MOŽNOSTI ZNOVUPOUŽITIA ČASTÍ NÁVRHOVÝCH VZOROV

Jaroslav Jakubík
Fakulta informatiky a informačných technológií,
Slovenská technická univerzita v Bratislave,
jakubik@fiit.stuba.sk

ABSTRAKT:
Vzory sú v súčasnosti súčasťou rôznych katalógov a stále častejšie sa používajú v rôznych fázach vývoja softvéru od analýzy cez hrubý návrh architektúry, detailný návrh až k implementácií. Při použití návrhových vzorov je identifikovaných viacero problémov. Jedným z nich je i znovupoužitie častí vzorov na úrovni zdrojového kódu, ktorých implementácie sa opakujú v odlišných doménach. Článok sa zaoberá návrhovými vzormi, konkrétne znovupoužitím všeobecných častí návrhových vzorov s využitím v súčasnosti dostupných prostriedkov, analyzuje prístup k znovupoužitiu vzorov formov frameworkov, knižníc, CASE nástrojov a rozšírení jazyka o špeciálne konštrukcie.

KĽÚČOVÉ SLOVA:
návrhový vzor, znovupoužitie, CASE nástroj, knižnica vzorov

1 MOTIVÁCIA

V súčasnosti sú dostupné množstvá katalógov a zbierok vzorov [2], [5] atď., či už návrhových alebo iných, zameraných na konkrétnu oblasť v životnom cykle softvérového projektu (analýza, architektúra [3], návrh [5] atď.) alebo na konkrétnu doménovú oblasť (skladové hospodárstvo, bankovníctvo, poisťovníctvo atď.). Použitie vzorov sa stalo odporúčaným prístupom k vývoju aplikácií vo všeobecnosti.
Vzory v rôznych katalógoch sú často krát opísané na postačujúcej úrovni podrobnosti pre fázy analýzy a návrhu. Úroveň podrobnosti je v rôznych katalógoch na rôznej úrovni, väčšinou ide o neformálny ale štruktúrovaný opis dopĺňaný rôznymi modelmi vyjadrených formou diagramov tried, interakčnými, sekvenčnými a ďalšími diagramami. Z tohto opisu v závislosti od úrovne podrobnosti je možné pochopiť základný princíp vzoru, ústrednú myšlienku, spolupracujúcich participantov i niektoré vzťahy medzi nimi. Súčasťou katalógov sú i jednoduché príklady implementácie vzoru.
Pokiaľ ale jednoduchý príklad nepostačuje na pochopenie všetkých vlastností vzoru je potrebné hľadať ďalšie zdroje k použitiu konkrétneho vzoru. Takýmto spôsobom vznikajú rôzne implementácie vzorov, ktoré sa často krát vymykajú pôvodnej myšlienke aplikovaného vzoru. Znovupoužitie nezávislých častí vzoru, ku ktorému smerujeme, nie je jednoduché vzhľadom na skutočnosť, že vzory kombinujú všeobecné časti s doménovo závislými. Tieto dve časti vzoru sú úzko spolu spojené a nie je vždy jednoduché ich prehľadným a všeobecným spôsobom oddeliť.
Existujú rôzne pokusy ako vzory znovupoužiť či už na báze CASE prostriedkov, knižníc alebo inými viac alebo menej podobnými spôsobmi. Každý z prístupov má svoje výhody i nevýhody a je iba na rozhodnutí návrhára, ktorý zo spôsobov znovupoužitia implementácie časti vzoru si vyberie a tým čiastočne odľahčí vývojára pri implementácií vytváranej aplikácie.
Správnym výberom môže návrhár získať možnosť znovupoužiť už predtým vytvorené doménovo nezávislé časti. Na druhej strane, nesprávnym výberom môže dôjsť

k skomplikovaniu vývoja aplikácie, nutnosti preštudovať a vyskúšať nové technológie a požadovaný efekt nemusí na záver spĺňať pôvodné očakávania.
Tento článok sa zaoberá návrhovými vzormi definovanými v [5] resp. možnosťami znovupoužitia častí vzorov rôznymi metódami s použitím rôznych prostriedkov. V nasledujúcich kapitolách budeme analyzovať rôzne možnosti znovupoužitia častí vzorov. V kapitole 2 špecifikujeme znovupoužitie vzorov vo frameworkoch podľa [8], v kapitole 3 v knižniciach podľa [1] a [6], v kapitole 4 sa sústredíme na CASE nástroje a ich podporu práce s návrhovými vzormi [7] a v kapitole 5 spomenieme špeciálny prístup cez rozšírenie jazyka podľa [7]. Záver a sumárne porovnanie jednotlivých prístupov spolu s námetmi pre ďalšiu prácu sú uvedené v kapitole 6.

2 ZNOVUPOUŽITIE ČASTÍ VZOROV VO FRAMEWORKOCH

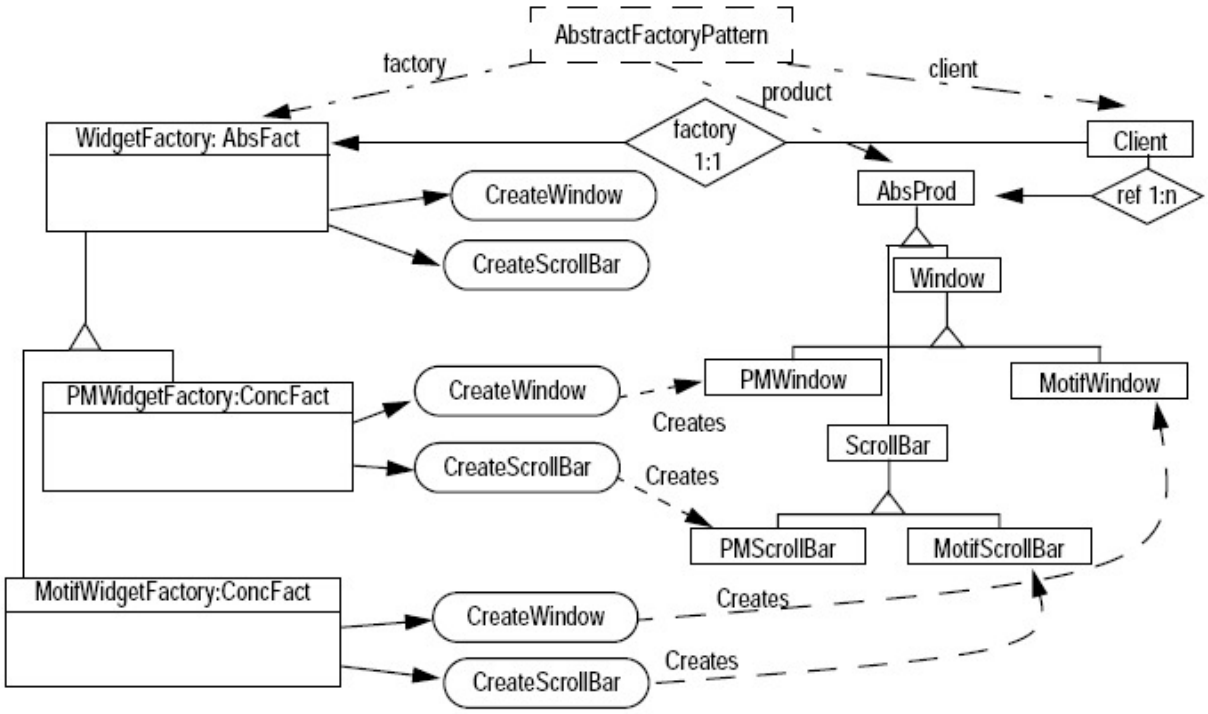
Framework je softvérový produkt, resp. balík spájajúci skupinu produktov, špeciálne určený pre podporu zjednodušeného vývoja aplikácie v konkrétnej doméne. Podpora vývoja je najčastejšie realizovaná formou znovupoužitia. [1]
Pri použití klasických objektovo orientovaných frameworkov je potrebné detailne poznať doménu vyvíjanej aplikácie i mechanizmy frameworku (stromy dedenia, mechanizmy špecializácie atď.) na úrovni podobnej ako tvorca frameworku. Navyše vývojár aplikácie je obmedzený na jazyk použitý vo frameworku.

2.1 FACE FRAMEWORK

FACE (Framework Adaptive Composition Environment) je v porovnaní s klasickými objektovo orientovanými frameworkami prostredie pre vytváranie aplikácií, v ktorom je použitý framework. Aplikácie sú vytvárané jednoduchým skladaním vzorov založenom na modelovacom prístupe. Vytvorenie aplikácie pozostáva z vytvorenia konzistentného a úplného modelu využitím modelovacích primitív definovaných frameworkom. FACE definuje nasledujúce modelovacie primitívy:

- triedy a ich roly vo vzore
- operácie a ich roly vo vzore
- vzťahy medzi triedami a operáciami špecifické pre vzor
- ďalšie parametre na došpecifikovanie pre vzor konkrétneho správania

FACE spája abstrakciu na úrovni vzorov priamo s ich implementáciou a podporuje inkrementálny vývoj aplikácie, čím čiastočne rieši problém rozdielov medzi návrhom a implementáciou (design – implementation gap) identifikovaný v [8].
Návrhové vzory resp. ich schémy sú koncentrované vo frameworku FACE. Každý návrhový vzor je vo FACE frameworku rozdelený na prvotnú a doménovú schému.
Prvotná schéma definuje abstraktné triedy a ich vzťahy, definuje základné štruktúry doménovej schémy. Pre definovanie prvotnej schémy je použitý špeciálny modelovací jazyk. Doménová schéma je konkrétnou inštanciou prvotnej schémy, ku ktorej dopĺňa doménové triedy a operácie pre definovanie použitia vzoru.
Všeobecná a teda znovupoužiteľná časť vzoru je umiestnená v primal schéme. Primal schéma je v modeli aplikácie doplnená o konkrétne doménovo závislé (aplikačne špecifické) časti. Zdrojový kód aplikácie (resp. vzoru) nemôže byť vygenerovaný bez došpecifikovania konkrétnych tried k zodpovedajúcej primal schéme – model nie je konzistentný.
Zdrojový kód výslednej aplikácie je pri použití FACE frameworku generovaný až po vytvorení konzistentného modelu.
Na obrázku 1 je zobrazený model použitia návrhového vzoru Abstract Factory vo FACE frameworku.



Obr. 1: Schéma použitia návrhového vzoru Abstract Factory vo FACE frameworku

3 ZNOVUPOUŽITIE ČASTÍ VZOROV V KNIŽNICIACH

Realizácia znovupoužitia častí vzorov je možná formou uloženia všeobecných častí vzorov do knižnice. V článku [1] autori predstavujú knižnicu pod názvom DPL (Design Pattern Library) v jazyku Beta.

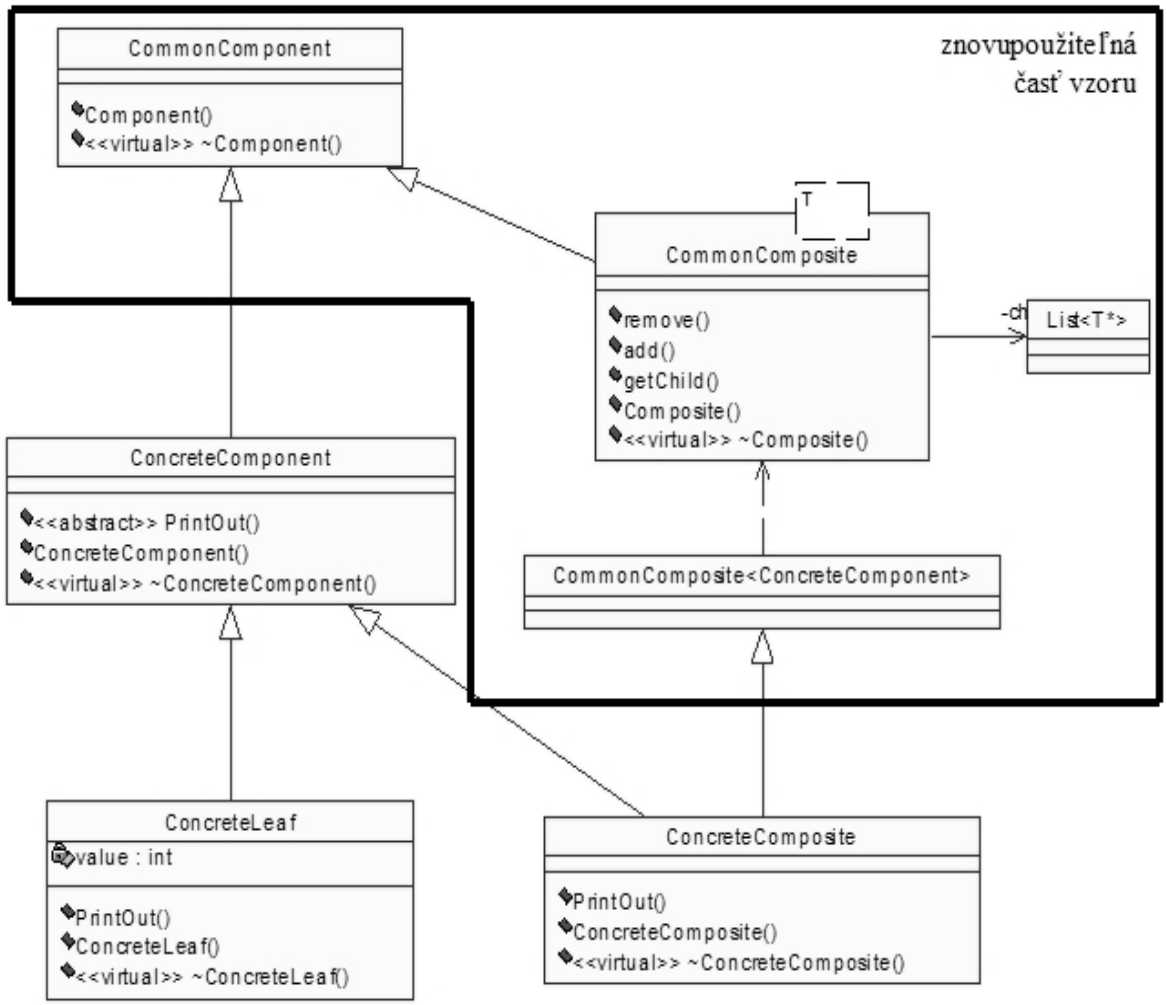
Jazyk Beta je špeciálny objektovo – orientovaný programovací jazyk. Beta disponuje špeciálnymi vlastnosťami a prostriedkami ako napríklad virtuálne zoznamy a premenovávanie.

Premenovávanie umožňuje dynamicky počas behu programu meniť názvoslovie používané v konkrétnej triede (resp. objekte). Vytváraný objekt je parametrizovaný názvom konkrétnej metódy podobným spôsobom ako parametrizácia šablóny v C++ konkrétnym typom.

Virtuálne zoznamy sú používané na definovanie metód v abstraktných triedach alebo rozhraniach počas behu programu. Abstraktné triedy sú opäť parametrizované podobným spôsobom ako šablóny v C++.

Autori v plnej miere využili ponúkané prostriedky jazyka Beta a vytvoril kompletnú knižnicu znovupoužiteľných častí vzorov podľa [5].

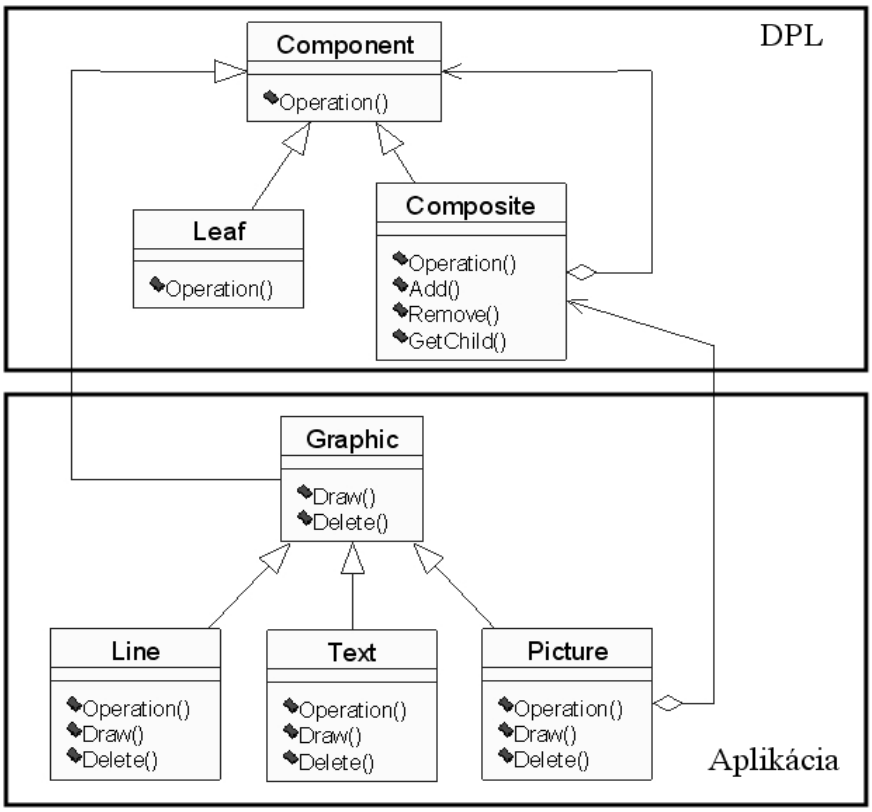
V [1] môžeme nájsť opis celej knižnice a tiež procesu vytvárania tejto knižnice. Každý zo vzorov je v knižnici rozdelený na dve časti. Prvá časť je súčasťou samotnej knižnice. Ide o znovupoužiteľnú časť vzoru, ktorá je nezávislá od použitia vzoru. Druhá časť je doménovo špecifická a teda je súčasťou konkrétnej aplikácie. Doménovo špecifická časť dopĺňa a prispôsobuje všeobecnú časť pre aplikáciu, v ktorej je konkrétny vzor použitý. Ako príklad rozdelenia vzoru na všeobecnú a doménovo závislú časť uvádzame rozdelenie vzoru Composite. Z obrázku 3 je jasné znovupoužitie metód (a ich implementácií) spojených s kontajner manažmentom vo vzore Composite v triede `Composite`. Na obrázku 3 sú tiež zobrazené vzťahy medzi časťami vzoru.



Obr. 2: Realizácia rozdelenia vzoru Safe Composite na znovupoužiteľnú a doménovo závislú časť v jazyku C++ [1]

V [1] autori použili špeciálny jazyk Beta, ktorý svojimi zložitými konštrukciami umožnil jednoduché a prehľadné rozdelenie vzorov na časť znovupoužiteľnú a aplikačne resp. doménovo závislú. V štandardných jazykoch sa ale pri takomto prístupe stretávame s problémom ako vybudovať špeciálne konštrukcie jazyka Beta inými prostriedkami.
Na obrázku 2 podľa [6] je v diagrame tried znázornený spôsob rozdelenia návrhového vzoru Composite (konkrétne Safe Composite) pomocou prostriedkov dostupných pre jazyk C++. V článku [6] je tiež popísaný postup pre rozdelenie oboch variácií vzoru Composite bezpečnej a transparentnej pomocou šablón a viacnásobného dedenia.

[1] Riešenie je zobrazené v diagrame tried doplnenom o zvýraznenie znovupoužiteľnej časti vzoru pomocou CASE nástroja Rational Rose



Obr. 3: Štruktúra použitia vzoru Composite z knižnice DPL v jazyku Beta

4 ZNOVUPOUŽITIE ČASTÍ VZOROV V CASE NÁSTROJOCH

Ďalšia možnosť ako znovupoužiť všeobecné časti návrhových vzorov je využitie CASE nástrojov konkrétne modulu pre prácu s návrhovými vzormi a modulu generovania zdrojového kódu. Zaoberali sme sa porovnaním CASE nástrojov na základe podpory práce s návrhovými vzormi. Definovali sme kritéria (Podpora vývojových jazykov, Podpora vzorov, Generovanie zdrojového kódu vzorov, Vlastnosti generovaného kódu, Vytváranie inštancie vzoru, Podpora mikro-architektúr, Možnosti rozšírenia množiny vzorov), na základe ktorých sme dokázali porovnať podporu práce s návrhovými vzormi, pričom sme sa sústredili na možnosti znovupoužitia zdrojového kódu u definovaných návrhových vzorov.
V tabuľke 1 je uvedené vyhodnotenie troch CASE nástrojov (Rational Rose, Rational XDE, Together) voči definovaným kritériám.

Tab. 1 Porovnanie CASE nástrojov na základe definovaných kritérií

	Rational Rose	Rational XDE	Together
Podpora jazykov	C++, Java	Java, C++, .NET jazyky	Java, .NET jazyky
Podporované vzory	20 GoF vzorov modifikované podľa [4]	23 GoF vzorov	23 GoF vzorov
Generovanie zdrojového kódu	na príkaz používateľa, po vytvorení modelu	na príkaz používateľa, po vytvorení modelu	v priebehu vytvárania modelu
Vlastnosti generovaného zdrojového kódu	kostra projektu, triedy, vzťahy medzi triedami,	kostra projektu, triedy, vzťahy medzi triedami,	kostra projektu, triedy, vzťahy medzi triedami,

	definície metód, podľa použitého vzoru	definície metód, čiastočné telá metód podľa použitého vzoru	definície metód a čiastočné telá metód podľa použitého vzoru
Vytváranie inštancie vzoru	veľmi jednoduchý sprievodca formou jediného dialógového okna, cez ktoré sa obsadzujú jednotlivé roly vo vytváranej inštancií vzoru	možnosť výberu z jednoduchého alebo názornejšieho sprievodcu pre obsadenie rolí vo vytváranej inštancií vzoru	názorný sprievodca spolu s vysvetlením obsadzovaných rolí vytváranej inštancie vzoru
Podpora mikro – architektúr [2]	nepodporuje	nepodporuje	nepodporuje
Možnosti rozšírenia podporovanej skupiny vzorov	nepodporuje	Pomocou tzv. pattern template a code template	v špeciálnom režime, z modelov, ktoré sú k dispozícii

Porovnávané nástroje až na výnimky majú rovnakú podporu pre prácu s návrhovými vzormi. Podporujú takmer všetkých 23 návrhových vzorov podľa [5] resp. podľa modifikácií vzorov pre konkrétny implementačný jazyk napr. pre JAVA podľa [2].
Nevýhodou u všetkých nástrojov je nepodporovanie mechanizmov pre implicitné použitie mikro – architektúr a tiež iba minimálna možnosť zviditeľniť vzor v návrhu systému resp. v návrhu konkrétnej jeho časti.
Generovanie zdrojového kódu u návrhových vzorov je v prípade Rational XDE a Together na vysokej úrovni. Môžeme povedať že ide o istú formu, znovupoužitia zdrojového kódu návrhových vzorov.

5 ZNOVUPOUŽITIE FORMOU ROZŠÍRENIA JAZYKA

Ďalšou z možností ako oddeliť doménovo závislú a všeobecnú časť vzoru môže byť obohatenie jazyka o špeciálne konštrukcie, ktoré sú určené práve na ľahšiu manipuláciu a doplnenie doménovo závislej časti. Všeobecná časť vzoru je opäť uložená v knižnici a do aplikácie sa vnáša s využitím špeciálnych konštrukcií jazyka. Tieto špeciálne konštrukcie nemusia byť primárne určené pre použitie s návrhovými vzormi.
Takýto prístup je ilustrovaný v článku [7], kde autori s využitím koncernov definujú adaptéry, ktoré slúžia na prispôsobenie vzoru uloženého v knižnici pre potreby aplikácie. V článku je definovaných nasledujúcich šesť typov adaptácie:

- implementácia rozhraní
- fúzia tried a metód
- pridanie a redefinícia metód
- pridanie triedy, inštančnej premennej
- zachytenie Before, After, Around a OnException pri metódach
- zachytenie prístupu k triedam a inštančným premenným tried.

Toto rozšírenie je navrhnuté bez obmedzení v rovine programovacieho jazyka a je teda použiteľné nielen s jazykom JAVA.
Autori tento prístup ilustrujú na príklade návrhového vzoru Observer, ktorý je považovaný za jeden koncern. Vzor Observer adaptujú na jednoduché grafické používateľské rozhranie reprezentované niekoľkými tlačidlami a popiskami.

Pri tomto prístupe by bolo možné všetky návrhové vzory opísať pomocou koncernov a uložiť ich do knižnice. Pomocou adaptérov následne prispôsobovať vzory konkrétnej doméne.
V príklade 1 je uvedený kompozičný protokol pre adaptáciu vzoru Observer do kontextu grafického používateľského rozhrania. Súčasťou príkladu 2 je adaptér spájajúci vzor Observer s aplikačným grafickým rozhraním. Oba príklady sú implementované v jazyku JAVA.

```
01  package designpattern.observer
02  abstract adapter ObserverAdapter {
03
04    abstract Class target ‘‘class(es) being used as an observable ’’ : observableClass
05    abstract Method target ‘‘ method(s) triggering observer’s changes ’’ : notifyingMethod
06       require notifyingMethod in observableClass.*
07
08    adaptation becomeObservable ‘‘Modify class in order to make it observable ’’ :
09       extend class ImplObservable with observableClass
10    adaptation notifyingObserver ‘‘ Alter notifyingMethods to tell observers about modification ’’ :
11       extend method notifyingMethod( ... ) with after { notifyObservers(); }
12 ----------------------------------------------------------------------------------
13    abstract Class target ‘‘class being used as an observer ’’ : observerClass
14
15    adaptation becomeObserver ‘‘Modify class to make it an observer ’’ :
16       inherit Observer in observerClass
17    abstract adaptation observerUpdate ‘‘Introduce the updateObserver method in the observer class ’’ :
18       introduce method public void updateObserver(Observable o) in observerClass
19 ----------------------------------------------------------------------------------
20    abstract Class target ‘‘Class(es) initializing observable or observers objects ’’ : applicationInitClass
21    abstract Method target ‘‘Method(s) creating observable or observer objects ’’ : applicationInitMethod
22       require applicationInitMethod in applicationInitClass.*
23    abstract Attribute target ‘‘Attributes(s) pointing out observable objects ’’ : observableInstance
24       require observableInstance in applicationInitMethod.*
25    abstract Attribute target ‘‘Attributes(s) pointing out observer objects ’’ : observerInstance
26       require observerInstance in applicationInitMethod.*
27
28    adaptation initApplication ‘‘Alter applicationInitMethod to insert observers in the list of observables ’’ :
29       extend method ApplicationInitClass.applicationInitMethod( ... ) with after {
30          ObservableInstance.addObserver( ObserverInstance );
31       }
32    }
```

Príklad 1: Kompozičný protokol pre adaptáciu návrhového vzoru Observer [7]

```
01 package application.ihm
02 compose designpattern.observer.* with application.ihm
03 adapter ApplicationIHM extends ObserverAdapter {
04    target observableClass = application.ihm.Button
05    target observerClass = application.ihm.Label
06    target notifyingMethod = application.ihm.Button.fireActionPerformed()
07    target applicationInitClass = application.ihm.ApplicationInterface
08    target applicationInitMethod = applicationInitClass.applicationInterface()
09    target observableInstance = applicationInitMethod.button*
10    target observerInstance = applicationInitMethod.label*
11
12    adaptation observerUpdate : introduce method public void updateObserver(Observable o) {
14       setText(‘‘I have been notified of a button click’’);
15    } in observerClass
16 }
```

Príklad 2: Adaptér pre modifikáciu vzoru Observer pre potreby GUI [7]

6 ZÁVER

Návrhové vzory už z názvu primárne definujú znovupoužitie na úrovni návrhu. Pomocou predchádzajúcich metód je ale možné priamo spojiť návrhové vzory z ich implementáciou. Navyše spomínané metódy umožňujú, každá iným spôsobom, rozdeliť vzor na všeobecnú a doménovo závislú časť. Všeobecná časť je potom znovupoužívaná v rôznych aplikačných doménach.
CASE nástroje využívajú vlastné štruktúry pre uloženie informácií o návrhových vzoroch. Návrhové vzory sú aplikované najčastejšie cez diagram tried a upravené, doplnené pre konkrétnu doménu. Napriek rôznym možnostiam modelovacích prístupov nie je viditeľnosť

vzoru dostatočné výrazná a pri použití viacerých vzorov je návrh neprehľadný a zbytočne zložitý.
Knižnice explicitne delia návrhový vzor na všeobecnú a konkrétnu časť. Všeobecná časť vzoru je súčasťou knižnice a konkrétna časť sa pripája ku knižnici použitím prostriedkov jazyka. Návrhové vzory spolu úzko spájajú obe spomenuté časti a je preto náročné dosiahnuť všeobecné, generické a prehľadné riešenie v kontexte štandardného objektovo orientovaného jazyka.
Rozšírením jazyka o špeciálne konštrukcie získame prístup k neštandardným možnostiam, ktoré môžu byť využité pre znovupoužitie návrhových vzorov resp. ich častí. Všeobecné časti sú opäť externalizované do knižníc a následne s využitím doplnených konštrukcií jazyka použité v aplikácií. Príslušnosť tried alebo operácií ku konkrétnemu vzoru je zachytená v použití ponúkaných prostriedkov jazyka a nezaťažuje samotný model nepotrebnými detailmi. Navyše knižnice umožňujú zachytenie mikro - architektúr, ktoré môžu byť použité v návrhu bez explicitnej znalosti návrhára resp. vývojára.
Prístup s využitím rozšírení jazyka by sme chceli použiť pri vytváraní knižnice návrhových vzorov. Tomuto procesu bude predchádzať bližšie štúdium samotnej štruktúry vzorov ich kategorizácie a ich detailných vlastností.

POĎAKOVANIE

Táto práca je čiastočne podporovaná z projektu Štátneho programu „Budovanie informačnej spoločnosti“ č. 1025/04 a grantu VEGA č. VG1/3102/06.

LITERATURA

[1] Agerbo, E., Cornils, A.: Theory of language support for design patterns. Master's thesis, 1997, Computer Science Department, Aarhus University, Denmark.
[2] Alur, D., Crupi, J., Malks, D.: Core J2EE Patterns: Best Practices and Design Strategies. Prentice Hall / Sun Microsystems Press, 1st edition, June 2001, ISBN 0-130-64884-1.
[3] Bushman F., Meunier R., Rohnert H., Sommerland P., Stal M., A system of patterns. John Wiley & Sons Ltd., 1996, Anglicko, ISBN 0-471-95869-7.
[4] Cooper, J.: Java Design Patterns: A Tutorial. Addison-Wesley Personal Education, February 2000, Massachusetts, ISBN 0-201-485394.
[5] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns, Elements of Reusable Object-Oriented Software. Addison-Wesley Publishing Company, 1995, Massachusetts. ISBN 0-201-63361-2.
[6] Jakubík J., Izolácia všeobecných častí vzoru Composite, Objekty 2005, Sborník príspevku desátého ročníku konference. Václav Snášel (ed.), Vydala Fakulta elektrotechniky a informatiky, VŠB – Technická unverzita Ostrava, 2005, ISBN 80-248-0595-2, pp. 74 - 84
[7] Lahire, P., Quintian, L.: New perspective to improve reusability in object – oriented languages. In: Journal of Object Technology, Vol. 5., No. 1, January – February 2006
[8] Meijler, T., Demeyer, S., Engel, R.: Making design patterns explicit in FACE. Proceedings of the 6th European conference held jointly with the 5th ACM SIGSOFT international symposium on Foundations of software engineering. Zurich 1997, pp. 94 – 110. ISBN 0163-5948.