VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV TELEKOMUNIKACÍ
DEPARTMENT OF TELECOMMUNICATIONS

ZÁSUVNÝ MODUL PRO PROSTŘEDÍ ECLIPSE
PLUG-IN FOR ECLIPSE ENVIRONMENT

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **Vsevolod Zaytsev**
AUTHOR

VEDOUCÍ PRÁCE — **doc. Ing. Petr Sysel, Ph.D.**
SUPERVISOR

BRNO 2018

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Bakalářská práce

bakalářský studijní obor **Teleinformatika**
Ústav telekomunikací

Student: Vsevolod Zaytsev ***ID:*** 173784
Ročník: 3 ***Akademický rok:*** 2017/18

NÁZEV TÉMATU:

Zásuvný modul pro prostředí Eclipse

POKYNY PRO VYPRACOVÁNÍ:

Seznamte se s vnitřní architekturou vývojového prostředí Eclipse. Zaměřte se především na Dynamic Languages Toolkit, který umožňuje rozšiřování vývojového prostředí o podporu skriptovacích jazyků. Na základě získaných zkušeností navrhněte zásuvný modul pro podporu skriptovacího prostředí Octave (Matlab). Modul by měl minimálně umožňovat spouštět skripty uvedených jazyků a zobrazit výsledky zpracování.

DOPORUČENÁ LITERATURA:

[1] Scarpino, M.; Good, N. A. Use the Dynamic Languages Toolkit (DLTK) to create your own IDE. 2011.

[2] Java 6 výukový kurz. Brno: Computer Press, 2007. 1. vydání. ISBN 978-80-251-1575-6.

Termín zadání: 5.2.2018 ***Termín odevzdání:*** 29.5.2018

Vedoucí práce: doc. Ing. Petr Sysel, Ph.D.
Konzultant:

prof. Ing. Jiří Mišurec, CSc.
předseda oborové rady

ABSTRAKT

Tato bakalářská práce má za cíl vytvoření nového zásuvného modulu na základě Dynamic Language Toolkit (DLTK), do vývojového prostředí Eclipse. Teoretická část této práce se zabývá architekturou vývojového prostředí Eclipse a vysvětluje možnost jeho rozšíření o vlastní zásuvné moduly, zejména pro podporu skriptovacího jazyku Octave. Praktická část práce popisuje vytvoření nového zásuvného modulu pro prostředí Eclipse, jeho nasazení a aktivace.

KLÍČOVÁ SLOVA

DLTK, dynamické jazyky, Eclipse, Octave, plug-in, zásuvný modul

ABSTRACT

This bachelor's thesis has a goal to design a new DLTK-based (Dynamic Languages Toolkit) plug-in for the Eclipse development environment. Theoretical part of the thesis engages on the architecture of the Eclipse and describes the possibility of its extension through its own plug-ins, particularly for support of the Octave script language. Practical part of the thesis describes the creation of new plug-in for the Eclipse development environment, it's integration and activation.

KEYWORDS

DLTK, dynamic languages, Eclipse, Octave, plug-in

ZAYTSEV, Vsevolod. *Zásuvný modul pro prostředí Eclispe*. Brno, 2018, 57 s. Bakalářská práce. Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací. Vedoucí práce: Ing. Petr Sysel, Ph.D.

PROHLÁŠENÍ

Prohlašuji, že svou bakalářskou práci na téma „Zásuvný modul pro prostředí Eclispe“ jsem vypracoval(a) samostatně pod vedením vedoucího bakalářské práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor(ka) uvedené bakalářské práce dále prohlašuji, že v souvislosti s vytvořením této bakalářské práce jsem neporušil(a) autorská práva třetích osob, zejména jsem nezasáhl(a) nedovoleným způsobem do cizích autorských práv osobnostních a/nebo majetkových a jsem si plně vědom(a) následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

Brno ....

podpis autora(-ky)

PODĚKOVÁNÍ

Rád bych poděkoval vedoucímu semestrální práce panu Ing. Petru Syselovi, Ph.D. a odbornému programátorovi panu Bc. Pavlu Savrovovi za odborné konzultace a výpomoc v práci.

Brno ....

podpis autora(-ky)

OBSAH

SEZNAM OBRÁZKŮ

SEZNAM TABULEK

SEZNAM VÝPISŮ

ÚVOD

Vývojové prostředí *Eclipse* je velice populární a rozšířené. Je určené pro programování v jazyce Java. Ale filozofie tohoto vývojového prostředí je stavena na myšlence flexibility a rozšířitelnosti, což umožňuje zvětšit počet podporovaných jazyků.

Při studiu zejména technických oborů se nevyhnout matematickým výpočtům a simulací. K tomuto účelu se velice často využívá interaktivní programové prostředí *MATLAB*. V některých případech je vhodné použít výsledky matematických výpočtů v kódu programu. Za tímto účelem existuje zásuvný modul *Matlabu* pro *Eclipse*. Jenže *MATLAB* je program placený. Existuje ale podobné programové prostředí — *GNU Octave*. Je volné a má podobnou sadu funkcí, využívá podobnou syntaxe . Ačkoli zásuvný modul *Octave* pro *Eclipse* již existuje dlouhou dobu (od roku 2013) je ale neaktuální a nemůže spolu-fungovat se současnou verzí *Eclipse*. Proto vznikla potřeba vytvoření aktuálního zásuvného modulu, čímž se zabývá daná semestrální práce.

Zadáním bakalářské práce je návrh zásuvného modulu pro vývojové prostředí *Eclipse*, který by měl za účel navázaní spolupráce tohoto prostředí a skriptovacího jazyku *Octave*. Pro splnění požadavků byl navržen zásuvný modul „***Octo-clipse***“.

První kapitola seznamuje čtenáře s architekturou vývojového prostředí *Eclipse* a také jeho existujícími verzí. Také se stručně probírá sada nástrojů pro vývoj zásuvných modulu na základě skriptovacích jazyků — *DLTK*. Na konci se uvádí popis vývojového prostředí a skriptovacího jazyku *Octave*.

Druhá kapitola probírá mechanizmus návrhu zásuvných modulů pro vývojové prostředí *Eclipse*, a to zejména pomocí sady nástrojů pro skriptovací jazyky *DLTK*. Uvádí se základní body vývoje plug-inů tímto způsobem: *OSGi* rámec, soubory pro definici rozšíření a také závislosti zásuvného modulu.

Další, praktická část obsahuje popis vytvoření nového zásuvného modulu *Octave* pro *Eclipse (IDE)*. Vytváří se textový editor *Octave*, dělení zdrojových dokumentů, preferenční stránky a naváže se spojení mezí konzolou *Eclipse* a interpretem *GNU Octave*.

1 VÝVOJOVÉ PROSTŘEDÍ ECLIPSE

Eclipse je známým integrovaným vývojovým prostředím (*IDE — Integrated Development Environment*). Je to multiplatformní aplikace s otevřeným zdrojovým kódem, což znamená, že může být spouštěna na každém z těch nejpoužívanějších operačních systémech (Windows, Mac OS a Linux).

Eclipse je především určen pro programování v jazyce *Java*, nicméně kvůli své filozofii umožňuje rozšiřování o další nástroje, aplikace, moduly apod., za volnou dostupností zdroje. Tato možnost zvyšuje počet používaných jazyků. V současné době *Eclipse* podporuje *C/C++*, *PHP* a některé další.

1.1 Architektura vývojového prostředí *Eclipse*

Vývojové prostředí *Eclipse* je sestaveno z velkého množství zásuvných modulů, jsou to elementární stavební jednotky. Proto se mluví o „plug-in architektuře“.

Eclipse je postaven na myšlence zásuvných modulů. Zásuvné moduly jsou strukturovanými balíky kódu a/nebo daty, které přispívají k funkčností systému. Funkčnost může být podporována ve formě knihoven (Java třídy s veřejným API — Application Programming Interface), rozšíření platformy a dokonce v podobě dokumentaci. Zásuvné moduly mohou definovat rozšíření — jasně vymezená místa, kam jiné zásuvné moduly mohou přidávat své funkce.

Každý dílčí systém je sestaven ze sady zásuvných modulů a implementuje některou klíčovou funkci. Některé zásuvné moduly přidávají viditelné prvky k platformě použitím rozšiřovacího modelu. Jiné dodávají knihovny, které mohou být použité pro systémová rozšíření.

Na obr. 1.1 je znázorněno, že celé vývojové prostředí se skládá ze tří hlavních části. Jsou to:

- **Eclipse platform** — sada prostředků pro vývoj programů;
- **Java Development Tools (JDT)** — nástrojové zásuvné moduly, implementující podporu vývoje jakékoliv Java aplikací;
- **Plug-in Developer Evironment (PDE)**, poskytuje nástroje pro vytvoření, vývoj, testováni, ladění a nasazení Eclipse pluginů.

Eclipse SDK (Software Development Kit) zahrnuje základní platformu a navíc dva hlavní nástroje, užitečné pro vývoj zásuvných modulů.

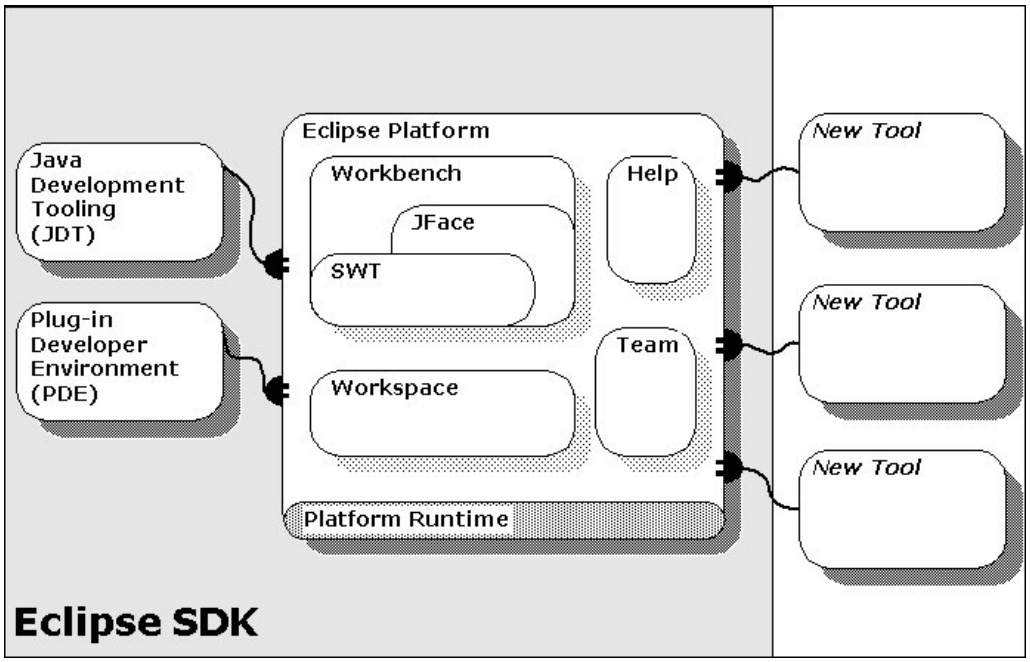
Java Development Tools — implementuje plně funkční návrhové prostředí Java *(IDE)*. Umožňuje uživateli psát, kompilovat, testovat, ladit a upravovat programy napsané v programovacím jazyce *Java*.

(JDT) si využívá mnoha rozšíření a rámců popsaných v *Platform Plug-in Developer Guide* [1]. Je možné chápat *JDT* jako sadu modulů, které dodává určité chování

do modelu základních zdrojů platformy a poskytuje *JVM* specifické pohledy, editory a akce pro *Workbench* (viz dál.).

Plug-in Development Environment (PDE) přidává specifikované nástroje, které zefektivňuje vývoj zásuvných modulů a rozšíření. Pak se skládá z dalších dílčích modulů.

Této nástroje slouží nejen k užitečnému účelu, ale také ukazují, jak nové nástroje mohou být integrované do platformy pomocí návrhu zásuvných modulů.



Obr. 1.1: Architektura vývojového prostředí Eclipse.

Je to kostra projektu *Eclipse*, kde ***Eclipse Platform*** je nejdůležitějším dílem. Obsahuje jenom nezbytné funkce. Skládá se z dalších části:

- *Workbench*, vývojové prostředí pro integraci nástrojů, správu a navigaci zdrojů pracovního prostoru, např.:
  - *editory* (editors) — umožňují uživateli provádět změny v *Workbench*;
  - *pohledy* (views) — poskytují informaci o některém z objektů, kterým uživatel právě pracuje v *Workbench*;
  - *perspektivy* (perspectives) — výchozí perspektivou je *Java Perspective*, hlavní rozhraní *JDT* projektu v *Eclipse*, která se používá ve vývoji agentské aplikace spojené s *Java Virtual Machine* pro synchronizaci dat;
  - *menu lišta* (menu bar) — představuje základní prvky, jakými jsou Soubor (File), Upravit (Edit), Nápověda (Help) a apod.;
  - *nástrojová lišta* (toolbar) — umožňuje rychlý přístup k různým příkazům.
- *Workspace*, pracovní prostor.

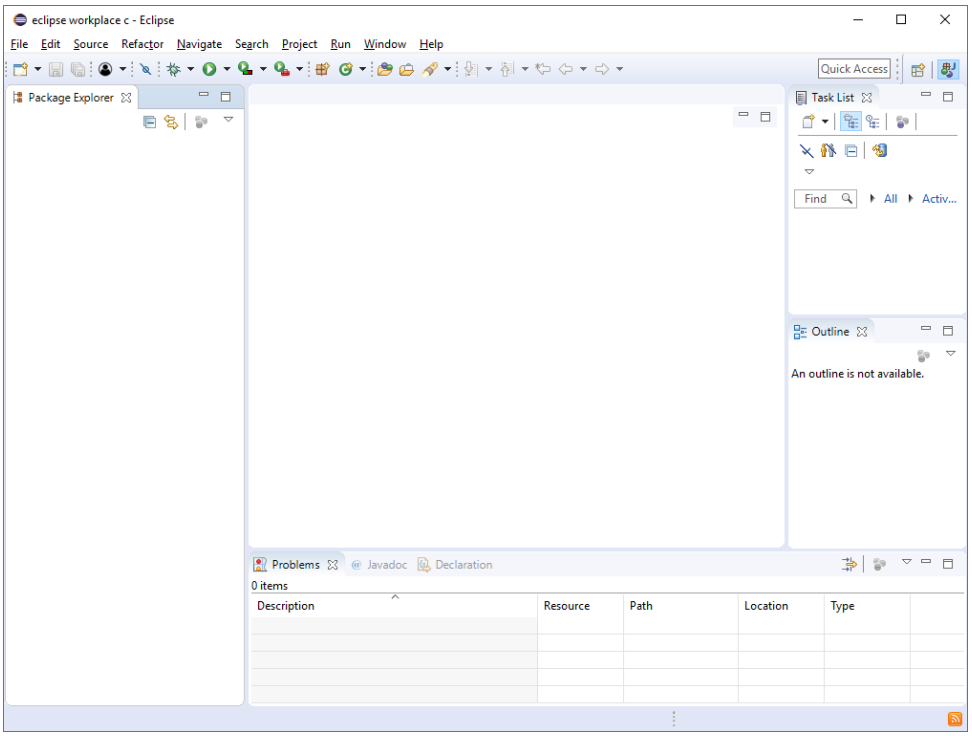
- *Platform Runtime*, definuje moduly, na nichž jsou všechny ostatní moduly závislé.

Workbench se vztahuje k plošnému vývojovému prostředí. Má za úkol dosáhnout souvislé integraci nástrojů a kontrolované otevřenosti přes nabídku běžného modelu vytváření, řízení a zdrojové navigace.

Jakmile se spouští *Workbench*, první se zobrazí dialogové okno, které umožňuje zvolit kde se *Workplace* má nacházet. *Workplace* je to adresář, kde Váše projekty budou ukládaný. Zde se budou ukládat všechna uživatelská data. Je možné vyčlenit tři základní typy zdrojů pracovního prostředí: projekty, adresáře a soubory.

Projekty jsou největší strukturní jednotkou, kterou pracovní prostředí používá. Obsahují adresáře a soubory, je možné jich otevírat, zavírat, anebo kompilovat. ***Adresáře*** mohou obsahovat další adresáře a soubory. ***Soubory*** obsahují určitá data, popisující chování projektu nebo jeho části.

Potom co adresář *Workspace* je vybrán, se zobrazí jednotlivé *Worbench* okno (obr. 1.2). Toto okno nabízí jednu anebo více perspektiv. Perspektivy obsahují editory a pohledy, jakým např. je *Package Explorer*. Souběžně mohou být otevřené několik *Workbench* oken. Dá se lehce mezi nimi přepínat a provádět činnosti v několika pracovních prostředích paralelně. *Eclipse* je ale schopen pracovat jenom v jednom *workbench* okně. Každý pracovní prostor může být nastaven individuálně, včetně umístění jednotlivých pohledů, editorů a také klávesových zkratek.



Obr. 1.2: Vizuální struktura vývojového prostředí Eclipse.

Platform Runtime definuje org.eclipse.osgi a org.eclipse.core.runtime zásuvné moduly, na kterých všechny ostatní moduly závisí. Toto běhové prostředí je zodpovědné za definici a strukturu zásuvných modulů, jejich detaily (balíky a načítaní tříd) za běhu *JVM (Java Virtual Machine)*. *Platform Runtime* je také zodpovědné za hledání a vykonání hlavní aplikace *Eclipse*, udržbu registru zásuvných modulů a jejich rozšíření.

1.2 Verze *Eclipse*

Projekt *Eclipse* má velké množství připravených paketů pro různé cíle. V dnešní době existují další verze *Eclipse*:

- **Eclipse IDE for Java Developers**
- **Eclipse IDE for Java EE Developers**
- **Eclipse IDE for C/C++ Developers**
- **Eclipse IDE for Eclipse Committers**
- **Eclipse for PHP Developers**
- **Eclipse IDE for JavaScript and Web Developers**
- **Eclipse IDE for Java and DSL Developers**
- **Eclipse Modeling Tools**
- **Eclipse for RCP and RAP Developers**
- **Eclipse for Parallel Application Developers**
- **Eclipse for Scout Developers**

Je vždy možné doinstalovat potřebné rozšíření do již existující soupravy. Pro bakalářskou práci je potřeba mít verzi *Eclipse for RCP and RAP Developers* s doinstalovaným nástrojem *Dynamic Languages Toolkit (DLTK)*. Toto rozšíření obsahuje kompletní sadu nástrojů pro vývoj zásuvných modulů pro Eclipse, aplikace pro koncové uživatele anebo aplikace se vzdálenou platformou (*RCP+RAP*). Navíc, do modulu jsou také zahrnuty *Maven* a *Gradle* nástroje, *XML* editor a *EGit* nástroje pro přístup k repositářům systému pro správu verzí Git.

1.3 *Dynamic Languages Toolkit (DLTK)*

Dynamic Languages Toolkit (DLTK) je nástrojem vhodným pro prodejce, výzkumníky a koncové uživatele, kteří používají dynamické programovací jazyky. *DLTK* je sestaven ze sady rozšiřitelných struktur, navržených pro snížení složitého vytváření plně upravených vývojových prostředí pro dynamické jazyky, jakými jsou *PHP* a *Perl*. Mimo to sada struktur DLTK poskytuje ukázkové šablony pro vytvoření vývojového prostředí pro jazyky *Tcl*, *Ruby* a *Python*.

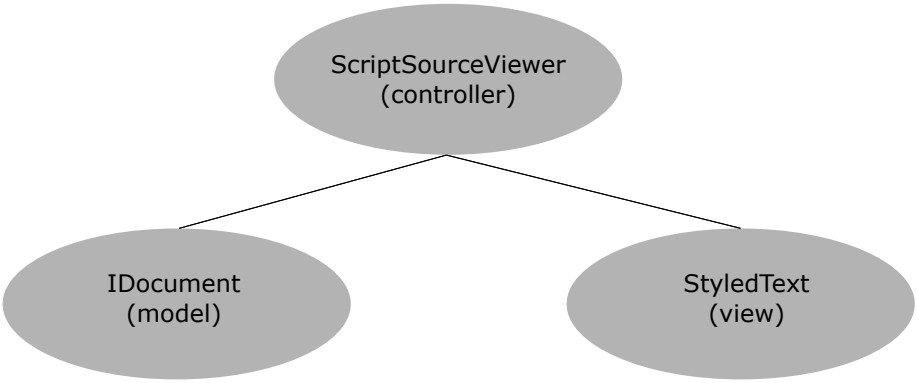
1.3.1 DLTK textový editor

Zjednodušeně řečeno, textový editor DLTK je běžným widgetovým (*SWT — Standard Widget Toolkit*) nástrojem pro stylizaci s mnoha doplňkovými vlastnostmi. Této vlastnosti spadají do jedné z následujících kategorii:

- **Responding to user events**, odezva na akce uživatele. Změna obrazovky editoru podle stisknutí kláves uživatelem.
- **File I/O**, souborový vstup a výstup. Přenos dat mezi vývojovým prostředím a soubory v systému uživatele.

V *DLTK* editorech probíhající událost se řídí objektem `ScriptSourceViewer`. Tento objekt, vytvořený `ScriptEditor`'em, využívá widget `StyledText` a řídí jeho odezvu na stisknutí uživatele. Typicky, tato odezva zapojí změnu prezentace widgetu nebo aktualizace vnitřních dat editoru. Tato data jsou zahrnuta v objektu `IDocument`, který nejenom ukládá text, ale také informaci spojenou s čísly, pozicemi, a regiony, tzv. *partitions*, linek.

`ScriptSourceViewer` aktualizuje `IDocument`. Udělá to tak, že získá objekt, řešící souborové operace — `SourceModuleDocumentProvider`. DLTK se stará o všechny interakce souborů editoru.



Obr. 1.3: Základní elementy DLTK editoru.

Na obr. 1.3 jsou znázorněny vztahy uvnitř *DLTK* editoru. `IDocument` slouží jako model editoru (*Model*), obsahující data nezávisle na jejich prezentaci. `StyledText` widget reprezentuje pohledový aspekt editoru (*View*), a komponenta prohlížeče `ScriptSourceViewer` slouží jako kontrolér, řídící komunikaci mezi modelem, pohledem a uživatelem. Informace o *DLTK* struktuře a funkcích jeho nástrojů je vzata z manuální stránky vytvoření pluginu *Octave* [3].

1.4 Octave — skriptovací jazyk

Octave — je vědeckým programovacím jazykem. Má mocnou matematicky orientovanou syntaxi s vystaveným kreslením a vizualizací nástrojů. Toto programové vybavení je volně dostupné a spustitelné na *GNU/Linux*, *macOS*, *BSD* a *Windows* systémech. Je kompatibilní s mnoha skripty *Matlabu*, což staví ho do výhodné pozice vůči němu, protože je neplacený.

Octave nabízí pohodlné rozhraní pro řešení lineárních a nelineárních problémů v podobě čísel, a vykonání dalších numerických experimentů, pomocí jazyka, který je převážně kompatibilní s Matlab'em. Může být taky použít jako objektově orientovaný jazyk.

Octave má rozsáhlé nástroje na řešení obecných lineárních algebraických problémů, hledání kořenů nelineárních rovnic, integraci obyčejných funkcí, manipulaci s mnohočleny, a integraci obyčejných diferenciálních a diferenciálně algebraických rovnic. Je to snadno rozšířitelné a uzpůsobitelné prostředí dle potřeb uživatele, přes jím definované funkce, napsanými ve vlastním jazyce *Octave*, anebo pomocí použití dynamicky nahrávaných modulů napsaných v jazycích *C++*, *C*, *Fortan*, nebo dalších.

Octave interpret může být spouštěn v režimu uživatelského rozhraní (*GUI*), konzoli anebo může být zavolán jako část skriptu.

2 VYTVOŘENÍ ZÁSUVNÉHO MODULU

Navrhnout nový zásuvný modul je celkem jednoduché. Obecně řečeno, stačí implementovat vlastnosti, použitím mnohonásobných modulů: jeden zahrnující třídy jádra, jeden zahrnující třídy uživatelského rozhraní, jeden zahrnující třídy ladění, atd.

Nejdůležitějšími soubory v celém projektu zásuvného modulu jsou: `MANIFEST.MF` a `plugin.xml`.

`MANIFEST.MF` je metadata[1] souborem nacházejícím v JAR souboru[2]. Definuje rozšíření a objektově závislá data. Obsahuje páry jméno-hodnota uspořádané do sekcí. Pokud JAR soubor je určený pro použití jako spustitelný soubor, manifestový soubor specifikuje hlavní třídu aplikace. Manifestový soubor má být prvním vstupem do komprimovaného souboru.

Zásuvný modul je popsán v souboru `plugin.xml`, který je součástí rozmístění souborů. Manifestový soubor říká běhovému prostředí aplikace co potřebuje vědět aby zaregistrovalo a aktivovalo plugin. Manifestový soubor v podstatě slouží jako smlouva mezi zásuvnými komponenty a běhovým prostředím.

2.1 Založení nového DLTK projektu

Daná bakalářská práce vyžaduje přítomnost zásuvných modulů **PDE (Plug-in Development Environment)** a **DLTK (Dynamic Languages Toolkit)**. Zde jsou použité Dynamic Languages Toolkit verze 5.8.0.201706030400 a Eclipse Plug-in Development Environment verze 3.13.1.v20171009-0537. Jejích aktuální verze můžete si stáhnout a nainstalovat pomocí zdrojů [4] a [5]. Počáteční postup vytvoření nového DLTK projektu je popsán v přílohách, části A.

2.2 OSGi Framework

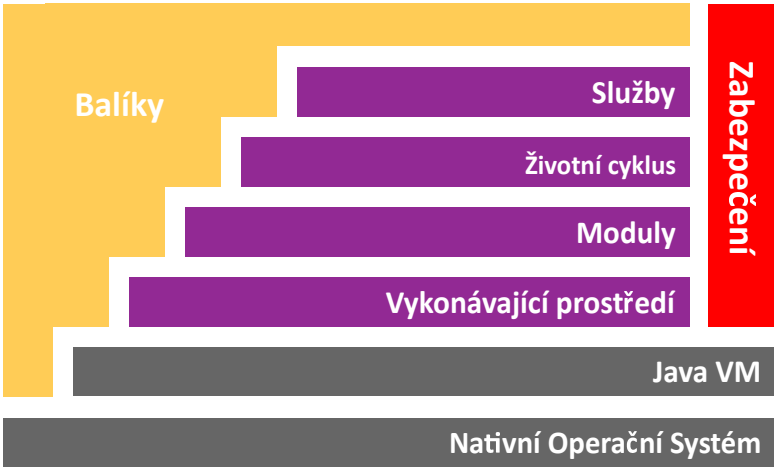
Za prvé je potřeba vědět, co to je *OSGi* rámec (*OSGi Framework*), protože běhové prostředí *Eclipse* je založeno na této technologii.

Technologie *OSGi* [6] usnadňuje představu komponent softwarových modulů a programů, zajišťuje vzdálené řízení, kompatibilitu aplikací a služeb na velkém množství zařízení. Obecně řečeno, je to sada specifikací, definující dynamický systém *Java*.

OSGi specifikace umožňuje komponentům schovávat jejích implementace před ostatními účastníky během komunikace služeb, které objekty sdělují mezi komponenty. Má následující architekturu (viz obr. 2.1):

[1] Metadata — data poskytující informace o dalších datech

[2] JAR — Java ARchive, souborový formát založený na populárním ZIP formátu



Obr. 2.1: Vrstvový model architektury *OSGi* rámce.

- Balíky — *OSGi* komponenty, vytvořené vývojáři;
- Služby — spojuje balíky dynamicky tak, že nabízí model „vydej-vyhledej-svaž“ pro obyčejné *Java* třídy;
- Životní cyklus — *API* pro instalace, spouštění, zastavení, aktualizace a odinstalování balíků;
- Moduly — vrstva definující jak balík může importovat a exportovat kód;
- Zabezpečení — vrstva udržující aspekty bezpečnosti;
- Vykonávající prostředí — definuje jaké metody a třídy jsou dostupné v určité platformě.

Mezi projekty pracující na *OSGi* rámci se vyznačuje jeden, který je základem celého *Eclipse IDE* — ***Eclipse Equinox*** [7]. Z pohledu kódu, *Equinox* je implementací *OSGi* rámce, sadou balíků, které realizují různé volitelné *OSGi* služby a další infrastruktury pro běh systémů založených na této technologii. Každá dílčí část *Eclipse IDE* je tvořena *OSGi* balíkem (*JDT*, *PDE*, *SWT*, *JFace* a další) a při spouštění vývojového prostředí jsou všechny části pomocí specifikace *Eclipse Equinox* poskládány do výsledného běhového prostředí *Eclipse IDE*.

2.3 Plug-in manifest

`MANIFEST.MF` a `plugin.xml` soubory definují jak se daný zásuvný modul vztahuje ke všem ostatním souborům systému. *OSGi* rámec definuje sadu hlaviček manifestu, jakými jsou exportované balíky (`Export-Package`) a nastavení cesty balíku (`Bundle-Classpath`), které vývojáře balíků používají pro dodání popisné informace o balíku. *Equinox OSGi* rámec implementuje úplnou specifikaci rámce *OSGi* a všechny služby rámce jádra (*Core Framework services*), které zahrnují:

- *Package Admin Service Specification* (neschvalovaný) — uvažuje nad použitím nového `org.osgi.framework.wiring` balíku;
- *URL Handlers Service Specification*;
- *Start Level Service Specification* (neschvalovaný) - uvažuje nad použitím nového `org.osgi.framework.startlevel` balíku;
- *Conditional Permission Admin Specification*;
- *Permission Admin Service Specification*.

Každý `MANIFEST.MF` soubor obsahuje vstupy pro názvy, identifikátory, verze, aktivátory a poskytovatele.

Výpis 2.1: Vstupy souboru `MANIFEST.MF` na příkladě projektu bakalářské práce.

```
1  Bundle-Name: Console
2  Bundle-SymbolicName: com.zajcev.octoclipse.console
3  Bundle-Version: 1.0.0.201712130714
4  Bundle-Activator: com.zajcev.octoclipse.console.
     OctConsolePlugin
5  Bundle-Vendor: ZAJCEV
```

Identifikátor zásuvného modulu (`Bundle-SymbolicName`) je navržen, aby unikátně identifikoval plug-in a je typicky konstruován využitím mechanizmu jmenování *Java* balíků (např., `com.<companyName>.<productName>`, anebo jak v daném příkladě, `com.zajcev.octclipse.console`). Pokud více zásuvných modulů jsou součástí jednoho produktu, každý plug-in může mít 4 nebo 5 částí, jak to je např. u `com.zajcev.octclipse.console.history`.

Každý zásuvný modul specifikuje svou *verzi* (`Bundle-Version`) pomocí tří čísel rozdělených periodami. První a druhé čísla indikují hlavní a vedlejší čísla verzi, třetí — číslo revize. Je možné specifikovat volitelný kvalifikátor, který může obsahovat alfanumerické znaky, např. `1.0.0.beta_1` nebo `1.0.0.2008-06-26` (bez mezer). Pokud při spouštění prostředí *Eclipse* existují několik plug-nů se stejnými identifikátory, zvolí se ten nejaktuálnější tak, že se postupně porovnají hlavní, vedlejší a číslo revize (a taky kvalifikátor, jestli je).

Jméno zásuvného modulu (`Bundle-Name`) a *jméno dodavce* (`Bundle-Vendor`) jsou čitelnými texty, proto nemusí být unikátními.

Aktivátor zásuvného modulu (`Bundle-Activator`) se specifikuje volitelně. Podle výchozího nastavení poskytuje metody pro přístup k statickým zdrojům uvnitř pluginu, a také pro přístup a inicializaci typických pro zásuvné moduly možnosti a další stavové informace.

Veškeré nastavení se provádí v `plugin.xml` souboru, kde se popisuje o jaké další moduly musí být rozšířen daný projekt. Díky tomu jiné plug-iny jsou schopní rozšířit funkčnost daného zásuvného modulu kontrolovaným způsobem. Tento mechanizmus

poskytuje rozdělovací vrstvu, díky které originální zásuvný modul nepotřebuje vědět ob existenci rozšířujících modulů v okamžik, když se plug-in kompiluje. Zásuvné moduly deklarují rozšíření (*extension points)* jako část svých plug-in manifestů.

2.4 Plug-in dependencies

Nakladač zásuvných modulů (*plug-in loader*) ilustruje samostatnou třídu pro každý nakládaný modul, a využívá `Requier-Bundle` deklaraci manifestu, aby určil jaké další moduly, tudíž třídy, budou viditelné pro daný modul během jeho konání.

Výpis 2.2: Příklad závislostí zásuvného modulu.

```
1 Require-Bundle: org.eclipse.ui,
2 org.eclipse.core.runtime
```

Závislosti nezbytné pro bakalářskou práci jsou uvedené v příloze C.1.

Když se nakladač chystá k nakládaní zásuvného modulu, skenuje v popise *Require-Bundle* závislé plug-iny a určuje polohy požadovaných zásuvných modulů.

Další položkou závislostí manifestu je `Import-Package`, je podobná položce předchozí, jenže importované balíky specifikují požadované služby, když `Requier-Bundle` specifikují poskytovatele služby.

3 KONFIGURACE ZÁSUVNÉHO MODULU OCTO-CLIPSE

Tato kapitola bakalářské práce se zabývá samotným vývojem zásuvného modulu *Octo-clipse*. Dále zde se budou popsány jednotlivá rozšíření a třídy, které tato rozšíření deklaruje.

3.1 Octave editor

Prvním důležitým rozšířením je editor (viz výp. 3.1).

Toto rozšíření obsahuje několik důležitých atributů. Atribut `class` stanoví hlavní třídu, která zahrnuje editor: `OctaveEditor`, který bude vytvořen později. Atribut `contributorClass` identifikuje třídu, která poskytuje činy editorovi. Atribut `default` tvrdí, že `OtaveEditor` má být implicitním editorem pro související soubory. Atribut `name` identifikuje text, kterým editor bude označen. Jak je definováno v `plugin.properties` souboru, vlastnost `OctaveEditor.name` odpovídá editoru skriptů *Octave*.

Výpis 3.1: Rozšíření vývojového prostředí o editor zdrojových souborů *Octave*.

```
1  <extension point="org.eclipse.ui.editors">
2    <editor
3      id="org.eclipse.dltk.octoclipse.editor.OctaveEditor"
4      class="org.eclipse.dltk.octoclipse.editor.
         OctaveEditor)"
5      contributorClass="org.eclipse.dltk.actions.
         OctaveActionContributor"
6      default="true"
7      icon="icons/oct.gif"
8      name="OctaveEditor">
9      <contentTypeBinding
10       contentTypeId="org.eclipse.dltk.octaveContentType">
11     </contentTypeBinding>
12   </editor>
13 </extension>
```

Poslední položka v rozšíření, `contentTypeBinding`, vyžaduje vysvetlení. *Eclipse* udržuje rejstřík typů obsahu, které reprezentují souborové formáty a pojmenování. Přidáním elementu `contentTypeBinding` do editorového rozšíření máte možnost

spojit editor se souborovými příponami, jmény souborů, a taky s odkazy na soubory. Ale pro *Octave* potřebujeme jenom soubory s rozšířením `*.m`.

3.2 Content Type

Rozšíření `contentTypes` umožňuje vývojovému prostředí *Eclipse* identifikovat, že konkrétní soubor má určitý obsah, v našem případě je to kód *Octave*. Identifikace se provede buď kontrolou souborové přípony, nebo jeho obsahu. Přípona souboru jazyku *Octave* je `*.m` (stejně jak u *Matlab*) a je specifikována v atributu `file-extensions` (viz výp. B.1 v přílohách).

Atribut `priority` oznamuje, že pokud soubor bude asociován s více typy, daný typ si dostane vyšší prioritu.

3.3 Project Nature

Pracovní prostor podporuje pojem *project natures* (*„natures“* pro zkrácení). *Nature* spojuje chování životního cyklu s projektem. Jsou nainstalovány na základě projektů pomocí použití metody `setDescription` definované v třídě `IProject` z balíku `org.eclipse.core.resources` (viz výp. 3.2).

Natures se sestavují, když je přidáme do projektu, a naopak, rozebírají se, když je odstraníme z projektu. Tato rozšíření umožňují vývojářům zaregistrovat jejich *nature* implementace pod symbolickým jménem, které se potom používá v pracovním prostředí pro hledání a konfiguraci.

Výpis 3.2: Project nature.

```
1 <extension id="nature" point="org.eclipse.core.resources.
    natures">
2   <runtime>
3     <run class="com.zajcev.octoclipse.core.OctaveNature">
4   </runtime>
5 </extension>
```

`OctaveNature.java` rozšiřuje třídu `ScriptNature`, která má nástroje pro řízení a nastavení implementujícího builderu projektu.

Atribut `describer` slouží k analýze obsahu souboru (převážně pro soubory, které nemají příponu). Např. je možné zkontrolovat, jestli prvním řádkem ve souboru je *„#\!/usr/bin/octave“*.

3.4 DLTK Language Toolkit

Třída `OctaveLanguageToolkit` je jádrem skriptovacího prostředí založeného na *DLTK*. Je vstupním bodem pro přístup ke všem vlastnostem specifickým pro daný používaný jazyk. *Language Toolkit* nástroj je přidán do projektu pomocí rozšíření `org.eclipse.dltk.language` (viz výp. B.2 v přílohách).

Pomocí metod z rozhraní *DLTK Language Toolkit* (`IDLTKCoreLanguageToolkit`, `IDLTKUILanguageToolkit`, atd.) jsme schopni nastavit funkce vývojového prostředí. Dá se nastavit mnoho aspektů: rozdělení řádků, použití barev, uživatelská nastavení a přístup k interpretu. V tabulce C.2 (viz přílohy) jsou popsány některé z těchto metod.

Všechny metody v rozhraní `IDLTKUILanguageToolkit` patří do jedné ze dvou skupin: buď vrací třídu, která ovládá nastavení editoru, anebo poskytují ID konfigurace objektu stanoveného v `plugin.xml`.

3.5 Vytvoření Octave editoru a textových nástrojů

Dále je nutné vytvořit třídu `OctaveEditor`. `ScriptEditor` (to je rodičovskou třídou `OctaveEditor`'u) ováda většinu operací. Všechno, co je potřeba — několik konfiguračních metod. V `OctaveEditor`'u nejdůležitějšími z nich je `getTextTools()`, která vrací `ScriptTextTools` objekt.

Třída `ScriptTextTools` je velice pohodlná. Tato hlavní třída využívá několik jiných tříd, které poskytují schopnosti editoru. Pomocí dělení na podtřídy editor může uzpůsobit představení těchto schopností. *Octave* editor vytváří podtřídu `OctaveTextTools`, která poskytuje přístup k důležitým objektům uvedeným v tabulce C.3 (viz přílohy).

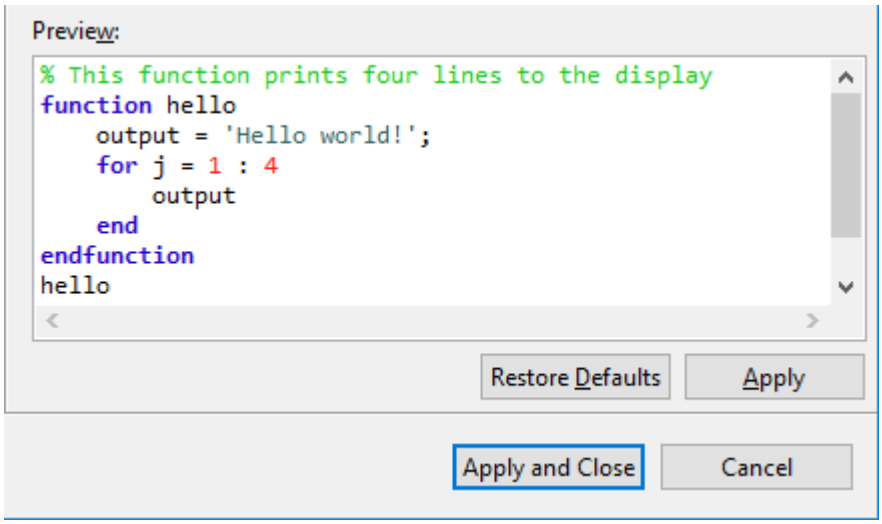
Tyto objekty se stávají důležitými při implementaci zvýraznění syntaxe obarvením textu.

3.6 Zvýraznění syntaxe v DLTK editoru

Zvýraznění syntaxe je jedním z nejpatrnějších aspektů jakýchkoliv profesionálních editorů kódu. Zvýraznění syntaxe nejen zvýrazňuje kód, ale dělá chyby zcela evidentní. Obrázek 3.1 ukazuje jak se komentáře, čísla, klíčová slova a řetězce zvýrazňují v kódu.

Postup zvýraznění syntaxe *DLTK* má 4 kroky:

1. `ScriptSourceViewer` reaguje na stisknutí upozornění děliče dokumentu.



Obr. 3.1: Příklad zvýraznění syntaxe ve vývojovém prostředí Octave.

2. Dělič dokumentu dělí dokument na části dle pravidel.
3. Když se rozdělení skončí, prohlížeč upozorní dělič, že má analyzovat změněnou část dokumentu.
4. Dělič zavolá *damager/repairer* opraváře, aby použil zvýraznění syntaxe.

Tímto způsobem editor používá strategii *divide-and-conquer* (rozděl-a-zvítěz) aby obarvil text. Za prvé, tímto se dokument rozdělí na části. Poté, editor zanalyzuje změněnou část a určí, kde se zvýraznění syntaxe má zaktualizovat, jestli vůbec má.

3.6.1 Barevné konstanty

Budeme používat tři typy zvýraznění: klíčová slova, řetězce a komentáře. Jakékoliv další typy je možné lehce přidat. *DLTK* poskytuje sadu základních tříd pro konfiguraci jednoduchého zvýraznění.

Je nutné vyplnit několik kroků:

- Stanovit barevné konstanty,
- přidat rozšíření *org.eclipse.core.runtime.preferences* a nastavit hodnoty pro barevné konstanty,
- Definovat pravidla pro zvýraznění,
- přidat konfigurační stránky zvýraznění, použitím jednoduchých *DLTK* tříd.

Klíčová slova je možné uvést v nějaké třídě anebo v samostatném souboru, jak je to udělané v projektu dané bakalářské práce (`keywords.txt` v kořenovém adresáře projektu).

3.7 Rozdělení dokumentu (document partitioning)

Navíc k textu, `IDocument` ukládá pole (array) pozic, které slouží jako hranice oblastí uvnitř textu. Tyto oblasti umožňují nejen zvýraznění syntaxe, ale také použit různé nástroje pro různé části textu. Znamená, že analyzátor skript nemusí zpracovávat zakomentované řádky a může se soustředit na skutečném kódu.

Rozdělení dokumentu je dosaženo pomocí dvou objektů: jeden implementuje `IDocumentPartitioner`, druhý — `IPartitioningTokenScanner`. Rozdělovač dodá text skeneru, který ho zanalyzuje a vyrobí `ITokens` (elementární klíčové symboly). Rozdělovač používá tyto tokeny pro stanovení pozic uvnitř dokumentu.

Bylo by dobré, kdyby `OctaveEditor` automaticky inicializoval rozdělovač kdykoli uživatel otevírá soubor s příponou `*.m`. Ale konfigurace dokumentu musí být jasně definována pomocí rozšíření `org.eclipse.core.filebuffers.documentSetup` (viz výp. B.3 v přílohách). Toto rozšíření identifikuje třídu, která se musí být zavolána, aby nakonfigurovala dokument během jeho inicializace.

`OctaveDocumentSetup` volá objekt `ScriptTextTools` pro inicializaci `*.m` dokumentů. Automaticky se vytvoří `FastPartitioner`, který bude sloužit jako rozdělovač dokumentu.

Každý *DLTK* editor má vytvořit vlastní objekt `IPartitionerTokenScanner` pročtení znaků a vytvoření příslušných tokenů. Například, pokud si chcete vytvořit rozdělení textu, který se začíná symboly „*/“ a ukončuje „*/“, potřebujete nakonfigurovat skener tak, aby vytvořil příslušné tokeny, když se narazí na tuto posloupnost. V tab. C.4 (viz přílohy) je popsáno, jak *Octo-clipse* dělí text do tří segmenty: komentáře, řetězce a samotný kód.

Tyto segmenty jsou definované v rozhraní `IOctavePartitions`. Používají se pro inicializaci `FastPartiotioner`, který jich posílá do skeneru segmentů.

Segmenty *Octave* editoru mohou být definované pomocí sady logických pravidel, takto `OctavePartiotionerScanner` rozšíří třídu `RuleBasedPartiotionScanner`. Nejdůležitější metodou této třídy je `setPartiotionRules`, která akceptuje pole `IPredicateRule` objektů. Cílem těchto pravidel je převod segmentů na množinu `ITokens`. Je těžko kódovat pravidla úplně od začátku, ale naštěstí *Eclipse* poskytuje mnoho užitečných implementací rozhraní `IPredicateRule`. Vývojové prostředí *Octave* používá dvě z nich pro rozpoznávaní segmentů: `SingleLineRule` a `MultiLineRule`. Obě dvě třídy mají podobné konstruktory, a jsou inicializované pomocí následujících informací:

- Znak, kterým se začíná segment;
- Znak, kterým se segment ukončí;
- Příslušný k určenému segmentu `IToken`.

Ve výp. B.4 (viz přílohy) je ukázáno jak se vytvoří pravidla pro řetězce a ko-

mentáře. Token řetězců vznikne kdykoli skener se narazí na jednoduché uvozovky ('řetězec'). Komentář se pozná pomocí znaku procenta a konce řádku („%“ a „\n“).

Tím, že jsme vytvořili `OctavePartiotionerScanner` a inicializovali jeho pravidla, ukončili jsme konfiguraci rozdělení textu editorem. Dalším krokem musíme říct editorovi, jak má skenovat a zvýrazňovat text uvnitř segmentů.

3.8 Analýza textu uvnitř segmentů

Jak jsme si řekli, *DLTK* editory jsou závislé na objektu `ScriptSourceViewer`, kvůli čemuž jsou schopní reagovat na události uživatele (např. úhozy na klávesnici). Je možné upravovat operace prohlížeče pomocí `ScriptSourceViewerConfiguration` objektu. Stejně jako třída `ScriptTextTools`, konfigurační objekt dělá jen malou práci. Místo toho, jeho metody identifikují třídy, které usnadňují provoz zdrojového prohlížeče. Když jde o zvýrazňování syntaxe, existují dvě nejdůležitější metody: `getPresentationReconciler()` a `initializeScanners()` (viz tab. C.5).

Tyto dvě metody těsně souvisí. Dále probereme, jak jsou implementované vývojovým prostředím *Octave* pro realizaci zvýrazňování syntaxe.

3.8.1 `IPresentationReconciler`

V jazyce *Eclipse reconciler* je objektem, který sleduje obsah dokumentu a reaguje na změny (tzv. synchronizátor). Reprezentace takového synchronizátoru je *reconciler*, který reaguje na změny v dokumentu úpravou reprezentaci textu. Přesněji řečeno, `IPresentationReconciler` zavolá objekty ze dvou tříd: `IPresentationDamager` a `IPresentationRepairer` (viz tab. C.6 v přílohách).

Tyto dva objekty operují jeden po druhém. Když uživatel upraví text v segmentu, objekt `IPresentationConciler` předává jméno tohoto segmentu do objektu `IPresentationDamager`. Potom co tento objekt identifikuje region, který byl modifikován, `IPresentationRepairer` ho analyzuje, aby se zjistilo, kde a jak reprezentace textu musí být zaktualizována.

Ve smyslu jednoduchosti, *Eclipse* poskytuje `DefaultDamagerRepairer`, který slouží jako škůdce (*damager*) a opravář (*repairer*). Vývojové prostředí *Octave* sestavuje `DefaultDamagerRepairer` pro každý ze třech typů segmentů a inicializuje každý s jiným skenerem. Takový synchronizátor je ukázán ve výp. B.5.

Stejně jak skener segmentů identifikuje samotné segmenty použitím pravidel, každý z těchto skenerů vytváří tokeny podle podobných pravidel. Další podsekce podrobněji prodiskutuje textové skenery.

3.8.2 Skenery textu

Pro vytvoření skeneru konstruktor `ScriptSourceViewerConfiguration` volá metodu `initializeScanners()`, které potřebujeme pro analyzovaní textu. Implementace metody `initializeScanners` z třídy `OctaveSourceViewerConfiguration` reprezentuje výp. B.6 (viz přílohy).

Kvůli tomu, že skener segmentů již identifikoval řetězce a komentáře, tyto segmenty se nemusí znovu analyzovat. A proto `stringScanner` a `commentScanner` patří k `SingleTokenScriptScanners`. Tato třída nerozhoduje na základě pravidel, ale vždycky vrací stejný typ `IToken`.

Objekt `codeScanner` je víc zapojený do práce a vykonává většinu textové analýzy *IDE*. Je to `OctaveCodeScanner` a stejně jak `PartitionTokenScanner` vytváří tokeny na základě pravidel. Namísto ale `MultiLineRule` nebo `SingleLineRule` spoléhá na implementaci `IRule` (viz tab. C.7 v přílohách).

Tato pravidla se vytváří metodou `createRules` z třídy `OctaveCodeScanner` (viz výp. B.8 přílohách)

Je důležité rozlišovat tokeny vytvořené ve výpisu B.8 od těch, co byly vygenerovány ve výpisu B.4. Konstruktor tokenů akceptuje jakýkoli objekt jako svůj argument, tento objekt říká svému příjemci jaké akce má provést. Tokeny z výpisu B.4 jsou inicializovány pomocí jmen segmentů, které říkají objektu `FastPartiotioner`, jak má zaktualizovat rozdělení dokumentu. Ve výpisu B.8, tokeny jsou inicializovány pomocí objektů `TextPresentation`, které říkají vývojovému prostředí jak se text má zobrazovat v editoru.

Kód `OctaveCodeScanner` volá metodu `getToken()`, aby každý token byl asociován s objektem `TextPresentation`. Tato metoda hledá ve vytvořeném uživatelem seznamu preference. Pro lepší pochopení jak se tento seznam aktualizuje je důležité pochopit jak vůbec fungují preference *DLTK*.

3.9 Konfigurace *DLTK* preferencí

Předchozí diskuse ukázala, jak reprezentace textu závisí na preferenci uživatele. Daná sekce ukáže, jak vytvořit stránky, které budou obdržovat preference uživatele. Dostat se k těm stránkám lze přes **`Window>Preferences`**. Na konci dané sekce se vysvětlí jak je možné integrovat interpret *Octave* do *Eclipse IDE* pomocí uživatelských preferencí.

3.9.1 Preference store

Je to tak zvaný sklad preferencí a je podobný souboru s vlastnostmi: obsahuje páry jméno-hodnota, které identifikují preferována uživatelem nastavení. Jsou tady ale

dva důležité rozdíly. Navíc ke skutečné hodnotě, každá preference drží svou výchozí hodnotu, a každopádně musí patřit k jednomu ze základních typů:

- `Boolean`
- `int`
- `long`
- `float`
- `double`
- `String`

Preference se skladují a získávají voláním metod z rozhraní `IPreferenceStore`. Metody `setValue()` a `setDefaultValue()` přidávají a aktualizují preference. Pro získání hodnoty preference lze zavolat jednu z metod: `getBoolean`, `getInt`, `getLong`, `getFloat`, `getDouble` anebo `getString`. Pro výchozí hodnoty existují podobné metody (např. `getDefaultDouble`).

Pro přístup ke preferencím souvisejícím s *SWT* objekty (*RGB*, *Rectangle*, *FontData*) lze zavolat metody třídy `PreferenceConverter`.

Během inicializace, třída `ScriptEditor` vytváří seznam `ArrayList` čtyř objektů `IPreferenceStore`:

- `ScopedPreferenceStore` vytvořený pro plug-in *Octave* (`com.zajcev.octoclipse`)
- `ScopedPreferenceStore` vytvořený pro plug-in textového editoru *Eclipse* (`org.eclipse.editors.text`)
- `EclipsePreferenceAdapter`, který poskytuje preference související s projektem obsahujícím upravovaná skripta
- `PreferenceAdapter`, který obsahuje preference asociované s jádrem *DLTK*

Tato skladiště jsou zahrnuta do `ChainedPreferenceStore`. Pokud dva nebo více z těch objektů obsahují stejné preference, hodnota se získá odtud, kam byla přidána jako první.

3.9.2 Inicializace preferencí

Pomocí rozšíření `org.eclipse.core.runtime.preferences` *Eclipse* umožňuje nastavovat preference během inicializace. *Octave IDE* rozšiřuje tento bod a identifikuje třídu `OctavePreferenceInitializer` jako dodavatele výchozích hodnot pro preference *IDE*, které rozšiřují abstraktní třídu `AbstractPreferenceInitializer`, čí jediná požadována metoda je `initializeDefaultPreferences()`. Prostředí vyvolává tuto metodu během inicializace. Ve výpisu B.7 (viz přílohy) je ukázáno jak je to naprogramováno.

Tato jednoduchá implementace zpřístupňuje úložiště uživatelských preferencí plug-inovi a volá třídu `OctavePreferenceConstants`, aby inicializovala hodnoty

skladu preferencí. Je to podtřída mateřské třídy *DLTK* — `PreferenceConstants`, která definuje a inicializuje širokou škálu obecných skriptovacích preferencí. Ve výpisu B.9 (viz přílohy) je ukázáno, jak se uskutečňuje inicializace zvýrazňování syntaxe.

První preference *Octave* se zabývá klíčovými slovy. Kód konfiguruje jak barvu tak typ písma (tučné, kurziva) klíčových slov ve vývojovém prostředí a vyžaduje dva kroky. Prvním krokem se zavolá `PreferenceConerter`, aby vytvořila asociaci `DLTKColorConstants.DLTK_KEYWORD` s modrou barvou. Dále, `store.setDefault` metoda spojuje klíčová slova s charakteristikou `.EDITOR_BOLD_SUFFIX`. Tyto řádky zajistí, že pokud nějaký token je inicializován s `DLTKColorConstants.DLTK_KEYWORD`, příslušný text se zobrazí modře a tučně.

3.9.3 Stránky preferencí a konfigurační bloky

Dále se probere, jak lze zpřístupnit uživatelovi nastavení preferencí. *Eclipse* pro tyto účely nabízí stránky preferencí (preference pages). *DLTK* zjednodušuje proces vytvoření těchto stránek pomocí objektů — konfiguračních bloků (configuration blocks). Tato podsekce vysvětlí obě dvě třídy a jak spolupracují, aby zajistily uživatelské preference v *Octave IDE*.

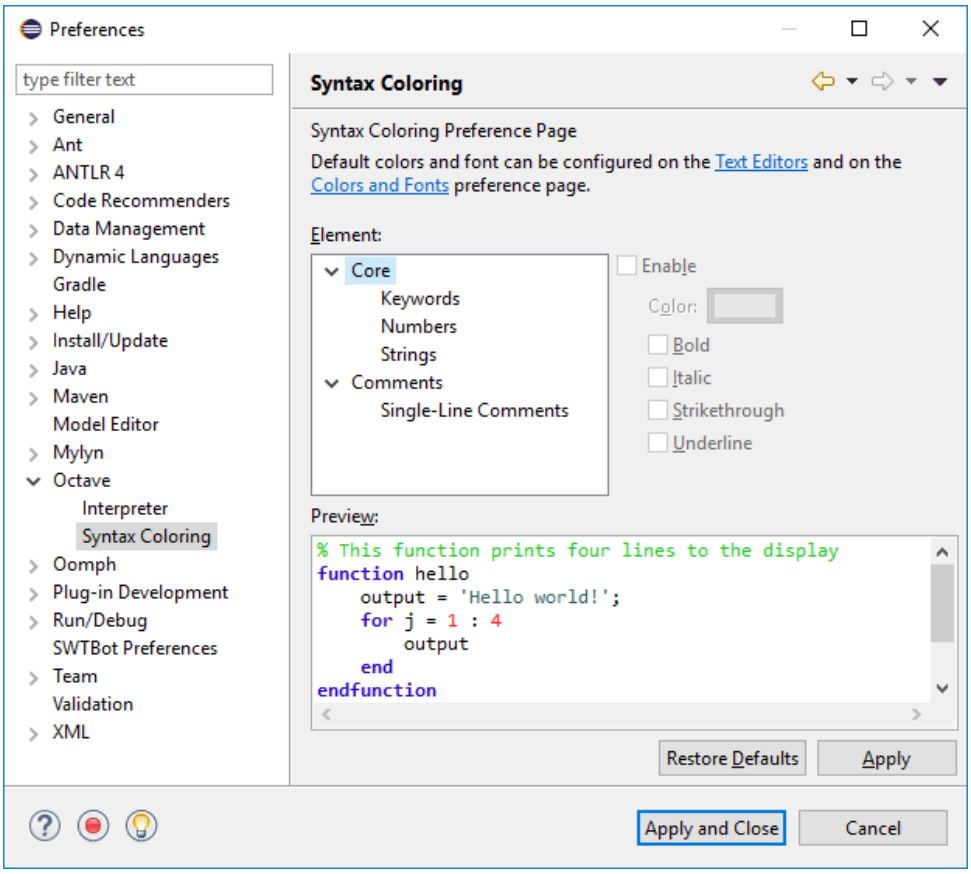
Stránky preferencí *Eclipse*

Když projdete cestou **`Window>Preferences`** v *Eclipse*, v levé části otevřeného okna uvidíte hierarchii opcí, jichž může nastavit uživatel. Pro *Octave IDE* budeme chtít preferenci vyšší úrovně *Octave* spolu s preferencemi o úroveň nižšími pro zvýrazňování syntaxe a interpret *Octave*. Na obrázku 3.2

Ve výpisu B.10 (viz přílohy) je uvedena definice rozšíření pro vytvoření stránek preferencí. Každé rozšíření vyžaduje id, jméno a atributy třídy. Dvě preference nižší úrovně mají jednu položku navíc — kategorie, která nazývá id rodičovské preference.

Atribut třídy nazývá třídu, která implementuje `IWorkbenchPreferencePage`, potomka `IPreferencePage`. *DLTK* nástroje poskytuje vlastní implementaci třídy `IWorkbenchPreferencePage` — `AbstractConfigurationBlockPreferencePage`. *DLTK* také poskytuje několik podtříd pro specifické účely. Hierarchie těchto tříd je zobrazena na obrázku 3.3.

Všechny tři třídy uvedené ve výpisu B.10 jsou přímými podtřídami (potomky) `AbstractConfigurationBlockPreferencePage`. Ve svých Java souborech nemají moc kódu. Každá třída poskytuje ID, označení (label) a přístup k úložišti preferencí plug-inu. Dále, každá třída `AbstractConfigurationBlockPreferencePage` musí



Obr. 3.2: Okno preferencí *Octave*.

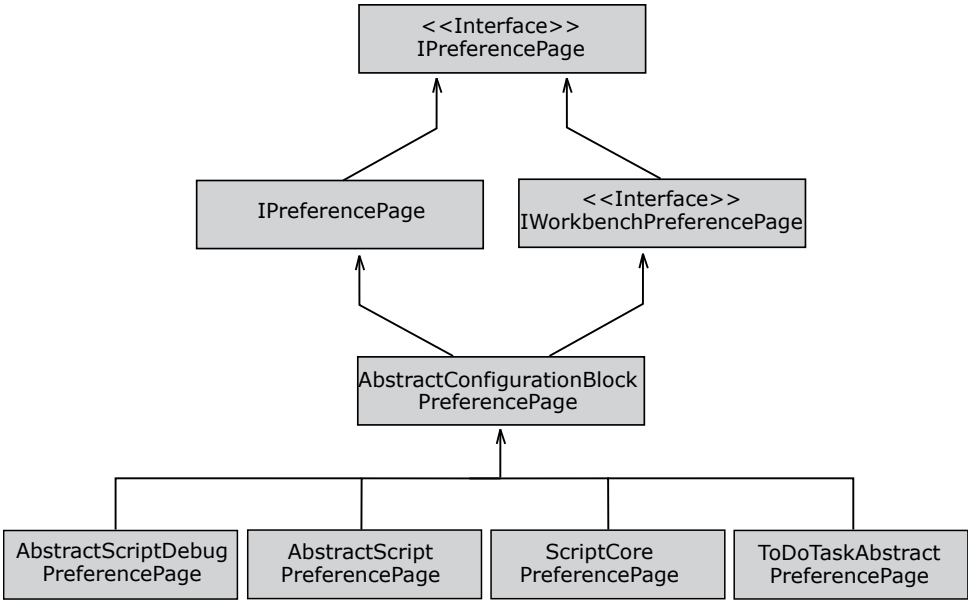
implementovat metodu `createConfigurationBlock()`, která vytváří objekt poskytující grafický aspekt stránky. Tento objekt musí implementovat rozhraní *DLTK* — `IPreferenceConfigurationBlock`. Toto důležité rozhraní se probere dále.

Konfigurační bloky *DLTK*

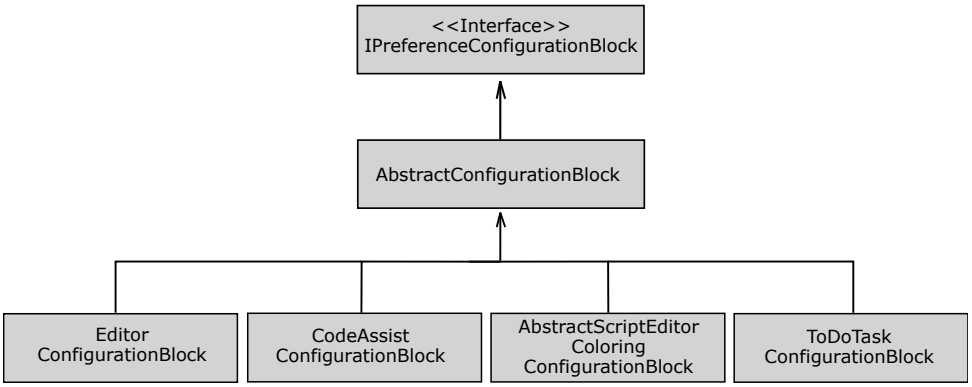
Když jde o reprezentaci preferencí *IDE*, nejsložitější částí je vytvoření grafického rozhraní. Například preference zvýrazňování syntaxe *Octave* nejen umožňuje uživatelovi volit barvy a typy písma, ale také poskytuje zabudovaný textový editor, který ukazuje jak stylizování skutečně vypadá. Vytvoření takových grafických ovladačů od nuly je časově náročné, ale *DLTK* třída `IPreferenceConfigurationBlocks` to zjednodušuje. Hierarchie těchto konfiguračních bloků je znázorněna na obrázku 3.4.

Konfigurační blok zvýrazňování syntaxe

První třída, která nás zajímá — `AbstractScriptEditorColorConfigurationBlock`, byla vytvořena speciálně aby poskytovala uživatelovi preference zvýrazňování syntaxe. Metoda `createSyntaxPage` provádí hlavní práci — vytváří grafickou stránku



Obr. 3.3: Hierarchie tříd stránek preferencí.



Obr. 3.4: Hierarchie tříd konfiguračních bloků.

preferencí. Podrobněji řečeno, formuje `TreeViewer`, který zobrazuje různé typy syntaxe a sadu tlačítek reprezentujících barvy a stylové preference. Dále, volá metodu `createPreviewer` pro vytvoření zabudovaného textového editoru.

Kód plug-inu *Octave* obsahuje `OctaveSyntaxColorConfigurationBlock` podtřídu, patřící k rodičovské `AbstractScriptEditorColoringConfigurationBlock` třídě. Kromě konstruktoru, jsou v ni také metody (viz tab. C.8 v přílohách).

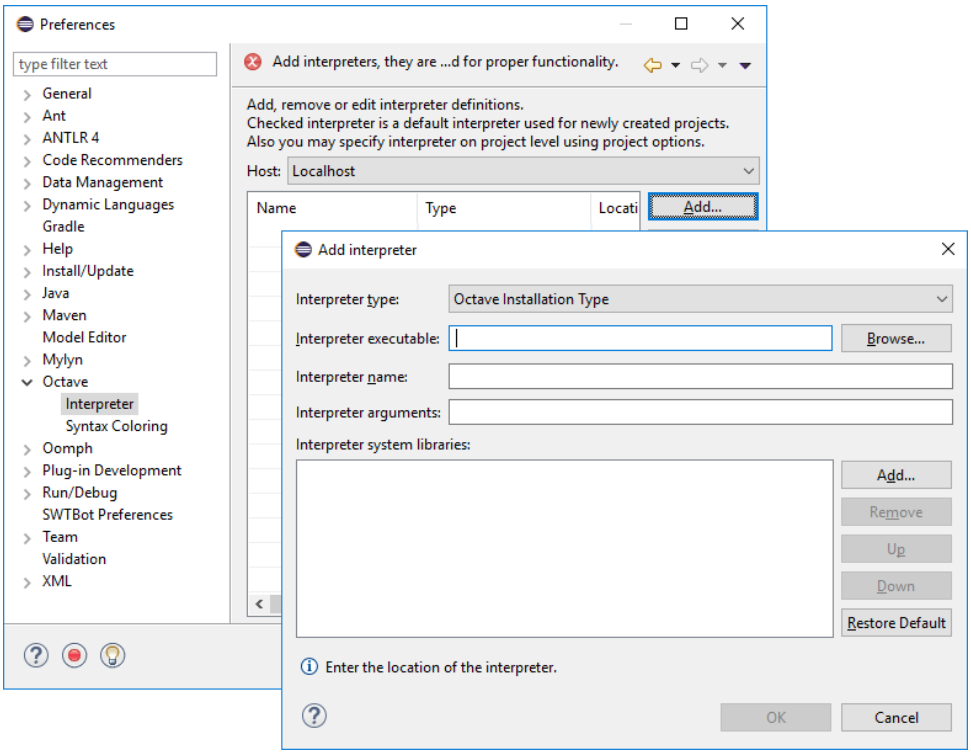
První tři metody jsou jasné, protože zabudovaný textový editor nepotřebuje žádný složitý prohlížeč, objekt konfigurace prohlížeče nebo účastníka pro nastavení dokumentu. Nicméně, poslední metoda, `getSyntaxColorListModel()`, vyžaduje vysvětlení. Tato metoda poskytuje dvojrozměrné pole čí elementy se shodují s typy syntaxe uvnitř prohlížeče stránky `TreeViewer`. Každý element si vyžaduje tři kusy informace: jméno, kategorie a shodující klíč uvnitř úložiště preferencí stránky.

Jak je vidět na obrázku 3.2, model zvýrazňování textu *Octave* rozlišuje docela čtyři typy: klíčová slova, čísla, řetězce a jednořádkové komentáře. První tři typy spadají do obecného záhlaví, když poslední typ patří k záhlaví komentáře. Ve výpisu B.11 je uvedeno, jak jsou ty tři typy konfigurovány.

Když uživatel změní nastavení preferencí a stiskne tlačítko „OK“, stránka zaktualizuje shodující hodnotu v úložišti preferencí. Editor odpovídá na změnu preferencí tím, že zaktualizuje reprezentaci textu.

3.9.4 Preference interpretu

Poslední preferenční opce zobrazena na obrázku 3.2 poskytuje nastavení pro skriptovací interpret *IDE. DLTK* nástroje poskytují dvě třídy, které to dělají možné: `ScriptInterpreterPreferencePage` a `InterpretersBlock`. Poslední třída vytváří grafické ovladače stránky, které se skládají z označení (Label) poskytující instrukce; `CheckboxTableViewer`, který udělá seznam dostupných interpretů; a sadu tlačítek, které umožňují uživatelovi přidávat, upravovat, kopírovat, odstraňovat nebo hledat interprety. Na obrázku 3.5 je ukázáno, jak tato stránka vypadá spolu s oknem, které vznikne, když uživatel zmáčkne tlačítko „Add“.



Obr. 3.5: Preferenční stránka a okno interpretu.

Abstraktní třída `InterpreterBlock` vyžaduje od podtříd implementaci dvou metod: `getCurrentNature` a `createInterpreterDialog`. První metoda vrací ře-

těžec, který reprezentuje interpret, jazyk zdrojového kódu a další spojené objekty. Tato příroda (nature) je často identifikována v rozšířeních *DLTK*, a je to důležité. Pokud máte instalovaný *Eclipse*, který obsahuje více projektů založených na *DLTK*, jste schopní je lehce rozlišovat podle jejich přírod.

Druhá metoda, `createInterpreterDialog`, vytváří okno, které vzniká, když uživatel stiskne tlačítko „Add“ na preferenční stránce. Toto okno (viz obr. 3.5) musí zařídit minimálně čtyři kusy informace: umístění spustitelného souboru interpretu, jeho název, typ a umístění jakýchkoli knihoven potřebných pro spouštění. K tomu, aby metoda `createInterpreterDialog` vytvořila toto okno, potřebuje příklad *DLTK* třídy `AddScriptInterpreterDialog`.

Konstruktor `AddScriptInterpreterDialog` přijímá pole typů interpretu a objekt implementující `IInterpreterInstall`. Přítomnost několika typů interpretu umožňuje *IDE* podporovat několik interpretů pro stejný jazyk. *Octave IDE* podporuje jenom jeden instalovány typ a je definován v souboru `plugin.xml` pomocí rozšíření `org.eclipse.dltk.launching.interpreterInstallTypes`. Jsou uvedeny ve výpisu B.12.

Atribut třídy identifikuje objekt, který implementuje `IInterpreterInstallType`. Každému instalovanému typu se přidělí jméno, ID, příroda (nature, řetězec, který identifikuje interpret a typ zdroje), umístění interpretu a požadovaných knihoven. Také, každý `IInterpreterInstallType` musí vytvořit jeden anebo více objektů `IInterpreterInstall`.

I když objekt `IInterpreterInstallType` definuje třídu instalací interpretu, tedy objekt `IInterpreterInstall` musí být vytvořen pro každou konkrétní instalaci. Tento objekt slouží jako primární objekt interpretu pro ukládaní dat, a drží informace z tabulky C.9.

Konečně, každý `IInterpreterInstall` musí zajistit přístup k objektu třídy `IInterpreterRunner`. Tento objekt zodpovídá za skutečné spouštění interpretu. Dále se probere nejen jak to funguje, ale také jak spolupracuje s *IDE* v celku.

3.10 Interpret a konzola *DLTK*

Nejdůležitější prvkem *DLTK* aplikaci není textový editor nebo preference, ale schopnost zahájit interpret kliknutím tlačítka a zobrazit jeho výstupy v konzole *IDE*. *DLTK* interpret a konzola jsou těsně propleteny, jejich řízení je složitým procesem. Tady je stručný přehled akcí, které probíhají, když uživatel spustí projekt:

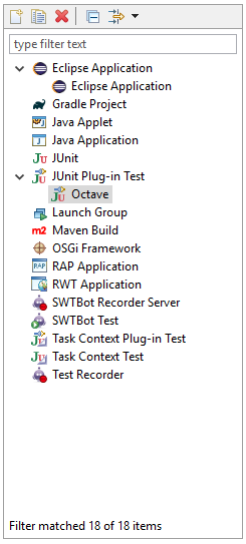
1. *IDE* odpovídá na stisk tlačítka vytvořením spouštějícího objektu, sbíráním informace o spouštění a přidáním procesu do spouštění.
2. Vytvoření spouštění oznamuje všech posluchače spuštění, včetně konzolového manažeru konzoly.

3. Manažer konzoly získává informace o spouštění a využívá ji pro vytvoření tří objekty: server konzoly, objekt interpretu a výrobce konzoly (console factory).
4. Server konzoly vytváří soket (výchozí port 25000) a čeká příchozí spojení.
5. Výrobce konzoly spouští interpret a vytváří konzolu.
6. Manažer konzoly dostává se k plug-inu konzoly *Eclipse* a přidává novou konzolu.

DLTK třídy obdržují velkou část toho zpracování, ale stále potřebujeme vytvořit mnoho sami. *Octave IDE* potřebuje podporovat spouštění spojených s *Octave*, dále, potřebuje třídy specifické pro *Octave* pro řízení konzoly a interpretu. Tato sekce probírá každý z těchto subjektů.

3.10.1 Konfigurace spouštění *DLTK*

V *Eclipse* proces spouštění aplikace se nazývá spouštění aplikace v „Run“ módu. Jelikož *Eclipse* podporuje mnoho spustitelných souborů, musíme zajistit, že interpret *Octave* se spustí, když uživatel spustí *Octave* skriptu. Toto propojení se realizuje pomocí konfiguračních typů. Jsou umístěny v záložce „**Run Configurations...**“ viz obrázek 3.6



Obr. 3.6: Konfigurační typy *Eclipse*

Pro vytvoření konfiguračního typu *Octave* prvním krokem je přidání rozšíření do souboru `plugin.xml` (viz výp. B.13 v přílohách).

Daný projekt by měl spouštět skripty *Octave*, ale bez tak zvaného „debuggingu“ (ladění). Proto atribut „modes“ je nastaven na „run“, ale ne „run,debug“. Nejdůležitějším atributem je „delegate“. Tento atribut identifikuje třídu implementující `ILaunchConfigurationDelegate`. Toto rozhraní definuje jednotlivou metodu,

spouštění, která se zavolá, když uživatel vytvoří konfiguraci pro spouštění a kliká tlačítko „Run“.

DLTK poskytuje vlastní implementaci `ILaunchConfigurationDelegate` jmenující se `AbstractScriptLaunchConfigurationDelegate`. Když se zavolá spouštěcí metoda této třídy, zavolají se metody uvedené v tabulce C.10 (viz přílohy).

První dvě metody jednoduše uspořádají informaci o spouštění. Postupně, objekt `InterpreterConfig` obdržuje informaci převzatou od konfigurace spouštění, a `IInterpreterInstall` ukládá informaci vzatou od preferencí. Je důležité rozumět, že třetí metoda nespouští interpret. Ona vytváří `IProcess` pro interpret a přidává jej do spouštějícího objektu.

3.10.2 Konzolový manažer *DLTK*

Když se spouští *DLTK* modul `org.eclipse.dltk.console.ui`, zpřístupňuje *Eclipse Debug UI* plug-in a přidává příklad `ScriptConsoleManager` jako posluchače spouštění. Tyto posluchače jsou uvědoměny kdykoli spouštění se přidává, mění nebo odstraňuje. `ScriptConsoleManager` jenom odpovídá na přidávaní spouštění a jeho prvním úkolem je ověřování, jestli spouštění má rozpoznatelnou přírodu skript. Pokud ano, manažer vytváří tři důležité objekty: server konzoly, objekt interpretu a výrobce konzoly.

Tento manažer vytváří `ScriptConsoleServer` pro řízení komunikace mezi interpretem a konzolou. Během její inicializace, svazuje `ServerSocket` s portem 25000 a říká ji, aby čekala na příchozí požadavek spojení. Tyto požadavky jsou zahrnuty do objektů `ConsoleRequest`, a jakmile si server dostane nový požadavek, zavolá metodu `consoleConnected`.

Až se vytvoří server, `ScriptConsoleManažer` zpřístupňuje `ConsoleRequest` reprezentující interpret. Proto ve souboru `plugin.xml` se hledá konzolové rozšíření `org.eclipse.dltk.console.scriptInterpreter` v souboru `plugin.xml`. Toto rozšíření identifikuje označení přírody a třídu, která implementuje `IScriptInterpreter` a výpomocné rozhraní `ConsoleRequest`. Potom jak získá server a objekt interpretu, manažer konzoly může začít vytvářet konzolu. Za tímto účelem volá výrobce konzoly.

3.10.3 Výrobce konzoly a konzola

Běžně, pokud chcete se dostat ke konzole *Eclipse* v nějakém plug-in projektu, potřebujete jenom jedno rozšíření — `org.eclipse.ui.console.consoleFactories`. Toto rozšíření musí identifikovat třídu, která implementuje `IConsoleFactory`. Pak, když uživatel stiskne „Open Console“ v konzolovém pohledu, *Eclipse* zavolá metodu `openConsole` pro vytvoření nové konzoly.

*DLTK*to má trochu složitější. `ScriptConsoleManager` potřebuje se dostat k výrobci konzoly, ale vyžaduje třídu, která by implementovala `IScriptConsoleFactory`. Prakticky obě dvě třídy, `IConsoleFactory` a `IScriptConsoleFactory`, vyžadují stejnou metodu `openConsole`. Nicméně, je potřeba vytvořit dvě rozšíření: jedno pro výrobce konzoly *Eclipse*, druhé pro výrobce konzoly *DLTK*. Jsou uvedeny ve vpisu B.14 (viz přílohy).

Když `ScriptConsoleManager` zpřístupňuje výrobce skriptovací konzoly, volá metodu `openConsole`, která vytváří všechny důležité objekty `ScriptConsole`. Stejně jako textový editor *Eclipse*, tato konzola zobrazuje text použitím metodiky založené na principu dokumentů; má `ConsoleDocument`, `IConsoleDocumentPartitioner` a pole pozic a regionů.

Když `ScriptConsole` je vytvořena, volá `setInterpreter` s odkazem na objekt `IScriptInterpreter`, který vytváří manažer konzoly. Tato metoda navazuje spojení mezi konzolou a interpretem ve formě objektu `InitialStreamReader`. Tento sledovač potoku se přiděluje k interpretovi voláním metody `getInitialOutputStream`. Dále, `InitialStreamReader` načítá každý řádek z výstupu interpretu a posílá jej do displeje konzoly. Když zobrazení textu je hotovo, konzola se přepne do režimu upravování.

Po vytvoření `ScriptConsole`, manažer konzoly zpřístupňuje `ConsoleManager` prostředí *Eclipse* a přidává nový objekt do seznamu dostupných konzol. Až teď nová konzola bude viditelná pro uživatele a přístupná pro interpret.

3.10.4 Spouštění interpretu

Teď se zblízka podíváme na objekt `IScriptInterpreter`, který se vytváří třídou `ScriptConsoleManager`. Toto rozhraní rozšiřuje rozhraní `ConsoleRequest`, což znamená že se může předat do `ScriptConsoleServer` pro zpracování. Manažer konzoly obdržuje toto doručení, pak když server si dostane požadavek interpretu, zavolá `consoleConnected(IScriptConsoleIO protocol)`.

Tato metoda má dva důležité úkoly: generuje `InitialStreamReader`, který umožňuje transfer dat mezi konzolou a interpretem, pak argument metody identifikuje protokol pro komunikaci. Rozhraní `IScriptConsoleIO` definuje dvě další důležité metody, které dostávají data od interpretu:

- **`InterpreterResponse execInterpreter(String command)`**. Posílá skriptovací příkaz interpretu a přijímá odpověď;
- **`ShellResponse execShell(String command, String[] args)`**. Posílá příkaz příkazovému interpretu a přijímá odpověď.

Tyto dvě metody směrují příkazy do interpretu a přijímají jeho výstup. Každý `IScriptConsole` protokol se inicializuje pomocí `InputStream` a `OutputStream`. Když

se zavolá `execInterpreter`, protokol pošle příkaz interpretu do jeho výstupního `OutputStream`. Když interpret odpovídá, protokol `InterpreterResponse` (podle výchozího nastavení — řetězec) a směruje ji do konzoly přes `InputStream`.

Všechna komunikace spoléhá na metody protokolu. Např., `IScriptInterpreter` rozhraní obsahuje `exec(String command)` která posílá příkazy do interpretu. Tento příkaz volá `IScriptConsoleIO.execInterpreter`, aby doručil příkaz. Stejně, metoda zavíraní konzoly spoléhá na `IScriptConsoleIO.execShell`.

4 ZÁVĚR

Cílem bakalářské práce bylo seznámení s vnitřní architekturou vývojového prostředí *Eclipse* a návrh zásuvného modulu pro ně, který by podporoval skriptovací prostředí *Octave*.

Pro řešení první části zadání bylo nezbytným se blíže seznámit s vývojovým prostředím *Eclipse*. Dále bylo nutné se seznámit s nástrojem *Dynamic Languages Toolkit (DLTK)*, který umožňuje práci se skriptovacími jazyky.

Ukázalo se, že nástroje *DLTK* poskytují mnoho funkcí k vytvoření vývojových prostředí pro skriptovací jazyky. Návrh takového prostředí nevyžaduje moc kódování, ale je nutné rozumět jednotlivým funkcím daných nástrojů.

Byla dosažena komunikace mezí konzolou vývojového prostředí *Eclipse* a interpretem *Octave*. Pomocí konzoly navržený modul dokáže posílat příkazy do prostředí *Octave* a dostávat výsledky, ale jen pro jednodušší případy. Např., při pokusu nakreslit graf se komunikace přeruší a bude potřeba spustit novou konzolu. Předpokládanou příčinou této chyby je malý buffer *Eclipse*, do kterého se předávají výstupní data prostředí *Octave*.

Navržený zásuvný modul umožňuje uživatelovi nastavit parametry zvýrazňování syntaxe (barva, typ písma) a zvolit interpret pro skripty napsáné v jazyce *Octave* pomocí preferenčních stránek. Také je možné vytvořit nový *Octave* projekt, naplnit ho soubory s příponou `*.m` a upravovat kód *Octave* přímo uvnitř *IDE Eclipse* pomocí textového editoru *Octave*.

LITERATURA

[1] Platform Plug-in Developer Guide [online]: *Průvodce vývojem zásuvných modulů.* Dostupné z URL: <http://help.eclipse.org/kepler/index.jsp?nav=%2F2.

[2] Help - Eclipse Platform [online]: *Manuální stránky Eclipse.* Dostupné z URL: <https://help.eclipse.org/neon/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Fguide%2Farch.htm>.

[3] Build an Eclipse development environment for Perl, Python, and PHP. [online]. Copyright [cit. 14.12.2017]. Dostupné z URL: <https://www.ibm.com/developerworks/opensource/tutorials/os-eclipse-octave/index.html

[4] The Eclipse Foundation. [online]: *Aktualizační zdroj Dynamic Languages Toolkit (DLTK) pro IDE Eclipse.* Dostupné z URL: <http://download.eclipse.org/technology/dltk/updates-dev/latest/>.

[5] The Eclipse Foundation. [online]: *Aktualizační zdroj Plug-in Development (PDE) pro IDE Eclipse.* Dostupné z URL: <http://download.eclipse.org/eclipse/updates/4.7>.

[6] About Us – OSGi™ Alliance. OSGi™ Alliance – The Dynamic Module System for Java [online]. Copyright © 2018 [cit. 23.05.2018]. Dostupné z URL: <https://www.osgi.org/about-us/>.

[7] Equinox | The Eclipse Foundation. Open Innovation Community - Eclipse IDE | The Eclipse Foundation [online]. Copyright © Eclipse Foundation, Inc. All Rights Reserved. [cit. 23.05.2018]. Dostupné z URL: <http://www.eclipse.org/equinox/>.

SEZNAM SYMBOLŮ, VELIČIN A ZKRATEK

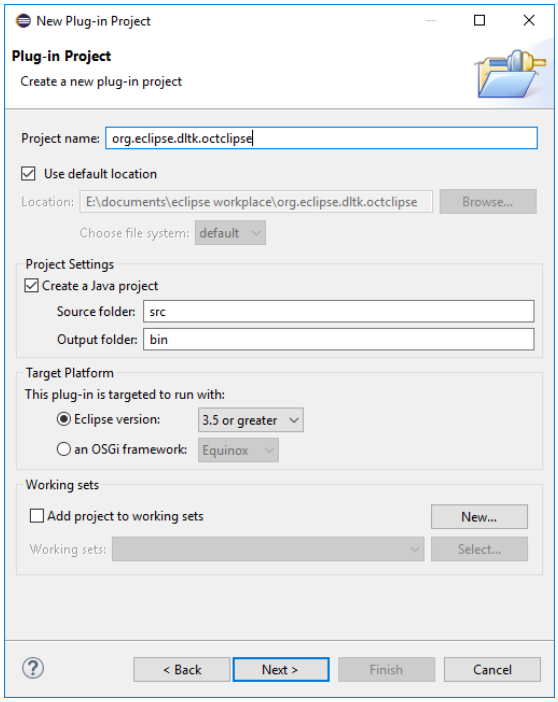
DLTK	nástroje pro práci s dynamickými jazyky — *Dynamic Languages Toolkit*
GNU	svobodný operační systém projektu *GNU* (ne *UNIX*)
IDE	integrované vývojové prostředí — *Integrated Development Environment*
API	rozhraní pro programování aplikací — *Application Programming Interface*
JDT	nástroje pro vývoj *Java IDE* — *Java Development Tools*
PDE	nástroje pro vývoj plug-inů *Eclipse* — *Plug-in Development Environment*
SDK	sada nástrojů pro vývoj aplikací — *Software Development Kit*
JVM	sada programů pro spouštění dalších programů — *Java Virtual Machine*
RCP	„open-source“ projekt pro tvorbu flexibilních a rozšířitelných „stand-alone“ aplikací — *Rich Client Platform*
RAP	„open-source“ projekt pro tvorbu aplikací založených na technologii Ajax — *Rich Ajax Platform*
XML	obecný značkovací jazyk — *eXtensible Markup Language*
Git	distribuovaný systém správy verzí
SWT	knihovna grafických uživatelských prvků pro platformu Java — *Standard Widget Toolkit*
GUI	uživatelské rozhraní, které umožňuje ovládat počítač pomocí interaktivních grafických ovládacích prvků — *Graphical User Interface*
JAR	kompresní souborový formát, používaný platformou Java — *Java ARchive*
OSGi	specifikace dynamického modulárního systému pro programovací jazyk Java — *Open Services Gateway initiative*
URL	řetězec znaků sloužící k přesné specifikaci umístění zdrojů informací — *Uniform Resource Locator*

SEZNAM PŘÍLOH

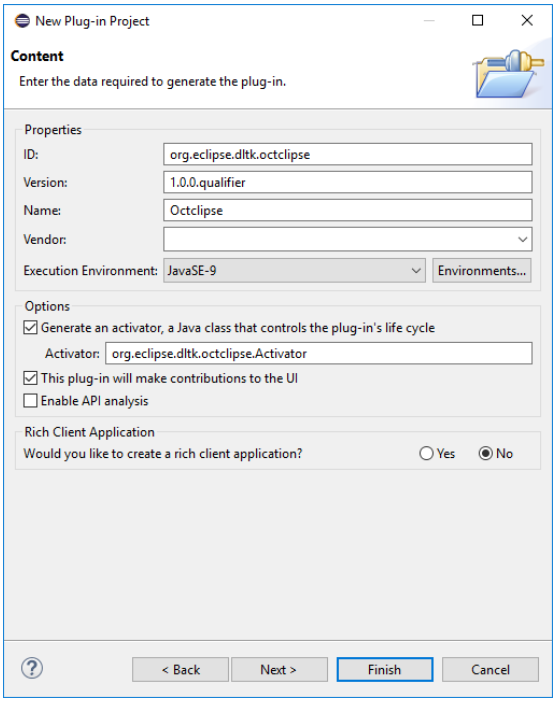
A POČÁTEČNÍ POSTUP VYTVOŘENÍ NOVÉHO PROJEKTU ZÁSUVNÉHO MODULU

Prvním krokem návrhu zásuvného modulu musíme projít průvodcem nového projektu, který navíc má několik opce generování kódu, jakými jsou pohledy, editory a akce pro vzorový kód plug-inu.

Projdete cestou `File > New > Project` — z nabídky **Plug-in Development** zvolte možnost **Plug-in Project**. Následovným stisknutím tlačítka **Next** se začne vytvoření projektu zásuvného modulu.

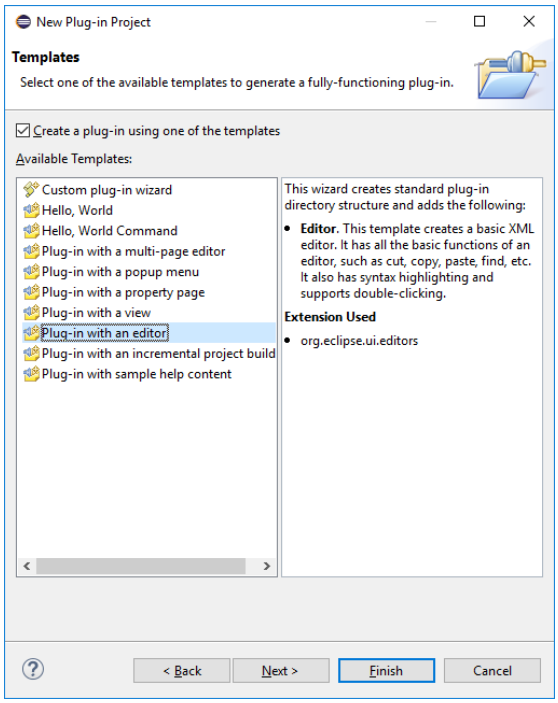


Obr. A.1: Zadání jména projektu

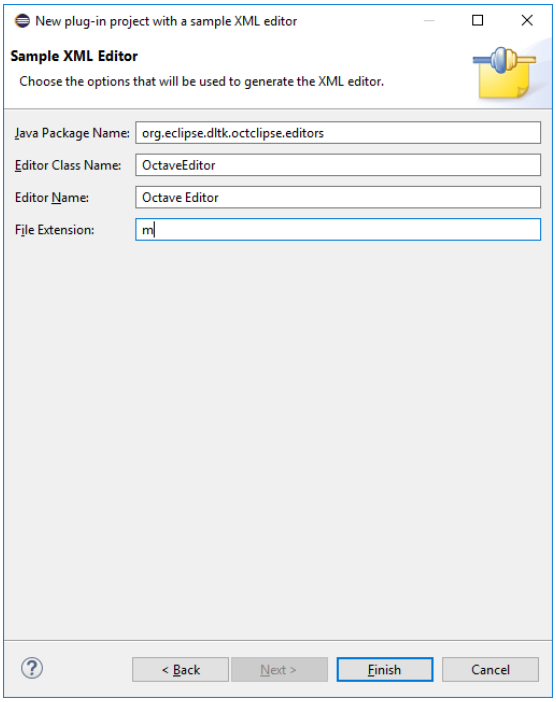


Obr. A.2: Okno obsahu

V prvním okně A.1 zadejte jméno projektu, cestu jeho umístění, adresáře zdrojových a výstupních souborů a verzi *Eclipse*, na které by plugin měl fungovat. Popřípadě můžete přidat projekt do nějakého pracovního setu. V následujícím okně A.2 zadejte další vlastnosti projektu.



Obr. A.3: Šablony pro vytvoření nového modulu



Obr. A.4: Zadání informace pro editor

V okně A.3 můžete si vybrat jednu z šablon, podle které bude založen nový projekt, anebo vytvořit prázdný projekt zrušením možnosti **Create a plug-in using one of the templates**. Šablona s editorem usnadní práci, protože takto budete mít předchystaný editor už na začátku. Pokud jste si zvolili tuto šablonu, v dalším okně A.4 doplňte informaci o Vašem editoru podle potřeb.

B UKÁZKY KÓDŮ ZÁSUVNÉHO MODULU *OCTO-CLIPSE*

Výpis B.1: Definice typu souborů podporovaným editorem.

```
<extension point="org.eclipse.core.runtime.contentTypes">
  <content-type
    base-type="org.eclipse.core.runtime.text"
    describer="com.zajcev.octoclipse.core.
       OctaveContentDescriber"
    file-extensions="m"
    id="com.zajcev.octoclipse.content-type"
    name="OctaveContentType"
    priority="high">
  </content-type>
</extension>
```

Výpis B.2: DLTK Language.

```
<extension point="org.eclipse.dltk.core.language">
  <language
    class="com.zajcev.octoclipse.core.
       OctaveLanguageToolkit"
    nature="com.zajcev.octoclipse.nature"
    priority="0">
  </language>
</extension>
```

Výpis B.3: Konfigurace dokumentu

```
<extension point="org.eclipse.core.filebuffers.
   documentSetup">
  id="com.zajcev.octoclipse.editor.OctaveDocumentSetup"
  name="%documentSetupName"
  <participant
    extensions="m"
    class="com.zajcev.octoclipse.editor.
       OctaveDocumentSetup">
  </participant>
</extension>
```

Výpis B.4: Pravidla pro vytvoření tokenů.

```
IToken string = new Token(IOctavePartitions.OCTAVE_STRING
   );
IToken comment = new Token(IOctavePartitions.
   OCTAVE_COMMENT);

// Create the list of rules that produce tokens
List<IPredicateRule> rules = new ArrayList<IPredicateRule
   >();
rules.add(new SingleLineRule("%", "\n", comment));
rules.add(new SingleLineRule("'", "'", string));
IPredicateRule[] result = new IPredicateRule[rules.size()
   ];
rules.toArray(result);
setPredicateRules(result);
```

Výpis B.5: Configurace synchronizátoru (*presentation reconciler*).

```
// Vytvoří DefaultDamagerRepairer pro samotný kód
DefaultDamagerRepairer dr = new DefaultDamagerRepairer(
   codeScanner);
reconciler.setDamager(dr, IDocument.DEFAULT_CONTENT_TYPE)
   ;
reconciler.setRepairer(dr, IDocument.DEFAULT_CONTENT_TYPE
   );

// Vytvoří DefaultDamagerRepairer pro řetězce
dr = new DefaultDamagerRepairer(stringScanner);
reconciler.setDamager(dr, OctavePartitions.OCTAVE_STRING)
   ;
reconciler.setRepairer(dr, OctavePartitions.OCTAVE_STRING
   );

// Vytvoří DefaultDamagerRepairer pro komentáře
dr = new DefaultDamagerRepairer(commentScanner);
reconciler.setDamager(dr, OctavePartitions.OCTAVE_COMMENT
   );
reconciler.setRepairer(dr, OctavePartitions.
   OCTAVE_COMMENT);
```

Výpis B.6: Vytvoření skenerů skript v třídě `OctaveSourceViewerConfiguration`

```
1  private AbstractScriptScanner codeScanner, stringScanner,
       commentScanner;
2
3  protected void initializeScanners() {
4
5  // Create a scanner for the script code
6  codeScanner = new OctaveCodeScanner(getColorManager(),
       fPreferenceStore);
7
8  // Create a scanner for string partitions
9  stringScanner = new SingleTokenScriptScanner(
       getColorManager(),
10 fPreferenceStore, IOctavePartitions.OCTAVE_STRING);
11
12 // Create a scanner for comment partitions
13 commentScanner = new SingleTokenScriptScanner(
       getColorManager(),
14 fPreferenceStore, IOctavePartitions.OCTAVE_COMMENT);
15 }
```

Výpis B.7: Inicializace výchozích preferencí

```
1  public void initializeDefaultPreferences() {
2  IPreferenceStore store = OctavePlugin.getDefault().
       getPreferenceStore();
3  OctavePreferenceConstants.initializeDefaultValues(store);
4  }
```

Výpis B.8: Vytvoření skenerů skript v třídě `OctaveCodeScanner`

```
1  protected List<IRule> createRules() {
2
3  // Vytváří tokeny
4  IToken keywordToken = getToken(DLTKColorConstants.
       DLTK_KEYWORD);
5  IToken numberToken = getToken(DLTKColorConstants.
       DLTK_NUMBER);
6  IToken defaultToken = getToken(DLTKColorConstants.
       DLTK_DEFAULT);
7
8  // Vytváří naplňuje seznam
9  List<IRule> ruleList = new ArrayList<IRule>();
10
11 // Vytváří pravidlo pro detekci klíčových slov
12 WordRule wordRule = new WordRule(new OctaveWordDetector
       (), defaultToken);
13 for (int i = 0; i < IOctaveKeywords.keywords.length; i
       ++)
14 wordRule.addWord(IOctaveKeywords.keywords[i],
       keywordToken);
15 ruleList.add(wordRule);
16
17 // Vytváří pravidlo pro detekci čísel
18 NumberRule numberRule = new NumberRule(numberToken);
19 ruleList.add(numberRule);
20
21 // Nastaví vracený token pro výchozí text
22 setDefaultReturnToken(defaultToken);
23 return ruleList;
24 }
```

Výpis B.9: Nastavení preferencí zvýrazňování syntaxe

```
1  public static void initializeDefaultValues(
       IPreferenceStore store) {
2
3  // Nastaví výchozí preference pro editor
4  PreferenceConstants.initializeDefaultValues(store);
5
6  // Zvýrazní klíčová slova modře a tučně
7  PreferenceConverter.setDefault(store,
       DLTKColorConstants.DLTK_KEYWORD,
8  new RGB(40, 0, 200));
9  store.setDefault(DLTKColorConstants.DLTK_KEYWORD +
10 PreferenceConstants.EDITOR_BOLD_SUFFIX, true);
11
12 // Nastaví výchozí hodnoty pro ostatní preference
13 PreferenceConverter.setDefault(store,
14 DLTKColorConstants.DLTK_SINGLE_LINE_COMMENT, new RGB
       (25, 200, 25));
15 PreferenceConverter.setDefault(store,
       DLTKColorConstants.DLTK_NUMBER,
16 new RGB(255, 25, 25));
17 PreferenceConverter.setDefault(store,
       DLTKColorConstants.DLTK_STRING,
18 new RGB(50, 100, 100));
19 }
```

Výpis B.10: Rozšíření *Octave* pro stránky preferencí

```
<extension point="org.eclipse.ui.preferencePages">
  <page
    class="com.zajcev.octoclipse.preferences.
      OctaveMainPreferencePage"
    id="com.zajcev.octoclipse.preferences.
      OctaveMainPreferencePage"
    name="%MainPreferencePage.name"/>
  <page
    category="com.zajcev.octoclipse.preferences.
      OctaveMainPreferencePage"
    class="com.zajcev.octoclipse.preferences.
      OctaveSyntaxColorPage"
    id="com.zajcev.octoclipse.preferences.
      OctaveSyntaxColorPage"
    name="%SyntaxColorPreferencePage.name"/>
  <page
    category="com.zajcev.octoclipse.preferences.
      OctaveMainPreferencePage"
    class="com.zajcev.octoclipse.preferences.
      OctaveInterpreterPreferencePage"
    id="com.zajcev.octoclipse.preferences.
      OctaveInterpreterPreferencePage"
    name="%InterpreterPreferencePage.name"/>
</extension>
```

Výpis B.11: Vytvoření modelu seznamu zvýrazňovní syntaxe

```
protected String[][] getSyntaxColorListModel() {
return new String[][] {
{ "Single-Line Comments", DLTKColorConstants.
   DLTK_SINGLE_LINE_COMMENT, sCommentsCategory},
{ "Keywords", DLTKColorConstants.DLTK_KEYWORD,
   sCoreCategory },
{ "Strings", DLTKColorConstants.DLTK_STRING,
   sCoreCategory },
{ "Numbers", DLTKColorConstants.DLTK_NUMBER,
   sCoreCategory }};
}
```

Výpis B.12: Instalované typy interpretu

```
1 <extension point="org.eclipse.dltk.launching.
    interpreterInstallTypes">
2   <interpreterInstallType
3     class="com.zajcev.octoclipse.launch.
        OctaveInterpreterInstallType"
4     id="com.zajcev.octoclipse.launch.
        OctaveInterpreterInstallType">
5   </interpreterInstallType>
6 </extension>
```

Výpis B.13: Rozšíření konfiguračního typu *Octave*

```
1 <extension point="org.eclipse.debug.core.
    launchConfigurationTypes">
2   <launchConfigurationType
3     id="com.zajcev.octoclipse.launch.
        OctaveLaunchConfigurationType"
4     delegate="com.zajcev.octoclipse.launch.
        OctaveLaunchConfigurationDelegate"
5     modes="run"
6     public="true"
7     name="Octave␣Launch␣Configuration"
8   </launchConfigurationType>
9 </extension>
```

Výpis B.14: Identifikace výrobce konzoly *Octave*

```
1  <extension point="org.eclipse.dltk.console.ui.
       scriptConsole">
2    <scriptConsole
3      class="org.dworks.octaveide.launch.
           OctaveConsoleFactory"
4      natureID="org.dworks.octaveide.nature" />
5    </extension>
6
7  <extension point="org.eclipse.ui.console.consoleFactories
       ">
8    <consoleFactory
9      class="org.dworks.octaveide.launch.
           OctaveConsoleFactory"
10     icon="icons/oct.gif"
11     label="%OctaveConsole.Console" />
12 </extension>
```

C TŘÍDY ZÁSUVNÉHO MODULU, OBJEKTY, METODY A JEJICH POPIS

Tab. C.1: Závislosti zásuvného modulu *Octo-clipse*

Název balíku	**Popis balíku**
`org.eclipse.ui`	Rozhraní aplikace pro interakci s *Eclipse* platformou
`org.eclipse.core.runtime`	Základ běhu platformy
`org.eclipse.ui.console`	Rozhraní aplikace pro interakci s *Eclipse* konzolou
`org.eclipse.dltk.core`	Vývojové prostředí zaměřené na dynamické jazyky
`org.eclipse.core.resources`	Poskytuje základní podporu pro řízení pracovního prostoru a jeho zdrojů
`org.eclipse.ui.editors`	Poskytuje standardní textový editor
`org.eclipse.ui.ide`	*API* pro části uživatelského rozhraní *Eclipse* platformy charakteristických pro *IDE*
`org.eclipse.dltk.ui`	Rozhraní aplikace pro interakci s nástrojem *DLTK*

Tab. C.2: Metody rozhraní `IDLTKLanguageToolkit`.

Název metody	**Popis**
`getEditorId(Object elementID)`	Vrací ID editoru
`getTextTools()`	Vrací `ScriptTextTools` objekt, který určí, jak text bude reprezentovan v editoru
`createSourceViewerConfiguration()`	Vrací `ScriptSourceViewerConfiguration` objekt, který nastaví operaci prohlížeče zdroje editoru
`getPreferenceStore()`	Vrací `IPreferenceStore` objekt, který obsahuje nastavení editoru
`getEditorPreferencePages()`	Vrací ID stránek nastavení spojených s nastavením editoru
`getInterpreterPreferencePage()`	Vrací ID stránek nastavení spojených se skriptovým interpretem
`getIntepreterId()`	Vrací ID skriptového interpretu

Tab. C.3: Objekty podtřídy `OctaveTextTools`.

Objekt	**Popis**
`OctavePartitionScanner`	Načítá text editoru a určuje hranice rozdělení řádků
`OctaveSourceViewerConfiguration`	Nastavuje `ScriptSourceViewer` a způsob, jakým ten reaguje na události uživatele

Tab. C.4: Segmenty do kterých *Octave* plug-in dělí text.

Segment	**Popis**
Komentáře (comments)	Identifikuje se pomocí `OCTAVE_COMMENT`, co je stejné jak „`__octave_comment__`“
Řetězce (strings)	Identifikuje se pomocí `OCTAVE_STRING`, co je stejné jak „`__octave_string__`“
Kód	Identifikuje se pomocí `IDocument.DEFAULT_CONTENT_TYPE`, co je stejné jak „`__dftl_partition_content_type`“

Tab. C.5: Pomocné metody pro zvýrazňování syntaxe.

Název metody	**Popis**
`getPresentationReconciler()`	Vrací `IPresentationReconciler`, který kontroluje reprezentaci textu při změnách
`initializeScanners()`	Vrací pole `AbstractScriptScanners`, které provádí stejnou analýzu textu, jak `RuleBasedPartitionScanners` popsaný dříve

Tab. C.6: Objekty třídy `IPresentationRekonciler`

Název objektu	**Popis**
`IPresentationDamager`	Určuje rozsah změn dokumentu
`IPresentationRepairer`	Vytváří objekt `TextPresentation` pro odpověď na modifikaci, kterou identifikoval `IPresentationDamager`

Tab. C.7: Implementace `IRule` používané skenerem `OctaveCodeScanner`

Název objektu	**Popis**
`WordRule`	Vrací token, když se narazí na určitá slova
`NumberRule`	Vrací token, když se narazí na čísla

Tab. C.8: Metody třídy `OctaveSyntaxColorConfigurationBlock`

Název metody	**Popis**
`createPreviewViewer`	Vrací prohlížeč pro obalení kolem zabudovaného textového editoru
`createSimpleSourceViewerConfiguration`	Vrací základní objekt pro konfiguraci prohlížeče uvnitř textového editoru
`setDocumentPartitioning`	Volá `IDocumentSetupParticipant` aby nastavil rozdělení dokumentu
`getSyntaxColorListModel`	Vrací dvojrozměrné pole, které identifikuje různé typy syntaxe, které mohou být zvýrazněny barevně, a také shodující klíč úložiště preferencí

Tab. C.9: Data ukládané objektem `IInterpreterInstall`

Pole	**Popis**
`Jméno (name)`	Označení (label) pro inicializaci interpretu
`ID`	Logický identifikátor pro instalaci interpretu
`Příroda (nature)`	Logický identifikátor pro *IDE* (interpret, zdrojové soubory atd.)
`Umístění pro instalaci (installation location)`	Cesta instalaci interpretu
`Umístění knihoven (library location)`	Cesty k potřebným knihovnám
`Prostředí pro spouštění (executional environment)`	Proměnné prostředí a konfigurace procesu
`Argumenty interpretu (interpreter arguments)`	Parametry adresované interpretu

Tab. C.10: Metody volány během spouštění

Metoda	**Popis**
`createInterpreterConfig`	Vytváří objekt `InterpreterConfig`, který získává všechny informace od spouštějící konfigurace.
`getInterpreterRunner`	Přistupuje k objektu vytvořenému instalačním typem interpretu a využívá jej pro formování `IInterpreterRunner`.
`runRunner`	Začíná proces interpretu a přidává jej do spouštění.

D OBSAH PŘILOŽENÉHO DVD

Přiložené DVD obsahuje elektronickou verzi bakalářské práce (ve formátu PDF) a zdrojový soubor s kódem zásuvného modulu. Hlavní dokument elektronické verze bakalářské práce „xzayts00.pdf“ se nachází v adresáři „dokumentace“. Zdrojové soubory jsou zabaleny do archivu „octoclipse.zip“ a umístěny v adresáři projekt. Zdrojové soubory lze po rozbalení importovat do vývojového prostředí Eclipse jako existující projekt (Existing Projects into Workspace). Zbylý soubor `README.txt` obsahuje dodatečné informace k instalaci zásuvného modulu.

- `/` ..... kořenový adresář DVD
  - `dokumentace`
    - `"xzayts00.pdf"` ..... elektronická verze bakalářské práce
  - `projekt`
    - `"octoclipse.zip"` ..... zdrojové soubory zásuvného modulu
    - `"octoclipse.1.0.0.jar"` ..... JAR soubor zásuvného modulu
  - `README.txt` ..... návod na instalaci do *Eclipse IDE*