

# POUŽITÍ VZORU COMMAND K FLEXIBILNÍ VÝMĚNĚ AKCE UVNITŘ KOMPONENTY BEZ JEJÍ REKOMPILACE

RNDr. Ilja Kraval, září 2009

<http://www.objects.cz>

## CO DĚLÁ SYSTÉM TRANSPARENTNÍM A FLEXIBILNÍM?

Když posuzujeme vlastnosti navrženého informačního systému, zřetelně vidíme dva extrémní případy návrhu: Informační systémy, které jsou vytvořeny tak zvané „elegantně“, a systémy navržené jako „paskvily“. Jinak řečeno při pohledu na IS v realizaci až do kódu spatřujeme na jedné straně „kód pěkně napsaný“ a na straně druhé vyslovíme odsudek kulantně řečeno „to je ale čuňácký kód!“.

Zajímavé na tomto rozdělení systémů podle kvality návrhu je skutečnost, že oba systémy, jak elegantně navržený systém, tak paskvil, mohou být dobře funkční, a přesto jeden z nich odsoudíme a druhý pochválíme. Při posuzování kvality návrhu totiž vidíme zřetelné rozdíly v těchto dvou rozhodujících vlastnostech systému: Transparence systému a jeho flexibilita.

### **Transparence systému**

Informační systémy napsané elegantně jsou dobře přehledné, dobře čitelné, a i když řeší složité věci, jsou jednoduché. Paskvil je nečitelný, nepřehledný a je zbytečně složitý, mnohdy překvapivě i u jednoduchých systémů. Není to však jenom otázkou dokumentace (například analýzy apod.) nebo dobrých poznámek a komentářů v kódu (i když i ty jsou velmi důležité!). Hlavní rozdíl je jinde: Spočívá v samotné skladbě návrhu systému a tedy následně i skladbě kódu. Pěkně napsaný kód (oproti škaredému) je dobře čitelný již svou podstatou, tj. jeho konstrukce je taková, že princip návrhu IS je zřetelně vidět dokonce s minimem poznámek a dokumentace. Paskvilně navržený IS můžeme dovybavit spoustou doprovodné textové dokumentace, dodat další množství komentářů a poznámek... a přesto zůstane nečitelným a nesrozumitelným.

## **Flexibilita návrhu systému**

Navíc u paskvilu kromě nízké transparence zjišťujeme další nepříjemnou vlastnost: Při každém pokusu o změnu se musíme vypořádat až s příliš velkou režíí a pracností nutnou pro požadovaný zásah do systému.

Samozřejmě na jedné straně se jedná o přímý důsledek nízké transparence systému, protože máme-li něco změnit, musíme nejprve v systému přesně dohledat tyto body změn, a to je v nízko-transparentním paskvilním systému opravdový horor. Poté, když už víme, co je třeba změnit, jsme nemile překvapeni tím, že tyto změny jsou z hlediska otevření kódu, jeho přepsání a opětovné recompile dost bolestivé. To je další vlastnost paskvilu, kterou programátoři nazývají slangově jako „kožený kód“.

Míra flexibility návrhu je dána odpovědí na jednoduchou otázku: „Jak nás to bude bolet, když se daná určitá věc v návrhu systému změní?“. Samozřejmě uvedená otázka nemusí mít smysl, protože „daná určitá věc v návrhu systému“ se nemusí nikdy změnit. (Mimochodem připomenu jeden z Murphyho zákonů platných pro práci analytika: „Prvním požadavkem od zákazníka na změnu systému po nasazení je právě ten, o kterém zákazník při analýze tvrdil, že nikdy nenastane.“)

## **Společným jmenovatelem elegantně navržených systémů je re-use**

Vidíme, že elegantně vytvořený software vykazuje dvě základní vlastnosti: Vysokou míru transparence (a to i s minimem dokumentace) a vysokou míru flexibility.

Tyto dvě vlastnosti jsou dány společným jmenovatelem a tím je dodržování opětovné použitelnosti, neboli dodržování znovupoužitelnosti, slangově „re-use“.

Platí totiž přímá úměra: Čím více dodržujeme opětovnou použitelnost, tím více je systém transparentní a tím více je flexibilní a přesně naopak. Pokud máme prvky v systému kódovány unikátně (dodržujeme v maximální míře re-use), tyto prvky jsou přesně na svých logických místech a nikde jinde. Sám systém se tedy stává sebe-popisný v tom, kde co logicky najdeme. To se týká nejen modulů a tříd v nich, ale také funkcionalit.

Mimochodem požadavek na maximální re-use logicky vede k nutnosti modelovat již na úrovni analýzy a designu (tj. kód nestačí), jinak si totiž nemůžeme opětovnou použitelnost prvků pojmenovat a pojmově vystihnout. Například pokud se dva programátoři z různých oddělení baví u oběda o své práci a oba s překvapením zjistí, že pracují na stejné agendě, tj. tvoří totéž (například oba tvoří agendu evidence pohledávek a to je přesně proti principu re-use), tak se v té chvíli baví na analytické úrovni a nikoliv jako programátoři. Z toho je tedy zřejmé, že se bez analytického modelování, které opětovnou použitelnost vystihuje pojmově, v podstatě neobejdeme, pokud chceme dodržovat maximální re-use.

S flexibilitou je to malinko složitější: Pokud totiž chceme dosáhnout skutečné nirvány v re-use (tj. 100% re-use), nevyhneme se v návrhu systému zavedení polymorfního chování a to již na analytické úrovni. Teprve až polymorfismus zavádí ten nejvyšší stupeň re-use a bez něj 100% opětovné použitelnosti nikdy nedocílíme.

Problém je v tom, že bez zavedení polymorfismu nemůžeme vyměnit část funkcionality za jinou, aniž bychom do daného kódu zasahovali. Jako příklad lze uvést situaci, kdy se potřebuje znovu použít tentýž kód jinde, ale v „malé části uprostřed“ se daný kód „trochu“ liší. Opravdovou „čuňárnou proti re-use“ je zvolit postup, kdy se kód zkopíruje do požadované druhé pozice a v této kopii se v dané části natvrdo změní (rozdílná část se v kopii překóduje). Menší „čuňárnou“ (ale stále nikoliv 100% re-use) je zavedení větvení v daném bodě (tj. switch přes hodnotu proměnné), který by v daném bodě přepínal tyto dvě funkcionality. Před spuštěním se nastaví 1 nebo 2 a podle toho se přepínačem „vymění“ v daném bodě funkcionality. Problém s re-use je v tom, že pokud chceme přidat další použití (tj. další funkcionality pro větev 3), musíme tento kód překopat a přidat novou větev. A jak vidíme, to již není opětovné použití, ale překopání kódu. Nirvány re-use dosáhneme až teprve nasazením polymorfismu, tedy použije se jeden z odpovídajících vzorů objektového návrhu (vzory zvané v literatuře jako GOF podle počtu čtyř autorů „gang of four“). Zde se jedná buď o budování dokumentu z části systému (vzor BUILDER), anebo volba algoritmu (vzor STRATEGY) resp. jiný vzor GOF.

Je třeba zdůraznit, že problém zavedení polymorfního chování není jen otázkou techniky „jak někde vyměnit kus kódu“. Jedná se o jeden ze základních principů, jak navrhnout elegantní opětovně použitelný návrh již na analytické a na technologické úrovni a poté v kódu. Proto by měli vývojáři (a to jak analytici, tak technologové a programátoři) dobře znát Design Patterns - GOF, jinak jim hrozí „paskvilnost“ návrhu IS díky nenasazení polymorfismu tam, kde je opravdu třeba jej použít.

*Poznámka: Doporučoval bych věnovat pozornost [cenově výhodnému školení Design Patterns přímo v Praze](#), jehož termín se vzhledem k datu vydání článku blíží (poslední možnost přihlášení je 25.9.2009).*

Jako příklad takového správného postupu jsem pro tento článek zvolil problém „flexibilně vyměnitelná akce v subsystému“.

## JEDEN Z MOTIVŮ VZORU COMMAND

Vysvětlení zahájíme motivem vzoru. Obecně doporučuji, abyste vždy, když studujete vzor, začali od motivu. Je to část vzoru, která zavádí jeden nebo více problémů k řešení jako příklady. Ty se vyřeší a následně se nalezené řešení zobecní do vzoru. Motiv je tedy velmi dobrým vodítkem k pochopení fungování daného vzoru a proto je dobré začít studium vzoru právě motivem.

Jedním z motivů vzoru COMMAND je následující zadání:

Představme si, že jsme navrhli a vytvořili část systému, která bude realizována jako komponenta v Javě nebo .NET anebo jako modul (C++, Pascal apod.). Zjistíme, že v tomto modulu se v určitém bodě funkcionality má vykonat nějaká akce, o které se dá předpokládat, že bude v budoucnu vyměnitelná anebo ji chceme vyměňovat v závislosti na konfiguraci produktu u zákazníka.

Jako příklad takovéto akce bych uvedl spuštěnou reakci na výjimku, kterou chceme do budoucna vyměnit. Do včerejška se reagovalo na výjimku „takto“, odedneška jinak, a to chceme flexibilně vyměnit.

Jako další příklad bych uvedl akci spuštěnou jako ošetření dané vstupní hodnoty editačního pole, ale samo ošetření hodnoty se liší zákazník od zákazníka. Tedy u zákazníka X (konfigurace A) se pole ošetřuje takto, u zákazníka Y (konfigurace B) jinak.

Jiným příkladem může být spuštěná akce při přechodu ze stavu X do Y, přičemž se dá předpokládat, že tato spuštěná akce se může časem vyměnit za jinou anebo bude vyměnitelná podle konfigurace zákazník od zákazníka.

Úkolem je zvolit takové řešení, aby se ve všech těchto případech dala akce jednoduše vyměnit, ale daná komponenta (tj. modul) obsahující tuto akci se při výměně nebude v kódu otevírat. Kód komponenty se tedy nebude měnit a komponenta se nebude překompilovávat, přesto však k požadované výměně dojde.

Ještě než přistoupíme k řešení pomocí vzoru COMMAND, dovolím si několik poznámek k tomuto problému z hlediska řízení projektu, z pohledu kompetence rolí v systému a ve vztahu ke stabilitě vývoje systému.

Je zřejmé, že na daném řešení se podílí analytik, protože je to on, kdo by měl s budoucím zákazníkem vyhledávat takovéto body vyměnitelných akcí, tj. určit, kde budou vyměnitelné akce nasazeny a kde jich není třeba. Z toho důvodu by měl analytik vědět, že řešení vyměnitelných akcí pomocí vzoru COMMAND je technicky možné, z čehož logicky plyne nutná znalost analytika v otázce podstaty fungování vzorů GOF. Takže i analytikovi by měly

být známy základy Design Patterns GOF (zde například podstata vzoru COMMAND). Úkolem analytika je následně spolu se zákazníkem odhalit, kde všude se vyžaduje nasadit takovéto vyměnitelné akce, a to je problém již na úrovni analytických požadavků.

*Poznámka: Jen na okraj upozorním, že z hlediska jazyka UML existuje pro vyjádření vyměnitelných akcí několik možných technik zápisu v diagramech případů užití (USE CASE DIAGRAM), to je však námětem jiného článku a jiných školení než Design Patterns.*

Za druhé, při pohledu na úkol motivu je zřejmé, že tento problém se týká i vedoucího projektu. Pokud totiž jako vývojáři známe a nabídneme řešení, kdy se do „starého kódu“ nechodí, potom srdce vedoucího projektu jenom zaplesá. Každý zásah do kódu je pro něj totiž potencionálním zdrojem chyb (*pozn.: Další Murphyho zákon dokonce říká, že každý zásah do kódu včetně opravy vnáší do kódu alespoň jednu chybu*). Proto i vedoucí projektu bude jenom rád, pokud se výměna akce provede flexibilně bez zásahu do stávajícího kódu. Z toho důvodu i vedoucí projektu by měl být v hrubých rysech seznámen se smyslem a posláním jednotlivých vzorů (zde COMMAND), aby měl tu možnost posoudit kvalitu vývoje u svého týmu. Měl by vědět, že se buď vzory GOF nenasazují vůbec (katastrofická varianta), proto se kód neustále zbytečně otevírá a předělává, anebo v opačném ideálním případě se vzory používají správně jako dobrá flexibilní řešení a systém je vývojově stabilnější. Vedoucí projektu by měl vědět, že existují možnosti výrazně zvýšit stabilitu vyvíjeného systému, protože se minimalizují zásahy do již existujícího kódu a zkompileovaných komponent. Proto by měl vedoucí projektu v hrubých rysech o vzorech GOF vědět, tj. měl by znát problémy, které GOF řeší a měl by chápat logickou podstatu řešení těchto problémů. Jinak mu připadá normální, že kód modulů je ve vývoji stále otevřen, a přitom ani nebude tušit, že se tento jev dá pomocí GOF výrazně omezit.

Ještě horší situace nastane, pokud vedoucí z neznalosti problému odmítá vyslat své podřízené na školení, ve kterém by se pracovníci z jeho týmu seznámili s postupy, jak tvořit flexibilní software. Rádoby ušetřený čas díky odmítnutému školení se pak negativně projeví dost drasticky o několik řádů vyššími časovými ztrátami vzniklými neustálými úpravami neflexibilních paskvilních systémů jak při jejich vývoji, tak poté i při jejich údržbě.

### **Pokračování příště**

**[Věnujte pozornost cenově výhodnému 2-dennímu školení Design Patterns přímo v Praze](#)**