaxi UML tato skutečnost znamená, že tato třída je tzv. abstraktní (v UML se značí pomocí modelovacího atributu *IsAbstract*, který je zde nastaven na „True“). Třída, která není abstraktní, se nazývá konkrétní.

Abstraktní třída je taková třída, ze které nemá smysl tvořit instance a ani to není žádoucí. Například tvořit objekty jako obrazce ze třídy `CObrazec` v tomto případě je zřejmý nesmysl. Abstraktní třída slouží pouze jako předloha pro dědění. Velmi často budeme ve vzorech rozlišovat, zda se jedná o třídu abstraktní anebo o třídu konkrétní. Moderní programovací jazyky mají tuto syntaxi nějak zavedenu, protože se jedná o velmi silnou omezující podmínku vedoucí k vyšší stabilitě programu, jedná se o podstatné a správné omezení re-use při tvorbě objektů (abstraktní třída se nesmí použít pro konstrukci objektů).

Také u polymorfních operací, které zavádějí předka a jsou přepsány potomkem (viz `DejObsah()`), se rozlišuje, zda se jedná o abstraktní operaci. Pokud je operace označena jako abstraktní, potom ji nelze nikdy volat přímo na úrovni deklarace, ale musí být vždy přepsána zespodu. Jinak řečeno, abstraktní operace musí být povinně přepsána potomkem, aby mohla být použita. Tak trochu „od sekery“ řečeno, abstraktní operace zavede pouze hlavičku (do interfacu) proto, aby ji nějaký potomek převzal a přepsal. Abstraktní operace tedy slouží k zavedení operace v interfacu, ale její implementace je vždy přenechána až potomkovi. Všimněme si, že například `DejObsah()` je také na úrovni deklarace v `CObrazec` abstraktní operací.

Operace, která není abstraktní, ale je polymorfní, se někdy nazývá virtuální. Modelovací jazyk UML zavádí pro takovouto operaci pouze pojem polymorfní operace. Některé jazyky (a některé nikoliv) mají všechny dynamické operace (operace instancí - objektů ze třídy) zavedeny jako virtuální automaticky, u některých musejí být virtuální operace deklarovány přímo pomocí rezervovaného slova (nejčastěji pomocí slova `virtual`).

Pozor na jednu záludnost: Obecně nemusí platit, že abstraktní třída musí obsahovat všechny svoje operace jako abstraktní! Znamená to, že i v abstraktní třídě se může vyskytnout nějaká operace, která není abstraktní. Například nějaká operace má své definované a implementované chování v metodě již na úrovni abstraktní třídy. Dokonce tato operace ani nemusí být polymorfní (virtuální), protože se její „způsob chování“ od potomka k potomkovi nemění a všichni ji tedy volají zespodu jako „hotovou metodu“.

Zvláštním případem je taková abstraktní třída, která nemá žádný atribut, žádnou objektovou referenci (žádnou asociaci a agregaci) a všechny její metody jsou abstraktní. Jedná se o „čistou abstraktní třídu“. Taková třída zavedla pouze množinu operací (dokonce bez implementace) a nic víc. Programátorsky řečeno, takováto čistá abstraktní třída zavedla pouze „hlavičky“ operací. Tato třída se nazývá interface. Interface je tedy prázdnou čistou abstraktní třídou s abstraktními operacemi.

V UML se interface značí jako třída pomocí stereotypu <<*interface*>>:

obrázek 6 Zavedení interfacu

Pokud nějaká třída získá nový interface, znamená to, že z něj de facto podědí a poděděné abstraktní operace přepíše do svých metod (implementuje tyto abstraktní operace). Některé jazyky pro tento postup „podědit z interfacu a přepsat abstraktní metody interfacu“ zavedly ustálený pojem „implementace interfacu“. Tento pojem se promítl i do syntaxe těchto jazyků rezervovaným slovem `implements`. Jedná se však opět o „skryté dědění“, v tomto případě dědění od čisté prázdné abstraktní třídy s abstraktními operacemi. Většinou jazyky umožňují několikanásobnou implementaci interfaců (tj. několikanásobné podědění od čisté abstraktní třídy s abstraktními operacemi).

V dalších kapitolách jsou uvedeny a vysvětleny jednotlivé vzory.

ABSTRACT FACTORY

Klasifikace

Object

Creational

Smysl

Zavádí interface pro tvorbu rodiny souvisejících nebo závislých objektů, přičemž klient je izolován od volání těchto tříd.

Alias

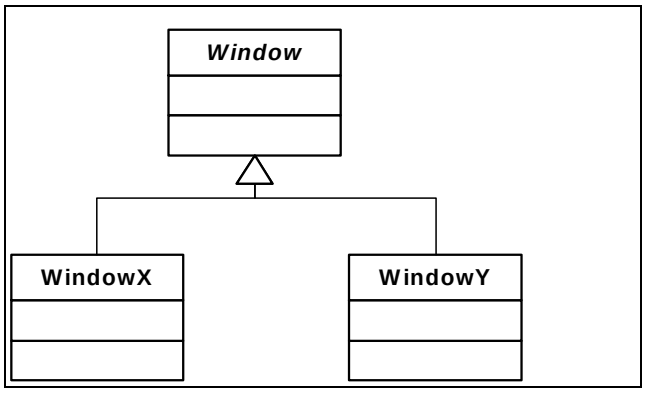
Kit

Motiv

Představme si, že budeme pracovat s nějakým prvkem a zjistíme, že můžeme použít buď prvek z technologie X anebo prvek z technologie Y. Aplikace by měla umět pracovat s oběma prvky z obou technologií a to tak, že se například konfigurací z dat nastaví „budiž X“ a pracujeme s prvkem X, anebo se nakonfiguruje aplikace na „budiž Y“ a pracujeme s Y. Přepnutí je možné principiálně z toho důvodu, že oba prvky jsou dědicové nějakého abstraktního prvku, svého předka.

Například budeme v aplikaci zavádět dva standardy GUI, jeden jako X a druhý jako Y. Znamená to, že aplikace může zavést okno ze třídy standard X, označme jako `WindowX` anebo okno ze třídy standard Y, označme třídu těchto oken jako `WindowY`. Je pochopitelné, že v běhu programu pracujeme v GUI s prvky buď v jednom standardu anebo ve druhém standardu, avšak aplikace může pracovat s oběma, například přepínat standardy v běhu, měnit je při startu podle konfiguračních souborů apod.

Jeví se jako výhodné zavést společného předka abstraktní `Window`. Operace předka si potomci přepíší podle svého „standardu“:



obrázek 7 Dva standardy GUI, X a Y, reprezentované dvěma dědici Window

Potom při práci s oknem stačí ve správném bodě, například při konstrukci objektu, zvolit tu správnou třídu. Po tomto přepnutí, tj. volbě X nebo Y, bude již práce s oknem stejná jak v případě X, tak Y, protože podporují stejný interface získaný od třídy `Window`.

První (a nepříliš šťastné) řešení by mohlo spočívat v zavedení nějaké „hodnoty“ přepínače nějak takto (neuvádíme viditelnost, přepínač by měl být viditelný „odevšad“):

```
integer gPrepinac;
```

a do této „datové“ proměnné se umístí hodnota přepínače, například 0 pro X, 1 pro Y. Tato hodnota by se například načetla při čtení konfiguračního souboru anebo by ji mohla obsluha změnit v runtime.

Potom bychom volali konstruktor okna v sekvenci nějak takto:

```
...
Window MojeWindow;
switch gPrepinac
{
  0 : MojeWindow = new WindowX();
  1 : MojeWindow = new WindowY();
}
```

(zavedli jsme na první pohled srozumitelnou pseudosyntaxi pro switch - větvení kódu podle hodnoty.)

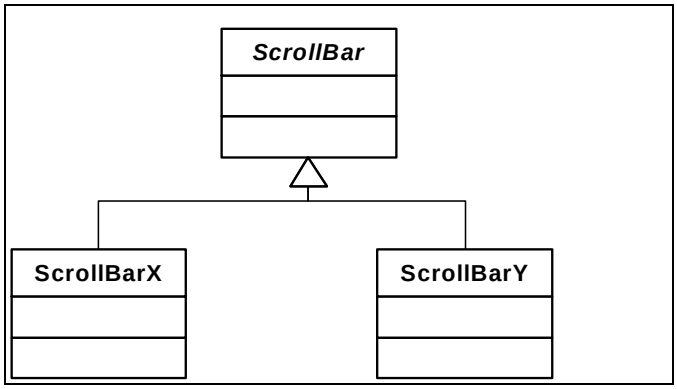
Další kód bude již společný pro oba typy oken, pracuje se totiž s interfacem `Window`.

Již na první pohled je zřejmé, že tento kód větvení by bylo určitě třeba „umístit někam centrálně“, abychom jej nepsali tolikrát, kolikrát budeme konstrukci objektu volat. Pokud bychom chtěli použít strukturované programování (a mohli, například Delphi nebo C++), asi bychom na řešení této situace použili funkci. Strukturálně pojaté řešení by tedy vypadalo takto:

- globálně viditelná proměnná `integer gPrepinac`
- globálně viditelná funkce, například `FunkceCreateWindow()`
- klient obsluhuje `gPrepinac`
- klient volá pro nové okno funkci nějak takto: `MojeWindow = FunkceCreateWindow()`

Podobně by bylo možné řešení, kdy by se proměnná přepínače stala vstupním parametrem této funkce. Ta by pak měla tvar `FunkceCreateWindow(integer a_Prepinac)`. Avšak v každém případě bychom si museli přepínač do nějaké globální proměnné „trvale“ uschovat, abychom pracovali stále nad stejným standardem.

Avšak neexistuje pouze prvek GUI okno, reprezentovaný třídou `Window`. Existují i další prvky, například `ScrollBar`, a jiné GUI prvky. I pro ně se zavede podobná konstrukce abstraktního předka:



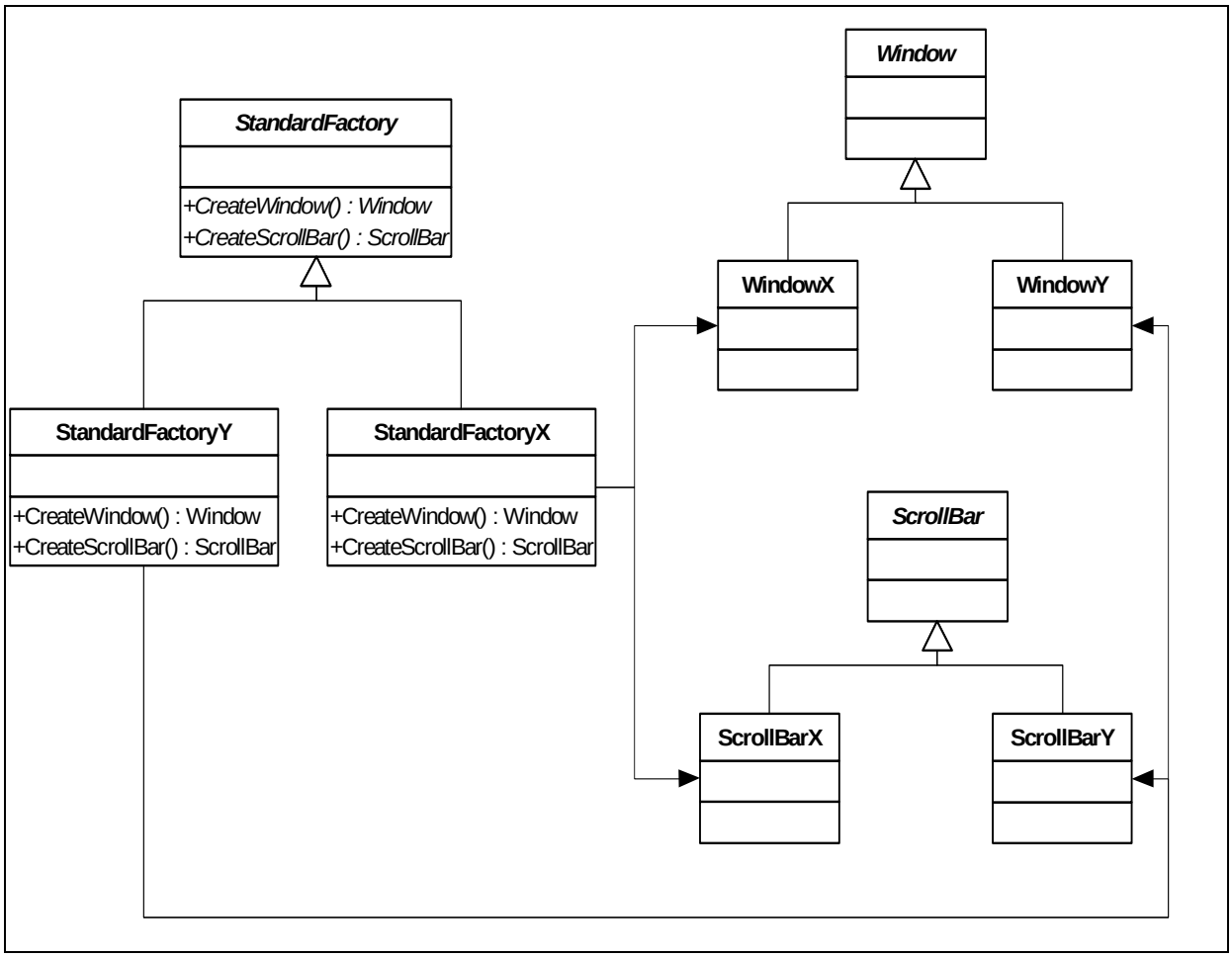
obrázek 8 Abstraktní třída pro další prvek GUI

Znamenalo by to zavést i pro tyto prvky funkce zrodu, například `FunkceCreateScrollBar()` apod.

Vzor zvaný ABSTRACT FACTORY doporučuje nezavádět obdoby volání funkcí, ale zavést přímo „přepínač“ pomocí polymorfního volání interfacu objektu Factory.

V tomto našem příkladu u dvou standardů GUI X a Y se nejprve zavede abstraktní třída `StandardFactory` a pro ni dva dědicové `StandardFactoryX` a `StandardFactoryY`.

Třída `StandardFactory` zavádí interface jako abstraktní operace pro získání nových prvků. Nazvěme tyto operace (podobně jako u funkcí) jako `CreateWindow()` a `CreateScrollbar()`. Dědicové tyto operace přepisují tak, že operace vracejí odpovídající typy daného standardu X anebo Y. Tuto složitou větu můžeme znázornit vysvětlujícím obrázkem:



obrázek 9 Zavedení interfacu Abstract Factory pro tvorbu GUI prvků

Myšlenka je velmi jednoduchá: Třída `StandardFactory` zavádí interface, který obsahuje (zde) dvě abstraktní operace `CreateWindow()` a `CreateScrollBar()`. Typy návratových hodnot těchto operací jsou deklarovány po řadě `Window` a `Scrollbar`, tedy jako „společné“ typy předků prvků GUI. Třída `StandardFactory` má dva dědice, třídu `StandardFactoryX` a třídu `StandardFactoryY`, které tyto operace přepisují a implementují je. Například objekt ze třídy `StandardFactoryX` vrací při volání operace `CreateWindow()` prvek ze třídy `WindowX` (prvního dědice `Window`), objekt ze třídy `StandardFactoryY` vrací prvek pro tutéž operaci objekt ze třídy `WindowY` (druhého dědice `Window`). Přerušované čáry se šipkami ukazují, které Factory rodí které skupiny objektů.

Jedna z možných implementací metody `CreateWindow()` (pozor, nikoliv jediná možná!) by mohla v našem pseudojazyce vypadat takto:

```
class StandardFactoryX : StandardFactory;
 {
    public override Window CreateWindow();
    {
        return new WindowX();
```

```
    }
}
```

Poznámka: V některých jazycích není rezervované slovo `override` povinné, v naší syntaxi je budeme mnohdy uvádět proto, abychom zdůraznili, že metoda přepisuje existující metodu předka (v tom je podstata vzoru). Navíc dvojtečka v deklaraci třídy za jejím názvem označuje dědění. V Object Pascalu se zavádí pomocí závorek.

Volba „rodiny“ objektů GUI daného standardu je takto převedena na volbu objektu „továrny“ buď ze třídy `StandardFactoryX` (vrací prvky standardu X), nebo ze třídy `StandardFactoryY` (vrací prvky ze třídy Y). Volba objektu je tedy volbou „továrny“ této rodiny, název továrna je tedy opravdu výstižný :) .

Můžeme si představit, že dva objekty `Factory` pro oba standardy jsou umístěny do kolekce, nazvěme tento objekt jako `ManagerFactory`. Jedná se o kolekci, jejímiž prvky jsou objekty podporující interface `StandardFactory` a jsou přidávány do kolekce spolu s klíčem (například `integer`), přes který lze „vytáhnout“ zpět (podle klíče) odpovídající `Factory`. Kód pak může vypadat v pseudosyntaxi nějak takto:

```
...
//init manager...
MojeStandardFactoryX = new StandardFactoryX();
MojeStandardFactoryY = new StandardFactoryY();
MujManager.Add (MojeStandardFactoryX, key = 1);
MujManager.Add (MojeStandardFactoryY, key = 2);
...
// v idFactory je 1 nebo 2, například vybrala obsluha...
StandardFactory MojeFactory  = MujManager.GetItem(idFactory);
...
// kód je dále nezávislý na výběru Factory:
...
MojeWindow = MojeFactory.CreateWindow();
...
MujScrollBar = MojeFactory.CreateScrollbar();
...
```

Pokud toto řešení porovnáme se strukturálním, vidíme, že jsme přepínání v datové proměnné převedli na „přepínání ve výběru“ objektu `MojeFactory`. Několik funkcí typu create jsme převedli na množinu jeho operací tj. na interface objektu `Factory`. Toto objektové řešení se vyznačuje vyšší konzistencí, je systematičtější a určitě přehlednější (a v určitém směru vybaveno vyšší flexibilitou – lze například relativně snadno zavést nový standard Z jako nového dědice Factory).

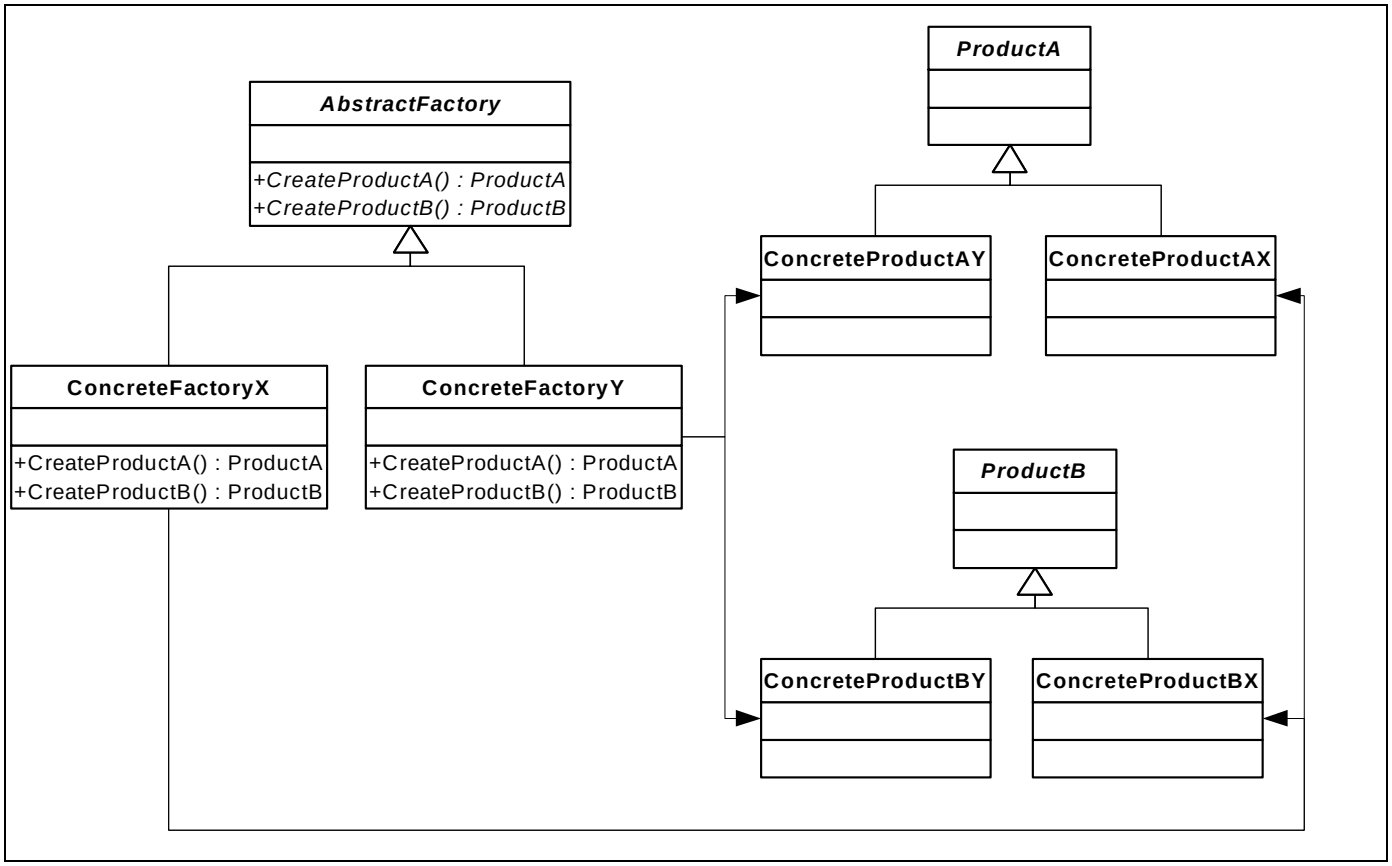
Jedinou otázkou zůstává, jak v OOP (Java a C#) zabezpečit, aby vybraný objekt `MojeFactory` byl vždy přístupný sdíleným způsobem z libovolného bodu programu, protože je třeba, aby byl pouze jeden „globálně“ viditelný. Tento problém řeší vzor zvaný SINGLETON.

Struktura vzoru a popis vzoru

V předešlé kapitole jsme již možná pochopili, jak vzor funguje, ale je třeba si uvědomit, že se jedná pouze o „jedno použití vzoru“ konkrétně se zabývající pouze případem výměny „standardů GUI prvků“.

Nyní si vzor zobecníme, tj. již v něm nebudou vystupovat konkrétní prvky konkrétního použití, ale účastníci, tedy role.

Strukturu vzoru ukazuje následující obrázek:



obrázek 10 Struktura vzoru Abstract Factory

Z obrázku vyplývá význam účastníků tohoto vzoru: Zobecnění jsme provedli tak, že jsme prvky GUI nahradili obecně pojmem `Product` a „továrny factory“ jsme zobecnili do stromu `Factory`, kde jsme zavedli roli předka `AbstractFactory` (odtud název vzoru ABSTRACT FACTORY) a role potomků (`ConcreteFactory`), kteří implementují metody pro tvorbu objektů.

Spolupráci ve vzoru bychom mohli definovat tak, že `AbstractFactory` zavádí interface (množinu operací) pro tvorbu objektů. Dědicům přenechává v implementaci volbu „rodiny typů produktu“. Klient pro tvorbu objektů nevolá jejich konstruktory, ale používá pouze operace interfacu definované v `AbstractFactory`.

Poznámky k použití

Odstínění, přepínání rodin objektů dynamicky, konzistence rodin

Tento vzor se jeví jako výhodný tehdy, když potřebujeme odstínit volání konstruktoru (flexibilita výměny produktů) a navíc pracujeme s „rodinami“ objektů, které potřebujeme „přepínat“ (viz kapitola Motiv). Nejenom, že klient je odstíněn od rozhodování, jakou třídu u kterého produktu volit (je odstíněn přes interface `Factory`), ale v tomto případě dojde k jednomu „nárazovitému přepnutí“ jednou výměnou konkrétního `Factory` za jiný.

Můžeme tím zamezit efektu práce „miš-maš“ s rodinami produktů a zabezpečíme tím konzistenci práce s právě jednou rodinou objektů. K této výměně navíc může dojít dynamicky v běhu programu.

Použití ve frameworks

Vzor ABSTRACT FACTORY se velmi často používá u produktů typu framework, u dodávaných knihoven apod. Klienti nevolají přímo zrody objektů z těchto knihoven, ale jsou odstíněni od tohoto volání a používají nabízené interfacy různých `Factory`.

Nevýhoda obtíže s novým produktem

Nevýhodou konkrétně tohoto vzoru jsou určité obtíže při rozšíření o nový produkt. Pokud totiž dodáme nový produkt (viz obrázek struktury vzoru), například `ProductC` (na obrázku není uveden), potom musíme do všech dědiců `Factory` přidat novou operaci a přepsat ji u všech dědiců.

Nevýhoda u málo rozdílných rodin

I když se dvě rodiny liší jen velmi málo, musíme zavést novou konkrétní třídu `Factory`. Množina operací je totiž „nedělitelná“, nelze vyměnit pouze jednu operaci, musíme zavést nového dědice a implementaci operace v něm změnit.

PROTOTYPE

Zajímavou alternativou k přepisování metod je použití vzoru PROTOTYPE (uvedeme v příslušné kapitole). Tento vzor řeší právě problém málo rozdílných rodin uvedený v předešlé kapitole.

Globální viditelnost

`Factory` v instancích jsou většinou zavedeny jako „odevšad viditelné objekty“, tedy jako SINGLETONY. Důvod je nasnadě: Objekt `Factory` určuje rodinu objektů ke zrodu a tuto informaci je třeba „držet“ po celou dobu její potřeby, je třeba ji „vidět odevšad“ a „stále“.

Definice extensivních factory

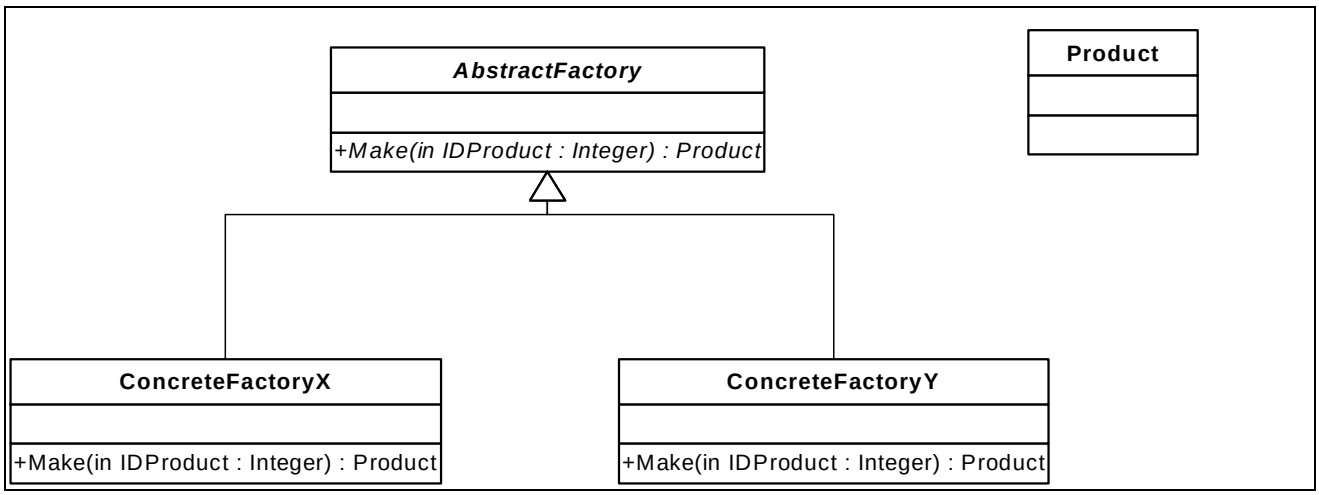
V uvedené struktuře vzoru ABSTRACT FACTORY (viz obrázek 10 Struktura vzoru Abstract Factory) dochází k volbě požadovaného produktu přes název operace interfacu `AbstactFactory`. Například pokud chceme `ProductA`, voláme operaci `CreateProductA()` s návratovou hodnotu typu `ProductA`.

Určitou modifikací tohoto vzoru jsou „extensivní flexibilní factory“, které nerozlišují volání požadavku na nový produkt názvem operace, ale vstupním parametrem, což může být `integer`, `string` apod. jako „klíč“ tohoto produktu. V interfacu `Factory` tak existuje pouze jedna operace (interface se „scvrkl“ na jednu operaci), například se nazývá `Make` nebo `Create`, která má nějaký vstupní parametr identifikující produkt, například nějak takto: `Make(string NazevProduktu)` apod. Musí však být v tomto případě splněn požadavek, že návratová hodnota této operace je vždy objekt se stejným interfacem. Jinak řečeno, všechny produkty musí patřit do jedné rodiny s jedním předkem definující interface návratového typu této operace. V tom je určité omezení tohoto přístupu.

Pokud potřebujeme pracovat se specifikou daného produktu, která není definovaná v tomto obecnějším interfacu, musí se provést již zmíněný *downcast.*

Poznámka: Obecně platí, že downcast je „nebezpečná“ operace v tom smyslu, že nemusí zaručit tzv. typovou bezpečnost. Program v tom případě vyvolává výjimku v runtimu na nekompatibilitu typů, která je jinak zabezpečována již při kompilaci (platí pouze u typových statických jazyků jako je JAVA, C# nebo Delphi. Dynamické netypové jazyky jako FOXPRO nebo VB script apod. jsou takto nebezpečené vždy, ale zase mají vyšší flexibilitu). Problém nebezpečí je v tom, že v instanci může být „jiný dědic“, než jaký je čekáván.

Druhá poznámka: Uvedené řešení se vstupním parametrem vypadá podobně jako řešení se vstupním parametrem u funkcionálního řešení, které jsem uváděli hned v úvodu. Ale pozor, zde vstupní parametr odlišuje produkty (tj. metody pro CreateWindow(), CreateScrollBar() atd.), kdežto u uvedeného funkcionálního řešení vstupní parametr odlišoval rodiny produktů. Znamená to, že bude existovat opět jedno Abstract Factory definující jako předek interface a několik konkrétních Factory, přičemž operace bude jenom jedna. Porovnejte následující obrázek se strukturou vzoru. Rozdíl je „pouze v tom“, že interface tvořený ve vzoru několika operacemi se zmenšil na jednu a názvy operací se převedly na hodnotu vstupního parametru IDProduct. Na obrázku je ještě zdůrazněna existence další třídy „navíc“ a tou je třída Product zavádějící společný interface pro produkty.



obrázek 11 Volba produktu nikoliv názvem operace, ale vstupním parametrem

Uvedené řešení pomocí vstupního parametru je tedy omezeno požadavkem na existenci společného návratového typu pro všechny produkty. Na druhé straně je možné zavést flexibilně extensivní přidávání produktů klasickým postupem přepisování metod dalšími novými dědici `Factory`.

Představme si, že původně existují pouze dva produkty `ProductA` a `ProductB`. Interface `Factory` pomocí `Make()` vrací buď `ProductA` nebo `ProductB` podle hodnoty vstupního parametru (typ návratové hodnoty je obecnějšího typu `Product`).

V pseudojazyce bychom mohli tuto metodu pro jedno `Factory` zavést například takto:

```
class ConcreteFactoryX : AbstractFactory;
```

```
{
   public override Product Make(integer IDProduct);
   {
       switch IDProduct
      {
       1 : return new ProductAX;
       2 : return new ProductBX;
      }
   }
}
```

Namísto dvou operací `CreateProductA()` a `CreateProductB()` máme tedy jednu operaci `Make()` a uvnitř ní je (v tomto případě) přepínač `switch`. Pokud se přidají nové produkty `ProductCX` a `ProduktDX`, potom můžeme zavést nového dědice uvedeného `ConreteFactoryX` a přepsat `Make()` „ještě jednou“ takto:

```
class ConcreteFactoryX2: ConcreteFactoryX;
 {
   public override Product Make(integer IDProduct);
   {
       switch IDProduct
      {
       3 : return new ProductCX;
       4 : return new ProductDX;
       else return super.Make(IDProduct);
       }
 }
```

Poznámka: volání `super.operace` v našem pseudojazyce značí volání operace předka.

Nejprve se tedy určí, zda klient chce produkt C nebo D. Pokud se zjistilo, že nikoliv, přenechá se rozhodnutí `Make()` na předkovi. I když je teoreticky možné zvolit v pořadí výběru nejprve na předkovi a potom „moje“, doporučuje se nejprve umístit vlastní rozhodování a teprve poté se obrátit na předka. Zvýší se tím flexibilita díky možnosti přepsat a tak změnit již existující pravidlo zavedené v předkovi.

Vzor FACTORY jako klon ABSTRACT FACTORY

Někdy se uvedená odrůda vzoru se vstupním parametrem (viz předešlý odstavec) zavádí, aniž by se vůbec požadovalo přepínat rodiny objektů. Prostá skutečnost, že se nevyžaduje přepínat rodiny objektů má jednoduchý důvod - pracuje se pouze s jednou rodinou a víc než jedna rodina se v řešení nevyskytuje (není co přepínat).

V takovém případě můžeme tento vzor nalézt v literatuře pod názvem FACTORY (nikoliv ABSTRACT FACTORY). Tento „klon“ vzoru potom slouží pouze k odstínění volání konstruktoru a je zaveden jako pomocný objekt `Factory` ke stromu generalizace - specializace tříd. Namísto volání konkrétního konstruktoru třídy se volá operace `Make()` se vstupním parametrem ID produktu. Návratovým typem je typ určený vrcholem stromu dědičnosti. Vzor FACTORY je takto schopen obsloužit žádosti o nové objekty z tříd z libovolné pozice stromu dědičnosti převedením volání na parametr, což je oproti přímému volání konstruktoru (přímé oslovení třídy) flexibilní a dynamické.

Pro ilustrativní a pěkný příklad použití tohoto klonu vzoru s názvem FACTORY bychom nemuseli chodit příliš daleko. Vzpomeňme na příklad s obrazci. V jedné chvíli jsme upozornili na „ne-flexibilitu“ při volání požadavku na zrod obrazce, kdy například v MenuItemu se přímo specifikuje volání konstruktoru a k čemu to vede. Pokud zavedeme FACTORY, jednak „centralizujeme“ volání konstruktoru a také odstiňujeme klienta od tohoto volání. To vede k flexibilnímu řešení. například k zavedení dynamicky tvořeného menu a tedy rozšiřitelného menu bez zásahu do jeho struktury a kódu.

Zavedeme třídu `CObrazecFactory` a zavedeme operaci `MakeObrazec()` se vstupním parametrem ID, která vrací podle čísla ID různé typy obrazců. Návratovým typem této operace je `CObrazec`. Takto jsme použili vzor FACTORY, klon vzoru ABSTRACT FACTORY.

V MenuItemu se poté vždy volá jeden „stejný“ kód:

```
...
CObrazec MujNovyObrazec = MojeFactoryObrazcu.Make(ID);
...
```

což v důsledku vede k možnosti parametrizovat vznik menu pomocí instancí hodnot číselníků.

Zavedeme-li číselník například takto:

kód	text
1	Nový trojúhelník
2	Nový obdélník
3	Nový kruh
4	Nový složený obrazec

Mohli bychom Menu vytvořit dynamicky. Do textů MenuItemů se načtou z tohoto číselníku texty a prvky menu se datově hodnotu prováží (například přes property `TAG` anebo podle pořadí) na odpovídající prvky tohoto číselníku. Po vybrání MenuItemu se identifikuje, jaká hodnota kódu byla vybrána a volá se `MojeFactory.Make(kód)`

Poznámka: Setkal jsem se i s chybným návrhem FACTORY, kdy se funkcionalita `Make()` umístila jako statická operace třídy do vrchního stromu dědičnosti. Zdůvodněním tohoto chybného postupu je snaha umístit „vše včetně FACTORY do jednoho stromu“. Nedoporučuje se takto činit ze dvou důvodů: Jednak se provede vzájemné cirkulární provázání předka a potomků jak zespodu nahoru (to je v pořádku), tak shora dolů, tj. v předkovi se objevují závislosti na entitách jeho potomků, což je chybou.

Druhým důvodem, proč se tomuto vyhnout, je skutečnost, že statické operace nebývají v těchto jazycích zaváděny jako polymorfní a nelze je tedy přepisovat dědici. Doporučuje se zavést FACTORY s operací `Make()` úplně bokem od stromu produktů jako zvláštní strom tříd.

Související vzory

ABSTRACT FACTORY je realizován pomocí FACTORY METHOD (přepisování metod, viz dále) resp. se používá PROTOTYPE jako konkurenční vzor.

ABSTRACT FACTORY se většinou zavádí jako SINGLETON (viz dále).

BUILDER

Klasifikace

Object

Creational

Smysl

Odděluje konstrukci složitého objektu (tj. postup jeho tvorby) od jeho reprezentace (tj. od jeho konkrétního složení).

Motiv

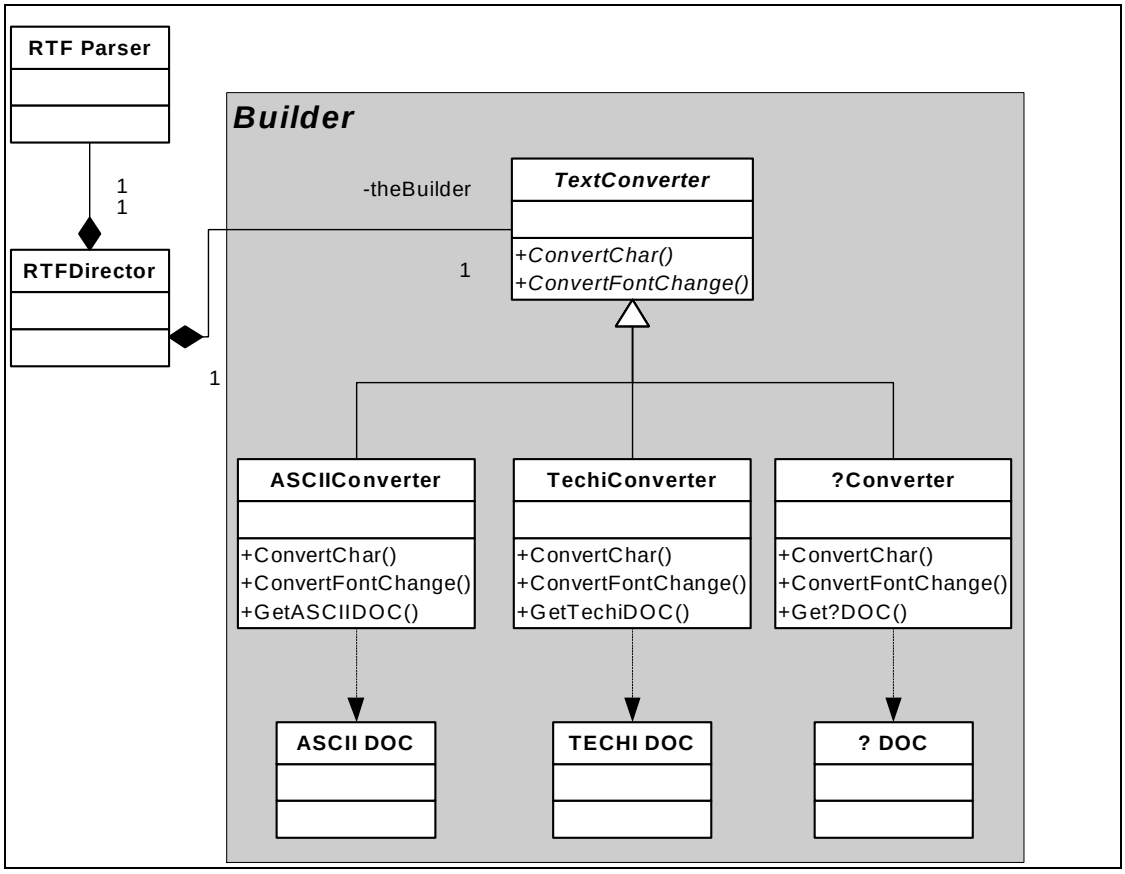
V některých případech je třeba „budovat“ relativně složité objekty složené z jiných objektů. Například může se jednat o složené obrazce (viz kapitola o polymorfismu), nebo se jedná o poskládání objektů při nějaké hře (bludiště, hrací plán apod.). Klient „skládá“ složitý objekt pomocí jeho částí. Je otázkou, o co se má vlastně klient starat (co vidí a co používá) a o co se již starat nemá (od čeho je odstíněn).

Vzor BUILDER navrhuje jednoduché a opětovně použitelné řešení jako odpověď na tuto otázku „co klient má vidět a od čeho je odstíněn“. Uvedeme si dva klasické příklady motivu uváděné v literatuře.

V literatuře (např. [5]) se používá jako příklad pro motiv BUILDERU „konverze dokumentu z jednoho určeného typu (například RTF) do několika jiných typů“. V tomto případě existuje nějaký Parser čtoucí a „louskající“ RTF dokument a existuje nějaký další objekt dokumentu, který je tvořen, například textový (ASCII) dokument nebo HTML dokument nebo jiný typ dokumentu. Dochází k výstavbě výsledného dokumentu na základě toho, co Parser v RTF „rozlouskl“. V prvé řadě se jeví jako výhodné odstínit samotný problém čtení a „louskání“ (parsování) RTF dokumentu od stavby výsledného objektu. To je sice zřejmé, ale pozor, existuje ještě „druhá“ úroveň odstínění.

V tomto případě se ukazuje výhodné ještě navíc oddělit samotný problém výstavby (co se má stavět) od samotné konstrukce a reprezentace objektu, který je stavěn. Jinak řečeno, klient, který staví složitý objekt posílá své požadavky „nějakému interfacu“ s požadavky „co si přeje přistavět“, avšak samotná výstavba je schována až za tento interface. V našem příkladu s RTF dokumentem tento přístup funguje tak, že při získání výsledku od Parseru RTF (například změna fontu) se tento požadavek posílá nikoliv výslednému dokumentu (rovnou do objektu HTML dokumentu), ale nějakému interfacu, který jej převezme a za ním teprve dojde k aplikaci toho požadavku výstavby. Klient tohoto interfacu se nazývá Director a interface těchto požadavků stavby je zaveden pomocí abstraktní třídy Builder. Director v našem případě převezme výsledek od Parseru RTF a zavolá odpovídající operaci interfacu Builder a přitom *Director neví a ani jej nezajímá, o jakou reprezentaci objektu za interfacem Builderu se jedná.* Až jsou požadavky na konstrukci ukončeny, požádá se Builder o vydání poskládaného objektu.

Výsledný návrh překlápění dokumentů může potom vypadat nějak takto:



obrázek 12 Builder pro stavbu dokumentu

Na obrázku je znázorněna spolupráce mezi objektem typu `RTFDirector` a objektem `theBuilder`. `RTFDirector` vidí pouze interface zavedený `TextConverterem` (abstraktní třída zavádějící operace výstavby), ale neví, „kdo se za tímto interfacem skrývá“ (ani jej to nezajímá) . Volá vždy pouze zavedené operace stavby a „víc nevidí“. Kompatibilní záměnou (může být dokonce v runtime) jednotlivých konkrétních builderů pro různé typy dostaneme konverzi do různých typů dokumentů. Na konci konstrukce se požádá Builder o vydání poskládaného dokumentu, což je operace již specifická pro daný typ konvertoru (není součástí interfacu obecného konvertoru).

Co bývá matoucí u tohoto vzoru je ta skutečnost, že výsledné složené objekty, zde dokumenty ASCII DOC, TECHI DOC a nějaký další, nemusí povinně patřit do společné rodiny tříd a nemusí mít stejný interface. Obecně vzato tyto výsledně složené objekty nepatří do jedné rodiny stromu dědičnosti (pozor - Builder oproti tomu ano). Mimochodem tato skutečnost vede k zavedení speciální operace k navrácení výsledku na úrovni potomka a nikoliv k zavedení nějaké obecné abstraktní operace návratu výsledku na úrovni předka.

Matoucí je to, že v některých případech mohou z povahy věci výsledné skládané objekty patřit do společného stromu a mít zavedeného společného předka (což není zrovna tento případ!). Pak je teoreticky možná konstrukce taková, že interface builderu patří přímo do interfacu abstraktního předka skládaných objektů. Tento interface „buildování“ je zaveden přímo v této rodině, tedy builder je součástí chování samotných skládaných objektů. Potom se každý člen rodiny umí rovnou a přímo polymorfně „buildovat“ podle svého algoritmu bez „nové“ pomocné třídy Builderu. Uvedený přístup

bychom mohli chápat jako klon vzor Builderu v klasifikaci „class pattern“. Většinou bývá interface pro sestavení objektů zaváděn jako „primitivní“ chování pro skládání u kompatibilních objektů, má však určitou odlišnost od zde zavedeného vzoru BUILDER. BUILDER totiž zavádí objektovou vazbu přes interface a nikoliv statickou vazbu realizovanou děděním. Jinak řečeno vzor BUILDER zavádí třídu s operacemi a teprve za ní je v konkrétní implementaci zavedena interakce objektové vazby na výsledný skládaný objekt. Nevyžaduje se tak žádná dědičnost a příbuznost mezi produkty. Tím se v mnoha případech zvyšuje flexibilita, protože pak není třeba po výsledném objektu požadovat žádnou kompatibilitu se svými druhy, tedy s jinými skládanými objekty. „Kdovíco je za interfacem Builderu...“

Poznámka: Požadavek na kompatibilitu mezi výslednými objekty však může vyplynout z něčeho jiného, než je skládání objektu.

Tuto rozdílnou filosofii si vysvětlíme na druhém jednoduchém příkladu. Představme si, že chceme vytvořit GUI prvek typu „výběr z textů“. Může se jednat například o výběr typu Combobox, Listbox nebo sada Radiobuttonů apod. Vytvořme následující konstrukci: Bude existovat společný předek obecného prvku typu výběr z textů, označme jej (například) jako `AbstractGUIStrList`. V této třídě zavedeme abstraktní operaci pro přidání nového prvku `AddString(string NewString)`, která má přidat nový text do seznamu zobrazovaných textů. Každý z konkrétních potomků, kterého zavedeme, tuto operaci přepíše „podle svého“. Například u prvku sady radiobuttonů s názvem `GUIStrRadiobuttons` přibude v této implementaci nový radiobutton s tímto textem. Je zřejmé, že jsme takto zavedli „budování“ těchto složených objektů přímo do jejich interfacu.

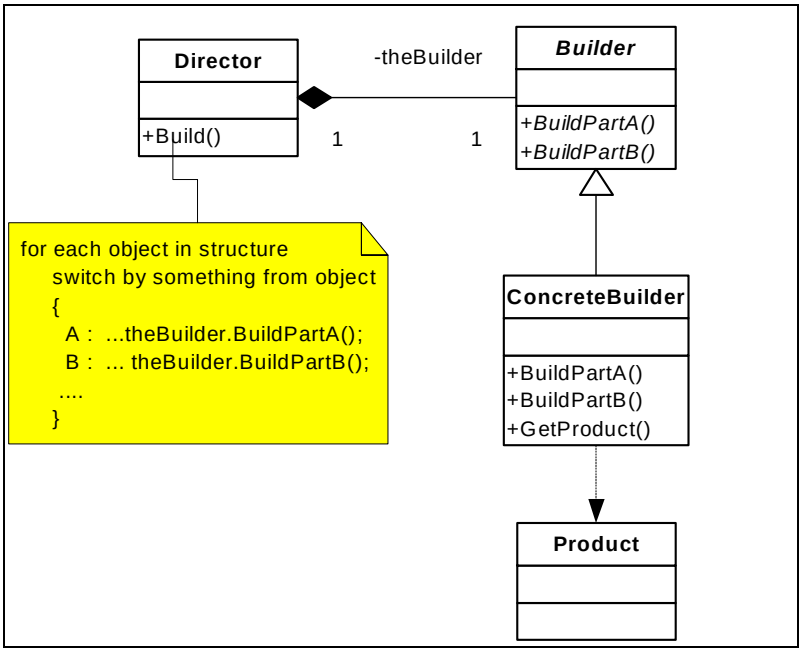
Vzor BUILDER však pracuje malinko odlišně. Jeho interface se zavádí „bokem“ od této struktury skládaných objektů a tím jej lze „předřadit“ před naše objekty. Tímto pracujeme s referencí na objekt a nikoliv s interakcemi typu dědění mezi třídami. Díky tomu nemusí být obecně požadována nějaká kompatibilita mezi těmito skládanými prvky. Tato kompatibilita pro „buildování“ se převedla na kompatibilitu mezi Buildery.

Je zřejmé, že pokud chceme dosazovat složené objekty za sebe (to však již není otázka tvorby složeného objektu!), potom se díky tomuto požadavku objeví nutnost kompatibility mezi složenými objekty. To by byl asi případ našeho příkladu s GUI prvky výběr textů, kde lze zvolit jeden ze tří a dosadit na jedno a totéž místo.

Jiná je však situace u dříve uvedeného příkladu s dokumenty. Nebudeme například potřebovat žádnou kompatibilitu mezi zavedenými typy dokumentů (nemá smysl ji hledat). V aplikaci se prostě přepíná volba výběru práce s typy dokumentů a to je vše.

Struktura a popis vzoru

Strukturu vzoru lze popsat následujícím obrázkem:

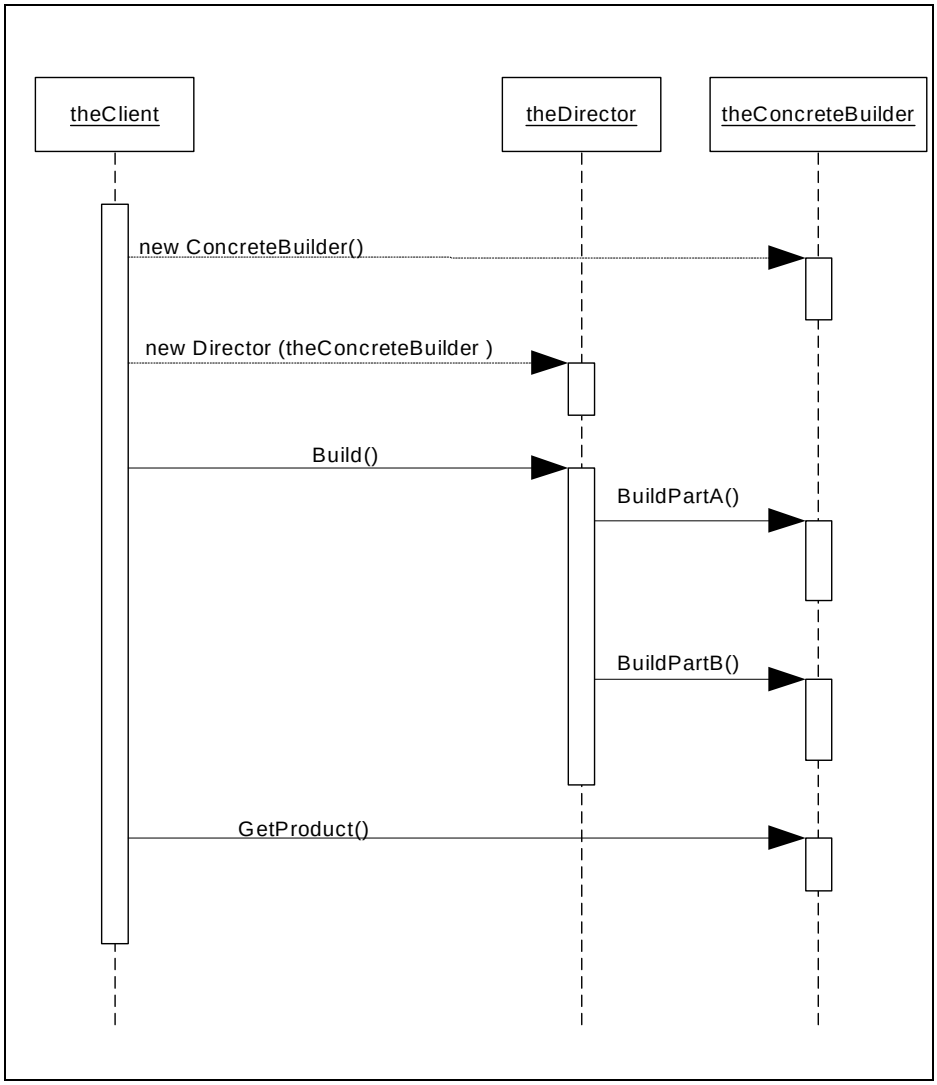


obrázek 13 Struktura vzoru BUILDER

Jednotliví účastníci vzoru (viz předešlý obrázek) jsou následující:

- `Builder` specifikuje abstraktní interface pro tvorbu („stavbu“) částí výsledného objektu.
- `ConcreteBuilder` implementuje tento interface do konkrétní podoby a to tak, že konstruuje konkrétní objekt daného `Produktu`. Navíc může ještě držet konzistenci stavby objektu, může definovat cestu „jak se konkrétně staví“ (a jak není povoleno stavět). Může definovat cestu a stav postupu této stavby („držet kursor stavby“). Zavádí interface (operaci) pro návrat výsledku.
- `Director` zkonstruuje skládaný objekt použitím volání interfacu `Builder`, aniž by viděl přímou reprezentaci skládaného objektu.
- `Product` reprezentuje skládaný konstruovaný objekt.

Pomocí sekvenčního modelu můžeme znázornit jeden z možných způsobů spolupráce těchto účastníků v celkovém scénáři:



obrázek 14 Spolupráce účastníků u vzoru BUILDER

Postup spolupráce znázorněný na obrázku je následující:

Klient vybere a požádá o nový konkrétní `Builder`, se kterým bude pracovat.

Poznámka: Na obrázku je tento výběr a zrod znázorněn přímo voláním konstruktoru (čárkovaně), tj. `new ConcreteBuilder()`*. Ale hodil by se sem objekt* `BuilderFactory` *s operací* `Make(integer IDBuilder),` *není-liž pravda :) ? Pak by se toto volání dalo flexibilně parametrizovat.*

Poté klient požádá o zrod samotného `Directoru` a „vnutí“ mu přímo v konstrukci jeho konkrétní `Builder` (konkrétní `Builder` podporuje interface předka `Builderu`). Poté klient zavolá operaci `Build` (to je konstrukce objektu), která volá jeho vnitřní „vnucený“ `Builder`. Na konci požádá klient o výsledný produkt voláním konkrétního `Builderu`.

Poznámky k použití

ABSTRACT FACTORY jako limitní případ BUILDERU

Pokud porovnáme oba probrané vzory, zjistíme, že mají něco společného. Vzor ABSTRACT FACTORY lze v určitém slova smyslu chápat jako „degenerovaný“ neboli limitní případ BUILDERU.

U vzoru ABSTRACT FACTORY volá klient určitou operaci `CreateProduct()` a ihned jako návratovou hodnotu dostává výsledek jako nový produkt. U vzoru BUILDER také klient žádá o nový produkt, ale žádost o výsledek je jakoby odložena na později (doslova: „...ještě si to nech, ještě jsme neskončili...“). Ve vzoru BUILDER se tedy konstruuje objekt „per partes“ nějakým celým scénářem volání několika operací.

Uschování vnitřní struktury budovaného objektu a odstínění tříd

`Director` je díky interfacu `Builderu` odstíněn od problematiky „jak vlastně objekt vypadá“. Nová reprezentace objektu (nový složený objekt se stejnou podstatou budování) znamená zavést nový konkrétní `Builder`.

Navíc klient vůbec nepoužívá konstruktory konkrétních tříd objektů skládajících výsledný objekt a dokonce je vůbec ani nemusí vidět. Volá se pouze požadavek na konstrukci (...`BuildPartX()`...) bez specifikace tříd. Současně je třeba připomenout vlastnost, že výsledné produkty nemusí vůbec patřit do společného stromu dědičnosti.

Úplnost interfacu a virtuální operace `Builderu`

Vzor je implementován tak, že se zavede abstraktní třída `Builder` a dědicové konkrétně implementují operace výstavby složených objektů. Pro některé typy složených objektů nemusí mít všechny operace význam, například pro konvertor do ASCII se u změny fontu neděje nic (na operaci změny fontu prostě nereaguje). Z toho vyplývají dva závěry:

- Jednak interface abstraktní třídy předka `Builder` musí být úplný ve všech operacích pro všechny operace všech konkrétních tříd `Builder` (úplný přes všechny `Buildery`). Pokud tomu tak není, některý z `Builderů` nemůže být plně funkční. Klient povinně nezná nic jiného, než je množina operací abstraktního `Builderu` a co tam nenajde, to nemůže použít.
- Druhým závěrem je skutečnost, že je výhodnější nevytvářet operace na úrovni společného předka jako abstraktní operace, ale pouze jako virtuální operace s prázdnými těly operací. Protože někteří potomci budou potřebovat implementovat pouze některé operace a některé z operací budou chtít ponechat s prázdnými těly. Stačí takto přepsat pouze potřebné operace a ostatní ponechat v implicitní podobě s prázdným tělem. Pokud bychom vytvořili všechny operace abstraktní, potom bychom donutili každého potomka přepsat všechny operace, a to i ty, které mají v této implementaci prázdné tělo.

Kontrola skládání

Zavedení vzoru BUILDER má ještě jednu výhodu, která souvisí s jeho strukturou: Poskytuje mnohem lepší kontrolu nad samotným procesem výstavby složitějších objektů. Pokud se nám podaří najít operace `Builderu`, potom jsme vlastně problém výstavby složitého objektu dekomponovali do sub-částí výstavby na abstraktní úrovni. Použití BUILDERU výrazně zpřehledňuje jinak složitá řešení skládání objektů.

Mezivýsledky stavby

V některých případech je třeba, aby klient získával mezivýsledky i během stavby. Například při stavbě bludiště se bude obsluha rozhodovat, kam umístit dveře v místnosti apod. V tom případě lze požádat konkrétní `Builder` o „mezivýsledek“ a pokračovat dále ve výstavbě. Je možná varianta, že neposkytne „celý“ výsledek jako mezivýsledek (nemusí se volat pouze `GetProduct()`), ale například se požádá o nějakou část produktu apod. (subnodes apod.). Někdy například potřebuje „něco v některém prvku vyplnit“ apod. Záleží na konkrétní situaci a konkrétním řešení. Důležité je, že nejsme omezeni pouze scénářem „buduj, buduj a na konci teprve uvidíš konečný výsledek“, ale můžeme obecně žádat o mezivýsledky během stavby.

Související vzory

Vzor BUILDER a ABSTRACT FACTORY jsou si podobné, liší se tím, že BUILDER buduje objekt „per partes“, ABSTRACT FACTORY buduje (i například složitější objekt) najednou.

Mnohdy bývá struktura zavedená pomocí COMPOSITE (viz příklad na složené obrazce) stavěna pomocí BUILDERU.

FACTORY METHOD

Klasifikace

Class

Creational

Smysl

Definuje interface (operaci) pro zrod objektu na své úrovni předka, ale ponechává potomkovi k rozhodnutí, jakého konkrétního typu (z jaké konkrétní třídy) bude tento rozený objekt pocházet.

Alias

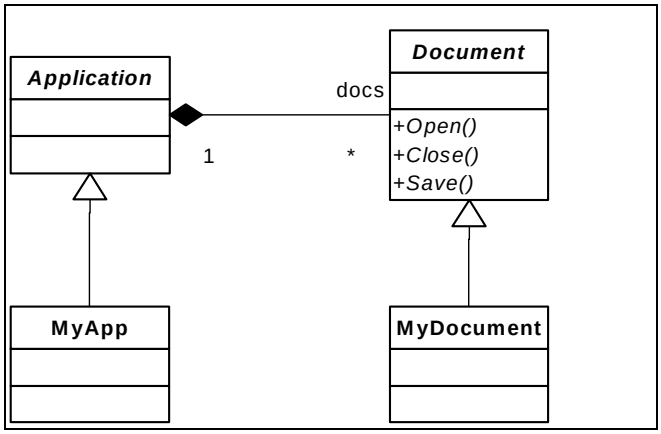
Virtual Constructor (virtuální konstruktor)

Motiv

Jedno použití FACTORY METHOD jsme již zavedli ve vzoru ABSTRACT FACTORY, pouze jsme v té chvíli nevěděli, že se jedná právě o využití tohoto vzoru. Vzpomeňme na interface zavedený v ABSTRACT FACTORY. Tento interface obsahuje operace `CreateProduct()`. Návratovými typy těchto operací jsou vrcholy stromu daných produktů (tj. třídy produktů na nejvyšší úrovni abstrakce stromu gen.-spec. produktů). Každá třída konkrétního Factory (tj. `ConcreteFactory`) přepisuje tyto operace tak, že dosazuje do návratové hodnoty „svůj“ typ jako dědice produktu. Všimněme si, že jsme provedli přesně to, co je popsáno ve smyslu vzoru FACTORY METHOD: Zavedli jsme operaci na úrovni předka (`CreateProduct()`), ale ponechali na potomkovi k rozhodnutí, jakého konkrétního typu je tvořený objekt. Nyní je jasná věta, že ABSTRACT FACTORY může být zaveden pomocí FACTORY METHOD, a to tak, že potomci přepisují operace `CreateProduct()`.

FACTORY METHOD zavádí operaci pro zrod objektu, ale potomci ji přepisují (odtud i druhý název *virtuální konstruktor*) a dosazují své konkrétní typy za návratové hodnoty.

Druhý motiv bychom mohli nalézt například v [5]. Představme si, že budeme vyvíjet produkt typu framework (nikoliv konkrétní „uživatelskou aplikaci“, ale sadu knihoven pro vývoj uživatelských aplikací). Bude se jednat o knihovnu pro tvorbu editorů. Bude existovat několik možných editorů podle typů dokumentů. Tuto skutečnost vyjádříme pomocí modelu tříd:



obrázek 15 Model tříd pro knihovnu editorů

Předešlý obrázek vyjadřuje skutečnost, že daný typ aplikace pracuje s daným typem dokumentu, například aplikace pro malování obrázků pracují dokumenty typu „dokumenty - malovátka“. Zavedení abstraktního předka `Application` a abstraktního předka `Document` umožňuje zavést efektivní re-use na vyšší abstraktní úrovni pro všechny typy editorů . Existují totiž scénáře, které jsou společné pro všechny typy aplikací a všechny typy dokumentů. Jako příklad takového scénáře můžeme uvést například následující scénář pro vznik nového dokumentu::

- zrodí se nový dokument
- přidá se mezi již existující dokumenty do seznamu `docs`, viz předešlý obrázek
- otevře se (metoda `Open()` bude přepsána podle typu dokumentu)

Pokud bychom tento slovní zápis scénáře zapsali do metod, mohla by být taková metoda definována na úrovni předka `Application` . Tato metoda provede popsaný scénář a vrací nový dokument (pozor, následující konstrukce obsahuje chybu!):

```
class Application;
{
 public Document NewDocument();
 {
  doc = new Document();
  docs.Add(doc);
  doc.Open();
  return doc;
 }
}
```

Snaha autora předešlého kódu je zřejmá: Rád by napsal scénář, který by se mohl volat zespodu pro libovolného potomka a dělal by tedy pro každého potomka vždy totéž: zrodil, přidal a otevřel dokument. Uvedený scénář by se tedy naprogramoval pouze jednou pro všechny potomky. Problém je v tom, že takto napsaný kód by korektně nefungoval (dokonce pokud by třída `Document` byla abstraktní, ani by se nedal spustit). Důvod je prostý: voláme v kódu vznik objektu ze třídy `Document` a přitom potřebujeme u potomka, tj. u konkrétní aplikace dokument odpovídajícího typu (na předešlém obrázku `MyApp` potřebuje, aby vznikl dokument typu `MyDoc`). V kódu se však vyskytuje volání

`doc = new Document()`.

Přepsat celou metodu `NewDocument()` však nechceme, protože tím naše snaha ztrácí smysl, celou tuto konstrukci děláme pro re-use.

Řešením je nevolat konstruktor přímo, ale zavést operaci, nazvěme ji například `CreateDoc()`, která bude tvořit nový dokument. Tato operace je definována na úrovni předka (tj. `Application`), potomek ji podědí a přepíše její obsah tak, aby vracela odpovídající požadovaný typ.

Kód se tedy změní tak, že volání `doc = new Document()` nahradíme voláním metody `CreateDoc()`, ostatní se nemění.

```
public Document NewDocument();
 {
  doc = CreateDoc ();
  docs.Add(doc);
  doc.Open();
  return doc;
 }
}
```

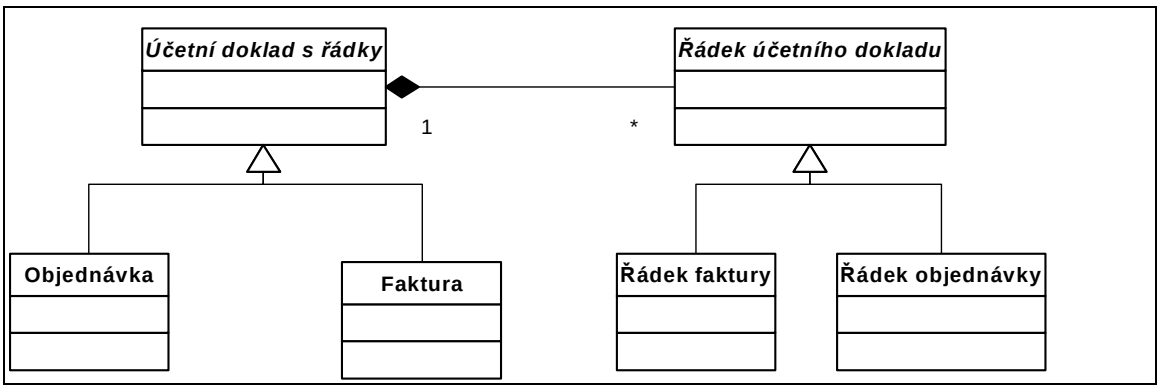
a potomek `MyApp` přepíše metodu `CreateDoc()` takto:

```
public override Document CreateDoc();
{
  return new MyDoc();
}
```

Metoda `CreateDoc()` je realizací vzoru FACTORY METHOD.

Poznámka: V uvedeném příkladu je uschován ještě jeden vzor, který budeme teprve brát a tím je TEMPLATE METHOD. Tento vzor zavádí scénáře na abstraktnější úrovni předků a potomci používají tyto scénáře tím, že přepisují metody v tomto scénáři, a tím tyto scénáře naplňují konkrétně.

Jako třetí podobný příklad (z praxe) uveďme následující: Při tvorbě stromu dědičnosti u účetních dokladů jsme dospěli k následující situaci:



obrázek 16 Příklad na Factory method z účetnictví

Význam tohoto diagramu je jasný: Jak objednávka, tak faktura je schopna mít řádky, každá svého typu řádku. Opět podobně jako u aplikace editorů můžeme zavést operaci pro nový řádek, která

- vytvoří nový řádek,
- přiloží ho do seznamu,
- vrátí jej jako návratovou hodnotu (pro okamžité použití, například editaci).

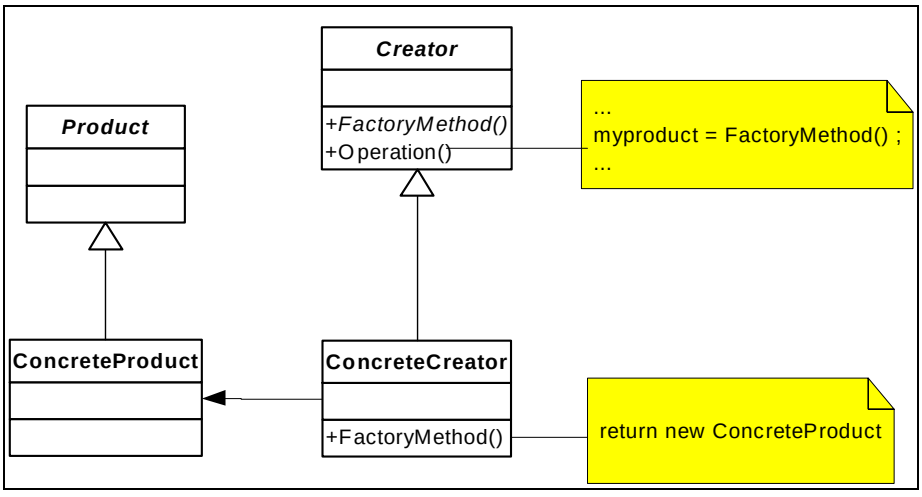
Vidíme, že situace je velmi podobná jako u předešlého příkladu a také problém, na který narazíme, bude stejný, a také se bude řešit stejně: Zavedeme na úrovni abstraktního předka „Účetní doklad s řádky“ operaci `CreateRadek()`. Volání této operace vložíme do scénáře operace `NewRadek()`.

Kód je takřka totožný jako v předešlém příkladu, pouze se nepracuje s aplikacemi a dokumenty, ale s doklady a řádky:

```
public Radek NewRadek();
 {
  mujradek = CreateRadek();
  docs.Add(mujradek);
  return mujradek;
 }
}
```

Struktura vzoru

Strukturu vzoru FACTORY METHOD vystihuje následující obrázek:



obrázek 17 Struktura vzoru FACTORY METHOD

Ve třídě `Creator` je definována operace `FactoryMethod()`, která vstupuje do scénáře operace `Operation()` v okamžiku, kdy má vzniknout požadovaný objekt. Třída `ConcreteProduct` přepisuje metodu `FactoryMethod()` tak, že dosazuje do návratové hodnoty odpovídající správný typ. Třída předek takto přenechává třídě potomkovi k rozhodnutí, jakého typu má být zrozený objekt. Vzor je klasifikace „class“ (na rozdíl od předešlých vzorů), protože ke spolupráci ve vzoru dochází na úrovni tříd děděním a nedochází k realizaci vzoru spoluprací „živých instancí - objektů“.

Poznámky k použití

Řešení situací, kdy „vrch ví, kdy rodit, ale neví, co rodit“

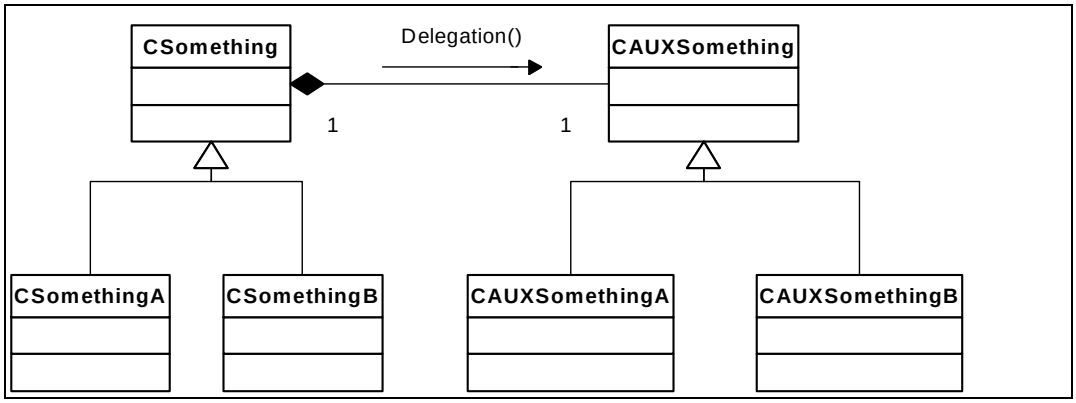
Tento vzor se obecně používá tam, kde třída předek sice „ví“, kdy je třeba nechat vzniknout objektu, ale neví (a ani nemůže vědět), jaký objekt má vlastně vzniknout (z jaké podtřídy). Pomocí polymorfního chování metody FactoryMethod se docílí požadovaného stavu, kdy potomek „už ví“ a proto může rozhodnout. Stručně řečeno, používá se velmi často ve scénáři, kdy „vrch ví, kdy rodit a neví, co rodit, a spodek ví, co rodit“.

Paralelní hierarchie stromů

Mnohdy nastane situace, kdy existují dva paralelní stromy vedle sebe propojené nějakou vazbou a přitom patří vždy „svůj ke svému“ v každém ze stromů. V našem motivu jsme narazili na dva takovéto případy, jednak motiv Aplikace versus Dokument a motiv Doklad versus Řádek dokladu. V tom případě lze opět použít tento vzor FACTORY METHOD.

Paralelní hierarchie u pomocných objektů

V některých případech dojde ke vzniku paralelní hierarchie díky tomu, že nějaký objekt deleguje část své funkcionality na „pomocný objekt“. Pokud existuje vztah, že nějaký objekt z jedné třídy má pomocný objekt z určité třídy, použije se opět tento vzor. Vznikají totiž opět paralelní dvojice tříd ze dvou stromů, například symbolicky znázorněné takto:



obrázek 18 Pomocné objekty a paralelní hierarchie

Poznámka: AUX zde značí zkratku jako „pomocné“.

Objekt ze třídy `CSomethingA` používá jako pomocný objekt ze třídy `CAUXSomethingA` a objekt ze třídy `CSomethingB` používá odpovídající pomocný objekt ze třídy `CAUXSomethingB`. Stromy

dědičnosti vyjadřují skutečnost, že tuto lze spolupráci zobecnit a lze tvořit abstraktní scénáře na vrchní úrovni. Avšak musíme opět řešit problém „svůj ke svému“. Nabízí se samozřejmě řešení pomocí vzoru FACTORY METHOD (zavedou se polymorfní operace zrodu objektů).

Má být třída `Creator` abstraktní?

Pokud se podíváme zpět na příklad s aplikacemi a dokumenty, můžeme si představit, že jedna z aplikací se svým typem dokumentu může být implicitní aplikace. V tom případě by nemuselo být výhodné tvořit vrchní třídu jako abstraktní. Nahoře by existovala jedna (pro instance použitelná, tj. konkrétní) implicitní třída (již se svou implementací) a ostatní dědicové by ji přepisovali.

Na druhé straně, pokud se podíváme na příklad s účetními doklady, zjistíme, že v tomto případě bude existovat abstraktní třída účetních dokladů a ostatní dědicové jako konkrétní třídy zavádějí konkrétní implementace.

Odpověď na otázku „`Creator` abstraktní ano nebo ne“ je tedy zřejmá: Záleží na případu a je třeba vždy posoudit, o kterou z variant se jedná.

ABSTRACT FACTORY a FACTORY METHOD

Vraťme se v úvahách k předešlému vzoru ABSTRACT FACTORY. Nyní je jasné, že pokud zavedeme tento vzor tak, jak je uvedeno na příslušné struktuře vzoru, tj. pomocí přepsání metod, jedná se vlastně o aplikaci FACTORY METHOD speciálně pro zrod rodin objektů.

FACTORY METHOD s parametrem

Předešlou úvahu můžeme rozvinout dále. Zvláštním klonem ABSTRACT FACTORY byl případ, kdy se interface „scvrkl“ pro zrod rodiny do jedné operace `Make()` se vstupním parametrem. Poté jsme tuto metodu přepisovali za účelem kompatibilního rozšíření množiny prvků rodiny.

Mohli bychom to považovat také za zvláštní případ použití FACTORY METHOD, kde se zavede uvedená Factory metoda se vstupním parametrem. Tuto variantu jsme podrobněji probrali již ve vzoru ABSTRACT FACTORY.

Související vzory

ABSTRACT FACTORY je mnohdy implementována pomocí FACTORY METHOD (viz konkrétní příklad motivu ABSTRACT FACTORY).

FACTORY METHOD se hojně používají u vzoru TEMPLATE METHOD. Jedná se o vzor, kde je napsán scénář poskládaný z několika polymorfních operací na horní úrovni abstrakce a spodní třída použije tento scénář tak, že tyto polymorfní operace přepíše podle svých potřeb a zavolá tento scénář. Jeden poskládaný scénář je reprezentován jednou metodou, což je právě „jedna TEMPLATE METODA“. V našem motivu je metoda `NewDocument()` právě takovouto TEMPLATE METODOU.

Zajímavý je vztah ke vzoru PROTOTYPE, který řeší podobné situace žádající odstínění volání konstruktoru, ale paradoxně nepotřebuje provádět dědění třídy `Creator`. Blíže viz další kapitola.

PROTOTYPE

Klasifikace

Object

Creational

Smysl

Zavádí skupinu objektů vznikajících klonováním prototypových instancí pomocí jednotného interfacu.

Motiv

Statické jazyky mají jednu velkou výhodu – jsou typově bezpečné. Na druhou stranu práce s typy bývá mnohdy velmi obtížná. Právě otázka, jak vhodně parametrizovat práci s třídami, je jedním z problémů statických jazyků. Mnohdy musíme v aplikaci nějakým způsobem řešit problematiku výběru třídy (tj. produktu) parametrem (například výběrem obsluhy, která vybere „číslo“, loading z dat apod.).

Některé předešlé vzory podávaly určité návody pro tato řešení, viz například klon FACTORY METHOD jako FACTORY se vstupním parametrem (resp. stejná myšlenka u ABSTRACT FACTORY se vstupním parametrem). Existovala v tom případě metoda `Make()` (nebo `Create()` apod.) se vstupním parametrem ID a navracely se různé produkty.

Problém je v tom, že přes všechny výhody je toto řešení poněkud „tvrdé“ vůči malým změnám. Pokud přidáme pouze jeden produkt, i tak musíme přepisovat celou metodu, i když použijeme re-use děděním. Navíc nelze dynamicky tento seznam produktů v běhu programu měnit.

Existuje však ještě jeden známý vzor, který řeší podobné situace. Je překvapivě velmi účinný, a přitom velmi jednoduchý. Dokonce lze konstatovat, že se vyznačuje vyšší flexibilitou než řešení nabízené předešlými vzory.

Jako první motiv zvolíme již známý příklad s obrazci. Představme si, že chceme v aplikaci provést výběr nového obrazce obsluhou. Máme možnost použít již zmíněný vzor FACTORY se vstupním parametrem, v tomto případě ID obrazce:

```
//... obsluha vybrala nějak IDObrazec...
CObrazec MujObrazec = MojeFactory.Make(IDObrazec);
...
```

Přitom jsme operaci `Make()` s návratovým typem `CObrazec` implementovali pomocí konstrukce `switch` apod.

Existuje ještě druhá zajímavá varianta. Představme si, že by každý obrazec ze stromu dědičnosti obrazců podporoval operaci `Clone()`, tj. operaci takovou, že když ji zavoláme, tak instance ze sebe vydá nový objekt jako svou kopii. Přitom je tato operace `Clone()` zavedena již jako abstraktní v předkovi `CObrazec` a je tedy přepisována každým potomkem, takže nezáleží na tom, s kterou instancí pracujeme. Instance jsou ve volání `Clone()` kompatibilně zaměnitelné.

Takovouto instanci umístíme do nějakého seznamu (kolekce, arraylist apod.) spolu i s klíčem. Tyto instance budeme nazývat prototypy. Seznam prototypů, tj. kolekci prototypů nazvěme „prototype

manager“. Zásada je taková, že pokud budeme potřebovat nový objekt obrazce, nebudeme volat konstruktor, ale budeme žádat tuto kolekci, tj. prototype manager, o novou instanci obrazce. Pošleme mu jako vstupní parametr odpovídající klíč. Uvnitř kolekce se najde podle klíče odpovídající instance. Tato nalezená instance se požádá pomocí `Clone()` o kopii, která se vydá ven jako návratová hodnota..

Poznámka: Slovy z českého známého filmu „Jak primitivní,.. ale jak účinné... :)

Zajímavé na tomto postupu je to, že volání o nový objekt bude velmi podobné, jako v předešlém kódu s FACTORY o pár řádku výše:

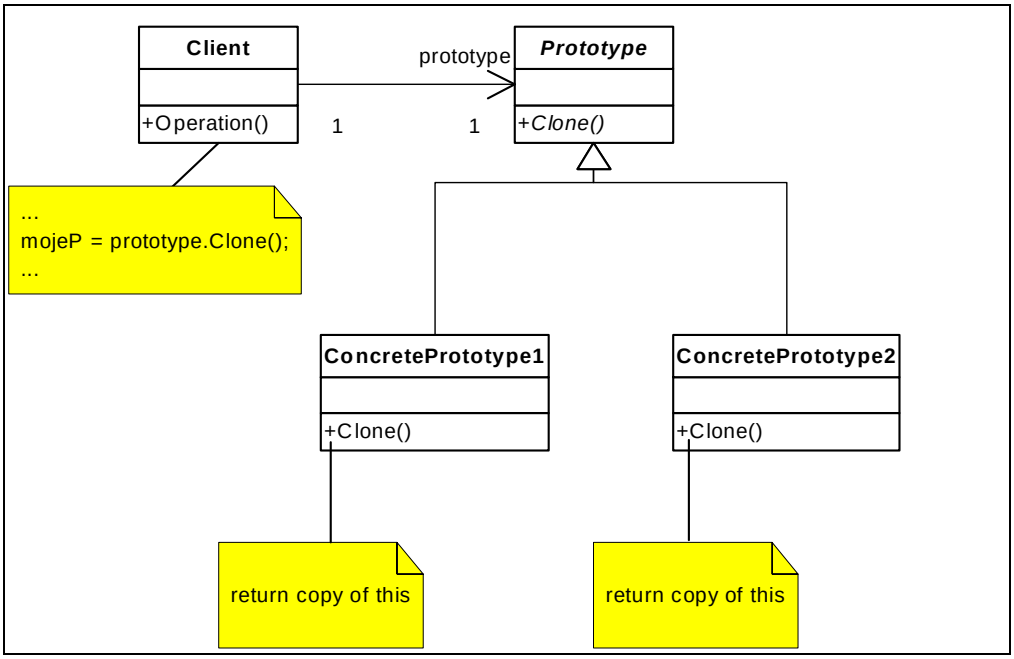
```
CObrazec MujObrazec = MujPrototypeManager.Make(IDObrazec);
```

Avšak mezi předešlou konstrukcí s FACTORY a touto s prototypy není rozdíl pouze v názvech. Přístup přes prototypy umožňuje „živou“ práci s instancemi prototypů. Je třeba si uvědomit, že někde v inicializaci se volá odpovídající metoda registrace daného prototypu do kolekce, tj. nějaké `Add()` do tohoto seznamu:

```
//init prototype manager...
...
CKruh MujPrototypKruh =  new CKruh();
MujPrototypeManager.Add(MujPrototypKruh, key=constKruh);
...
```

Podle toho, jaké prototypy budou přidány nebo odebrány (a to i například dynamicky obsluhou!), takový bude výsledný seznam nabízených produktů. Zde je vidět výhoda práce s instancemi. Můžeme s nimi snadno „handlovat“. S třídami a kódem v nich je to už horší. Všimněme si, že klient, který používá prototypy, může jejich seznam obsluhovat dynamicky. To jen jedna z možností, kterou tento vzor poskytuje. V kapitole *Poznámky k použití* si uvedeme další možnosti tohoto vzoru.

Struktura vzoru



obrázek 19 Struktura vzoru Prototype

Na předešlém obrázku abstraktní třída `Prototype` definuje interface pro operaci klonování (tvorbu kopie objektů). Potomci tuto operaci implementují. Klient žádá o nový objekt, přičemž používá tento interface, aniž by se zajímal o to, kdo jej implementuje.

Poznámka: Je třeba si uvědomit důležitý fakt, že podstatou tohoto vzoru není pouze to, že se instance umějí klonovat, ale navíc ještě to, že daná „rodina objektů" je v klonování kompatibilní, tj. klient používá jeden pro celou rodinu společný interface s operací `Clone()` poděděný od abstraktní třídy a implementovaný na konkrétní úrovni.

Poznámky k použití

Prototype Manager

Ukazuje se výhodné zavést pro registraci prototypů jejich správce. Jeho funkcionalita byla dostatečně popsána v předešlých kapitolách.

Specifikace produktů v runtime

Je zřejmé, že tento vzor je velmi výhodný (pro svou jednoduchost) tam, kde se jinak vytvářejí seznamy tříd v runtime. Klient může přes obsluhu prototype managera instalovat a odstraňovat nové prototypy v runtime, a to velmi jednoduše.

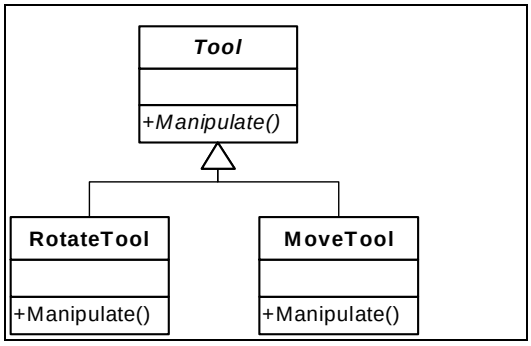
Konfigurace v runtime

Podobně jako v předešlém případě lze využít prototypy pro konfiguraci systému, například při načtení údajů z dat při startu systému apod. Postup je v podstatě stejný jako v předešlém odstavci.

Řešení problémů při přechodu od křížení tříd na jiný typ interakce

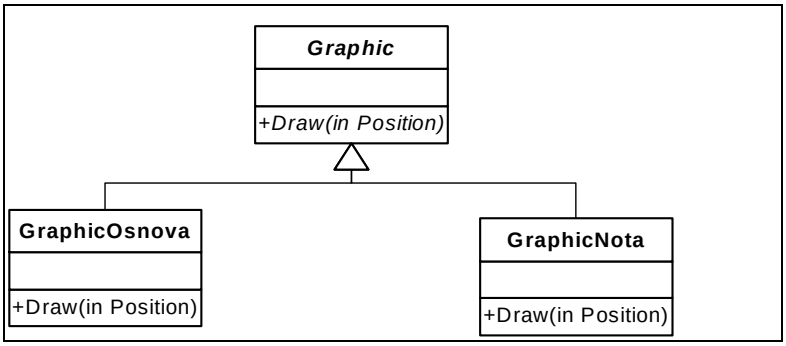
V některých případech by otrocké použití dědění vedlo k velmi silnému křížení mezi třídami a doslova k „množení“ tříd. V tom případě se používá jiný typ interakce než dědění (blíže bude rozvedeno ve vzoru BRIDGE), kdy se „přemostí“ dva stromy dědičnosti přes vrcholové instance, tj. nahradí se interakce „dědění každý s každým„ interakcí „skládání na horní úrovni abstrakce“.

Jako klasický příklad bychom mohli uvést příklad z [5], kde se popisuje problém konstrukce grafického editoru hudby. Uživatel může vybírat noty a jiné prvky notace znázorněné graficky, posouvat s nimi apod. a vytvářet tak notaci hudby. Návrh grafického editoru je takový, že existuje skupina tříd umístěná do stromu dědičnosti nazvaná Tools (nástroje). Jedná se o třídy zavádějící operace pro manipulaci s grafickými objekty:



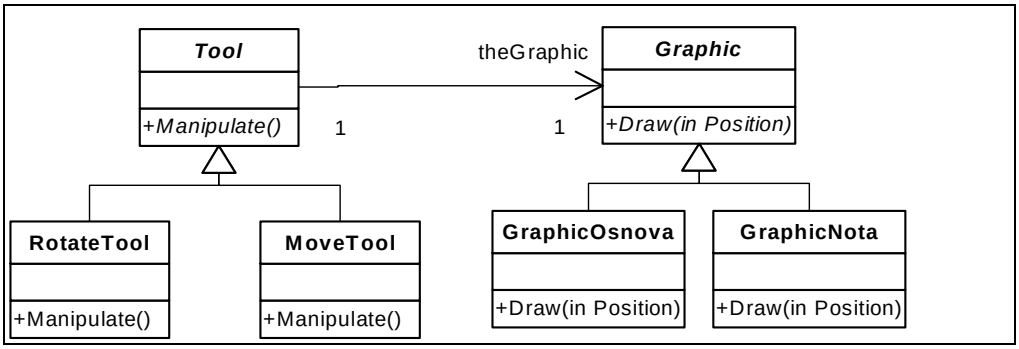
obrázek 20 Třídy pro manipulaci s grafickými objekty

Editor s těmito třídami již „umí“ pracovat (obsluha „umí“ vybrat správnou operaci a nástroj se „použije“ voláním operace `Manipulate()`. Na druhé straně existuje strom grafických útvarů pro zápis not, který je reprezentován třídami, jejichž objekty se umějí namalovat:



obrázek 21 Grafické třídy umí malovat prvky notace hudby

Oba předešlé obrázky bychom potřebovali nějak „sloučit“ dohromady. Jedna z možností je provést interakci na úrovni tříd přes dědění a vytvořit „křížence“ nějak takto: „speciální tool versus speciální grafický prvek“. To však vede k velkému počtu tříd. Druhá varianta spočívá v tom, že bychom propojili oba stromy na vrcholové úrovni skládáním objektů. Do vrcholové třídy `Tool` je vložen objekt s interfacem `Graphic`. Výsledkem je následující obrázek:



obrázek 22 Interakce stromů skládáním

Například pro zápis nové noty obsluha provádí dvě volby: volí třída `MoveTool` a volí se typ zrodu objektu jako typ nová `GraphicNota`, která hraje roli `theGraphic`. Otázkou je, jak flexibilně a jednoduše zvolit třídu `GraphicNota`. Ukazuje se jako výhodné použít vzor PROTOTYPE, tj. problém se převede na výběr prototypické instance. Ve vrcholové třídě `Graphic` se zavede abstraktní operace `Clone()`, kterou každý dědic přepíše. Výběr „co je skutečně vloženo za interface `Graphic`“ je převeden na problém výběru prototypické instance. Jedná se o jednotný mechanismus, protože operace `Clone()` je polymorfní, tj. vždy se použije tentýž kód (tj. vždy se volá `Prototype.Clone();`) pro všechny případy dědiců `Graphic`, ať už byla vybrán libovolný z dědiců. Tímto je tento scénář zrodu nezávislý na výběru dědice `Graphic`.

Pseudotřídy jako implicitní stavy prototypů

V seznamu prototypů se pracuje s instancemi, přičemž vůbec není specifikováno, z jaké třídy daná instance pochází (to je důsledek zavedení polymorfní operace `Clone()`). Není tedy vyloučeno, aby se v seznamu nacházely dvě instance z téže třídy. Této skutečnosti můžeme využít pro možnost zavedení nových produktů bez zavedení nové třídy. Bude se jednat o takové produkty, které se liší od existujících pouze implicitními stavy. Tento způsob je určitě výhodnější a přehlednější, než nastavovat tyto stavy vždy při zrodech objektů.

Uživatelské objekty jako prototypy

Protože lze s prototypy pracovat stejně jako s každou jinou instancí, můžeme si představit situaci, kdy uživatel poskládá nějakou strukturu a tuto prohlásí za prototyp. Pochopitelně povaha struktury tomu musí odpovídat (musí existovat možnost algoritmu skládání ve volání Clone). Například by se jednalo o editor skládaných elektrosoučástek apod. Pokud lze zobecnit Clone do této polohy (složená struktura se umí klonovat), výrazně to rozšiřuje možnosti práce s takovýmto editorem. Uživatel potom sám vytváří nové prototypy a rozšiřuje tak množinu produktů, které lze používat.

Kopie je samostatná

Kopie by měla být opravdu „novým objektem“ a nikoliv pouze „nový ukazatel“ na již existující prototyp. Druhý případ by odpovídal sdílení objektů a nikoliv zrodu objektu.

Kopie přes data

V některých vývojových prostředích lze s výhodou využít již existující funkcionalitu pro „přesypání“ údajů z prototypu do kopie. Pokud existuje možnost nějakého „automatického“ uložení do dat a „automatického“ vytěžení z dat, potom prototyp uložíme a natáhneme údaje do kopie. Většinou se jedná o zavedené implicitně podporované operace „streamování“ objektů jako marshalling a unmarshalling, použití zavedeného interfacu `IPersist`, operace `SaveToStream()` a `LoadFromStream()` apod.

Související vzory

Vzor ABSTRATC FACTORY a PROTOYPE lze považovat za konkurenční.

SINGLETON

Klasifikace

Object

Creational

Smysl

Zabezpečí, aby třída měla pouze jednu instanci globálně viditelnou ze všech bodů programu.

Motiv

V mnoha případech je třeba, aby ze třídy vznikla pouze jedna sdílená instance pro celý program. Již jsme na tento požadavek nejednou narazili. Určitě bychom požadovali, aby objekt `Factory` zavedený vzorem ASBTRACT FACTORY byl pouze jeden objekt a byl viděn odevšad. Podobně bychom požadovali, aby prototype manager ze vzoru PROTOTYPE byl také pouze jeden a byl globálně viditelný. Takových objektů (většinou služebních) může být spousta: manažer tiskáren, datový objekt pro přístup do databáze apod.

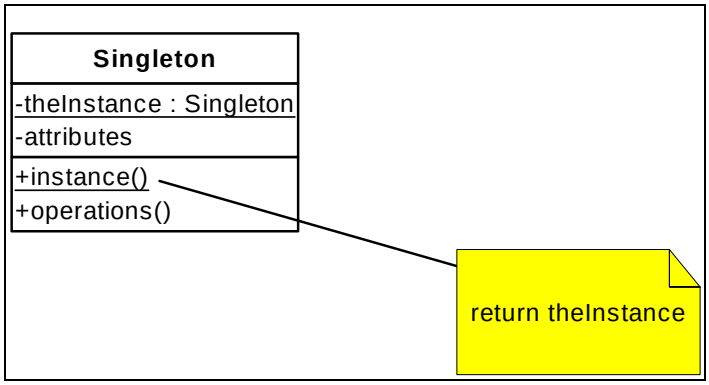
Naskýtá se několik možných variant, jak tuto situaci řešit, máme na výběr podle prostředí, v jakém se nacházíme.

1. První, nepříliš šťastné řešení by mohlo využít strukturálního přístupu (například v Delphi). Zavedla by se globální instance takového objektu (například public proměnná typu objekt ze třídy v interfacu unity).
2. Druhý podobný přístup je ve strukturálním programování zavést modul a v něm skupinu public funkcí. Tyto modulární funkce reprezentují metody jedno-instančního objektu.
3. Další podobné řešení (již v OOP jazyce) by bylo zavést v nějaké třídě objektovou statickou public proměnnou. Jedná se o obdobu řešení odstavce 1., pouze nepoužíváme modulární programování, ale objektové. Modul je nahrazen třídou, globální proměnná statickým public členem této třídy.
4. Zavedeme třídu a v něm skupinu public statických metod. Jedná se opět o obdobu 2., kde jsme využili obdoby modulárního programování. Třída reprezentuje jeden modul a její statické metody nahradily public funkce modulu.

Všechna tato řešení mají několik nevýhod. Například u 1 a 3 se musíme potýkat s otázkou inicializace objektu, současně nastává dost velké a nepřehledné zaplnění globálního prostoru. U 2 a 4 máme problém, jak případně extensivně měnit kód takovéto struktury, aniž bychom zasahovali do klienta. Další problém lze očekávat díky tomu, že statické metody nelze (v uvedených jazycích) přepisovat.

Vzor SINGLETON zavádí jednotné a (jak se ukazuje) nejvýhodnější řešení. Základní princip tohoto vzoru je ten, že instance se umístí jako privátní statický člen, zpřístupňuje se přes public statickou metodu a vlastnosti instance jsou v téže třídě jako prky dynamické struktury, tj. členové a operace.

Struktura vzoru



obrázek 23 Struktura vzoru SINGLETON

Poznámka: V UML značí podtržení „vztahující se k danému prvku – instanci“, tedy v tomto případě podtržení znamená statické metody a statické atributy třídy (vlastnosti přímo třídy). Nepodtržené jsou vlastnosti instance ze třídy.

V jedné třídě máme umístěno celé řešení jednoho objektu globálně viditelného. Je zavedena třída `Singleton`, v ní existuje jedna instance `Singletonu` s názvem `theInstance` a je umístěna jako privátní statický člen (podtrženo s minusem). Tato instance zásadně není zvně třídy viditelná. Existuje statická operace s názvem `instance()` (podtržená s plusem), která vrací tuto privátní instanci. Další atributy a operace znázorňují strukturu a chování samotné instance ze `Singletonu`, tj. chování navráceného globálně sdíleného objektu. Klient může volat pouze statickou operaci `instance()`, např. takto:

```
MujSingleton = Singleton.instance();
```

A poté může volat operace samotné sdílené instance:

```
MujSingleton.operation();
```

Nebo „přímo“ bez mezi-proměnné, pokud to syntaxe jazyka dovolí:

```
Singleton.instance.operation();
```

Poznámky k použití

Unikátnost instance

Díky jedné public statické operaci vracející jednu a tu samou privátní instanci je zabezpečeno, že všichni klienti dostanou tutéž instanci Singletonu. Nezavádějí se žádné „globální proměnné“.

Dědicové Singletonu

Dynamické operace Singletonu lze podědit a přepsat, což znamená možnost použít kompatibilně „jiný“ Singleton, například vyměnit Singletony z téže rodiny při konfiguraci apod.

Zrod instance Singletonu a skrytý konstruktor

V uvedené konstrukci je ukryt způsob, jak zabezpečit zrod instance. Statická operace vracející instanci nejprve zjistí, zda instance existuje, pokud nikoliv, zrodí ji a teprve poté ji vrátí, zapsáno symbolicky např. takto:

```
public static Singleton instance();
 {
  if theInstance = nil then
       { theInstance = new Singleton };
  return theInstance;
 }
```

Pro zvýšení stability programu se doporučuje umístit konstruktor pro zrod instance do viditelnosti takové, aby nebylo možné tento konstruktor volat zvně. Klient Singletonu nesmí mít možnost konstruktor zavolat. Vzor pro unikátní a sdílenou instanci by tímto přestal fungovat korektně, klient by mohl tento požadavek přímým voláním konstruktoru narušit.

Existuje také možnost v některých jazycích zavolat implicitně konstruktor a přenechat zavolání zrodu přímo prostředí , symbolicky zapsáno např.:

```
...// class declaration
private static Singleton theInstance = new Singleton;
...
```

Poznámka: Tímto jsme ukončili výklad o vzorech sloužících k řešení situací zrodu objektů, tj. „Creational Patterns“.

ADAPTER

Klasifikace

Class nebo Object (možné dvě varianty)

Structural

Alias

Wrapper

Smysl

Object scope: Zavádí objekt jako spojku mezi klienta, který očekává určitý interface a neznámý interface, takže klient může používat cizí interface..

Class scope: Zavádí třídu spojující neznámý a očekávaný interface, takže klient může používat cizí interface.

Motiv

Při vývoji aplikací někdy nastane situace, že máme sice k dispozici nějakou třídu a mohli bychom ji používat, ale problém je v tom, že nám tato třída „nepasuje“ do již existujících struktur. Například tato třída podporuje interface, který v dané rodině „vypadá jinak“. Podobně zní otázka „jak zapojit cizí prvky do naší již existující aplikace“.

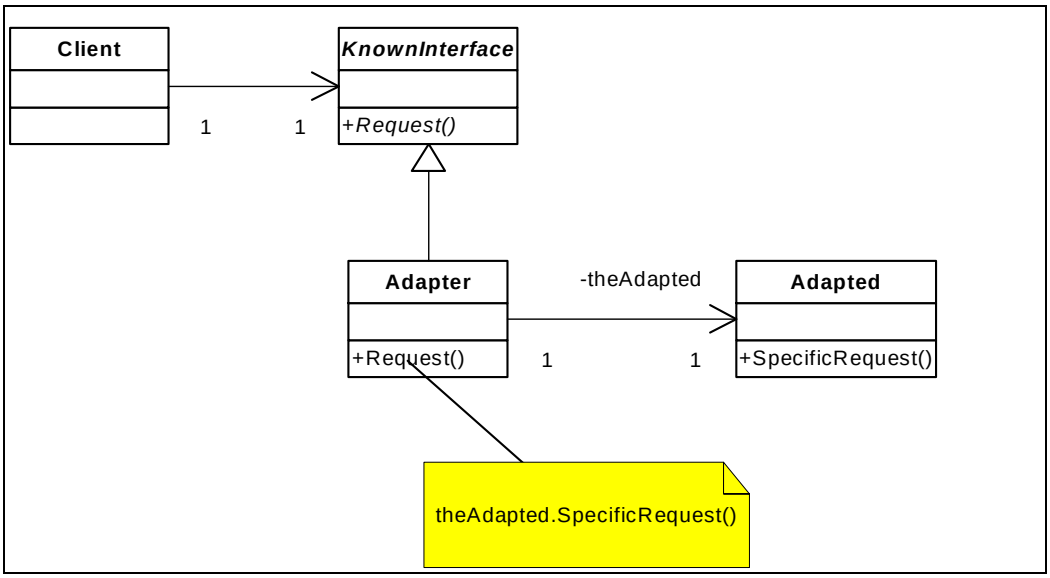
Odpovědí je návrh podle vzoru ADAPTER, jehož posláním je přizpůsobit „cizí“ nekompatibilní interface ke klientovi, který očekává nějaký známý interface.

Poznámka: Tento vzor je velmi často používán aniž by se vědělo o jeho existenci. Prostě se v dané situaci nalezne odpovídající konkrétní řešení “jak prvek začlenit do existující aplikace“ .

Struktura vzoru

Existují dvě varianty tohoto vzoru, jedna je class varianta (class pattern), druhá je object pattern. Častěji se používá druhá varianta, tj. object pattern.

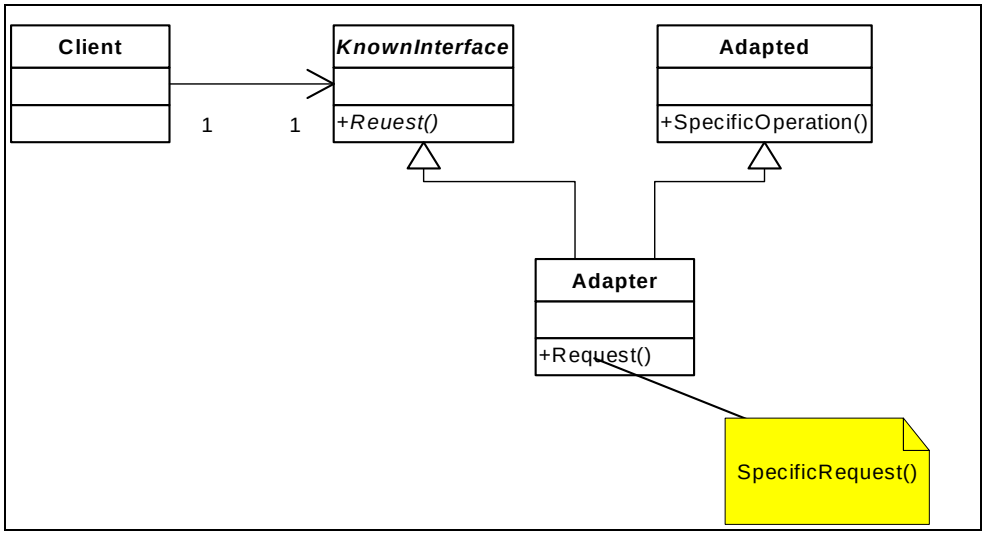
Struktura Object ADAPTERU



obrázek 24 Struktura vzoru object ADAPTER

Klient používá u objektu `Adapter` interface `KnownInterface`, který očekává. Zavoláním operace `Request()` se tento požadavek přesměruje na vložený objekt `theAdapted`. `Adapter` zastává roli spojky k objektu `theAdapted`. Tento způsob se používá běžněji než druhý (viz dále).

Struktura Class ADAPTERU



obrázek 25 Struktura Class ADAPTERU

V tomto případě je zvolena interakce nikoliv vložením objektu, ale interakce dědění mezi třídami, tj. `Adapter` „umí“ `SpecificRequest()` díky tomu, že je dědicem `Adapted` (a nikoliv díky tomu, že má vloženu instanci z této třídy). Tento přístup předpokládá, že můžeme od třídy `Adapted` vytvořit dědice, což nemusí být vždy splněno.

Poznámka: Postup může být i takový, že od třídy `Adapted` podědíme třídu `Adapter` a implementujeme interface `KnownInterface`.

Poznámky k použití

Object ADAPTER versus Class ADAPTER

Na rozdíl od Class ADAPTERU umožňuje Object ADAPTER pracovat jednak s objektem ze třídy `Adapted`, ale také se všemi jeho dědici, kteří díky tomu podporují stejný interface, což se jeví jako výhoda. Class ADAPTER tedy nepoužijeme v tom případě, kdy se vyžaduje takováto spolupráce s potomky podporující interface `Adapted`. Na druhou stranu Object ADPATER neumožňuje jednoduše přepsat metody třídy `Adapted`.

Jak chytrý má Adapter být?

Neexistuje pravidlo, které by určovalo, co všechno má Apdater jako „spojka“ mezi dvěmi částmi systému provádět. Může se jednat o pouhé přejmenování operací anebo může být vybaven nějakou dodatečnou funkcionalitou.

Související vzory

Existuje spíše podoba s jinými vzory: BRIDGE je velmi podobný, ale má jiný význam a smysl (viz dále). Podobně lze nalézt společné rysy se vzorem PROXY (viz dále). PROXY je také „spojka“, ale tam je klientovi nabídnut stejný interface, jaký má původní objekt, zde jsou dva interfacy, jeden přizpůsobovaný, druhý známý.

BRIDGE

Klasifikace

Object

Structural

Smysl

Odděluje abstrakci od implementace, takže lze obě dvě měnit nezávisle na sobě

Alias

Handle / Body

Motiv

V mnoha případech se používá pro vztah mezi abstrakcí a implementací dědění. Vytvoří se abstraktnější úroveň řešení a „pod ní“ jako dědicové se vytvoří „konkrétnější“ implementační úroveň.

Klasickým příkladem jsou aplikace, které řeší „technologii něčeho“. Představme si aplikaci, která řídí nějakou telefonní ústřednu v reálném čase. V aplikaci budou existovat třídy jako Telefon, Hovor apod. Tyto třídy budou dále „rozvinuty“ o úroveň níž například do nějaké technologie připojení telefonů k systému. Nižší „implementační“ třídy umí implementovat telefony a pracovat s externími částmi systému (realizují hovory konkrétně nějakou technologií komunikace) . Na abstraktnější úrovni se vytvoří vztahy jako „Telefon drží Konferenční Hovory“ apod. Nižší třídy budou tyto vztahy konkretizovat do nějakých technologií telefonů. Můžeme použít dědění, například třída `TelefonIMPLX` je dědicem obecnějšího abstraktního `Telefonu`. Třída `TelefonIMPLX` implementovala telefon do konkrétní technologie X.

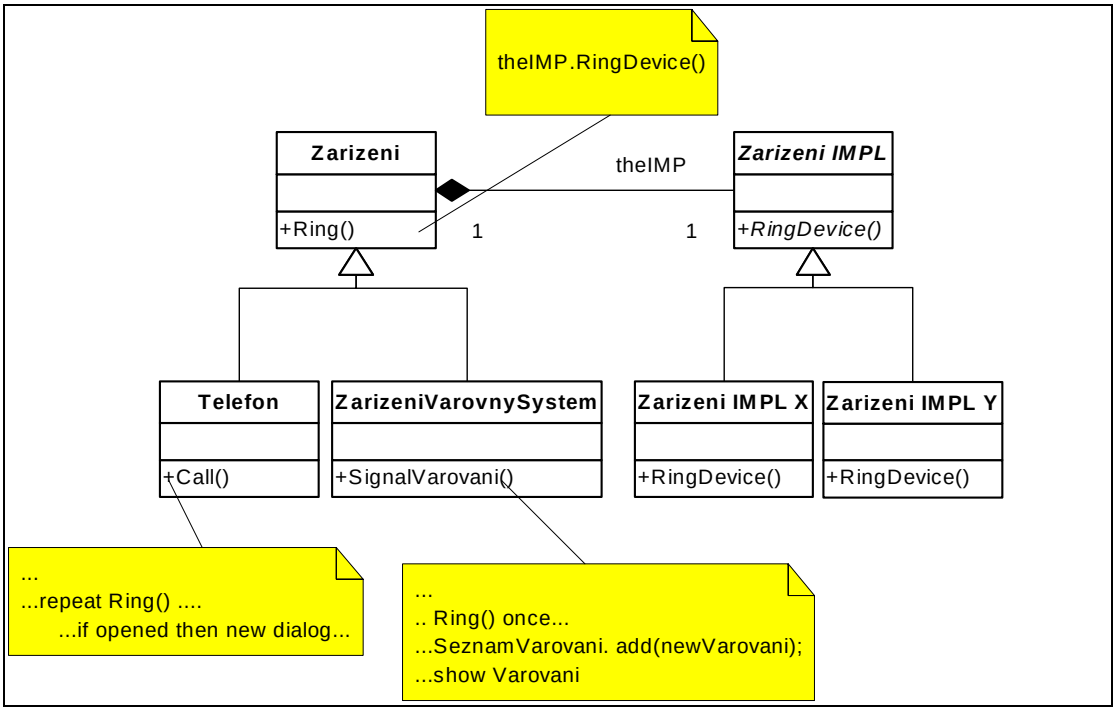
Tímto přístupem ale mohou nastat problémy.

Pokud existuje několik podtříd na abstraktní úrovni řešení a současně několik možných implementací, potom kombinatorika je v tomto případě neúprosná. Představme si, že v abstraktní rovině existují dva funkcionální typy zařízení (telefonů), například `ZařízeníA` a `ZařízeníB`, které se liší nezávisle na zvolené technologii telefonů nějakými vlastnostmi (například jak se děje hovor). Z toho plyne, že na abstraktní úrovni existuje strom dědičnosti mezi zařízeními z hlediska funkcionalit.

Kromě toho lze zavést několik různých technologií implementací telefonů, tj. jakoby nějaké různé třídy pro různé „telephon devices“ (tyto třídy schovávají konkretizaci komunikace). V tom případě, pokud použijeme dědičnost, musíme pro každou podtřídu abstrakce vytvořit dvojici způsobem: „Pro každou abstrakci zaveď implementaci“. Třídy se začínají množit vzájemným křížením.

Řešení tohoto případu je v mostu, tj. BRIDGE. Postupuje se tak, že se v prvé řadě zavede abstraktní třída implementace (což zní tak trochu divně, ale opravdu je tomu tak). Vrcholová třída abstrakce se propojí s touto vrcholovou abstraktní třídou implementace objektovou vazbou (přemostí se oba stromy).

Náš příklad (trochu upravený) by potom mohl vypadat takto:

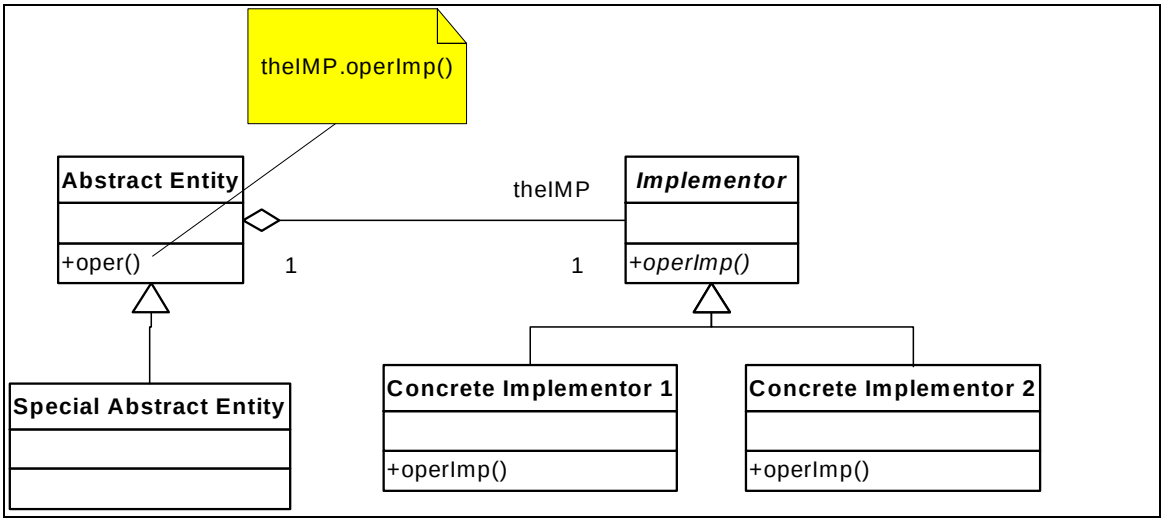


obrázek 26 Motiv na BRIDGE

Oba typy zařízení volají operaci `Ring()`, která je vyvedena do společného abstraktního předka `Zarizeni`. Všimněme si, jak je použita a jak je dovedena do implementace operace `Ring()`. Levý strom ukazuje strom abstrakce, tj. jak to chodí s různými zařízeními, hovory atd. apod. na abstraktní úrovni. Napravo je strom implementace, tj. použití různých technologií, tj. „device telefonů“ (přes port COM, síť apod.). Strom implementace má vrcholovou abstraktní třídu. Operace `Ring()` je implementována do konkrétní technologie nikoliv děděním, ale tak, že se deleguje tento požadavek na objekt `theIMP`. Za interfacem tohoto objektu se může vyskytnout jak objekt ze třídy `ZarizeniIMPLX`, tak objekt ze třídy `ZarizeniIMPLY` (podle zvolené technologie komunikace).

Není problém přidat novou implementaci, tj. řešit případ „nový dodavatel nové technologie připojení telefonů do systému“. Také není problém přidat nový typ zařízení do levého stromu (nový typ po logické stránce jinak fungujícího zařízení). Oba stromy lze nyní měnit a rozšiřovat nezávisle na sobě.

Struktura vzoru



obrázek 27 Struktura vzoru BRIDGE

Hlavní a nosná myšlenka vzoru BRIDGE je zavést abstraktní třídu implementace a tuto provázat s abstrakcí přes objektovou referenci, tj. použít implementaci skládáním objektů.

Poznámky k použití

Abstrakce a implementace jsou odděleny i fyzicky

Použití vzoru BRIDGE nejenže zabraňuje „množení“ tříd, ale navíc odděluje implementaci od abstrakce přímo fyzicky přes hranici interfacu. Znamená to, že různé implementace mohou být umístěny do samostatných celků, a to nejenom na úrovni logických celků, nebo zdrojových kódů, ale dokonce i celků zvlášť kompilovaných. Je tedy možné kompilovat novou implementaci, aniž by se musela kompilovat abstrakce. Jinak řečeno lze účinně zavést komponentní technologii, tj. třídy stromu `Implementor` mohou být v samostatných komponentách (např. v Javě packages resp. JavaBeans, v DOT NET v Assemblies, v Delphi např. v ActiveX DLL komponentách) a tyto lze dynamicky přilinkovávat resp. vyměňovat.

Nezávislost v rozšíření stromů

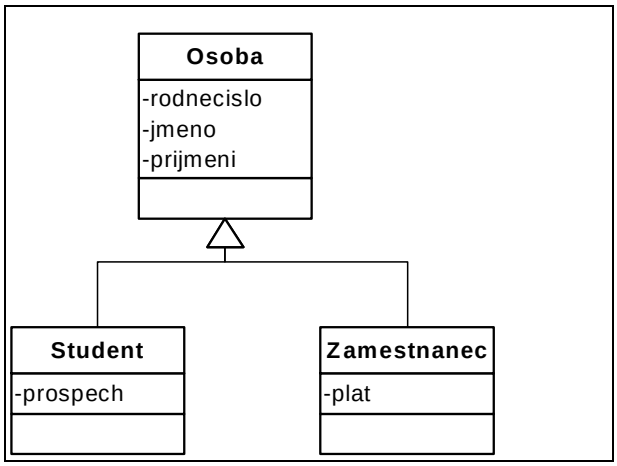
Oba stromy lze rozšiřovat o nové třídy aniž by to druhý strom pocítil. Abstrakce a implementace jsou samostatně extensivní.

Řešení chyby putování instancí a množení tříd

Vzor BRIDGE byl zaveden hlavně pro smysl uvedený v předešlých kapitolách, tj. oddělení abstrakce od implementace. V praxi jsem se však setkal s určitým klonem tohoto vzoru BRIDGE (podstata přemostění je stejná) u situací, kdy nastávaly problémy s „množením tříd“ již v analýze. Důvodem je chybně vytvořený strom generalizace - specializace, kdy není dodržena zásada disjunkce typů ve stromu a kdy může dojít k putování instancí z typu do typu. Důsledkem je nárůst podtříd exponenciálním způsobem.

Nejprve příklad:

Budeme navrhovat informační systém pro vysoké školy. Zavedeme třídu Osoba, která nese jméno, příjmení a rodné číslo. Protože v systému budou vystupovat „speciální“ osoby Zaměstnanec a Student, zdá se být logické, že zavedeme strom gen – spec (o vztahu gen spec viz např. [1]):

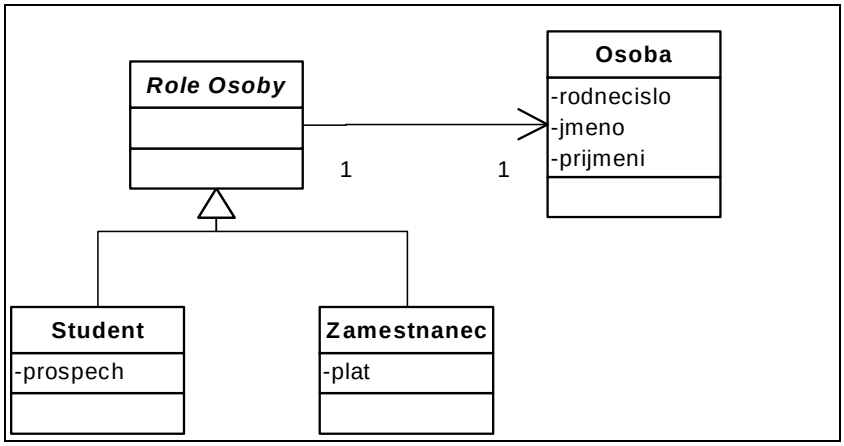


obrázek 28 Návrh gen spec pro osoby v systému (chybný)

Všimněme si, že tato konstrukce bude fungovat, ale má v sobě skrytu záludnou chybu. Co když bude osoba současně Zaměstnanec a současně Student? Taková Osoba by měla mít jak prospěch, tak i plat současně. Provedeme násobné dědění od obou tříd? Pokud ano, tušíme problém, který vynikne ještě více, když zavedeme třetí požadavek: Bude se evidovat i vysokoškolská knihovna a v ní čtenáři. To je další třída a kombinatorický počet je neúprosný, protože čtenářem může být kdokoliv. Vyskytl se efekt „množení tříd“, nazývaný v literatuře jako *explosion of subclasses*. Současně s ním se objevuje také problém putování instancí ze třídy do třídy, tj. osoba může být nejprve studentem a poté může být zaměstnancem.

Problém, který zde nastal, je důsledkem jedné chyby: Nedodržel se princip disjunkce (neprolínání) instancí ve stromu hierarchie. Osoba jako třída předek totiž nemůže být „sdílena“ v instanci mezi dvěma typy. Instance se při dědění musí vyskytovat buď v jedné, nebo druhé třídě (a ne v obou současně!). Podle našeho řešení se instance vyskytuje v obou třídách Student a Zaměstnanec, ale pouze jednou v Osobě!

Pokud se někde vyskytne případ množení tříd v projektu, vzpomeňte na vzor BRIDGE. Je třeba provést přemostění. Znamená to, že se zavede namísto interakce typu dědění interakce přes interface (v tomto případě běžná asociace). Výsledný model tedy bude vypadat takto:



obrázek 29 Řešení problému množení tříd o osob, vznik „mostu“

Pokud pracujeme se Studentem, nechá se vzniknout objekt ze třídy Student, pokud pracujeme se Zaměstnancem, tak vznikne nezávisle na Studentovi druhá instance ze třídy Zaměstnanec. Obě různé instance však mohou mít „za sebou“ jednu stejnou Osobu. Jsou to instance vedle sebe stojící a nezávislé. Provést porovnání, zda se jedná o stejnou osobu, v tomto případě znamená porovnat, zda došlo ke shodě stejné osoby v instanci, například přes ID osoby takto:

```
if MujZamestnanec.Osoba.ID = MujStudent.Osoba.ID then ...
```

Je třeba si uvědomit, že v tomto případě jsou již obě instance nezávislé, a tedy typy v instancích disjunktní (nesdílejí se role, sdílejí se osoby přes vazbu běžné asociace).

Související vzory

Pro volbu třídy implementace lze zvolit některý ze vzorů klasifikace „Creational Patterns“, například ABSTRACT FACTORY nebo PROTOTYPE.

COMPOSITE

Klasifikace

Object

Structural

Smysl

Skládá objekty do stromové struktury, přičemž klient přistupuje k prvkům stromu jednotným způsobem, tj. klient přistupuje k prvku bez rozlišení, zda se jedná o uzel větvení stromu nebo o koncový prvek.

Motiv

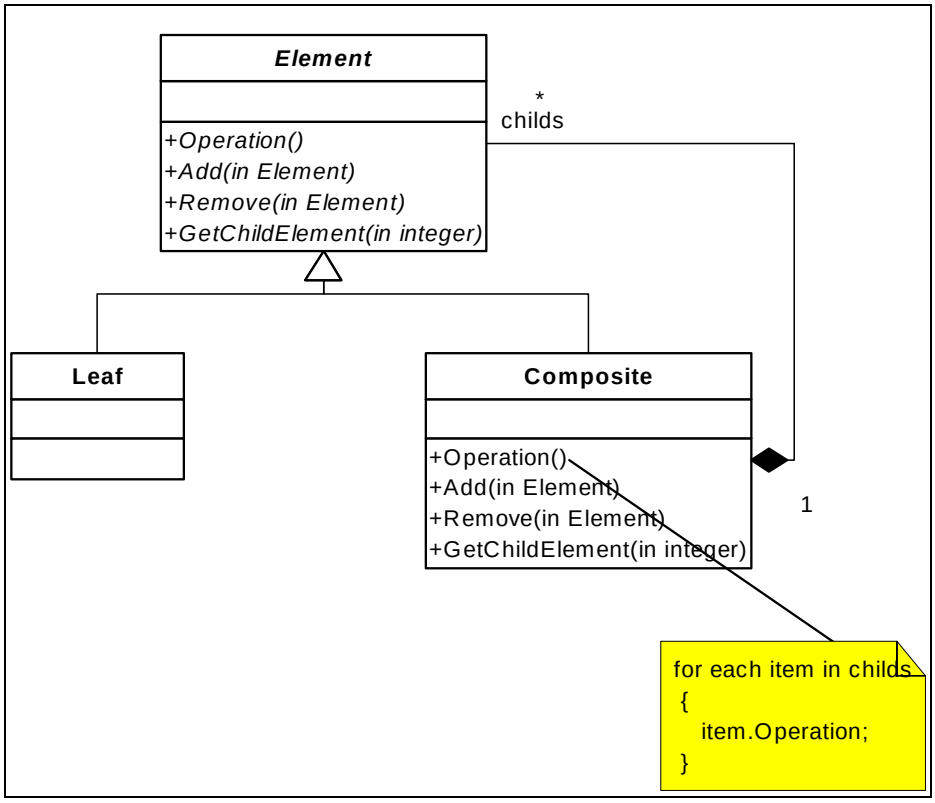
Pro motiv samozřejmě nemusíme chodit příliš daleko. Vraťme se k příkladu se složenými obrazci, viz kapitola „Příklad na polymorfismus“ a odpovídající řešení polymorfní operace `DejObsah()`.

Prvním krokem řešení složených obrazců bylo začlenění třídy složeného obrazce do rodiny obrazců a začlenění polymorfní operace `DejObsah()` do společného interfacu s rekurzívním voláním prvků složeniny. V tom je samozřejmě ukryto použití vzoru COMPOSITE, ale tím COMPOSITE nekončí.

Pokud bychom se zastavili a přijali toto řešení, potom bychom rozdělili prvky stromu do dvou velkých skupin: Dělíme prvky stromu na ty, které jsou listy stromu (neznají operace pro děti), a na uzly dalšího dělení (znají operace pro děti). Práce s takovým stromem se stává více složitější než v případě, že bychom všechny prvky stromu unifikovali. Řešení bude transparentnější, pokud všechny prvky stromu zařadíme pod jednu abstraktní třídu, která bude obsahovat maximalisticky všechny operace pro všechny prvky (tedy nejenom polymorfní `DejObsah()`). V předkovi se budou vyskytovat dokonce i operace pro správu dětí (jako `Add()`, `Remove()` apod.). Všechny prvky se tak stávají unifikovanými a práce s nimi bude jednodušší.

Diskuse viz kapitola *Poznámky k použití*.

Struktura vzoru



obrázek 30 Struktura vzoru COMPOSITE

Je zavedena abstraktní třída `Element` pro všechny prvky stromu. Operace `Operation()` se z hlediska skládání chová rekurzivně, tj. volají se prvky, ze kterých je `Composite` složen . Všechny elementy stromu podporují stejný interface.

Poznámky k použití

Maximalizace interfacu Element a filosofie vzoru COMPOSITE

Všimněme si, že vzor doporučuje umístit do horní úrovně „maximum“ operací, které dědicové přepisují. To se může jevit jako rozpor proti návrhu hierarchie tříd. Přece nebudeme „cpát něco někomu, co nemá umět“!

Doporučení maximalizace horního interfacu je však vhodné dodržet s tím, že pohled na hierarchii tříd a kompetenci jednotlivých úrovní se trochu změní, tj. analyticky se vyjádří „trochu jinak“. I při této maximalizaci zůstane podstata řešení stejná, jak si dále ukážeme.

Vysvětlíme si to na příkladu. Z předešlého vzoru vyplývá (viz obrázek), že operace správy „childs“ jsou na horní úrovni. Znamená to snad, že toto mají umět i objekty ze třídy `Leaf`? Ti přece děti nemají a ani je mít nemohou!

Představme si, jak by vypadala situace, kdybychom „operace pro správu dětí“ umístili opravdu „korektně“ dolů až do třídy `Composite`. Prvek `Leaf` by tedy tyto operace neznal. Nyní přichází důležitá a pro tento vzor podstatná úvaha:

Klient pracuje se stromem. Jakého návratového typu dostane prvky ze stromu? Zásadně typu `Element`, protože takto musí být typ návratové hodnoty definován, aby se v něm mohly vyskytovat jak Leaf, tak `Composite`. Klient si totiž může vybrat libovolný prvek stromu, což značí, že se mu prvky jeví jako obecné `Elementy`. Tuto úvahu můžeme zkrátit: Programátorsky řečeno, návratový typ operace `GetChildElement()` (případně podobných pro průchod stromem) musí být typu `Element`.

Představme si scénář, kdy uživatel vybere nějaký prvek stromu a chce k němu přidat dítě. V GUI se mu objeví graficky znázorněný strom a pomocí něj se identifikuje prvek stromu. A teď se vžijme do pozice programátora, který bude řešit tuto situaci. V této chvíli drží v ruce vybranou instanci prvku stromu a nový přidávaný element. Nazvěme tyto objekty jako `VybranyElement` a `NovyElement`. Může programátor zavolat operaci `VybranyElement.Add(NovyElement)`? Při naší konstrukci „operace dole“ nikoliv, protože tuto operaci `Element` (který programátor „drží v ruce“) neumí. Programátor by tento příkaz `Add()` u prvku `VybranyElement` ani nezkompiloval (mluvíme o statických typově bezpečných jazycích).

Programátor musí provést *downcast*, například takto (samozřejmě záleží na syntaxi prostředí, zde zvolena syntaxe našeho pseudojazyka):

```
Composite MyComposite =  VybranyElement; //to je přetypování na potomka...
MyComposite.Add(NovyElement);
```

K přetypování došlo přiřazením, v tomto přiřazení je uschována žádost o jiný interface. Druhý způsob by mohl být pomocí závorkování nějak takto: `Composite(VybranyElement)`.

Musíme však navíc ošetřit i tzv. „blbovzdornost“. Co když uživatel vybral (buď omylem anebo zlomyslně) prvek ze třídy `Leaf`? Potom kód u přetypování spadne. Musíme jej buď ošetřit, anebo se zeptáme (nějak) daného prvku, zda je daného typu, například nějak takto (zde je zvolena varianta, kdy si `Element` ukazuje do číselníku typů elementů):

```
...
if VybranyElement.TypElementu.ID <> constClassComposite then
   {
     Message ( “Nelze přidat, vybraný prvek není composite!”);
     exit;
  }
...
```

Pohled z hlediska klienta jako „návod k použití stromu“ je složitý proto, že se rozpadá na dvě nesouvisející části spojené nestandardní operací přetypování prvků.

Vzor COMPOSITE doporučuje: Nedělte prvky stromu a všem dejte pokud možno maximálně stejný interface. Takže i prvek `Leaf` bude mít operaci `Add()`. Otázkou je, co s ní. Existuje několik možných návrhů, všechny postavené na stejném principu: Třída `Leaf` přepíše obsah této metody podle svého.

Možnosti jsou tyto:

- `Leaf` má operaci `Add()`, ale nereaguje na ni. Tělo `Add()` je prázdné. Je to jednoduché (například je takto definována metoda na horní abstraktní úrovni a nemusí se proto pro `Leaf` přepisovat). Je to sice i „blbovzdorné“, ale vůči uživateli tak trochu „zlomyslně poťouchlé“. Představme si, že uživatel omylem vskutku vybere nějaký list stromu a bude se mu pokoušet přidat dítě. Prvek si samozřejmě nic nepřidá a „mlčí“. Dovedu si živě představit, jak po několikátém pokusu uživatel zuří a tvrdší nátury si svůj vztek vylévají na servisním hot-line, kde se mu dostane vysvětlení, že k listu stromu nelze přidávat prvky.
- `Leaf` má `Add()` a reaguje na něj vyvoláním výjimky. Toto „tvrdé“ řešení je doporučováno v literatuře (např. [5]). Analyticky se věta „list nemá děti“ převedla na větu „můžete list požádat o přidání dítěte, ale reaguje na to velmi podrážděně - výjimkou“. Klient musí zavedenou výjimku ošetřit.
- Operace `Add()` má návratovou výstupní hodnotu, například konstantu typu `integer` apod. `Leaf` si tedy nepřidá dítě, ale přitom navíc vrátí hodnotu, za kterou je uschována informace „error, nejsem Composite“. Je to „měkčí“ varianta předešlého způsobu.

Všimněme si, jak se práce se stromem z pozice klienta objektu zjednodušila. Programátor dostane mnohem „jednodušší návod“. V něm bude uvedeno pouze to, co se může očekávat při volání `Add()` u libovolného prvku stromu. Například v posledním případě vyjmenovaných možností implementace `Add()` v prvku `Leaf` s návratovou hodnotou může kód vypadat nějak takto:

```
// jsme ve formuláři...držíme VybranyElement a NovyElement...

if VybranyElement.Add(NovyElement) = constErrNOTCOMPOSITE then
   {
     Message ( “Nelze přidat, vybraný prvek není composite!”);
     exit;
   }
```

Parent reference

Doporučuje se vždy, aby prvky měly referenci na svého majitele - parenta. Pokud tuto referenci nezavedeme, nastávají problémy s pohyby ve stromu nahoru. Tato reference je zavedena již na úrovni třídy `Element` (tj. `Element` je sice abstraktní třída, ale nikoliv čistá).

Vyvstávají otázky, jak lze „handlovat“ s dětmi, zda je lze přenášet z prvku jednoho parenta do jiného prvku parenta, zda je lze sdílet apod. Sdílení prvků například vede k „pochybným“ stromům, které jsou složitější, než jak je popisuje tento vzor. Je třeba vždy zvážit povahu vazby. Vzor COMPOSITE funguje bez problémů u vazby typu komposice.

Vazba na parenta se mnohdy využívá ve vzoru CHAIN OF RESPONSIBILITY (bude pojednáno v odpovídající kapitole).

COMPOSITE s „klasickou“ komposicí a předávání prvků

U Composite s agregací typu „děti jsou komposice“ se někdy reference parent nastavují pouze při zrodu jako vstupní parametr konstruktoru a neexistuje žádná jiná operace pro její změnu. `Element` takto může žít pouze u jednoho parenta a nikdy nikde jinde. Pokud má dojít ke změně parenta, chápe se tato změna jako „zmizení“ této instance a zrod jiné instance u jiného parenta s přesypáním údajů z jednoho prvku do druhého (např. operací `Clone()`).

Pokud se má předat prvek od parenta k parentovi, je druhou možností skutečné „předání“ prvku jako předání instance. V tom případě se jedná o scénář bez kopírování `Elementu` se skutečným předáním reference. Instance proputuje z jedné kolekce do druhé, tj. zmizí reference v jedné kolekci a přibude reference v druhé kolekci, přičemž instance stále žije. Vynechá se tím jedna konstrukce objektu, jedno kopírování a jedna destrukce.

V obou případech musíme ošetřit, aby nedošlo ke zdvojení informace: V prvém případě by v případě chybného scénáře mohly existovat dvě kopie ve dvou kolekcích, ve druhém případě by mohlo dojít ke sdílení instance ve dvou kolekcích. Druhému případu by odpovídala vazba sdílené agregace a nikoliv komposice.

Flexibilita nových typů

Pokud přidáme nový typ jako dědice třídy `Element` a ten bude podporovat interface `Element`, klient tuto změnu nepocítí.

Poznámka: Je třeba však scénáře práce s prvky psát pokud možno v dynamice flexibilně. Například pokud bychom napsali kód pro ošetření návratové hodnoty přesně tak, jak je uvedeno v předešlém kódu nahoře v kapitole „Maximalizace interfacu Element a filosofie vzoru COMPOSITE“, mohli bychom se v případě přidání dalších návratových hodnot u dalších typů „dostat do problémů“. Naštěstí v tomto případě to asi hrozit nebude.

Pokud je však zřejmé, že může dojít k takovému rozšíření, asi by bylo lepší použít nějakou polymorfní operaci s možností flexibilně ji přepsat, nebo například použít vzor COMMAND (viz dále příslušná kapitola) apod. Požadavek na tuto flexibilitu je však obecný požadavek a nesouvisí pouze s tímto vzorem.

Související vzory

ITERATOR (viz dále) se mnohdy používá pro průchod stromem zavedeným vzorem COMPOSITE.

Vzor CHAIN OF RESPONSIBILITY mnohdy využije link na parenta (viz dále).

Mnohdy bývá DECORATOR (viz dále) i prvek vzoru COMPOSITE.

Prvky stromu může postupně navštívit prvek vzoru VISITOR a provést na nich novou operaci definovanou mimo prvky stromu (viz dále).

DECORATOR

Klasifikace

Object

Structural

Smysl

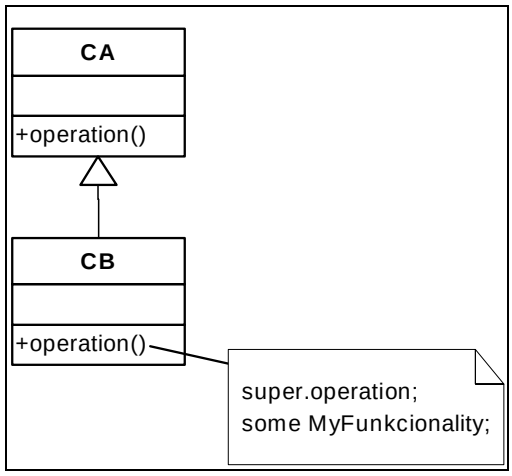
Přidává další přídavnou funkcionalitu k objektu dynamicky, čímž zavádí flexibilní alternativu k dědění.

Motiv

Představme si situaci, kdy používáme nějakou třídu například `CA`, u které voláme operaci `operation()`. Poté chceme vytvořit novou třídu tak, že potřebujeme doplnit třídu `CA` o další „přídavnou“ funkcionalitu k této `operation()`. Pod přídavnou funkcionalitou máme na mysli situaci, kdy se vyžaduje vykonat `operation()` a k tomu (buď předtím nebo potom) vykonat ještě „něco navíc“.

Jedno z řešení je využít dědičnosti: Provedeme dědění a v dědicovi `CB` přepíšeme operaci tak, že tato operace zavolá předka a poté (nebo předtím) přidá něco svého. Tímto postupem jsme rozšířili funkcionalitu při zachování kompatibility, protože můžeme použít jak předka, tak potomka.

Jako řešení nám vyjde následující struktura:



obrázek 31 Rozšíření funkcionality pomocí dědění

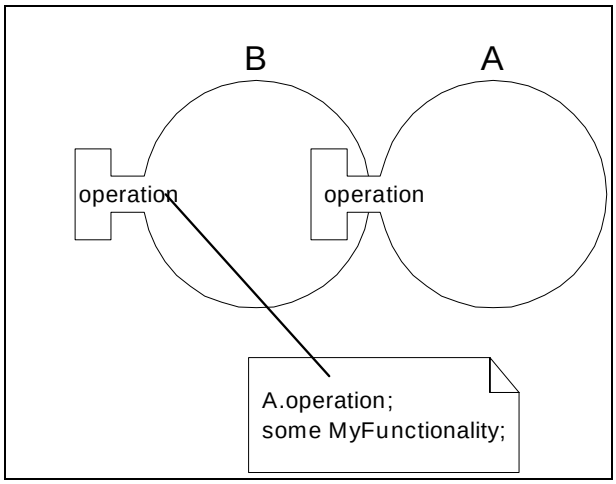
Poznámka: super zde značí volání předka. V některých jazycích se používá rezervovaného slova base, nebo inherits, někde se uvádí název třídy a před název operace oddělovač dvě tečky apod.

V literatuře (např. [5]) se používá jako klasický příklad skládání nových GUI prvků. Například prvek TextView má být vybaven navíc Borderem. Můžeme kombinovat jednotlivé třídy tak, že použijeme dědění, avšak tento přístup má zřejmou nevýhodu. Dědění je statickou vazbou, kterou již nadále nezměníme. Pokud ji použijeme, musíme počítat s omezenou možností kombinací. Vlastně můžeme

použít pouze těch dědění, které jsme zavedli. Pokud potřebujeme novou kombinaci, tak musíme provést další nové dědění, což nelze učinit v runtime.

Podívejme se na daný obrázek jinak. Dědic volá funkcionalitu předka a poté svou. Představme si obrázek nikoliv „shora dolů“, ale vedle sebe a prvky seřaďme jako objekty.

Například dva objekty s operací `operation()` vložíme za sebe takto:

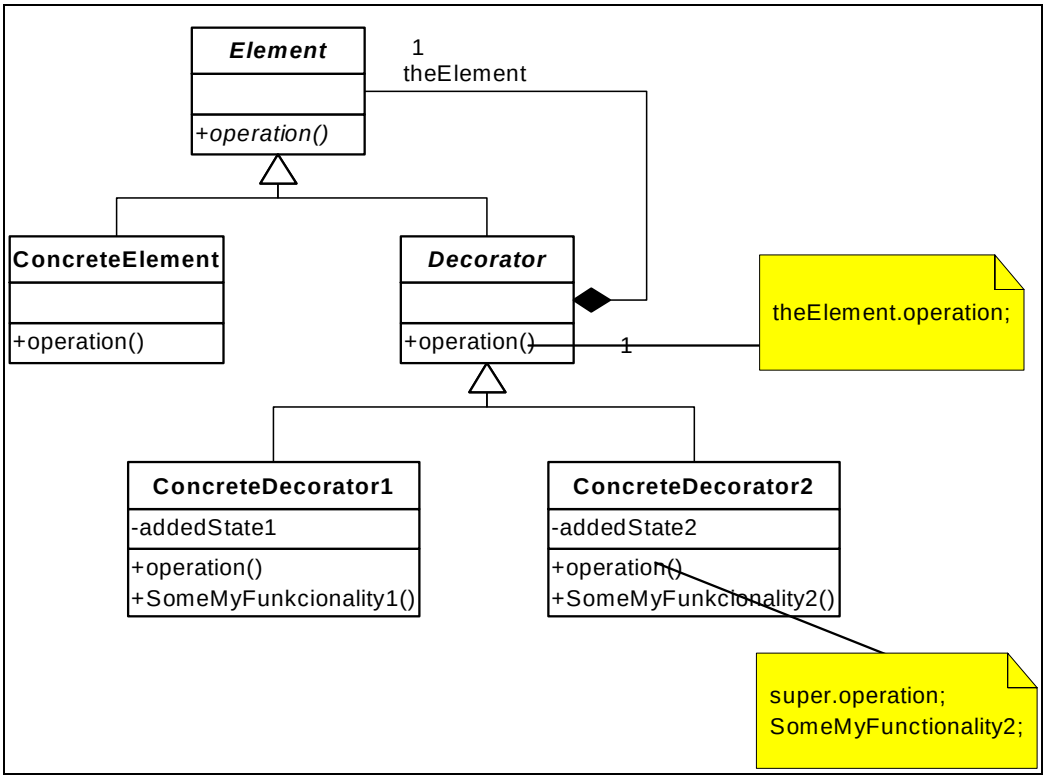


obrázek 32 Řazení objektů za sebe

Všimněme si, že získáváme vlastně stejný scénář jako u předešlého příkladu s děděním. Je zavolána operace prvku A (předtím to byla super), poté je přidána vlastní funkcionalita.

Vzor DECORATOR ukazuje, jak dosáhnout této struktury nahrazující extensi funkcionality nikoliv děděním, ale flexibilním skládáním. Pokud si představíme více prvků za sebou, tato struktura připomíná zřetězený seznam. Pro plnou flexibilitu musí být shodný interface propojení mezi prvky. Scénář volání operací za sebou používá jeden stejný kompatibilní interface. V tomto případě je reprezentován jedinou operací `operation()`. Potom lze prvky v tomto řetězu vyměňovat, řadit apod. i v runtime.

Struktura vzoru



obrázek 33 Struktura vzoru DECORATOR

V obrázku je uschována obdoba „zřetězeného seznamu objektů se stejným interfacem“. Všechny prvky seznamu jsou chápány jako `Element` (podporují interface abstraktní třídy `Element`). Třída `Element` má potomky `ConcreteElement`. Jejich funkcionalita je tímto vzorem flexibilně rozvíjena. Jsou to koncové prvky zřetězeného seznamu a nemají schopnost rozvíjet funkcionalitu někoho (základní kameny funkcionality).

Abstraktní třída `Decorator` (nikoliv čistá třída) zavádí ty prvky, které mohou mít za sebou další právě jeden prvek, tj. `Element`. Tímto prvkem `Element` může být buď další `Decorator`, nebo koncový `ConcreteElement`. `Decorator` je specializován do tříd, které mají své vlastní přídavné funkcionality a stavy. Jak stavy, tak funkcionalita, jsou takto přidávány k prvku, před který se předřazují.

Přímo z obrázku je patrno (viz odpovídající kódy ve žlutých Notes), jak je původní volání předka převedeno na volání přes interface `Element`.

Poznámky k použití

Společný interface

Základní myšlenka vzoru je ta, že konkrétní `Decorator` i konkrétní `Element` mají společný interface. Proto musí mít společného předka, abstraktní třídu `Element`. Z tohoto pohledu se díváme na

`Decorator` jako na novou „masku“ obalující (předřazenou) před `Element` (což může být například další „maska“).

Kdy se používá DECORATOR

Jednoduše řečeno, použijte DECORATOR tam, kde by řešení extenze (přidání) funkcionality děděním odpovídalo obrázku obrázek 31 , tj. jedná se o rozšíření funkcionalit, ale toto řešení děděním by přinášelo problémy se vznikem kombinací výsledných řešení, s hloubkou stromu apod., tj. problémy se statickou a již neměnnou vazbou. Většinou se jedná o GUI prvky, ale může se jednat i o prvky nevizuální.

Ve volně dostupných knihách [3], [4], jsou vzory DECORATOR ukázány v kódu JAVA a VB.NET, přičemž v [3] je ukázka DECORATORU pro nevizuální prvek jako rodiny Streamů v Javě. Zavádí se dodatečná funkcionalita ke Streamu. Přitom Streamů je celá řada (blíže viz [3]).

Podobně jsou Stream a jeho „kvasi-dědicové“ v podobě vzoru DECORATOR pro ASCII Stream, Compressing Stream rozebráni v knize [5].

Flexibilita a počet tříd je výhodou

Řešení extenze funkcionality pomocí DEORATORU je o mnoho flexibilnější než dědění, protože převádí problém extenze na skládání objektů za sebe, tj. na práci s instancemi. Lze vytvářet řetěz i v runtime. Navíc vzor zabraňuje vzniku „hodně hlubokých“ stromů.

Počet objektů a možnost libovolné kombinace je nevýhoda

Na druhou stranu vzor přináší také nevýhody: Problém se převedl na řetěz objektů. Ten se sice dobře sestavuje, ale hůře kontroluje. Například ten, kdo testuje, může mít problémy s orientací v kódu.

Navíc flexibilita obecně přináší problém, že můžete obecně zřetězit cokoliv, což by vzhledem určitým restrikcím nemuselo být vždy korektní.

DECORATOR a komponentní technologie, black-box inheritance

Pokud vložíme dvě komponenty do sebe a provedeme implementaci interfacu vnitřní komponenty do vnější, provádíme tak trochu podobný postup, jako doporučuje DECORATOR (skládání namísto dědění).

Rozdíl je v tom, že DECORATOR navíc vyžaduje, aby vložená instance měla „univerzální interface“ a nikoliv interface konkrétního nejbližšího předka. Postup uváděný jako „black-box inheritance“ je následující:

- Implementuj do třídy CB interface třídy CA
- Vlož „pomocnou“ instanci ze třídy CA do CB
- Volání operací implementovaného interfacu v CB směruj na pomocnou instanci, můžeš přidat vlastní funkcionalitu (zvláštní zavedení polymorfismu)

Postup ve vzoru DECORATOR je sice podobný (také vkládá a deleguje), ale rozdíl je v tom, že požaduje, aby třída CA nebyla nejbližším předkem třídy CB, ale aby se jednalo o společného abstraktního předka zavádějícího společný interface (viz struktura vzoru).

Pokud se nám podaří najít v našem řešení prvky DECORATORU, je samozřejmě výhodnější jej použít, než zavést „black-box inheritanci“. Například uvedená rodina Streamů je řešena pomocí DECORATORU a nikoli pomocí „black-box inheritance“.

Související vzory

Vzor je podobný ADAPTERU, resp. PROXY v tom smyslu, že za každým DECORATOREM se vyskytuje kompatibilní interface. ADAPTER je však určen k přizpůsobení prvků a provádí propojení mezi dvěma různými interfacy (jeden „cizí“, druhý „očekávaný“).

PROXY lze sice chápat jako „řetěz s právě dvěma prvky“ se stejným interfacem, ale rozdíl spočívá v tom, že PROXY je zaveden pro vložení nějaké „skryté“ funkcionality mezi klienta a objekt, přičemž PROXY je určen pro řízení přístupu k objektu. Ve vzoru DECORATOR se oproti tomu jedná o „obalení, masku“ za účelem rozšíření funkcionality.

Vzor DECORATOR lze (z hlediska matematického) považovat za degenerovaný vzor COMPOSITE. Pokud si porovnáme obě struktury, zjistíme, že obě vyznávají jeden kompatibilní Element (ze kterých jsou struktury složeny), pouze za DECORATOREM se vyskytuje jeden Element namísto N Elementů. Rozdíl je však nejenom v počtu, ale hlavně ve smyslu vzoru: DECORATOR předřazuje (doslova dekoruje), kdežto COMPOSITE řeší skládání stromu.

Podobný je vzor STRATEGY (viz dále), u kterého se mění vnitřek objektu, tj. zavádí možnost jiného algoritmu při stejné kompatibilitě. Ve STRATEGY se mění „vnitřnosti“, ve vzoru DECORATOR se předřazuje „maska“.

FACADE

Klasifikace

Object

Structural

Smysl

Zavádí jeden nebo více interfaců pro řízený přístup k subsystému, čímž zjednodušuje přístup klienta ke složitému subsystému.

Motiv

Tento vzor je obecně znám a je také často používán. Jeho význam ještě více vzrostl s použitím komponentní technologie (zejména u technologických komponent).

V mnoha případech se ukazuje, že je pro užití subsystému výhodnější nabídnout namísto interakce s celou množinou tříd v subsystému jeden (nebo několik málo) interfaců „navíc“ jako obdobu brány, tj. „fasády“ k subsystému. Klient subsystému potom nemusí používat relativně složitý systém mnoha tříd a jejich interfaců, ale používá pouze tento „fasádový interface“.

Příklady motivu:

V [5] je uveden příklad na kompilátor, což je relativně složitý subsystém používaný pro kompilaci,. Tento subsystém obsahuje množství tříd jako jsou Scanner, Parser, ProgramNode, ByteCodeStream apod. Některé specializované aplikace však budou používat několik málo přesně definovaných funkcionalit, například aplikace potřebuje pouze něco zkompilovat. Tento požadavek se vyvede jako nový interface s několika málo operacemi a aplikace používá pouze tento zjednodušený interface.

Obecně se fasáda používá hlavně u subsystémů, které jsou relativně složité, ale vyvádí se určitá část funkcionalit jako jakési „funkcionální view“, tj. pohled na subsystém pouze z určitého pohledu užití. Zavedením fasády se ostatní funkcionality (ty jsou nezbytné například z jiných funkcionálních pohledů) pro tuto aplikaci doslova odstíní.

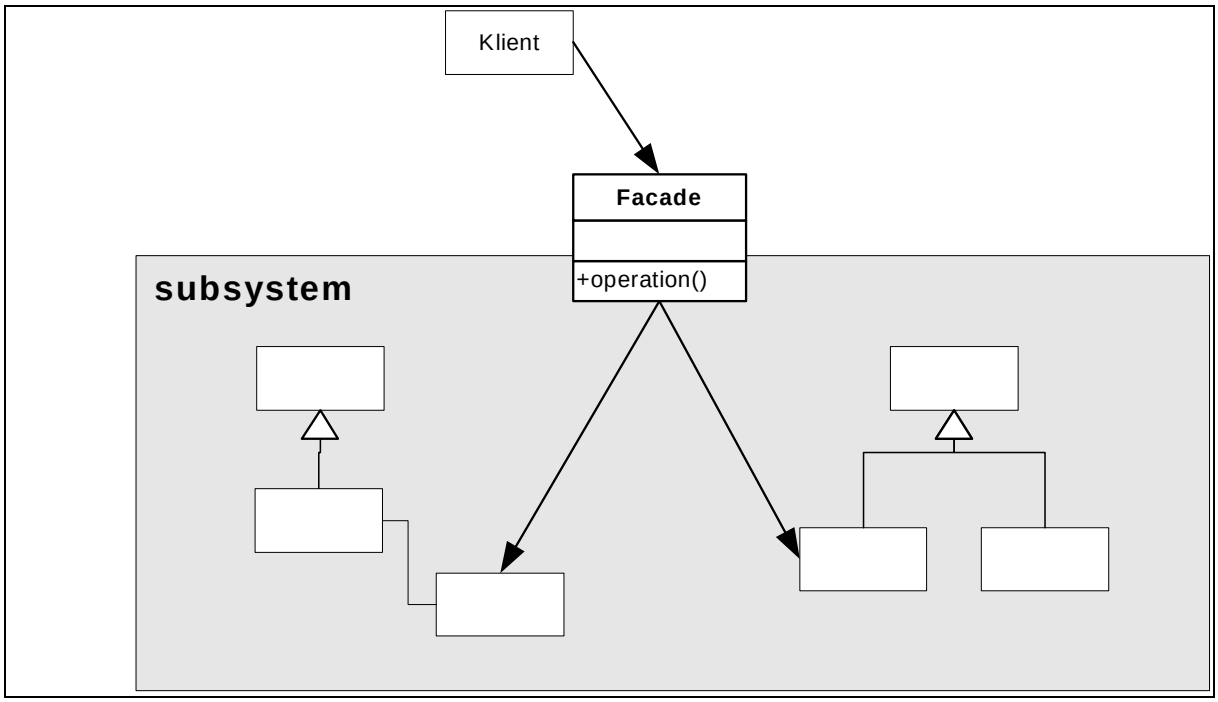
Například budeme řešit relativně složitou aplikaci rezervace míst v několika hotelech s možností výstupu na Web. Zákazníci hotelů mohou provádět rezervaci přes formuláře na Webu. Dovedeme si představit, že speciálně pro tuto část aplikace, kde klient používá browser a kde za tím účelem existují nějaké (dynamické) stránky, bude zaveden interface vyvedený ven ze subsystému jako fasáda určená speciálně pro toto objednávání. Klient (ve smyslu část systému užívající tento subsystém) nemusí vidět celou složitou aplikaci v celém modelu tříd, ale dostane k dispozici jednoduchý interface s několika málo operacemi. Tento interface se (například pomocí nějaké komponentní technologie) začlení do spolupráce s dynamicky tvořenými stránkami (doslova se vloží do ASP, JSP, ASP.NET apod.). Viděno z druhé strany však bude existovat ještě spousta jiných funkcionalit, které zrovna tento interface neobsahuje a tím je odstínil, například práce s administrací seznamu hotelů, s detaily jednotlivých hotelů atd.

U technologických komponent, které pracují v módu sdílení instancí mezi procesy, je fasáda řešením přístupu ke sdíleným objektům uvnitř sdílené instance subsystému.

Poznámka: Pod komponentou pracující v módu sdílení instancí máme na mysli situaci, kdy dva klienti ze dvou procesů (například na dvou strojích) přistupují k jediné sdílené instanci komponenty, (například na serveru).

Například pokud použijeme Message Queue Server pro frontování požadavků, pak klientům nebudeme nabízet celý model tříd, ale pouze několik málo interfaců pro práci se zprávami, jejich posíláním do front a vyzvedáváním apod.

Struktura vzoru



obrázek 34 Struktura vzoru FACADE

Interface třídy Facade je viděn a používán klientem. Facade zná a používá některé vnitřní třídy subsystému. Klient je takto odstíněn od jinak složitého modelu tříd.

Poznámky k použití

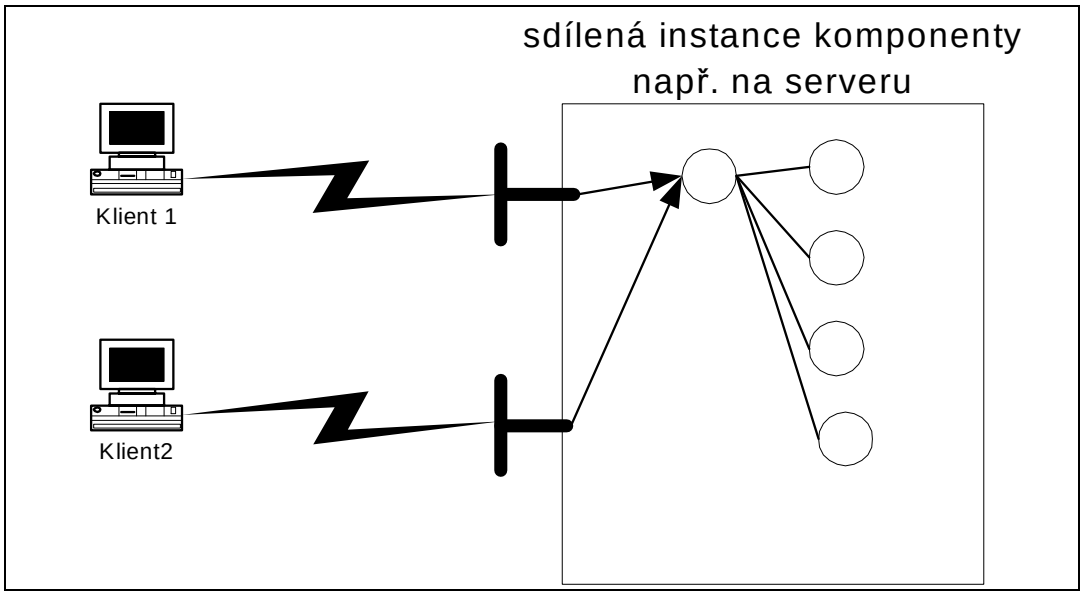
Odstínění pro tento případ použití nepotřebných tříd

Protože klient používá pouze interface Facade, nevidí ostatní vztahy (ani ho nezajímají), takže subsystém se pro něj stává z hlediska použití jednodušší.

Toto odstínění navíc provádí přesně definované oddělení subsystému od klienta přesně definovaným interfacem, takže subsystém se dík fasádě snáze oddělí od klienta až do fáze implementace.
V důsledku se lépe zavede komponentní technologie, protože klient a subsystém jsou díky oddělení fasádou implementováni odděleně.

Sdílené instance komponent a použití FACADE

U sdílených instancí komponent, kde se jedná o re-use nejenom na úrovni tříd, ale na úrovni instancí tříd, tj. na úrovni sdílení objektů v paměti, je použití obdoba vzoru FACADE také řešením přístupu ke sdíleným objektům. Následující obrázek můžeme chápat jako aplikaci vzoru FACADE, na obrázku jsou znázorněny dvě instance interfacu FACADE pro dva klienty:



obrázek 35 FACADE u sdílených instancí komponent

Problémy s obecností a předvídání situací u klienta

Na druhou stranu ne vždy může být použití FACADE až tak velkou výhodou. Může se stát, že zavedení fasády sice zprůhlední použití subsystému, ale na druhé straně povede k příliš silnému omezení možností použití tohoto subsystému, zejména s výhledem do budoucna. V některých případech je výhodný kompromis v ponechání možností: Buď klient přistupuje k nabízené fasádě anebo si může vybrat určité (jiné) třídy k použití. Tato a podobná rozhodnutí (velikost fasády, apod.) hodně závisí na povaze aplikace.

Související vzory

Při použití vzoru FACADE se jeví jako výhodné zavést vzory odstiňující volání tříd v konstrukci objektů, pokud chceme zachovat „jednoduchou“ fasádu bez velkých specifikací další velké množiny tříd.

U řešení pomocí nesdílených (izolovaných) instancí komponent (tj. synonyma v technologiích: „in-proc server“, „session scope“, re-use na úrovni tříd, knihovna tříd, přilinkování tříd a tvorba objektů s klientem) postačí většinou jedna instance fasády pro celý prostor klienta.

U sdílených instancí (tj. „exe-server“, „application scope“, „application enterprise bean“, sdílená instance přes prostory, sdílená instance komponenty na serveru) je třeba zrod pro každého klienta subsystému jedné instance fasády, navíc je třeba řešit multithreading, zamykání sdílených instancí apod., pokud tyto problémy neřeší samo prostředí.

FLYWEIGHT

Klasifikace

Object

Structural

Smysl

Řeší problém systémů s velkým počtem objektů pomocí sdílení.

Motiv

Třídy vzniklé z analytického modelování (pojmy v analýze) lze namapovat do designu 1:1 jako třídy OOP takřka vždy (mapovat „1:1 ve třídách“ znamená „co pojem v analýze, to třída v OOP programu“).

V mnoha případech však nelze model v analytickém modelování (někdy označován jako logický model) namapovat do designu také i z instancí pojmů do instancí objektů v OOP 1:1. Znamená to, že instance chápané v analýze jako výskyty informací se někdy do OOP mapují jinak, než ve vztahu 1:1 (mapovat „1:1 ve výskytech“ znamená „co výskyt informace v analýze, to instance jako objekt v paměti“).

Důvod je prostý: Výskytů informace je tolik, že jejich převedení na instance objektů není technicky reálné. Například analytická věta „Systém eviduje cca asi 100 000 firem“ (to je odhad počtu výskytu informací pojmu firma v analýze) neznamená, že budeme „držet v paměti“ 100 000 objektů ze třídy firmy. V těchto případech nejsou výskyty informace namapovány přímo na objekty alokované v paměti, ale na nějakou kombinaci údajů plus nějaké instance objektu.

Příkladem v literatuře (viz [5]) je editor textu (s fonty apod.). V analytickém modelu si dovedeme představit model, kdy text na stránce se skládá ze sloupců (chápáno jako novinové sloupce), sloupce se skládají z řádků a řádek se skládá z písmen obsahujících polohu, font, charakter apod. Pokud bychom tento (po analytické stránce správný) model namapovali do OOP v instancích podle vztahu „1:1“, nastal by problém s počtem instancí alokovaných v paměti. Například tento text knihy napsaný od začátku až do konce má okolo 230 000 znaků. Představa, že by všechny znaky libovolného dokumentu žily v paměti, je nereálná.

Vzor FLYWEIGHT (v překladu „muší váha“) znamená vlastně synonymum pro odlehčení systému a navrhuje řešení tohoto problému. Základní úvodní jednoduchá myšlenka tohoto vzoru je následující:

U objektů, jejichž počet je třeba redukovat (např. znaky), vyveďme určité stavy objektu „ven“ a umístěme je do polohy vstupních parametrů požadovaných operací (u znaku např. operace vykreslení `Draw()`). Je zřejmé, že někde tyto stavy musí existovat a „být drženy“. Nejenom to, z kontextu musí být jasné, že tyto parametry, které má objekt dostat, patří původně tomuto objektu. To je problém daného mapování logické informace na design informaci.

Tato operace vyvedení parametrů ven musí mít za následek možnost nasdílet objekty tak, že jeden objekt lze použít několikrát, a to s různými hodnotami vstupních parametrů, čímž vznikne jiný kontext (jiná logická instance informace) použití téhož objektu v paměti. Sdílené objekty se chovají podobně jako šablony, přes které necháváme proplouvat vstupní hodnoty. Někdo (nějaký „znalý“) samozřejmě musí tento proces řídit a mít k dispozici nějaký „datový zdroj držící datově stejný kontext“ a tím dodávat sdílenému objektu „ty správné“ vstupní parametry odpovídající danému původnímu logickému modelu.

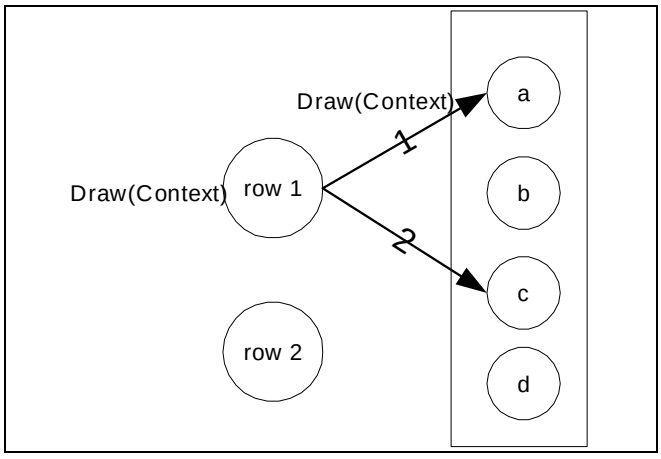
Představme si, že jsme problém textového editoru vyřešili takto: Zavedeme pouze tolik objektů, kolik je celkem možných znaků. Vznikne tak jakýsi „pevný číselník znaků“. Prvky tohoto číselníku jsou objekty a ty obsahují pouze každý ten „svůj“ znak. Objekt znaku se umí vymalovat zavoláním operace `Draw()`, přičemž vstupním parametrem této operace jsou hodnoty „posice plus font“ (dále tuto dvojici

nazývejme obecněji kontext). „Někdo řídící“ umí pracovat nad datovým zdrojem dokumentu a současně nad tímto číselníkem. Nazvěme tohoto „řídícího“ `Mapper`. Objekt `Mapper` „zná“ a umí pracovat s namapováním logické instance informace (znak logicky viděný v dokumentu) na řešení „kombinace údaje v datech (identifikátor, posice, font, tj. kontext) plus objekt číselníku“. `Mapper` načte z dat identifikátor znaku, posici a font, požádá číselník o vydání objektu s daným identifikátorem, zavolá `Draw()` se vstupním parametrem načtené posice a fontu, tj. vstupní parametr je kontext. Jeden a tentýž znak jako objekt je takto použit tolikrát, kolikrát se v dokumentu vyskytuje. Vstupní parametry vyvedené ven můžeme chápat jako „vnější stavy“ (extrinsic state) a udávají kontext použití daného objektu. Znaky v číselníků se nazývají FLYWEIGHT (odlehčené objekty).

To je základní myšlenka, ale úvahy ohledně tohoto postupu mohou jít dále. Některé kontexty nemusí být předávány přímo z dat do uvedeného číselníku znaků rovnou jedním Mapperem, ale mohou existovat ještě mezi-úrovně objektů „těch, kdo řídí kontext“, ale současně jsou tito považováni také za FLYWEIGHT, ale již nesdílené (objektů tak trochu může v počtu „přibýt“).

Například v našem případě můžeme chápat instanci řádku jako objekt přijímající operací kontext (je FLYWEIGHTEM), tedy i on zapadá do rodiny FLYWEIGHTS, ale přitom řídí jiné FLYWEIGHTS, na něž si ukazuje. Jinak řečeno, objekty jako řádek, sloupec se také chovají jako FLYWEIGHTS - přijímají svůj kontext (znají polymorfní operaci `Draw()`), ale na rozdíl od znaků nejsou sdílené (rodí se a zanikají). Přidávají a ubírají se ze seznamu všech FLYWEIGTHS. Přitom nesdílené FLYWEIGTHS (řádek) řídí kontextem „své podřízené FLYWEIGHTS“, na které si ukazují (sdílené znaky).

Příklad na sdílené a nesdílené FLYWEIGHTS, řádek, znaky, ale bez fontů:



obrázek 36 Nesdílený FLYWEIGHT Row a sdílený Char

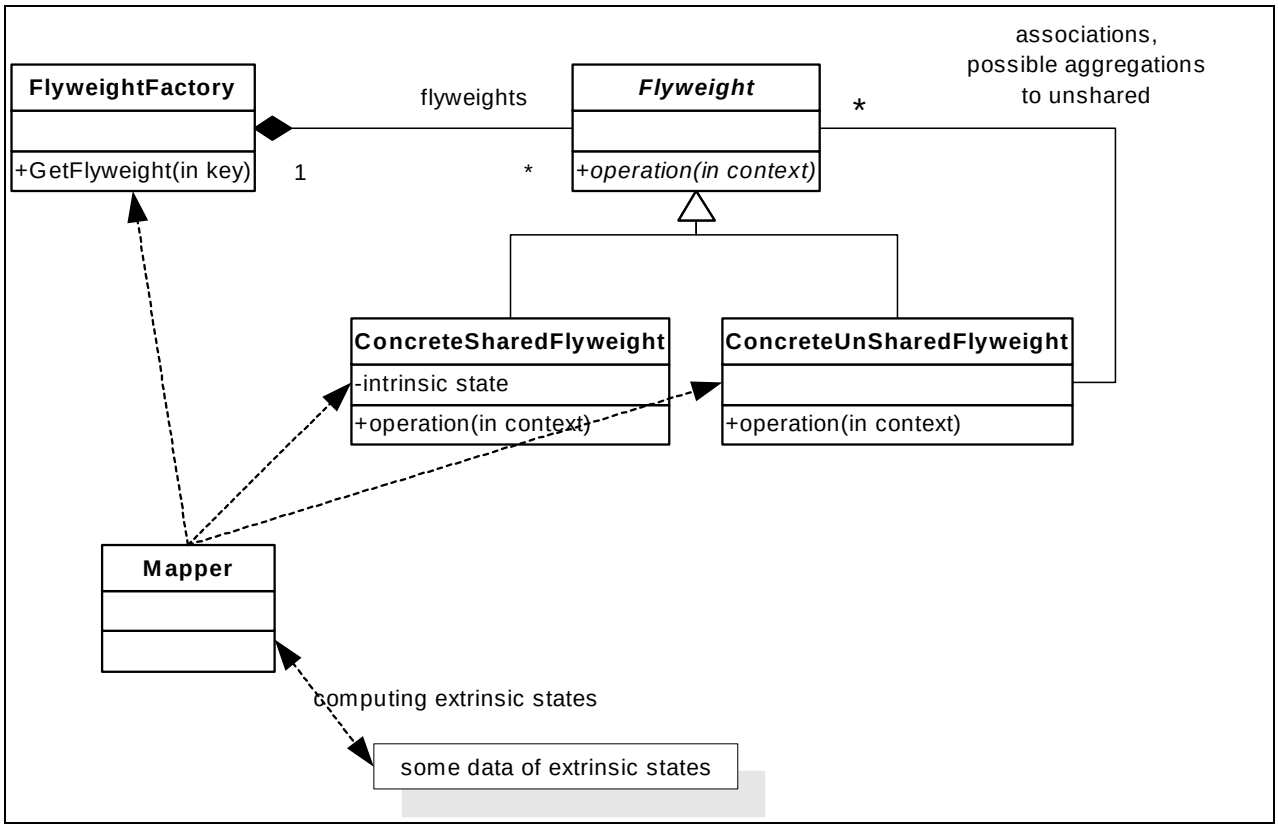
Jediný rozdíl mezi sdíleným a nesdíleným FLYWEIGHT je v tom, že sdílené FLYWEIGHTS existují „od počátku v číselníku“ a na ně se ukazuje jako do stabilního číselníku, naopak nesdílené se rodí a zanikají (přibývají a ubývají v seznamu všech FLYWEIGTHS). Všichni v číselníku (sdílení a i nesdílení) umějí operaci `Draw()`, každý podle svého.

V podstatě nesdílené FLYWEIGHTS (např. řádek) napomáhají (mimo jiné) k snazšímu a transparentnějšímu přenesení kontextu od původních dat až ke sdíleným FLYWEIGHTS, a to zavedením polymorfní operace zpracování kontextu. Přenesení kontextu operací se tak děje na mezi-úrovních zpracování (po sloupcích, po řádcích apod.), což je více transparentní a odpovídá to mimo jiné původnímu logickému modelu 1:1 (proto je transparence vyšší).

Je zřejmé, že nesdílené FLYWEIGHTS vedou k opětnému nárůstu objektů v systému, například v tomto případě bude „žít v paměti“ tolik objektů řádků, kolik se jich skutečně vyskytuje v dokumentu, ale objektů znaků bude podstatně míň, než kolik je „počet znaků v dokumentu“.

Struktura vzoru

Strukturu vzoru FLYWEIGHT ukazuje následující obrázek:



obrázek 37 Struktura vzoru FLYWEIGHT

Interface `Flyweight` definuje operaci pro přijetí a zpracování kontextu, tj. přijetí vnějších stavů daného objektu. Tato operace je provedena daným „správně z číselníku vybraným“ objektem `Flyweight`. Třída `ConcreteSharedFlyweight` zavádí sdílené objekty číselníku, ty obsahují své vnitřní stavy. Tento stav není závislý na kontextu (může být a je sdílen). Číselník objektů `Flyweight` je zde zaveden s názvem `FlyweigthFactory`. Vnější (extrinsické) stavy jsou uloženy resp. počítány u klienta, zde třída `Mapper`, která ve spolupráci s číselníkem `FlyweigthFactory` získává odpovídající `Flyweight` a tyto získané vnější stavy pošle danému správnému `Flyweight` jako kontext dané operace. Nesdílené objekty ze třídy `ConcreteUnSharedFlyweight` jsou také součástí rodiny `Flyweight`, také mají polymorfní operaci zpracování kontextu, také jsou prvky číselníku `FlyweightFactory`, ale jsou přidávány a odebírány v běhu programu. Vstupují do interakce buď s jinými nesdílenými objekty (sloupce drží řádky) anebo se sdílenými objekty (řádky vidí znaky).

Poznámky k použití

Kdy se nedá vzor použít

Pokud je splněna alespoň jedna z těchto podmínek, není vhodné, nebo dokonce není možné vzor zavést:

- Aplikace nemá příliš mnoho objektů (potom mapujte z logického modelu na návrh v instancích 1:1)
- Nelze vyvést vnější stavy ven (není co sdílet)
- Sice lze vyvést vnější stavy ven, ale efekt zmenšení počtu objektů je minimální (je toho málo ke sdílení)
- Objekty potřebují ponechat objektovou identitu (ztráta reference). Zavedení tohoto vzoru totiž vede k tzv. *ztrátě reference*. Původní instance z analýzy mají mezi sebou vazby, tj. linky. Tyto vazby na logické úrovni se tímto vzorem nepřevedly na odpovídající vazby objektových referencí. Je třeba vždy zvážit, zda tato ztráta vede ke ztrátě funkcionality v daných scénářích anebo nikoliv. Naštěstí se ve většině případech dá tento problém buď z povahy věci ignorovat (například se vždy pracuje najednou pouze s jednou instancí z dané třídy apod.), anebo se dá obejít drobnými změnami v návrhu (použití kopií apod.). V každém případě se dodržuje zásada porovnávat vždy shodu dvou objektů hodnotami, nikoliv porovnáním shody objektových referencí. Zásada je „nepoužívat pro porovnání shody objektů shodu ukazatelů, ale identifikátorů“. Mnohdy k řešení tohoto problému ztráty reference stačí pouze dodržet tuto zásadu.

Nárůst režie zpracování

Zavedení tohoto vzoru může velmi efektivně napomoci v systémech s velkým počtem objektů. Jednak se zmenší požadavek na alokovanou paměť objektů, jednak ubudou nároky na režii zrodu a zániku velkého počtu objektů.

Na straně druhé, mapování objektem `Mapper`, tj. vytěžení a výpočet vnějších stavů, získání správného objektu `Flyweight` vyžaduje také nějakou režii, která je zde navíc. Někdy může být algoritmus přenosu od uložených dat až k odpovídajícímu objektu `Flyweight` poměrně dost složitý a je třeba zvážit způsob a efektivitu tohoto mapování (provázání vnějších a vnitřních stavů do jedné instance). Například se může jednat o klasický spor „pomocná odložená data jako mezivýpočty versus složitý algoritmus“ apod..

Navíc je třeba zvážit efekt „proplutí hodnot“ objektem, tj. objekt si své vnější stavy ze zásady „nepamatuje a nedrží“. To může některé algoritmy výrazně zpomalit (vždy se pro ně musí chodit do Mappera).

Správce všech objektů `Flyweight` je vždy číselník, např. `FlyweightFactory`

Je zřejmé, že klient `Mapper` musí pro jeden daný klíč pracovat vždy se stejným objektem `Flyweight`, jinak hrozí ztráta kontextu. Znamená to, že manažerem objektů `Flyweight` a jejich roditelem je sám číselník a nikdo jiný. Klient nikdy nerodí sám objekty `Flyweight`, protože by mohlo dojít k nejednoznačnosti jejich použití.

COMPOSITE a FLYWEIGHT

V kombinaci FLYWEIGHT se mnohdy používá COMPOSITE a to tak, že listy stromu jsou sdílené jako objekty `Flyweight`. Existuje tedy číselník „sdílených listů“ a strom se dá různě skládat z nad-úrovní, uzlů (to jsou nesdílené objekty `Flyweight`). Vnějším stavem listu vyvedeným ven a vráceným jako

vstupní parametr operace v tomto případě musí bezpodmínečně být (mimo jiné) vazba na parenta. Každý list pod jiným uzlem má samozřejmě jinou vazbu na parenta a tuto vazbu proto nemohou sdílet. Příklad na editor můžeme chápat jako takto zavedenou kombinaci obou vzorů.

Správa vnějších stavů

Při použití tohoto vzoru si je třeba uvědomit, že jsme namapovali logický problém jinak než 1:1. Znamená to ve svém důsledku, že musíme provést opětné mapování i u dalších operací, které přímo nesouvisí s uvedeným scénářem operace ve vzoru.

Například pokud někdo v dokumentu smaže znak, musí se provést určitý scénář mazání u objektu `Mapper` nad jeho zdrojem dat. Musíme tedy všechny scénáře pracující nějak v analýze s instancemi daných informací namapovat do odpovídajících struktur provázání mezi daty u objektu `Mapper` a daným scénářem použití objektů `Flyweight`.

Zvláštní klon vzoru FLYWEIGHT u rozsáhlých systémů s hlubokými kolekcemi

Jako zvláštní případ použití tohoto vzoru lze chápat zavedení optimalizace „omezeného scénáře izolovaného klienta“ (viz [1]). Představme si, že máme scénář v analýze popsaný takto (stačí odstavec jako ukázka):.

Nová faktura

... obsluze se zobrazí seznam firem, obsluha vybere firmu a ta se dosadí do faktury jako odběratel...

Pokud bychom mapovali tento scénář 1:1 i v instancích, nastane problém s tím, že firem je řádově více než 100 000. Proto se rozhodneme pro jinou realizaci tohoto scénáře (jiný design) než v instancích 1:1. Použije se vzor FLYWEIGHT.

Bude se pracovat pouze s jednou instancí firmy (to je odlehčení) a nikoliv se všemi. Jeden z možných scénářů (již v designu) je následující:

Formulář dostane od seznamu firem (ten je z hlediska instancí firem prázdný) skupinu dat pro zobrazení (formulář dostane kontext zobrazení). Pokud obsluha požádá o další data, tj. posune se v zobrazení, dostane další skupinu dat pro zobrazení. Až obsluha vybere, získá se identifikátor a pošle se jako požadavek zpět do kolekce. Ta vydá prvek podle tohoto identifikátoru, ale uvnitř existuje jeden item a ten se naplní z databáze. Jiná podobná možnost je, že se vytvoří instance firmy, předá se jí ID zavolá se její operace `LoadFromDB()` (resp. se vstupním parametrem ID) a naplní se z databáze.

V obou případech se použilo mapování nikoliv 1:1, ale mapování do designu na jeden „sdílený“ objekt (v paměti) a další údaje jsou jako její kontext v databázi. Provazujícím prvkem, přes který se realizuje mapování, je ID. Databáze zde hraje roli uložiště dat za objektem `Mapper` ve vzoru (viz obrázek vzoru). Sdíleným objektem je vybraný objekt a je v poolu (číselníku) jediný (v předešlém příkladu s editorem bylo takovýchto objektů tolik, kolik bylo písmen). Objektem `Mapper` (to je ten, kdo provazuje a dává správný kontext data versus objekt Flyweight), je buď seznam (první případ), anebo objekt sám (druhý případ s `Load()`). Vždy se ale používá pomocný služební objekt datové vrstvy.

Také zde platí „ztráta reference“. Pokud provedeme nový `Load()` s jiným ID, máme tentýž objekt s jiným obsahem (jiný kontext). Tuto skutečnost musíme ve scénářích zohlednit.

Další variantou tohoto klonu je zavést (poněkud složitější) kolekce s prvky a cache pomětí („cachované kolekce“).

Související vzory

Vzor FLYWEIGHT bývá často kombinován se vzorem COMPOSITE.

Mnohdy se používá pro implementaci vzorů STATE a STRATEGY (viz dále).

PROXY

Klasifikace

Object

Structural

Smysl

Zavádí zástupce resp. držitele ukazatele před objekt tak, že kontroluje přístup k objektu.

Alias

Surrogate (zástupce)

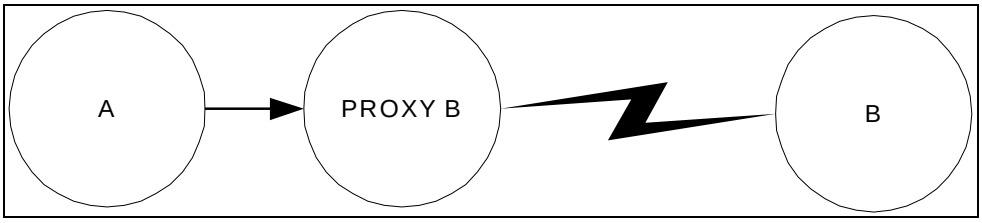
Motiv

Opět se jedná o vzor často používaný, takže není třeba motiv příliš rozvádět.

Klasický příklad je použití PROXY v remote technologiích na síti. Úkolem je vytvořit technologii přístupu ke vzdálenému objektu na jiném stroji. Objekt klient se nachází na jednom stroji a vzdálený volaný objekt na druhém stroji. Vyžaduje se, aby klient pracoval s tímto objektem stejně, jako by byl u něj na stroji (tj. volá stejným způsobem jeho operace).

Jako řešení se použije „podvrhnutí objektu spojky“ vydávající ven stejný interface jako původní objekt. Tento podvržený objekt (PROXY) uschovává technologii propojení mezi stroji.

Například klientovi (objektu A) na následujícím obrázku je „podvržen“ namísto objektu B jiný objekt a to objekt PROXY B, který podporuje stejný interface jako B. Klient tak používá objekt na jiném stroji stejně, jako by byl na stejném stroji jako klient:



obrázek 38 objektu A je podvržen PROXY B namísto B

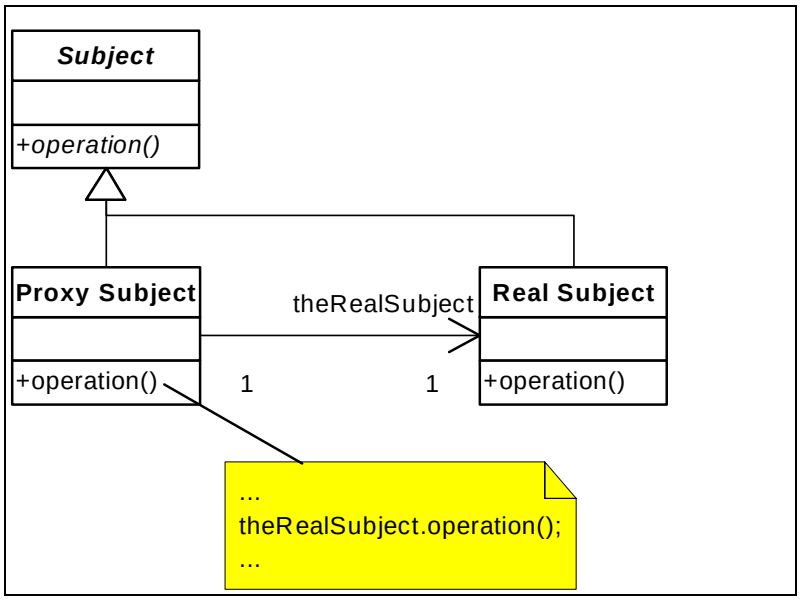
Pokud má být skutečně tento „podvrh“ realizovatelný, musí objekt PROXY B povinně podporovat interface objektu B (objekt a jeho podvrh musí být zaměnitelné, tj. kompatibilní).

Celá myšlenka vzoru PROXY je založena na používání takovéhoto „podvrhu“, kdy se na místo původního objektu dosadí objekt PROXY a ten jako neznámý host „něco mezi klientem a objektem provede“.

Poznámka: Technicky vzato musí u předešlého příkladu existovat i druhá strana propojení na druhém stroji, takže reference od PROXY objektu k původnímu objektu nemůže být přímo přes ukazatele, ale nepřímo přes nějaké hodnoty. Myšlenka podvrhu PROXY za původní objekt je však stejná.

Jediné, co se mění případ od případu, je důvod, proč se tento „podvrh“ děje. Zde to bylo volání vzdáleného objektu na jiném stroji.

Struktura vzoru



obrázek 39 Struktura vzoru PROXY

`ProxySubject` a `RealSubject` podporují stejný interface `Subject` s operací `operation()`. Na místo objektu `RealSubject` je dosazen objekt `ProxySubject` jako „podvrh“ a klient volá jeho implementovanou operaci `operation()`, aniž by to „věděl“. `ProxySubject` provede „něco skrytého při tomto volání“ a v určitém okamžiku může požádat o funkcionalitu `operation()` původního objektu `theRealSubject`.

Poznámky k použití

Seznam hlavních použití

Princip „podvrhu“ je vždy a všude stejný. Otázkou je, kdy se používá:

- *Remote proxy* (bylo uvedeno jako motiv). Většinou je tato technologie již zavedena ve vývojových prostředcích (JAVA, DOT.NET).
- *Protection proxy.* Na místo původního objektu je dosazen „PROXY objekt“, který má zabudováno volání přístupových práv s předmětem práv „povolení operací objektu“. Po vyhodnocení práva buď pokračuje voláním původního objektu anebo operaci objektu znepřístupní.
- *Virtual proxy*. Objekt PROXY nahradí původní objekt a provádí určité operace až ve chvílích, kdy jsou opravdu třeba (například naplnění daty - load obrázku z disku apod.). Po dobu „než třeba“ hraje skutečně PROXY pouze virtuální roli interfacu, za kterým je úplně „prázdný objekt bez dat“.

- *Smart reference*. Je určen k řízenému přístupu k objektům, například počítání referencí, zamykání objektů apod. Počítání referencí je již mnohdy řešeno v dané technologii prostředí. Zamykání objektů přes PROXY může napomoci při problémech přístupu ke sdíleným objektům.

Copy-on-write

Zajímavým postupem je technika copy-on-write používaná při kopírování objektů. Základní myšlenkou tohoto postupu je fakt, že není třeba kopírovat objekt do objektu do té doby, dokud nezačneme na kopii provádět změny. V té chvíli se provede kopírování a do té doby se používá „prázdný“ objekt s odkazem na původní objekt

Související vzory

ADAPTER je také spojka, ale se dvěma různými interfacy.

DECORATOR je podobný, tj. „spojka“, dokonce se stejným interfacem, ale v řetězu za sebou. Avšak důvod použití vzoru DECORATOR je jiný: DECORATOR přidává funkcionalitu, PROXY řídí přístup k objektu jako jeho zástupce.

Poznámka: Někdy je však otázkou, kde vlastně leží hranice mezi vzory. Někdy by dané konkrétní řešení mohlo být chápáno jako použití jednoho nebo druhého vzoru a těžko rozsoudit, kam vlastně řešení spadá. Například „protection proxy“ uvedené v této kapitole se zabudovanými právy přístupu k objektu by bylo možné považovat za použití vzoru DECORATOR, tj. rozšíření objektu o přístupová práva „předřazením“. Z praktického hlediska jsou tyto otázky spíše “akademické“.

Poznámka: Tímto jsme ukončili výklad o vzorech sloužících k řešení situací struktur objektů, tj. „Structural Patterns“.

CHAIN OF RESPONSIBILITY

Klasifikace

Object

Behavioral

Smysl

Zavádí flexibilně měnitelný řetěz objektů propojených stejným interfacem k předání a zpracování nějakého požadavku. Klient může odeslat požadavek libovolnému objektu z tohoto řetězu a tento požadavek bude v řetězu zpracován.

Motiv

Při tomto vzoru si vždy s úsměvem vzpomenu na problém „putování po úřadech“. Představme si, že úřad bude fungovat tak, že každý úředník na uvedený požadavek bude reagovat podle algoritmu: „Když umím, zpracuji, a když neumím, tak v tom případě vím, kdo umí, a pošlu mu tento požadavek“. Úsměvná je rekurze tohoto „úředního algoritmu“: Ten druhý v řadě, komu je posláno (další v pořadí), pracuje podle téhož algoritmu: „Když umím, zpracuji, a když neumím, tak v tom případě vím, kdo umí, a pošlu mu tento požadavek“. Důsledkem je putování po úřadech od čerta k ďáblovi.

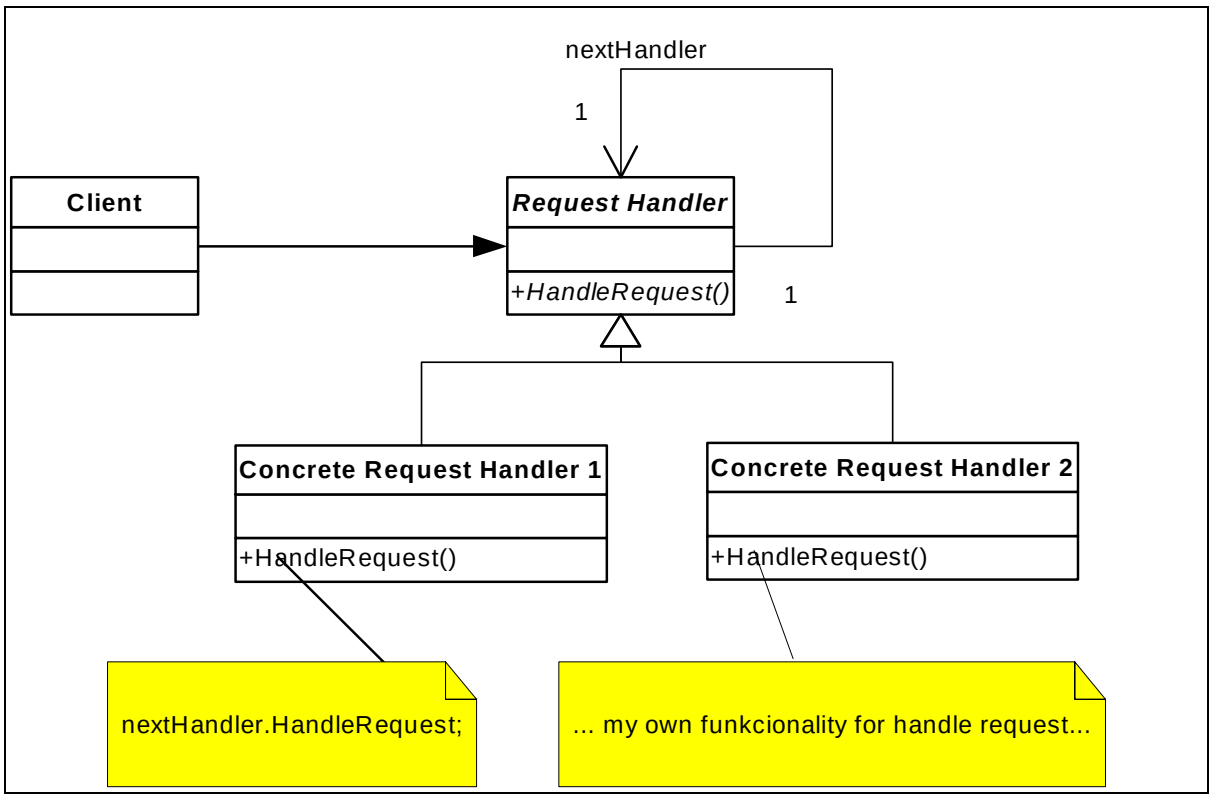
Poznámka: Nikdy mne nenapadlo, že lidové rčení „od čerta k ďáblovi“ vlastně vyjadřuje kompatibilitu interfaců.

Stejně pracuje vzor CHAIN OF RESPONSIBILITY (řetěz odpovědnosti). U našich úředníků si musíme představit (stejně jako v OOP), že všichni mají stejný „interface“ (jsou zvně k nerozeznání) a vnitřní implementaci při posílání požadavku nevidíme. V tom případě je tento algoritmus flexibilní ve dvou směrech:

- Každý úředník může být osloven vnějším klientem a umí požadavek splnit
- Každý úředník se může stát klientem jiného úředníka

V literatuře se většinou uvádí jako motiv zpracování Helpu. Představme si, že konstrukce Helpu nad GUI strukturou bude taková, že u každého prvku bude požadavek na zobrazení Helpu. Pokud tento prvek Help má, zpracuje se. Pokud jej nemá, potom pošle tento požadavek jinému objektu, což bude jeho parent v GUI (majitel prvku). Každý prvek GUI postupuje tímto algoritmem. V některých případech může nastat situace, že „nikdo nemá help“, takže požadavek proputuje až nahoru k nejvyššímu prvku a zobrazí se nějaké „unifikované hlášení“ resp. Help celé aplikace apod.

Struktura vzoru



obrázek 40 Struktura vzoru CHAIN OF RESPONSIBILITY

Abstraktní třída `RequestHandler` zavádí interface společný pro všechny prvky stromu. Objekty z tohoto řetězu pocházejí ze tříd `ConcreteHandler`. Některé prvky „ukončují“ řetěz svou vlastní funkcionalitou, některé posílají požadavek dál pro následníka `nextHandler`.

Poznámky k použití

Flexibilita a dynamicky tvořené řetězy

Všimněme si, že řetěz lze poskládat dynamicky (nikoliv staticky). Znamená to, že lze řetěz konfigurovat v runtime. Navíc díky zavedení společného interfacu lze problém zpracování požadavku nad každým objektem `Handler` řešit sice unifikovaně (implementuje se stále tatáž operace), ale u každého prvku odděleně a nezávisle na jiných prvcích.

Chybné řetězy

Předešlá flexibilita (jako každá flexibilita) v sobě uschovává riziko chybných konfigurací. Mohlo by se stát, že sestavíme řetěz tak, že nakonec nedojde vůbec ke zpracování požadavku, tj. na konci řetězu dojde k delegování na prázdnou referenci. Také můžeme řetěz zacyklit (poznámka: to je velmi běžné v užití tohoto vzoru na úřadech) anebo ukončíme řetěz a přitom za jeho konec zbytečně skládáme další prvky. Tyto situace jsou daň za flexibilitu, měli bychom na ně pamatovat a případně je ošetřit.

Vzor COMPOSITE a link na next handler

Při aplikaci vzoru CHAIN OF RESPONSIBILTIY se většinou pro vazbu následníka (tj. `nextHandler`) použije již existující vazba, protože zavedení tohoto řetězu je již dáno z povahy věci. Například ve vzoru COMPOSITE je struktura objektů již dána a vazba v CHAIN OF RESPONSIBILITY může díky logice věci splynout s odkazem na parenta prvku stromu. Tento případ například nastal u GUI prvků v příkladu Help. Poskládané GUI prvky v kompozicích lze totiž chápat jako aplikování vzoru COMPOSITE a delegování zpracování helpu odpovídá přesně této struktuře.

Někdy se může stát, že pro použití vzoru CHAIN OF RESPONSIBILITY je třeba zavést vazby mezi objekty zvlášť. V každém případě se však jedná o běžnou asociaci, tj. reference na `nextHandler` se provádí dosazením.

Související vzory

Vzor CHAIN OF RESPONSIBILTY bývá většinou aplikován spolu se vzorem COMPOSITE. Požadavek delegovaný na následníka v tom případě putuje podél linku na parenta.

COMMAND

Klasifikace

Object

Behavioral

Smysl

Zapouzdřuje požadavek do podoby objektu (objektové proměnné), takže lze s požadavkem pracovat jako s každou jinou proměnnou, což vede k možné parametrizaci požadavků, dynamická tvorbě seznamu požadavků, dosazení požadavku za jiný apod.

Alias

Action

Motiv

Tento vzor je tak „silný“, že motivů pro tento vzor můžeme uvést hned několik.

V jednom z předešlých příkladů u vzoru COMPOSITE jsme učinili poznámku, že při zpracování návratové hodnoty značící error bychom měli tuto hodnotu nějak ošetřit a podle ní se zachovat a to pokud možno flexibilně. Zadejme si jako motiv tento příklad: Flexibilně zpracovat návratovou hodnotu.

Klient objektu volá operaci a ta vrací hodnotu jako integer (např. konstantu error apod.) a na základě této hodnoty se máme dále rozhodnout, co se bude dít. Pokud bude návratová hodnota 1, potom se volá nějaká operace objektu A zahajující scénář zpracování, pokud je návratová hodnota 2, volá se nějaká jiná operace jiného objektu B, která zahajuje scénář jiného zpracování. Chceme, abychom mohli flexibilně přidat další hodnotu s dalším zpracování.

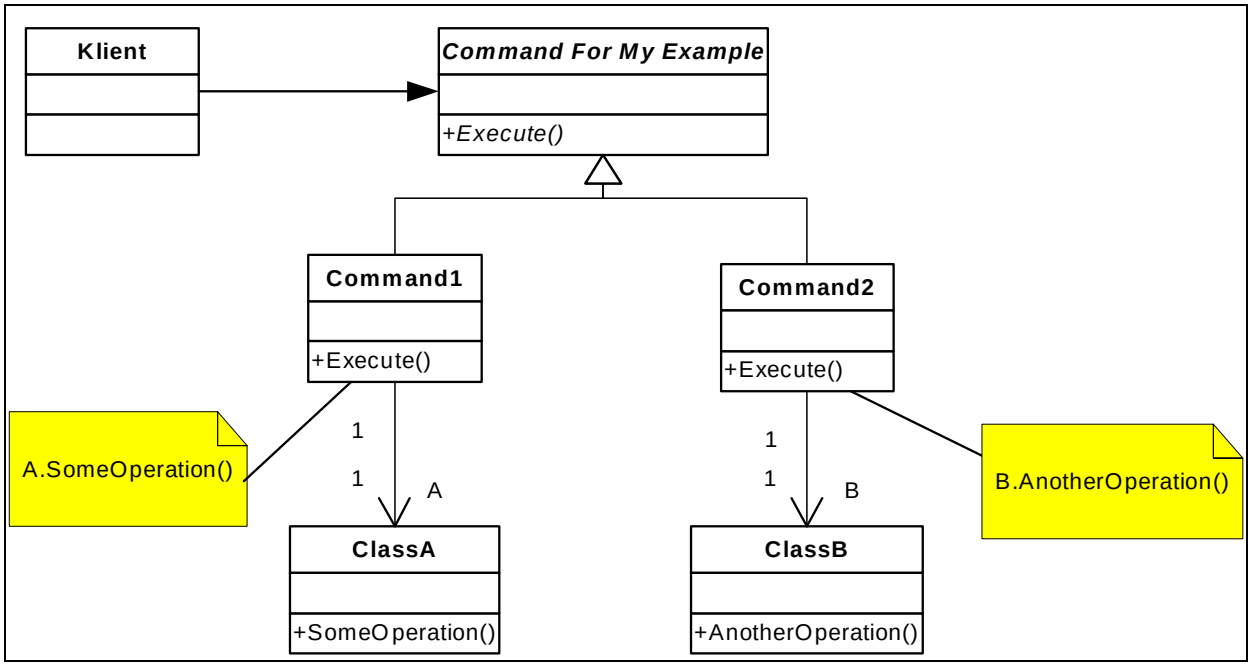
Původní řešení se switchem by vypadalo nějak takto:

```
...
switch Hodnota; //toto je návratová hodnota...
  {
   1 : A.SomeOperation;
   2 : B.AnotherOperation;
 }
```

Toto řešení je sice funkční, ale není flexibilní. Přidání nové hodnoty znamená přepsat tento kód. Jedno řešení jsme již nastínili: Toto zpracování bude polymorfně přepsáno jinou metodou nějakého potomka, jak bylo ukázáno u řešení polymorfní operace `Make(ID)` u klonu vzoru FACTORY.

Existuje však jiné a lepší řešení využívající vzor COMMAND. Spočívá také (jak jinak) ve využití polymorfismu.

Zaveďme abstraktní třídu `CommandForMyExample` s operací `Execute()`. Pro každé zpracování návratové hodnoty zavedeme dědice této třídy a implementujeme operaci `Execute()` podle toho, co se má v dané větvi rozhodnutí odehrát. Prvního dědice označme jako `Command1` a implementace jeho metody `Execute()` bude kód `A.SomeOperation`. Druhého dědice označme jako `Command2` a implementace jeho metody `Execute()` bude kód `B.AnotherOperation`. Model tříd tohoto řešení ukazuje následující obrázek:



obrázek 41 Model tříd pro řešení příkladu pomocí vzoru COMMAND

Všimněme si, že třídy `Command1` a `Command2` přepisují `Execute()` a „velmi tence“ obalují původní objekty A a B. Polymorfním přepsáním metody `Execute()` se „unifikuje“ volání operací na jednotný a kompatibilní způsob volání jediné operace.

Výběr ze tříd `Command1` nebo `Command2` (odpovídají dvěma hodnotám 1 a 2) převedeme na výběr instancí z těchto tříd. Zaveďme objekt `CommandManager` jako SINGLETON, který umí registrovat nový `Command` operací `Add()` a umí vydat příslušný `Command` podle klíče. Zaregistrujeme každý potřebný `Command` do tohoto seznamu operací `Add()`:

```
...
Command1 MyCommand1 = new Command1();
MyCommand1.SetA(MyA);
CommandManager.Add(MyCommand, key = 1);
...
```

podobně pro `Command2`

```
...
Command2 MyCommand2 = new Command2();
MyCommand2.SetB(MyB);
CommandManager.Add(MyCommand2, key = 2);
...
```

Zpracování návratové hodnoty můžeme potom převést na volání operace `Execute()` (poznámka: objekt `CommandManager` má nějak ošetřen výstup nenalezeného klíče, například vrací „žádný objekt“ jako `nil` a podobně):

```
...
Command MyResultCommand = CommandManager.GetItem(Hodnota);
If MyResultCommand <> nil then MyResultCommand.Execute();
...
```

Vzhledem k zavedení polymorfismu a přepsání metody `Execute()` proběhne po zavolání `Execute()` implementovaná metoda podle vybraného objektu daného podtypu `Commandu`.

Přidání resp. odebrání prvku ze seznamu objektů `Command` v objektu `CommandManager` změnilo původní „neflexibilní“ switch na flexibilní a dynamické „handlování“ s instancemi objektů `Command`.

Podobný motiv bychom nalezli u problému dynamického poskládání GUI prvků. Pokud například v MenuItemech napíšeme „natvrdo“, co se má odehrát (tj. rovnou na click: `A.SomeOperation()`), nelze takovéto menu budovat flexibilně. Převedení na vzor `Command` by mohlo fungovat i v tomto případě, pouze je třeba datově provázat dané MenuItemy s odpovídajícími objekty `Command`.

Poznámka: Existuje ještě řešení pomocí vzoru MEDIATOR.

Představme si číselník MenuItemů uložený např. v tabulce „název, text pro uživatele, kód Command“. Struktura se načte, vybuduje se menu podle textů a volá se při kliknutí pouze jeden a ten stejný kód:

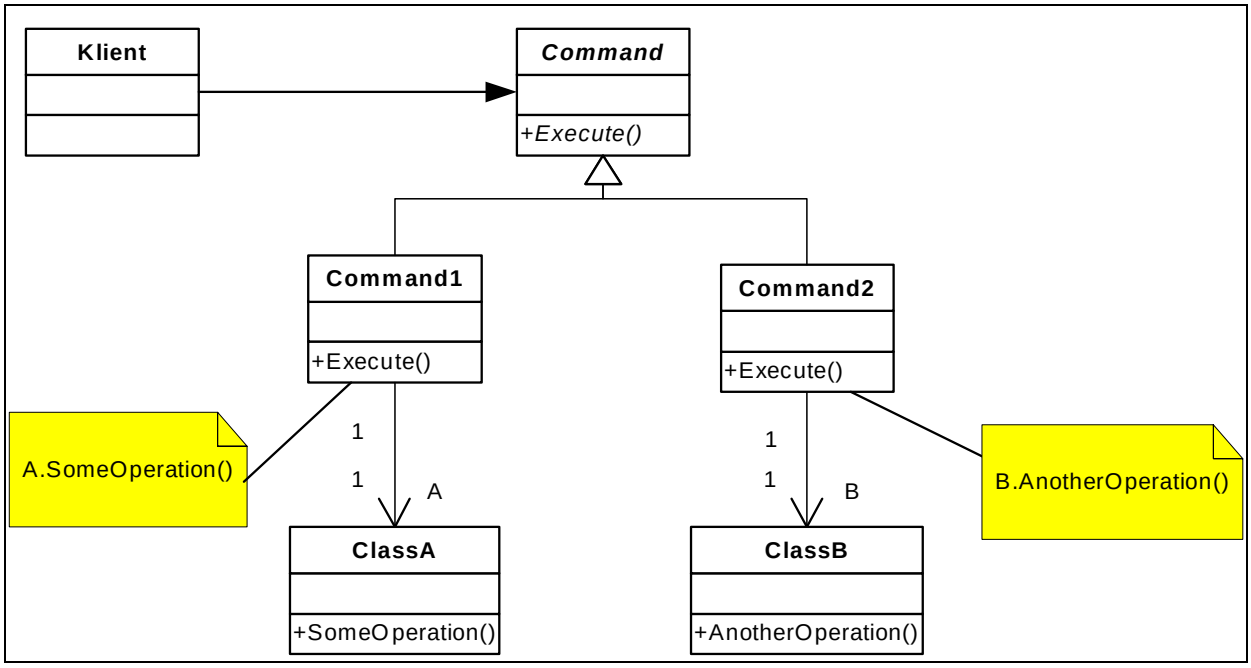
```
// onclick Menuitem:
Command MyCommand = CommandManager.GetItem(MenuitemKodCommand);
If MyResultCommand <> nil then MyResultCommand.Execute();
```

Další motiv najdeme u uživatelsky konfigurovatelných akcí, jako je měsíční uzávěrka systému apod. Uživatel si chce sám poskládat nějakou akci z jiných akcí a tuto konfiguraci uložit. Vzor COMMAND mu toto umožňuje.

Jiný příklad: Podobně jako u zpracování návratových hodnot chceme vyvolávat akce při zpracování chyb, pádech systému apod. a tato volání akcí chceme zavádět flexibilním způsobem (konfigurací apod.).

Struktura vzoru

Struktura vzoru je uschována již v předešlém obrázku, pouze zobecníme konkrétní příklad do posice vzoru:



obrázek 42 Struktura vzoru COMMAND

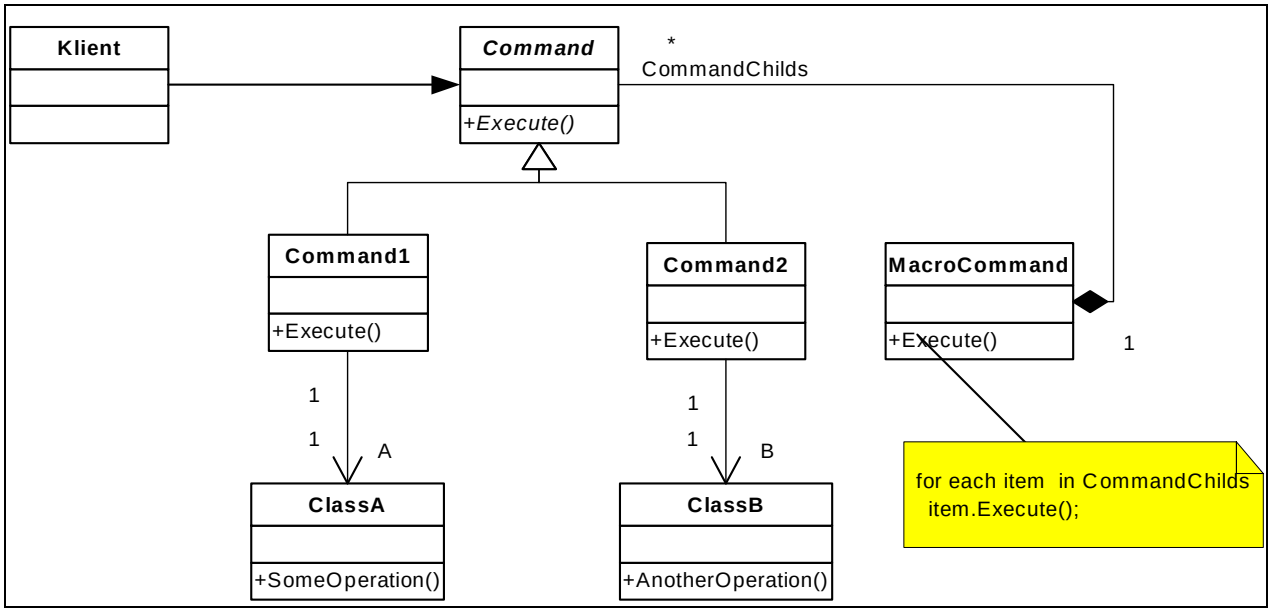
Klient je odstíněn od konkrétního volání operací objektů ze tříd `ClassA` a `ClassB` a volá jednotně interface zavedený abstraktní třídou `Command`. Dědicové přepisují metodu `Execute()` a tím vyvolávají konkrétní zpracování.

Z pohledu klienta se jedná o unifikovaný požadavek, se kterým lze pracovat jako s objektovou proměnnou.

Poznámky k použití

Výhody použití Commandu

- umožňuje přidávat flexibilně nové třídy Command.
- je možné s objekty ze tříd dědiců Command pracovat jako s proměnnými (seznamy, výběr, vazba do persistence apod.).
- odděluje klienta od realizace požadavku, například proto lze flexibilně vyměnit klientovi objekt Command „pod rukou“, aniž by to pocítil.
- kombinací se vzorem COMPOSITE lze objekty Command skládat do (rekurzivních) MacroCommandů, viz obr.:



obrázek 43 Zavedení MacroCommandu pomocí vzoru COMPOSITE

Procedurální programování a Command

Obdobou vzoru COMMAND v procedurálním programování je volání funkcí „přes pointer“, se kterým lze „handlovat“ (callback apod.). Například v Delphi je obdobou COMMAND použití proměnné typu `procedure`.

Hloubka funkcionality a složitost Commandu

V motivech jsme zavedli objekty ze tříd Command, které pouze přesměrovaly volání na operace jiných objektů.

Mohou existovat takové objekty Command, které nedelegují funkcionalitu na jiné objekty, ale navíc samy „něco vykonají“. Obecně objekt Command může mít své interní stavy související s vykonáváním operace.

Undo operace

Objekt ze třídy `Command` kromě `Execute()` může obsahovat i operaci `Undo()`, která vrací objekt do původního stavu před voláním `Execute()`. (Blíže o `Undo()` viz vzor MEMENTO.) Vykonané objekty `Command` lze řadit do kolekce „hotových“ objektů `Command` („history Command list“). Můžeme se pohybovat se po této kolekci kursorem a získávat zpět původní stavy objektů před voláním operací (násobné *Undo*). Je zřejmé, že do tohoto seznamu se musí ukládat kopie původních vykonaných objektů Command, nikoliv původní objekty Command. O operaci `Undo()` bude blíže pojednáno u vzoru MEMENTO.

Související vzory

Vzor COMPOSITE zavádí `MacroCommand`.

Vzor MEMENTO se používá pro Undo operace.

Pro vkládání kopie do history listu je výhodné použít vzor PROTOTYPE.

INTERPRETER

Poznámka: Tomuto vzoru nebudeme v této knize věnovat až takovou pozornost, protože řeší velmi speciální případ „zavedení vlastního interpreteru“. Navíc tento interpret nesmí mít příliš složitou gramatiku. Uvádíme jej pro úplnost.

Klasifikace

Class

Behavioral

Smysl

Zavádí reprezentaci gramatických pravidel pomocí modelu tříd a k tomu zavádí odpovídající interpret pomocí operací objektů z těchto tříd.

Motiv

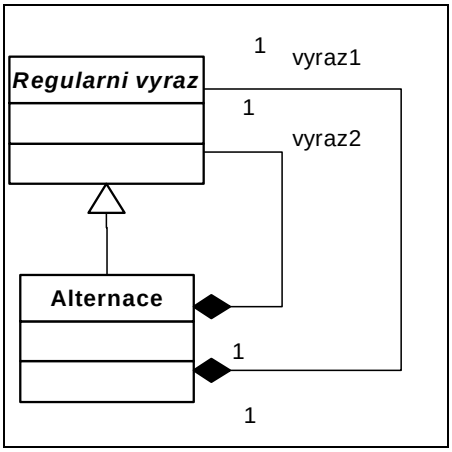
V některých případech je vhodné neřešit opakující se partikulární problémy znovu a znovu, ale je lepší zavést „obecné řešení na meta úrovni“ (pokud to lze). Konkrétní partikulární problém se poté řeší jako konkrétní instance tohoto řešení. Takto zobecněné řešení však vede k problému „gramatiky“ ve smyslu tvorby určitých pravidel a poté k interpretaci těchto pravidel v konkrétní instanci řešení.

Vzor INTERPRETER zavádí způsob, jak definovat jednoduchou gramatiku pro jednoduché jazyky a jak ji interpretovat na konkrétní instanci. Vzor zavádí vztahy na meta úrovni (pravidla) pomocí vztahů mezi třídami a vztahy mezi instancemi jsou konkrétní aplikací těchto pravidel. Například alternace dvou výrazů je objekt obsahující dva objekty apod. Všechny třídy patří do téhož stromu dědičnosti a mají tutéž operaci pro práci s instancemi (například `Match()` kontroluje shodu vybudovaného řetězce se syntaxí). Po vybudování objektového modelu ze třídy z dané instance se operace „prožene“ jednotlivými prvky a tím se struktura interpretuje.

Příklad:

```
alternace =  ( výraz | výraz )
```

Tomu v modelu tříd odpovídá následující vztah:

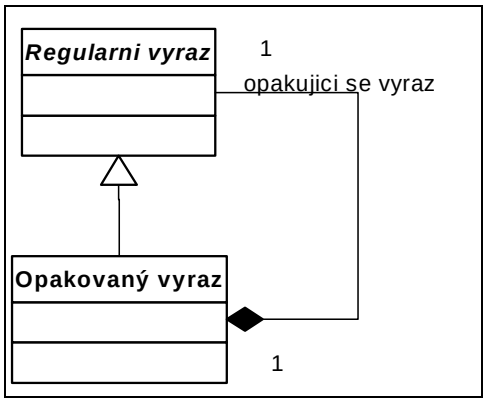


obrázek 44 Výraz Alternace v modelu tříd výrazů

Podobně bychom mohli definovat výraz typu „opakování výrazu“:

`opakovaný výraz = repeat (výraz)`

a odpovídající model tříd:



obrázek 45 Výraz Opakovaný výraz v modelu tříd

Všimněme si, že předek obou typů výrazů (opakování a alternace) je společný, takže například instancí opakujícího se výrazu může být například opět objekt ze třídy `Opakovaný výraz` anebo objekt ze třídy `Alternace` (např. můžeme takto složit opakující se alternaci).

Další výraz bude například `Sekvenční výraz`:

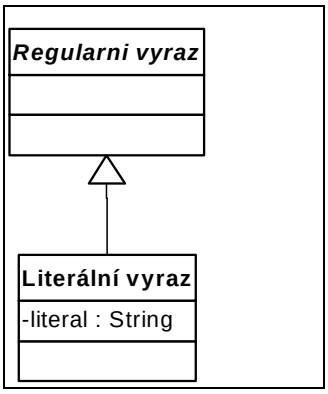
`sekvenční výraz = ( výraz & výraz )`

Obrazem bude třída obsahující vztah kompozice na dvě instance třídy `Regulární výraz`.

Zvláštní postavení mezi těmito regulárními výrazy mají tzv. terminální symboly (stavební kameny výrazu). V našem příkladu můžeme zavést podtyp literal (což je obdoba string):

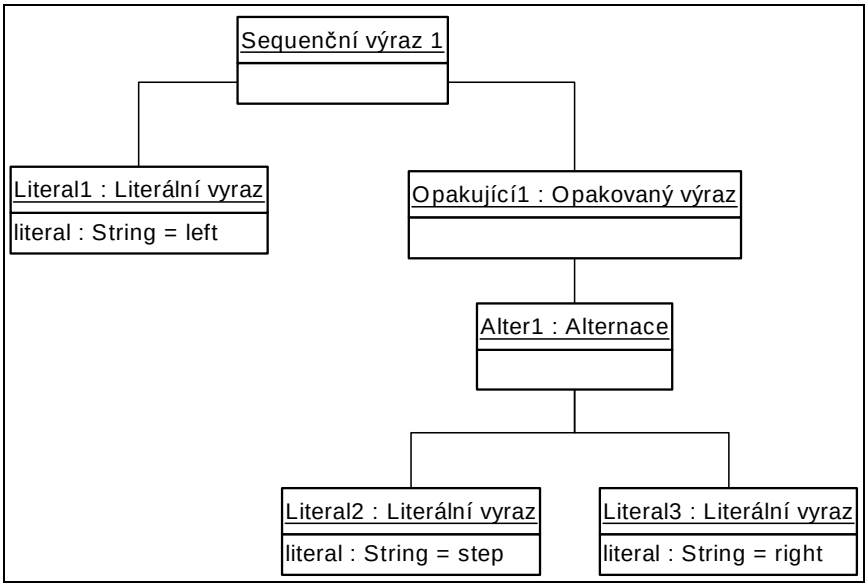
`literal = ‘a’ | ‘b’ | ‘c’ | … repeat (‘a‘ | ‘b‘ | ‘c‘...)`

`literal` nechá vzniknout dalšímu podtypu třídy `Regulární výraz`, třídě `Literalní výraz`:



obrázek 46 Výraz typu Literální výraz

Pokud dáme celý model tříd dohromady (může obsahovat ještě další třídy jako pravidla), zjistíme, že lze z výrazů jako objektů z těchto tříd postavit stromovou strukturu. Listy stromu jsou objekty ze třídy `Literární výraz`, nad ním stojí ve skládaných strukturách skladby jako interakce výrazů u uvedených pravidel. Určitý objektový model jako příklad namalujeme v této podobě:



obrázek 47 Příklad instancí regulárního výrazu (objektový model) ve tvaru:
`left & repeat (step | right )`

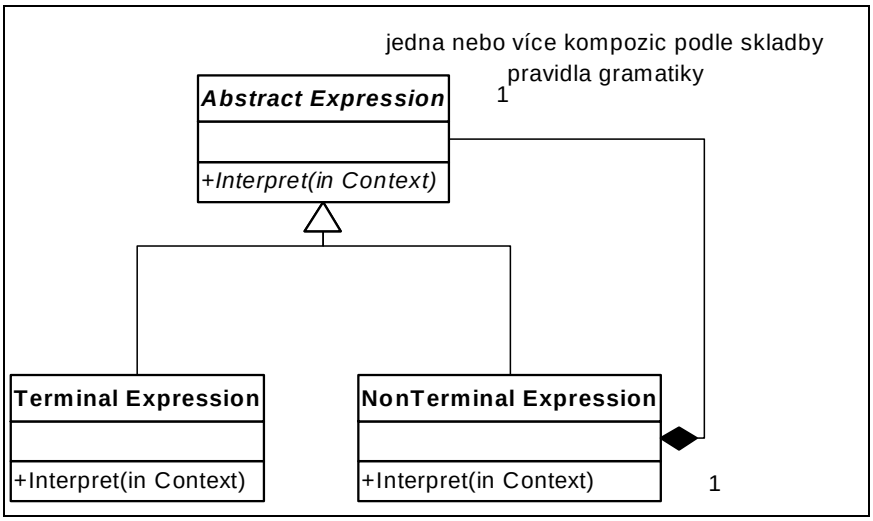
Poznámka: Čáry v diagramu označují spojení objektů (instancí) a jsou instancemi asociací (typu kompozice) mezi třídami.

Uvedený obrázek odpovídá v textu Regulárnímu výrazu:

```
left & repeat (step | right )
```

Nad tímto stromem může pracovat polymorfní operace, například `Match()` apod., která projde celým stromem uzel po uzlu. Takovou operaci lze zavést například i pomocí vzoru VISITOR (viz další kapitoly).

Struktura vzoru



obrázek 48 Struktura vzoru INTERPRETER

Vzor INTERPRETER vytváří strom, jehož listy jsou objekty ze třídy `TerminalExpression`, uzly jsou objekty ze tříd `NonTerminalExpression`, což jsou objekty s interakcemi odpovídající pravidlům gramatiky. Všechny uzly stromu znají polymorfní operaci `Interpret()`, která interpretuje výraz průchodem celým stromem.

Poznámky k použití

- Vzor není vhodný pro složité gramatiky, protože každé pravidlo zavádí nějakou třídu, což vede k velkému počtu tříd.
- Vzor je extenzivní v přidání nového pravidla přidáním další třídy.
- Vzor neukazuje, jak vybudovat strom objektů daného výrazu, ale jak jej „využít“, když už je vybudován. Například pokud bychom dostali k dispozici výraz z předešlého příkladu jako: `left & repeat (step | right )`ve tvaru textu, tak vzor neříká, jak z něj dostat odpovídající strom ukázaný jako objektový model na obrázku. Vzor ukazuje, jak jednoduše zavést jeho interpretaci. Museli bychom tento strom „nějak“ vybudovat bez pomoci tohoto vzoru (parser apod.).

Související vzory

Strom regulárního výrazu je tvořen jako COMPOSITE.

Pro průchod stromem výrazu lze použít vzor ITERATOR (viz dále).

Ne-terminální symboly mohou být optimalizovány pomocí FLYWEIGHT (viz poznámka o COMPOSITE ve vzoru FLYWEIGHT zavedené jako sdílené listy stromu).

Pro zavedení operací nad prvky stromu lze použít vzor VISITOR (viz dále).

ITERATOR

Klasifikace

Object

Behavioral

Smysl

Zavádí pro klienta jednoduchý a přehledný způsob sekvenčního přístupu ke složité struktuře objektů, přičemž tato struktura zůstane klientovi skryta.

Alias

Cursor

Motiv

Myšlenka vzoru ITERATOR je jednoduchá: Zaveďme interface s operacemi sekvenčního průchodu strukturou, jako jsou operace `First()`, `Next()`, `EOF()` (konec seznamu), `Item()` (vrací současný prvek) apod.

Předřaďme tento interface před složitou strukturu. Klient bude přistupovat ke složité struktuře nikoli přímo, ale přes interface s těmito operacemi. Výsledkem je (mimo jiné) zjednodušený pohled klienta na jinak složitou strukturu a její snadné ovládání.

Nutno podotknout, že tato myšlenka je využita již ve vývojových prostředích a jazycích (C#, JAVA, Delphi), které nabízejí objektové struktury nějakých seznamů (TList, arraylist apod.). Klient má k dispozici určitý interface s možností sekvenčně procházet struktury a přitom tato struktura seznamu objektů je klientovi skryta. Průchod seznamem se jeví klientovi natolik jednoduchý, že si ani nepředstavuje, jak složité struktury jsou za tímto interfacem skryty.

Aplikace vzoru ITERATOR však nekončí pouze zavedením „klasických seznamů“ vývojového prostředí. Vzor nejlépe využijeme tehdy, pokud se zamyslíme nad našimi vlastními strukturami a jim předřadíme objekt s interfacem podle vzoru ITERATOR. Nabídneme tak naše uživatelské struktury klientům nikoliv ve své složitosti, ale jednotně s možností jednoduchého přístupu k těmto strukturám.

Navíc se někdy může jevit jako výhodné použít již existující interface zavedeného ITERATORU daného prostředí. Našemu objektu můžeme „vnutit“ tento interface z prostředí. Struktura tak zapadne do rodiny ostatních seznamů prostředí (například díky tomu bude umět `for each` příkaz apod.).

Ukažme si zavedení ITERATORU na našem příkladu se složenými obrazci. Je zřejmé, že není problém procházet u složených obrazců jeho dětmi. Tak trochu složitější se může jevit problém, jak projít všechny jeho potomky (a to včetně dětí jeho dětí). V tom případě řešíme vlastně úlohu „průchod stromem“. Musíme samozřejmě znát algoritmus tohoto průchodu.

Poznámka: Těchto algoritmů je několik, například nejprve projdi děti na jedné úrovni jako sourozence, potom projdi děti těchto dětí, anebo vždy nejprve projdi děti dětí a potom teprve sourozence

Klient nebude řešit tento algoritmus přímo nad strukturou, ale sami mu algoritmus předřadíme před naší strukturu a nabídneme mu interface. Klient dostane tento interface algoritmu průchodu k dispozici.

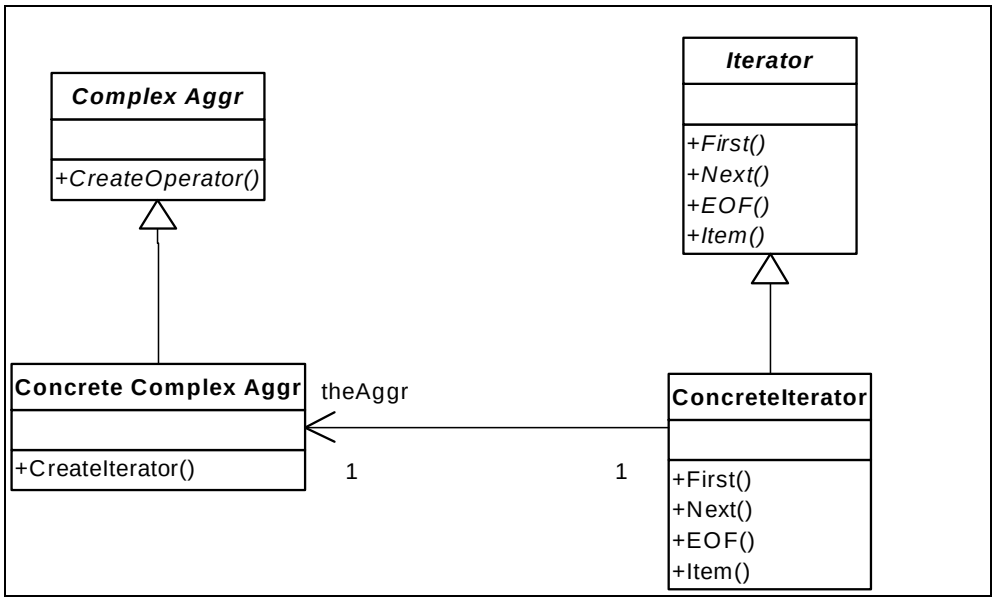
Například voláním operace nejprve `First()` a poté několikrát `Next()` až po `EOF()` projde všechny potomky (včetně potomků potomků) složeného obrazce sekvenčně, aniž by „věděl a tušil“, že vlastně prochází stromem.

Nejenže má klient k dispozici jednoduchý způsob průchodu stromem, ale může například v případě potřeby zavést najednou dva a více objektů `Iterator` nad strukturou a pracovat se dvěma a více kursory najednou.

Otázkou je, kdo vlastně má nechat objekt `Iterator` zrodit. Klient by měl být od tvorby objektu `Iterator` ušetřen. Jako dobré řešení se jeví nechat zrodit `Iterator` přímo od dané struktury, tj. klient může u složeného obrazce zavolat operaci `CreateIterator()`, jehož návratovou hodnotou je nový `Iterator`.

Navíc při použití vzoru FACTORY METHOD můžeme získávat různé algoritmy průchodu se stejným interfacem `Iterator`.

Struktura vzoru



obrázek 49 Struktura vzoru ITERATOR

Klient zvolí konkrétní agregátor, se kterým pracuje (tj. volí `ConcreteComplexAggr`), a požádá jej o `Iterator`. Dostane odpovídající konkrétní `Iterator`, kterému je při zrodu do instance `theAggr` dosazena reference na konkrétní agregátor, tj. při tvorbě iterátoru se dosadí za `theAggr` objektová reference `this`. Klient používá pouze interfacy `ComplexAggr` a `Iterator`.

Poznámka: Jsou možné i jiné kombinace spolupráce při zrodu Iteratoru (viz Creational Patterns). Osobně bych považoval použití vzoru FACTORY METHOD v této podobě za určité omezení. Dovedu si představit zavedení vstupního parametru v operaci `CreateIterator(ID_Iterator)`, který může zabezpečit vazbu jedna ku N ve smyslu „složený objekt versus algoritmus iterator“. Jeden složený objekt může mít N algoritmů průchodu.

Poznámky použití

Případy použití vzoru

Vzor ITERATOR použijeme tehdy, když

- chceme ukrýt složitou strukturu složeného objektu, a přitom chceme nechat klientovi možnost přistupovat jednoduše k prvkům složité struktury sekvenčně.
- když chceme klientovi nabídnout možnost přistupovat pomocí kurzoru, tj. průchodu s „držením cesty a stavu“ průchodu (to je vlastně pseudonym pro zavedení kurzoru) resp. když chceme, aby klient mohl přistupovat ke struktuře s násobným kurzorem.
- pokud chceme zavést polymorfní průchod různými typy struktur.
- pokud chceme zavést efektivně re-use algoritmů průchodu stejnými typy struktur. Představme si, že vyřešíme náš případ se složeným obrazcem a zavedeme pro složené obrazce Iterator. Poté budeme řešit průchod adresáři a soubory na PC. Zjistíme (pochopitelně), že řešíme něco velmi podobného. Při správném návrhu budou mít Iteratory re-use do nějakého obecnějšího Iteratoru pro obecný průchod stromem.
- Iterator může podporovat i další komfortnější operace, jako `SkipTo(Index)`, `Previous()` apod.
- Pokud provedeme změny ve složeném objektu „za iteratorem“, je třeba zvážit, zda se může tato změna projevit také v Iteratoru, zejména v případě indexovaných Iterátorů apod. Iterator, který dobře reaguje na změny svého složeného objektu, se nazývá „robustní Iterátor“.
  K řešení tohoto problému synchronizace Iteratoru a jeho procházeného objektu můžeme použít buď události (což podporuje každé OOP prostředí) anebo vzor OBSERVER (viz dále).

Související vzory

Struktury zavedené vzorem COMPOSITE se doplňují o funkcionalitu vzorem ITERATOR.

Pro tvorbu ITERATORU k danému vybranému složenému objektu lze použít nějaký vzor z Creational Patterns (FACTORY METHOD resp. PROTOTYPE)

Pro robustní ITERATOR lze využít vzor OBSERVER

MEDIATOR

Klasifikace

Object

Behavioral

Smysl

Zavádí objekt jako prostředníka mezi jinými objekty, který oddělí a následně zprostředkuje jinak složitou komunikaci mezi mnoha objekty, takže tyto objekty nemusí mít přímé vazby mezi sebou.

Alias

Controller

Motiv

V literatuře (např. [3], [4], [5]) se uvádí motiv spolupráce objektů v GUI. Pokud se změní (např. editací, výběr itemu prvku apod.) určitý GUI prvek, potřebujeme, aby se následně změnily i ostatní GUI prvky. V případě mnoha prvků a moha takovýchto komunikací přímo mezi prvky „každý s každým“ se tvorba GUI stává nepřehlednou a dochází k nepříjemnému mnohanásobnému provázání mnoha prvků mezi sebou.

Řešením je použití vzoru MEDIATOR. Zavede se objekt, který „vidí“ všechny komunikující prvky (jsou do něj dosazeny jejich objektové reference) a všechny prvky „vidí“ tohoto prostředníka. Tohoto prostředníka nazývejme Mediator. Při změně prvku volá tento prvek nějakou pro tento účel zavedenou specifickou operaci v Mediatoru a Mediator volá v této operaci ostatní prvky.

U tohoto motivu na GUI musíme být tak trochu opatrní, protože některá vývojová prostředí umožní problém tohoto motivu dobře zakrýt způsobem, jakým se v tomto prostředí buduje GUI. Pokud totiž použijeme nějaký designér GUI prostředí a vkládáme prvky GUI z nějakého *„GUI component tools“* vývojového prostředí metodou *„drag and drop“* (například v DELPHI, VB apod.), tak se většinou k událostem těchto prvků provazují metody formuláře a nikoliv přímo volání jiného prvku. Znamená to, že při změně prvku (resp. obecné události nad prvkem) se volají vždy operace formuláře a v nich se volají odpovídající operace jiných prvku GUI. V tomto případě vlastně sám formulář přebírá automaticky roli Mediatoru. Vzor je takto v tomto prostředí zaveden automaticky.

Avšak i v tomto případě lze doporučit u velmi složitých formulářů zavedení jednoho nebo více pomocných objektů Mediator a vyvést je jako součásti neviditelné části formuláře (jsou nazývány také jako Controllery) provazující funkcionalitu prvků. Samotný kód formuláře se tak velmi zjednoduší, protože složitý kód je vyveden ven do několika jiných objektů.

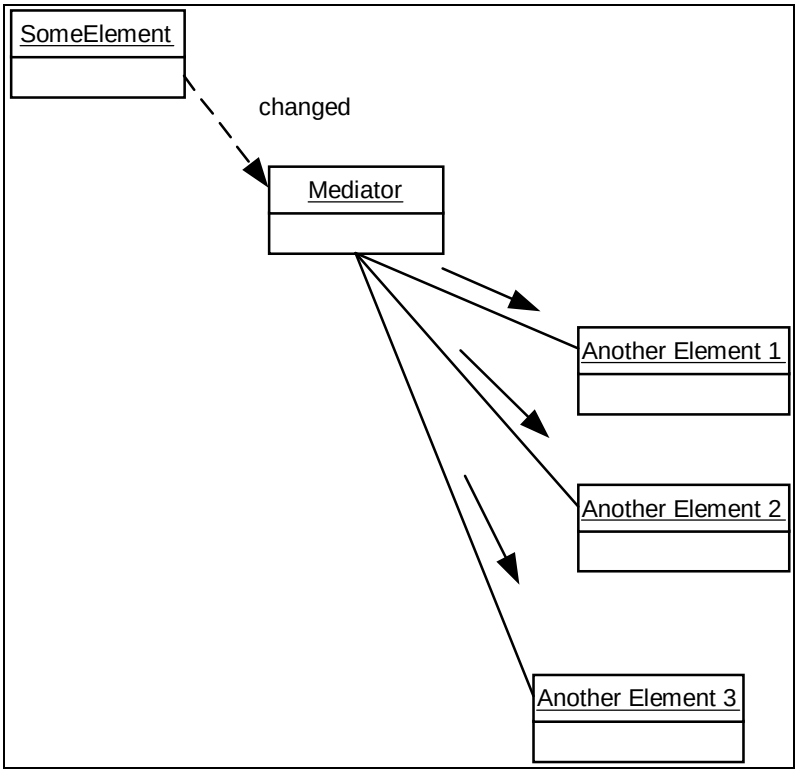
Použití vzoru MEDIATOR pro návrh GUI vynikne v tom případě, když použijeme klasickou konstrukci pomocí provázání událostí „listener s registrací“ (viz např. JAVA). V tom případě lze provazovat prvky GUI libovolně a použití vzoru MEDIATOR se ukazuje jako řešení oné možné složité situace komunikace „každý s každým“. Příklad na GUI je podrobně uveden v [3] a [4].

Následující složitější konstrukci musíme použít tehdy, když určitý prvek volá Mediator napřímo a bez události, tj. prvek volá rovnou operaci Mediatoru (nikoliv přes vyvolání události). Pokud použijeme události, potom odstíníme toto volání, pokud nepoužijeme události, každý prvek musí obsahovat instanci Mediatoru (aby ji mohl volat) a vidí jeho interface. Chceme-li zachovat obecnost řešení, tak pochopitelně v tom případě prvek musí volat unifikovanou operaci abstraktního interfacu Mediatoru, nazvěme ji `ElementChanged()`. V opačném případě, tj. při specifikaci konkrétní operace Mediatoru (například `MyMediator.ListboxOsobChanged()`), budeme situaci změny prvku řešit znovu a znovu pro další Mediatory a zavádět tak nové prvky s jinými instancemi Mediatoru, což není výhodné.

Vstupním parametrem operace musí být sám prvek (jak jinak mu říci kdo a co se změnilo) a pokud má být tedy tato operace opravdu „společná“ pro každý případ, musí být tento parametr typu obecný prvek. Celá jinak poměrně dost složitá konstrukce vyplývá jen a pouze z toho důvodu, že sám prvek „přímo vidí Mediator“ a „přímo jej volá“, tj. otázkou je, co vlastně musí libovolný prvek volat a koho (viz struktura vzoru a vysvětlení vzoru).

Této složité konstrukce s polymorfní operací `ElementChanged()` se můžeme zbavit v případě vyvolání události a jejím odchycení Mediatorem.

Struktura vzoru



obrázek 50 Struktura vzoru MEDIATOR v objektovém modelu

Při změně elementu `SomeElement` se u objektu `Mediator` spustí metoda, která obslouží ostatní elementy. Otázkou je, jak se spustí správná metoda objektu `Mediator`.

Buď tuto metodu volá přímo `SomeElement`, takže má zpřístupněn přímo interface objektu `Mediator`. V tom případě je do prvku vložena instance `Mediator`. Doporučuje se, aby existovala abstraktní třída `Mediator` zavádějící interface s unifikovanou operací `ElementChanged (Element)`. Pokud toto neučiníme, potom pro druhý jiný `Mediator` nemůžeme již připravený `SomeElement` použít. Pokud má konkrétní `Mediator` předka a je zaveden abstraktní interface, lze v objektu `SomeElement` vyměnit kompatibilní objekty z různých tříd konkrétních Mediatorů.

V případě, že se použije systém událostí, odpadají starosti, jak má vypadat abstraktní a konkrétní Mediator. Událost změny se prováže na metodu nějakého Mediatoru.

Poznámky k použití

Raději událost (listener a registrace) než přímé volání interfacu MEDIATORU

Nedoporučuji, aby elementy volaly interface MEDIATORU (byť zavedený na abstraktní úrovni) přímo, protože to zbytečně zesložiťuje řešení. Přehlednější je použít systém událostí v daném prostředí nebo vzor OBSERVER (viz dále).

Formulář jako MEDIATOR

V některých prostředích, jak bylo řečeno, jsou události obslouženy metodami formuláře. I tak se mnohdy doporučuje některý kód „vytlačit ven“ do druhého samostatně ve formuláři stojícího MEDIATORU, který se někdy nazývá CONTROLLER. Formulář potom pouze drží jako kontejner GUI prvky a jeden (nebo více) objektů Controllerů, které řídí logiku a obsahují kód obsluhy prvků. Potom ve formuláři není (takřka) žádný kód.

COMMAND a MEDIATOR

Pro vyvolání odpovídající metody lze použít také vzor COMMAND. Elementy při požadavku na spolupráci neoslovují MEDIATOR a ani nevyvolávají událost, ale volají vždy a jen nějaký `MyCommand.Execute()` a nic víc. Každá metoda Mediatoru je „obalena“ jednou třídou konkrétního `Command` s konkrétně implementovaným `Execute()`. Tato metoda `Execute()` je implementována tak, že volá pouze odpovídající metodu Mediatoru. Musí existovat tolik tříd konkrétních Commandů, kolik je potřebných metod Mediatoru (což může být nevýhodou použití tohoto vzoru). Například výměnou Commandu může daný prvek použít jiný Mediator. Pokud COMMAND nepředřadíme před Mediator, ale „začleníme“ přímo do Mediatoru, dostáváme podobné řešení, jako bylo původní řešení s voláním abstraktního interfacu Mediatoru.

OBSERVER a MEDIATOR

Pro vyvolání odpovídající operace Mediatoru lze použít vzor OBSERVER, který je de facto synonymem pro zavedení událostního systému v trochu jiném pohledu (viz dále u tohoto vzoru).

Související vzory

Jak bylo řečeno, pro volání metod MEDIATORU lze použít OBSERVER nebo COMMAND.

MEMENTO

Klasifikace

Object

Behavioral

Smysl

Aniž by došlo k porušení principu zapouzdření, objekt vydá svůj stav klientovi za účelem možnosti návratu tohoto objektu do původního stavu

Alias

Token

Motiv

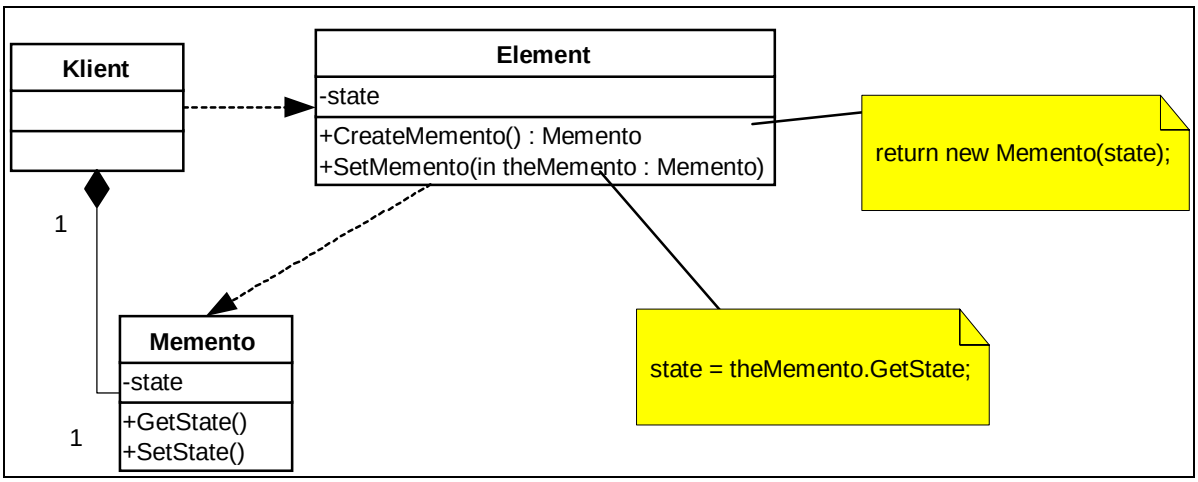
V mnoha případech nastává situace, že se u objektu zavolá určitá operace a poté se zjistí, že by bylo záhodno, aby se objekt vrátil do toho stavu, ve kterém byl těsně před zavoláním oné operace. Ukazuje se, že není dobrým řešením pro návrat do původního stavu zavádět u objektů inverzní operace. Mnohdy jsou totiž přechody inverzních operací zpět nepřesné, resp. „nevratné“.

Například při přesunu obrazce o nějaké hodnoty `deltaX` a `deltaY` operací `Move(deltaX, deltaY)` nemusí dojít k návratu zpět do původního stavu pomocí posunu o minus hodnoty, tj. `minus deltaX, minus deltaY`. Stačí malá nepřesnost řádu pixel anebo změna nějakého posice díky posunu a projeví se velké nepřesnosti. Například konektor mezi obrazci (vypočítaná spojnice od hran) mezi dvěma obrazci se posune do úplně jiné polohy (na vedlejší stěnu), než v jaké byl původně před posunem, při návratu zůstane na vedlejší hraně apod. Jednoduše řečeno spoléhat se na inverzní operace v programování není dobré řešení pro návrat do původního stavu.

Efektivnější a správnější je „zapamatovat si“ výchozí stav a poté se do něj vrátit jednoduše tak, že objekt se na základě tohoto „zapamatovaného“ stavu zpětně zrestauruje. Otázkou je, jak má vypadat správný scénář návratu do původního stavu.

Vzor MEMENTO je právě obecným, efektivním a jednoduchým řešením této situace.

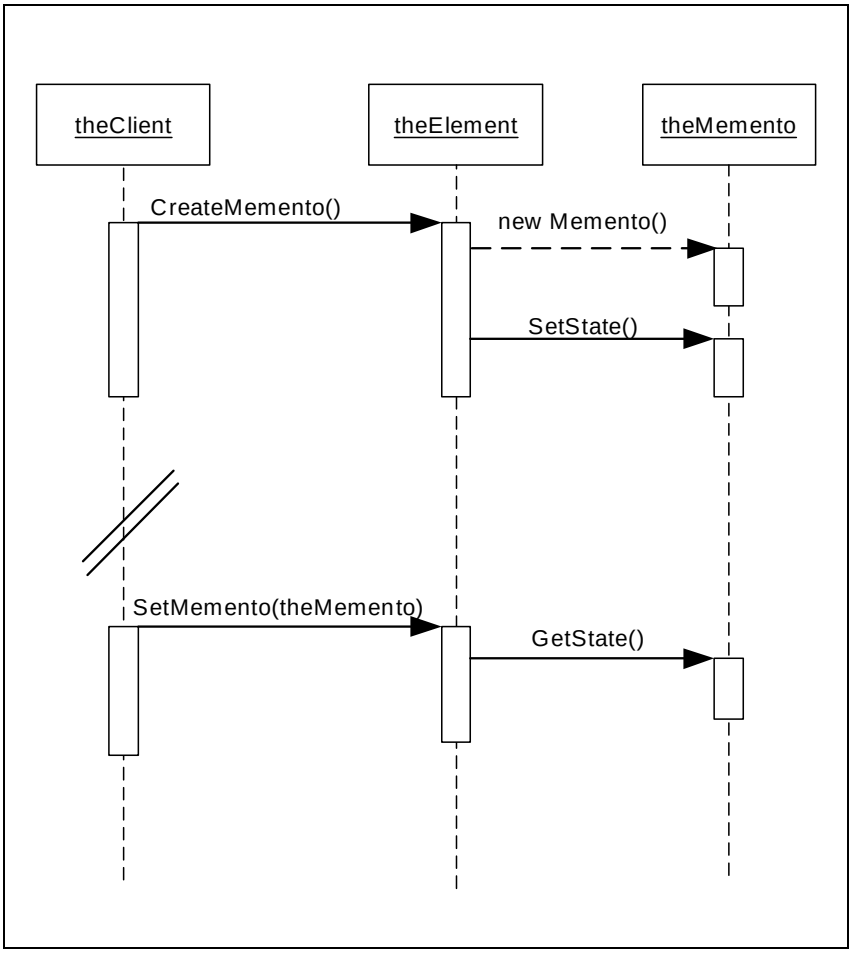
Struktura vzoru



obrázek 51 Struktura vzoru MEMENTO

Princip vzoru je velmi jednoduchý. Klient používá objekt ze třídy `Element` a rád by u něj zavolal nějakou operaci (není znázorněna), ale před tím je třeba podržet stav objektu pro pozdější návrat do tohoto stavu. Klient proto zavolá operaci `CreateMemento()`. Uvnitř objektu `Element` dojde k vytvoření nového objektu `Memento` a současně se zrodem se tomuto objektu `Memento` přesypou hodnoty vnitřních stavů objektu `Element`, které je třeba podržet pro pozdější obnovu (na obrázku znázorněno v kódu přímo jako parametr konstruktoru rodícího se objektu `Memento`, využije se přitom operace `SetState()`) . Objekt `Memento` se s obsahem stavu vydá ven a Klient si toto `Memento` podrží. Pokud Klient bude potřebovat návrat objektu `Element` do tohoto původního stavu, zavolá jeho operaci `SetMemento(Memento)` a jako vstupní parametr mu předá „zpátky“ to `Memento`, které obdržel při předešlém volání `CreateMemento()`. Uvnitř operace `SetMemento(Memento)` se převezme ze vstupního parametru `Memento` původní stav objektu (viz operace `GetState()`) a objekt si tento stav přesype do svého stavu `state`.

Scénář vyjádřený sekvenčním modelem by mohl vypadat například takto:



obrázek 52 Spolupráce objektů ve vzoru MEMENTO

Všimněme si podstatné myšlenky tohoto vzoru. Vzor se skládá ze dvou pozic scénáře:

Klient dostane Memento podobně jako zapečetěný dopis, do něhož nevidí a neotvírá jej. Drží jej až do doby, než se rozhodne vrátit jej Elementu (anebo jako nepotřebný zahodit), aby se Element restauroval. Klient přesně ví, kdy se má Memento použít, ať už zrodit, vrátit zpět anebo zahodit, ale ze zásady neví, co v něm je. Klient zná kontext „kdy“ a nezná kontext „co“.

Element vydává Memento a ví, jak je stvořit a jak se podle něj restaurovat, ale neví kdo a kdy bude o Memento žádat. Element nezná kontext kdy, ale zná kontext co. Prostě pouze nabízí svým (kdoví jakým) Klientům vydání, resp. přijetí Mementa. Kontext použití Mementa ve scénáři tedy drží Klient, způsob tvorby Mementa drží Element.

Poznámka: Tento vzor považuji za velmi přínosný pro pochopení OOP.

Poznámky k použití

Zapouzdření

Je zachován princip zapouzdření, který velmi zprůhledňuje scénář. Z hlediska Klienta je práce s Mementem primitivní.

Pokud to syntaxe jazyka dovolí, doporučuje se umístit interface Mementa pro práci se stavy do pozice viditelnosti `friend` vůči Elementu a `private` vůči Klientovi. Klient tak nevidí operace `SetState()` a `GetState()`.

Klient i Element jsou jednodušší

Přesné rozdělení rolí vede k zjednodušení problému. U Elementu se programuje pouze tvorba Mementa a přesypání stavů. U Klienta se programuje pouze „handlování“ s Mementy.

Klient může mít nečekaný problém

Protože Klient neví, co je Memento zač, může mít jeho program „nečekané technologické“ problémy. Je dobré zdokumentovat Element a Memento tak, aby ten, kdo bude navrhovat práci Klienta, přesně věděl, jaké problémy ho mohou čekat, například s obsazením paměti (velikost Mementa) apod.

Zapouzdření neznamená, že „nikdo nic neví“. Zapouzdření znamená hlavně kompetenci ve scénáři.

Inkrementální Memento

Pokud se pracuje s Mementy vždy sekvenčně „po krocích tam a po krocích zpátky“, může se zavést Memento pouze inkrementální, tj. zaznamenávající pouze změněné stavy. Tento případ by byl možný, pokud se například pracuje s historií Mement. Je třeba správně navrhnout odpovídající (jiné) operace `GetState()` a `SetState()`, které odpovídají inkrementálním změnám stavů.

COMMAND, MEMENTO a history list

Velmi často používanou kombinací je vzor COMMAND a MEMENTO. Každý objekt Command, který přes `Execute()` volá operaci určitého objektu, může podporovat ještě jednu další operaci `Undo()`. Tato operace vrací objekt zpátky do původního stavu. Používá se k tomu (jak jinak) vzor MEMENTO.

Při realizaci `Execute()` si `Command` nejprve vyžádá od objektu nový objekt `Memento` a poté teprve volá operaci daného objektu. Pokud klient objektu `Command` zavolá operaci `MyCommand.Undo()`, potom `Command` pošle dané uschované `Memento` zpátky do objektu operací `SetMemento(Memento)`.

Protože s objekty Command lze „handlovat“ jako s proměnnou, dovedeme si představit seznam (kolekci, TList, arraylist) z vykonaných Commandů, nazvěme jej „history list“. Můžeme po tomto seznamu chodit „nahoru dolů“ a volat `Undo()`, resp. `Execute()` podle dané situace. Pouze mějme na paměti, že v tomto seznamu Commandů se nemohou vyskytovat přímo dané Commandy, které byly vykonané, ale jejich kopie, protože každá instance vykonaného Commandu má své Memento.

Jedná se však pouze o „mělké kopie“, objekt s volanou operací se předá do kopie Commandu pouze referencí. Dvě kopie jednoho stejného Commandu takto mají dvě Mementa, ale ukazují si na původní jeden objekt, nad jehož operacemi pracují a od nějž si žádají a vracejí zpět Memento.

Související vzory

COMMAND používá MEMENTO pro Undo operace.

Průchod MEMENTY resp. COMMANDY v historii stavů lze provádět ITERATOREM.

OBSERVER

Klasifikace

Object

Behavioral

Smysl

Zavádí možnost sledování změn u objektu tak, že když objekt změní stav, ostatní objekty na tuto změnu zareagují, přičemž nedojde k přímé vazbě od sledovaného objektu k těmto objektům.

Alias

Dependents

Motiv

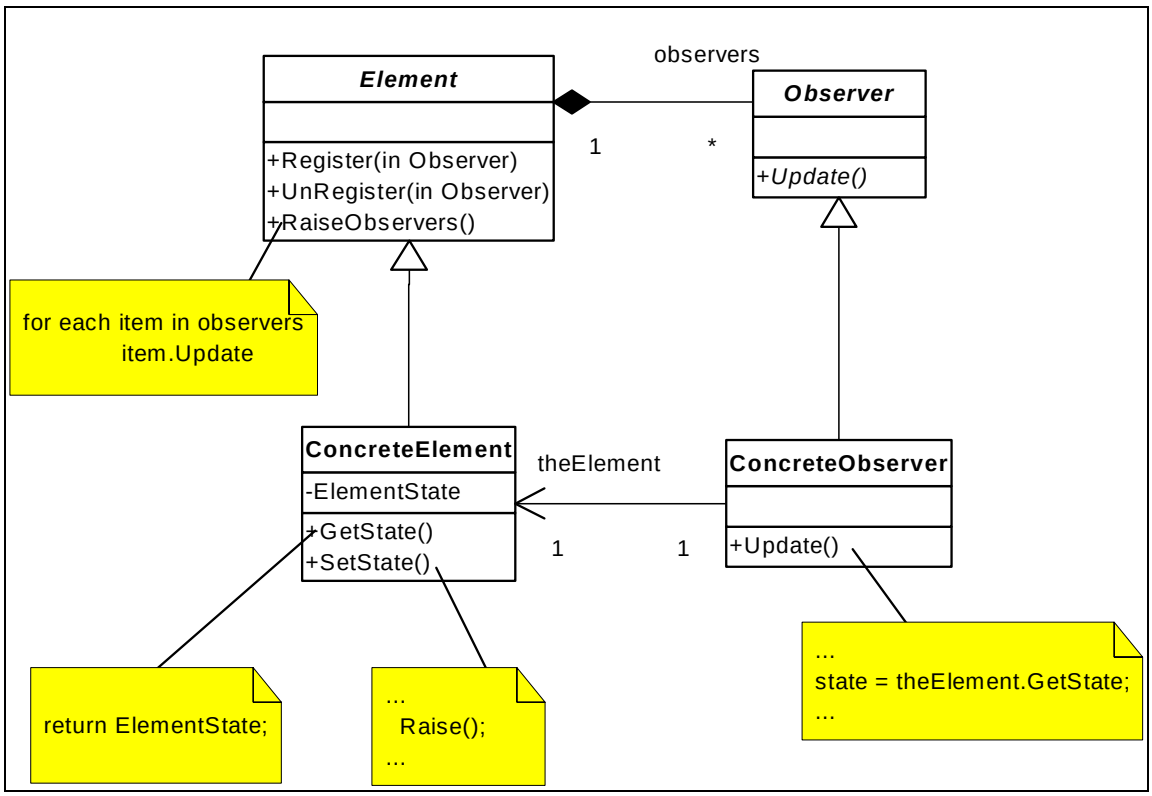
U použití vzoru MEDIATOR existovalo několik variant, jak může nějaký Element, u něhož proběhla změna, vyvolat odpovídající a správnou operaci MEDIATORU (který v této metodě „obtelefonuje“ ostatní Elementy). Pokud by změněný Element zavolal „napřímo“ určitou operaci konkrétního MEDIATORU, došlo by k přímé vazbě konkrétního Elementu na tento konkrétní Mediator a jeho konkrétní operaci. Kód by se již nedal použít pro žádný jiný případ jiného Mediatoru (což by nevadilo, pokud by byl pouze jeden jediný Mediator).

Je možné využít zabudovaný systém událostí daného prostředí. Událostní řízení lze chápat jako obdobu služby *callback* v tom smyslu, že určitý objekt je schopen se zaregistrovat k této službě jiného objektu jako dvojice „objekt plus jeho operace“. V okamžiku, kdy nastane událost u sledovaného objektu, tento vyvolá všechny registrované operace u všech zaregistrovaných objektů, aniž by věděl, koho a co vlastně volá. Systém tak pracuje „dvoukrokově“: V prvním kroku se nějaký objekt se svou operací registruje ke službě události jiného objektu, v druhém kroku nastane událost a všechny registrované operace se zavolají.

Je zřejmé, že problémem v této konstrukci je, jak docílit kompatibilitu mezi všemi registrovanými operacemi tak, aby je bylo možné všechny zavolat jedním kódem. Jinak bez této kompatibility samozřejmě nemůžeme dávat všechny operace „na jednu hromadu“ do jednoho seznamu registrovaných operací a pak je zavolat cyklem.

Tento problém řeší vzor OBSEREVER, který není nic jiného, něž zavedení obdoby systému s událostmi, a to „svépomocí“ a vlastními prostředky. Z toho důvodu bude tento vzor například těm, kteří znají JAVU, velmi povědomý.

Struktura vzoru



obrázek 53 Struktura vzoru OBSERVER

Abstraktní třída `Element` zavádí konkrétně implementované operace pro přidání objektu `Observer` (operace `Register()` je vlastně synonymem pro operaci `Add()` do seznamu), resp. odebrání objektu `Observer` (operace `UnRegister()` je vlastně `Remove()` ze seznamu). Všichni „observeři“ mají jednu polymorfně zavedenou operaci `Update()` a každý konkrétní `Observer` jako dědic tuto operaci přepisuje podle svých potřeb. Díky tomu mohou být všichni konkrétní observeři přidáni do společné kolekce `observers`, protože podporují stejný interface jako dědicové abstraktní třídy `Observer`. Jinak řečeno každý objekt ze třídy `ConcreteObserver` může figurovat jako vstupní parametr obou operací registrace a de-registrace.

V určitém běhu nějaké metody `SetState()` u konkrétního elementu dojde k zavolání operace `RaiseObservers()`. To je bod, kde nastala „událost“. V ten moment se „obtelefonují“ všichni observeři v seznamu `observers`, a to zavoláním operace `Update()`. Tím se polymorfně zavolá konkrétně přepsaná metoda konkrétního observeru.

Objekt `ConcreteObserver` navíc může „vidět svůj objekt“ ze třídy `ConcreteElement` (k němu se registroval) a v rámci svého `Update()` si může vyžádat stavy potřebné ke zpracování události, takže tyto stavy nemusí putovat přes vstupní parametry operace `Update()` (obecně vzato nemusí, ale mohou).

Poznámky k použití

Odstínění a nepřímé volání objektů

Obecně se události, resp. ekvivalentní vzor OBSERVER používá tam, kde není dopředu jasné, kdo má změnu objektu `Element` zpracovat, a dokonce kdy se vyžaduje, aby seznam observerů byl flexibilní a dále kdykoliv rozšiřitelný. Znamená to, že konkrétní `Element` neví, koho obsluhuje, volá observery „nepřímo“ přes interface jejich předka. To je podstatný rozdíl oproti „přímému“ volání, kdy instance přesně „ví“, koho volá. Volání přes Observer tedy odstraňuje volání „natvrdo“ přímým oslovením konkrétní instance.

Vzor (resp. použití události) odstiňuje od elementu konkrétního observera. Všimněme si na obrázku, kdo koho potřebuje a kdo koho si tedy musí přilinkovat (ve smyslu možnosti tvorby svazků – package v UML modelu viz např. [1]):

- Konkrétní Element musí znát abstraktní třídu Element, Element zná abstraktní třídu Observer.
- Konkrétní Observer zná Konkrétní Element a abstraktní Observer.
- Konkrétní Element nepotřebuje znát Konkrétní Observer (ale přitom „vyvolá“ jeho operaci Update!).

Pozor na příliš velkou flexibilitu

Pokud nepotřebujeme flexibilitu „přidávání“ dalších možných observerů uvedenou v předešlém odstavci, uvažme, zda stojí za to události resp. tento vzor zavádět. V případě, že události nebo použití vzoru OBSERVER „přeženeme“ a dáváme jej tam, kde netřeba, systém se stane totálně nepřehledný a těžko udržovatelný. Pokud je například jasné, že na změnu budou vždy reagovat určité jasně dané instance bez dalšího možného rozšíření, použijte raději přímé volání s přímým oslovením objektů.

Příklad: Faktura obsahuje řádky faktury. Zavoláme uložení faktury a tato zavolá uložení všem svým řádkům. Druhá varianta: faktura při uložení vyvolá událost a řádky ji odchytí, resp. ekvivalentně podle vzoru OBSERVER řádky faktury jsou observery pro uložení faktury a spustí uložení mechanismem tohoto vzoru. Druhá varianta je sice více flexibilní, ale systém velmi ztrácí transparenci. Pokud není třeba, nepoužívejme jej.

Mnohdy však přímo z povahy věci vyplyne, že seznam „reagujících“ není uzavřen a může se (přesněji musí se) flexibilně rozšiřovat. V tom případě je vhodné tento vzor resp. události zavést. Jedná se o situace, kdy bychom zpětně museli provádět spoustu změn i v jiných částech kódu, než je samotná jedna změna v přidání kódu.

Zacyklení Update

Problém příliš silného použití vzoru vynikne i tam, kde se začnou události množit tím, že spuštěním jednoho zpracování se vyvolají další a další události. V případě použití tohoto vzoru to znamená, že samotné `Update()` vyvolává další `Raise` a následně `Update()` dalších observerů (a ti mají další observery). Pozor v tomto případě na zacyklení událostí. Upozorníme také na totální nepřehlednost takto navrženého systému.

Pozor na konzistenci stavů konkrétního Elementu při RaiseObservers

V okamžiku, kdy konkrétní element volá operaci `RaiseObservers()`, měl by se tento objekt nacházet v konzistentním stavu. Jinak řečeno, neměl by být ve stavu „rozdělaném“ (půlka atributů tak a půlka jinak). V okamžiku vyvolání observerů jim totiž předává řízení a může potom nastat situace, že observer bude číst jeho nekonzistentní stavy a dojde ke kolizi.

Multithreading

OBSERVER (a obecně události) jak vidět neřeší problém paralelního zpracování a vyvolání nového threadu. Volání observerů v seznamu observerů je sekvenční. Mylný názor ohledně vyvolání nového threadu se objevuje hlavně u Basicářů. Důvodem je určitá možnost ve VB 6.0 vyvolat nový thread speciální konstrukcí využívající události ve VB 6.0.

Pokud bychom chtěli, aby observeři pracovali v jiných threadech, museli bychom tento vzor doplnit o další funkcionalitu (například ji umístit před nebo za interfacem observeru), která je schopna vyvolat nový thread, anebo se použijeme technologie Message Queue Serveru.

Vymazání observera

Představme si, že dojde k situaci, kdy se nějaký konkrétní observer zaregistruje k nějakému „vyburcování“ k Elementu. V běhu aplikace dojde k vymazání observeru, který již nemá být použit. Je třeba zvážit, co s jeho registrací. Pokud se jedná o systémy vyvíjené v prostředí bez podpory garbage collection (např. Delphi), může dojít i k pádu systému voláním uvolněného objektu. V tomto případě by bylo vhodné vyloučit ze seznamu prvky s hodnotou „nil“.

Pokud systém pracuje v režimu s garbage collection, k pádu systému sice nedojde, ale nastanou jiné nekorektnosti: Element „tvrdohlavě“ drží referenci na objekt, který by již existovat neměl, a proto není uvolněn. Je třeba tuto situaci řešit („odregistrovat“, nebo zavést příznak neplatnosti a poté odregistrovat apod.).

Asociativní třída pro registraci

Dalším možným řešením, jak lze registrovat observery k elementům, je zavést nikoliv přímou vazbu „element drží odkazy své observery“, ale zavést asociativní třídu, tj. vybudovat vazbu přes objekty jako asociativní dvojice (Element + Observer). Tato dvojice odpovídá asociativní třídě (asociativní třídy viz např. [1]).

Zavede se nějaká kolekce (seznam) jako správce těchto objektů dvojic, nazvěme jej `RegisterManager`. Registrace tedy neprobíhá žádostí od Observeru k Elementu voláním jeho metody `Register()`, ale voláním tohoto Manageru a vstupní parametry jsou „dvě informace“ Observer plus Element. Tímto vznikne centralizovaný seznam všech dvojic.

Správná funkcionalita asociativní třídy (viz [1]) vyžaduje, aby `RegisterManager` mohl pracovat s filtrací odpovídajících asociovaných prvků. Například v tomto případě se může volat „obtelefonování“ od Observera nějak takto (vstupní parametr je typu Element):

```
MyManager.RaiseMyAllObservers(this);
```

resp. přes vstupní parametr jako klíč, který Element zná a získal jej jako návratovou hodnotu při svém přidání mezi všechny Elementy Manageru:

```
MyManager.RaiseMyAllObservers(mMyElementKey);
```

Manager musí (nějak) filtrovat a dostat všechny dvojice, které mají daný Element a poté všem prvkům Observerům na druhé straně dvojice zavolat jejich polymorfní `Update()`.

U daného Managera je vhodné přímo jeho operace zavést jako polymorfní. Manager jako správce může totiž vyžadovat další flexibilní změny funkcionality.

Lze si představit další možné rozšíření této funkcionality, kde bude existovat ještě další dodatečná informace (například vztah do číselníku) udávající událost vedoucí k vybuzení, což může být další vstupní parametr:

```
MyManager.RaiseMyAllObservers(MyElementKey , why);
```

Tedy může existovat několik „různých“ kolekcí Observerů sledujících pokaždé „něco jiného“ v Elementu. Toto rozšíření je možné i v původním řešení, kdy si Element drží odkazy na své observery.

Související vzory

Vzor COMMAND vykonává podobnou funkcionalitu, ale jedná se o obecné spouštění nějaké funkcionality, nikoliv sledování nějaké změny v objektu.

Pro zavedení Managera registrace v předešlém odstavci je vhodné použít vzor SINGLETON.

STATE

Klasifikace

Object

Behavioral

Alias

Object for States

Smysl

Provádí změnu stavu objektu výměnou vnitřního objektu reprezentujícího stav.

Motiv

Jednou ze základních myšlenek OOP je zavedení „vnitřních“ stavů objektu. Při volání operací se objekt „rozhoduje“ na základě hodnot těchto stavů. V některých případech však jednoduchá implementace tohoto principu pomocí přepínání (`if` nebo `switch`) může vést k velmi nepřehledným konstrukcím, zejména u složitých technologických objektů (různé konekty, komunikace, řízení procesů apod.).

Představme si, že existuje v nějakém takovém objektu atribut, který reprezentuje nějaký stav tohoto objektu. Označme tento atribut jako `atrState` a nechť může nabývat tří hodnot 1, 2, 3. Na základě tohoto stavu se objekt při volání operace `OperationA()` v konkrétní metodě rozhoduje, například takto:

```
Public OperationA() ...
...
switch atrState;
{
  1 : A1();
  2 : A2();
  3 : A3();
}
```

Podobně zjistíme, že i pro `OperationB()` nastává obdobná situace:

```
Public OperationB()...
...
switch atrState;
{
  1 : B1();
  2 : B2();
  3 : B3();
```

```
}
```

Operace `A1()`, `A2()`, `A3()`, `B1()`, `B2()` a `B3()` jsou symbolicky zapsány nějaké vnitřní (například privátní) operace objektu, resp. se jedná o bloky kódu zpracování dané větve. Při tomto návrhu se nám kód větví a pokaždé znovu a znovu. Kód se stává nepřehledný s možností zanesení chyb do stavového mechanismu.

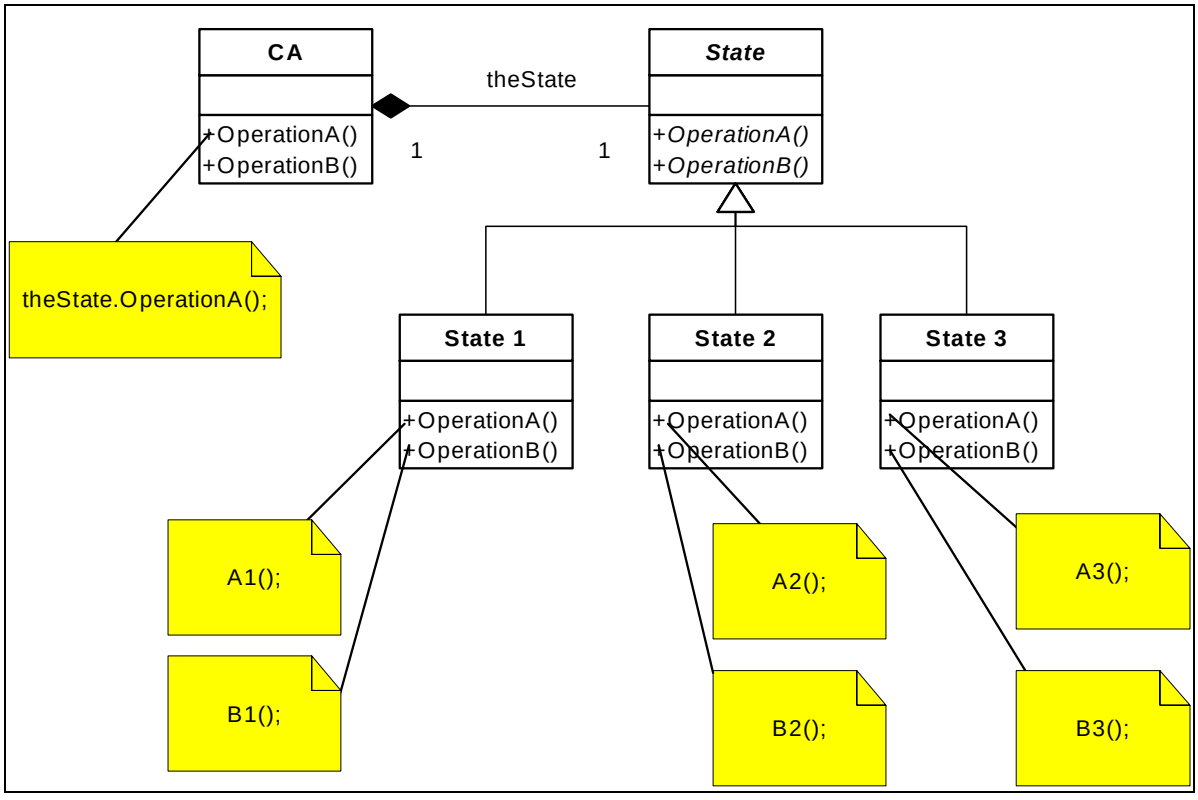
Navrhněme jiné řešení:

Nebude se provádět větvení tak, jak je ukázáno v předešlém odstavci. Rozdělení kódu se provede v „druhém směru“, tj. dají se nějak k sobě operace pro stejný stav: `A1` a `B1`, `A2` a `B2` atd. Toto rozdělení se provede pomocí nové třídy.

Zavede se jedna nová abstraktní třída, nazvěme ji `State`. Tato třída zavede polymorfní operace pro operace `OperationA` i `OperationB`. Znamená to, že obě operace se nyní vyskytují dvojmo, jednou v původním objektu a jednou v interfacu `State` (ale jsou to jiné dvojice operací).

To je první krok.

Druhý krok spočívá v tom, že se vytvoří tolik dědiců třídy `State`, kolik je stavů. V našem případě jsou to tři stavy a tedy tři třídy. V každém dědici třídy `State` se odpovídající operace přepíše tak, že se do ní vloží pouze kód dané větve. V našem případě tedy vzniknou tři nové třídy (plus jedna abstraktní horní). Do objektu A se vloží jedna instance podporující interface `State`. K přepnutí stavu dojde výměnou celého objektu State za jiný, který reprezentuje jiný stav. Znázorněme si tento model se stromem dědičnosti pomocí class modelu:



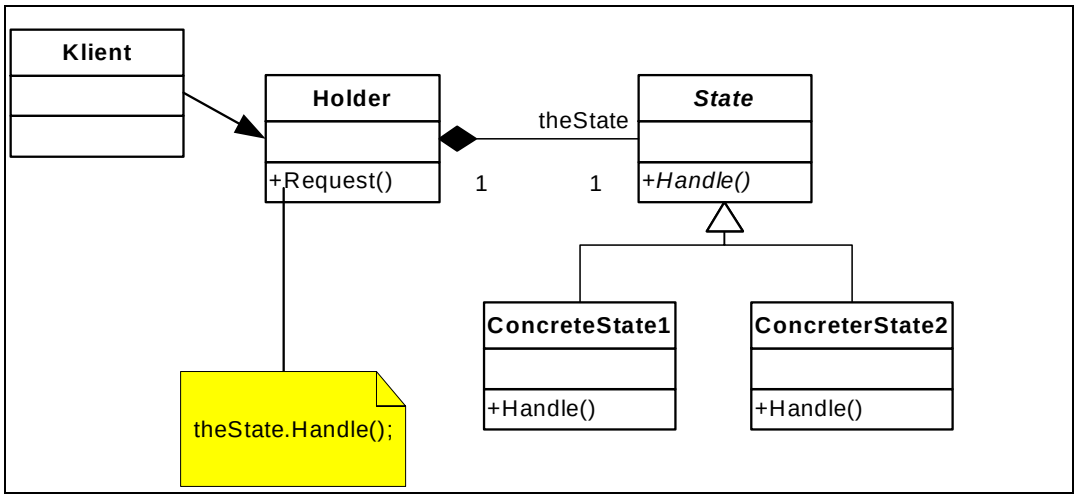
obrázek 54 Řešení složitého přepínače v předešlém kódu pomocí STATE

Pokud se v objektu A (ze třídy `CA`) nachází instance ze třídy `State1`, potom zavoláním `OperationA()` objektu `theState` zavolá implementovaná metoda, v níž je kód `A1()`. Pokud se bude nacházet objekt ve stavu 2, za interface `theState` bude objekt ze třídy `State2` a na stejnou operaci se zavolá kód `A2()` atd. Totéž platí pro operaci B.

Znamená to, že změny stavu objektu A ze třídy `CA` již nejsou reprezentovány (a uskutečněny) změnami hodnot atributů, ale dějí se výměnou instance z různých tříd stavů. Přepnutím stavu, a tedy výměnou objektu za interfacem v instanci `theState`, se změní chování objektu „najednou“ pro všechny operace a není již třeba přímo v operacích provádět `switch` podle stavu.

Kód je přehlednější (elegantnější) a u složitých mechanismů není tolik nenáchylný k omylům jako u původního řešení.

Struktura vzoru



obrázek 55 Struktura vzoru STATE

Klient volá interface objektu `Holder`, který „drží stavy“. `Holder` přesměruje požadavek na instanci `theState`. Přepínání stavů znamená kompatibilní výměnu instance `theState` za jinou instanci jiného dědice třídy `State`. Podle toho, v jakém se `Holder` nachází stavu, se za tímto interface nachází instance z dané třídy a tedy konkrétní `Handle` daného konkrétního stavu.

Poznámky k použití

Přehlednost a flexibilita

Oproti původnímu kódu je druhý způsob nejen transparentnější, ale i flexibilnější, protože při změnách se nemusí zasahovat do všech jinak dlouhých kódu všech přepínačů u všech operací. Stačí přidat nového dědice.

Vyšší konzistence

Použití vzoru omezuje výrazně zavlečení chyb daných nekonzistencí stavů. Přepnutí stavů výměnou celého jednoho objektu zamezí chybám typu „objekt je (omylem) napůl v jednom stavu a napůl v druhém stavu“. Objekt má buď instanci z jedné třídy anebo druhé a k přepnutí dojde naráz.

Řízení přechodů

Je vhodné, aby byl klient sám ušetřen problémů, jaký stav po jakém stavu má následovat a která instance stavu se má nyní dosadit za roli `theState`. Je lepší, aby sami dědicové ve svých metodách `Handle()` určovali, kdo je „následník“. Znamená to ovšem, že objekt `theState` musí mít zpřístupněn interface objektu `Holder`, což vede ke zpětné referenci (anebo aspoň na jeho části), aby konkrétní objekt stavu mohl tuto operaci výměny u objektu `Holder` zavolat.

Pokud se budou držet stavové objekty v seznamu, lze následníka vybrat přes unikátní klíč, tj. jeden určitý dědic nemusí mít přímou vazbu na jiného dědice (jak je uvedeno jako nedostatek tohoto postupu v [5]). Volání změny může být provedeno z daného `Handle()` například takto:

```
mMyHolder.SetNextState( 3 );
```

Stavové veličiny, atributy

Je možné, že všichni dědicové stavů mají nějaké nutné společné atributy. V tom případě třída `State` není čistou abstraktní třídou (není pouze interface). Nedoporučuji umisťovat tyto atributy do této pozice horní třídy, což by se na první pohled mohlo jevit jako dobré řešení. Raději „vytkněme tyto údaje ven“ a provážeme je zpět asociací, stejně jako v příkladu Osoba u BRIDGE. Pokud toto neuděláme, potom budeme muset vždy hodnoty stavů přesypávat z jednoho objektu do druhého při každé výměně objektu. Jako dobré řešení se jeví umístit tyto atributy do objektu `Holder`, protože tak jak tak musí mít `State` na něj referenci, aby `State` mohl určit objektu `Holder` svého následníka.

STRATEGY

Klasifikace

Object

Behavioral

Smysl

Definuje množinu algoritmů, které jsou kompatibilně vyměnitelné.

Motiv

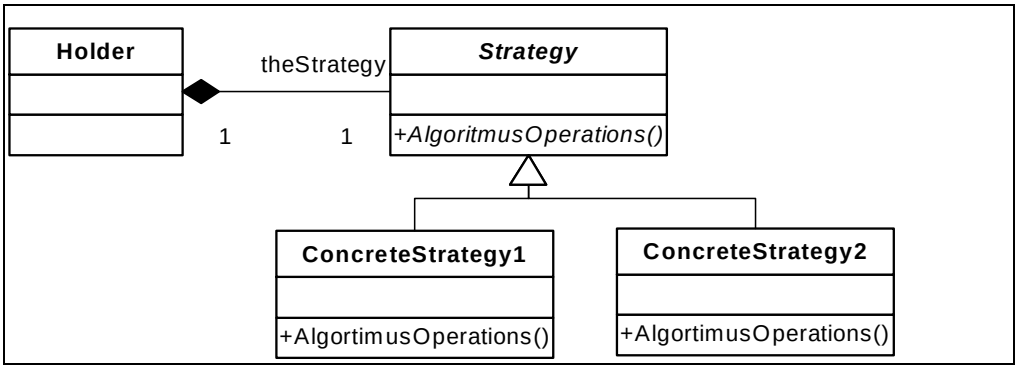
V ekonomickém nebo finančním systému se může pro danou částku počítat úrok. Existuje několik možných algoritmů počítání úroku. Od zákazníka, odběratele systému, je vyžadováno, aby algoritmus úročení byl pro daný případ (například účet) vybrán obsluhou, a to dokonce v runtime. Například při zakládání účtu, typu účtu apod., aby se vybral algoritmus úročení daného účtu, resp. načetl z konfiguračních dat apod.

Takže je třeba, aby se částka uměla úročit různými algoritmem podle toho, jaký algoritmus bude vybrán. Nepříliš šťastné řešení je opět přepínačem: Daný držitel vybraného algoritmu si bude držet datovou proměnnou udávající, jaký algoritmus byl vybrán (např. 1 = algoritmus první, 2 = algoritmus druhý). Při zavolání zúročení se podle této hodnoty bude rozhodovat (opět složitý a neflexibilní `switch`). Dostáváme se do velmi podobného problému, jako u předešlého vzoru STATE.

Lepší řešení je (jak jinak) postaveno na polymorfismu.

Představme si, že bude existovat třída pro výpočet úroku a bude mít operaci `VypoctiUrok()`. Bude existovat jedna abstraktní třída zavádějící interface s touto operací a dědicové ji budou přepisovat podle zvoleného algoritmu, tj. podle zvoleného objektu z dané podtřídy . Výměna objektu (se stejným interfacem) do téže posice za stejný interface znamená vlastně změnu algoritmu úročení. Tato změna je kompatibilní, flexibilní a navíc přehledná.

Struktura vzoru



obrázek 56 Struktura vzoru STRATEGY

`Holder` zavádí interface pro volání požadavků od klienta a z druhé strany drží instanci `theStrategy`. Výběrem objektu z podtřídy konkrétního `Strategy` dojde k výměně algoritmu (což může být jedna nebo více operací).

Poznámky k použití

Dědicové Holderu?

Podobného řešení bychom mohli dosáhnout také děděním už na úrovni třídy `Holder` (`Holder` by měl různé dědice a ti by měli různé algoritmy). Je to však řešení mnohem „tvrdší“ ke změnám, statické, musí se totiž vyměňovat celý `Holder`, což je pro klienta nepříjemné.

Umístění změnitelného algoritmu zvlášť do `Strategy` umožňuje změnit algoritmus nezávisle na třídě `Holder`. Klient výměnu objektu `Strategy` vůbec nepocítí a nestará se o ni.

Zpětná vazba

V některých případech může `Strategy` potřebovat ke zpracování nějaké informace. Opět lze zavést zpětnou vazbu na objekt `Holder`, avšak v tom případě se jedná o algoritmus samozřejmě závislý na Holderovi. Jiná možnost je zavést vstupní parametry, avšak může být problém v tom, že ne každý algoritmus potřebuje všechny parametry.

Související vzory

Pokud je STRATEGY jako algoritmus sdílené, lze je použít jako SINGLETON (například pouze algoritmy výpočtu apod.), anebo vedou na FLYWEIGHTS.

Vzor STRATEGY je strukturou velmi podobný vzoru STATE, což je pochopitelné: Nastavení algoritmu lze také chápat jako změnu stavu.

TEMPLATE METHOD

Klasifikace

Class

Behavioral

Smysl

Zavádí scénář na abstraktnější horní úrovni předka složený z několika polymorfních metod. Dědicové používají scénář tak, že polymorfní operace přepíší (vyplní) a scénář zavolají.

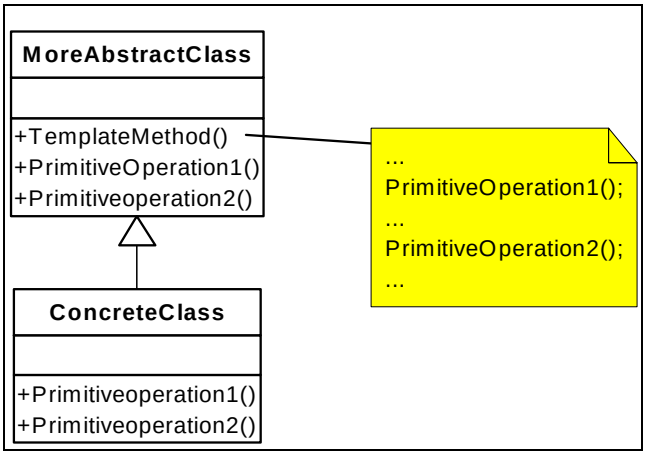
Motiv

S tímto vzorem jsme se nepřímo setkali u FACTORY METHOD, kde byl zaveden Dokument, který se měl otevřít a přidat mezi ostatní dokumenty. V tomto vzoru jedna operace tvoří nějaký scénář a přitom se skládá z několika operací, které jsou polymorfní. Dědicové tyto operace konkrétně vyplní a scénář použijí. Například si lze představit v uvedeném scénáři nového dokumentu nějakou operaci `CanOpenDocument()` (vrací `true` nebo `false`) a tato operace bude polymorfní. Dědicové musí specifikovat konkrétně, jak se bude návratová hodnota zjišťovat. Potom na vyšší úrovni předka `Application` můžeme psát v pseudokódu:

```
public ... OpenDoc();
 {
  If ( not CanOpenDocument() ) then return actualDoc;
  Mydoc = Createdoc();
  Docs.Add (Mydoc);
  MyDoc.Open();
  actualDoc = Mydoc;
}
```

Přitom operace `CanOpenDocument()` a `Createdoc()` jsou polymorfní a budou se dědicem přepisovat (pozn.: `Createdoc()` je dokonce FACTORY METHOD). Poté, co jsou specifikovány „náplně polymorfních metod“, může dědic zavolat operaci `OpenDoc()` a využije tak již jednou naprogramovaný scénář otevírání dokumentu. Operace `OpenDoc()` je příkladem TEMPLATE METHOD.

Struktura vzoru



obrázek 57 Struktura vzoru TEMPLATE METHOD

Předek zavádí operaci `TemplateMethod()`, která obsahuje volání nějakých polymorfních primitivních operací. Sama operace `TemplateMethod()` není polymorfní a tvoří kostru (scénář) řešení. Dědicové jako `ConcreteClass` přepisují primitivní operace a poté volají operaci `TemplateMethod()`.

Třída předka `MoreAbstract` nemusí být povinně třídou abstraktní a primitivní operace také nemusejí být povinně abstraktní. Může existovat implicitní (default) implementace, kterou zavádí přímo předek, resp. některé operace předka mohou mít své implicitní chování a není povinnost je přepsat. Například `CanOpenDocument()` může mít implicitní kód pouze `{ return true }`, který nebudou přepisovat ti dědicové, kteří vůbec nemají takovéto rozhodování zapotřebí.

Poznámky k použití

Výhoda přesně definovaných postupů

Při extenzi systému o další podtřídy lze definovat jednotný postup: Vyplň polymorfní operace v TEMPLATE METHOD a zavolej ji. To se jeví jako výhoda nejenom z hlediska re-use, ale i z hlediska řízení tvorby softwaru.

Všeobsahující scénáře

Uvedený vzor předpokládá, že kostra je dána a stačí ji vyplnit. Problém může být v tom, že v dosud známých scénářích nemusí být vždy všechny operace, které by se mohly vyskytnou v dalších scénářích. Tuto situaci vystihuje věta: „Kdo mohl (safra) předpokládat, že se bude třeba ptát na `CanOpenDocument()`, když jsme to zatím nepotřebovali?“

V [5] se doslova píše o určité „předvídavosti“ a „kreativitě“, tj. zavádět takovéto operace dopředu. Mé osobní zkušenosti z projektů mne spíše před takovouto přílišnou kreativitou varují. Aplikace je potom „prošpikována“ dopředu připravenými prázdnými operacemi (`BeforeOpenDoc()`, `AfterOpenDoc()` atd.). Hlavní výhoda postupu je abstraktní, opětovně použitelný scénář s přehledností a ta se může ztratit.

Dá se říci, že pravda je „někde uprostřed“: Předvídat, ale střídmě a prakticky.

Související vzory

Vzor FACTORY METHOD je používán ve vzoru TEMPLATE METHOD.

VISITOR

Klasifikace

Object

Behavioral

Smysl

Představuje operaci, která může účinkovat na prvky objektové struktury, aniž by se měnil kód ve třídách těchto prvků. Zavádí flexibilní alternativu k přidávání kódu polymorfních operací do tříd.

Motiv

Tento vzor je trochu zvláštní v tom, že jeho užití není až tak časté (zejména kvůli neznalosti), ale má velmi zajímavou a oproti jiným vzorům trochu složitější konstrukci. Důvodem je zavedený dvojitý mechanismus polymorfismu (polymorfismus se zde vyskytuje aplikovaný dvakrát po sobě).

Vraťme se k našemu příkladu na obrazce s operací `DejObsah()`. Zařadili jsme tuto operaci jako polymorfní operaci do každého prvku. Představme si, že strom dědičnosti našich obrazců je již hotov (stabilizován) a je jich hodně. Pokud obrazce operaci `DejObsah()` neumějí a chtěli bychom tuto operaci přidat, potom bychom museli každou třídu otevřít a přidat „její“ kód. To je poměrně dost nepříjemná záležitost. Pro každou novou operaci budeme muset znovu a znovu kód otevírat.

Existuje i druhá zajímavá varianta k postupu, jak přidat do tohoto stromu dědičnosti novou polymorfní operaci (raději budeme říkat pseudooperaci) pomocí vzoru VISITOR. Přitom se do původního kódu nezasáhne.

Na první pohled to zní jako zázrak. Jak chcete přidat novou operaci, aniž byste měnili kód tříd a neporušili přitom zapouzdření? Tady je třeba podotknout jeden důležitý fakt: Vzniklá nová pseudooperace nebude používat vnitřní atributy (privátní členy) objektů, ale pouze již existující operace prvku, z nichž se poskládá.

V našem příkladu na `DejObsah()` to znamená, že když ji obrazce neznají a chceme ji zavést pomocí vzoru VISITOR, musí existovat zavedené operace vracející Poloměr, Základnu, Výšku, Stranu (například jako property). Z návratových hodnot těchto operací lze získat všechny údaje pro výpočty obsahů. Bez tohoto předpokladu bychom novou pseudooperaci pomocí VISITORU nezavedli. Proto také budeme této visitorské operaci raději říkat pseudooperace. Klasická operace totiž může pracovat s vnitřními privátními členy objektů.

Myšlenka našeho postupu u VISITORA je následující:

První krok:

Každý prvek stromu obrazců (vzpomeňme, že nahoře je abstraktní obrazec) bude mít jednu polymorfní operaci pro přijetí Visitora. Nazvěme tuto operaci pro přijetí Visitora jako

`Accept(CVisitor aVisitor)`.

Tuto operaci zavedeme u všech prvků a to polymorfně, tj. budou se u potomků obrazců přepisovat.

To je „první polymorfismus“. Každý obrazec tedy umí přijmout Visitora pomocí operace `Accept(CVisitor aVisitor)`, každý trochu jinak. Jak se tato operace přepisuje, uvedeme dále.

Poznámka: Vzhledem k jednomu slavnému sci-fi filmu bychom mohli přijmout alias tohoto vzoru jako „Alien“.

Druhý krok:

Bude existovat několik tříd Visitorů, z nichž každý reprezentuje nějakou pseudooperaci. Tuto větu si zapamatujme:

Každá třída Visitora reprezentuje pseudooperaci.

Pro náš případ existuje nyní jeden Visitor pro jednu operaci obsahu, nazvěme tuto třídu jako `VisitorObsah`.

Důležité je, že také Visitoři jsou jednou rodinou, tj. i oni mají svůj strom dědičnosti. Nahoře je jeden abstraktní `Visitor` a jeho dědicové jsou `ConcreteVisitor`, každý dědic reprezentuje jednu pseudooperaci.

Podle této konstrukce máme nyní dvě třídy ve stromu `Visitor`: jednoho „nejvyššího“ abstraktního předka `Visitor` a jeho dědice `VisitorObsah`.

U operace `Accept()` zavedené u Elementu je typem vstupního parametru nejvyšší abstraktní `Visitor`, takže libovolný prvek stromu obrazců může spolknout libovolného dědice `ConcreteVisitor`, tedy i našeho `VisitorObsah` (a možná až zavedeme, tak dalšího).

Zapamatujme si druhou větu:

Každý obrazec umí „spolknout“ libovolného Visitora.

Dvojitý polymorfismus se projeví nyní:

Každý Visitor má zavedeny operace pro návštěvu jednotlivých obrazců. Každý Visitor má tedy tolik operací pro návštěvy, kolik je tříd v obrazcích, tedy pro každého zvlášť. Přitom tyto operace jsou také polymorfní a každý Visitor si tyto operace přepíše podle svého. Pokud se to začíná jevit jako složité, nezoufejte, ono to složité je, protože je tady dvojitý polymorfismus a pro normální mysl je vždy to, co je dvojité, složité (například dvojitá negace, dvojitá abstrakce, dvojitý polymorfismus, dvojitý panák, který by se teď hodil, apod.).

Nazvěme tyto operace symbolicky jako `Visit<KonkrétníObrazec>`, kde za „šablonu“ `<Konkrétní obrazec>` je třeba postupně dosadit všechny primitivní obrazce. Přitom vstupním parametrem této operace bude onen člen, který se navštěvuje, tedy šablona operací Visitora vypadá nějak takto:

```
Visit<KonkrétníObrazec> ( <KonkrétníObrazec> the<KonkrétníObrazec>)
```

U každého Visitora našeho příkladu bychom museli podle této šablony definovat tyto operace:

```
VisitKruh (CKruh Kruh)
VisitObdelnik (CObdelnik Obdelnik)
VisitTrojuhelnik (CTrojuhelnik Trojuhelnik)
```

Jiný `Visitor` (vzpomeňme větu, kterou jsme si měli zapamatovat) reprezentuje jinou pseudooperaci, což se konkrétně projeví tak, že každý `Visitor` si přepíše tyto operace tak, aby se tato pseduoperace realizovala podle algoritmu této pseudooperace. Operace `VisitKruh` u našeho `VisitorObsah` vezme vstupní parametr Kruh a vypočítá z něj obsah, u Visitora pro obvod z něj vypočítá obvod.

Protože vstupním parametrem je přímo daný obrazec, tak `Visitor` si může od tohoto obrazce vyžádat vše potřebné pro vykonání pseudooperace (zde je právě platná úvaha o property resp. jiné operace, které musí být z obrazce vyvedeny ven). U kruhu stačí vyvést poloměr.

Máme nyní zapamatované dvě věty (reprezentují dva polymorfismy),

Každý obrazec umí spolknout libovolného Visitora.

Každý Visitor umí navštívit Obrazec a provést tak pseudooperaci.

které se následně ve vzoru spojí dohromady takto:

Spolknutí Visitora daným konkrétním obrazcem se převede na zavolání visitorské návštěvy tohoto konkrétního obrazce.

To se v kódu projeví přímo takto:

Kruh přepíše `Accept (Visitor aVisitor)` takto:

```
{
  aVisitor.VisitKruh(this);
}
```

Obdélník přepíše `Accept(Visitor aVisitor)`:

```
{
  aVisitor.VisitObdelnik(this);
}
```

atd.

Výjimku tvoří složený obrazec, kde lze přímo delegovat `Accept()` cyklem na jeho děti:

```
Public void Accept(Visitor aVisitor);
 {
   for each item in obrazce
    item.accept(aVisitor);
 }
```

Scénář práce s Visitorem je tedy následující:

1. Na začátku se vybere pseudooperace, tedy z jaké třídy bude Visitor. V našem případě toho moc na výběr nemáme, bude to objekt ze třídy `VisitorObsah`.
2. Zavolá se operace nějakého obrazce `Accept(Visitor)` a vstupním parametrem bude náš objekt ze třídy `VisitorObsah`. Tuto operaci spolknutí Visitora umí každý obrazec.
3. Teď záleží na tom, jaký obrazec jsme si to vybrali. Nechť je to například kruh, potom `Accept` tohoto obrazce Kruh obsahuje kód: `aVisitor.VisitKruh(this);` tj. vrací se řízení Visitoru se slovy: „Dělej si se mnou co chceš (ale jako s kruhem!) a tady mne máš jako vstupní parametr, dám ti vše, co třeba...“
4. Pokračujeme ve scénáři dále: Teď jsme ve Visitoru a uvnitř jeho metody `VisitKruh` a máme k dispozici `Kruh` jako vstupní parametr, a nyní záleží na tom, jaký `Visitor` jsme úplně na začátku vybrali, protože podle toho se bude něco s kruhem dít. My jsme vybrali `VisitorObsah`, takže u kruhu se vyžádá poloměr (jako property) a spočte se obsah kruhu. Tento údaj, tj. obsah, si jako výsledek `Visitor` si převezme „do sebe“ a může jej na požádání vydat klientovi spouštějící `Accept()`. `Visitor` si takto hodnoty může držet a může navštívit celou strukturu a posbírat údaje od všech obrazců (viz složené obrazce) a nasčítat je.

5. Uvedený scénář je flexibilní v tom, že můžeme přidat nového sub-Visitora, například `VisitorObvod` a naprogramovat všechny jeho `VisitObrazce` a pustit jej na libovolný obrazec stejnou operací `Accept()`. Samozřejmě `Visitor` při návštěvě musí mít u obrazce dostatečný interface pro daný algoritmus. Například je zřejmé, že pro obvod trojúhelníka nestačí property základna a výška.

Pro představu, jak je implementována metoda `VisitKruh` u „Visitora sbírajícího obsah“ v pseudokódu (pozn.: bez optimalizace, asi bychom property R nevolali dvakrát, zde pro přehlednost ponecháno):

```
class VisitorObsah : Visitor;
{
...
Public void override VisitKruh(CKruh aKruh);
{
mObsah = mObsah + aKruh.R * aKruh.R * constPI
}
```

kde `mObsah` je vnitřní proměnná objektu `VisitorObsah`

u jiného dědice, například `VisitorObvod`, bude tatáž operace implementována jinak:

```
class VisitorObvod : Visitor;
{
...
Public void override VisitKruh(CKruh aKruh);
{
 mObvod = mObvod + 2 * aKruh.R * constPI
}
```

Například použití Visitora pro obsah může vypadat takto:

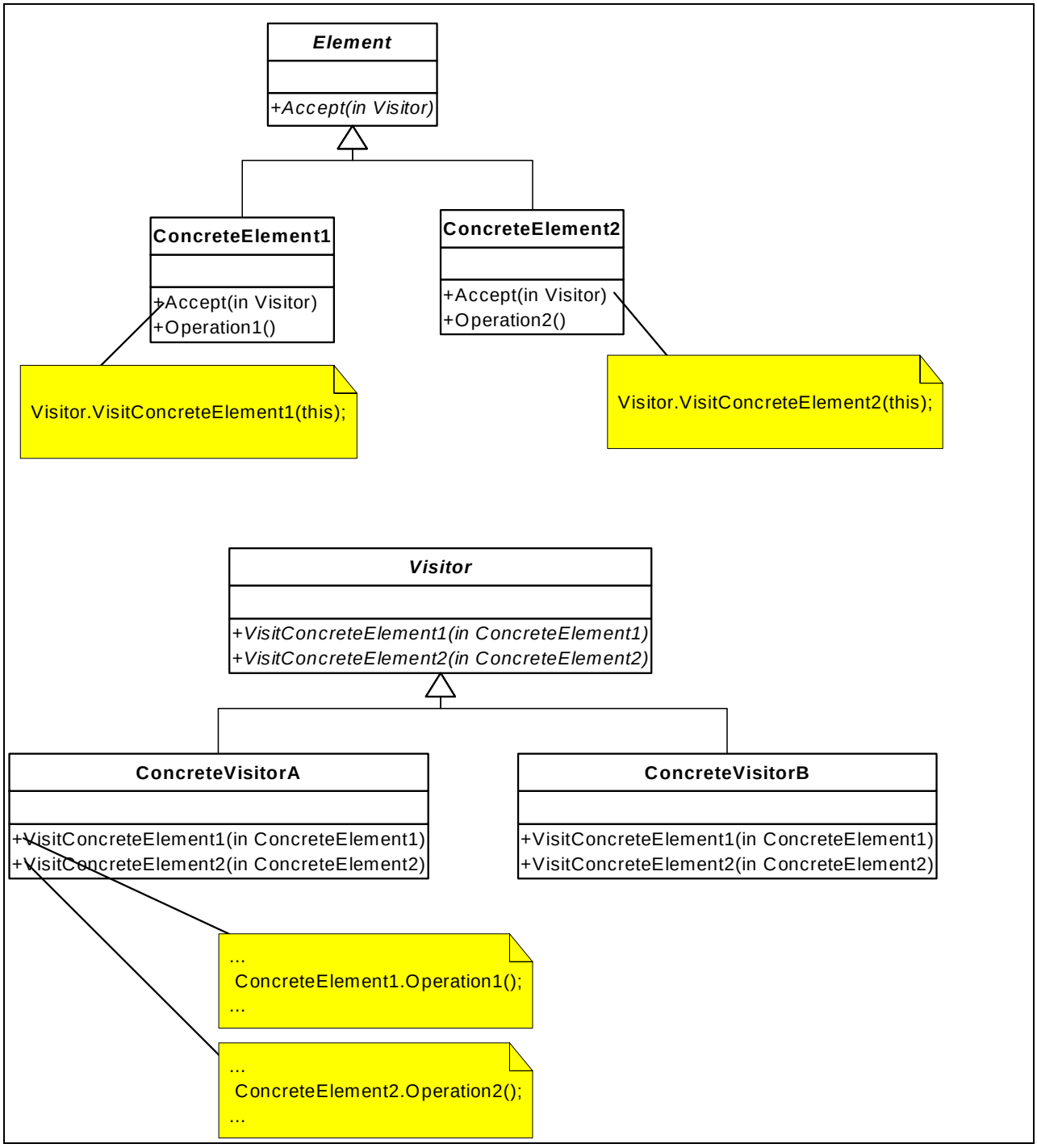
```
VisitorObsah  MyVisitor = new VisitorObsah() //... mObsah = 0 ...
MyObrazec.Accept(MyVisitor);
ResultObsah = MyVisitor.DejObsah();
```

(a podobně pro obvod...)

Přitom například `MyObrazec` může být rekurzivně složený obrazec a `MyVisitor` „projde“ celým stromem (viz u složeného obrazce volání cyklem `Acccept()`). Pokud se jedná pouze o jeden

primitivní obrazec, `Visitor` jej „navštíví“, vypočte tento jeho obsah (vlastně výpočet připočte k nule a končí) a vydá jej na požádání ven.

Struktura vzoru



obrázek 58 Struktura vzoru VISITOR

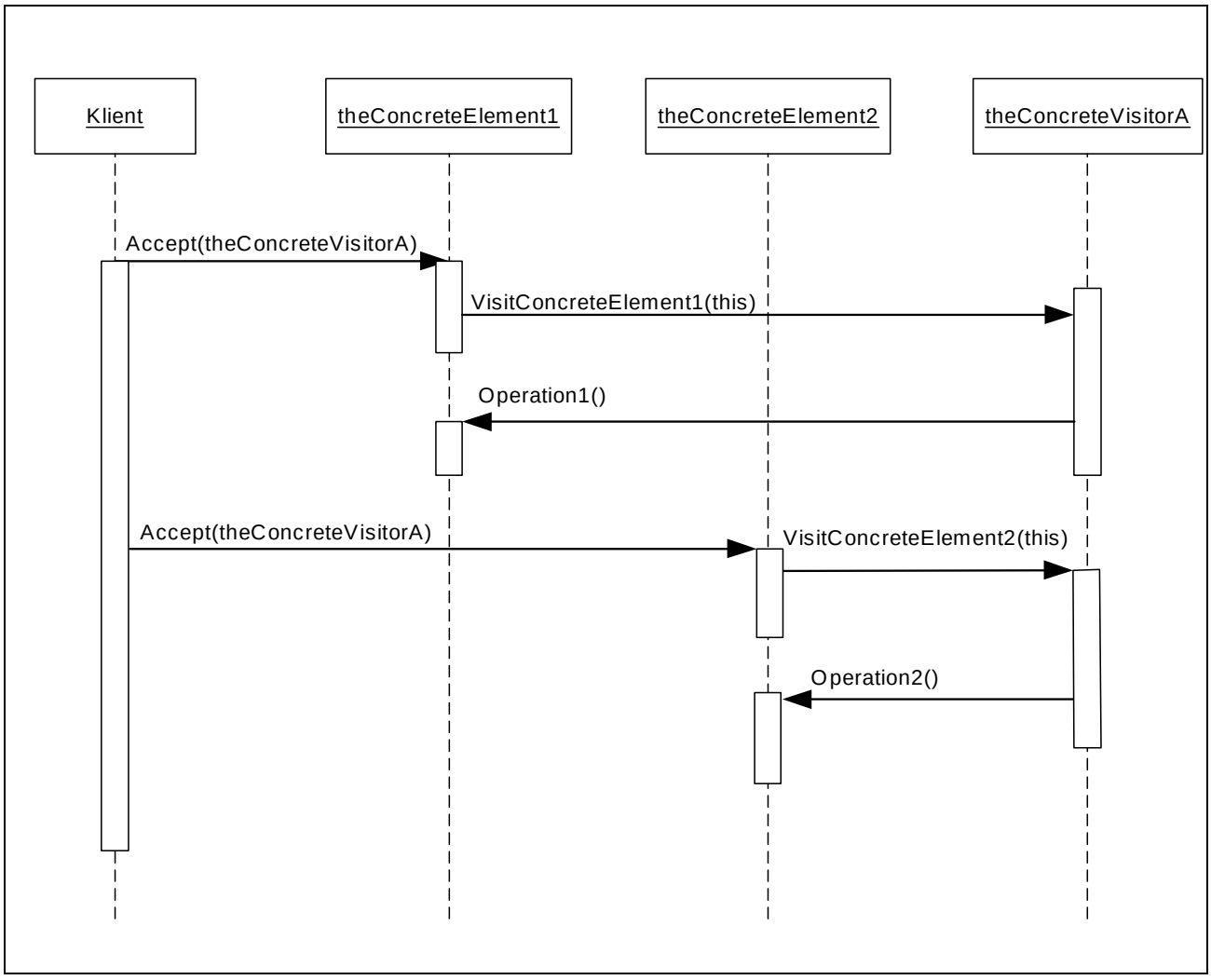
Každý `Element` umí přijmout objekt s interfacem `Visitor` operací `Accept(Visitor)`. Tato operace má pro každý element jiný obsah podle vzorce:

`Visit<`*`Konkrétní Element`*`>(<`*`Konkrétní Element`*`>)`

(Výjimku může tvořit kompozitní `Accept()` s delegací na děti složeného objektu.)

Tím dojde k vyvolání odpovídající návštěvy správného elementu. Pro každou pseudooperaci lze zavést nového Visitora, který přepisuje tyto operace návštěv. Podle zvoleného Visitora se aplikuje vybraná pseudooperace.

V sekvenčním modelu můžeme dynamiku předešlého modelu tříd znázornit takto:



obrázek 59 Spolupráce objektů Elementů a Visitora ve vzoru VISITOR

Poznámky k použití

Flexibilita operací, ne-flexibilita elementů

Je zřejmé, že (pokud nám to seznam operací prvků dovolí) můžeme zavést novou operaci nad libovolnou strukturou, a to pouze dodáním nové třídy. Zákazník tak může dostat velmi rychle nějaké přehledy nad strukturami resp. jiné algoritmy zpracování složených objektů.

Problém nastává, pokud přidáme nový konkrétní Element. V tom případě se nám objeví opět původní problém: U přidání další třídy Element musíme totiž „hrábnout“ do všech již napsaných Visitorů a přidat další operaci pro návštěvu tohoto prvku.

Sběrné Visitory

Visitor se dá použít k „čemukoliv“, ale nejčastěji se používá ke sbírání údajů, které se získávají polymorfně od členů struktury. Podobně si můžeme představit sběr jiných údajů, než je obsah obrazců (nad zaměstnanci, nad cennými papíry apod.).

Související vzory

VISITOR se mnohdy pouští do zpracování struktury COMPOSITE.

Lze použít také ITERATOR, kdy se klient dostává sekvenčně k prvkům a nechá projít VISITORA strukturu pomocí ITERATORU.

Struktura může použít i FLYWEIGHTS, je třeba zvážit dopad tohoto řešení na funkcionalitu VISITORA.

Poznámka: Tímto jsme probrali všechny vzory. Existuje ještě vzor RDB Persistor (vytvořený ze zkušeností z projektů), který má samostatný produkt v edici „Open Model and Source“, viz http://www.objects.cz

Co dále...?

Kniha je dočtena, předložené vzory prostudovány. Předpokládám, že čtenář v této chvíli usoudil, že návrhové vzory jsou opravdu pěkná, potřebná a užitečná věc. Vzniká otázka, jak pokračovat...

Tato otázka má dvě roviny:

- Osobní studium
- Firma, kde působím

Osobní studium

Pro vaše osobní studium doporučuji doplnit literaturu o knihy a dokumenty uvedené na konci. Domnívám se, že tato e-kniha, kterou jste právě dočetli, vám bude velice nápomocna při čtení jinak obtížné knihy [5]. Kniha [5] totiž není zaměřena na nějaké hluboké vysvětlení, ale určitě v ní naleznete podnětné a podrobněji rozebrané myšlenky…

Další věc, kterou doporučuji, sledujte stránky www.objects.cz a také diskusní fórum, které bylo založeno právě k této knize o Design Patterns. Odkaz na toto diskusní fórum najdete také na www.objects.cz . Máte-li připomínky, názory, podněty apod., napište prosím do diskusního fóra.

Firma

Osobní studium není dostatečné, pokud pracujete v týmu. Jak dosáhnout toho, aby ostatní ve firmě také přijali tuto myšlenku? Jak nastartovat proces zavádění vzorů v mé firmě? V tomto ohledu je na počátku nejdůležitější navodit správnou atmosféru zájmu opravdu všech pracovníků.

K tomuto účelu se hodí jako první krok ***„Školení Design Patterns“***. Toto školení je dvoudenní, probíhá ve vaší firmě a přednáší je autor této e-knihy.

Školení má jednu velkou výhodu - zapojuje všechny účastníky do procesu přijímání Design Patterns a startuje tak nezbytné vnitřní mechanismy zájmu a nadšení pro tuto oblast v celé firmě. Pokud se vám jeví správné vzory používat, navrhněte a prosaďte myšlenku školení Design Patterns ve vaší firmě.

Bližší informace naleznete také na stránkách www.objects.cz .

Přání

Jako autor této e-knihy doufám, že se s vámi, čtenáři, potkám, ať už osobně na školení, konferencích anebo na diskusních stránkách k této knize. Také doufám, že oblast Design Patterns vysvětlená v této knize se stane přínosnou pro rozšíření vašeho odborného obzoru a povede ke zkvalitnění vaší práce při tvorbě SW.

Seznam literatury

[1] Ilja Kraval: Objektové modelování a UML 2000, http://www.objects.cz (zdarma)

[2] Ilja Kraval: Úvod do sémantiky UML 1.3 s komentářem, http://www.objects.cz (zdarma)

[3] James W. Cooper: The Design Patterns JAVA Companion, (zdarma)

[4] James W. Cooper: Design Patterns and Object Oriented Programming in Visual Basic 6 and VB.NET (zdarma)

[5] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: Design Patterns - Elements of Reusable Object-Oriented Software, ISBN 0 - 201 - 63361 - 2