Java a Návrhové Vzory

BAKALÁRSKA PRÁCA

Filip Gschwandtner

UNIVERZITA KOMENSKÉHO V BRATISLAVE
FAKULTA MATEMATIKY,FYZIKY A INFORMATIKY
KATEDRA INFORMATIKY

9.2.1 INFORMATIKA

Vedúci záverečnej práce
RNDr. Richard Ostertág

BRATISLAVA 2007

Čestne prehlasujem, že som túto bakalársku prácu vypracoval samostatne len s použitím citovaných zdrojov.

. . . . . .

Errata

Abstract

Cieľom bakalárskej práce s názvom Java a Návrhove vzory je vytvoriť plugin do vývojového prostredia Eclipse(verzia 3.2), ktorý by mal priniesť do vývojového prostredia podporu pre používanie návrhových vzorov. Dôvodom voľby danej témy je nedostatočná podpora návrhových vzorov v prostredí Eclipse. Samotný plugin sa sklada z 2 funkcionálne odlišných častí. Prvá časť zavádza podporu pre návrhové vzory v oblasti používania šablón (templates). Pri tvorbe tejto časti sa myslelo hlavne na programátorov pracujúcich v už spomínanom prostredí, pretože takáto funkcionalita dokáže ponúknuť časové zrýchlenie ich práce. Programátori nemusia prácne implementovať mnoho tried,metód či premenných. Stačí spustiť wizard, pozmeniť nastavenia wizardom ponúknuté a návrhový vzor sa lahko vygeneruje. Druhá časť funkcionality súvisí s hľadaním už implementovaného návrhového vzoru v danom projekte. Táto funkcionalita by mala pomôcť programátorovi v rýchlejšom zorientovaní sa a pochopení cudzieho zdrojového kódu. Ak sa v zdrojovom kóde nachádza návrhovy vzor, tak pomocou tohto pluginu bude schopné ho detekovať a tým ziska programátor cenné informácie o vnútornej štruktúre kódu, ktoré z použitia návrhového vzoru vyplývajú.

Kľúčové slová: návrhové vzory, Eclipse, plugin.

Obsah

Úvod

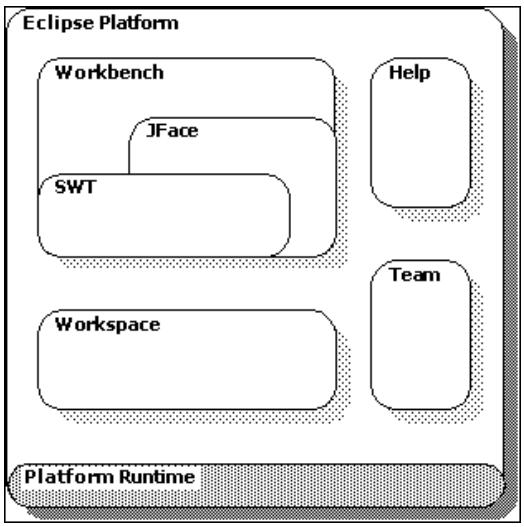
V prvej kapitole sa budeme zaoberať architektúrou vývojového prostredia Eclipsu 3.2. Preberieme si možnosti pripojenia nových pluginov, naučíme sa rozoberať zdrojové java súbory až do najmenších java elementov a ukážeme si ako pracovať s visuálnymi komponentami Eclipsu.
Druhá kapitola bude pojednávať o modeli návrhového vzoru, presnejšie o jeho súborovej XML reprezentácií a o jeho pamäťovej reprezentácií ako skupina tried. Vysvetlíme si tiež ako narábať s XML súbormi.
Posledná tretia kapitola bude pojednávať o implementačných problémoch nesúvisiacich s informáciami o vývojovom prostredí. Stretneme sa s problémom generovania mien tried, metód a iných súčastí modelu návrhového vzoru na účely tvorby templatu. Objasníme si i podstatné myšlienky detekcie návrhového vzoru v existujúcich projektových štruktúrach.

Kapitola 1

Architektúra prostredia Eclipse 3.2

Všetky informácie tejto kapitoly pochadzaju len z 1 officialneho zdroja [1]. Vývojové prostredie Eclipse bolo vytvárané ako velmi pružné prostredie skladajúce sa s veľkého počtu dobre integrovaných pluginov. Využíva sa v ňom idea tzv. otvorenej architektúry. Jedná sa o spôsob vytvárania zložitejšej architektúty vývojoveho prostredia na základe určitého pridávanie vrstiev pluginov. Na samom spodku existuje len relatívne malý runtime engine spravujúci všetky pluginy. Jeho prácou je dynamicky detekovať pluginy, načítavať ich a spúštať. Všetko ostatné je už v Eclipse vytvorené cez pluginy. Nové vrstvy pluginov sa viažu na predošlé pomocou špecialných bodov tzv. **extension points**. Samozrejme, že nové vrstvy môžu definovať vlastné extension pointy, na ktoré sa naviažu ďalšie vrstvy. Samotný Eclipse nie je voľne distribuovaný len ako samotný runtime engine ale už aj s množstvom vrstiev(viď. obr. 1.1), ktoré zabezpečujú skoro všetko potrebné od editora, cez nahľady až po debugovanie a iné.

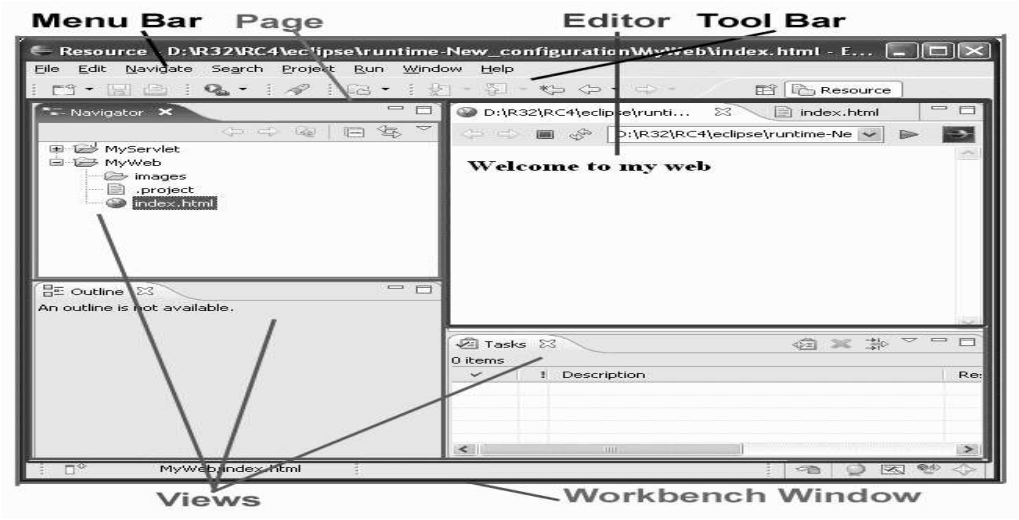
Tabuľka 1.1 mapuje hlavne komponenty Eclipse runtime. Pre nás budú zaujimavé len prvé 3 komponenty uvedené v tabuľke. S ostatnými komponentami nebolo nutné pri implementácií pracovať. Ako prvé pri implementácii pluginu si musíme uvedomiť, ktoré extension pointy chceme použiť. Kedže ideme rozširovať Eclipse o komponenty, ktoré sa budu vnútri Eclipsu graficky zobrazovať tak musíme začať stavať na extension pointoch definovaných komponentom **Workbench User Interface**. Jeho základom je **workbench window** (implementácia interfacu `org.eclipse.ui.IWorkbenchWindow`), ktorá predstavuje celé grafické okno spusteného prostredia Eclipse. Workbench win-



Obr. 1.1: Eclipse Architecture

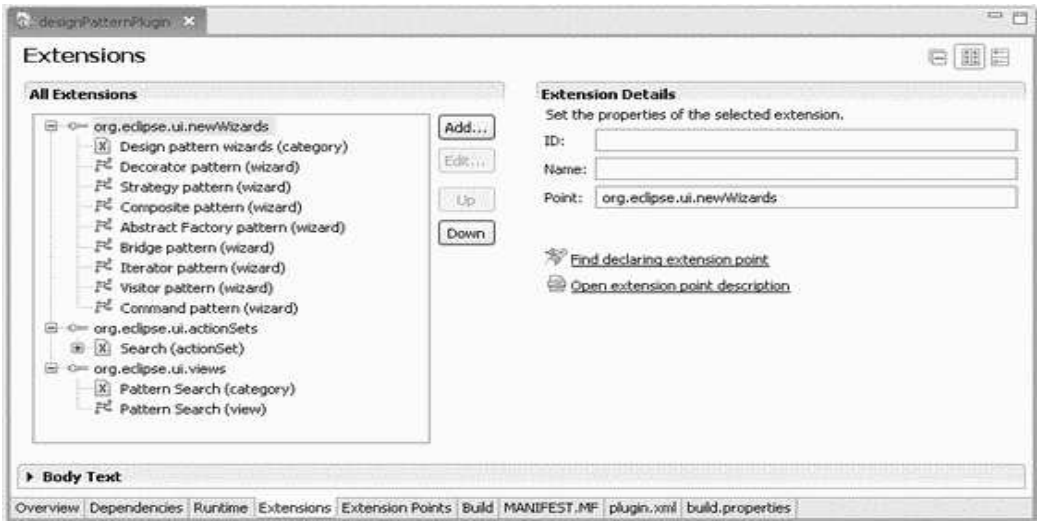
Runtime komponent	**Význam**
Platform runtime	Definuje extension point a plug-in model. Spravuje pluginy (hladanie, spúšťanie).
Resource management (workspace)	Definuje API na vytváranie a správu zdrojov(resources): projekty, subory a priečinky.
Workbench UI	Implementácia user interfacu, definuje extension pointy pre ďalšie UI komponenty a poskytuje ďalšie nástroje (JFace,SWT) na tvorbu user interfacov.
Help system	Definuje exptensin points pre pluginy poskytujúce nápovedu.
Team support	Podpora programovania v skupinach.
Debug support	Definuje model na debugovanie a UI triedy pre tvorbu debuggerov a spúšťačov(launchers).
Other utilities	Pomocné pluginy (vyhľadávanie, porovnávanie zdrojov a iné).

Tabuľka 1.1: Komponenty Eclipse runtime



Obr. 1.2: Workbench Decomposition

dow obsahuje okrem iného i vlastné stránky (pages), z ktorých je aktivné práve 1 stránka. Stránka predstavuje zoskupenie náhľadov (views) a editorov. Obrazok 1.2 znázorňuje čiastočny rozklad Workbench UI na menšie celky, ktoré definujú väčšinou už len 1 extension point. My využijem extension point definovaný časťou **menu bar** (extension point org.eclipse.ui.actionSets), **views**(extension point org.eclipse.ui.views) a časťou, ktorý obr. 1.2 neobsahuje, ale táto časť je zodpovedná za wizardy, ktoré sa používajú pri vytváraní nových tried, projektov, interfacov a iných vecí (extension point org.eclipse.ui.newWizards). Prvé 2 extension pointy sa použijú na 2.časť pluginu. Cez hlavné menu sa zvolí položka (extension point org.eclipse.ui.actionSets). Následne sa vytvorí okno (Dialog), ktoré naplní používateľ údajmi o hľadaní návrhového vzoru. Zatvorením okna pomocou tlačidla OK sa spustí požadovaná akcia a výsledky sa zobrazia v nami nadefinovanom náhľade (view), ktorý definujeme pomocou extension pointu org.eclipse.ui.views. Posledný extension point sa použije na pripojenie Wizardu, ktorý vytvára template návrhoveho vzoru, k prostrediu Eclipse.
Ideologicky by už aj bola vysvetlená architektúra Eclipse a pripájenie sa ďalších pluginov k tomuto prostrediu, takže prejdime ku praktickej stránke pripájania pluginov. Pozrime sa na to čo taký plugin project potrebuje obsahovať, aby bol skutočný projekt vytvárajúci plugin pre Eclipse a nie len



Obr. 1.3: Editor pre plugin.xml

bežný java projekt spustiteľný v ktoromkoľvek inom prostredí. Prvým viditeľným rozdielom je existencia dvoch súborov : **manifest.mf** a **plugin.xml**. Súbor manifest.mf obsahuje nízko úrovňové informácie ohľadom zaobalenia pluginu. Samotný platform runtime (viď. obr. 1.1) je implementovaný ako OSGi framework a bundle (balík) ,základny kameň frameworku, prestavuje samotný plugin. Súbor manifest.mf hovorí frameworku základné informácie o bundle, t.j. pluginu. Jedná sa o základné informácie ako meno,ale jedná sa napríklad aj o to, ktoré iné bundles daný bundle potrebuje pre svoj správny chod. Súbor plugin.xml je z pohľadu pripájania oveľa zaujimavejší, pretože obsahuje informácie o všetkých extension pointoch, ktorý daný plugin využíva. Ako prípona prezrádza, jedná sa o xml súbor, avšak prostredie Eclipse má v sebe implementovaný vhodný editor presne na súbor plugin.xml (viď.obr.1.3) Každý extension point pridáva daľšie súčasti k celkovej architektúre Eclipsu a editor umožnuje ľahké pridávanie, odoberanie a editáciu týchto súčastí. Tak napríklad extension point org.eclipse.ui.newWizards pridáva wizardy a kategórie, do ktorých môžeme wizardy uložiť. Pomocou editora si pridáme 1 kategóriu a pár wizardov. Pre kategóriu nastavíme meno, pod ktorým ju chceme vidieť, a jednoznačné[1] ID (identifikačný String). Pre wi-

[1]Jednoznačné pre celý Eclipse,t.j. pre všetky pluginy (v prípade napojenia triedy ku súčasti sa použije celé meno triedy)

zard sa tiež nastaví meno,ID a ešte navyše trieda, ktorá implementuje wizard (nutne musí implementovať interface `org.eclipse.ui.INewWizard`,väčšinou potomok `org.eclipse.jface.wizard.Wizard`), meno kategórie, v ktorej sa nachádza (t.j. kategória,ktorú sme si definovali), a ikona, ktora sa bude zobrazovať pri vstupe do wizardu. Samozreme, že by sa dalo nastaviť oveľa viac vecí, ale s týmto si vystačíme. Podobne budeme postupovať v prípade extension pointu org.eclipse.ui.views len s tým rozdielom, že wizard bude view. V prípade extension pointu org.eclipse.ui.actionSets sa môžu pridávať len actionSets, pričom actionSet je množina definujúca nové prvky do hlavného menu Eclipse (prvky typu menu) ale aj nové prvky reprezentujúce akciu (prvky typu action). Vytvoríme si 1 actionSet a doňho pridáme 1 prvok typu menu a 1 prvok typu action.
Menu mechanizmus v Eclipse funguje tak, že prvok typu menu vytvorí top level prvok (ako sú napr. „File“,„Search“) a definuje mu určité skupiny v určitom poradí. Tieto skupiny budú neskôr slúžiť prvkom typu action na určenie ich cesty v menu. Takto sa prvok typu action pomocou správne nastavenej cesty attribútom `menubarPath` zaradí do menu. Presnejšie povedané, zaradí sa na koniec za všetky doteraz pridaná prvky typu action v zadanej skupine vo vnútri menu. V prípade vyvolania tohto action prvku v menu prichádza na rad trieda associovaná k action prvku, ktorá implementuje interface `org.eclipse.ui.IWorkbenchWindowActionDelegate` (v určitých prípadoch `org.eclipse.ui.IWorkbenchWindowPulldownDelegate`). Presnejšie povedané, zavolá sa metóda run(IAction action), ktorá je súčasťou predka obidvoch interfacov (`org.eclipse.ui.IActionDelegate`).
Okrem vytvorenie potrebného menu a jeho skupín, nastavenia attribútu `menubarPath` a `class` pre action prvok, treba este dodefinovať maličkosti ako jednoznačné ID a label pre prvok typu menu a typu action. Taktiež ikonka pre prvok typu action by bola vhodná.
Prvok typu menu vytvorí vo všeobecnosti nový top level prvok v menu Eclipsu, avšak dá sa použiť aj na prepísanie už existujúceho top level prvku menu a práve to je možnosť, ktorú využijeme, pretože by sme chceli pridať action do už existujúceho menu („Search“). Toto existujúce menu bolo už vytvorené iným pluginom, ktorý si zadefinoval vlastné skupiny vnútri tohto menu. Na existenciu tohto menu a teda aj existenciu daných skupín sa nedá spoľahnúť, kedže pluginy sa môžu pridávať a odoberať podľa ľubosti, a tak musím vytvoriť presne také isté menu aké očakávam, že už existuje. Funkcie nášho pluginu ani pluginu pôvodne vytvárajúceho menu nebudú narušené, avšak musím poznamenať, že takéto jednanie nie je zrovna štandartné. Nie

je štandartné používať menu iných pluginov. Nemožno sa totiž spoliehať na to, že vyššie verzie iných pluginov budú používať menu v rovnakej forme (t.j. rovnaké skupiny) ako doteraz a tak je funkčnosť vyšších verzií iných pluginov ohrozená. Náš plugin je ale vyvíjaný špecificky pre verziu Eclipse číslo 3.2 a tak si to môžeme dovoliť.
Už sme si teda ukázali ako pripojiť funkčný kód do prostredia Eclipse. Avšak 2 súbory (manifest.mf, plugin.xml) ešte plugin z projektu nerobia, pretože je ešte treba, aby existovala trieda, cez ktorú bude Eclipse runtime (viď. 1.1) spúšťať a ukončovať plugin. Táto trieda musí byť potomkom triedy `org.eclipse.core.runtime.Plugin`. Dôležité sú metódy `start(BundleContext context)` a `stop(BundleContext context)`, ktoré sú volané na odštartovanie pluginu a jeho zastavenie. Ich preťaženie môže plniť rôzne inicializačné či ukončovacie funkcie. Je však treba mať na pamäti, že i keď Eclipse runtime spúšťa plugin až vtedy ak je to naozaj nutné, tak táto nutnosť môže byť spôsobená napríklad i požidavkou na ikonu daného pluginu a teda inicializácia prípadných veľkých dátových štruktúr daného pluginu v takomto prípade nie je nutná. Takéto prípady by sa mali pri inicializácií ošetriť. Náš plugin nerobí skoro žiadnu inicializáciu a tak metódy nemeníme.
V tejto chvíli už máme projekt vytvárajúci plugin popísaný. Najrýchlejšou formou implementácie je vygenerovanie „Hello,World!“ templatu pre pluginy a jeho následnou zmenou.

1.1 Podpora práce so štruktúrami jazyka Java

Skôr ako začneme pracovať s Java štruktúrami, mali by sme si povedať niečo o reprezentácií súborov a projektov ako o externom zdroji dát.

1.1.1 Správa zdrojov (Resource management)

Existuje určitá skupina všetkých súborov, ktoré sú z vývojového prostredia Eclipse viditeľné. Táto skupina sa nazýva **workspace** (implementácia interfacu org.eclipse.core.resources.IWorkspace). Vo workspace sa nachádzajú jednak java projekty, ale aj iné, interné súbory Eclipsu. Z pohľadu pluginu je

workspace vždy aktívny a prístupny[2]. Inštancia workspacu sa dá získať pomocou statickej metódy `IWorkspace org.eclipse.core.resources.ResourcesPlugin.getWorkspace()`. Vnútro workspacu je tvorené 3 typmi elementov:

- **Project**(org.eclipse.core.resources.IProject) - reprezentuje projekt, obsahuje priečinky a súbory
- **Folder**(org.eclipse.core.resources.IFolder) - reprezentuje priečinok, obsahuje súbory
- **File**(org.eclipse.core.resources.IFile) - reprezentuje súbor

Elementy sa nachádzajú v stromovej štruktúre, presne ako v súborovom systéme, v ktorom sa fyzicky nachádzajú. Koreňom tejto stromovej štruktúry workspacu je implementácia interfacu `org.eclipse.core.resources.IWorkspaceRoot`, ktorej inštanciu dostaneme pomocou metódy workspacu `IWorkspaceRoot getRoot()`. Koreň nám prezradí pomocou metódy `IProject[] getProjects()` všetky projekty. Kedže vnútorne prístupné projekty sú len tie projekty, ktoré sú práve otvorené[3], tak sa budeme ďalej v plugine zaoberať len tými otvorenými. Otvorenosť projektu overíme metódou `boolean IProject.isOpen()`. Sme teda už schopný vytvoriť zoznam otvorených projektov. Na tomto mieste by sme sa zastavili, pretože popísaný zoznam tvorí už základný odrazový mostík pre ďalšiu prácu s java štruktúrami.

1.1.2 Balík nástrojov na pracu s Javou

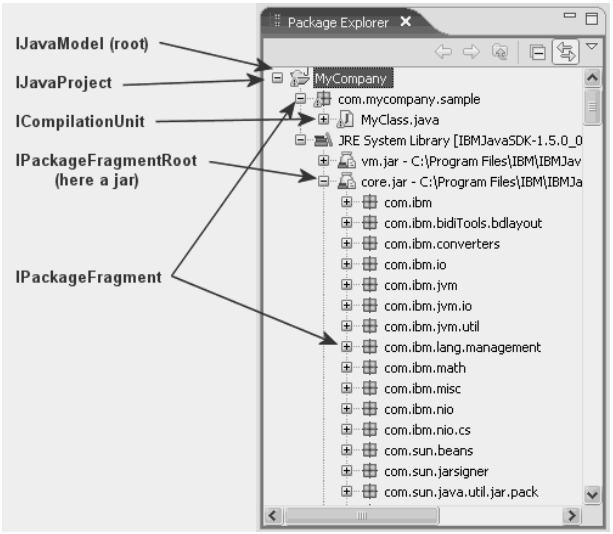
Dosiaľ sme hovorili o základnej štruktúre vývojového prostredia Eclipse. Poďme sa bližšie pozrieť na to ako sú štruktúry jazyka Java (projekty, triedy, premenné,...) sprístupnené, ako sa dajú meniť a čo všetko sa s nimi dá urobiť.
Vývojové prostredie Eclipse prichádza s veľkým balíkom nástrojov na prácu s Javou. Tento balík sa nazýva **Java development tooling** (**JDT**) a umožňuje používateľom písať, kompilovať, testovať, debugovať a upravovať programy písané v Jazyku Java. JDT nie je žiadnou špecialnou časťou architektúry Eclipsu. Môžeme ho vnímať ako akúsi skupinu pluginov, ktorých meno sa

[2]Pred vykonaním 1.požiadavky na workspace sa workspace automaticky vytvorí a zostáva aktívny až do skončenia fungovania (shut down) celého prostredia Eclipse

[3]otvorené v zmysle, že ich vývojové prostredie Eclipse má označené ako otvorené

Časť JDT	Význam
JDT Core	Obsahuje infrastruktúru pre kompiláciu a manipuláciu java kódu
JDT UI	Obsahuje UI triedy pracujúce s Java elementami,...
JDT Debug	Podpora spúšťania a debugovania programov napísaných v jazyku Java

Obr. 1.4: Hlavné časti balíka Java Development Tooling



Obr. 1.5: Typy Java štruktúr v projekte modelu JDT

začína predponou `org.eclipse.jdt`. Celé JDT by sa dalo rozdeliť na 3 celky (viď.tabuľka 1.4) Z pohľadu vyvíjaného pluginu nás bude zaujímať hlavne časť **JDT Core**(`org.eclipse.jdt.core`).V predošlej časti o správe zdrojov(1.1.1) sme si ukázali ako získať zoznam otvorených projektov(IProject). Pomocou statickej metódy `IJavaProject create(IProject project)` získame z interfacu `IProject`, predstavujúci zdroj súborov a priečinkov niekde uložených, interface `org.eclipse.jdt.core.IJavaProject`, predstavujúci java projekt obsahujúci package. Urobili sme len akúsi zmenu vnímania pojmu projekt (viď.obr 1.5). Pomocou metódy `IPackageFragmentRoot[] IJavaProject.getAllPackageFragmentRoots()` zistíme všetky prvky typu `org.eclipse.jdt.core.IPackageFragmentRoot`, ktoré predstavujú priečinok, JAR alebo Zip archív. Deti každého takéhoto prvku sú implementáciou interfacu `org.eclipse.jdt.core.IPackageFragment` a dostaneme ich

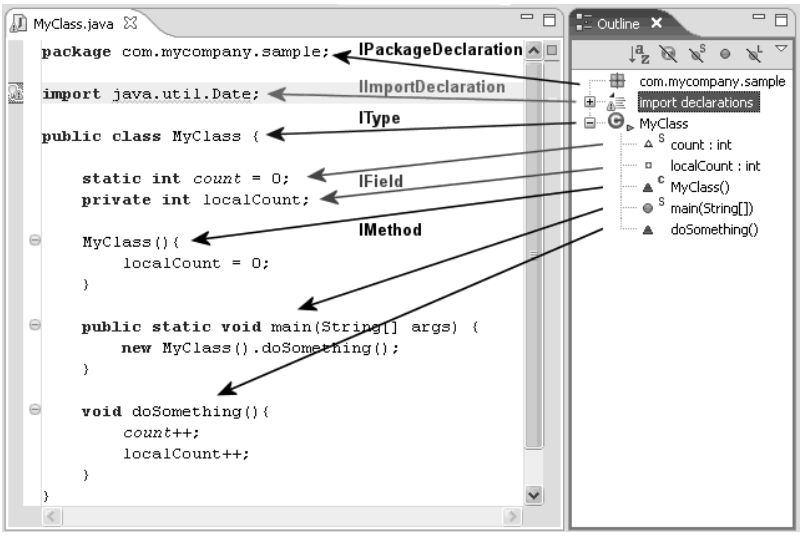
metódou `IJavaElement[] getChildren() throws JavaModelException`[4]. `IPackageFragment` predstavuje package alebo časť packagu. Jediný rozdiel medzi packagom a package fragmentom je ten, že package je zjednoteným všetkých package fragmentov, ktoré majú prefix mena rovnaký ako meno packagu. Package fragment môže byť 2 rôznych typov. Môže byť označený ako zdrojový alebo binárny. Keď je package fragment označený ako zdrojový, tak sa predpokladá výskyt zdrojových súborov (.java) vo vnútri package fragmentu. V prípade binárneho označenie zas binárne súbory (.class). To však neznamená, že v binárnom package fragmente nemôže byť zdrojový súbor alebo naopak. Znamená to len to, že v binárnom package fragmente sa nebudú hľadať zdrojové súbory a v zdrojovom package fragmente sa nebudú hladať binárne súbory. Kedže vytváraný plugin bude vytvárať a pracovať len so zdrojovými subormi, tak môžeme binárne package fragmenty odfiltrovať. Dokážeme to pomocou metódy `int IPackageFragment.getKind()`, ktorá zistí či package fragment je zdrojový (návratová hodnota je `IPackageFragmentRoot.K_SOURCE`) alebo binárny(návratová hodnota je `IPackageFragmentRoot.K_BINARY`).

Ďalšou metódou interfacu `IPackageFragment` sa už dostaneme na úroveň zdrojových súborov. Prvá časť pluginu týkajúca sa vytvárania templatov by si už s poskytnutými informáciami o JDT vystačila, pretože potrebuje vedieť len presné miesto (projekt a package) kam má automaticky vytvárané[5] templety uložiť. Avšak 2. časť pluginu potrebuje detekovať vnútornejšie java štruktúry a tak pokračujeme hlbšie.

Už spomínaná metóda, ktorá nás dostane na úroveň zdrojových súborov, je `ICompilationUnit[] getCompilationUnits()` a interface `org.eclipse.jdt.core.ICompilationUnit` predstavuje zdrojový súbor. Zdrojový súbor sa dá samozrejme tiež ďalej deliť na menšie časti (viď.obr 1.6). Deklarácia packagu (interface `org.eclipse.jdt.core.IPackageDeclaration`) a deklarácie importu (interface `org.eclipse.jdt.core.ImportDeclaration`) nás z pohľadu detekcie tried a ich vnútra vôbec nemusí trápiť. Metódou `IType[] ICompilationUnit.getTypes()` zistíme v súbore všetky top level java štruktúry, t.j. štruktúry, ktoré nie sú zaobalené v ďalších java štruktúrach v súbore. Interface `org.eclipse.jdt.core.IType` predstavuje takéto top level štruktúry. V našom plugine sa budeme zaoberať len 2 typmi top level štruktúr: **In-**

[4] `org.eclipse.jdt.core.JavaModelException` je Exception vytvorená pri zlyhaní Java modelu v JDT.

[5] Templaty sa nebudú vytvárať pomocou JDT (i keď by sa dali), ale pomocou vpisovania stringov do súboru



Obr. 1.6: Typy Java štruktúr v triede modelu JDT

terface a **Class**. Rozlíšime ich od ostatných top level štruktúr pomocou metódy `boolean IType.isClass()` a metódy `boolean IType.isInterface()`. Ďalšie vlastnosti vybranej top level štruktúry (a nie len tej top level) sa dajú zistiť pomocou metódy `int getFlags() throws JavaModelException`, ktorá vracia zakódovanú informáciu o skupine modifikátorov java elementu. Informácia sa dá rozkódovať pomocou statických metód triedy `org.eclipse.jdt.core.Flags`, ktoré dávajú odpoveď či zakódovanie obsahuje daný modifikátor alebo nie. Príkladom takýchto metód tejto triedy sú metódy `static boolean isPublic(int flags)`, `static boolean isSynchronized(int flags)`, `static boolean isInterface(int flags)`, `static boolean isAbstract(int flags)`, `static boolean isFinal(int flags)`, atď.... Bohužiaľ jedná sa len o modifikátory, ktoré sú v súbore napísané. Tak napríklad metóda deklarovaná v interface je ponímaná ako public, avšak modifikátor public sa aj tak nedetekuje, ak tam nebude fyzicky v súbore zapísané „public“.

Budeme aj potrebovať zistiť meno triedy(mená interfacov), ktoré daná trieda rozširuje(implementuje), či mená interfacov , ktoré sú predkom daného interfacu. S týmto problémom nám pomôžu metódy `String getSuperclassTypeSignature()` a `String[] getSuperInterfaceTypeSignatures()`. Prvá metóda zisťuje **signatúru** bezprostredného rodiča triedy. Druhá metóda v prí-

byte	„B“
char	„C“
double	„D“
float	„F“
int	„I“
long	„J“
short	„S“
void	„V“
boolean	„Z“

Obr. 1.7: Primitívne typy a ich reprezentácia ako signatúra.

pade triedy zisťuje signatúry implementovaných interfacov a v prípade interfacu zisťuje priamých interfacových predkov interfacu. Obidve metódy vracajú signatúry. Jedná sa o akési zakódovanie celej informácie o type[6] java prvku. Prácu so signatúrami umožňuje trieda `org.eclipse.jdt.core.Signature`, ktorá sa skladá zo statických metód. Nebudeme sa podrobne zaoberať signatúrami, pretože to nie je potrebné. Pre nás bude dôležitá len metóda `static String Signature.getSimpleName(String name)`, ktorá vráti meno typu, ktorý potrebujeme. V prípade primitívneho typu[7] táto metóda vracia len jednopísmenkové reťazce predstavujúce primitívne typy (viď.tabulku 1.7). Treba teda ešte prepísať tieto reťazce na vhodné primitívne typy.

Premenné deklarované vnútri triedy alebo interfacu predstavuje interface `org.eclipse.jdt.core.IField`. Zoznam všetkých takýchto premenných (v takom poradí ako sú zadeklarované) možno zistiť pomocou metódy `public IField[] IType.getFields() throws JavaModelException`. Z interfacu `IField` sa môžeme dozvedieť meno premennej (metóda `String getElementName()`) a signatúru jeho typu (metóda `String getTypeSignature()`).

Metódy deklarované vnútri triedy alebo interfacu predstavuje interface `org. eclipse.jdt.core.IMethod`. Zoznam všetkých takýchto premenných (v takom poradí ako sú zadeklarované) možno zistiť pomocou metódy `public IMethod[] IType.getMethods() throws JavaModelException`. Interface

[6]Vo všeobecnosti signatúra nie je zakódovanie len typu.

[7]Signatúru budeme používať aj neskôr a to na miestach, kde bude primitívny typ možné nájsť

`IMethod` obsahuje viacero užitočných metód:

- `String getElementName()` - zistí meno metódy
- `boolean isConstructor()` - zistí či je metóda konštruktor
- `String getReturnType()` - vráti signatúru návratového typu
- `int getNumberOfParameters()` - zistí počet parametrov metódy
- `String[] getParameterTypes()` - vráti signatúry typov parametrov (v tom poradí ako sú deklarované)

Vnútro metódy sa už nedá ďalej skúmať pomocou získavanie ďalších interfacov ako tomu bolo doteraz. Na zistenie vnútornej štruktúry tela metódy treba použiť parser, presnejšie **Abstract Syntax Tree Parser** (`org.eclipse.jdt.core.dom.ASTParser`). `ASTParser` patrí do packagu `org.eclipse.jdt.core.dom`, ktorého úlohou je modelovať zdrojový kód java programu ako štruktúrovaný dokument. Tento package obsahuje všetky potrebné triedy a prostriedky, ktoré budeme potrebovať na zistenie štruktúry kódu.

Najprv si vytvoríme inštanciu parsera pomocou statickej metódy `ASTParser newParser(int level)`, kde ako level nastavíme konštantu `jdt.core.dom.AST.JLS3`. Tá hovorí metóde, aby novo vytvorený parser používal API vyhovujúce 3.vydaniu špecifikácie jazyka Java (Java Language Specification, Third Edition), ktoré je schopné rozlíšiť kód java programu napísaného v Jave 1.6 (1.5,1.4,...). Keď už máme inštanciu parsera, treba jej povedať čo má parsovať. Túto úlohu plní funkcia `void ASTParser.setSource(char[] source)`, kde source je zdrojový kód java programu. Interface `IMethod` obsahuje metódu `public String getSource() throws JavaModelException`, pomocou ktorej získame požadovaný kód. Parseru treba ešte povedať na aký typ stromu má kód sparsovať (a teda aj aký typ kódu sme parseru priradili).Túto funkciu plní metóda `setKind(int kind)`. Kedže parsujeme telo metódy, tak ako parameter zoberieme konštantu `ASTParser.K_STATEMENTS`, ktorý zabezpečí vygenerovanie stromu ako postupnosti výrokov (statements). Samotné vygenerovanie stromu a vrátenie jeho koreňa zabezpečí metóda `ASTNode createAST(IProgressMonitor monitor)`, kde monitor je implementácia interfacu slúžiacemu na monitorovanie priebehu vykonávanej akcie. Kedže my nepotrebujeme monitorovať priebeh, tak parameter bude „null“.

ASTNode (`org.eclipse.jdt.core.dom.ASTNode`) je abstraktná trieda pre prvky stromu(AST). Kedže sme spôsob generovania stromu nastavili na `ASTParser.K_STATEMENTS`, tak získaný koreň je inštancia triedy **Block** (`org.eclipse.jdt.core.dom.Block`).Pomocou metódy `List Block.statements()` dostanem list obsahujúci len výroky (inštancie potomkov triedy `org.eclipse.jdt.core.dom.Statement`). Trieda `Statement` je abstraktná trieda zastrešujúca asi 21 typov výrokov. Nebudeme sa zaoberať všetkými typmi, vyberieme 1 typ ako ukážkový príklad. Popíšeme for cyklus (`org.eclipse.jdt.core.dom.ForStatement`). Štruktúra for cyklu je

for ([inicializácia]**;**[výraz (Expression)] **;** [aktualizácia (Update)] **)**
výrok(Statement).

Inicializácia je reprezentovaná ako list výrazov (metóda `List initializers()`), podmienka for cyklu ako výraz (metóda `Expression getExpression()`), aktualizácia (riadiacich) premenných ako list výrazov (metóda `List updaters()`) a telo for cyklu znova ako výrok (metóda `Statement getBody()`). Novozískané výroky môžem znova rozkladať na menšie časti. Rozkladom sme ale získali i výrazy (inštancie potomkov `org.eclipse.jdt.core.dom.Expression`). Trieda `Expression` je podobne ako trieda `Statement` abstraktná trieda a to pre všetky typy výrazov, ktorých je 26. Nebudeme ich všetky rozoberať, ale môžeme spomenúť, že najjednoduchší z nich[8] predstavuje 1 písmenko (trieda `org.eclipse.jdt.core.dom.CharacterLiteral`) a najzložitejší predstavuje volanie metódy (trieda `org.eclipse.jdt.core.dom.MethodInvocation`).

V tejto chvíli sme už schopný rozanalyzovať zdrojové súbory do najmenších detailov.

1.2 Visuálne komponenty

Už sme si spomínali, že o grafickú časť vývojového prostredia Eclipse sa stará **Workbench UI** (viď.tab.1.1). Čo sme si ale nespomenuli je, že Eclipse nepoužíva grafické komponenty ponúkané distribúciou jazyka Java, ale používa vlastné grafické komponenty. Grafické komponenty vývojového prostredia Eclipse by sa dali rozdeliť do dvoch skupín: `JFace` a `SWT`. JFace sú gra-

[8] Najjednoduchší, ktorý sa nám zdal.

fické komponenty s predponou `org.eclipse.jface` a SWT (Standart widget toolkit) s predponou `org.eclipse.swt`. Na prvý pohľad by sa mohlo zdať, že sú to 2 balíky grafických komponentov, ktoré sú relatívne ekvivalentné a môžeme ich používať podľa ľubosti. V skutočnosti je však JFace určitou nadstavbou SWT. SWT predstavuje skupinu najzákladnejších grafických komponentov ako napríklad Button(`org.eclipse.swt.widgets.Button`) či Label(`org.eclipse.swt.widgets.Label`). SWT definuje API, ktoré sa ľahko prenáša a je úzko späté s konkrétným natívnym GUI[9] operačného systému, na ktorom beží platforma Eclipse. SWT sa snaží používať natívne komponenty daného OS, kdekoľvek je to možné. Tým sa zabezpečuje ľahká adaptovateľnosť na grafické zmeny operačného systému a zároveň sa zachováva jednotný grafický model pre grafické komponenty Eclipse. JFace slúží ako nadstavba SWT v tom zmysle, že jej grafické komponenty sú zložené z komponentov SWT do zložitejších grafických komponentov, ktoré sú dosť užitočné pri rozširovaní vývojového prostredia.

1.2.1 Mechanizmus zreťazenia komponentov a bity štýlu

Komponenty v Eclipse sa nepridávajú visuálnym rodičom tak ako u visuálnych komponentov jazyku JAVA, t.j. cez metódu `add(Component comp)` triedy `Container`. Náväznosť na rodiča sa vytvára priamo konštruktorom visuálneho komponentu, ktorý si priamo ako 1.parameter vyžaduje visuálneho rodiča. Z toho samozrejme vyplýva, že v ako poradí sú komponenty vytvárané, v takom sú aj zobrazované. Ďalším dôsledkom je, že vizuálne komponenty, ktorých vnútro má byť zapĺňané ďalšími komponentami, musia ponúkať určitú metódu na preťaženie[10], ktorá ako parameter má komponent, ktorý bude pridávaným komponentom slúžiť ako visuálny predok. Takáto metóda sa väčšinou nazýva `createControl(Composite parent)`(ale aj `createDialogArea(Composite parent)` a pod.).
Niektoré vlastnosti vizuálných komponentov musia byť známe pri vzniku týchto komponentov a počas života komponentu sa nemenia. Pre takéto vlastnosti boli zavedené **bity štýlu** (style bits). Jedná sa o statické konštanty implementované triedou `org.eclipse.swt.SWT`, ktoré predstavujú jednotlivé vizuálne vlastnosti. Keď chceme komponentu nastaviť takéto vlastnosti, tak

[9] Grafické rozhranie pre používatela.
[10] Preťaženie sa deje v potomkovi tohto komponentu.

Layout	Stručný popis
FillLayout	(`org.eclipse.swt.layout.FillLayout`) Ukladá komponenty do 1 stĺpca či 1 riadku, pričom núti komponenty, aby mali rovnakú veľkosť.
FormLayout	(`org.eclipse.swt.layout.FormLayout`) Ukladá komponenty pomocou definovania triedy FormAttachment (`org.eclipse.swt.layout.FormAttachment`) pre každý komponent. Trieda FormAttachment určuje pozíciu, šírku aj výšku komponentu.
GridLayout	(`org.eclipse.swt.layout.FormLayout`) Ukladá komponenty podľa mriežky. Parametre každého komponentu voči mriežke sa nastavujú priradením objektu typu GridData (`org.eclipse.swt.layout.GridData`) ku objektu.
RowLayout	(`org.eclipse.swt.layout.RowLayout`) Ukladá komponenty podobne ako FillLayout len s tým rozdielom, že nenúti komponenty mať rovnakú veľkosť.

Tabuľka 1.2: Eclipse Layouts

pospájame jednotlivé želané konštanty operáciou logický OR („|“) a výsledný int použijeme ako parameter pre konštruktor daného komponentu. Každému komponentu sa samozrejme nedajú nastaviť všetky visuálne vlastnosti, preto daný komponent bude reagovať len na tie bity, ktoré zodpovedajú jeho vlastnostiam. V dokumentácií každého komponentu sú bity štýlu, na ktoré reaguje, uvedené. V prípade, že nechceme aktivovať ani jednu takúto vlastnosť komponentu, tak použijem len konštantu `org.eclipse.swt.SWT.NULL`.

1.2.2 Layouts

Layout alebo rozloženie komponentov na ploche iného (rodičovského) vizuálneho komponentu, je veľmi dôležitá vlastnosť vizuálneho systému. Pomocou neho sa vytvára prehľadné, ľahko použiteľné používateľské rozhranie[11]. Vývojové prostredie Eclipse si s vlastnými vizuálnymi komponentami definuje aj vlastné typy layoutov. Sú štyri a mapuje ich tabuľka 1.2. FillLayout a RowLayout sú nevhodné layouty pre naše účely, kedže vytvárajú len 1 stĺpec

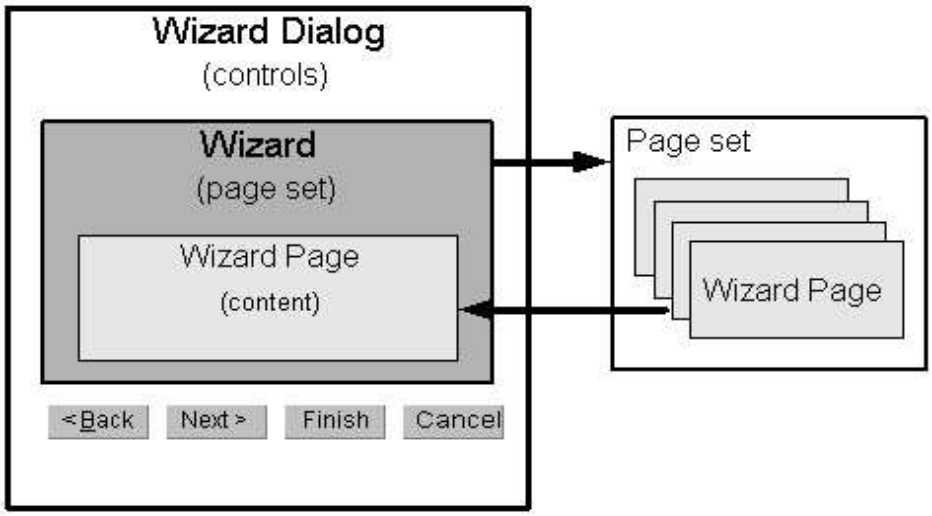
[11] „user friendly“ interface

Meno attribútu	**význam**
horizontalAlignment	horizontálne zarovnanie vnútri bunky
verticalAlignment	vertikálne zarovnanie vnútri bunky
horizontalIndent	počet pixelov, o ktoré posuniem komponent doprava
horizontalSpan	počet zabratých buniek do šírky
verticalSpan	počet zabratých buniek do výšky
grabExcessHorizontalSpace	povolenie na resize komponentu do šírky pri resize okna
grabExcessVerticalSpace	povolenie na resize komponentu do výšky pri resize okna
widthHint	požadovaná šírka komponentu
heightHint	požadovaná výška komponentu

Tabuľka 1.3: Nastavenia zaobalené triedou GridData

alebo riadok. Ako univerzálny layout, ktorý sa bude v našom plugine používať, sme zvolili GridLayout. FormLayout by sa síce tiež dal použiť avšak tento layout sa väčšinou používa[12] v prípadoch, keď pozícia 1 komponentu závisí od pozície niakeho iného konkrétneho komponentu. GridLayout naproti tomu porovnáva všetky komponenty voči jednotnej mriežke a tým sa dá v rozložení komponentov lepšie[13] zorientovať.
V rozložení GridLayout sa nastavujú vizuálne vlastnosti komponentov pomocou triedy GridData. Táto trida sa následne pripojí k vizuálnemu komponentu v tomto rozložení pomocou metódy `setLayoutData(Object layoutData)`. Možnosti nastavenia triedy GridData ukazuje a vysvetluje tabuľka 1.3. Tvorbu triedy GridData sme si pri implementácií uľahčili statickou metódou `GridData designpatternplugin.wizards.WizardGenerator.getGrid-DataObject(int,int,int,int,int,boolean,boolean,int,int)`.
Okrem už spomínaného informačného zdroja [1] nám veľmi osvetlil sitáciu i článok o Layouts [4].

[12]Dokumentácia a vzorové príklady o tom svedčili
[13]Iným programátorom to môže vyhovovať inak.



Obr. 1.8: Vrstvy wizardu.

1.2.3 Popis použitých vizuálnych komponentov

Ďalej sa budeme zaoberať grafickými komponentami, ktoré boli reálne v našom plugine využité. Prvým, o ktorom budeme hovoriť je wizard (`org.eclipse.jface.wizard.Wizard`), resp. o jeho potomkovi, ktorý vytvoríme.

Wizard (`org.eclipse.jface.wizard.Wizard`)
Úlohou wizardu je vypýtať si od používateľa relatívne vaľa informácií, pričom otázky môže zoskupiť na jednotlivé stránky wizardu a stránky usporiadať do niakej logickej postupnosti. Obrazok 1.8 prezrádza, že Wizard mechanizmus sa skladá z vrstiev. Vrchnú vrstvu tvorí `org.eclipse.jface.wizard.WizardDialog`, ktorý má na starosti spravovanie wizardov (nižšia vrstva) a tlačidiel (Next, Back, Cancel, ...). Nižšia vrstva je implementáciou interfacu `org.eclipse.jface.wizard.IWizard` a riadi celkové chovanie wizardu. Táto vrstva sa väčšinou implementuje ako rozšírenie triedy `org.eclipse.jface.wizard.Wizard`, pretože táto trieda implementuje `org.eclipse.jface.wizard.IWizard` a stará sa aj o iné implementačné detaily tak, aby wizard mal štandartné[14] chovanie. Takéto rozšírenie musí spĺnať 2 základne funkcie: tvorbu,

[14]t.j. také aké by sme od wizardu očakávali

pridávanie stránok (nižšia vrstva) a funkcionalitu spojenú s uspešným skončením wizardu. Najnižšou vrstvou je wizard page (implementácia interfacu `org.eclipse.jface.wizard.IWizardPage`). Väčšinou sa používa rozšírenie triedy `org.eclipse.jface.wizard.WizardPage`. Úlohou tejto vrstvy je vytvorenie SWT komponentov, ktoré budú tvoriť danú stranu wizardu, a reagovať na ich udalosti (events), aby dokázala táto vrstva povedať, či používateľ poskytol dostatočnú informáciu, aby mohol wizard prejsť na ďalšiu stránku. Kedže náš plugin sa pripája už na existujúci extension point, tak Eclipse má už najvrchnejšiu časť vytvorenú a nám ponúkne len pridanie vlastnej implementácie interfacu `org.eclipse.jface.wizard.IWizard` do 2.vrstvy. Výhodou takejto situácie je, že nemusíme implementovať 1.vrstvu, avšak sme ochudobnení o metody 1.vrstvy, kedže ju neovládame.
Rozšírenie triedy `org.eclipse.jface.wizard.Wizard` nám dáva možnosť (znovu)implementovať užitočné metódy ako napríklad:

- `performCancel()` - metóda volaná pri neúspešnom skončení wizardu
- `performFinish()` - metóda volaná pri úspešnom skončení wizardu
- `dispose()` - metóda, ktorá by mala uvoľnovať(cez dispose) všetke stránky wizardu
- `addPages()` - metóda, ktorá naplní wizard stránkami (pomocou `org.eclipse.jface.wizard.Wizard.addPage(IWizardPage page)`)

V rozšírení triedy `org.eclipse.jface.wizard.WizardPage`, ktoré využijeme ako stranu vo wizarde, implementujeme metódu `void createControl (Composite parent)`, ktorou naplníme stránku vizuálnými komponentami, a možno ešte metódu `boolean isPageComplete()` hovoriacu o úplnosti vyplnenia povinných údajov stránky.

Dialog (`org.eclipse.jface.dialogs.Dialog`)
Dialog nie je vôbec zložitý komponent. Stačí ho rozšíriť a reimplementovať metódu `Control createDialogArea (Composite parent)` , ktorou pridám iné vizualizačné komponenty na plochu dialogu. Následne stačí vytvoriť inštanciu takéhoto rozšírenia a zavolať metódu `int open()`, ktorá spustí samotný dialog. Po ukončení dialogu sa návratová hodnota vráti ako návratová hodnota zavolanej metódy. Návratová hodnota môže byť buď `org.eclipse. jface.window.Window.OK` alebo `org.eclipse.jface.window. Window.CANCEL`.

Bity štýlu sa nastavujú v konštruktore cez metódu `setShellStyle(int newShellStyle)`. V našom jedinom vytvorenom Dialogu sme použili štýly `SWT.TITLE` (zobrazenie titulku), `SWT.RESIZE` (povolenie zmeny veľkosti dialogu), `SWT.APPLICATION_MODAL` (modálne správanie dialogu) a `SWT.CLOSE` (zobrazenie tlačidla zatvorenia okna v pravom hornom rohu).

TreeViewer (`org.eclipse.jface.viewers.TreeViewer`)
Úlohou TreeViewera je graficky reprezentovať strom. K TreeVieweru musia byť po vytvorení pripojené 2 životne dôležité objekty: implementácia interfacu `org.eclipse.jface.viewers.ITreeContentProvider` a implementácia interfacu `org.eclipse.jface.viewers.IBaseLabelProvider`. Prvý interface sa dá priamo implementovať a jeho úlohou je poskytnúť TreeVieweru obsah stromu, ktorý má zobrazovať. Druhý interface by nemal byť priamo implementovaný a tak použijem rozšírenie triedy `org.eclipse.jface.viewers.LabelProvider`. Jeho úlohou je poskytnúť ikonku každému prvku stromu. Prvá implementácia sa pripája pomocou metódy `setContentProvider(IContentProvider provider)` a druhá implementácia pomocou metódy `setLabelProvider(IBaseLabelProvider labelProvider)`. Okrem týchto 2 pripojených objektov sa dá pripojit ku TreeVieweru ešte aj objekt, ktorý by plnil sortovaciu alebo filtrovaciu funkciu na prvkoch stromu.
Pozrime sa na implementáciu interfacu `ITreeContentProvider` bližšie. Treba implementovať nielen metódy v interfaci priamo definované, ale aj metódy predkov tohto interfacu. Konkrétne sa jedná o metódy:

- `Object[] getChildren(Object parentElement)` - vráti deti daného objektu v strome
- `Object getParent(Object element)` - vráti rodiča daného objektu v strome
- `boolean hasChildren(Object element)` - zistí či daný objektu v strome má deti
- `Object[] getElements(Object inputElement)` - vráti objekty stromu, ktoré sa majú zobraziť, keď vstupný objekt stromu je `inputElement`
- `void dispose()` - uvoľnenie dodatočne vytvorených zdrojov
- `void inputChanged(Viewer viewer,Object oldInput,Object newInput)` - reakcia na zmenu vstupného objektu

Z metód vyplýva, že strom sa skladá z inštancií (potomkov) triedy `java.lang.Object`. Mohli by sme vytvoriť rozšírenie triedy `java.lang.Object` a používať ho ako element Stromu.
Pri implementácií rozšírenia triedy `org.eclipse.jface.viewers.LabelProvider` sa ponúka hneď niekoľko metód, ktoré sa ponúkajú[15] na preťaženie. Spomenieme si len 2 najpoctatnejšie:

- `Image getImage(Object element)` - vracia ikonu k danému prvku stromu
- `String getText(Object element)` - vracia text k danému prvku stromu, ktorý ho bude vizuálne reprezentovať.

Dôležité pre TreeViewer je aj správne inicializačné nastavenie vstupného objektu, ktoré môže v implementácií interfacu `ITreeContentProvider` spôsobiť inicializačný proces.

Button (`org.eclipse.swt.widgets.Button`)
Komponent predstavujúci tlačidlo. Užitočné[16] štýly:

- `SWT.CHECK`[17] - mení tlačidlo na odškrtávací box s textom
- `SWT.RADIO` - mení tlačidlo na vyplniteľný krúžok s textom

Text tlačidla sa dá zistiť a nastaviť pomocou metód `String getText()` a `setText(String string)`. Stav tlačidla (pri štýloch `SWT.CHECK`,`SWT.RADIO`) sa dá zistiť a nastaviť metódami `boolean getSelection()` a `setSelection (boolean selected)`. Button podporuje pridanie listenera reagujúceho na označenie komponentu (interface `org.eclipse.swt.events.SelectionListener`) cez metódu `addSelectionListener(SelectionListener listener)`.
Listener obsahuje 2 metódy:

- `widgetSelected(SelectionEvent e)` - metóda je volaná v prípade označenia komponentu
- `widgetDefaultSelected(SelectionEvent e)` - metóda je volaná v prípade „default“ označenia komponentu (jedná sa o prípady nepriameho označenie komponentu (napr. odznačenie 1 komponetu spôsobí označenie iného))

[15] Dokumentácia hovorí o vhodnosti ich reimplementácie.
[16] t.j. použité v plugine alebo v praxi častejšie využívané
[17] Skrátený tvar pre `org.eclipse.swt.SWT.CHECK`

Combo (`org.eclipse.swt.widgets.Combo`)
Predstavuje textový riadok s meniteľným textom pomocou editácie, ale aj pomocou zoznamu textov. Užitočné štýly:

- `SWT.DROP_DOWN` - zoznam textov na výber je na báze drop-down menu (vylučuje štýl `SWT.SIMPLE`)
- `SWT.READ_ONLY` - zmena textu iba pomocou zoznamu textov
- `SWT.SIMPLE` - zmena textu pomocou zoznam textov graficky implementovaná tlačidlami so šipkami (vylučuje štýl `SWT.DROP_DOWN`)

Combo sa chová podobne ako implementácia interfacu `java.util.List<String>`, môžu sa pridávať Stringy do zoznamu (`add(String string)`), vyberať (`remove(String string)`, `remove(int index)`) či zisťovať veľkosť zoznamu (`int getItemCount()`). Navyše existuje 1 označený String v zozname, ktorý sa dá zistiť (`int getSelectionIndex()`), ale aj nastaviť (`select(int index)`).

Composite (`org.eclipse.swt.widgets.Composite`)
Hlavná podstata tohto komponentu spočíva v tom, že dokáže obsahovať iné komponenty. Užitočné štýly obsahuje až predok `org.eclipse.swt.widgets.Scrollable`:

- `SWT.H_SCROLL` - umožnuje horizontálne rolovanie
- `SWT.V_SCROLL` - umožnuje vertikálne rolovanie

Vizuálne komponenty, ktoré daný komponent obsahuje, možno zistiť pomocou metódy `Control[] getChildren()` a pomocou metódy `setLayout(Layout layout)` nastaviť layout pre ne. Zaujimavosťou je, že trieda neimplementuje[18] žiadny mechanizmus na odstraňovanie už pridaných vizuálnych komponentov.

Label (`org.eclipse.swt.widgets.Label`)
Predstavuje Text alebo obrazok, ktorý sa nedá označiť. Label má síce rôzne štýly, ale ani jeden sme nepoužili v plugine. Text či obrázok sa nastaví komponentu pomocou metód `setText(String string)` a `setImage(Image image)`. Naopak, pomocou metód `String getText()` a `Image getImage()` sa zistí text či obrazok ku komponentu priradený.

[18] t.j. žiadny sme podľa dokumentácie nenašli

Layout (`org.eclipse.swt.widgets.Layout`)
Layout je abstraktná trieda, ktorej rozšírenia sú už spomínané 4 typy Layoutu (viď.tabulku 1.2).

List (`org.eclipse.swt.widgets.List`)
Predstavuje vizuálny zoznam textových položiek, v ktorom sa dajú jednotlivé položky označiť. Užitočné štýly:

- `SWT.SINGLE` - označiť sa môže iba 1 položka v zozname,vylučuje `SWT.MULTI`
- `SWT.MULTI` - označiť sa môže aj viac položiek v zozname,vylučuje `SWT.SINGLE`

List sa správa velmi podobne ako komponent **Combo**, pretože implementuje rovnaké metódy podobné interfacu `java.util.List<String>`. Navyše podporuje aj označovanie viacerých položiek pomocu metódy `select(int[] indices)` či zisťovanie zoznamu označených položiek (metóda `int[] getSelectionIndices()`).

Spinner (`org.eclipse.swt.widgets.Spinner`)
Predstavuje vizuálny komponent, ktorý umožnuje nastavovať číselnú hodnotu. Žiadne štýly tohto komponentu neboli v plugine použité.
Komponentu sa dá nastaviť maximálna (metóda `setMaximum(int value)`) a minimálna hodnota (metóda `setMinimum(int value)`). Komponent obsahuje vizuálne tlačítka, ktorými sa dokáže zvyšovať alebo znižovať hodnota. Veľkost zmeny hodnoty pri použití týchto tlačidiel sa dá nastaviť metódou `setIncrement(int value)` (`setPageIncrement(int value)`). Samotná hodnota sa dá nastaviť a prečítať pomocou metód `setSelection(int value)`, `int getSelection()`. V prípade nastavovania desatinných čísiel sa nastaví počet desatinných miest pomocou metódy `setDigits(int value)` a návratová hodnota metódy `int getSelection()` je správne interpretovaná až po posune desatinej čiarky. Tak napríklad, ak metóda vrati číslo 314 a počet desatinných miest je 2, tak správna interpretácia údaju je 3.14 .
Komponent okrem možnosti pripojenia listenera `SelectionListener` (`org.eclipse.swt.events.SelectionListener`) ponúka možnosť pripojenia aj listenera `ModifyListener` (`org.eclipse.swt.events.ModifyListener`), ktorý obsahuje metódu `modifyText(ModifyEvent e)` volanú v prípade zmeny textového(číselného) obsahu komponentu.

Table (`org.eclipse.swt.widgets.Table`)
Predstavuje vizuálny zoznam textových alebo obrázkových položiek, v ktorom sa dajú jednotlivé položky označiť. Svojou funkciou sa podobá komponentu `List`. Užitočné štýly tohto komponentu sú rovnaké ako u komponentu `List`. Od `Listu` sa ale predsa len líši a to tým, že dokáže zobrazovať aj obrázky a dá sa v tomto komponente vytvoriť aj viacej stĺpcov. Pridanie stĺpca sa vykoná vytvorením inštancie triedy `org.eclipse.swt.widgets.TableColumn`, ktorej konštruktor si ako parameter vyžiada komponent `Table`. Trieda reprezentujúca stĺpec má viacero užitočných metód:

- `setMoveable(boolean moveable)` - nastavuje možnosť zmeny pozície stĺpca medzi ostatnými stĺpcami
- `setResizable(boolean resizable)` - nastavuje možnosť zmeny šírky
- `setText(String string)` - nastavuje nadpis pre stĺpec
- `setWidth(int width)` - nastavuje šírku stĺpca

Po nastavení stĺpcov komponentu `Table` pridáme položky. Kedže položky musia byť viacstĺpcové a môžu byť aj obrázkové, tak položkou nemôže byť jednoduchý `String` ani pole `String`ov. Položkami budú inštancie triedy `org.eclipse.swt.widgets.TableItem`. Pripájajú sa ku komponentu `Table` rovnako ako sa pripájal nový stĺpec. Hlavnou funkcionalitou triedy predstavujúcu položku je nastavenie textu/obrázku v danom stĺpci (metódy `setImage(int index, Image image)` a `setText(int index, String string)`) a ich zistenie v danom stĺpci (`Image getImage(int index)`, `String getText(int index)`).

Text (`org.eclipse.swt.widgets.Text`)
Predstavuje textove pole slúžiace na editáciu textu. Užitočné štýly:

- `SWT.SINGLE` - nastavuje 1 riadkové editovanie textu, vylučuje `SWT.MULTI`
- `SWT.MULTI` - nastavuje viacriadkové editovanie textu, vylučuje `SWT.SINGLE`
- `SWT.READ_ONLY` - zablokovanie editovania

Medzi najzákladnejšie funkcie patrí nastavovanie textu (metóda `setText(String string)`) a zisťovanie textu z komponentu (metóda `String getText()`).

Na zisťovanie zmeny textu existuje možnosť pripojenia implementacie interfacu `org.eclipse.swt.events.ModifyListener`[19].

[19]Podrobnejšie popísaný pri komponente `Spinner`

Kapitola 2

Model Návrhového vzoru

Kedže plugin by mal podporavať prácu s návrhovými vzormi, je nutné, aby plugin poznal štruktúru návrhových vzorov. Mohli by sme každý jeden návrhový vzor „napevno zadrôtovať“[1] do kódu programu, ale znamenalo by to veľmi prácne pridávanie nových patternov. Namiesto toho vytvoríme dátový model, pomocou ktorého sa budú môcť zadefinovať všetky uvažované návrhové vzory, a v kóde programu budeme pracovať jednotne s týmto modelom.

2.1 Xml ako prostriedok na ukladanie informácií

Návrhový vzor, ktorý si zadefinujeme pomocou dátového modelu, musíme byť schopný uložiť do súbora, aby sme mohli návrhové vzory používať dynamicky. Súbor s návrhovým vzorom bude súčasťou pluginu. Problémom je ako tento súbor po nainštalovaní pluginu nájsť. Odpoveďou je statická metóda `java.net.URL org.eclipse.core.runtime.FileLocator.find(Bundle bundle, IPath path, Map override)`, ktorá si ako 1. parameter zoberie bundle nášho pluginu a ako 2. parameter cestu k súboru (relatívna voči inštalačnému priečinku pluginu). Tretí parameter nastavíme na `null` (defaultné nastavenie na substitúciu argumentových stringov sa použije). Návratová

[1] t.j. rozhodovať sa vnútri kódu podľa toho, s ktorým návrhovým vzorom práve pracujeme

trieda `URL` otvorý súbor pomocou metódy `InputStream openStream()`. Takéto riešenie som našiel pomocou zdroja [3].

Dátovy model zahrňujúci vlastnosti návrhových vzorov, nie je taký jednoduchý, aby sme mohli použiť niaky vlastný spôsob ukladania informácií do súbora. Jedná sa totižto o skupinu objektov s určitými vlastnosťami, ktoré sú poprepájané medzi sebou určitými vzťahmi. Pri modelu s pár objektami by vlastný spôsob ukladania informácií mohol spôsobiť ťažkú pochopiteľnosť návrhové vzoru pre ľudí, ktorý by sa tento model snažili pochopiť. Siahneme preto po už existujúcich spôsoboch ukladania informácie. Ako prvá sa ponúka možnosť **XML** (Extensible Markup Language). V XML sa ľahko nadefinujú objekty ako tagy[2], ktoré budú mať vlastnosti a prepojenia k iným objektom zadefinované vo vnútorných tagoch. Druhou uvažovanou možnosťou by bolo **UML** (Unified Modeling Language), ktoré sa zaoberá viac vnútornými vzťahmi objektov. Nakoniec sme sa však rozhodli pre XML a to nielen kvôli jednoduchosti, ale hlavne kvôli podpore XML v jazyku Java. Umožní nám to zbytočne nepripájať cudzie nástrojové knižnice, ale použiť už vhodne zabudovanú súčasť jazyka Java.

Existujú 2 možné programátorske prístupy ako sa dostať k informáciam zakódovaných v XML. Prvým je **SAX**(Simple API for XML) a druhým **DOM**(Document Object Model). SAX je skupina tried a metód, ktorá pripomína parser XML kódu. Pristupuje k informáciam sekvenčne ako parser, t.j. nie je možné cez SAX zistiť napríklad to čo bolo v XML súbore parsované 5 prvkov dozadu. SAX taktiež nepozná, v ktorom leveli vnorenia tagov sa práve nachádza. Z týchto dóvodov nie je možné SAX v našom plugine dobre využiť. Naproti tomu DOM má úplne inú filozofiu. Vytvorí z celého XML súboru stromovú štruktúru, ktorá predošlé 2 nedostatky celkom odstraňuje. Jedinou nevýhodou DOM je, že drží celú informáciu XML súbora v pamäti, čo nie je vždy želané. Avšak veľkosť návrhového vzoru zakódovaného v XML súbore je relatívne malá a tak pokojne môžeme použiť prístup DOM. Pre veľké XML súbory by bol vhodnejší SAX prístup.
Informácie o SAX a DOM ako aj o ďalšom použití DOM som čerpal jednak s technickej dokumentácie jazyka Java [5], ale aj z knihy Java `&` XML [6].

Najprv si vytvoríme inštanciu DOMparsera (`com.sun.org.apache.xer-`

[2]V skutočnosti je základným kameňom XML markup, ale v HTML sa markups nazývajú tagy a my skutočne nebudeme naplno využívať výhody markupu, ale len vlastnosti aké poznáme z tagov

ces.internal.parsers.DOMParser). Potom inštanciu triedy org.xml.sax.InputSource, kde ako parameter konštruktora použijeme inštanciu triedy java.io.InputStream (otvorený xml súbor). Trieda InputSource slúži na zaobalenie informácií o zdroji do 1 objektu pre potrebu parsera. Potom nechám obsah triedy InputSource sparsovať parserom metódou void DOMParser.parse(InputSource input) a výsledok ako inštanciu interfacu org.w3c.dom.Document dostanem pomocou metódy Document DOMparser.getDocument(). Trieda Document vlastní vygenerovaný strom. Nachádza sa 1 level nad koreňom tohto stromu.

Z interfacu Document dostanem metódou Element getDocumentElement() koreň DOM stromu. Všetky uzly tohto stromu implementujú interface org.w3c.dom.Node.
Metóda String Node.getNodeName() zistí meno tagu.
Metóda NamedNodeMap Node.getAttributes() zistí mapu atribútov tagu. Jednotlivé attribúty sa dajú pomocou interfacu org.w3c.dom.NamedNodeMap metódou Node item(int index) zistiť postupne alebo jednoducho podľa mena vyhladať metódou Node getNamedItem(String name).
Metóda NodeList Node.getChildNodes zistí všetky uzly stromu priamo pod daným uzlom a vráti výsledok vo forme zoznamu (interface org.w3c.dom.NodeList). Zoznam je celkom intuitívne spravený, pretože metóda Node item (int index) vracia prvok s poradovým číslom index a metóda int getLength() vracia veľkost zoznamu. Ak tag pod sebou v hierarchií už nemá ďalšie tagy ale obsahuje text, tak v DOM strome obsahuje ešte jeden tag. Tento tag je typu Node.TEXT_NODE (node.getNodeType()==Node.TEXT_NODE) a samotný text je hodnota tohto tagu (String getNodeValue()).

Viac metód nepotrebujeme vedieť, pretože model bude založený len na do seba vnorených tagoch, ich atribútoch a textu vnútri tagu.

2.2 Súborový Dátový model návrhového vzoru

Najprv uvedieme schéma súborového dátového XML modelu návrhového vzoru a potom sa bližšie pozrieme na jednolivé tagy. Upozorňujem, že niektoré konštrukcie či pravidlá sa možno budú zdať na prvý (a asi aj na druhý) pohľad nepochopiteľne reštriktívne, avšak pre jednoduchšiu implementáciu (t.j. neošetrovanie prípadov, ktoré ani nevyužijem) je to nutné.

Schéma[3]:

```
<design_pattern name="...">(2)
.<helpLabel name="..." is_list="...">(27)
..<defaultValue>(28)
...value
.<clientobjects>(3)
.<nonclientobjects>(4)
..<interface name="..." parent="...">(25)
...<interfaceVariable name="...">(26)
...<method name="..." is_constructor="..." label="...">(13)
....<return_type>(14)
.....returnType
....<list_of_parameters>(15)
.....<parameter type="...">(16)
......parameterName
..<objects name="..." parent="..." abstract="..." label="...">(6)
..<object name="..." parent="..." abstract="..." label="...">(5)
...<implementInterface>(7)
....interfaceName
...<is_high_level_list_to>(8)
....ObjectName
...<reference name="..." reffers_to="..." is_list="..."
             assign_from="...">(9)
....<object_with_access>(11)
.....ObjectName
....<client_access/>(12)
...<unspecified_variable name="...">(10)
....<object_with_access>(11)
.....ObjectName
....<client_access/>(12)
...<template>(24)
....templateText;
...<method name="..." is_constructor="..." label="...">(13)
....<return_type>(14)
```

[3]Schéma obsahuje tagy bez ich ukončovacej časti (2.tagu, ak je tag párový). Tag v schéme obsahuje všetky tagy pod ním v riadkoch, ktoré sú zarovnané viac doprava ako tag (t.j. majú pred sebou viac bodiek). Riadok bez tagu (čistý text) znázorňuje textový obsah tagu nad ním. Číslo v zátvorke za tagom je odkaz na tag v zozname tagov.

```
.....returnType
....<list_of_parameters>(15)
.....<parameter type="...">(16)
......parameterName
....<demands>(17)
.....<template>(24)
......templateText
.....<call_method use_reference="..." method_name="..."
                  is_return_call="..." is_list_command="">(18)
......<list_of_parameters_for_call>(19)
.......<use_reference>(20)
........referenceName
.......<use_parameter>(21)
........parameterName
.......<this/>(22)
.....<added_behaviour/>(23)
```

Zoznam tagov s ich významom a umiestnením:

1. `<?xml version="1.0" encoding="UTF-8"?>`
   `Význam:` Začiatočny tag XML súbora.

2. `<design_pattern name="..."></design_pattern:>`
   `Umiestnenie:` 2.tag v subore, root tag.
   `Význam:` V sebe definuje celý navrhový vzor. Parameter *name* definuje názov návrhového vzoru.

3. `<clientobjects></clientobjects>`
   `Umiestnenie:` Vnútri[4] tagu `design_pattern` (2).
   `Význam:` Obsahuje objekty,ktoré klient nielen vytvára,ale aj s nimi pracuje ,prípadne ich len inicializuje.

4. `<nonclientobjects></nonclientobjects>`
   `Umiestnenie:` Vnútri tagu `design_pattern`(2).
   `Význam:` Obsahuje objekty, ktoré client síce vytvára(alebo aj nie),avšak čo je dôležitejšie, nepracuje s nimi ďalej. Patria sem všetky objekty, ktoré nepatria k objektom v `clientobjects` tagu (3).

[4]t.j. priamy potomok v stromovej štruktúre

5. `<object name="..." parent="..." abstract="..." label="..."></object>`
   `Umiestnenie:` Vnútri tagov: `clientobjets`(3),`nonclientobjects`(4).
   `Význam:` Definuje sa Trieda pomocou tohto tagu.Parameter *name* definuje názov triedy, parameter *parent* hovorí o mene priameho predka triedy (ak sa trieda predka nevyskytuje v štruktúre návrhového vzoru či je nepodstatný tak hodnota je *null*),parameter *abstract* hovorí či je trieda abstraktná alebo nie (hodnoty *true* alebo *false*). Ohľadom attribúta *label* viď. časť 2.2.1.

6. `<objects name="..." parent="..." abstract="..." label="..."></objects>`
   `Umiestnenie:` Vnútri tagov: `clientobjets`(3),`nonclientobjects`(4).
   `Význam:` Rovnaký ako pre tag `object`(5),avšak s 1 zmenou.Tag `object` definoval 1 triedu a tento tag definuje neurčitý počet tried,ktoré sú identické až na jemné odchylky v mene a na odlišnosti vo vnútornej implementácií častí tiel metód,ktoré daný návrhový vzor bližšie nešpecifikuje. Dobrým príkladom sú podobné triedy *ConcreteStrategy* v návrhovom vzore *Strategy*.

7. `<implementInterface>...</implementInterface>`
   `Umiestnenie:` Vnútri tagov: `object`(5),`objects`(6).
   `Význam:` Hovorí o implementácií interfacu vnútri triedy.

8. `<is_high_level_list_to>...</is_high_level_list_to>`
   `Umiestnenie:` Vnútri tagov: `object`(5),`objects`(6).
   `Význam:` Medzi týmito tagmi sa nachadza meno triedy, na ktorej pole drží táto trieda[5] vysoko-úrovňové (high level) rozhranie. Tento tag hovorí síce sám dosť vysoko-úrovňovú informáciu avšak je to napríklad kvôli definícií triedy ObjectStructure v návrhovom vzore Visitor.

9. `<reference name="..." reffers_to="..." is_list="..." assign_from="..."></reference>`
   `Umiestnenie:` Vnútri tagov: `object`(5),`objects`(6).
   `Význam:` V triede sa nachádza referencia na objekt inej triedy. Parameter *name* by mala obsahovať meno tejto premennej (môže obsahovať aj hodnotu *null* v prípade,že meno pri definícií návrhového vzoru nebolo

[5]Mám na mysli triedu,v ktorej *object* tagu(5) práve som.

niak špecifikované), parameter *reffers to* definuje meno triedy objektu, na ktorý referencia má ukazovať, parameter *is list* len hovorí či sa jedná o referenciu na 1 objekt alebo na pole objektov rovnakej triedy (hodnota *true* alebo *false*) a posledný parameter hovorí o identite iniciatora tejto referencie (hodnotou môže byť meno triedy zo štruktúry návrhoveho vzoru alebo *Client* alebo *null*,v prípade nenastavenia(t.j. defaultnej hodnoty *null*)).

10. `<unspecified_variable name="..."></unspecified_variable>`
    `Umiestnenie:` Vnútri tagov: `object`(5),`objects`(6).
    `Význam:` Definuje 1 premennú vnútri triedy,pričom o premennej vieme len meno definované cez parameter *name*. Takto definovaná premenna je potrebná napríklad ako premenná stavu v návrhových vzoroch.

11. `<object_with_access></object_with_access>`
    `Umiestnenie:` Vnútri tagov: `reference`(9),`unspecified_variable`(10).
    `Význam:` Hovorí meno 1 triedy objektov,ktorá má prístup k danej premennej (definovanej tagom `reference` alebo tagom `unspecified_variable`). Nehovorí nič o spôsobe prístupu, t.j. či priamo alebo cez metódy.

12. `<client_access/>`
    `Umiestnenie:` Vnútri tagov: `reference`(9),`unspecified_variable`(10).
    `Význam:` Taký istý ako tag `object_with_access`(11), avšak meno triedy je neznáme, pretože tá trieda je klient.

13. `<method name="..." is_constructor="..."`
    `label="..."></method>`
    `Umiestnenie:` Vnútri tagov: `object`(5),`objects`(6),`interface`(25).
    `Význam:` Definuje metódu, ktorá má názov ako parameter *name*. Parameter *is constructor* hovorí o tom či je metóda konštruktorom(`true` alebo `false`). V prípade, že áno, tak sa ignoruje nastavené meno a návratová hodnota zadefinovanej metódy. Ohľadom attribúta *label* viď. časť 2.2.1. Presnejšia definícia metódy bude nasledovať vo vnútri tohto tagu pomocou ďalších tagov. V prípade, že sa nachádza tento tag vnútri tagu `interface`, tak už nemôže obsahovať tag `demands`(17). Tento tag by nemal definovať metódu preťažujúcu metódu z interfacu, ktorý trieda, v ktorej sa táto metóda nachádza, implementuje.

14. `<return_type>...</return_type>`
    `Umiestnenie:` Vnútri tagu `method`(13).
    `Význam:` Tento tag by mal obsahovať nazov triedy alebo nazov typu primitivnej premennej[6], ktorú daná metóda vracia, alebo *void* v prípade, že nič nevracia.

15. `<list_of_parameters></list_of_parameters>`
    `Umiestnenie:` Vnútri tagu `method`(13).
    `Význam:` Zaobalenie množiny tagov `parameter`(16).

16. `<parameter type="..."></parameter>`
    `Umiestnenie:` Vnútri zaobalovacieho tagu `list_of_parameters`(15).
    `Význam:` Definuje v poradí ďalší parameter metódy z tagu `method`. Hodnotou môže byť buď názov triedy[7] alebo názov typu primitívnej premennej.

17. `<demands></demands>`
    `Umiestnenie:` Vnútri tagu `method`(13).
    `Význam:` Vnútri tohto tagu bude nasledovať postupnosť požiadavok na danú metódu v takom poradí aké potrebujem, aby bolo dodržané vnútri metódy pri implementačnom plnení týchto požiadaviek.

18. `<call_method use_reference="..." method_name="."`
    `is_return_call="..." is_list_command="...">`
    `Umiestnenie:` Vnútri tagu `demands`(17).
    `Význam:` Hovorí, že v danej metóde má byť zavolaná ďalšia metóda. Parameter *use reference* hovorí či sa má použiť niaka referencia na objekt, ktorého metóda má byť zavolaná. Ak áno, tak tento parameter je menom tejto referencie (prípustne referencie: Referencia tejto triedy na inú triedu ,parameter pre túto metódu alebo slovíčko *super* ako označenie referencie na predka triedy) a parameter *method name* je názov volanej metódy cez zvolenú referenciu. Ak nie, tak má parameter *use reference* hodnotu *null*, *new* alebo *this* . Ak je to hodnota *null*,tak parameter *method name* obsahuje názov metódy triedy, v ktorej som, a tá sa má zavolať. V prípade hodnoty *new* je v parametri *method name* názov triedy, ktorej konštruktor sa má zavolať. Ak je hodnota *this*, tak

[6]Mám na mysli premennú typu *int*,*double*,*long*,atd.

[7]Tu by bolo asi najlepšie,keby som sa neobmedzoval len na mená tried zo štruktúry návrhového vzoru

v parametri *method name* je názov premennej tejto triedy, ktorá sa má poslať ako návratová hodnota tejto metódy (*is return call*==„`true`“). Parameter *is return call* vždy hovorí o tom, či sa má návratová hodnota zavolanej funkcie zobrať ako návratová hodnota funkcie, ktorú práve definujem. Parameter *is list command* hovorí či sa jedná o metódu, ktorá sa má zavolať na zozname referencií. Kedže tento parameter je nepovinný, tak defautné nastavenie tohto atribútu je `false` (t.j. metóda sa zavola pre každú referenciu v zozname). Prípustné hodnoty obidvoch parametrov sú *true* a *false*.

19. `<list_of_parameters_for_call></list_of_parameters_for_call>`
`Umiestnenie:` Vnútri tagu `call_method`(18).
`Význam:` Zaobalovací tag pre množinu tagov troch typov: `use_reference`, `use_parameter`, `this`. V prípade prázdnej množiny parametrov pre method call, tento tag nemusí byť implementovaný.

20. `<use_reference></use_reference>`
`Umiestnenie:` Vnútri zaobalovacieho tagu `list_of_parameters_for_call`(19).
`Význam:` Ďalším parametrom funkcie volanej z vnútra metódy okjektu, ktorý definujem, je referencia s menom, ktoré je medzi týmito tagmi.

21. `<use_parameter></use_parameter>`
`Umiestnenie:` Vnútri zaobalovacieho tagu `list_of_parameters_for_call`(19).
`Význam:` Ďalším parametrom funkcie volanej z vnútra metódy okjektu, ktorý definujem, je parameter tejto vnútornej metódy s menom medzi týmito tagmi.

22. `<this/>`
`Umiestnenie:` Vnútri zaobalovacieho tagu `list_of_parameters_for_call`(19).
`Význam:` Ďalším parametrom funkcie volanej z vnútra metódy okjektu, ktorý definujem, je referencia na seba,t.j. premenna *this*.

23. `<added_behaviour/>`
`Umiestnenie:` Vnútri tagu `demands`(17).
`Význam:` Tento tag vyžaduje v metóde medzi dvoma inými požiadavkami alebo medzi poslednou požiadavkou a koncom metódy niaky neprázdny java kód.

24. `<template>...</template>`
    `Umiestnenie:` Vnútri tagov: `demands`(17),`object`(5),`objects`(6).
    `Význam:` Text vnútri tohto tagu bude vpísaný do templatu (bez ohľadu na obsah textu). V prípade umiestnenia tagu do `demands` tagu sa vypíše text do metódy podľa umiestnenia tagu medzi ostatnými tagmi (`call_method`(18), `addedBehaviour`(23)). V prípade umiestnenia tagu inde sa text vypíše do templatu triedy na pozíciu za všetkými premennými a pred všetkými metódami. Prázdny text nebude akceptovaný.
    Výsledný výpis textu bude odsadený. Na tento účel sme zaviedli pár pravidel navyše. Za text sa bude považovať len obsah riadkov medzi párovým tagom (t.j. tie bez `<template>`, `</template>`). Odsadenie prvého riadku s textom sa bude považovať za default odsadenie a ostatné riadky budú odsadené v template tak, ako sú odsadené voči dafaultnému odsadeniu v xml súbore (avšak kratšie odsadenie voči defaltnému sa zmení na default odsadenie).

25. `<interface name="..." parent="...">`
    `Umiestnenie:` Vnútri tagov: `clientobjets`(3),`nonclientobjects`(4).
    `Význam:` Definuje sa Interface pomocou tohto tagu. Parameter *name* definuje názov interfacu a parameter *parent* hovorí meno 1 rodiča interfacu, ktorý musí byť tiež interface, alebo hovorí, že nemá rodiča hodnotou *null*. Interface v tomto modeli môže mať maximálne 1 rodiča.

26. `<interfaceVariable name="...">`
    `Umiestnenie:` Vnútri tagu `interface`(25).
    `Význam:` Definuje premennú s modifikátormi `static` a `final` vo vnútri interfacu.

27. `<helpLabel name="..." is_list="...">`
    `Umiestnenie:` Vnútri tagu `designpattern`(2).
    `Význam:` Definuje pomocný zoznam mien pre dynamickú tvorbu mien iných tried a metód, ktoré sa využijú pri tvorbe templatov. Jedná sa čisto o pomocnú štruktúru pre mená. Takýto pomocný zoznam treba definovať pred akýmkoľvek tagom vnútri tagov `clientobjets`(3), `nonclientobjects`(4) alebo ho nedefinovať vôbec (t.j. ho nepoužiť). Atribút *name* definuje meno zoznamu a atribút *is list* definuje či zoznam je jednoprvkový zoznam (hodnota `true` alebo `false`).

28. `<defaultValue>...</defaultValue>`
    `Umiestnenie:` Vnútri tagu `helplabel`(27).
    `Význam:` Naplňuje zoznam reprezentovaný tagom `helplabel` novými menami. Ak je zoznam jednoprvkový, tak sa naplní hodnotu posledného takéhoto tagu. Minimálne však 1 takýto tag musí zoznam implementovať.

Ukážkové príklady použitia tohto modelu sa nachádzajú v prílohe.

2.2.1 Štítky

Názvy niektorých tried v niektorých návrhových vzoroch závisia od názvov iných tried, preto statické pomenovanie objektov (a metód) nie je dostačujúce na zaznamenanie informácie o návrhovom vzore. Preto sme zaviedli pojem **štítka** (label). Jedná sa o možnosť označenia tried a metód štítkom s určitým menom, pričom sa tento štítok môže vyskytnúť v mene iných tried, metód,..., kde bude predstavovať výsledné[8] meno(mená) oštítkovaného objektu. Oštítkovanie prebieha priradením mena štítka do nepovinného attribútu *label* objektu, ktorý chceme oštítkovať. Konkrétne sa jedná len o tagy: `object`(5), `objects`(6), `method`(13), ktoré môžeme oštítkovať. Existuje niekoľko spôsobov použitia štítkovania:

1. Použitie má tvar `#menoštítka#` a bude sa týkať len attribútov *name* v tagoch `object`(5) a `method`(13). Takýto tvar sa vo výslednom mene (menách) premení na výsledné meno(mená) objektu, ktorý má štítok s daným menom. Keď štítok odkazuje na objekt s viacerými výslednými menami, tak počet výsledných mien mena, ktoré štítok obsahuje, sa znásobí. Tak napríklad ak meno je „decorator`#štítok1#`“ a štítok s meno štítok1 odkazuje na objekt s výslednými menami „1“,„2“,„3“, tak výsledné mená sú „decorator1“, „decorator2“, „decorator3“.

2. Použitie má tvar **...** (x-násobné použitie znaku *) a neodkazuje na štítok, ale na x-té použitie štítka spôsobom č.1 v mene triedy(tag `object`). Tak napríklad ak meno triedy je „super`#štítok1#`dobrá`#štítok2#`-trieda“, `#štítok1#` má výsledné mená „a“,„alebo“, `#štítok2#` má výsledné mená „jeho“,„moja“ a meno metódy je „pridaj**“, tak sa vyge-

[8] Treba odlišovať pojem mena a výsledného mena. Meno je text daný attribútom *name* v xml súbore, zatiaľ čo výsledné mená (output names) sú mená, ktorými sa bude prezentovať daná trieda(metóda) v template.

neruje template s triedami „superadobrájehotrieda“ (obsahujúca metódu „pridajjeho“), „superalebodobrájehotrieda“ („pridajjeho“), „superadobrámojatrieda“ („pridajmoja“), „superalebodobrámojatrieda“ („pridajmoja“).
Táto metóda sa môže použiť len v atribútoch *name* (tag `method`(13)), *type* (tag `parameter`(16), *method name* (tag `call_method`(18 s atribútom *use reference*==„new“)) a ešte v atribúte *parentname* (tag `object`(5)), ale tam len ak znaky „*“ budú predstavovať celý atribút. Druhý spôsob použitia štítkovania hovorí o akejsi zosynchronizovateľnosti mena triedy a mien prvkov vnútri triedy.

3. Použitie má tvar ++...++ (x-násobné použitie znaku +) a znamená to isté ako predošlé použitie len s tým rozdielom, že sa nepracuje s menom triedy, ale s menom metódy, v ktorej sa tag a aj atribút, kde sa takýmto spôsobom používa štítkovanie, nachádza. Táto metóda sa môže použiť len v atribútoch *type* (tag `parameter`(16) a *method name* (tag `call_method`(18) s atribútom *use reference*==„new“).

Znaky * a + by mali síce slúžiť len na štítkovacie účely tak ako je to už hore uvedené, avšak výnimkou je použitie reťazca „*“(„+“) v prípade, že trieda(metóda) v mene nepoužíva štítky. Vtedy tieto reťazce reprezentujú výsledné meno či mená triedy(metódy).
Pomocnou konštrukciou pre model je konštrukcia tagu `helplabel`(27), ktorý slúži ako pomocný zoznam mien a dá sa použiť ako objekt označený štítkom. Pre implementáciu zjednodušením, ale reštrikciou pre model, je fakt, že v tagu `interface`(25) a v tagov vnútri neho sa nesmú používať štítky[9]. Ďalšou ešte logickou reštrikciou modelu je, že metóda nemá viac ako 1 výsledné meno (output name) ak nepoužíva v mene štítky. Ale už nelogickou reštrikciou modelu je fakt, že atribút *name* (tag `object`, tag `method`) nesmie použiť štítok na metódu, ktorá pouziva štítky spôsobom č.2. Jednoducho niekde pri implementácií tento fakt veľmi zjednodušil situáciu a modelovaniu návrhových vzorov to neuškodilo.

2.3 Pamäťový dátový model návrhového vzoru

Pamäťový dátový model má slúžiť na prácu s návrhovým vzorom. Vzniká načítavaním súborového dátového modelu a ukladaním získaných informácií do

[9] Ani označovať štítkom, ani používať štítkovanie v atribútoch

meno tagu	**korešpondujúca trieda(štruktúra obsahujúca triedu) k danému tagu**
`object_with_access`	`Access`
`added_behaviour`	`AddedBehaviour`
`demands`	zoznam tried `Demand`(abstraktná trieda pre požiadavky)
`helpLabel`	`HelpLabel`
`is_high_level_list_to`	`HighLevelList`
`interfaceVariable`	`InterfaceVariable`
`method`	`Method`
`call_method`	`MethodCall`
`template`	`MethodTemplate` či `FakeMethod` (záleží od umiestnenia tagu)
`object, objects, interface`	`ObjectElement`
`list_of_parameters`	`Parameters`
`design_pattern`	`Pattern`
`reference`	`Reference`
`unspecified_variable`	`UnspecifiedVariable`

Obr. 2.1: Tabuľka prechodu od súborového k pamäťovému dátovemu modelu návrhového vzoru

dátových štruktúr jazyka Java. Jedná sa teda o skupinu tried s určitými vnútornými premennými. Tieto triedy sú uložené v našom plugine ako package `designpatternplugin.core.datatypes`. Kedže sa napĺňajú informáciami postupne ako sa načítavajú zo súborového modelu, tak ich štruktúra bude veľmi podobná štruktúre súborového modelu. Skoro ku každému tagu zo súborového modelu existuje korešpondujúca trieda, ktorá obsahuje premenné reprezentujúce atribúty tagu a referencie (zoznam referencií) na triedy reprezentujúce priame vnútorne[10] tagy tagu. Tabuľka 2.1 znázorňuje prechod od tagov ku triedam (či štruktúram z tried). Ostatné tagy nespomenuté v tabulke sú reprezentované buď primitívnymi typmi alebo kombináciou či zoznamom tried v tabuľke uvedených.

[10]Vnorené tagy do hĺbky 1 tagu.

Kapitola 3

Problémy netýkajúce sa prostredia Eclipse 3.2

V predošlých kapitolách sme si hovorili veľa o riešeniach rôznych problemov pri implementácií nášho pluginu. Je načase celý proces implementácie zosumarizovať a zamerať sa ešte na tie problémy, ktoré nie su doriešené.

Prvá časť pluginu, zameraná na templaty, sa implementačne začína napojením wizardu na správany extension point(viď. kapitolu 1) a pokračuje napĺnaním wizardu stránkami (viď. kapitolu 1.2.3), pričom sa využíva pamäťový dátový model návrhového vzoru (viď. kapitolu 2.3), ktorý vznikne načítaním súborového dátového modelu návrhového vzoru (viď. kapitolu 2.2). Po skončení wizardu (t.j. po vypýtaní všetkých informácií na tvorbu templatu) sa vytvoria súbory templatu jednoduchým vpisovaním kúskov textu do súbora, pričom text sa vytvorí na základe prechodu cez pamäťový model návrhového vzoru a informácií z wizardu. Jediným väčším problémom tejto časti, ktorý sme ešte nerozoberali, je problém generovania výsledných mien (viď. 3.1), ktorý pojednáva o generovaní mien pre triedy, metódy a premenné, ktoré sa použijú v template. Toto by nebol až taký veľký problém, keby model neobsahoval štítky(viď. 2.2.1).

Druhá časť je implementačne zaujimavejšia. Rozšírime menu Eclipsu o 1 prvok (viď. kapitolu 1). Pri jeho použití vytvoríme Dialog (viď. kapitolu 1.2.3), ktorým si vypýtame informácie nutné na hľadanie návrhového vzoru v určenom projekte. Návrhový vzor vyhľadáme a výsledky zapíšeme do náhľadu(view) (viď. kapitolu 1) pripojenému k prostrediu podľa vhodného extension pointu. Jedinou otvorenou otázkou zostáva ako budeme návrhové vzory vyhľadávať (viď. 3.2).

Implementácia 2.časti je dostatočne časovo zložitá. Prípadné prerušenie hľadania návrhového vzoru pomocou visuálných komponentov Eclipsu nebolo možné z dôvodu tejto časovej zložitosti, a preto sa zdalo nutné spustiť hľadanie návrhového vzoru ako nové vlákno (Thread), aby grafické komponenty Eclipsu reagovali v reálnom čase. Riadenie výpočtu ako aj výsledky výpočtu sa prenášali pomocou premenných, ku ktorým sa pristupovalo len pomocou synchronizovaných metód.

3.1 Problém generovania výsledných mien

Problémom nie su mená, ktoré nepoužívaju štítky ako súčasť mena. Týmto sa dajú ľahko nastaviť niake výstupné mená[1]. Ozajstným problémom sú mená používajúca štítky. Tieto mená sa musia určitým spôsobom vyhodnotiť na bezštítkovú formu výstupných mien. Deje sa tak zdanlivo veľmi jednoducho. Nájdem v mene všetky použitia štítkov a pokúšam sa ich nahradiť výstupnými menami objektov, na ktoré tieto štítky ukazujú. Ak výstupných mien pre niake štítky je viac, tak dosadzujeme tak, aby sme dosiahli všetky možné kombinácie dosadenia výsledných mien zo štítkov. Týmto spôsobom počet výsledných mien mena narastá.
Štítok môže ale odkazovať na objekt, ktorý používa v mene tiež štítok, a tak vyhodnocovanie mena so štítkom musí byť rekuzívne[2]. Problém nastáva ak by sa odkazovanie štítkami zacyklilo. V takom prípade sa meno nevyhodnotí a program padne[3] kvôli nedostatku pamäti. S tohto dôvodu sú cyklické odkazy zakázané. Avšak pre istotu[4] ich detekujeme a to podľa udržovania FIFO zasobníka[5] obsahujúceho mená štítkov, do ktorých sa vnárame pri rekurzií. Ak sa vnárame do štítka s meno, ktoré je už v zásobníku, tak sme sa zacyklili. Keby sme používali len 1.typ použitia štítka v mene (viď. 2.2.1), tak hore uvedeným spôsobom zistím výsledné mená pre prvky používajúce štítky v mene. Keď budeme potrebovať vygenerovať triedu, ktorá v mene používa štítky, tak zistíme výsledné mená a pre každé z nich vytvoríme triedu. Podobne postupujeme s metódami a premennými. Čo sa stane ak použijeme

[1]Mena použité pre daný prvok pri tvorbe templatu.

[2]Rekurzia nie je jediná možnosť, ale je to najjednoduchšia možnosť.

[3]Za predpokladu nenulovej spotreby pamäte 1 cyklu.

[4]Pri zložitejších vzťahoch objektov nie je vždy jednoduché ručne odsledovať či štítky nevytvárajú cyklus.

[5]First in,first out. Jedná sa o stack.

výsledné mená triedy A	výsledné mená metódy B
triedaA	prvámetódaA
triedaA	druhámetódaA
triedaB	prvámetódaB
triedaB	druhámetódaB

Obr. 3.1: Vygenerované zoznamy výsledných mien

2. alebo 3. typ použitia štítka (znaky +,*)? Ukážme si to na príklade. Nech trieda A používa v mene štítky a nech metoda B je metóda v triede A, ktorá používa štítkovanie 2.spôsobom (odkaz na použitie štítka v mene triedy A). Keby sme vygenerovali metóde B 1 zoznam výsledných miest, tak meno metódy B sa v templatu nemení podľa mena triedy, v ktorej sa nachádza, ale vygenerujú sa v triede A všetky možnosti triedy B. Preto treba metóde B vygenerovať zoznam výsledných mien, ktoré nebude používať znova a znova, ale len raz. Lenže ako povedať metóde, ktorú časť zoznamu má zobrať ako zoznam výsledných mien metódy B pre práve generovaný text triedy A s daným výsledným menom? Odpoveďou je padding[6] zoznamov výsledných mien pre triedu A tak, aby jeho dĺžka bola rovnaká s dĺžkou zoznamu výsledných mien metódy B (t.j. ku každému menu v zozname výsledných mien trieda A príslúcha 1 výsledné meno metódy B).
Názorne si to ukážeme pre konkrétne meno triedy A („trieda#štítok1#“) a konkrétne meno metódy B („#štítok2#metóda+“), kde štítok1 predstavuje výsledné mená „A“,„B“ a štítok2 výsledné mená „prvá“,„druhá“. Následne zoznamy výsledných mien trieda A a metódy B sú v tabuľke 3.1. Zoznam výsledných mien pre triedu A nie je síce celkom bez redundancie, avšak rieši daný problém.
Pri zoznamoch z tabuľky 3.1 by generácia templatu postupovala podľa nasledovných pravidiel:

1. Bude sa prechádzať zoznamom výsledných mien pre triedu A (a zároveň aj zoznamom pre metódu B, tak aby pozície v zoznamoch boli rovnaké)

2. Ak sa text výsledného mena triedy A zmení, tak sa práve prebiehajúce generovanie triedy A s určitým meno ukončí a začne sa generovať znova trieda A, ale už s novým menom.

[6]V tomto prípade to znamená vyplnenie zoznamu redundantnou informáciou.

3. Ak sa zmení text výsledného mena metódy, tak sa vygeneruje metóda s daným menom.

Zovšeobecnením tohto príkladu dostaneme spôsob riešenia problému. Vytvoríme 2 cykly do seba vnorené, pričom vonkajší cyklus bude cyklus meniaci výsledné meno triedy a vnútorný cyklus bude meniť výsledné meno metódy. Zoznam výsledných mien pre metódu, ale aj pre triedu, bude rásť vo vnútri vnútorného cyklu. Zoznamy ostatných prvkov používajúcich v mene štítky 2.a 3. spôsobom tiež budú rásť len vo vnútri vnútorného cyklu.

3.2 Problém hľadania návrhového vzoru

Pamäťový dátový model návrhového vzoru sme si už objasnili (viď. kapitola 2.3), tak ako sme si objasnili ako vybrané zdrojové súbory rozpitvať až na základné java elementy (viď. kapitola 1.1). Takže keby sme vymysleli spôsob ako rozumne priraďovať skupiny zložené z tried(a interfacov) ku objektom(`ObjectElement`) modelu návrhového vzoru (viď. 3.2.1) a následne by sme dokázali overovať nakoľko sa trieda(interface) podobá na objekt modelu návrhového vzoru, ku ktorému bol priradená,(viď. 3.2.2) tak by sme boli schopný generovať priradenia tried(interfacov) ku objektom modelu a povedať nakoľko sa také triedy(interface) priradenia podobajú na návrhový vzor. Stačilo by už len vybrať tie najpodobnejšie priradenia a problém by bol vyriešený.

3.2.1 Problém vytvárania konfigurácií

V implementácií sa týmto problémom zaoberá trieda `designpatternplugin.core.ClassListGenerator`, ktorá sa chová podobne ako interface `java.util.Iterator`.
Zadefinujme si pojem **konfigurácia**. Konfigurácia je zoznam obsahujúci zoznamy, ktoré obsahujú triedy a rozhrania. Podľa toho ako sú zoznamy v zozname usporiadané sa zoznamy pridaďujú jednotlivým ObjectElementom (reprezentácia objektu a interfacu v modeli návrhového vzoru).
Zadefinujme si pojem **levelu**. Level X v konfigurácií predstavuje x-tý zoznam v zozname. Výššie levely predstavujú zoznamy umiestnené ďalej od začiatku zoznamu. Najnižší level konfigurácie je level 0.

Konfigurácie sa vytvárajú postupne od najnižšieho levelu po ten najvyšší. Najprv sa priradia niake triedy či rozhrania najnižšiemu levelu. Potom levelu o 1 väčšom až sa naplní najvyšší level a máme prvú konfiguráciu. Ďalšiu konfiguráciu dostaneme naplnením najvyššieho levelu inou skupinou tried a rozhraní. Ak zistím, že sa už daný level nedá naplniť inou skupinou tried a rozhraní, tak sa presuniem o 1 level nižšie a snažím sa tam zmeniť skupinu tried a rozhraní. Ak sa mi to na tom nižšom leveli podarí, tak musím vyššie leveli naplniť inicializačnými[7] skupinami tried a rozhraní až po najvýšší level, aby som dostal ďalšiu konfiguráciu. Ak sa mi to na žiadnom nižšom leveli nepodarí, tak som z generovaním skončil. Jedná sa teda o akési napĺnanie konfigurácie všetkými kombináciami skupín tvorených triedami a rozhraniami.
Na začiatku generovania konfigurácií máme určitú veľkú skupinu tried a rozhraní, v ktorej máme hľadať návrhový vzor. Prvá skupina priradená levelu 0 okradne túto veľku skupinu o určité metódy a rozhrania, tak ako každá skupina priraďovaná k niakemu levelu. Táto veľka skupina je v skutočnosti skupina tried a rozhraní, ktoré môžeme ešte použiť na tvorbu skupín priraďovaných ku levelom konfigurácie.
Ako majú byť ale jednotlivé skupiny priraďované k levelom veľke? To bude závisieť od ObjectElementu prislúchajuceho k danému levelu, resp. od jeho mena. Ak meno nebude obsahovať štítky a ObjectElement bude vytvorený podľa tagu `object`, tak veľkosť bude vždy 1, pretože i template k takémuto ObjectElementu by obsahoval len 1 triedu. Ak je pôvod v tagu `objects` alebo meno obsahuje štítky tak sa bude veľkosť postupne zväčšovať, t.j. ak sa už nebude dať vytvoriť nová[8] skupina tried a rozhraní danej veľkosti, tak sa zväčší veľkosť. Napokon sa veľkosť zväčší natoľko, že sa nebude dať vytvoriť ani 1 skupina o danej veľkosti a vytváranie skupín pre daný level sa predbežne skončí (a nastane posun generácie skupín o level nižšie). Veľkosť skupiny bude predstavovať možný počet vytvorených tried z ObjectElementu pri tvorbe templatu. Veľkosť zväčšovania veľkosti skupiny bude závisieť od štítkov v mene. Inicializačne bude nastavený krok zväčšenia veľkosti na 1. Prenásobenie tohto kroku nejednotkovým číslom bude možne jedine v prípade obsahu štítka na iný ObjectElement, ktorého aktuálna veľkosť skupiny bude viac ako 1. Aby sme zistili aktuálnu veľkosť niakeho ObjectElementu, tak k nemu príslušný level musí byť nižší ako level, pre ktorý veľkosť po-

[7] Podľa určitých pravidiel tvorenia skupín, prvými (skupinami)
[8] Odlišná od predošlých skupín.

čítam[9]. Takáto vlastnosť poradia ObjectElementov sa dá preusporiadaním ObjectElementov pred začatím generácie konfigurácií dosiahnuť, pretože štítkové odkazy nemôžu byť cyklické a teda tvoria stromovú štruktúru, podľa ktorej ObjectElementy usporiadam.
Sme teda schopný vytvoriť číslo veľkosti pre každý level a pre niektoré ho vhodne zvyšovať. Ale ešte stále sme si presne nepovedali ako vyberieme skupinu o danej veľkosti zo skupiny voľných[10] tried a rozhraní. Voľné triedy a rozhrania sa dajú oindexovať číslami 0 až n a my hľadáme m rôznych indexov (výberov), kde m je veľkosť vyberanej množiny. Na túto situáciu sa dá použiť rovnaký princíp ako sme použili na tvorbu levelov pre konfiguráciu, t.j. rekurzívne napĺňať indexy. Namiesto vytvárania nových skupín len nájdeme ešte nepoužitý index.

Zatiaľ sme si ukázali metodický postup získavania konfigurácií. Jediným problémom takéhoto výberu je exponenciálny počet konfigurácií vzhľadom na veľkosť skupiny tried a rozhraní, v ktorých sa má pattern vyhladávať. To v praxi znamená veľmi dlhý čas hľadania návrhového vzoru. Preto sme implementovali 1 spôsob redukcie počtu konfigurácií vychádzajúci z nevyužitej informácie ohľadom hľadaného návrhového vzoru. Jedná sa o to, že pri štrukturálne najnižšej úrovni výpočtov, t.j. pri vyberaní m rôznych indexov z $n+1$ indexov, zredukujeme počet indexov, z ktorých si budeme môcť vyberať a to na základe ObjectElementu prislúchajúcemu danému levelu, pre ktorý počítam indexy. Každý ObjectElement môže mať(ale aj nemusí) definovaného predka (parent ObjectElement). Keby predok daného ObjectElementu bol na nižšom leveli konfigurácie, tak by mal už priradené triedy a rozhrania. Takže by sme z dostupných indexov pre skupinu voľných tried a rozhraní vyhodili tie, ktorých trieda(rozhranie) nemá priameho predka v skupine predka ObjectElementu. Tým sa zbavím tried a rozhraní, ktoré by porušovali systém dedenia použitom v modeli návrhového vzoru. Jedinou nevýhodou takejto redukcie je fakt, že predok daného ObjectElementi nemusí byť na nižšom leveli. Dali by sa síce ObjectElementy preusporiadať tak, aby tomu tak bolo, avšak mohla by sa poškodiť vlastnosť o štítkoch, kvôli ktorej sme už raz ObjectElementy preusporiadavali, takže preusporiadanie ako riešenie nevyhovuje. Museli sme sa uchýliť k drastickejšiemu riešeniu. V prípade výberu urcitej triedy alebo rozhrania (označme ju/jeho ako trieda A)do

[9]Levely vyššie ako ten, s ktorým pracujem, sú neaktívne a majú nulovú veľkosť
[10]t.j. ešte nepriradených do levelových skupín

skupiny k ObjectElementu A sme nastavili požiadavku na výber triedy alebo rozhrania pre level ObjectElementu B, kde ObjectElement B je rodič ObjectElementu A a level ObjectElementu B je vyšší ako level ObjectElementu A, aby obsahoval triedu(rozhranie), ktoré je predkom triedy A. Ide o akési zarezervovanie výberu vo vyšších leveloch. Implementačne sme pre indexy vyrábali pole(`int[]`), kde 0 pre index znamenalo, že si tento index nemôžeme vybrať, 1, že môžeme, a 2, že musíme. Pri takýchto komplikovaných požiadavkách nastáva veľké množstvo prípadov zaseknutia generácie (t.j. presun na inú veľkosť skupiny alebo presun na nižší level) a to sťažovalo implementáciu. Časovú zložitosť po takejto úprave som kvôli zložitosti a zbytočnosti z implementačného pohľadu nepočítal.

3.2.2 Problém podobnosti triedy a objektu modelu

Keďže potrebujeme zistiť podobnosť triedy(rozhrania) a ObjectElementu(objekt v modeli reprezentujúci triedu a rozhranie), tak si jednotlivé časti štruktúry ObjectElementu ohodnotíme podľa dôležitosti danej časti pre podobnosť k návrhovému vzoru. Jednotlivé časti štruktúry spolu s odôvodneným ohodnotením prezentuje tabuľka 3.2. Potom budeme prechádzať štruktúrou ObjectElementu a zároveň triedy(rozhrania) približne vždy po ekvivalentných úrovniach. Pozastavíme sa len pri ohodnotených štruktúrach a zistíme či trieda(rozhranie) má niečo podobné(rovnaké) alebo nie. Ak áno pripočítame ohodnotenie štruktúry, ak nie tak nič. Nakoniec dostaneme pre danú triedu(rozhranie) číslo, ktoré dáme do pomeru s maximálne možným ohodnotením pre ObjectElement a dostanem hodnotu podobnosti s ObjectElementom. V prípade viacerých tried určených pre 1 ObjectElement urobíme aritmetický priemer. Pre celý pattern sa nerobí aritmetický priemer, ale sa vypočíta výraz $\frac{\sum_{ObjectElementX} podobnostpreX * maxhodnotapreX}{\sum_{ObjectElementX} maxhodnotapreX}$

Problémy nastávajú pri metóde. K metóde návrhového vzoru vyberáme reálnu metódu takým spôsobom, že sa snažíme vybrať metódu s najväčším stupňom podobnosti. Problém nastáva, keď je takých metód viac a musíme vybrať len jednu. Ak to bude tá nesprávna, tak sa to môže nepríjemne dotknúť ohodnotenia demands (požiadavok na vnútro metódy), pretože demands nieleže nesadnú na danú metódu, ale aj metóda, ktorá bola takto zle odobratá možno má v návrhovom vzore ekvivalent a jeho demands sa tiež správne nedetekujú. Táto chyba bola hlavným dôvodom nízkeho ohodnotenie štruktúry methodDemandMethod.

časť štruktúry (ohodnotenie)	**dôvod ohodnotenia**
isInterface(750)	Modifikátory objektu sú dôležité, ale nie až tak ako parent.
isAbstract(750)	Modifikátory objektu sú dôležité, ale nie až tak ako parent.
parent(1000)	Základná hodnotová jednotka.
interfaceImplemented (2000)	Môže v sebe skrývať viac metód a tak sa mi zdá jeho úspešná implementácia ako veľmi cenná, ale nemusí obsahovať veľa metód, a tak kompromis.
referenceType(1000)	Spojenie cez referenciu môže byť rovnako významné ako spojenie cez parenta.
highLevelListImplemented(1750)	Rovnako ako pri interfaceImplemented, avšak detekcia nie je taká dobrá.
uVarImplemented(150)	Ľahko naplniteľná podmienka.
methodIsConstructor (1000)	Relatívne vzácna vlastnosť definujúca metódu.
methodReturnType (900)	Menej vzácna vlastnosť definujúca metódu.
methodParameterType (400)	Vlastnosť definujúca metódu, avšak môže byť mnohopočetná.
methodDemandAddedBehaviour(150)	Ľahko splniteľná podmienka.
methodDemandMethod (400)	Dôležitá vlastnosť, ale zle sa detekuje, a tak nižšia hodnota==nižšia chyba.

Obr. 3.2: Ohodnotenie častí štruktúry ObjectElementu.

Záver

Bakalárska práca s názvom **Java a Návrhové vzory** si predsavzala 2 ciele. Obidva súvisia s pluginovou podporou vývojového prostredia Eclipse 3.2 . Prvým cieľom bolo vytvoriť systém tvorby templatov návrhových vzorov a to sa skutočne podarilo. Druhým cieľom bolo vytvoriť detekčný systém návrhových vzorov pre už existujúce projektové štruktúry. Tento cieľ sa podarilo síce úspokojivo implementovať, avšak ešte stále je tu určitý priestor pre zlepšovanie detekcie či prípadne rýchlosti detekcie, ktorá je v praktickom svete často kľúčová.

Literatúra

[1] Elektronický help systém integrovaný do vývojoveho prostredia Eclipse 3.2

[2] Odpoveď na často kladenú otázku:Ako pristupuvať k aktívnemu projektu?
(http://wiki.eclipse.org/index.php/FAQ How do I access the active project?)

[3] Odpoveď na často kladenú otázku:Ako nájdem inštalačné miesto pluginu?
(http://wiki.eclipse.org/index.php/FAQ How do I find out the install location of a plug-in?)

[4] článok Understanding Layouts in SWT
(http://www.eclipse.org/articles/Understanding Layouts/Understanding Layouts.htm)

[5] Dokumentácia k jazyku Java 1.5 (http://java.sun.com/j2se/1.5.0/download.jsp#docs)

[6] MCLAUGHLIN,BRETT. Java & XML,2^{nd} edition. O'Reilly, 2001.

Prílohy (v CD forme)

1. Plugin do Eclipsu aj so zdrojovými súbormi (Cieľ bakalárskej práce).
2. Návod na inštaláciu pluginu a jeho používanie.
3. Vygenerovaná Java Documentácia v HTML forme.
4. Ukážkové príklady práce so súborovým XML modelom návrhového vzoru (PDF forma).
5. Vybrané uvedené zdroje. (článok Understanding Layouts in SWT[4], článok Ako pristupuvať k aktívnemu projektu?[2], článok Ako nájdem inštalačné miesto pluginu?[3])

.