VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

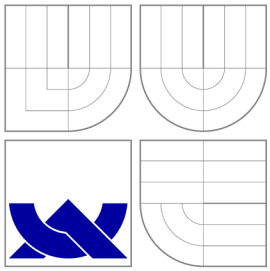
PROFILOVACÍ PERSPEKTIVA V PROSTŘEDÍ ECLIPSE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

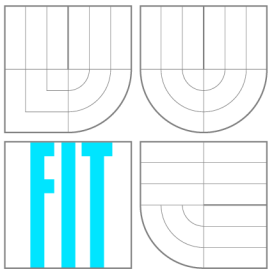
AUTOR PRÁCE
AUTHOR

JIŘÍ HYNEK

BRNO 2011



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

PROFILOVACÍ PERSPEKTIVA V PROSTŘEDÍ ECLIPSE

PROFILING PERSPECTIVE IN ECLIPSE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

JIŘÍ HYNEK

VEDOUCÍ PRÁCE
SUPERVISOR

prof. Ing. TOMÁŠ HRUŠKA, CSc.

BRNO 2011

Abstrakt

Tato bakalářská práce se zabývá problematikou tvorby plug-inu pro vývojové prostředí Eclipse. Výsledkem práce je pohled určený pro profilovací perspektivu vývojového prostředí projektu Lissom založeného na Eclipse. Úkolem pohledu je vykreslovat call grafy definované údaji v XML struktuře.

Abstract

This bachelor's thesis is concerned by question of creation a new plug-in for software development enviroment Eclipse. Product of this work is the view intended for profiling perspective of software development enviroment of project Lissom based on Eclipse. The aim of the view is draw call graphs defined by data from XML structure.

Klíčová slova

call graf, Eclipse, perspektiva, plug-in, pohled, profiler, SWT, XML, Zest, zoom

Keywords

call graph, Eclipse, perspective, plug-in, profiler, SWT, view, XML, Zest, zoom

Citace

Jiří Hynek: Profilovací perspektiva v prostředí Eclipse, bakalářská práce, Brno, FIT VUT v Brně, 2011

Profilovací perspektiva v prostředí Eclipse

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana profesora Tomáše Hrušky.

.....
Jiří Hynek
12. května 2011

Poděkování

Rád bych poděkoval celému týmu Lissom za možnost navštěvovat semináře. Z tohoto týmu především děkuji panu konzultantovi – Karlu Masaříkovi. Zvláštní poděkování rovněž patří panu Odřeji Ilčíkovi za odbornou pomoc s vývojovým prostředím Eclipse.

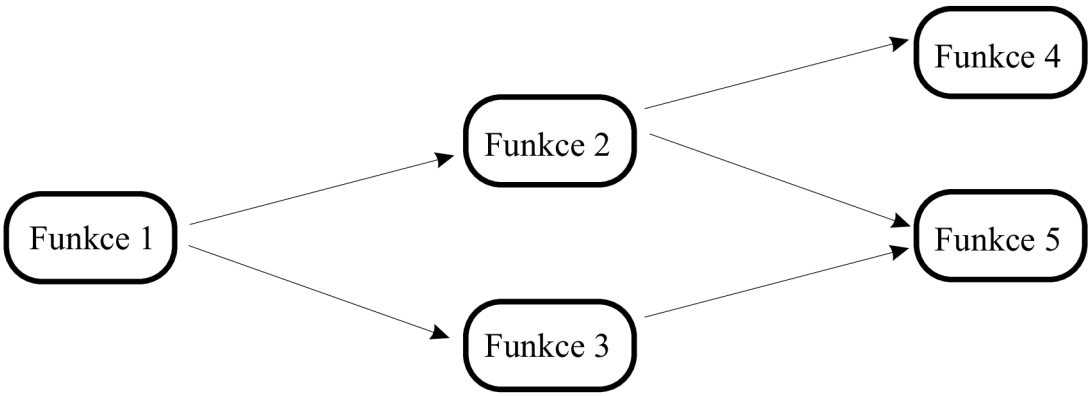
Obsah

Kapitola 1

Úvod

Pro kvalitní a efektivní programovaní jsou potřeba i kvalitní nástroje. Jedním z takových nástrojů je *profiler*. Jedná se o program, který bývá součástí vývojových prostředí. Jeho činností je analyzovat a vyhodnocovat kód programu, který programátor vytvořil. Zároveň sleduje jeho běh a podává programátorovi informace o časové náročnosti a struktuře vyvíjeného softwaru. Výsledkem analýzy jsou statistické informace, s jejichž pomocí vývojář může provádět optimalizace vedoucí k urychlení programu.

Cílem mé bakalářské práce bylo rozšířit profiler. Konkrétně se jedná o profilovací perspektivu vývojového prostředí, které je využíváno vývojovým týmem Lissom pro kvalitní návrh a realizaci mikroprocesorů [11]. Vývojové prostředí projektu Lissom je založeno na prostředí Eclipse, které nabízí nástroje pro snadné rozšíření a jeho přizpůsobení pro vlastní potřeby. Mojí prací proto bylo nejprve vytvořit plug-in pro Eclipse a až poté ho začlenit do prostředí projektu Lissom.



Obrázek 1.1: Příklad call grafu

Mé rozšíření má za úkol zobrazovat *call grafy*. Call graf se používá k vizualizaci toku programu. Lze ho definovat sjednocením množiny uzlů N (*Nodes*) a množiny spojení C (*Connections*). Každý uzel zobrazuje právě jednu funkci z analyzovaného programu. Spojení reprezentuje volání funkce, tedy je to relace (m, n), kde $m, n \in N$. Můj plug-in množiny N a C analyzuje a následně je zobrazuje uživateli. Výstupem této práce tedy bude pohled zobrazující call grafy, jejichž popis bude uložen ve stanovené XML struktuře vývojového prostředí projektu Lissom.

1.1 Etapy práce

V tomto dokumentu jsem postupně popsal jednotlivé etapy své práce. Nejprve jsem nastudoval vývojové prostředí Eclipse. Bylo nutné pochopit, jak toto prostředí rozšířit o další doplněk. Tyto informace lze vyčíst v kapitole 2. Čtenář se v ní dozví, jak si vytvořit v prostředí Eclipse vlastní plug-in – konkrétně pohled (*View*).

Eclipse je napsán v jazyce Java a využívá grafickou knihovnu Standard Widget Toolkit, která byla napsána přímo pro toto vývojové prostředí. Při vytváření plug-inů je doporučeno ji využít, proto se kapitola 3 zabývá tímto toolkitem.

Kapitolu 4 jsem věnoval samotné tvorbě plug-inu, který umožňuje zobrazovat call grafy. Jedná se již o konkrétní práci zabývající se vyvíjeným rozšířením, tudíž je této kapitole věnováno největší místo. Na úvod jsem v bodech popsal požadavky, které jsem si dal za cíl, aby můj plug-in splňoval. V následujících sekcích jsem je pak podrobně rozepsal.

Do tohoto textu bylo třeba použít mnoho názvů tříd a metod. Pro přehlednost jsem je znázornil `strojovým` písmem. Názvy a klíčová slova písmem *sklopeným*. Pro zvýraznění bodů ve výčtech jsem použil **tučný** řez. V ostatních věcech jsem se držel zásad stanovené šablony VUT FIT pro systém LaTeX.

Kapitola 2

Eclipse

Mým úkolem byla implementace plug-inu pro vývojové prostředí Eclipse. První věcí potřebnou pro splnění bakalářské práce bylo seznámení se s tímto vývojovým prostředím. Je to obsáhlé téma, přičemž pro tuto práci je podstatná pouze jeho část. Z toho důvodu jsem v této kapitole popsal pouze poznatky a informace podstatné pro implementační část mé bakalářské práce. Vycházel jsem přitom z knihy *Eclipse, Building Commercial-Quality Plugins* od pánů Clayberba a Rubela [1], která se zabývá především tím, jak rozšířit Eclipse o nový plug-in. Tato kniha stejně tak, jako většina dokumentací a textů, kterou jsem ke studiu použil, byla napsána v anglickém jazyce. Důsledkem toho bylo, že jsem ve vývojovém prostředí z důvodu přehlednosti rovněž pracoval s originálním jazykovým nastavením a neinstaloval žádnou lokalizaci. V následujícím textu proto budu často uvádět původní anglické názvy.

2.1 O prostředí

Eclipse je rozšířené open source vývojové prostředí původně vyvinuté firmou IBM. Je napsáno v jazyce Java a aktuálně se o jeho rozvoj stará nevýdělečná organizace Eclipse Foundation.

Jeho základní předností je snadná rozšiřitelnost v podobě plug-inů, na kterých je založeno. Nabízí proto podporu pro jejich vývoj v podobě PDE (*Plug-in Development Enviroment*), což je prostředí, které poskytuje nástroje pro tvorbu, testování a debugování plug-inů. Díky plug-inům se z Eclipse stává nástroj, který se dá dobře přizpůsobit požadavkům vývojaře či vývojového týmu. Nabízí velký výběr volitelných nastavení, které je možné migrovat z jednoho prostředí na druhé. Je multiplatformní, není tedy problém ho provozovat na celé řadě operačních systémů.

2.2 Workbench

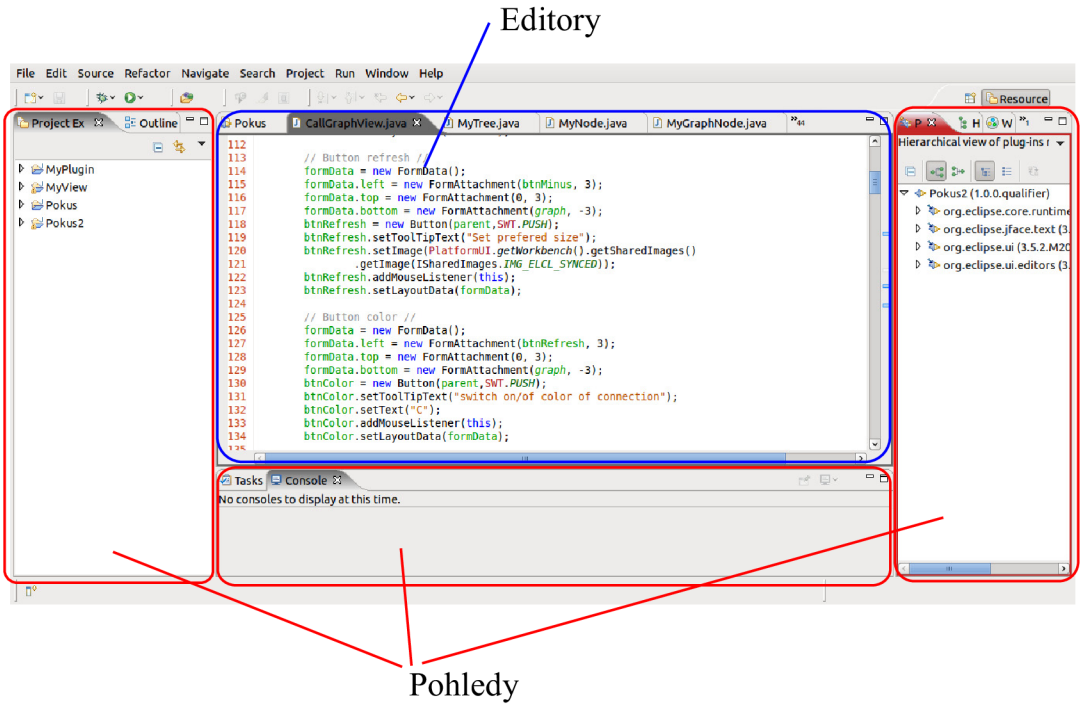
Workbench je standardní název pro pracovní prostředí. To se v Eclipse skládá ze dvou základních typů oken – z pohledů (*View*) a editorů (*Editor*).

Editor je okno určené k editaci zdrojových kódů. Může se jednat o klasický textový editor, do kterého se zapisuje přímo zdrojový kód, nebo se může jednat o specializovaný formulář k editaci určitých částí kódu. Toto okno je k nalezení ve středu pracovního prostředí. Je přitom možné mít otevřeno více editorů zároveň a přepínat mezi nimi.

Samotný editor umí díky plug-inům spoustu věcí, které dokáží výrazně usnadnit programátorovi práci – od zvýrazňování syntaxe přes refactorování (hromadné opravování kódu) až po různé nápovědy klíčových slov, metod a proměnných. To ovšem nebývá pro vývojáře, který se potřebuje orientovat ve zdrojových kódech a získávat informace ohledně překladu, dostačující. Z toho důvodu jsou k dispozici pohledy.

Pohled je další typ okna. Neslouží již k editaci zdrojových kódů, nýbrž k zobrazování pomocných informací, které ze zdrojových kódů může získat. Je používán především k navigaci ve stromové struktuře projektu, k zobrazení použitých metod, tříd a proměnných a například k zobrazení jejich hierarchického uspořádání. Pohledy jsou použity rovněž pro informace o překladu a pro nápovědu. Umístění pohledů není pevně stanoveno. Jsou zpravidla ukotveny okolo editorů. Uživatelé je mohou přemísťovat a seskupovat podle potřeb, případně i vyjmout ze základního pracovního prostředí a zobrazit jako samostatná okna.

Eclipse obsahuje velké množství pohledů. Aktivní jsou ovšem pouze některé v závislosti na tom, jakou práci vývojář právě dělá. Tím se dostávám k perspektivám (*Perspective*), což jsou seskupení editorů a pohledů. Každá perspektiva má svůj účel. Například pro implementaci nového programu slouží *Resource perspective*, která se soustřeďuje především na zobrazení navigace v hierarchii zdrojových kódů. Existuje celá řada dalších perspektiv (debugovací, pro týmovou synchronizaci, atd...). Je zároveň možné si vyrobit vlastní perspektivu [6].



Obrázek 2.1: Pracovní prostředí v Eclipse

Mým úkolem bylo rozšíření perspektivy o funkci zobrazování call grafů. Můj plug-in má za úkol data zobrazovat, nikoliv editovat. Musel jsem tedy zvolit variantu implementace nového pohledu. Problematika samotného vytváření plug-inu bude probrána v kapitole 4 na straně 19.

2.3 Infrastruktura

Eclipse není monolitický program. Plug-iny v něm hrají velkou roli, jelikož samotné vývojové prostředí představuje pouze malé jádro. To slouží jako zavaděč plug-inů (tzv. *plug-in loader*). Každý plug-in má na starost konkrétní činnost a může při ní využívat služeb jiných plug-inů. Vzniká tím rozsáhlá síť závislostí. A právě tuto síť musí zavaděč plug-inů sestavit při jeho spuštění.

Bylo by velmi nepraktické a hlavně náročné a pomalé, kdyby zavaděč musel procházet kódy jednotlivých knihoven a zkoumat v nich jednotlivá volání metod cizích doplňků. Každý plug-in proto obsahuje konfigurační soubory s metainformacemi, které stručně a přesto účelně popisují vlastnosti každého doplňku a zároveň také, jaké další doplňky vyžaduje. Zavaděč při spuštění projde tyto informace a vytvoří z nich strukturu závislostí. Díky konfiguračním souborům nemusí načítat jednotlivé plug-iny a zkoumat jejich chování.

Pomocná struktura je nahrána do paměti a zabírá tam místo. Je to ovšem efektivnější řešení, než-li načítat jednotlivé plug-iny, aniž by musely být aktuálně potřeba. Toto je základní logika Eclipse a nazývá se *lazy loading*, což by se dalo v češtině vyjádřit jako „líné nahrávání“.

2.4 Metainformace plug-inu

V sekci 2.3 jsem vysvětlil, že metainformace plug-inů hrají v Eclipse významnou roli. Vývojář doplňků se v této oblasti musí umět dobře orientovat, proto jsem pro ni vyčlenil speciální sekci.

Každy plug-in musí obsahovat dva základní soubory. Jsou jimi `META-INF/MANIFEST.MF` a `plugin.xml`:

- `META-INF/MANIFEST.MF` je textový soubor, který obsahuje základní informace o doplňku, jako jsou například unikátní identifikátor, verze a *aktivátor* (základní třída, kterou plug-in začíná).
- Druhý – `plugin.xml` je strukturovaný XML soubor, který popisuje strukturu doplňku, jaká rozšíření z jiných plug-inů využívá a zároveň jaká rozšíření poskytuje. Je možné v něm najít například seznam jednotlivých pohledů, který daný plug-in obsahuje.

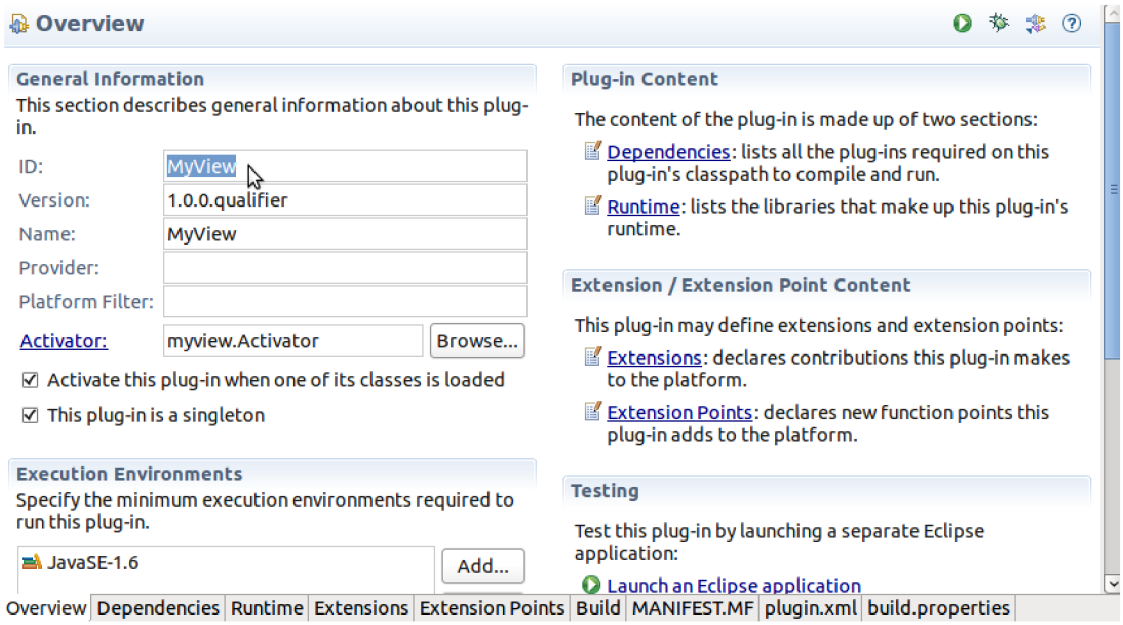
Neměl bych opomenout ještě jeden soubor. Jeho název je `build.properties` a říká mi, jaké soubory budou obsaženy ve výsledném archivu. Narozdíl od prvních dvou souborů, tento není nezbytný pro spuštění plug-inu. Je to soubor, který se podílí na překladu.

Konfiurační soubory se dají editovat přímou úpravou zdrojového kódu, ovšem je třeba dbát na jejich správný formát. Jejich špatná implementace by mohla mít fatální následky. Existuje proto formulářový editor, v kterém lze pohodlně vyplnit všechny potřebné údaje těchto souborů. Tento editor je pro přehlednost členěn do karet, kde každá karta reprezentuje část informací s vlastnostmi týkajících se daného tématu. V následujících podsekcích jsou postupně rozepsané.

2.4.1 Overview

Jedná se o úvodní kartu, která shrnuje základní atributy plug-inu. Část údajů je volitelná, jako jsou například informace o dodavateli a aktivační třídě (`Activator`). Povinné položky jsou ID, jméno a verze produktu.

- **ID** je unikátní identifikátor, který využívá konvenci pojmenovávání z Javy. Profesionálně se tedy plug-inům dávají identifikátory ve tvaru například `com.<jméno společnosti>.<produkt>.<část>`. Není to ovšem nezbytnost.
- **Verze produktu** je další důležitý atribut, zvláště pokud vznikají často nové verze. Skládá se ze tří čísel, kde číslo nejvíce vlevo má největší váhu (např. `1.0.0`). Tento údaj je potřebný zejména, kdy jeden plug-in využívá druhého a je stanoveno, které verze druhého jsou s prvním kompatibilní.
- Parametr **aktivační třídy** (`Activator`) reprezentuje odkaz na třídu, na kterou se jako první systém odkáže při spuštění plug-inu. Její roli rozebírám podrobněji v podsekci 2.5.1.



Obrázek 2.2: Ukázka karty Overview ve formulářovm editoru s pokusným pohledem `MyView`

Karta obsahuje ještě další parametry, jako například definici, zda se plug-in načte celý, pokud bude volána pouze jedna jeho třída, apod... Na úvod ovšem nejsou tolik podstatné. Informace jsou reprezentací části souboru `META-INF/MANIFEST.MF`.

2.4.2 Dependencies

V této kartě je možné zadat, jaké jiné plug-iny bude můj doplněk využívat. Tyto deklarace se nepoužívají při kompilaci, jako je tomu u deklarací knihoven v Javě, kdy se při překladu nakopírují do výsledného archivu. Informace jsou potřebné až při spouštění plug-inu. Špatné zadání může vést k výjimce `NoClassDefFoundError`.

Existují dva typy těchto závislostí. K tomu jsou poskytnuty dva formulářové sloupce. Do prvního se zadávají všechny plug-iny, které jsou vyžadovány pro spuštění, v druhém jsou vyjmenovány ty, které plug-in vyžaduje, nicméně se bez nich při spouštění obejde. Takové knihovny mohou být potřeba až při vyvolání určité operace, která s nimi přímo souvisí.

Dále je možné si u jednotlivých knihoven nastavit vynucení verze. Zadefinovat můžeme přesnou verzi nebo určitý rozsah verzí. Například značení `[3.0.0,3.1.0)` sděluje, že kompatiblní knihovna je od verze `3.0.0` (včetně) do verze `3.1.0` (která už do rozsahu ovšem nepatří).

Informace v této kartě jsou rovněž obsaženy v souboru `META-INF/MANIFEST.MF`.

2.4.3 Runtime

Zde můžu určit, jaké balíky bude poskytovat pro změnu můj plug-in jiným plug-inům.

2.4.4 Extensions a Extension points

Karta *Extensions* je během vytváření plug-inu velmi využívaná. Uvádí se zde totiž, jak vyvíjený plug-in využívá rozšíření jiných plug-inů. Oproti kartě *Dependencies* (2.4.2) je zde podstatný právě způsob rozšíření, ne seznam knihoven. Je možné zde deklarovat nové pohledy, editory, akce, atd...

Pátá karta *Extension points* sděluje to samé s tím rozdílem, že v opačném smyslu. Je možné nastavit dostupná rozšíření, která poskytne vyvíjený plug-in.

Obě karty reprezentují informace ze souboru `plugin.xml`.

2.4.5 Build

Jedná se o poslení formulář, který tentokrát zobrazuje dodatečný soubor `build.properties`. Jednoduchým označením se zde dají vybrat položky (zdrojové soubory), které mají být obsaženy do výsledného archivu. Lze rozlišovat mezi položkami, které budou použity pro překlad a které budou ponechány jako zdrojový kód.

2.4.6 Konfigurační soubory

Na konci v posledních třech kartách jsou dostupné zdrojové konfigurační soubory se zvýrazněnou syntaxí. Zde je možné je upravovat přímo, nicméně není to moc doporučované a sám jsem toho nevyužil, jelikož by se mi tím mohlo zvýšit riziko chyby.

2.5 Vytváření plug-inu

Po nastudování informací ohledně vývojového prostředí jsem vytvořil jednoduchý plug-in určený pro zkušební účely. K tomu bylo třeba vytvořit nový projekt, který byl typu *Eclipse plug-in*. To je v Eclipse možné provést pomocí intuitivního průvodce, který umí sám automaticky nagenerovat spoustu zdrojového kódu v závisloti na zvolených požadavcích a ušetřit tím programátorovi práci. Já jsem zvolil variantu pluginu s pohledem, jelikož moje práce se ubírala právě tímto směrem.

Ne vždy se ovšem tato vymoženost musí hodit. Je dobré umět se vyznat v nagenerovaných částech, proto jsem se rozhodl další sekce věnovat tomu, jak se dají jednotlivé části pluginu vytvářet ručně.

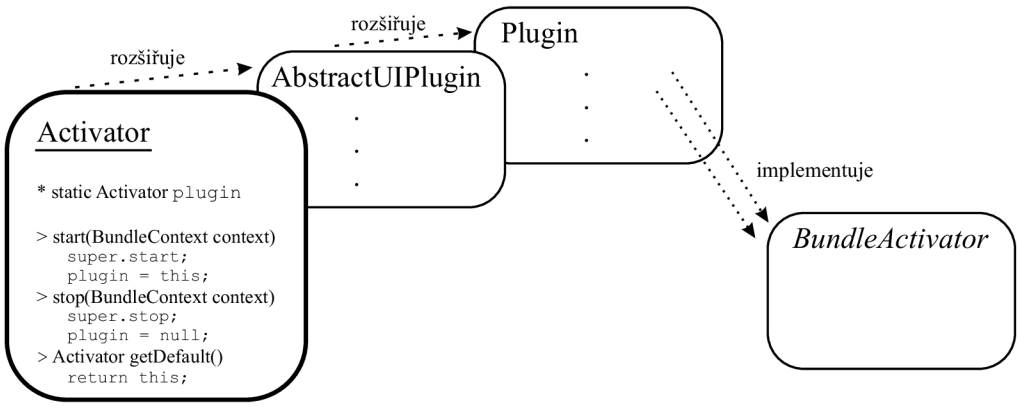
2.5.1 Struktura plug-inu

Každý plug-in by měl obsahovat základní třídu (tzv. `Activator`), na kterou se zavaděč může odkázat z metainformačního souboru `META-INF/MANIFEST.MF`. Tato třída poskytuje

metody pro přístup ke zdrojům používaným v plug-inu. Je to první třída, na kterou se systém odkáže při aktivaci plug-inu a následně pak vytvoří její instanci. Během jejího běhu nemůže být vytvořena žádná jiná instance této třídy. Často se proto do konstruktoru implementuje testování statické proměnné `plugin`, v které je uložena reference na instanci třídy `Activator`. Pokud v době volání konstruktoru ukazuje na nějaký objekt, vygeneruje se výjimka `IllegalStateException`.

Další podmínkou je, že plug-in bude implementovat rozhraní `BundleActivator`. To je zajištěno rozšířením třídy `Plugin` (resp. `AbstractUIPlugin`). Zde záleží na tom, zda plug-in bude obsahovat uživatelské rozhraní, nebo ne. Jelikož mým úkolem bylo implementovat nový pohled, zvolil jsem, že moje aktivační třída `Activator` bude rozšiřovat třídu `AbstractUIPlugin`[1]. Tato třída zajišťuje ukládání nastavení mého plug-inu (jako je například pozice pohledu) po zavření.

Dále je nutné implementovat metody `start` a `stop`. Přes tyto metody zavaděč plug-inů otevírá a zavírá dané plug-iny. Rodičovská třída `AbstractUIPlugin` nám je obě poskytuje a není tedy vhodné je nijak zásadně přepisovat. Vždy je nutné zavolat příslušné rodičovské metody klíčovým slovem `super` a až poté dopsat vlastní kód. Já do každé z nich dopsal jeden příkaz navíc. V metodě `open` přiřazuji referenci na novou instanci do statické proměnné `plugin`. V metodě `close` ji nastavuji na `NULL`. Srozumitelněji je to znázorněno na obrázku 2.3.



Obrázek 2.3: Zjednodušená ukázka struktury třídy `Activator`

Plug-in je spouštěn vždy, když si to uživatel vyžádá (pokud ještě nebyl otevřen a není již zapamatován v profilu uživatele). Je to základní vlastnost mechanismu *lazy loading*. Pro vynucení otevírání lze přidat rozšíření (*extension* – 2.4.4) `org.eclipse.ui.startup` a implementovat jeho rozhraní `IStartup`. To nám poskytuje metodu `earlyStartup` určenou pro časné otevírání.

2.5.2 Přidání pohledu

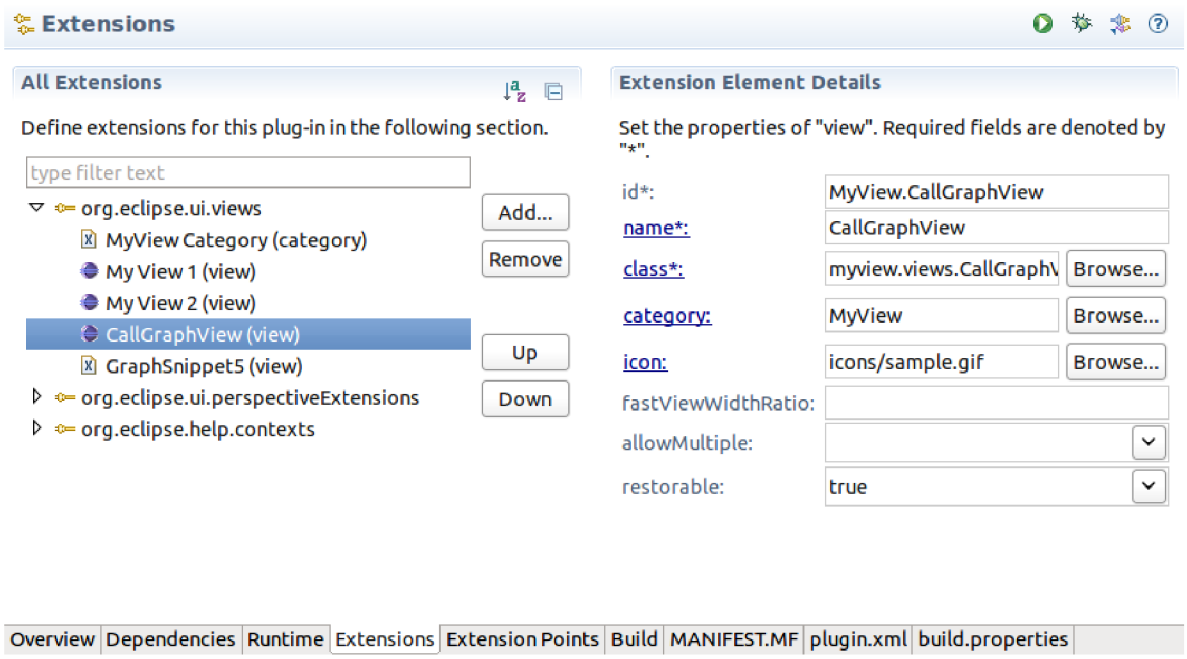
Jakmile je plug-in vytvořen, je ten správný čas přidat nový pohled. První věcí, kterou pro to bylo třeba udělat, bylo přidat nové rozšíření `org.eclipse.ui.views` v konfigurační kartě

[1] Je třeba rovněž mít v kartě závislostí přidán balík `org.eclipse.ui`, kterému třída náleží.

Extensions (2.4.4). Tím jsem dal najevo, že můj plug-in může obsahovat pohledy[2]. Samotné zvolení patřičného rozšíření ovšem nestačí. Je nutné zadeklarovat požadovaný pohled. To se provádí rovněž ve vybrané položce rozšíření. K dispozici je formulář, v kterém se musí vyplnit povinné atributy nového pohledu jako jsou ID, jméno a třída obsahující zdrojový kód pohledu:

- **ID** je unikátní identifikátor, u kterého je vhodné používat strukturování tak, jako u identifikátoru plug-inu v kartě *Overview* (<`id plug-inu`>.<`jméno pohledu`>).
- Stejně tak, jako se zadává aktivační třída plug-inu, se musí zadat **třída příslušící danému pohledu**. Já jsem si vytvořil ve jmenném prostoru `views` novou třídu `CallGraphView`, kterou jsem pro tento účel použil.
- Dále je možné přidat spoustu nepovinných parametrů. Je dobré zadat **ikonu pohledu**, která nám bude odlišovat náš pohled od ostatních.
- Vyplatí se vyplnit i položku **Category**. Jelikož je v prostředí Eclipse dostupných mnoho pohledů, je třeba je nějak třídit, aby se v nich dalo vyznat. Programátor může zvolit ze sady již vytvořených kategorií nebo si může vytvořit vlastní. To se provede přidáním nové položky typu *Category* pod rozšířením `org.eclipse.ui.views` v kartě *Extensions* (stejně tak jako u nového pohledu).

Po přidání pohledu v konfigurační kartě bylo třeba mít správně naimplementovanou třídu `CallGraphView`, čímž se dostávám k druhé části.



Obrázek 2.4: Formulář pro přidání nového pohledu

Každý pohled musí implementovat rozhraní `IViewPart` z balíku `org.eclipse.ui`. Toho jsem docílil tím, že jsem sdědil v mém pohledu třídu `ViewPart` z balíku `org.eclipse.ui`.

[2]Jednotlivá rozšíření lze jednoduše vybírat ze seznamu, takže jsem si zde mohl ověřit, zda mám všechny balíky dobře nainstalované.

`part`, která toto rozhraní implementuje. Nakonec mi zbývalo implementovat metodu `createPartControl`. Je to hlavní metoda, která vytváří obsah pohledu. Do této metody jsem naimplementoval jednotlivé komponenty (jako například instanci obsahující call graf).

2.5.3 Spuštění

Po splnění předchozích kroků jsem ověřil, zda se mi povedlo vytvořit zkušební pohled. K tomu Eclipse nabízí šikovný způsob. Volbou *Run As Eclipse Application* se spustí nové okno s aplikací Eclipse, která v sobě obsahuje nově vytvořený plug-in. V seznamu pohledů je možné najít vytvořený pohled, spustit ho a vyzkoušet jeho funkčnost tak, jako bych ho měl již nainstalovaný.

Kapitola 3

Standard Widget Toolkit

Dalším bodem mé práce bylo prostudování grafické knihovny, kterou Eclipse využívá. Jak již nadpis napovídá, vývojové prostředí je založené na toolkitu SWT. Výhodou je, že bylo navržené účelně pro prostředí Eclipse a stejně tak je to open source, který je veden pod licencí EPL (*Eclipse Public Licence*).

Byl navrhnut firmou IBM. Při jeho vývoji se programátoři zaměřili především na nevýhody, které obsahují knihovny AWT (*Abstract Windowing Toolkit*) a JFC (*Java Foundation Classes*), více známá jako *Swing*. První byl vyvinut toolkit AWT. Jeho velkým nedostatkem je, že obsahuje pouze základní grafické komponety (tlačítka, jednoduchá textová pole, atd. . .), což se postupem času stalo nedostačující. Proto byla vyvinuta firmou Sun knihovna Swing. Tento toolkit již obsahuje velké množství nástrojů k vytváření kvalitních programů v jazyce Java. Jeho typickou vlastností je, že obsahuje vlastní *look and feel*. Na různých operačních systémech tedy bude vypadat stejně, což je ne vždy bráno jako výhoda.

Vývoj SWT se vydal jiným směrem. Cílem bylo vytvořit jednoduchý toolkit, který by co nejvíce využíval systémovou vrstvu pro vykreslování oken a zároveň poskytoval dostačující soubor komponent pro pohodlné vytváření okenních aplikací. Narozdíl od Swingu, který si všechny okna vykresluje vlastním stylem, SWT používá nativní vzhled operačního systému. Zároveň se vývojáři snažili, aby byl štíhlejší a výkonnější než Swing. Při implementaci navázali na poznatky z toolkitu CW (*Common Widget*), který sloužil pro jazyk Smalltalk.

Do této kapitoly jsem opět uvedl pouze informace důležité pro moji práci. Zmínil jsem se o třídách `Display` a `Widget`, které tvoří pilíře SWT. Popsal jsem základní grafické komponenty, které je užitěčné znát pro vytváření programů. Další důležitou věcí, kterou jsem musel nastudovat, bylo rozmístění prvků v okně. Rozepsal jsem zde především `FormLayout`, který jsem použil v mém pohledu. Celou kapitolu jsem pak zakončil knihovnou JFace, která poskytuje náročnější nástroje pro tvorbu komponent. Poznatky jsem čerpal ze zmíněné knihy o vývoji plug-inů [1], dále jsem objevil zajímavou knihu věnovanou SWT od pánů Northovera a Wilsona [5], která se podrobně zabývá tímto tématem. Zde jsem čerpal z první kapitoly, která je volně dostupná na internetu[1].

3.1 Třída Display

Každá aplikace, která je založená na SWT, musí obsahovat právě jednu instanci třídy `Display`. Tato třída totiž zajišťuje spojení mezi SWT a spodní systémovou vrstvou vykreslování oken. Poskytuje zároveň aplikační rozhraní pro:

[1] http://my.safaribooksonline.com/9780321256638

- **event loop** (nekonečná smyčka, která čeká na zprávy (resp. události) a zároveň je obsluhuje),
- **časovače** (timery),
- **systémové zdroje**, jako jsou barvy, fonty, atd...

Typický *event loop* v SWT aplikacích, kde `shell` představuje okno aplikace a `display` je instance třídy `Display`, vypadá následovně:

```
while(!shell.isDisposed()) {
    if(!display.readAndDispatch()) display.sleep();
}
```

Metoda `readAndDispatch` čte v cyklu události z fronty a přeposílá je patřičnému příjemci. Pokud je fronta událostí prázdná, `display` přejde do stavu *sleep*. Jakmile je aplikace ukončena (okno je zrušeno), smyčka skončí. Za ní už bývají pouze metody `dispose`, které uvolní využívané prostředky (například zmiňovaný `display`).

Instance třídy `Display` vzniká při spuštění programu a zaniká při jeho ukončení. O její vytvoření a rušení se musí starat programátor. Protože jsem vytvářel plug-in pro Eclipse (již běžící program), nemusel jsem se o třídu `Display` a smyčku událostí starat.

3.2 Třída Widget

Další důležitou třídou je třída `Widget`. Je to abstraktní třída, z které jsou odvozeny všechny základní grafické prvky používané pro interakci s uživatelem. Třída `Widget` jim poskytuje základní metody, jako je například chování při událostech, apod... Nejdůležitější informací, kterou komponenta nese, je její stav. Pokud je tento stav změněn (většinou uživatelem), překreslí se.

3.2.1 Vytváření komponent

Komponenty jsou vytvářeny programátorem. Jejich typickými parametry v konstruktoru jsou `Widget parent` a `int style`.

- Prvním zmíněným parametrem je rodičovská komponenta. Celý grafický návrh aplikace totiž tvoří jasně danou hierarchii, kde každá komponenta je součástí jiné komponenty. Je přesně dané, jaká komponenta smí být pro zvolenou komponentu rodičovská. Například položka menu (`MenuItem`) může být „posazena“ výhradně do widgetu `Menu`. Na vrcholu stromu pak bývá hlavní okno (`Shell`), který musí mít zadefinovanou instanci třídy `Display`.
- Dalším parametrem je číselná hodnota, která určuje styl komponenty. Jako příklad uvedu často používanou komponentu – tlačítko (`Button`). Pokud mu v konstruktoru nastavím vhodný styl, můžu ho změnit na specifické tlačítko (například `CheckBox`, či `RadioButton`). Hodnoty vzhledů přitom získám z knihovny SWT, která nabízí sadu konstant určených nejenom pro styly. Styly je přitom možné kombinovat. Každá komponenta má definovanou množinu stylů[2] a jejich možné kombinace. Ke kombinaci více stylů slouží operátor `OR`.

[2] Je dobré vždy nahlédnout do manuálu před použitím nové komponenty.

3.2.2 Rušení komponenty

Jakmile prvky nejsou potřeba, je rozumné je zrušit, aby nám zbytečně nezabíraly místo v paměti. Narozdíl například od Swingu se o toto musí starat programátor sám. Ke správnému rušení se přitom stačí držet následujících dvou pravidel:

- *Vše, co si sám vytvořím, musím i zrušit.* Každý prvek, který není použitý ze systémových zdrojů (přes třídu `Display`), je třeba zrušit metodou `dispose()`, pokud již není potřeba. Toto se týká především vlastních fontů a barev.
- Druhé pravidlo říká, že *zrušením komponenty zruším i všechny jeho potomky.* Rodič nám tedy poslouží i k této věci a je vhodné mít přehled o hierarchii komponent, kterou jsme v programu vytvořili[3].

3.2.3 Události

Samotné komponenty moc užitěčné nejsou, dokud jim programátor nezadá úkoly, které budou vykonávat. K tomu slouží stejně tak jako u jiných grafických knihoven události (*Widget events*) a ovladače těchto událostí (*Listeners*). Každému prvku můžu přiřadit skupinu těchto listenerů, přes jejichž metody mu mohu zadefinovat chování. To se vykoná při události, se kterou je daný listener spjat. Pro příklad je možné si představit tlačítko, jemuž je přiřazen listener na stisk myší, který obsahuje kód otevření okna s informační zprávou.

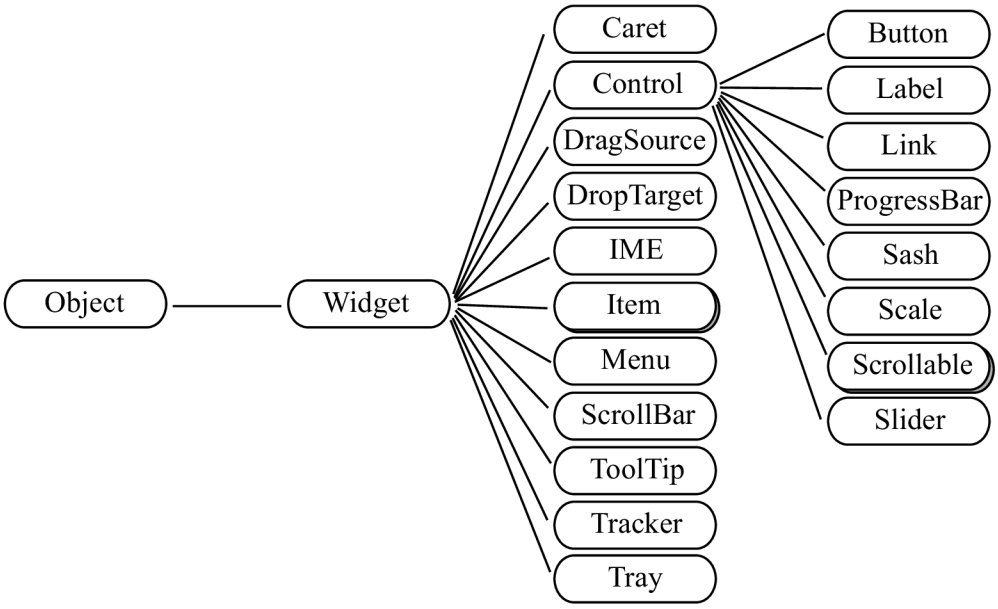
V SWT je dostupná celá řada různých událostí. Jejich výčet je velmi podobný jako například v knihovně AWT. Tomuto tématu proto nebudu věnovat zvláštní pozornost. Pro přehled je možné nahlédnout do manuálů k SWT [7].

3.2.4 Vybrané komponenty

Jak bylo sděleno v úvodu této sekce, každá komponenta dědí vlatnosti třídy `Widget`. Celkový strom je znázorněn na obrázku 3.1. Nejpoužívánější odvozenou třídou je třída `Control`. Z ní jsou totiž odvozeny všechny základní komponenty jako jsou tlačítka, rámečky, textová pole a především také komponenta `Shell`. Tato komponenta představuje rám okna aplikace, s kterým uživatalé pracují. V hierarchii *rodič-potomek* také bývá posazen nejvýše a do něho pak dále bývají zasazeny další komponenty. `Shell` se stará o jejich rozmístění, kterým se podrobněji zabývám v následující sekci 3.3. Dalším užitečným prvkem, který slouží jako konteiner pro další komponenty, je `Composite`. Rovněž jemu je možné zadefinovat layout.

Existují i další prvky, které nejsou odvozeny od třídy `Control`. Náleží jim vlastní třídy na stejné úrovni. Je to způsobeno tím, že je tak reprezentuje operační systém. Jsou jimi například menu, posuvné lišty (`ScrollBar`), kurzor (`Caret`), atd... Podrobné rozepisování jednotlivých komponent by bylo plýtvání místem. Jejich popis i s náhledem je možné najít na stránkách SWT [8]. Spousta užitečných tutoriálů je rovněž k vidění na webové stránce `java2s.com` v sekci SWT [3].

[3] Existují i jiné vztahy jako *rodič-potomek*, kde se toto pravdilo rovněž uplatní. Jsou jimi kaskádová menu, kde jedno menu, obsahuje další menu a vyskakovací okna (*PopUp*), kde se o jeho zrušení postará příslušná komponenta, které náleží.



Obrázek 3.1: Hierarchické uspořádání komponent

3.3 Pozicování

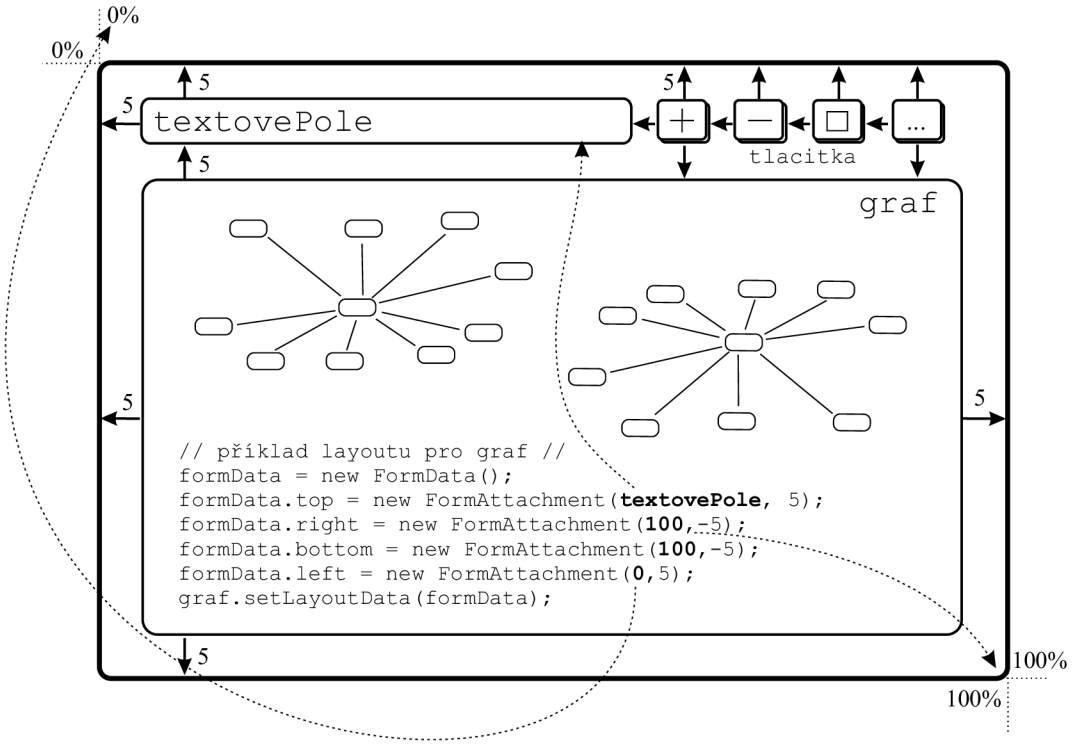
Každá komponenta je umístěna relativně k jeho rodiči. Základně vyplňuje celý jeho prostor. Pokud ovšem existuje více komponent, které mají stejného rodiče[4], musí se o dostupné místo rozdělit. To většinou nevytváří moc pěkný efekt, jelikož ve většině případů je na každou složku potřeba vymezit jiný prostor. Proto jsou v SWT stejně tak jako například ve Swingu dostupné *layouty*, které se starají o rozmístění komponent. Zde uvádím jejich výčet:

- Základní layout se jmenuje `FillLayout`. Snaží se objekty vyplnit celý prostor rodiče. Obsahuje dva základní styly – vertikální a horizontální. Tento layout je vhodné použít například pro řadu tlačítek, u kterých není nutné specifikovat podrobné rozmístění.
- O něco více specifický je `RowLayout`. Zde je již možné specifikovat vzdálenosti mezi jednotlivými komponentami a jejich zarovnání. Layout se tedy opět snaží vyplnit veškeré místo rodiče, ovšem už neskládá jednotlivé prvky přímo za sebe. Programátor se zde ovšem musí opět spokojit s dvěma styly (vertikálním a horizontálním) jako u `FillLayoutu`.
- Jelikož je potřeba často vytvářet různá okna s nastavením (dialogy, průvodce, ...), SWT obsahuje `GridLayout`. Je to layout určený především pro jednoduché formuláře, kde je jednodušší si plochu rozdělit na mříž. V nastavení je pak nutné zadefinovat počet sloupců a řádků a volitelně pak různé mezery a zarovnání.
- I `GridLayout` ovšem nemusí být dostačující, zvláště pokud bychom chtěli vytvářet okna s přesně určeným rozmístěním komponent. K složitému formátování v SWT existuje `FormLayout`. Je to nejpropracovanější layout, který SWT nabízí. Pomocí něho můžeme vytvořit libovolně formátované okno s komponentami. Nezadáváme zde

[4] V tomto případě si představte rodiče jako kontejner.

totiž vlastnosti celku, ale definujeme vlastnosti samotných objektů, které tento celek obsahuje. Jedná se o velikost (šířku a výšku) a vzdálenost od jiných komponent ze všech stran. Při vytváření okna s tímto rozmístěním je vhodné udělat si náčrtek.

Já jsem v mém pohledu rovněž využil síly `FormLayoutu`. První tři layouty jsou jednodušší na použití, pokud se ovšem programátor snaží vytvořit složitější rozmístění komponent, je vhodnější si vytvořit náčrtek a aplikovat `FormLayout`.



Obrázek 3.2: Ukázka použití FormLayoutu v mém projektu

Podrobnější popis layoutů je popsaný v knize o vývoji plugi-nů [1], případně v ilustrovaném článku stránkách `eclipse.org` [4].

3.4 JFace

V minulé sekci jsem popsal, že pro rychlé vytváření formulářů může posloužit `GridLayout`. Formuláře lze jednoduše vytvářet také pomocí knihovny JFace. Je to grafický toolkit, který se prolíná s knihovnou SWT a poskytuje sadu nástrojů pro jednoduchou tvorbu grafických prvků. JFace se snaží programátorovi ulehčit práci s rutinní prací. Ten se pak může více soustředit na samotný obsah jednotlivých částí místo jejich pracného sestavování z jednotlivých grafických komponent. Další výhodou je, že JFace zajistí programátorovi stejný styl, jako má Eclipse. Vyvíjený plug-in pak lépe zapadne do celého vývojového prostředí. K dispozici je následující sada nástrojů:

1. Jako první v abecedním výčtu uvádím **Actions and contributions**. Je to nástroj, který umožňuje měnit koncovému uživateli chování akcí, vyvolaných například tlačítky, bez nutnosti zásahu do kódu.

2. Dalším užitečným násrojem jsou **Dialogs and wizards**. Šikovným rozšířením těchto tříd lze snadno a rychle vytvořit dialogy a průvodce, které zajišťují interakci s uživatelem. Programátor přitom nemusí přemýšlet nad rozmístěním jednotlivých komponent.

3. **Field assist** je další užitečný nástroj, který je možné využít v textových polích určených pro získávání vstupu. Nabízí uživateli nápovědu o možných řetězcích.

4. **Image and font registries** poskytují třídy, které umí spravovat nové obrázky a fonty. Programátor přitom nemusí dbát na rušení těchto komponent, jak jsem se zmiňoval v kapitole 3.2.2. Tento mechanismus je využíván například při definování ikony pohledu, kterou jsem doporučoval při přidávání nového pohledu v podsekci 2.5.2.

5. Posledním a asi nejdůležitějším nástrojem balíku JFace jsou **Viewers**. Jedná se o třídy, které zajišťují správu nad komponentami, které je nutné uchovávat ve strukturách, jako jsou stromy, seznamy a tabulky. Poskytují propojení mezi uloženými hodnotami a jejich následným zobrazením. Díky tomu je zobrazená komponenta (například tabulka) synchronizována s uloženými daty. Dále poskytuje nástroje pro jejich řazení a filtrování.

Cílem mé práce nebylo podrobně studovat JFace. Podrobnější informace je možné nalézt v dokumentaci vývojového prostředí v sekci JFace [9].

Kapitola 4

Vyvíjený plug-in

Po nastudování vývojového prostředí Eclipse, grafického toolkitu SWT a přidání jednoduchého pohledu do vývojového prostředí jsem začal tvořit vlastní plug-in. Pro funkci programu jsem si musel stanovit požadavky. Mnoho z nich přitom vznikalo v průběhu schůzek s pány konzultanty v souladu s požadavky vývojářů týmu Lissom. Z tohoto důvodu jsem se musel snažit své vstupní podmínky upravit tak, aby co nejvíce odpovídaly požadavkům programátorů, pro které je určen.

Zde je k nahlédnutí již konečný seznam pažadavků na funkčnost programu, který přibližuje ve stručnosti postup mé práce:

1. **Zobrazované informace:**
   (a) Program bude přebírat stanovenou strukturu ve formátu XML a následně ji vizualizovat v grafu. Ten bude tvořen propojenými uzly reprezentující vzájemná volání.
   (b) Ke každému uzlu bude možné zobrazit doplňující informace stanovené ve struktuře, jako jsou například počty volání a výpadky vyrovnávacích pamětí.
   (c) Pokud uzel volá sám sebe, je potřeba zobrazit informaci pro každé volání zvlášť, jelikož v každém může být jiný počet výpadků paměti.
2. **Orientace ve grafu:**
   (a) Graf bude zobrazen tak, aby se jednotlivé uzly a spojení co nejméně překrývaly.
   (b) Bude možné jednoduše měnit pozice uzlů – přesouvat je.
   (c) Do pohledu bude zakomponovaný zoom, díky kterému bude programátor moci přibližovat a oddalovat graf.
   (d) Bude možné měnit pozici aktuálně zobrazeného pohledu – posouvat s celým grafem.
   (e) Programátor bude moci skrýt i expandovat volané funkce, což mu umožní procházení grafem.
   (f) Program bude obsahovat hledáček pro rychlé nalezení uzlu.
3. **Vzhled:**
   (a) Jednotlivé uzly budou rozlišeny v závislosti na jejich typu.

(b) Bude možné barevně zvýrazňovat spojení a měnit jejich styl v závisloti na uzlech, kterým náleží.

(c) Popisky uzlů bude možné schovávat.

(d) U rekurzivních uzlů budou speciální popisky s počtem rekurzivních volání.

4. **Uživatelské rozhraní:**

   (a) Při psaní do hledáčku bude k dispozici nápověda.

   (b) Program bude obsahovat toolbar a vyskakovací menu pro přívětivé uživatelské rozhraní.

5. **Integrace s profilovací perspektovou.**

Jednotlivé body se přitom vzájemně prolínají. V následujících sekcích a podsekcích budou postupně tyto požadavky vysvětleny.

4.1 Vizualizace informací

Na konzultacích bylo rozhodnuto, že není nutné psát novou grafickou knihovnu pro kreslení grafů, ale spíše se porozhlédnout, zda se již někdo něčím takovým nezabýval. Při hledání jsem objevil knihovnu GEF (*Graphical Editing Framework* [10]), která obsahuje toolkit *Zest*. Tento toolkit nabízí nástroje vhodné pro kreslení grafů, které se skládají z uzlů a spojení. Našel jsem tedy to, co bylo potřeba pro moji práci. Navíc GEF byl navrhnut přímo pro Eclipse a nyní je šířen pod licencí EPL (*Eclipse Public License*). Mohl jsem ho tudíž využít.

Práce s tímto nástrojem není složitá. Nejprve je potřeba vytvořit instanci třídy `Graph`. Ta slouží jako kontejner, který v sobě obsahuje jednotlivé komponenty – uzly (`GraphNodes`) a spojení (`GraphConnections`). Uzlům je možné zadat název, který bude zobrazen v jejich těle. Vytvořené uzly se propojí instancemi třídy `GraphConnection`. Těm je nutné v konstruktoru zadat zdrojový a cílový uzel. Spojení tyto dva uzly propojí.

Před samotným vykreslením je vytvořené instanci třídy `Graph` nutné zadat vykreslovací algoritmus. Ten se stará o počáteční rozmístění uzlů, což byla jedna z podmínek, kterou musel můj program splňovat. Nejedná se o triviální problém, jelikož je třeba rozmístit uzly tak, aby se co nejméně přerývaly a tvořily tím co nejméně chaotický celek. Toolkit Zest sadu vykreslovacích algoritmů nabízí. Já toho využil a do plug-inu jsem to implementoval způsobem, aby bylo možné za běhu mezi jednotlivými zobrazeními přepínat. Pro různé struktury se totiž může hodit jiný algoritmus:

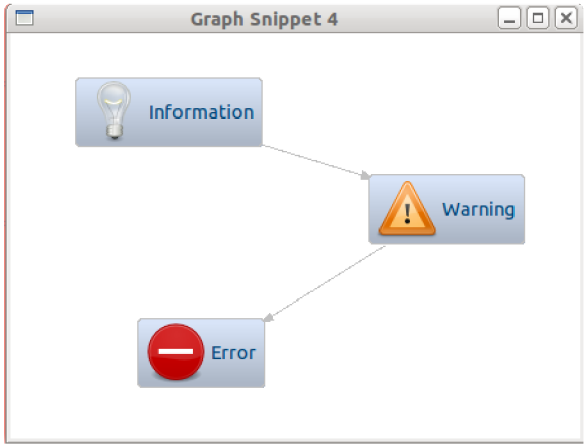
- `TreeLayoutAlgorithm` zobrazí uzly do stromu. Tento algoritmus existuje ve vertikální i horizontální verzi. Obě verze vytvoří ve většině případů nejžádanější rozmístění, jelikož struktura toku programu (neberu-li v potaz rekurze) je velmi podobná struktuře stromu.

- `RadialLayoutAlgorithm` umístí potomky každého uzlu v půlkruhu kolem daného uzlu.

- `GridLayoutAlgorithm` vytvoří z uzlů mřížku.

- K dispozici jsou další algoritmy jako například `SpringLayoutAlgorithm` a `ShiftLayoutAlgorithm`, které již nejsou tolik podstatné. Pro určité struktury se nicméně rovněž můžou hodit.

Po stanovení algoritmu se vypočítají pozice všech komponent a poté se vykreslí výsledný graf. Jednotlivé uzly je přitom možné po vykreslení dále přemísťovat, což je další velká výhoda toolkitu Zest.



Obrázek 4.1: Jednoduchý call graf vytvořený pomocí toolkitu Zest

Díky tomu, že knihovna Zest zajišťuje přesouvání uzlů a poskytuje sadu vykreslovacích algoritmů, s jejím použitím jsem splnil podmínky 2a a 2b ze svého seznamu. Z tohoto důvodu jsem se rozhodl tuto knihovnu využít. Samozřejmě Zest skrývá i jisté nevýhody, o kterých se zmíním v dalších sekcích. Prozatím jsem ovšem dosáhl toho, že bylo jednoduché sestavit elementární graf, který splňoval základní podmínky.

4.2 Interní struktura

Jako první bod jsem si stanovil, že plug-in bude zpracovávat XML soubor obsahující strukturu grafu. Tuto část jsem ovšem řešil až jako poslední, jelikož ze začátku nebylo stanoveno, jak daná struktura bude vypadat. Vznikala současně s celkovou profilovací perspektivou a o její podobu se starali kolegové z týmu Lissom.

Vytvořil jsem si proto vlastní dočasnou interní strukturu, která mi posloužila pro otestování nově vytvářených věcí. Naimplementoval jsem pro ni takové rozhraní, aby stačilo zadefinovat požadovaná spojení skládající se ze dvou řetězců (jmen uzlů) a program se již postaral o převedení do struktury toolkitu Zest (vytvoření patřičných uzlů a jejich propojení). Pro následnou správu uzlů mi stačilo vést si seznam všech uzlů a u každého uzlu seznam ukazatelů na jeho potomky.

4.3 Orientace ve grafu

Ve většině případů plug-in nebude zobrazovat graf o třech uzlech tak, jako jsem to znázornil na obrázku 4.1. Bylo nutné předpokládat, že programátor bude psát složitý a značně se větvící program o velkém počtu uzlů. Orientace v takovém grafu by pak mohla být nepřehledná, zvláště po použití vykreslovacích algoritmů, které se snaží uspořádat všechny uzly do právě zobrazeného pohledu (obdélníku o aktuální velikosti ukotvitelného panelu). Je proto rozdíl mezi tím, zda bude mít uživatel ukotvený pohled v rohu perspektivy, nebo

ho bude mít zvětšený na popředí. Já musel brát v potaz obě varianty. Z toho důvodu jsem naimplementoval následující doplňky, které ulehčují práci s programem a řeší varinatu, kdy bude pro graf vymezen malý prostor.

4.3.1 Zoom

Možnost změny přiblížení grafu představoval bod, který se stal jednou z hlavních priorit. Bez této funkce by byl plug-in z praktického hlediska nepoužitelný. Protože jsem nenašel žádný nástroj, který by se mi o přibližování uměl starat, rozhodl jsem se tento doplněk naimplementovat sám.

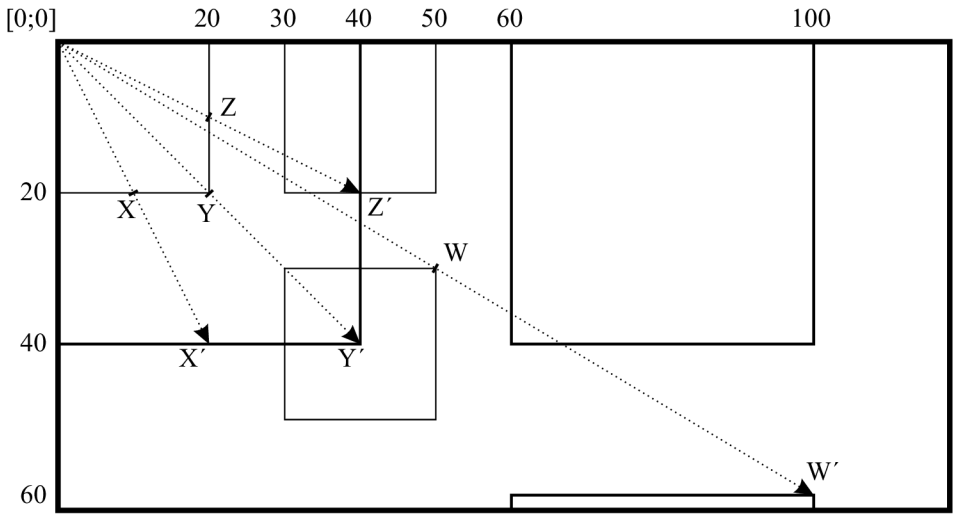
První možností bylo měnit velikost celého grafu jako celku. Třída `Graph` ve svém protokolu sice nabízí metodu pro změnu velikosti, ovšem nepřináší požadovaný efekt. Nastavením atributů jsem docílil změnu velikosti plátna, v kterém jsou umístěny jednotlivé komponenty. Tuto cestu jsem proto zavrhnul a upřednostnil druhé řešení, kterým bylo měnit velikost jednotlivých složek grafu. Touto cestou bylo možné dosáhnout stanoveného cíle.

Situace zde již není tak jednoduchá jako v prvním případě. Samotná změna velikosti nestačí. Je potřeba měnit i měřítko, v kterém na dané komponenty nahlížím. Při dvojnásobném přiblížení budou objekty jednak dvakrát větší, ale zároveň od sebe budou i dvakrát tolik vzdálené. Rovnice pro výpočet jsem proto stanovil následující:

$$w_{i+1} = w_i z \quad h_{i+1} = h_i z \tag{4.1}$$

$$x_{i+1} = x_i z \quad y_{i+1} = y_i z \tag{4.2}$$

Rovnice 4.1 značí výpočet nové velikosti objektu (width, height) a v rovnicích 4.2 je znázoroněn výpočet počátečního bodu nové pozice objektu. Proměnná z značí míru přiblížení (resp. oddálení). Pokud $z \in (0, 1)$, jedná se o oddálení, pokud $z \in (1, \infty)$, jedná se o přiblížení. Jelikož Zest se stará o spojení uzlů, stačí změnit pouze pozice uzlů a spoje se již napraví samy v závisloti na uzlech.



Obrázek 4.2: Ukázka myšlenky dvojnásobného zoomu

Otázkou zůstavalo, jakou hodnotu stanovit pro parametr z (zoom). V počítačové grafice se pracuje s celými čísly (počty pixelů), nicméně v mých rovnicích je při oddálení možné docílit desetinných čísel, a tím i chyby způsobené zaokrouhlováním. Při oddálení a následném přiblížení by uživatel proto nemusel dostat totožný obrázek, jako měl před oddálením. Je to z toho důvodu, že u každého uzlu by se mohla akumulovat jiná chyba zaokrouhlováním.

Problém jsem vyřešil tím, že jsem při vytváření uzlů zaokrouhlil hodnoty na hodnoty dělitelné čtyřmi. Parametr z jsem následně nastavil na hodnotu 2 (resp. 0,5 při oddalování) a oddálení povolil pouze čtyřnásobně (0,25) od základního zobrazení. Více by to nemělo smysl, jelikož uzly mají již v základním zobrazení malou velikost. Při čtyřnásobném se jejich rozměry blíží nule. Pozice objektů jsem, narozdíl od velikostí, nezaokrouhloval. Jednak je uživatel může měnit a jednak lze zaokrouhlení v tomto případě zanedbat.

Další nedostatek rovnic je ten, že zoomování se provádí vzhledem k bodu $[0; 0]$. Je to chování, které by mohlo uživatele zmást. Nebývá použito ani v praxi, pokud si ho uživatel nevyžádá. Vždy je stanoven referenční bod, vzhledem kterému se zoom mění. Je odvozen například z polohy kurzoru vzhledem ke kreslícímu plátnu, případně ze středu kreslícího plátna. Rovnice 4.2 pro změnu polohy jsem tedy musel opravit. Z důvodu pozdějších optimalizací jsem si je rozdělil podle toho, zda se vykonává přibližování či oddalování.

Jako první uvedu přibližování ($z > 1$):

$$x_{i+1} = x_i z - (p_x z - p_x) \qquad y_{i+1} = y_i z - (p_y z - p_y) \tag{4.3}$$

Do rovnic 4.3 přibyl referenční bod P, vzhledem ke kterému se bude přebližovat. Používám ho k výpočtu vzdálenosti, o jaký se bod P při zvětšování posune. Tento rozdíl odečítám od bodu, který jsem spočítal v rovnicích 4.2. Tím jsem docílil požadované pozice odvozené z referenčního bodu P. Vytknutím proměnné p jsem dále dostal:

$$x_{i+1} = x_i z - p_x(z - 1) \qquad y_{i+1} = y_i z - p_y(z - 1) \tag{4.4}$$

Rovnice 4.4 šly dále optimalizovat, jelikož jsem se rozhodl zvětšovat s dvojnásobným krokem (tedy $z = 2$). Ušetřil jsem tím jedno násobení:

$$x_{i+1} = x_i z - p_x \qquad y_{i+1} = y_i z - p_y, \quad \text{kde } z = 2. \tag{4.5}$$

Podobným způsobem jsem odvodil rovnice pro oddalování ($0 < x < 1$):

$$x_{i+1} = x_i z + (p_x - p_x z) \qquad y_{i+1} = y_i z + (p_y - p_y z) \tag{4.6}$$

Rovnice 4.6 jsou ve skutečnosti totožné s rovnicemi 4.3 pro přiblížení. Napsal jsem je ovšem v tomto tvaru, aby v nich bylo zdůrazněno, že p_y je v tomto případě větší jak součin $p_y z$ (pakliže nepracuji se zápornými čísly). Navíc jsem provedl jinou úpravu vedoucí k optimalizaci, jelikož se proměnná z nyní rovná číslu 0,5.

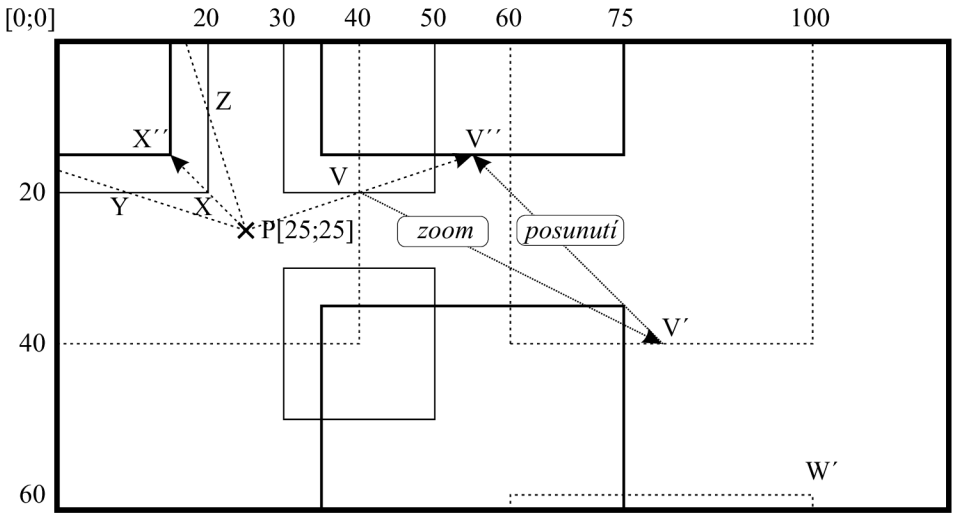
$$x_{i+1} = x_i z + p_x(1 - z) \qquad y_{i+1} = y_i z + p_y(1 - z) \tag{4.7}$$

Z toho dále:

$$x_{i+1} = \left\{ \left[x_i + p_x \left(\frac{1}{z} - 1 \right) \right] z \right\} \qquad y_{i+1} = \left\{ \left[y_i + p_y \left(\frac{1}{z} - 1 \right) \right] z \right\} \tag{4.8}$$

A pokud $z = 0,5$, můžu po úpravě napsat:

$$x_{i+1} = (x_i + p_x) z \qquad y_{i+1} = (y_i + p_y) z, \qquad \text{kde } z = 0,5. \tag{4.9}$$



Obrázek 4.3: Dvojnásoné přiblížení a následný posun vzhledem k bodu P

K úplné dokonalosti zbývalo dodělat několik detailů: změna tloušťky čar, změna velikostí fontů jednotlivých popisků, aby zoomování vypadalo co nejlépe – tedy, aby nad přibližováním uživatel nepřemýšlel. Samotné zoomování jsem propojil s kolečkem u myši. Bývá to takto provedeno v praxi (například v grafických programech). Já se tím inspiroval, jelikož jsem toto uživatelské rozhraní shledal jako praktické. Uživatel najede kurzorem na oblast, kterou hodlá přiblížit a teprve poté ji přiblíží. Implementoval jsem i variantu pro programátora, který bude pracovat na notebooku bez myši a jeho touchpad nebude scrollování podporovat. Je proto možné zoomovat i přes toolbar, či pravým tlačítkem přes vyskakovací menu.

Pokud je míra oddálení vyšší než základní a uzly tím mají menší velikost, než je jejich základní, nastavuji jim styl `NODES_FISHEYE`. To zajistí, že po najetí myší nad zmenšený uzel se uzel dočasně přiblíží a uživatel tím bude moci pohodlně přečíst jeho název.

4.3.2 Posouvání

Přiblížený graf se většině případů nevejde do plochy pohledu, ale je zobrazen pouze jeho výřez. Je proto nutné zajistit, aby se v něm programátor mohl určitým způsobem orientovat. Provedl jsem to několika způsoby.

První možností pro uživatele je použít šipky na klávesnici. Do grafu jsem totiž implementoval rozhraní `KeyListener`, které zpracovává signály z klávesnice. Jakmile je zaregistrovaná jedna ze čtyř šipek na klávesnici, všechny komponenty grafu se posunou v opačném směru šipky. Tím je dosaženo dojmu, že se uživatel v grafu přesmísťuje. Míra posouvání přitom záleží na zoomu. Pokud je graf nejvíce oddálen, posouvání se koná v malé míře (2 pixely). S postupným přibližováním se vzdálenost posunu zvyšuje.

Na konzultaci mi bylo doporučeno, abych navíc implementoval posouvání pomocí myši, kdy uživatel klikne levým tlačítkem do prostoru a tahem (se stále stisknutým tlačítkem) posune s grafem. Tímto lze docílit podobného efektu jako v předchozí variantě. Výhodou ovšem je, že je uživatelsky přívětivější.

Naprogramování této varianty bylo složitější než u posouvání pomocí klávesnice. Vydal

jsem se cestou implementace rozhraní `MouseListener` a `MouseMoveListener`. První mi posloužilo k odchytu signálu, když uživatel stiskne levé tlačítko myši a nevybere přitom žádnou komponentu. Já si pozici kurzoru uložím jako výchozí bod V, nastavím pomocné příznaky a zároveň změním tvar kuzoru z šipky na tvar ruky, aby byl uživatel informován, že může s grafem posouvat.

Pokud uživatel poté začne konat pohyb myší, začne se generovat událost `MouseMove`, kterou odchytávám v metodě druhého rozhraní `MouseMoveListener`. Přitom testuji, zda je stále stisknuté levé tlačítko myši. Z příchozích událostí je možné zjistit novou pozici kurzoru. Pomocí nové pozice a výchozího bodu V si mohu spočítat vzdálenost, kterou kurzor urazil, a aplikovat ji na přesun všech komponent grafu. Novou pozicí poté přepíši výchozí bod V a aplikuji ji v dalším kroku.

$$\Delta x = x_i - x_{i-1} \qquad \Delta y = y_i - y_{i-1} \tag{4.10}$$

V rovnici 4.10 je ukázán výpočet vzdálenosti, která se narozdíl od varinaty posouvání pomocí šipek na klávesnici aplikuje na přesouvání komponent ve stejném směru posunu.

Není problém zadat objektu pozici v záporných hodnotách. Nachází se pak sice mimo plátno (tedy není vidět), ovšem bez toho by nefungoval ani jeden z výše uvedených vzorců, včetně vzorců pro zoom.

4.3.3 Skrývání uzlů

I přesto, že je možné posouvat se v grafu, nastávají případy, kdy graf obsahuje velké množtví uzlů a pro uživatele je lepší zobrazit si nejprve kořenový uzel (například funkci `main`) a až poté si postupně generovat jeho potomky po určitých krocích. Uživatel tím dostane lepší přehled o topologii grafu. V mnoha případech uživatele může zajímat pouze určitá část grafu. Zobrazení celé struktury je proto nežádoucí.

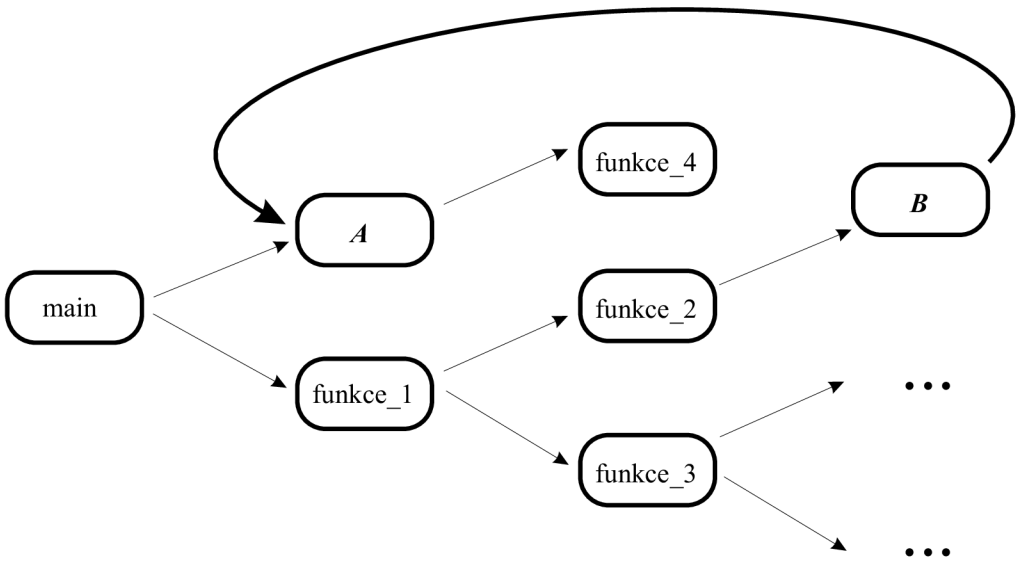
Řešení jsem nalezl v rozhraní komponent call grafu. To praví, že každému objektu je možné stanovit jeho viditelnost (`setVisible(boolean enable)`). Toho jsem využil při skrývání jednotlivých uzlů. Výhodou je, že pakliže schovám určitý uzel, knihovna Zest se postará, aby byly schovány i všechny spojení vedoucí z a do tohoto uzlu.

Nepraktickým řešením by bylo, aby uživatel nastavoval zvlášť každému uzlu jeho viditelnost. Vhodnější je tuto činnost zautomatizovat tím, aby změnu viditelnosti mohl aplikovat na skupinu uzlů – nejlépe na potomky vybraného uzlu. Je to z důvodu, že jednotlivé komponenty grafu jsou v paměti rovněž uloženy do stromové struktury. Díky ní je možné dotázat se každého uzlu na jeho potomky, což mi posloužilo právě pro tento problém.

Tímto jsem vyřešil, že uživatel může přímo schovat celou větev. Nastává ovšem problém, kdy jsou větve mezi sebou propleteny. V takovém případě je nutné hlídat, na jaké úrovni se vybraný potomek nachází. Úroveň uzlu je v mém programu znázorněna celým číslem. To udává, přes kolik spojení je možné se dostat do hlavního kořenového uzlu. Přitom je brána nejbližší cesta. Z možnosti prolínání větví vyplývá, že funkce A znázorněná na obrázku 4.5, která se nalézá na úrovni 1, může být volána funkcí B, která se nachází až na třetí úrovni. Schování potomků uzlu B přitom smí ovlivnit pouze uzly s nižší úrovní (úrovní s vyšší hodnotou). Jedná se o uzly, jejichž nejkratší cesta ke kořenovému uzlu vede přes vybraný uzel B. Uzel A při změně stavu uzlu B zůstane nezměněný. Toto byl jeden z důvodů, proč jsem třídu `GraphNode` rozšířil. Je vhodnější si do třídy `GraphNode` přidat celočíselný atribut, do kterého si budu parametr úrovně uzlu ukládat již při vytváření stromu, než ho pak zjišťovat při každé změně viditelnosti.

Mimo skrývání jednotlivých větví jsem naimplementoval možnost postupného expandování jednotlivých úrovní pro tyto větve. Programátor si tím může pomalu krokovat tok vizualizovaného programu. V této varinatě již není žádané, aby bylo zamezeno měnit viditelnost potomků, kteří mají atribut úrovně stejný či větší jak vybraný uzel. Bylo by matoucí, kdyby byla rozgenorována pouze podmnožina ze všech volaných funkcí. Tato varianta se stará pouze o expadnování, nikoli o skrývání, nebyl tudíž problém ignorovat atribut úrovně uzlu. Je to z důvodu, že pokud uživatel začne expandovat uzly od funkce `main`, uzel A bude nagenerovaný dříve jak uzel B.

Funkci skrývání uzlů jsem završil možností nechat si rychle zobrazit uzly do určité úrovně. Uživatel zadá číslo a na jeho základě se mu zobrazí odpovídající uzly. Zde jsem usoudil, že bude pro uživatele přívětivější zadat číslo značící, kolik úrovní chce zobrazit – tedy číslo o jedna větší, než číslo značící nejkratší cestu ke kořenovému uzlu. Mohlo by být totiž matoucí a tedy nepraktické, že pro zobrazení kořenové funkce je potřeba zadat číslo 0.



Obrázek 4.4: Problém při skrývání uzlů s nižší úrovní

Důležitou vlastností skrytých uzlů je, že nejsou filtrovány. Znamená to, že skryté uzly sice nejsou vidět, nicméně vykreslovací algoritmus, o kterém jsem psal v sekci 4.1, s nimi počítá. Při dynamické změně rozmístění pak upraví pozice všech uzlů, aby i skryté uzly měly svoje místo. Při rozgenerovávání větví by pak nemělo dojít ke kolizím. Doporučuji tedy, aby si uživatel při naběhnutí grafu nejprve nastavil zoom a vykreslovací algoritmus, pokud to je třeba. Až poté si přes toolbar, o kterém bude zmíňka dále (4.5.1), může skrýt například všechny uzly mimo hlavního, aby mohl po krocích procházet grafem. Nic mu samozřejmě nebrání provést to v opačném pořadí, nicméně při expandování uzlů se programátor může dočkat nepřiměřeného zhuštění uzlů vedoucímu k nepřehlednosti.

Příkaz pro zobrazení jedné úrovně jsem zaobalil do dvojkliku na daný uzel. Dále je možné přes vyskakovací menu (pravé tlačítko u myši) blíženji ovládat skrývání a zobrazování uzlů.

4.3.4 Skrývání rekurzivních uzlů

Pokud některý uzel volá sám sebe, jedná se o rekurzivní uzel. V takovém případě je nutné si uschovávat každé volání zvlášť, jelikož v každém z nich může mít uzel jiné atributy (například výpadky paměti cache). Uživatel si pak může jednotlivá volání nechat nagenerovat. Tyto uzly si proto vedu ve speciálních seznamech jako uzly odvozné od rekurzivního uzlu. Další důvod je popsán na příkladu 4.1.

Příklad 4.1 *Představme si analyzovaný program, který se bude skládat pouze z pěti funkcí (tedy uzlů). Jeden z těchto uzlů ovšem bude rekurzivní a bude sám sebe volat stokrát – tedy do grafu přibude dalších sto uzlů, které budou při počátečním vykreslení skryté. Podle odstavce ze sekce 4.3.3 by řešení bylo takové, že vykreslovací algoritmus nebude tyto uzly ignorovat, i když budou skryty. Všechny uzly tedy budou rozmístěny tak, jako by se v grafu nacházelo všech 105 uzlů. Rekurzivních 100 přitom nemá tak velkou váhu a uživatel je bude zobrazovat pouze v případě, že se bude zajímat konkrétně o jednotlivá volání této rekurze. Bylo by tedy vhodnější věnovat veškeré místo základním pěti uzlům a rekurzivním uzlům uvolnit prostor, až když je bude uživatel potřebovat.*

Řešení příkladu 4.1 přináší knihovna Zest, která pro filtrování objektů grafu obsahuje rozhraní `Filter`. Toto rozhraní mi vnutí implementovat metodu `isObjectFiltered`, která v parametru přebírá objekt, na který se aplikuje vykreslovací algoritmus. Vytvořil jsem třídu implementující toto rozhraní a její instanci jsem přidal do seznamu filtrů instance třídy `Graph`. Vždy když si uživatel vyžádá překreslení grafu s aplikací vykreslovacího algoritmu, pro každý objekt se zavolá metoda `isObjectFiltered`. Pokud je návratová hodnota funkce `isObjectFiltered` `true`, předávané objekty budou filtrovány. Aplikoval jsem proto filtr na všechny rekurzivní uzly, které jsou zároveň schovány. Pokud se uživatel rozhodne je zobrazit, návratová hodnota této funkce se změní u zvolených uzlů na `false` a uzly budou začleněny do vykreslovacího algoritmu. V takovém případě volám nové překreslení, jelikož je potřeba vypočítat nové pozice všech uzlů.

4.3.5 Hledání uzlů a nápověda

Dalším užitečným doplňkem pro uživatele je možnost najít určitý uzel – ať pro rychlé získání informací o uzlu, tak pro zvýraznění uzlu v rozsáhlé síti. K tomu jsem implementoval hledáček.

Do pohledu jsem přidal SWT komponentu `Text`. Je to klasické vstupní pole, do kterého je možné psát text. Se získávaným vstupem se dynamicky zvýrazňují uzly, jejichž název by mohl odpovídat tomuto textu. Ignoruje se přitom rozdíl mezi malými a velkými písmeny a hledají se všechny uzly, které na daný řetězec začínají, případně se mu rovnají.

K tomuto hledáčku jsem navíc implementoval nápovědu. Při stisku kombinace kláves `CTRL` a `SPACE` se objeví okno se seznamem uzlů, které danému řetězci odpovídají. Okno jsem implementoval pomocí toolkitu JFace (3.4) rozšířením komponenty `PopupDialog`. Její výhoda je ta, že může být zobrazena na popředí a nesebere přitom textovému poli *focus* (aktivitu, která objektu zajišťuje, že bude přijímat signály např. z klávesnice právě on). Při stisku dalších kláves přijímá signály od klávesnice nadále vstupní pole. Jiná situace nastává, pokud uživatel zmáčkne šipku dolů nebo nahoru. V takovém případě focus převezme seznam nabízených uzlů ve vyskakovacím okně a uživatel tak může přímo listovat tímto seznamem. Klávesou `ESC` nebo `ENTER` se aktivita opět vrátí textovému poli.

Nápovědu jsem přitom mohl vyřešit snadněji. V kapitole JFace jsem se ve výčtu nástrojů v bodě 3 zmiňoval o komponentě `FieldAssist`, která je přímo určená pro našeptávání

vstupů. `PopupDialog` má ovšem obecnější využití. Z toho důvodu jsem se chtěl seznámit spíše s tímto nástrojem a výsledně se rozhodl pro tuto variantu.

V některých případech se může hodit rychle najít uzly, které obsahují určitý znak, či kus řetězce a je přitom jedno, zda se nachází na začátku slova nebo uprostřed. Tímto typem hledání jsem se inspiroval v příkladech ke knihovně Zest [2] – konkrétně třídou `GraphSnippet5`. Zajistil jsem, aby můj graf reagoval mimo signálů šipek z klávesnice při posouvání (4.3.2) i na klávesy představující znaky, které mohou být obsaženy ve jménech funkcí. Stiskem kláves se pak generuje řetězec, který je aplikován na jména uzlů. I tímto způsobem lze tedy v mém plug-inu vyhledávat. Je to ovšem méně pohodlný způsob a je určen spíše pro rychlé zvýraznění určitých uzlů.

4.4 Vzhled

Hlavním účelem mé práce bylo zajistit funkčnost programu. Mým druhým cílem bylo vytvořit uživatelsky příjemný vzhled. Samozřejmě proto, aby ladil oku a také, aby patřičně zvýrazněné komponenty grafu poskytovaly další informace prospěšné pro jeho rychlejší orientaci.

4.4.1 Odlišení uzlů

V sekci 4.3 je uvedeno, že uzly mohou být různých typů a mohou se nacházet v různých stavech. Příkladem je například rekurzivní funkce, která má zagenerované uzly jednotlivých rekurzivních volání. Dalším příkladem je uzel, který má zagenerované svoje potomky. Takové uzly by bylo výhodné odlišit od ostatních, aby programátor ihned věděl, o jaký uzel se jedná a patřičně k němu i přistupoval.

Nejefektivnější variantou by bylo měnit tvar uzlu, jelikož to uživateli dá rychle najevo, že uzel je něčím speciální. Knihovna Zest ovšem pracuje na značně abstraktní úrovni a v jejím rozhraní jsem nenašel žádné metody pro změnu tvaru objektů. Zkoumal jsem i kód této knihovny, zda by to nešlo nějak obejít, ovšem nepřišel jsem na žádné rozumné řešení.

Přistoupil jsem tedy k méně efektivní záložní variantě, a to ke změně barvy uzlů. Tyto metody již v rozhraní existují (`setBackgroundColor(Color color)` a `setForegroundColor (Color color)`). Navíc mimo to, že jsem nastavil jiné pozadí a barvu textu, zvýraznil jsem i obrys (`setBorderColor(Color color)` a `setBorderWidth(int width)`). I při největším oddálení je pak zřetelně poznat typ uzlu, jelikož tloušťku obrysu při změně přiblížení nijak neměním.

V závislosti na uzlu jsem použil tyto barvy:

1. **Světle šedá** – základní nevýrazný uzel, který není ničím specifický. Jedná se o implicitní barvu, kterou používá knihovna Zest.
2. **Zelená** – uzel, který má schované své potomky.
3. **Červená** – uzel, který je volán rekurzivně a má schované své rekurzivní potomky.
4. **Černá** – uzel, který splňuje podmínku 2 i 3.

V přílohách jsou k nahlédnutí obrázky demonstrující toto zvýraznění.

4.4.2 Zvýraznění komponent

Mimo to, že je možné měnit barvy v závisloti na typu uzlu, je vhodné daný uzel dočasně zvýraznit. Tato možnost základně funguje jak pro uzly, tak pro spojení. Jakmile je uživatel vybere myší, žlutě se zvýrazní.

Já jsem tuto možnost rozšířil dále na vyhledávání a rozgenerovávání uzlů. Pokud uživatel používá hledáček (4.3.5), jsou nalezené uzly dynamicky zvýrazňovány barvou k tomu určenou. Dále, pokud si uživatel nechá nagenerovat potomky určitého uzlu (4.3.3), je vhodné nově zobrazené uzly také nasvítit a zdůraznit je tím uživateli.

K těmto věcem jsou k dispozici metody `highlight()` a `unhiglight()`. Jedná se o dočasné zapnutí druhé barvy uzlu určené k podsvícení. Vyhnul jsem se tím nutnosti měnit základní barvu objektu.

4.4.3 Změna vzhledu přechodů

Při testování přehlednosti se mi ukázalo užitečné barevně si odlišovat spojení, které vedou z odlišných uzlů. Základně jsou všechny uzly zobrazeny šedou barvou. Zajistil jsem proto, aby uživatel mohl přepnout do režimu, kdy budou tyto čáry odlišeny. V každém uzlu si totiž uchovávám barvu jeho spojení, kterou mu přiděluji při vytváření. Při zapnutí rozlišování pak projdu všechny spoje a dotazuji se na barvu uloženou v jejich zdrojovém uzlu.

Je samozřejmě možné stanovenou barvu spojení měnit jak pro individuální propojení, tak pro všechny spojení náležící zvolenému uzlu. Čáry reprezentující rekurzivní volání mají stanovenu speciální barvu a to tmavě červenou.

Mimo barvy je možné měnit i styl čáry. K dispozici je několik možností (přímé, tečkované, čárkované, atd. . .). Využívá se přitom stejného mechanismu jako u změny barev. Lze tedy měnit styl jak určitému spojení, tak přes uzel sdruženě více spojením.

Posloužily mi metody `setLineStyle(int style)` a `setLineColor(Color color)`.

4.4.4 Vlastnosti uzlu

Každý uzel převezme z XML struktury mimo jeho názvu a voláných funkcí i další atributy, které o něm nesou podrobnější informace. Tyto informace jsou pro vývojáře dosti podstatné. Z toho důvodu je vhodné, aby měl uživatel možnost si pod každým uzlem zobrazit rámeček s těmito informacemi. Přitom musí být zajištěno, aby ho mohl uživatel kdykoliv schovat.

Při implementaci jsem se setkal s problémem. Aktuální verze knihovny Zest nepodporovala žádné přidávání textových objektů, které by byly součástí uzlu. Našel jsem tam pouze komponentu `GraphContainer`. Je to šikovná komponenta, která vypadá stejně jako klasický uzel `GraphNode`. Liší se od něho tím, že má možnost „rozbalit se“ – tedy zobrazit pod sebou rámeček. Ten ovšem slouží jako plátno pro další graf, proto se také tento typ uzlu jmenuje `GraphContainer`. Navíc tento uzel nejde zmenšovat. Byl by tedy nepoužitelný v mém zoomu.

Abych tedy nemusel přepisovat knihovnu a dodělávat si vlastní komponentu, aplikoval jsem jiné řešení. S každým překreslením grafu (respektive plátna, na kterém se graf nachází), se generuje signál `PaintEvent`. Já do mého grafu zakomponoval rozhraní `PaintListener`, které tento signál umí odchytávat. S každým signálem `PaintEvent` jsem vykreslil pod každý uzel požadovaný rámeček s atributy daného uzlu. Řešení funguje přesvědčivě, jelikož signál překreslení je generován s každou změnou grafu. Pokud je například změněna pozice některého z uzlů, i rámeček příslušící tomuto uzlu změní svojí pozici, jelikož ji z něho odvozuje, stejně tak jako text, který má obsahovat. Signál překreslení je navíc možné generovat uměle

metodou `redraw()`. Stejným způsobem jsem vyřešil i popisky u rekurzivních funkcí, které znázorňují počty rekurzivních volání.

Jedinou nevýhodou je, že rámeček není brán jako komponenta grafu, tudíž na něj není aplikován vykreslovací algoritmus. Má sice svojí pozici odvozenou od příslušného uzlu, nicméně je zobrazen pod uzlem. Pokud bude zvoleno velké přiblížení a všechny komponenty přitom budou rozmístěné do aktuálního pohledu, může se stát, že se bude rámeček překrývat s cizím uzlem. S tím je tedy nutné počítat a je vhodné aplikovat vykreslovací algoritmus při větším oddálení. Uzly pak od sebe budou více vzdáleny, jelikož bude dostupný větší prostor pro vykreslení. To je obecné pravidlo, které je dobré brát v potaz, i když jsou rámečky s popisky skryty.

Při největším oddálení se rámečky automaticky vypínají na dobu, po kterou bude toto oddálení aktuální. Je to z důvodu, že největší oddálení slouží spíše pro náhled. Navíc by ani nebyla moc dobrá čitelnost těchto rámečků. V takovém případě je možné kliknout na daný uzel kolečkem a vyskočí `PopupDialog`, který bude tyto informace rovněž obsahovat. Implementoval jsem ho především kvůli tomu, že může obsahovat textové pole a uživatel si tedy může dané informace okopírovat v textové podobě.

4.5 Uživatelské rozhraní

Sekce 4.5 souvisí se všemi předchozími sekcemi v této kapitole. Jejím cílem je zajistit, aby uživatel mohl jednotlivé součásti pluginu (jako je zoom, posouvání, schovávání uzlů, atd. . .) snadno a efektivně používat. Většinu ovládání jsem přitom již v předchozích sekcích popsal. Bylo ovšem nutné sestavit ucelené ovládání, které by bylo rychle dostupné bez jakéhokoli zkoumání.

4.5.1 Toolbar

Prvním takovým ovladačem je toolbar. V něm jsem vytvořil tlačítka, která spouští základní operace, které je třeba často provádět. Jsou jimi například přibližování a oddalování. Když uživatel mění zoom, dělá to v určitých případech za účelem přenastavení rozměrů. Začlenil jsem tedy zároveň tlačítko s popiskem *Resize*, které způsobí znovuaplikování vykreslovacího algoritmu. Ten rozmístí všechny uzly na aktuálně zobrazenou plochu.

Dále jsem přidal tlačítka pro globální změnu barev a stylu přechodů, tlačítko pro možnost zobrazení popisků (4.4.4) a nakonec tlačítko pro zobrazení samotné hlavní funkce. To poslouží programátorovi pro procházení grafem. Dále je dostupné menu pro přepínání mezi vykreslovacími algoritmy

4.5.2 Vyskakovací menu

Dalším užitečným ovladačem je vyskakovací menu. Uživatel ho otevře pravým kliknutím do grafu. Program pokaždé otestuje, jakou komponentu přitom vybral a v závislosti na výběru zobrací odpovídající menu.

Pokud uživatel kliknul do pracovní plochy, zobrazí se obecné menu, v kterém je možné měnit zoom a různě filtrovat zobrazené uzly. Pokud vybere spojení, je možné měnit jeho styl a barvu. V případě výběru uzlu je možné skrývat a expandovat jeho potomky, jak již bylo popsáno v podsekci 4.3.3. Zároveň je možné hromadně měnit barvu a styl všem jeho spojením.

4.6 Začlenění do profilovací perspektivy

Po dokončení vlastního plug-inu jsem pracoval na jeho integritě s vývojovým prostředím projektu Lissom. Bylo nutné, aby plug-in místo vnitřní struktury, kterou jsem používal pro testování, přijímal XML strukturu, s kterou pracuje prostředí projektu Lissom. Do ní byly začleněny informace potřebné pro vykreslení call grafu.

4.6.1 XML struktura

XML struktura je soubor, v kterém jsou zaznamenány statistiky programu, který vyvojář týmu Lissom vyvíjí. Pro můj plug-in jsou podstatné informace o funkcích a jejich volání. Tyto data se nacházejí v části `PROFILER` mezi tegy `CALLGRAPH`. Pro představu uvádím příklad 4.2, který ukazuje uložení funkce `main`.

Příklad 4.2

```
<CALLGRAPH>
  <FUNCTION>
    <ID>main</ID>
    <FUNCTION>
      <!-- popis volané funkce -->
    </FUNCTION>
    <CYCLES>10</CYCLES>
    <CACHES>
      <CACHE>
        <ID>L1</ID>
        <HITS>5</HITS>
        <MISSES>10</MISSES>
      </CACHE>
    </CACHES>
  </FUNCTION>
</CALLGRAPH>
```

Každá funkce musí mít atribut ID. Jedná se o jméno funkce. Dále může obsahovat libolný počet tagů `FUNCTION`, kde v každém je uložena volaná funkce. Tímto zanořením se docílí znázornění vzájemných volání. Pokud se jedná o rekurzi, zanořená funkce ponese stejné `ID` jako funkce, v které se nachází.

Struktura obsahuje navíc značky, které nesou informace o tom, jak je daná funkce optimalizovaná. Tag `CYCLES` uvádí, kolik cyklů daná funkce trvala. Další tag je `CACHES`. Je to seznam rychlých vyrovnávacích pamětí, u kterých se sleduje, zda v nich procesor získal potřebná data, nebo musel hledat dál v hierarchii pamětí. Tyto události se nazývají *hits* (potřebná data byla dostupná) a *misses* (potřebná data nebyla dostupná). Ve struktuře je pak uveden jejich počet.

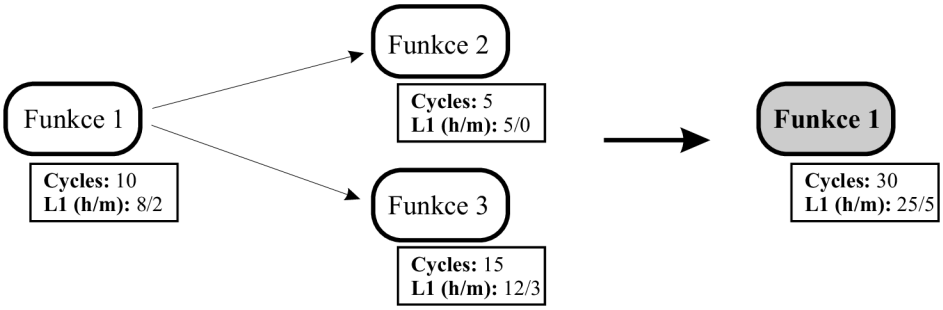
Atributy `CYCLES` a `CACHES` jsou informace, které jsem umístil do popisků uzlů (4.4.4). Musel jsem ovšem dbát na to, aby když uživatel skryje určitou větev, hlavní uzel této větve nesl informace o všech uzlech této větve – tedy zobrazoval sumu atributů `CYCLES` a `CACHES`.

Tento problém jsem implementoval tak, že jsem pro každý atribut uzlu vytvořil tři proměnné:

1. Do první jsem uložil hodnotu získanou z XML struktury.

2. Po zpracování XML struktury jsem uzly prošel znovu a pro každý z nich jsem vypočítal sumu atributů `CYCLES` a `CACHES` náležící všem jeho potomkům. K uložení výsledku mi posloužila druhá proměnná.

3. Třetí proměnnou jsem použil pro uložení aktuální hodnoty zvoleného atributu uzlu.



Obrázek 4.5: Přepočítání atributů při skrývání uzlů

Proměnné 1 a 2 získávám při vytváření grafu a dále je neměním. Slouží mi pro výpočet proměnné 3, která se mění v závislosti na tom, jak uživatel skrývá a zobrazuje jednotlivé větve.

4.6.2 Parsování XML struktury

Pro získání dat ze souboru jsem použil již existující parser vývojového prostředí projektu Lissom. Bylo potřebné ho rozšířit tak, aby uměl zpracovávat informace potřebné pro call graf.

Parser je vytvořen pomocí knihovny Simple [12]. Díky ní je možné převádět objekty jazyka Java do XML struktury a naopak. Těmto úkonům se říká *serializace* a *deserializace*. Pro mě byl důležitý převod XML struktury na objekty jazyka Java – tedy deserializace.

K tomu je potřebné vytvořit třídy příslušící jednotlivým tagům. Pro jednoduché vysvětlení je přiložen příklad 4.3.

Příklad 4.3

```
@Root(name = "STATISTIKA", strict = false)
public class Statistika {

    @Element(name = "JMENO")
    private String jmeno;

    @ElementList(name = "HODNOTY", entry = "HODNOTA",
                                   empty = true, required = false)
    private ArrayList<Double> hodnoty;
}
```

Napsaný parser bude zpracovávat soubory s tagy `STATISTIKA`, které budou obsahovat název a libovolný počet hodnot (`HODNOTA`) obalených v tagu `HODNOTY`. Seznam hodnot přitom není vyžadován, případně může být prázdný.

Do třídy `Statistika` je podstatné doplnit metody *get*, které budou navracet jednotlivé proměnné. Pro načtení struktury je dále nutné vytvořit instanci třídy `Serializer` a metodou `read` získat potřebné informace. Příklad 4.3 je pouze ilustrační. Pro více informací bych rád odkázal na tutoriál domovské stránky knihovny Simple [12].

4.6.3 Testování

Po té, co jsem implementoval parser a zajistil, aby plug-in pracoval se stanovenou XML strukturou (popsanou v 4.6.1), jsem získal pohled kompatibilní s profilovací perspektivou vývojového prostředí projektu Lissom. Před začleněním do tohoto vývojového prostředí jsem plug-in otestoval. Vytvářel jsem si XML soubory obsahující informace reperezentující call grafy a uměle je posílal vytvořenému plug-inu.

Je přitom nekonečné množství podob struktur call grafů a nebylo proto v mých silách vytvořit všechny možné vstupy. Snažil jsem otestovat především rekurze a popisky uzlů. V těchto částech jsem předpokládal, že by se mohlo vyskytovat nejvíce chyb. Ty chyby, které jsem nalezl, jsem odstranil. Počítám ovšem i s tím, že při používání tohoto plug-inu v profilavací perspektivě mohou vyjít na povrch i některé další chyby, které se mi nepodařilo objevit. V takovém případě se je budu snažit dodatečně odstranit.

Kapitola 5

Závěr

Mým úkolem bylo navrhnout, implementovat a otestovat nové rozšíření pro profilovací perspektivu vývojového prostředí projektu Lissom, které je založeno na vývojovém prostředí Eclipse. Seznámil jsem se proto s vývojovým prostředím Eclipse na úrovni potřebné pro vytváření plug-inů. Naučil jsem se tvořit nové pohledy.

5.1 Výsledek práce

Profilovací perspektivu jsem rozšířil o pohled, který převezme stanovenou XML strukturu a pomocí parseru dostupného z vývojového prostředí projektu Lissom, který jsem rozšířil, získá informace potřebné k vykreslení call grafu. Ze získaných dat sestaví call graf a vykreslí ho na plátno, které je umístěné v novém pohledu. Při vykreslování plug-in využívá knihovny Zest, která je součástí grafického toolkitu GEF. Tato knihovna se stará o vykreslovací algoritmy.

Pro vykreslený graf jsem implementoval doplňky, které zajišťují přibližování, oddalování a posouvání v grafu. Dále je možné filtrovat uzly, zobrazovat pouze vybrané větve nebo úrovně, případně si krokovat zobrazený graf. Ke každému uzlu je možné zobrazit popisky (pro samotný uzel, nebo pro celou větev). Dostupný je rovněž hledáček uzlů s našeptáváním názvů. Jednotlivé uzly jsou zvýrazněny v závislosti na jejich typu (rekurzivní funkce, uzel se schovanými potomky, ...). Spojení je možné barevně rozlišit v závislosti na jejich zdrojovém uzlu. Je přitom možné předefinovat základní barvy spojení. Pro snadné ovládání jsem implementoval toolbar a vyskakovací menu. Ovládání je rovněž možné pomocí klávesnice a stanovených gest myší.

Výsledný plug-in jsem otestoval na vzorových XML strukturách kompatibilních se strukturou vývojového prostředí projektu Lissom. Vytvořený pohled bude začleněn mezi již vytvořené pohledy profilovací perspektivy tohoto vývojového prostředí.

5.2 Možná rozšíření

Mým cílem do budoucna je zajistit, aby bylo možné měnit tvary uzlů. Odlišení uzlů jsem prozatím zajistil změnou barvy, což ovšem není tak výrazné.

Další nevýhodou mého plug-inu je, že požadavky na něho kladené se během roku měnily a upřesňovaly. Program jsem tedy několikrát měnil, čímž se plug-in stával čím dál více nepřehledný. Bylo stále složitější přidávat nové části. Rád bych tedy do budoucna sestavil

nový návrh, který by obsahoval nové požadavky a zároveň by se inspiroval z nedostatků plug-inu, které mohou vyplynout z jeho používání.

Při vytváření přílohy k této bakalářské práci, která se skládá ze snímků plug-inu, mě napadlo, že by bylo vhodné doimplementovat export call grafu do některého z rastrových formátů (`jpeg`, `png`, atd...). Toto je tedy rovněž jeden z bodů, který bych do budoucna rád zajistil.

Pohled zobrazující call graf vizualizuje pouze určitou část XML struktury. Do budoucna by bylo rovněž vhodné dodělat pohledy pro zbývající části struktury určené pro profiler. Příkladem jsou *TOP statistiky*, které nesou informace o tom, jak moc je která funkce používaná. Dále obsahuje seznam funkcí s největšími výpadky rychlých vyrovnávacích pamětí a například seznam funkcí, které nejsou používány.

5.3 Závěr

Díky bakalářské práci jsem získal informace o vývojovém prostředí Eclipse a naučil se, jak ho rozšířit o další doplňek. Zabýval jsem se rovněž problematikou call grafů a profilovaním ve vývojových prostředích. Získané informace s velkou pravděpodobností využiji při další práci.

V příloze této bakalářské práce jsou dostupné snímky obrazovky demostrující práci s pohledem.

Literatura

[1] Clayberg, E.; Rubel, D.: *Eclipse : Building Commercial-Quality Plugins*. Boston: Addison-Wesley Professional, druhé vydání, 2008, ISBN 987-0-321-42672-7.

[2] GEF: Graph Widget Snippets [online]. http://www.eclipse.org/gef/zest/snippets.php, 2011 [cit. 2011-03-27].

[3] java2s.com: SWT tutorials [online]. http://www.java2s.com/Tutorial/Java/0280__SWT/Catalog0280__SWT.htm, 2011 [cit. 2011-03-27].

[4] MacLeod, C.: Understanding Layouts in SWT [online]. http://www.eclipse.org/articles/article.php?file=Article-Understanding-Layouts/index.html, 2001 [cit. 2011-03-28].

[5] Northover, S.; Wilson, M.: *SWT : The Standard Widget Toolkit*, ročník 1. Boston: Addison-Wesley Professional, 2004, ISBN 978-0-321-25663-8.

[6] Springgay, D.: Using Perspectives in the Eclipse UI [online]. http://www.eclipse.org/articles/using-perspectives/PerspectiveArticle.html, 2001 [cit. 2011-03-27].

[7] The Eclipse Foundation: SWT : The Standard Widget Toolkit [online]. http://www.eclipse.org/swt/docs.php, 2011 [cit. 2011-03-27].

[8] The Eclipse Foundation: SWT Widgets [online]. http://www.eclipse.org/swt/widgets/, 2011 [cit. 2011-03-27].

[9] The Eclipse Foundation: Eclipse Documentation : The JFace UI framework [online]. http://help.eclipse.org/galileo/index.jsp?topic=/org.eclipse.platform.doc.isv/guide/jface.htm, 2011 [cit. 2011-03-28].

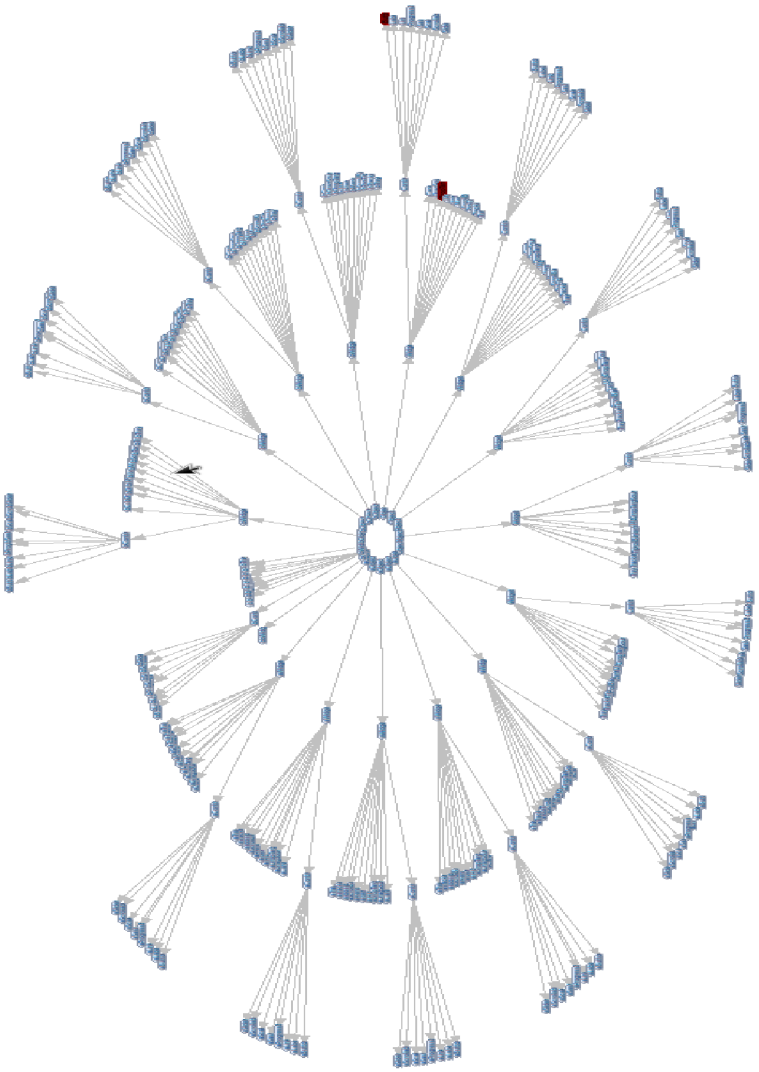
[10] WWW stránky: GEF : Graphical Editing Framework. http://www.eclipse.org/gef/.

[11] WWW stránky: Lissom. http://www.fit.vutbr.cz/research/groups/lissom/index.html.

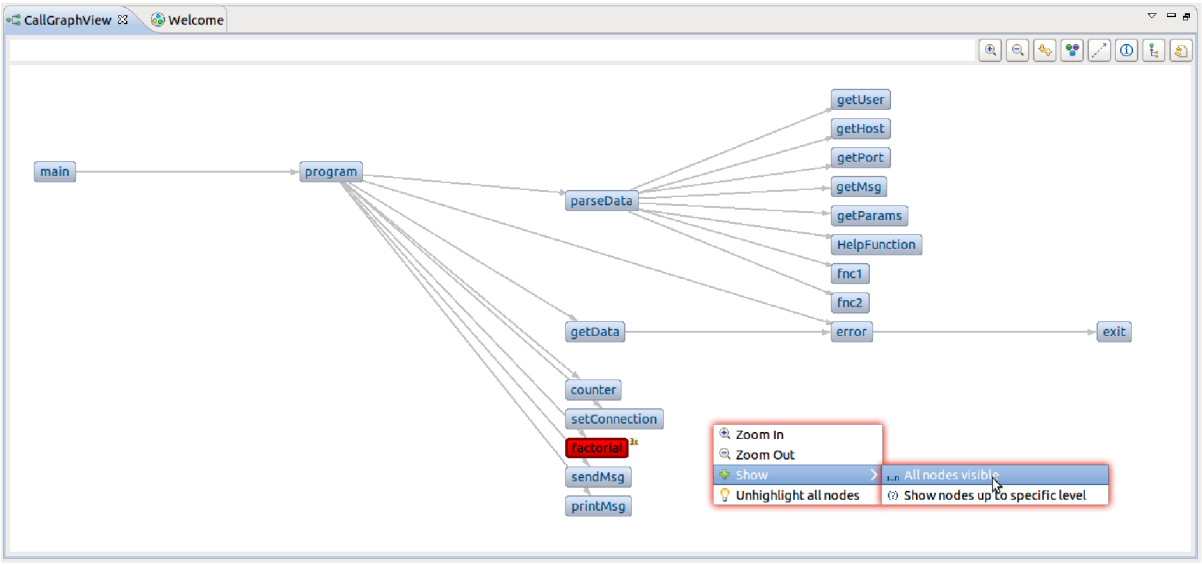
[12] WWW stránky: Simple : XML serialization. http://simple.sourceforge.net/home.php.

Příloha A

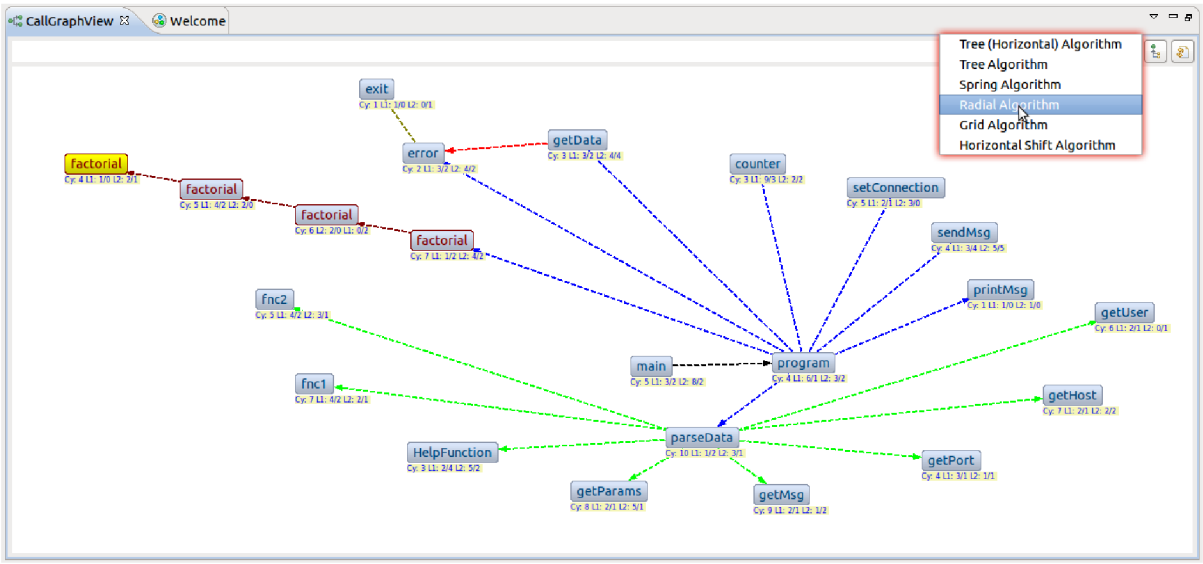
Snímky plug-inu



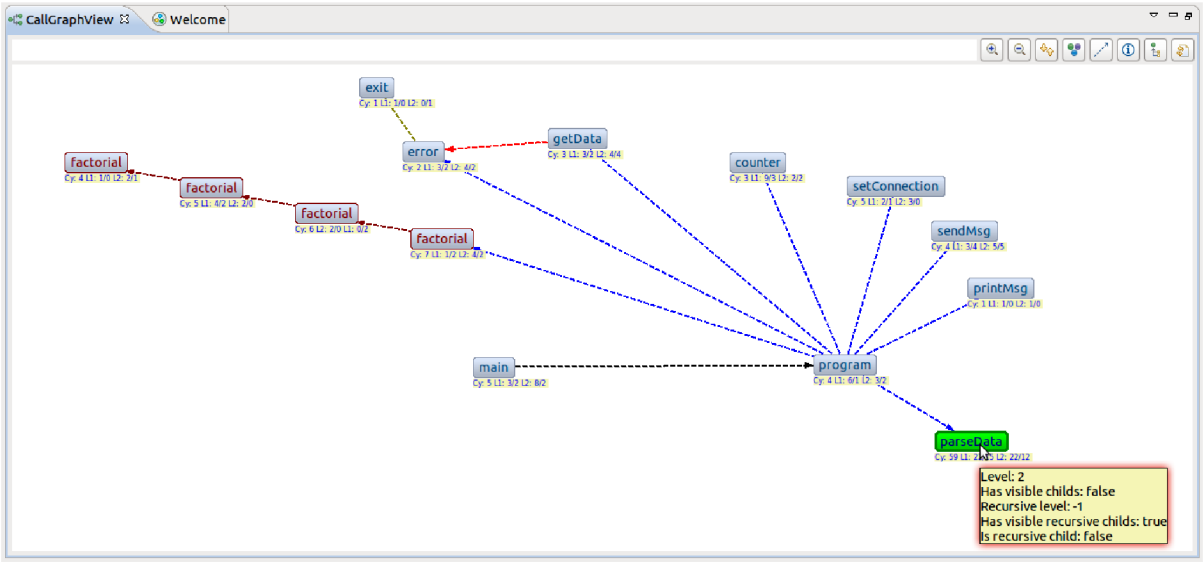
Obrázek A.1: Ukázka velkého počtu uzlů



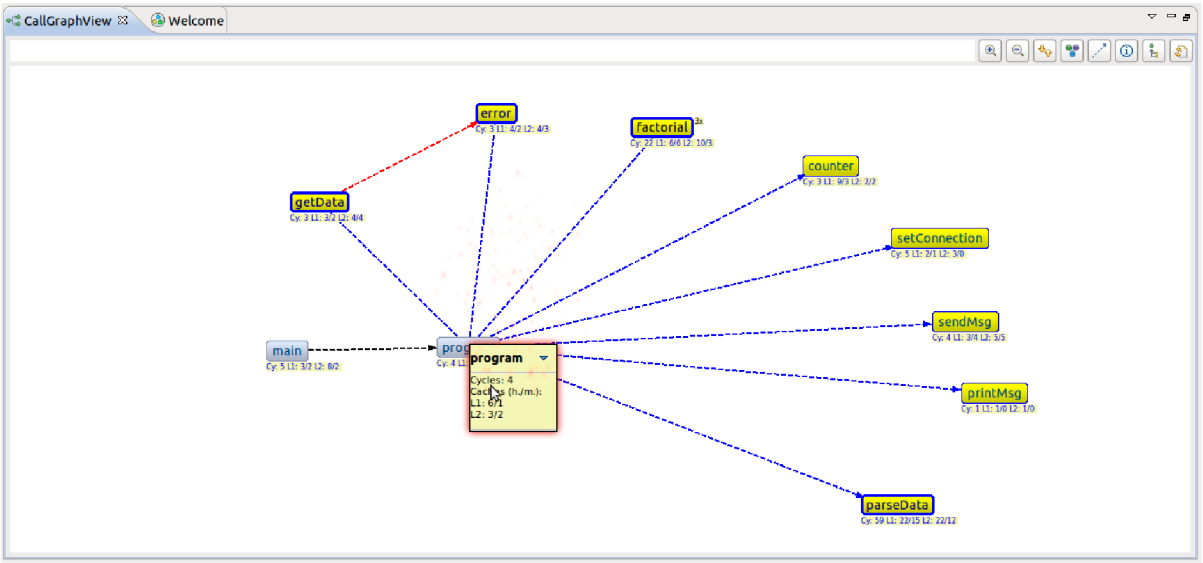
Obrázek A.2: Základní vykreslení grafu



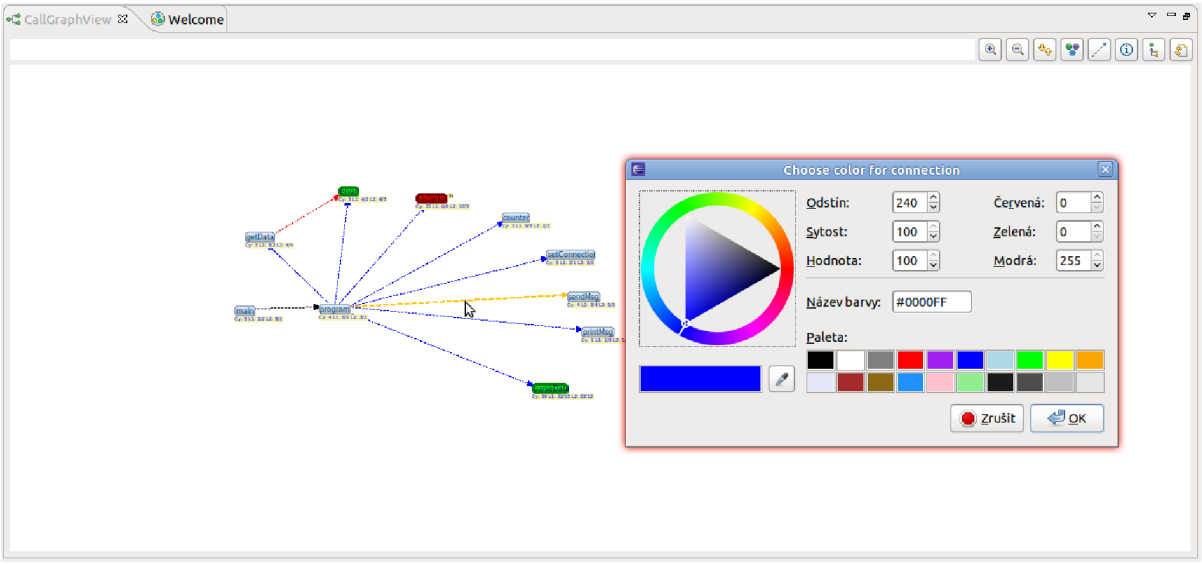
Obrázek A.3: Ukázka různých vykreslovacích algoritmů



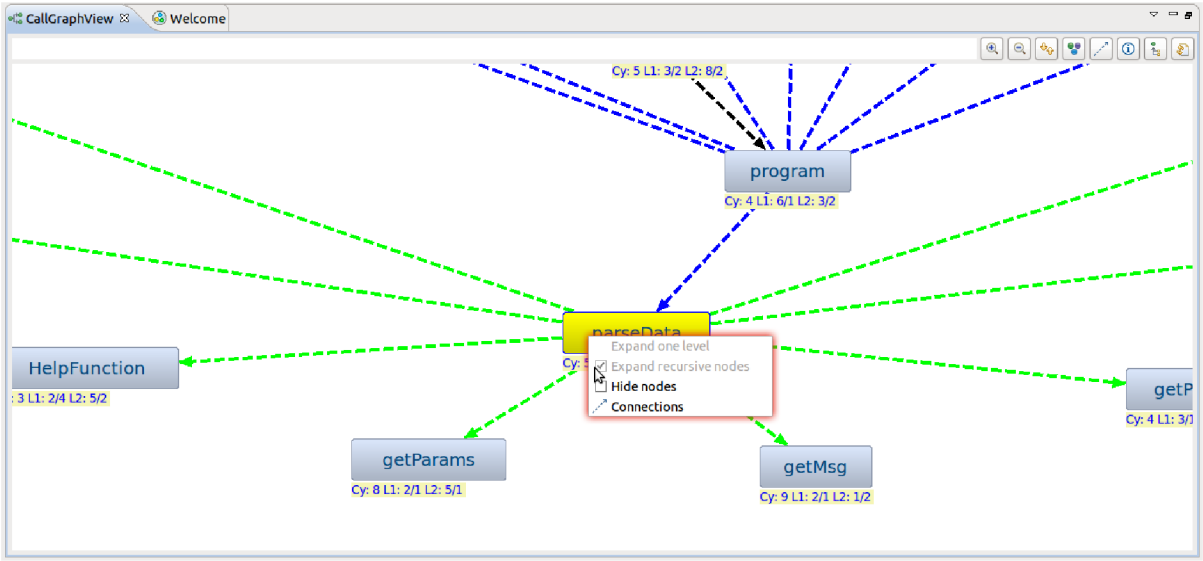
Obrázek A.4: Možnost skrývat uzly



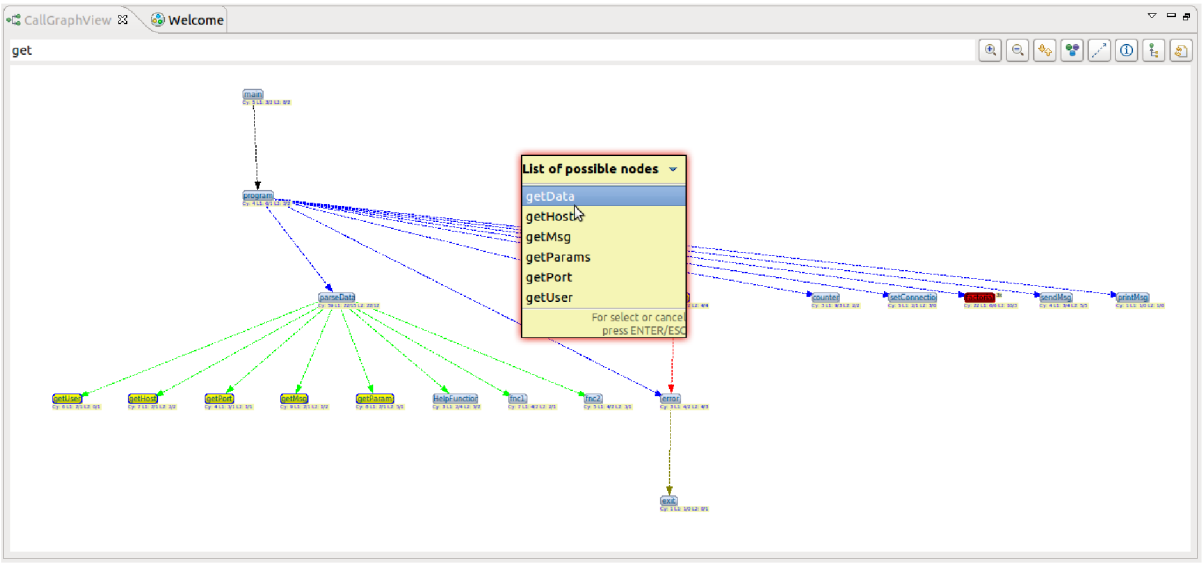
Obrázek A.5: Krokování grafem



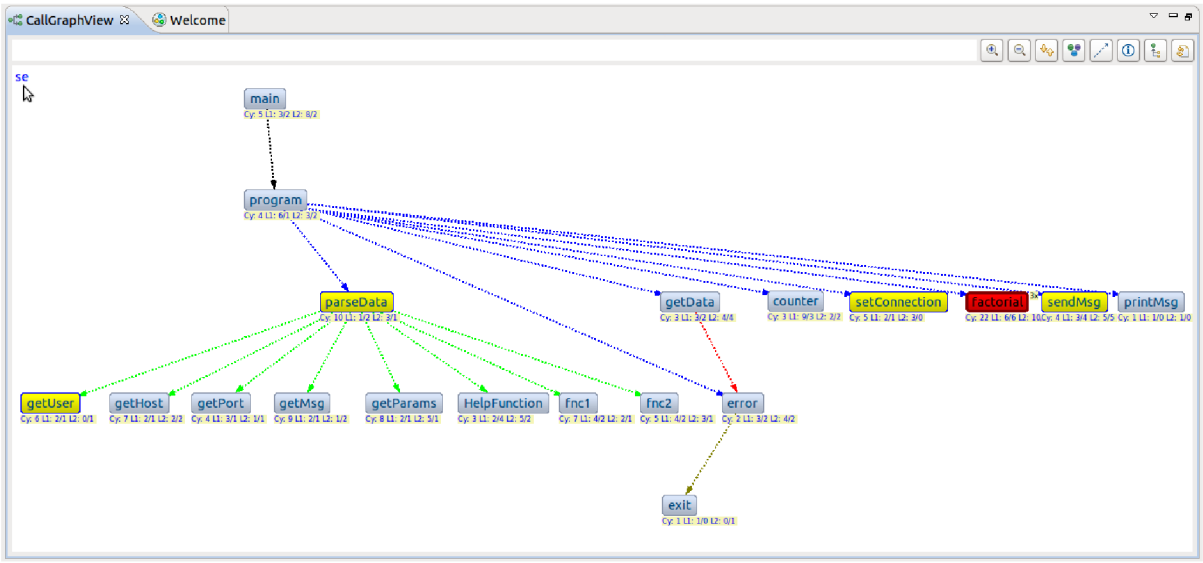
Obrázek A.6: Nastavování barev



Obrázek A.7: Zoom a vlastnosti uzlu



Obrázek A.8: Vyhledávání s nápovědou



Obrázek A.9: Rychlé vyhledávání