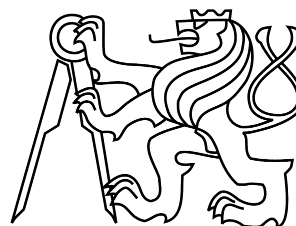
České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra softwarového inženýrství



Diplomová práce

Editace abstraktního syntaktického stromu

Bc. Matej Pankovčin

Vedoucí práce: Ing. Jiří Daněček

04.05.2015

Poděkování

Rád bych na tomto místě poděkoval Ing. Jiřímu Daněčkovi za inspiraci, odborné vedení, vstřícný přístup a cenné rady v průběhu vzniku této práce. Taky bych rád poděkoval Janu Bergerovi, Martinu Taichmannovi, Davidu Stránskému a Miloši Haleckému za pomoc při testování.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona.

V Praze dne 4.5.2015 ....

Abstract

This thesis has come to existence to examine the possibility of novel approach of developing code. Its main objective is to present a concept and deliver a tool for building abstract syntax tree using graphical interface. In the research section of this work, the nature and the purpose of the abstract syntax tree for the language compilation is explained. The research section also contains exploration of frameworks allowing the manipulation of abtract syntax trees. In implementation section, the proposed solution is presented and tested.

Abstrakt

Tato práce vznikla pro prozkoumání možnosti nového přístupu k vývoji kódu. Jejím hlavním cílem je představit koncept a dodat nástroj pro tvorbu abstraktních syntaktických stromů s použitím grafického rozhraní. V rešeršní části této práce je vysvětlena podstata a smysl abstraktních syntaktických stromů při kompilaci jazyka. Rešeršní část také obsahuje probádání frameworků umožňující manipulaci s abstraktním syntaktickým stromem. V implementační části je navržené řešení prezentováno a testováno.

Obsah

1 Úvod......**15**
- 1.1 Motivace......15
- 1.2 Cíl práce......16
- 1.3 Struktura práce......16

2 Abstraktní syntaktický strom......**17**
- 2.1 Java......17
- 2.2 Kompilace......17
- 2.3 Front-end......18
- 2.4 Syntaktický strom......19
- 2.5 Konkrétní a abstraktní syntaktický strom......20
- 2.6 Abstraktní syntaktický strom......21

3 Nástroje pro manipulaci s AST......**25**
- 3.1 JRefactory......25
- 3.2 CodeModel......26
- 3.3 JavaParser......26
- 3.4 Eclipse JDT......27
- 3.5 Diskuse......28

4 Java FX......**29**

5 Eclipse JDT......**31**
- 5.1 Java Model......31
- 5.2 JDT API......33

6 Syntaxe Javy v Eclipse JDT......**35**

7 Návrh řešení......**39**
- 7.1 Uzel......40
- 7.2 Spoje......40
- 7.3 Navržené operace......41
  - 7.3.1 Tvorba uzlu......41
  - 7.3.2 Mazání uzlu......42
  - 7.3.3 Editace uzlu......42
  - 7.3.4 Spojování uzlů a tvorba stromů......42
  - 7.3.5 Kopírování stromů a podstromů......43
  - 7.3.6 Vizuální reprezentace stromu......43
  - 7.3.7 Spuštění stromu......44
  - 7.3.8 Persistence......46

8 Implementace......**47**
- 8.1 Základní struktura......47
- 8.2 View......48
  - 8.2.1 Třída ScrollView......49
  - 8.2.2 Basic......49
  - 8.2.3 Rozhraní Jointable a třídy Joint......50
  - 8.2.4 Hub, ParentHub a ChildrenHub......51

8.2.5 NodeRepresentation.....51
8.2.6 Dialogy.....52
8.3 Controller.....53
8.3.1 Překlad do zdrojového kódu.....53
8.3.2 Inspekce stromu.....53
8.3.3 Budování stromu.....55
8.3.4 NodeService.....57
8.3.5 Překlad ze zdrojového kódu.....58
8.3.6 Třídy TranslationTable, TranslationTuple a TranslationRule.....59
8.3.7 Třída ConnectionCommand.....60
8.4 Model.....60
8.4.1 JointRules.....60
8.4.2 Rozhraní MNode.....61
8.4.3 Rozhraní MEditable.....62
8.4.4 Abstraktní třída MAbstractNode.....62
8.4.5 Struktura rozhraní v balíčku nodeTypes.....63
8.4.6 Statement.....63
8.4.7 Třída MIf.....64
8.4.8 Třída MBlock.....66
8.4.9 Expression.....67
8.4.10 Třída MLiteral.....67
8.4.11 Abstraktní třída MInfixExpression a její potomci.....68
8.4.12 Třída MMethodInvocation.....70
8.4.13 Třída MVariableDeclaration.....71
8.4.14 ExpressionStatement.....72
8.4.15 Rozhraní MAbstractType.....75
8.4.16 Rozhraní MBodyDeclaration.....76
8.4.17 Třída MParameter.....76
9 Testování.....79
9.1 Test.....79
9.2 Výsledky.....81
9.3 Diskuse.....82
9.4 Spuštění vytvořeného programu.....82
10 Závěr.....85
A Uživatelské informace.....91
B Obsah přiloženého CD.....95
C Uživatelská příručka.....96
C.1 Úvod.....96
C.2 Uzel.....96
C.3 Spoje.....97
C.4 Práce s uzlem.....98
C.4.1 Vytvoření nového uzlu.....98
C.4.2 Kontextová nabídka.....99
C.4.3 Mazání a zaměření.....100
C.4.4 Sbalení uzlu.....101

C.5 Práce s plochou....................101
C.6 Menu....................101
C.6.1 Výběr stromu pro operaci....................102
C.6.2 Položka menu Run....................103
C.6.3 Položka menu File....................106
C.6.4 Položka menu Tree....................107
C.6.5 Položka menu Language....................108
C.6.6 Nápověda....................109
C.7 Gesta....................109

Seznam obrázků

Obr 2.2.1: Kompilace a interpretace jazyka Java[1]....................18
Obr 2.4.1: Příklad kořenového stromu[2]....................19
Obr 2.5.1: Konkrétní syntaktický strom[3]....................20
Obr 2.5.2: Abstraktní syntaktický strom[3]....................21
Obr 2.6.1: Abstraktní syntaktický strom euklidova algoritmu[4]....................22
Obr 4.1: Technologie Java FX[10]....................29
Obr 5.1.1: Struktura projektu v Java Model [12]....................32
Obr 5.1.2: Struktura třídy v Java Model [12]....................32
Obr 7.1: Okno editoru a vytvořený strom....................39
Obr 7.1.1: Uzel připojen k rodičovi a potomkům....................40
Obr 7.3.1.1: Tvorba nového uzlu z kompletní nabídky....................41
Obr 7.3.1.2: Tvorba nového uzlu ze specifické nabídky následována automatickým přidáním povinného potomka....................42
Obr 7.3.6.1: Uspořádání stromu....................43
Obr 7.3.6.2: Strom se sbalenými podstromy....................44
Obr 7.3.7.1: Přeložení jedoduchého AST do zdrojového kódu....................44
Obr 7.3.7.2: Selhání překladu do zdrojového kódu....................45
Obr 7.3.7.3: Neúspěšný pokus o kompilaci stromu....................45
Obr 7.3.7.4: Podstrom pro psaní na standardní výstup....................46
Obr 7.3.7.5: Vykonaný "Hello World" program....................46
Obr 8.1.1: Diagram balíčků projektu....................47
Obr 8.2.1: Okno editoru....................48

Obr 8.2.1.1: Třída ScrollView..49
Obr 8.2.1.2: Kompozice třídy ScrollView...49
Obr 8.2.2.1: Diagram tříd v balíčku basic...50
Obr 8.2.3.1: Volání metod rozhraní Jointable v průběhu práce se spojem........................51
Obr 8.2.5.1: Uzel a jeho kompozice...52
Obr 8.2.6.1: Dva druhy vlastních dialogů editoru...53
Obr 8.3.2.1: Spoje po neúspěšné inspekci...55
Obr 8.3.5.1: Visitor pro překlad ze zdrojového kódu...58
Obr 8.3.6.1: Trojice tříd pro překlad z ASTNode...59
Obr 8.4.5.1: Diagram některých rozhraní v balíčku nodeTypes......................................63
Obr 8.4.6.1: Diagram tříd implementujících rozhraní MStatement...................................64
Obr 8.4.9.1: Diagram tříd implementujících rozhraní MExpression..................................67
Obr 8.4.14.1: Diagram tříd implementujících rozhraní MStatement a MExpression...........73
Obr 8.4.14.2: Zapojení přiřazení...73
Obr 8.4.14.3: Zapojení uzlu for s potomky uzly přiřazení...74
Obr 8.4.15.1: Diagram tříd implementujících rozhraní MAbstractType..............................75
Obr 8.4.16.1: Diagram tříd implementujících rozhraní MBodyDeclaration.........................76
Obr 8.4.17.1: Třída MParameter a její rozhraní...77
Obr 9.1.1: Testovací zapojení 1..80
Obr 9.1.2: Testovací zapojení 2..81
Obr 9.2.1: Tabulka naměřených výsledků pro úlohu 1..81
Obr 9.2.2: Tabulka naměřených výsledků pro úlohu 2..81
Obr 9.4.1: Vykonání cyklu..82
Obr C.1: Uzel typu Volání metody..96
Obr C.2: Uzel typu Volání metody s dynamicky přidanými spojy a popiskem....................97
Obr C.3: Připojení uzlu definující název metody a odmítnutí připojení nevhodného typu uzlu..97
Obr C.4: Dynamicky vytvořený spoj pro argumenty metody..98
Obr C.5: Vytvoření nového uzlu z kompletní nabídky..98
Obr C.6: Vytvoření nového uzlu z nabídky specifické pro spoj rodiče...............................99
Obr C.7: Editace uzlu..99
Obr C.8: Kopírování podstromu...100
Obr C.9: Hromadný výběr uzlu..100
Obr C.10: Sbalení podstromu..101
Obr C.11: Okno editoru s menu...102
Obr C.12: Množina možných stromů pro operaci..102
Obr C.13: Výběr stromu pro operaci...103
Obr C.14: Run submenu..103
Obr C.15: Přeložení jedoduchého AST do zdrojového kódu...104
Obr C.16: Selhání překladu..104
Obr C.17: Neúspěšný pokus o kompilaci stromu..105
Obr C.18: Vykonání "Hello World" programu...105
Obr C.19: File submenu...106
Obr C.20: Výběr formátu uložení stromu..106
Obr C.21: Tree submenu..107

Obr C.22: Uspořádání stromu..107
Obr C.23: Postupné rozbalování kompletně sbaleného stormu.......................................108
Obr C.24: Tree submenu...108
Obr C.25: Česká lokalizace...109
Obr C.26: Help submenu...109

1 Úvod

1.1 Motivace

Vývoj aplikací psaním instrukcí v textové formě do zdrojového kódu je synonymem programování. Zdrojový kód je pro člověka relativně pohodlná a lehce čitelná reprezentace programu. Horší je to již pro samotný počítač, který má program zpracovat. Zdrojový kód prochází několika fázemi transformace (dle konkrétního jazyka) do forem, kterým počítač rozumí.

Z tohoto pohledu lze říct, že zdrojový kód je pouhým náhledem na danou logickou strukturu, přesněji jedním z náhledů, který je vhodný pro lidské zpracování. Optikou softwarového inženýrství ale není úplně správným postupem manipulovat přímo se samotnou reprezentací a pak změnu propagovat.

Co by tedy mělo být oním modelem, na který je zdrojový kód pohledem? Jednoznačná odpověď nejspíše neexistuje – nabízí se ale jedna hojně užívaná datová struktura srozumitelná i pro člověka: abstraktní syntaktický strom. Lze si představit vývoj aplikace editací stromu?

Tato práce má sloužit jako prvotní pokus o implementaci takového přístupu.

1.2 Cíl práce

Pro splnění cíle této práce je potřeba nejdříve se obeznámit s významem abstraktního syntaktického stromu a jeho místem při kompilaci programů. Následně je potřeba zanalyzovat použitelnost dostupných prostředků k manipulaci s nimi. Dalším krokem je návrh grafického prostředí a celkové struktury programu, který by vykonával požadovanou funkcionalitu. Posledním bodem bude testování a demonstrace.

1.3 Struktura práce

Práce je rozdělena do několika kapitol:

Abstraktní syntaktický strom - obeznámení s problematikou AST

Nástroje pro manipulaci s AST – rešerše možných nástrojů

Java FX – velice krátké představení užité grafické technologie

Eclipse JDT – podrobnější náhled na použitý framework

Syntaxe Javy v Eclipse JDT – API frameworku

Návrh řešení - prezentuje vytvořený editor a jeho možnosti

Implementace – obsahuje popis technického řešení

Testování - demonstruje nasazení vytvořeného řešení

2 Abstraktní syntaktický strom

Význam abstraktních syntaktických stromů pro programovací jazyky si ukážeme na technologii Java.

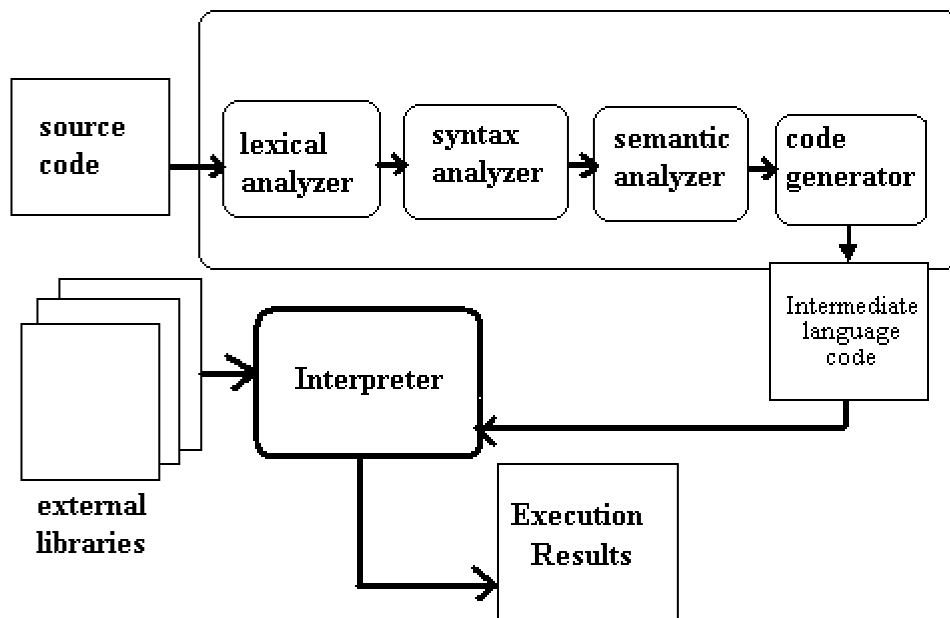
2.1 Java

Java je moderní, objektově orientovaný programovací jazyk navržený pro obecné použití. Jeho uvedení firmou Sun Microsystems se datuje k roku 1995 a od té doby se stal jedním z nejpopulárnějších a nejrozšířenějších programovacích jazyků vůbec. Jeho syntaxe je inspirována jazykem C (dá se dokonce mluvit o zjednodušené a upravené kopii C syntaxe). Mezi jeho další důležité vlastnosti patří robustnost, automatická správa paměti (garbage collector), distribuovanost nebo více-vláknovost.

Java se řadí mezi tzv. „interpretované“ jazyky, to znamená, že při spouštění zdrojového kódu je nejdříve vytvořen prostředník univerzální pro všechny platformy – bytekód. Ten je následně interpretován platformově specifickým způsobem.

2.2 Kompilace

Nyní se podíváme podrobněji na kompilaci, tj. proces transformace zdrojového kódu na bytekód.



Obr 2.2.1: Kompilace a interpretace jazyka Java[1]

Kompilátor obecně je program, který transformuje zdrojový kód napsaný v daném programovacím jazyce do jiného, většinou nižšího programovacího jazyka. To samozřejmě není triviální úkol a kompilátor se téměř vždy skládá z několika částí. Nejčastější hrubé rozdělení je na front-end a back-end.

Front-end – analyzuje zdrojový kód a vytvoří jeho interní reprezentaci. Navíc spravuje symbolickou tabulku.

Back-end – vstupem pro back-end je interní reprezentace vytvořená front-endem. Typicky provádí další analýzu, optimalizaci (tyhle dvě fáze jsou někdy považovány za vyčleněné z back-endu a součást middle-endu) a samotnou generaci výstupního kódu.

2.3 Front-end

Jak je naznačeno na obrázku v předchozí stati, front-end samotný se taky skládá z několika kroků:

Lexikální analyzátor:
Jeho úkolem je dekódovat vstupní proud znaků a vytvořit z nich tokeny vyššího řádu, které jsou srozumitelné pro parser (tak se někdy označují následující fáze front-endu). Token je atomická jednotka jazyka, například klíčové slovo nebo literál. Takže například z proudu znaků 'i', 'n', 't', 'v', 'a', 'r', '=', '6' a '6', je vytvořena sada tokenů „int“ „var“
„=“ a „66“.

Syntaktický analyzátor
Čte lineární sekvenci tokenů vygenerovanou lexikálním analyzátorem a buduje z ní interní reprezentaci programu dle pravidel formální gramatiky definující syntaxi jazyka. Obvykle

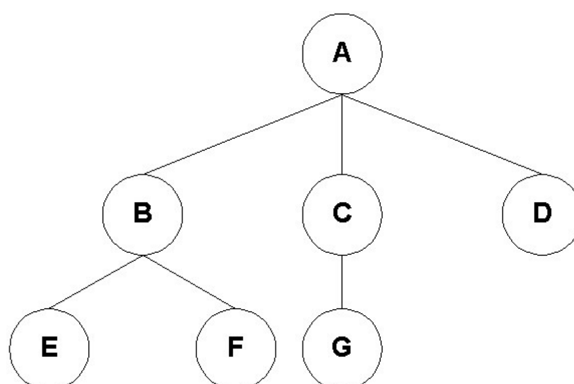
se jedná o stromovou reprezentaci, které se říká syntaktický strom.

Sémantický analyzátor
Zatímco v předchozí fázi se manipuluje s datovými strukturami čistě po formální stránce, v této fázi se pracuje s významy. K syntaktickému stromu jsou přidány sémantické informace a je vytvořena symbolická tabulka. Jsou provedeny kontroly jako například typová kontrola, nebo kontrola, zda byly všechny lokální proměnné inicializovány před použitím.

2.4 Syntaktický strom

Tématem této práce je právě interní reprezentace vzniklá v syntaktickém analyzátoru a užitá v sémantickém analyzátoru. Syntaktický strom je datová struktura, přesněji strom jak jej známe z teoretické informatiky – tedy neorientovaný souvislý graf, neobsahující kružnice. Navíc se jedná o strom tzv. „zakořeněný“, to znamená, že má kořen – uzel který určuje orientaci hran; všechny hrany vedou směrem od kořene. Díky tomu můžeme v rámci hierarchie mluvit o potomcích a dětech (uzel dál od kořene) a předcích a rodičích (uzel blíž ke kořenu) v rámci vztahů mezi uzly. Takovýto strom má kromě kořene dva druhy uzlů:

- list – uzel bez potomků;
- vnitřní uzel – uzel s jedním nebo více potomků.



Obr 2.4.1: Příklad kořenového stromu[2]

Syntaktický strom reprezentuje syntaktickou strukturu a vztahy mezi syntaktickými elementy v jiné formě než textová („stringová“) reprezentace kódu napsaného pomocí vyššího jazyka. Zatímco zdrojový kód je srozumitelnější pro člověka, pro počítač je čitelnější právě strom.

2.5 Konkrétní a abstraktní syntaktický strom

Až doteď jsem používal termín syntaktický strom pro označení dvou odlišných struktur používaných v kompilátoru: konkrétního syntaktického stromu (CST) a abstraktního

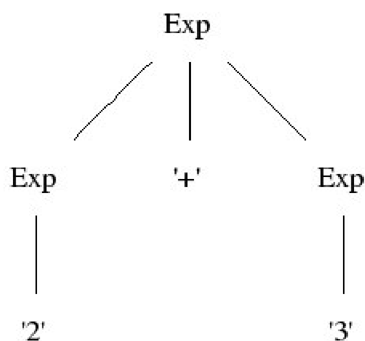
syntaktického stromu (AST). Rozdíl si ilustrujeme na velice zjednodušeném příkladu gramatiky s těmito pravidly[3]:

```
Exp ::= Exp "+" Exp ;
Exp ::= "2" ;
Exp ::= "3" ;
```

a z něj vygenerovaná věta:

2+3

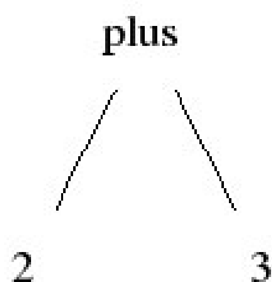
Konkrétní syntaktický strom
CST je strom vygenerovaný ze zdrojového kódu přímo pužitím gramatiky daného jazyka. Jedná se tedy o stromovou reprezentaci gramatiky jazyka a představuje detailnější (úplnou) formu kódu. Příklad kódu uvedeného výše by v tomto případě vypadal následovně:



Obr 2.5.1: Konkrétní syntaktický strom[3]

Jak je vidět, tento strom „bezeztrátově“ reprezentuje gramatiku jazyka a v něm napsaného kódu. Ve skutečných jazycích jsou pravidla samozřejmě neporovnatelně složitější, a to se odráží ve složitosti vygenerovaných CST.

Abstraktní syntaktický strom
AST vznikne zjednodušením CST o partikulárnosti dané gramatiky. Tím zůstane ve stromu jenom samotná syntaxe. Slovo „abstraktní“ tudíž značí, že strom není vybudován z konkrétní gramatiky. Měl by proto být více-méně stejný pro všechny jazyky.
V našem ukázkovém příkladě by AST vypadal následovně:



Obr 2.5.2: Abstraktní syntaktický strom[3]

Jednou z vlastností AST je, že kromě vynechání gramatických pravidel vynechává i některé formální prvky jazyka (typicky středník nebo některé závorky), které by byli v AST zbytečné protože jejich význam je implicitně obsažen ve struktuře stromu.

Terminologie
Termíny které používám v této práci můžou mít mírně odlišný význam než je běžné. Zatímco CST se někdy nazývá parsovací strom, nebo derivační strom, pro AST je používán taky název syntaktický strom. Tímto termínem jsem ale označil jak AST tak CST obecně, abych mezi nimi nemusel rozlišovat při prvním přiblížení problému.

2.6 Abstraktní syntaktický strom

Nyní se podíváme blíže na AST. Abstraktní syntaxe obecně je datová struktura nezávislá na konkrétní reprezentaci. Můžeme si například představit výraz „2+3“. Již samotná posloupnost znaků '2','+' a '3' je jednou z možností zápisu tohoto výrazu, tzv. infixový zápis. Další možnosti jsou:

(+ 2 3) - prefixový zápis
(2 3 +) - postfixový zápis

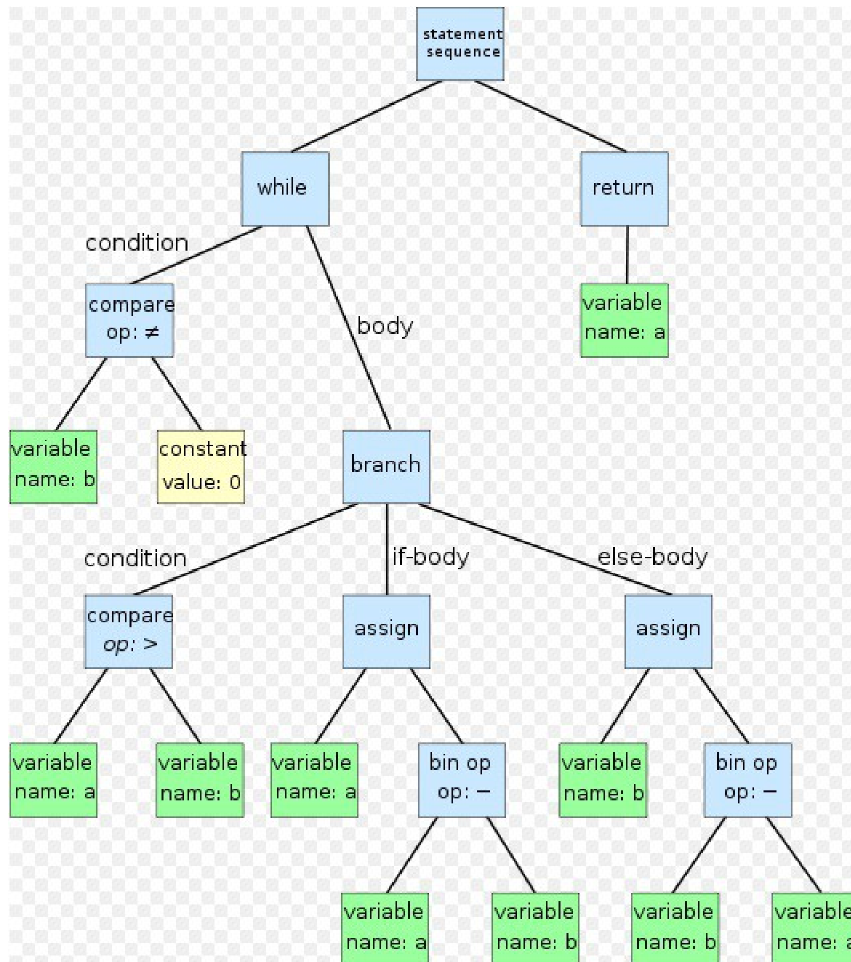
bipush 2 - bytekód
bipush 3
iadd

„součet dva a tři“ - čeština

Všechny zápisy výrazu jsou ekvivalentní ve smyslu, že mají stejný význam, i když se liší formou. Abstraktně bychom proto mohli výraz zapsat pomocí AST uvedeného na obrázku 2.5.2. Obdobně se můžeme podívat na Euklidův algoritmus pro určení největšího společného dělitele, zapsaný v pseudokódu[4]:

```
while b ≠ 0
        if a > b
                a := a − b
        else
                b := b − a
return a
```

Tento kód lze namodelovat následujícím AST:



Obr 2.6.1: Abstraktní syntaktický strom euklidova algoritmu[4]

Tento obrázek ilustruje další vlastnost AST: vnitřní uzly stromu jsou operátory a listy jsou operandy. Lze si tady taky lépe představit další využití AST - optimalizaci kódu. Například ve stromu, který reprezentuje booleovský výraz může překladač vynechat druhou větev, pokud je první vždy pravdivá. Podobným způsobem se taky editace AST používá pro automatickou refaktorizaci existujícího kódu.

3 Nástroje pro manipulaci s AST

Manipulace AST samozřejmě není v silách samotné Javy. Pro programátorskou manipulaci s AST je proto potřeba poohlédnout se po frameworku který nabídne požadovanou, nebo podobnou, funkcionalitu.

3.1 JRefactory

Nástroj pro refactoring programovacího jazyka Java[5] . Funguje jako plugin pro několik IDE. Umožňuje následující operace:

- přemístnit třídu mezi balíčky;
- přejmenovat třídu;
- přidat abstraktní rodičovskou třídu;
- přidat dědickou třídu;
- odstranit prázdnou třídu;
- extrahovat rozhraní;
- přemístnit členskou proměnnou;
- přejmenovat členskou proměnnou;
- přemístnit metodu;
- extrahovat metodu;
- přejmenovat parametr.

Zdá se tedy, že tento framework neumožňuje generování nového kódu, jak bychom požadovali.

3.2 CodeModel

Knihovna pro generaci kódu[6]. Projekt vznikl oddělením od projektu JAXB, jehož součástí je generování kódu. Jeho sílu lze ilustrovat na následujícím příkladu:

```
JCodeModel codeModel = new JCodeModel();
String className = "net.cardosi.MyNewClass";
JDefinedClass definedClass = codeModel._class(className);
String fieldName = "intVar";
JFieldVar field = definedClass.field(JMod.PRIVATE, int.class, fieldName);
String setterMethodName = "setIntVar";
JMethod setterMethod = definedClass.method(JMod.PUBLIC, Void.TYPE,
        setterMethodName);
String setterParameter = "intVarParam";
setterMethod.param(int.class, setterParameter);
setterMethod.body().assign(JExpr._this().ref(fieldName),
        Jexpr.ref(setterParameter));
codeModel.build(new File("."));[6]
```

Tento kód po spuštění vygeneruje následující třídu:

```
public class MyNewClass {
  private int intVar;
  public void setIntVar(int intVarParam) {
  this.intVar = intVarParam;
  }
}
```

Každý konstrukt jazyka je modelován příslušnou třídou a požadovaný výsledný kód je postaven jejich postupnou agregací. Nejedná se přitom o explicitní práci s AST, ale to by nemuselo vadit. AST bychom mohli používat jenom na front-endu, který by se „překládal“ do formy použitelné CodeModelem. Další inspekce naznačuje, že CodeModel je dostatečně robustní i pro práci s takovými konstrukty jako generiky.

3.3 JavaParser

Detailnější popis zní „Java 1.5 Parser with AST generation and visitor support“[7]. Parser generuje AST který je možno editovat, nebo umožňuje vytvářet nový.
Příklad práce s tímto frameworkem:

```
CompilationUnit cu = new CompilationUnit();
ClassOrInterfaceDeclaration type = new
ClassOrInterfaceDeclaration(ModifierSet.PUBLIC, false, "GeneratedClass");
ASTHelper.addTypeDeclaration(cu, type);
```

```
MethodDeclaration method = new MethodDeclaration(ModifierSet.PUBLIC,
        ASTHelper.VOID_TYPE, "main");
method.setModifiers(ModifierSet.addModifier(method.getModifiers(),
        ModifierSet.STATIC));
ASTHelper.addMember(type, method);
Parameter param = ASTHelper.createParameter(ASTHelper.createReferenceType(
        "String",0), "args");
param.setVarArgs(true);
ASTHelper.addParameter(method, param);
BlockStmt block = new BlockStmt();
method.setBody(block);
```

Vygeneruje následující kód:

```
public GeneratedClass{
        public static void main(String[] args){}
}
```

Tady se již jedná o skutečnou konstrukci AST. Vypadá to, že je poněkud pracnější než CodeModel.

3.4 Eclipse JDT

Rozsáhlá sada nástrojů známého IDE Eclipse[8]. Ten ji používá k různým účelům jako jsou wizardy, refactoring, náhled na AST, nebo anotační procesing. Ve formě pluginu do IDE je nabízeno programátorům komplexní editace a tvorba AST.

```
AST ast = AST.newAST(AST.JLS3);
CompilationUnit unit = ast.newCompilationUnit();
TypeDeclaration type = ast.newTypeDeclaration();
type.setInterface(false);
type.modifiers().add(ast.newModifier(
        Modifier.ModifierKeyword.PUBLIC_KEYWORD));
type.setName(ast.newSimpleName("HelloWorld"));
MethodDeclaration methodDeclaration = ast.newMethodDeclaration();
methodDeclaration.setConstructor(false);
List modifiers = methodDeclaration.modifiers();
modifiers.add(ast.newModifier(Modifier.ModifierKeyword.PUBLIC_KEYWORD));
modifiers.add(ast.newModifier(Modifier.ModifierKeyword.STATIC_KEYWORD));
methodDeclaration.setName(ast.newSimpleName("main"));
methodDeclaration.setReturnType2(ast.newPrimitiveType(PrimitiveType.VOID));
SingleVariableDeclaration variableDeclaration =
        ast.newSingleVariableDeclaration();
variableDeclaration.setType(ast.newArrayType(ast.newSimpleType(
        ast.newSimpleName("String"))));
```

```
variableDeclaration.setName(ast.newSimpleName("args"));
methodDeclaration.parameters().add(variableDeclaration);
org.eclipse.jdt.core.dom.Block block = ast.newBlock();
methodDeclaration.setBody(block);
type.bodyDeclarations().add(methodDeclaration);
unit.types().add(type);[8]
```

Tenhle kód vygeneruje třídu podobnou té v předchozím příkladě:

```
public class HelloWorld {
    public static void main(String[] args) {
    }
}
```

3.5 Diskuse

První ze zkoumaných možností, JRefactory, nesplňuje požadavky na funkcionalitu, proto nemá smysl o něm uvažovat.

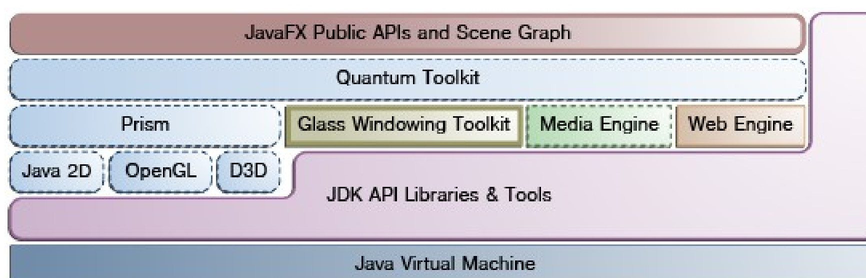
CodeModel se naproti tomu jeví jako schopný nástroj s intuitivním API, jehož pomocí by šlo simulovat tvorbu AST. Výhodou proti konkurentům se jeví nejúspornější aparát. JavaParser o mnoho nezaostává, a Eclipse JDT taky ne, ač pro vygenerování třídy bylo potřeba nejvíc kódu ze všech tří frameworků.

Eclipse JDT má ale obrovskou výhodu v tom, že je to projekt nejživější. CodeModel mírně zaostává v dokumentaci a příkladech na internetu. Poslední verze vyšla 02. 12. 2011 (nejspíše jako reakce na Java 1.7 která byla vydána několik měsíců předtím) a není jasné, zda je ještě ve vývoji. To JavaParser se zastavil na Javě 1.5 a co se týče dokumentace a „ekosystému“, nemůže se měřit ani s CodeModelem, natož s Eclipse JDT. Ten je stále ve vývoji (stojí za ním velký hráč, který nejspíš jen tak neodejde) a má již například taky podporu Javy 1.8, vyčerpávající dokumentaci a největší komunitu.
Volba frameworku pro tuto práci tudíž byla jasná, a detailněji si ho představíme v kapitole 5.

4 Java FX

Dalším stavebním kamenem této práce je JavaFX[9]. Jedná se o technologii pro front-endovou část a budu se jí věnovat jenom okrajově.

JavaFX je softwarová platforma určena pro tvorbu tzv. Bohatých internetových aplikací - Rich Internet Application (RIA) vyvíjena firmou Oracle. JavaFX má nahradit zastarávající Swing jako standardní nástroj pro tvorbu GUI javovských aplikací. V minulosti byl její součásti deklarativní skriptovací jazyk JavaFX Script, jehož vývoj byl ale zastaven. JavaFX aplikace se tak již vyvíjejí v tradičním Java kódu. Podobně jako Java samotná si i JavaFX zakládá na přenositelnosti a univerzálnosti.



Obr 4.1: Technologie Java FX[10]

Paradoxně je základním datovým typem JavaFX aplikací taktéž strom. Každý grafický prvek JavaFX aplikace je uzlem (Node) grafu uspořádaným ve stromu nazývaným

Scene Graph. Další obsah můžou tvořit stavy (State) nebo efekty (Effect) připojené k uzlům Scene Graphu.

5 Eclipse JDT

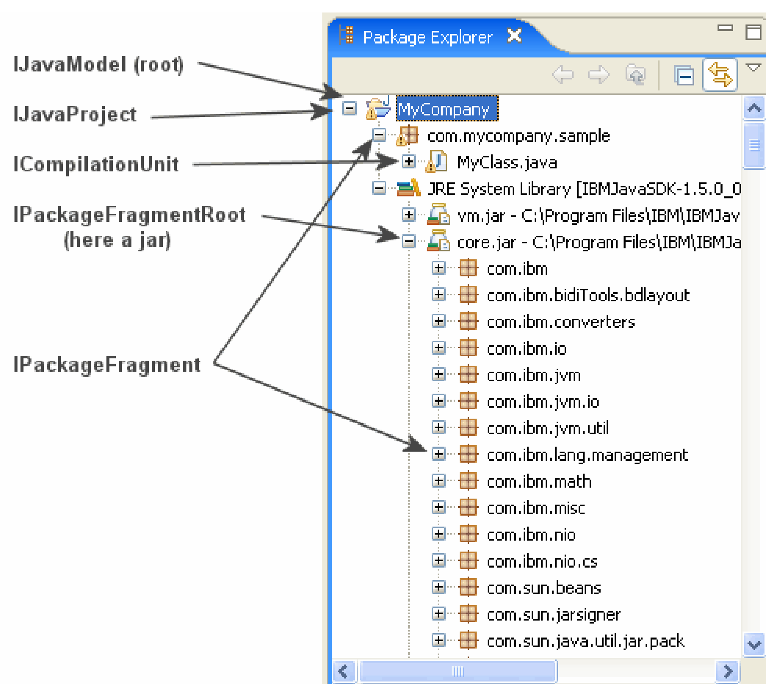
Eclipse Java development tooling (JDT) umožňuje uživateli psát, kompilovat, testovat, debugovat, a editovat programy napsané v programovacím jazyku Java. Dá se na něj nahlížet jako na sadu pluginů (ze které nás bude zajímat pouze podmnožina) přidávajících specifické chování k Java platformě potřebné pro interakci s javovskými programy.

Konkrétně nabízí tyto zajímavé možnosti[11]:

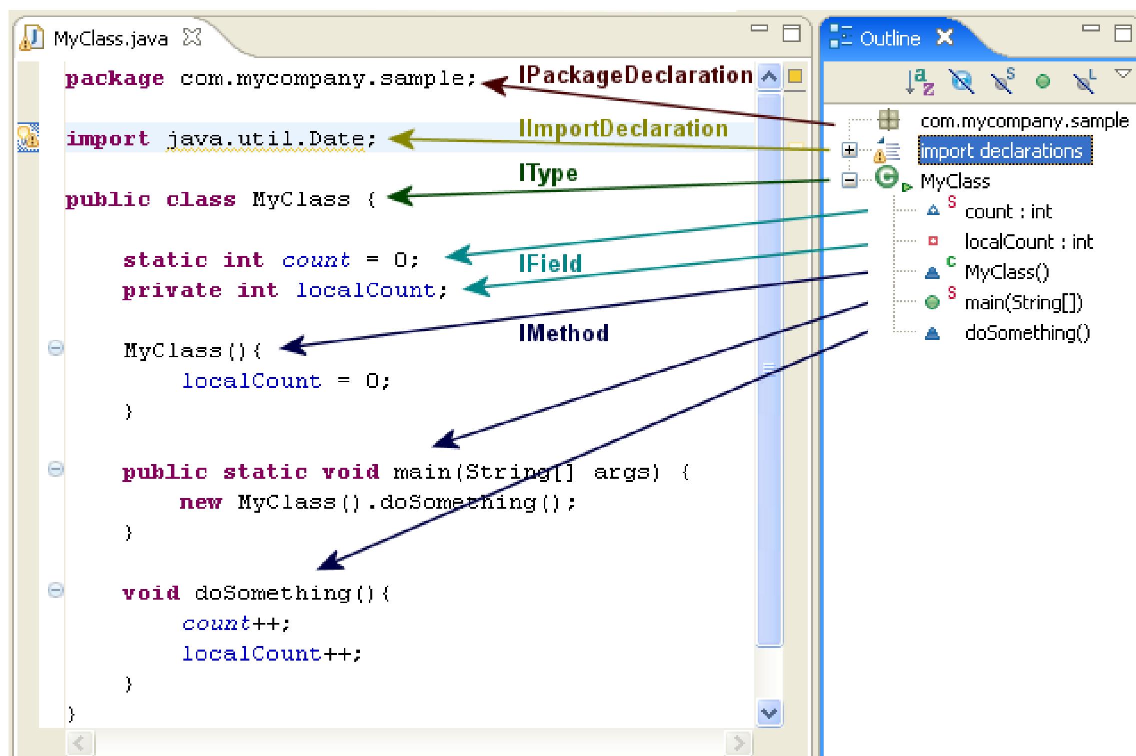
- programátorsky manipulovat javovskými zdroji, například vytvářet projekty; generovat nebo editovat zdrojové kódy či detekovat problémy v kódu;
- programátorsky spouštět Java programy.

5.1 Java Model

Množina tříd, které modelují struktury javovského projektu[12]. Činí tak pomocí tzv. Java elementů (konkrétně tříd implementujících rozhraní IJavaElement) uspořádaných v hierarchické formě. Existuje tak například IJavaProject nebo IPackageDeclaration, „nejnižším“ elementem je člen třídy (například metoda).



Obr 5.1.1: Struktura projektu v Java Model [12]



Obr 5.1.2: Struktura třídy v Java Model [12]

Java Model umožňuje jednoduchou manipulaci projektu na úrovni elementů pomocí Java element API. Například použití metod ***createField*** a ***createMethod*** vyvolá akce intuitivně pochopitelné z názvu. Bohužel ale Java Model neumožňuje úplnou manipulaci zdrojového kódu, takže pro naše účely nebude dostačující.

5.2 JDT API

Nad Java Model je postaven silnější nástroj, AST API. Ten byl představen již v předchozí sekci, a bylo demonstrováno, že s jeho pomocí lze manipulovat javovským kódem v plném rozsahu. Jedná se přitom skutečně o manipulaci se stromovou strukturou AST.
Existuje několik způsobů, jak získat CompilationUnit (AST):

Zaprvé použitím ASTParser. Jak název napovídá, tato třída parsuje vstupní soubor a vytváří z něj AST. Vstupem může být pole znaků, .class soubor (musí ale mít připojený zdrojový soubor) nebo jiná CompilationUnit. Příklad použití:

```
char[] source = ...;
ASTParser parser = ASTParser.newParser(AST.JLS3);
parser.setSource(source);
Map options = JavaCore.getOptions();
JavaCore.setComplianceOptions(JavaCore.VERSION_1_5, options);
parser.setCompilerOptions(options);
CompilationUnit result = (CompilationUnit) parser.createAST(null);
```

Další způsob je vytvořit CompilationUnit pomocí továrních metod třídy AST. Tento způsob byl předveden v části 3.4. Dále existují dva způsoby jak manipulovat s AST:

- přímá modifikace;
- zaznamenáváním změn v separátním protokolu a jejich následnou aplikací; spravovanou instancí třídy ASTRewrite.

První možnost byla opět předvedena v části 3.4, podotkneme jenom, že při manipulaci existující struktury je potřeba „zapnout nahrávání“ změn.

```
unit.recordModifications();
```

Blíže se podíváme na manipulaci pomocí ASTRewrite[13].

```
rewrite = ASTRewrite.create(unit.getAST());
VariableDeclarationStatement statement =
        createNewVariableDeclarationStatement(manager, ast);
int firstReferenceIndex = getFirstReferenceListIndex(manager, block);
ListRewrite statementsListRewrite = rewrite.getListRewrite(block,
Block.STATEMENTS_PROPERTY);
statementsListRewrite.insertAt(statement, firstReferenceIndex, null);
```

Aplikace změn do zdrojového kódu
Pokud byla CompilationUnit získána z existujícího zdroje, můžeme do ní změny v AST přepsat, a to pomoci objektu třídy TextEdit[14].

```
TextEdit edits = rewriter.rewriteAST();
Document document = new Document(unit.getSource());
edits.apply(document);
unit.getBuffer().setContents(document.get());
```

6 Syntaxe Javy v Eclipse JDT

V této části se ve zkratce podíváme na syntaxi Javy. Jednotlivé stavební prvky syntaxe jsou totiž uzly AST grafu, jejich znalost bude proto klíčová pro práci s editorem. Inspirací pro tuto část je oficiální dokumentace Eclipse JDT [15], čerpám z ní jak informace, tak názvy a notaci. Ta je celkem intuitivní - předpokládám, že je odvozena z rozšířené Backus-Naurovi formy pro zápis bezkontextových gramatik:

- [] značí 0 až 1 výskyt;
- {} opakování, 0 až * výskytů;
- | alterace.

Až na výjimky nebudu uvádět konstrukty, které nebyly v této práci implementovány.

Compilation Unit (kompilační jednotka): je v podstatě výsledkem psaní kódu – entita která je uceleným vstupem pro kompilátor. Lze říct, že je to vlastně jeden soubor s příponou *.java.* V AST představuje kořen stromu. Obsahuje:

[PackageDeclaration]
{ ImportDeclaration }
{ AbstractTypeDeclaration | ; }

První dva body zde opomeneme a podíváme se pouze na deklaraci třídy.

Type Declaration (deklarace typu): může být dvojího typu: deklarace třídy nebo rozhraní. Na stejné úrovni stojí ještě deklarace enumerace nebo anotace, ale těm (ani rozhraní) se nebudeme věnovat. Deklarace třídy tedy obsahuje:

{ ExtendedModifier } class Identifier
[< TypeParameter { , TypeParameter } >]
[extends Type]
[implements Type { , Type }]
{ { ClassBodyDeclaration | ; } }

Tedy to, co od deklarace třídy očekáváme: modifikátory, děděné třídy a rozhraní, případně typový parametr. Pro tuto práci ale bude zajímavé podívat se pouze na název a samotné tělo třídy.

Body Declaration (deklarace těla): na body declaration se můžeme dívat jako na abstraktní nadtřídu těchto možných případů:

- ClassDeclaration,
- MethodDeclaration,
- ConstructorDeclaration,
- FieldDeclaration.

ClassDeclaration zde je přitom samotná deklarace třídy. Třída se tedy skládá z množiny metod, konstruktorů, členských proměnných a případně vnořených tříd.

Method Declaration (deklarace metody) vypadá následovně:

{ ExtendedModifier }
[< TypeParameter { , TypeParameter } >]
(Type | void) * Identifier (
[FormalParameter{ , FormalParameter }]) **{[ ] }
[throws TypeName { , TypeName }]
(Block | ;) ***

Zde je zajímavá deklarace návratové hodnoty (*), parametrů (**) a samotného těla. Deklarace konstruktoru se liší jenom minimálně (pro naše účely konkrétně chybí deklarace návratového typu a jména), proto ji zde nebudu uvádět.

Field Declaration (deklarace členské proměnné)

{ ExtendedModifier } Type VariableDeclarationFragment

VariableDeclarationFragment zde odpovídá:

Identifier { [] } [= Expression]

Type (reference typu) použitá například při deklaraci návratových typů metod, přesněji jejich abstraktní nadtřída. Konkrétní implementací jsou například třídy reprezentující primitivní datové typy, typ void a SimpleType, což je třída která referencuje jakýkoliv jiný Type.

Expression (výraz): jeden z velmi základních konstruktů většiny jazyků. Zjednodušeně se dá říct, že je to konstrukt, který lze zredukovat na hodnotu (včetně jakýchkoliv objektů). Řadí se mezi ně tedy identifikátory (například jméno proměnné), literály ('c', 6 nebo true) a operátory (operátor + vrací hodnotu vzešlou ze součtu operandů). Operátorem je taky konstrukt volání metody (method invocation) na objektu. Konkrétních výrazů je větší množství, proto zde jejich syntaxi nebudu popisovat.

Statement (příkaz): druhý ze základních konstruktů jazyka. Na rozdíl od výrazu, příkaz nelze zredukovat na hodnotu. Příkaz je jednotkou exekuce programu. Většina z nich patří mezi tzv. *control flow statements,* to znamená, že řídí běh programu (například příkaz *for*).

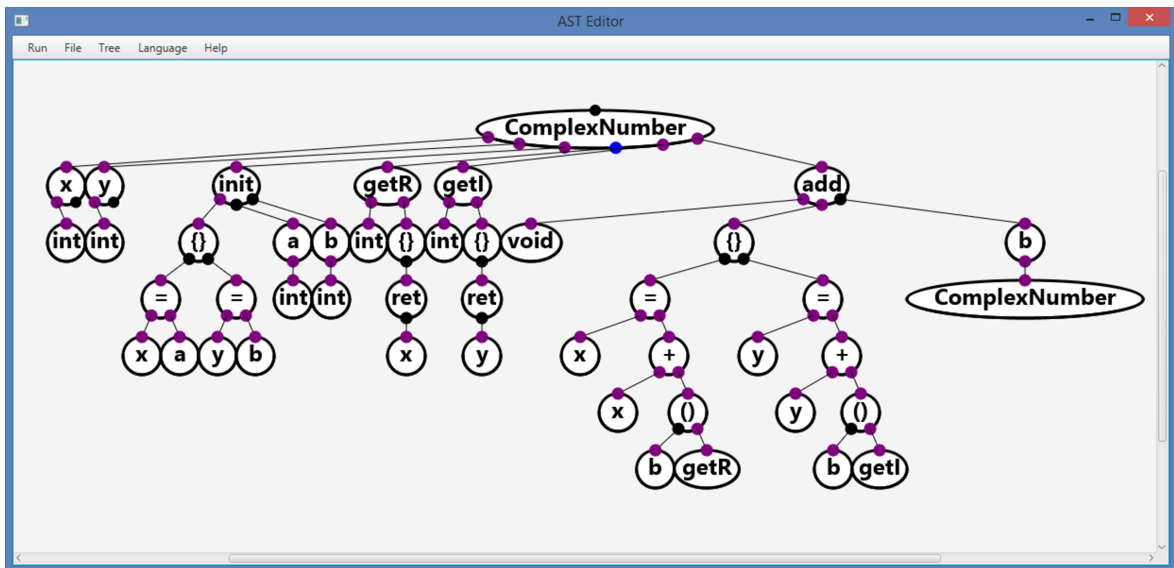
Expression Statement: speciální případ výrazu, který je zároveň příkazem.
Patří sem:

- přiřazení,
- inkrementace a dekrementace,
- invokace metody,
- invokace konstruktoru.

Jsou to všechno případy výrazů, které zároveň vykonávají nějakou činnost. Můžeme je použít pro jejich hodnotu (například návratovou hodnotu při invokaci metody jako parametr jiné metody) nebo jenom chtít vykonat danou funkcionalitu. Příkazem se stanou prostým použitím středníku. Eclipse JDT framework pro daný rys využívá uzel ExpressionStatement, který sám je Statement a má jediného potomka – Expression. Daný výraz tak „zaobaluje“. Já jsem ale použil jiný postup, který vysvětlím později.

7 Návrh řešení

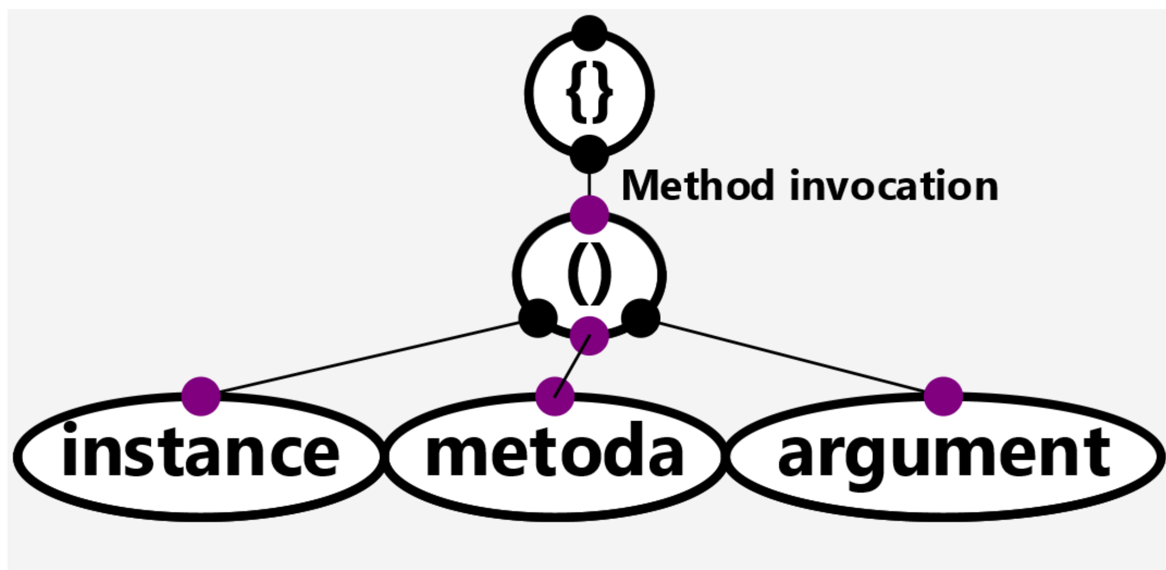
V této části představím řešení úlohy. Učiním tak prezentací grafického editoru, jeho možností a funkcí. Pro podrobnější informace ohledně použití viz příloha C.



Obr 7.1: Okno editoru a vytvořený strom

7.1 Uzel

Základní stavební prvek stromu je reprezentován elipsou (ve specálním případě obdélníkem) jenž má po obvodu kroužky (spoje) sloužící jako prostředník pro napojení dalších uzlů. Dále je uprostřed reprezentován štítkem udávajícím buď typ uzlu, nebo jeho hodnotu (například u literálů) a po najetí myší také popiskem obsahujícím explicitnější vyjádření typu uzlu.



Obr 7.1.1: Uzel připojen k rodičovi a potomkům

7.2 Spoje

Spojení uzlů se provádí tahem myší vycházejícím ze spoje. Každý spoj pro připojení potomků nese informaci o tom, jaký typ podstromu lze na danou pozici v rámci uzlu připojit. Například na obrázku 7.2.1 uzel typu Volání metody očekává na všech pozicích uzel typu Výraz (Expression) a na všech pozicích se mu dostávají uzly typu Identifikátor (což je potomek Výrazu). Pokus o připojení uzlu nevhodného typu je signalizován červeným zbarvením uzlu a je odmítnut.

Počet a typy spojů jsou determinovány typem uzlu. Některé uzly navíc umožňují připojení proměnného počtu některých podstromů – například volání metody může mít žádný až mnoho argumentů. Tyto uzly proto podporují dynamickou tvorbu spojů (najetím myší na příslušné místo) ve dvou různých módech: nové spoje se vytvářejí za statickými spoji, nebo má uzel pouze dynamické spoje.

7.3 Navržené operace

Vytvořený grafický editor podporuje následující sadu operací pro uzly a stromy:

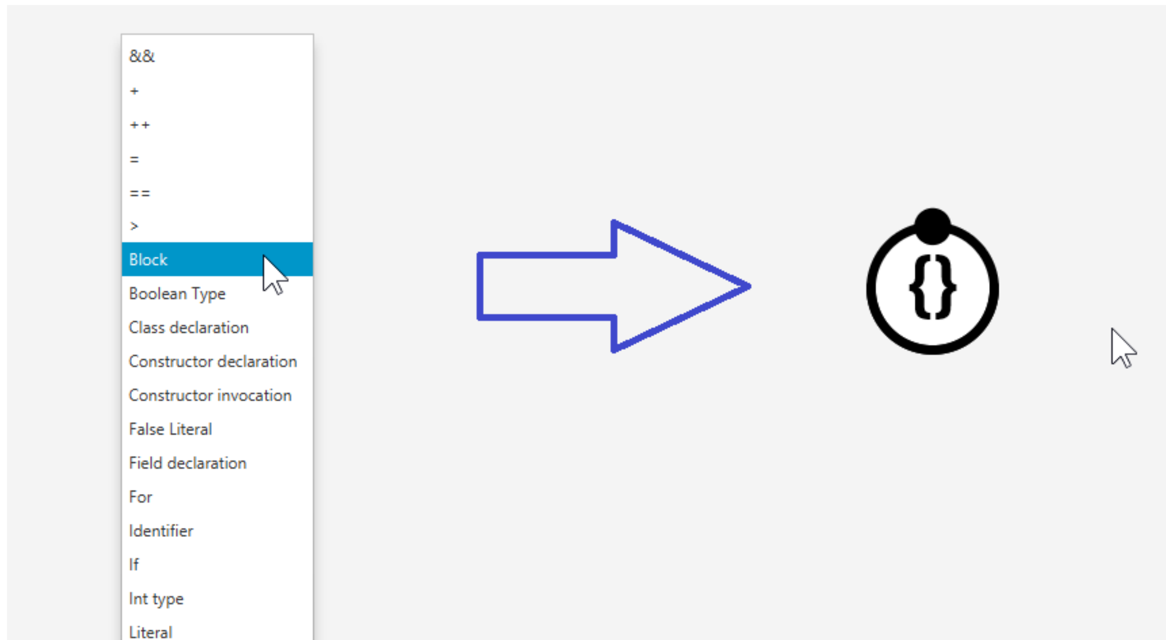
- tvorba nových uzlů;
- mazání uzlů;
- editace uzlů;
- spojování uzlů a tvorba stromů;
- kopírování stromů a podstromů.

A dále nabízí funkcionalitu spojenou s:

- úpravami vizuální reprezentace stromů;
- spuštěním vytvořených stromů;
- perzistencí (ukládání a načtení stromů).

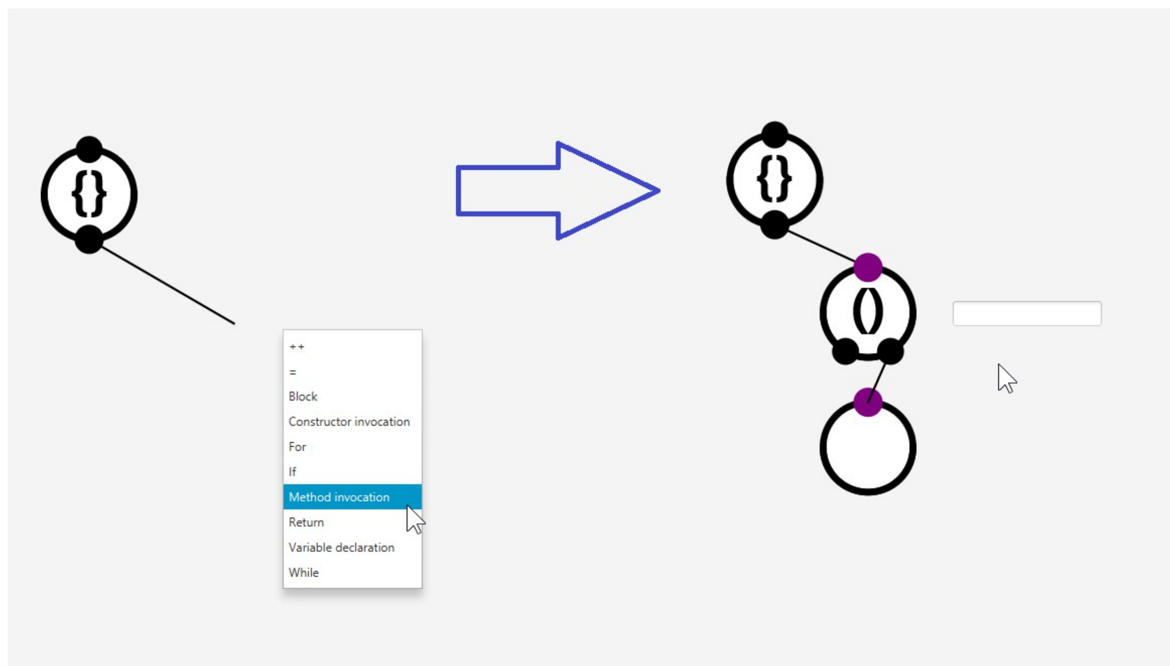
7.3.1 Tvorba uzlu

Nový uzel se vytvoří výběrem z rozbalovací nabídky, která obsahuje všechny uzly známé editoru. Nabídka se vyvolá zmáčknutím sekundárního tlačítka myši nad plochou.



Obr 7.3.1.1: Tvorba nového uzlu z kompletní nabídky

Druhou možností je využít informaci o přijatelných uzlech pro konkrétní spoj. Vyvoláním hrany ze spoje (drag) a jejím „puštěním“ nad plochou „drop“ se vyvolá nabídka omezená na uzly splňující podmínku zdrojového spoje. Pokud nabídka obsahuje jediný typ uzlu, je vytvořen automaticky.



Obr 7.3.1.2: Tvorba nového uzlu ze specifické nabídky následována automatickým přidáním povinného potomka

V některých případech, například při přidávání uzlu typu Volání metody, je automaticky přidán potomek (v tomto případě typu Identifikátor). Je to proto, že na dané pozici je povinný potomek a navíc je možný jediný typ.

7.3.2 Mazání uzlu

Nejjednodušší způsob jak smazat uzel je zmáčknutím kolečka myši nad uzlem. Další možnost je pomocí kontextové nabídky (která se vyvolá zmáčknutím sekundárního tlačítka myši nad uzlem). Zmáčknutím klávesy DELETE se vyvolá smazání všech uzlů, které aktuálně mají zaměření (focus). To lze využít pro hromadné mazání, protože zaměřeno může být najednou více uzlů. Pokud smažu uzel, který je kořenem sbaleného podstromu (viz 7.3.6), je smazán celý podstrom.

7.3.3 Editace uzlu

Editovat lze pouze uzly s měnitelnou hodnotou, například Identifikátor nebo Literál. Editace je velice jednoduchá – v kontextové nabídce uzlu vyberu možnost „Edit node“ a hodnotu upravím v dialogu. Identifikátoru lze zadat hodnotu neplatnou v jazyku Java, ale strom obsahující takovýto uzel nepůjde přeložit.

7.3.4 Spojování uzlů a tvorba stromů

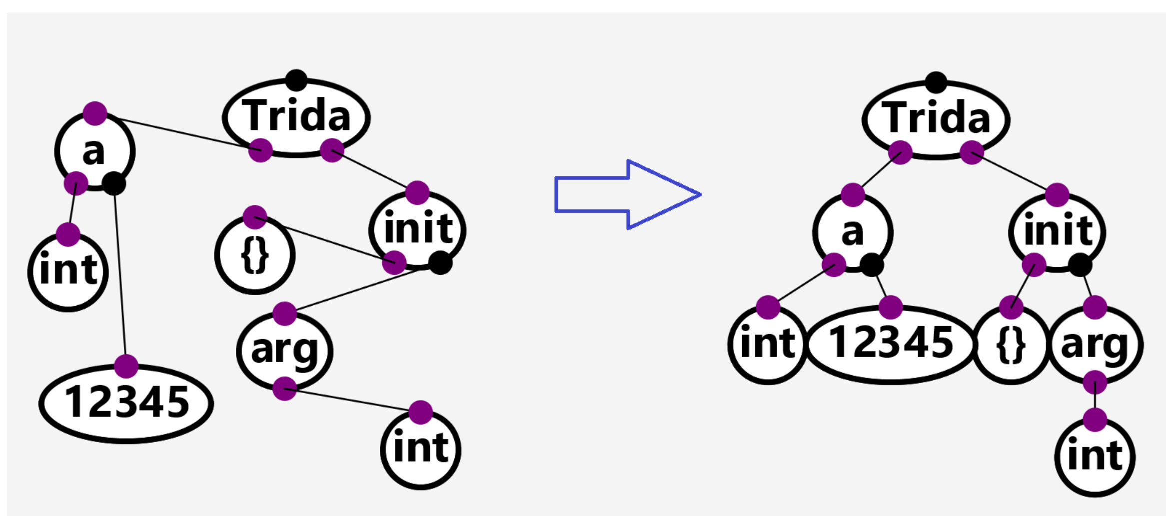
Jak již bylo zmíněno, uzly se napojují pomocí hran připojením k příslušným spojům. Hranu lze smazat kolečkem myši, nebo vytvořením nové hrany ze spoje, ke kterému je připojená. Editor může obsahovat několik stromů najednou. Nejvyšším kořenem může být Deklarace třídy, ale pracovat lze i se stromy s libovolným kořenem.

7.3.5 Kopírování stromů a podstromů

Editor umožňuje zkopírovat kompletní strom (nebo podstrom) a vložení kopie na plochu. Operace kopírování je přístupná skrze kontextovou nabídku kořenu. Vložení kopie je dále možné vyvolat z kompletní rozbalovací nabídky pro přidávání nových uzlů. V případě, že kořen kopie vyhovuje, je přístupná taky ze specifické nabídky spoje.

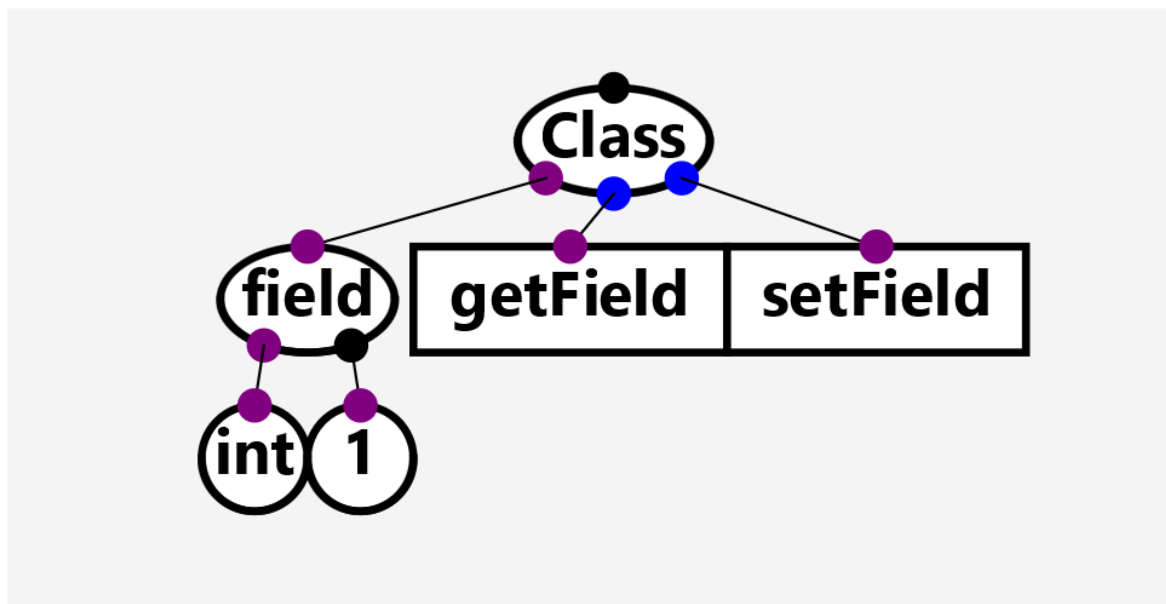
7.3.6 Vizuální reprezentace stromu

Uspořádání stromu
Strom je uživatelem tvořen do neuspořádané struktury. Pro lepší přehlednost nabízí editor možnost strom uspořádat do struktury pravidelné. Operaci lze vyvolat v menu, nebo pomocí gesta (viz C.7).



Obr 7.3.6.1: Uspořádání stromu

Sbalení a rozbalení stromu
Už při relativně malém rozsahu se strom stává docela nepřehledným. Aby bylo možné schovat části stromu, které nejsou v danou chvílí pro uživatele zajímavé, je možné sbalit podstromy.



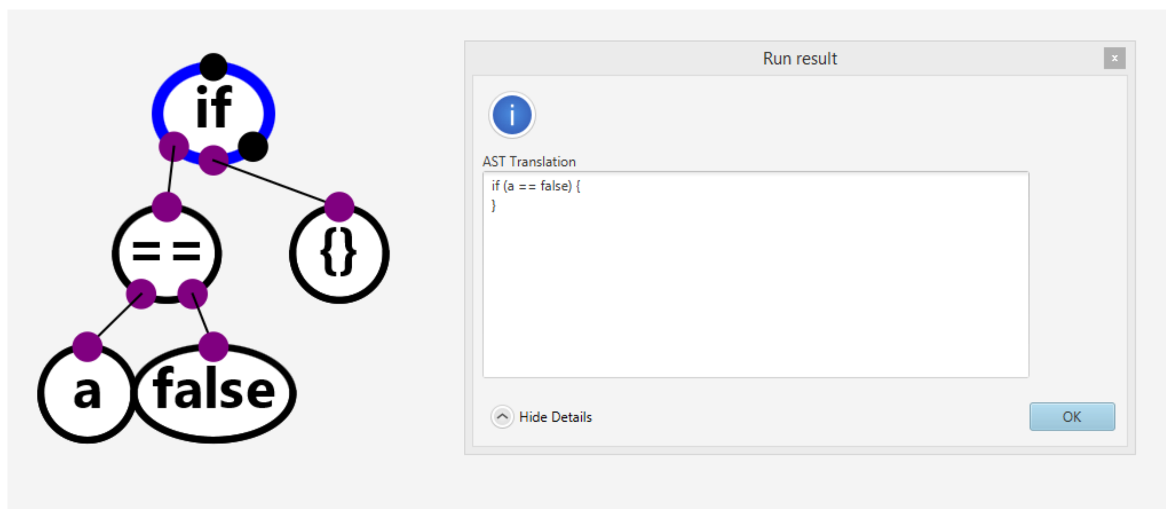
Obr 7.3.6.2: Strom se sbalenými podstromy

Konkrétní podstrom lze sbalit (rozbalit) dvojitým kliknutím na kořen. Globální sbalení a rozbalení je nabídnuto v menu.

7.3.7 Spuštění stromu

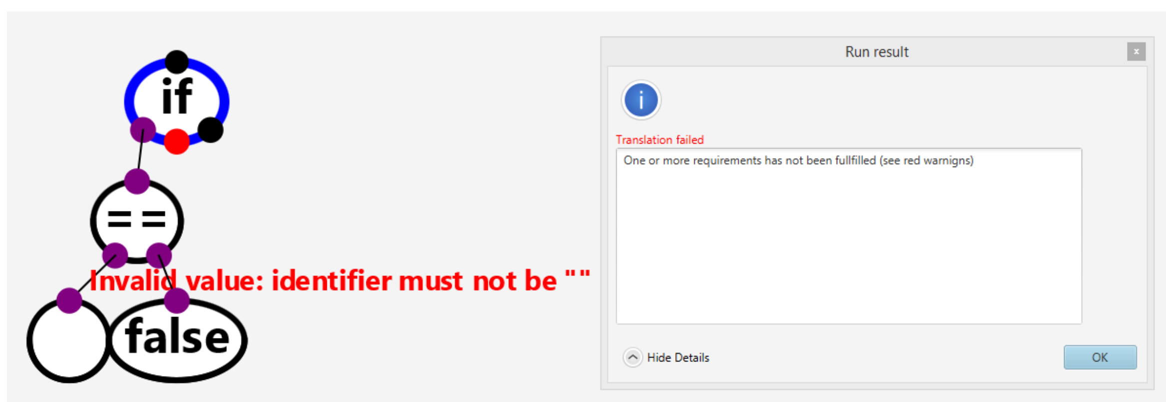
Vytvořený strom lze „spustit“ na třech úrovních. Jednotlivě je lze spustit z hlavního menu, všechny najednou pomocí gesta.

Překlad do zdrojového kódu



Obr 7.3.7.1: Přeložení jedoduchého AST do zdrojového kódu

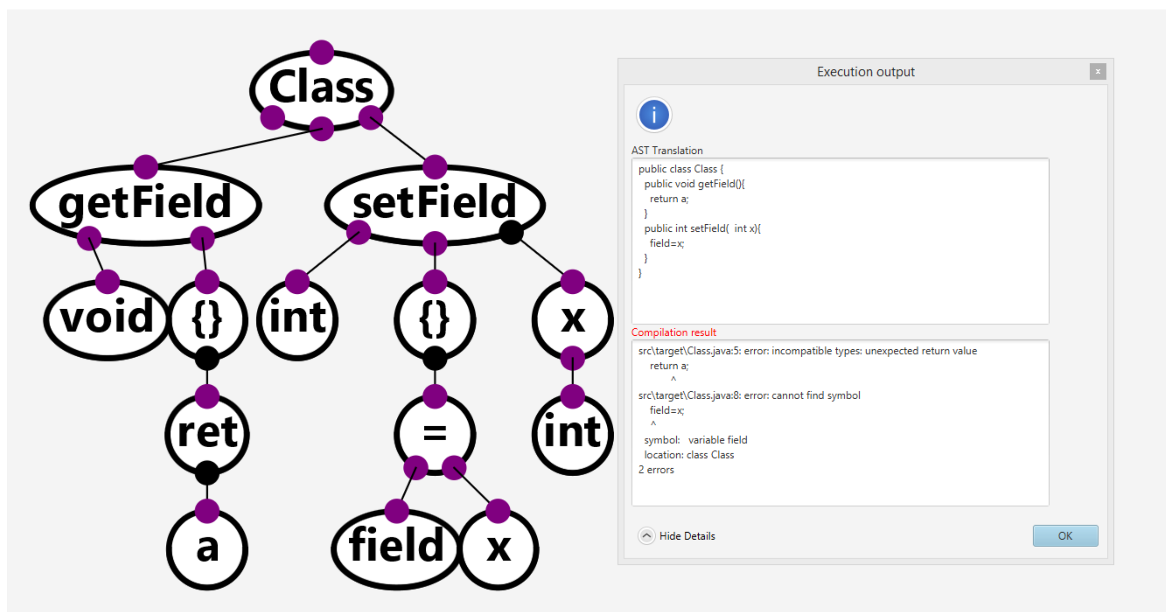
Tato operace vytvoří z AST zdrojový kód. Pro úspěšný překlad je potřeba splnit dva druhy podmínek. Musí být připojeny podstromy na všechny spoje, kde má daný uzel povinného potomka, a musí být splněny podmínky přímo na uzlech (například platné Java identifikátory). Další správnost stromu je dána restriktivní politikou připojování, ale může se stát, že ač půjde strom přeložit do zdrojového kódu, nepůjde zkompilovat.



Obr 7.3.7.2: Selhání překladu do zdrojového kódu

Kompilace

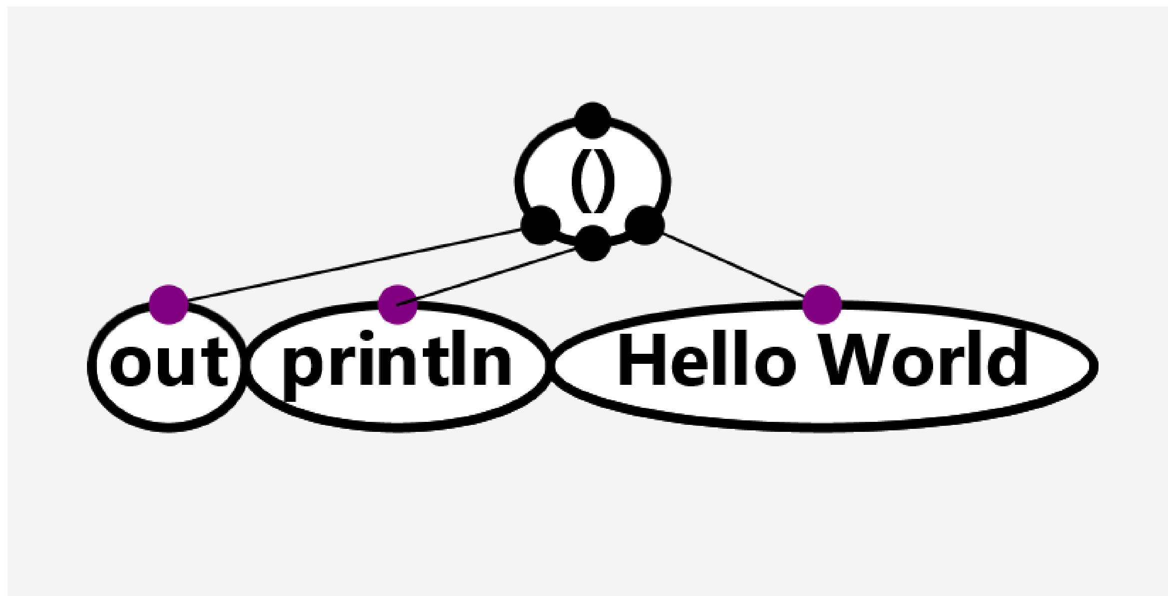
Proběhne-li úspěšný překlad, lze spustit kompilaci. Narozdíl od překladu, kompilovat lze pouze deklarace typů (třídy). Editor pracuje pouze se syntaktickým stromem, ale při kompilaci následuje syntaktickou analýzu analýza sémantická. Validní syntaktický strom proto nemusí splňovat sémantické podmínky a kompilace může selhat. V příkladu na obrázku 7.3.7.3 je například použit nedeklarovaný symbol. Využití java kompilátoru vyžaduje, aby byl editor spouštěn pomocí JDK, JRE není dostačující.



Obr 7.3.7.3: Neúspěšný pokus o kompilaci stromu

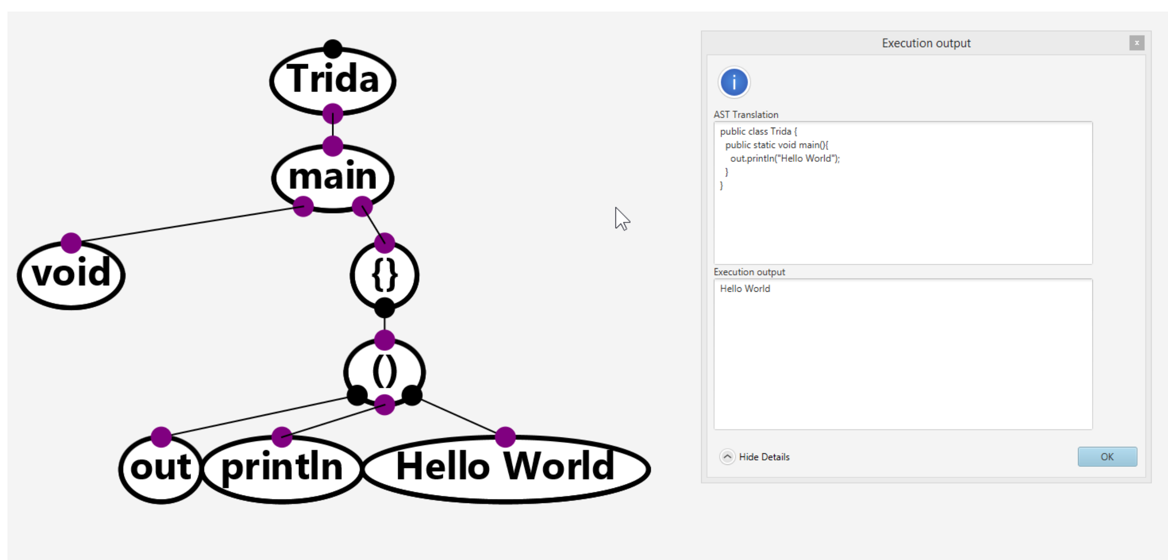
Vykonání

Po úspěšné kompilaci vznikne dočasný soubor s bytekódem a obsahuje-li metodu ***main*** (nemusí splňovat plný kontrakt pro metodu ***main***, kterou spouští JVM, stačí, aby se shodoval název), editor jej může spustit. Editor přesměruje výstupní proudy System.out a automaticky doplní jeho statický import, takže do konzole lze z AST psát následovně:



Obr 7.3.7.4: Podstrom pro psaní na standardní výstup

Vykonání kompletního „Hello World“ programu vypadá následovně:



Obr 7.3.7.5: Vykonaný "Hello World" program

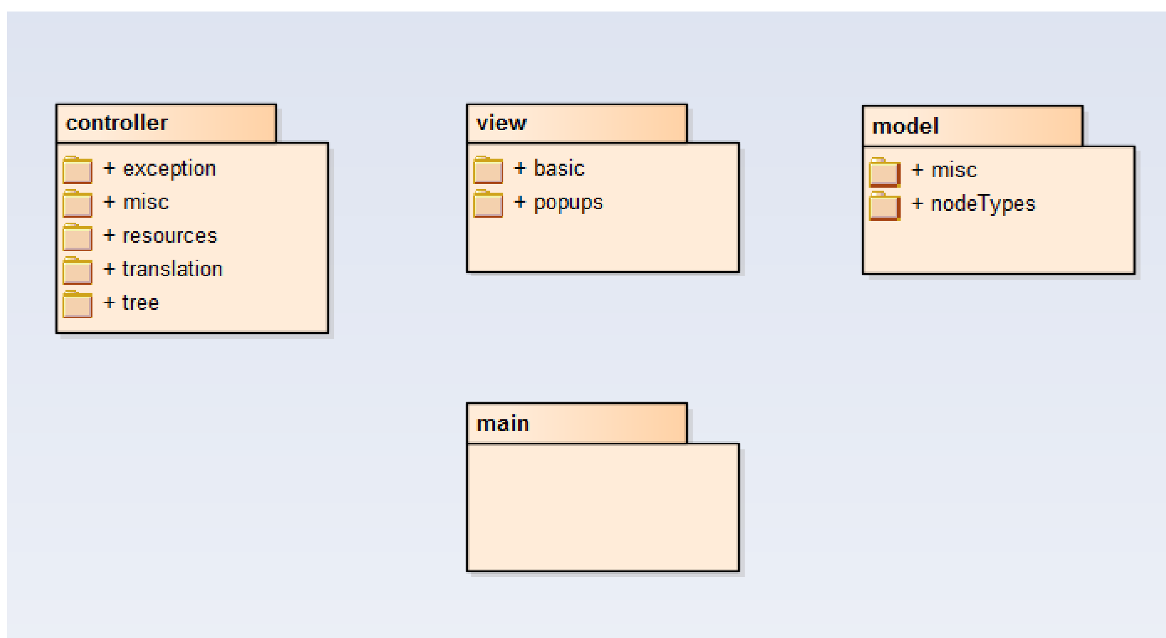
7.3.8 Persistence

Editor nabízí možnost uložení/načtení stromu ve dvou formátech. Prvním je práce se samotným zdrojovým kódem (.java nebo .txt). Při načítání a ukládání dochází k překladu tak, jak byl popsán v části 7.3.7. Načítat v tomto formátu lze pouze deklarace tříd. Druhým je vlastní formát editoru - .ast.

8 Implementace

Tato část se podívá blíže na implementaci jednotlivých součástí editoru. Postupně prozkoumáme strukturu třech hlavních balíčků projektu.

8.1 Základní struktura



Obr 8.1.1: Diagram balíčků projektu

Projekt je členěn do třech základních balíčků, dle návrhového vzoru MVC.

view – obsahuje základní třídu, která dodává hlavní scénu (Scene) JavaFX aplikace.
  basic – obsahuje všechny třídy pro grafickou reprezentaci stromu, uzlů, jejich částí atd.
  popups – editor využívá dvou vlastních „vyskakovacích“ grafických prvků pro komunikaci s uživatelem, jejich řešení se nachází v tomto balíčku

model – zde se nacházejí třídy které dávají grafické reprezentaci uzlu význam, určují jeho typ
  nodeTypes – obsahuje třídy modelující jednotlivé typy uzlů stromu ve smyslu, jak byly představeny v kapitole 6
  misc – obsahuje servisní třídu pro práci s Eclipse JDT objekty a některé další třídy

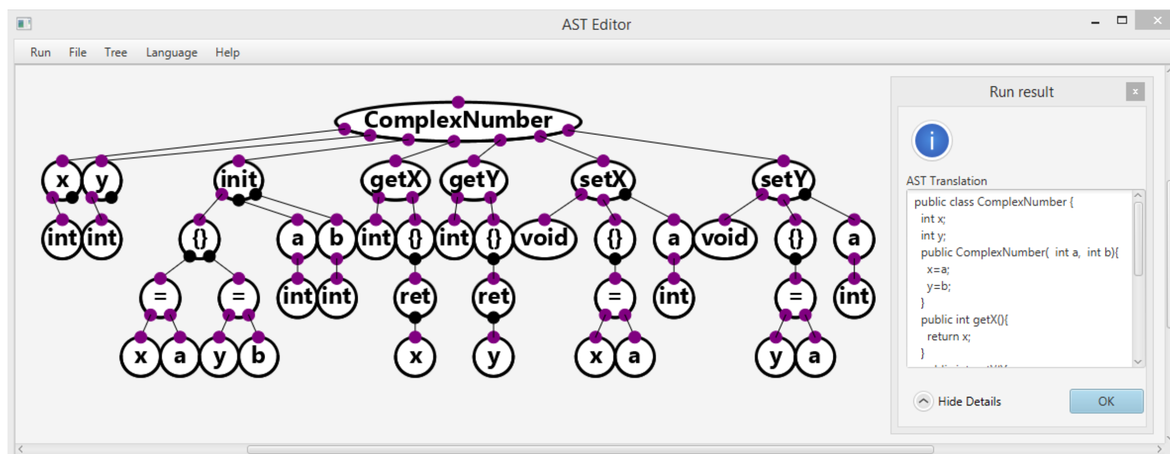
controller – hlavní třída Controller provádí globální řízení editoru
  tree – třídy a rozhraní nezbytné pro procházení stromu
  exception – třídy pro práci s výjimkami
  resources – obsahuje zejména jazykové lokalizace a některé konstanty
  translation – zde se nachází aparát potřebný pro načítání AST ze zdrojového kódu
  misc – některé další třídy nezbytné pro různe funkce editoru

main – zde se nachází jediná třída Start s metodou ***main***

Při detailním popisu budu postupovat „shora dolů“, podíváme se tedy nejdříve na grafickou reprezentaci a skončíme u modelu.

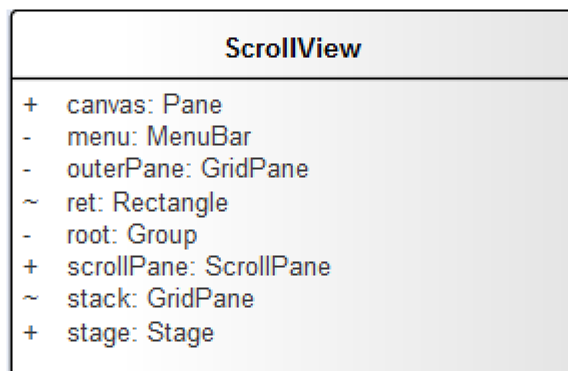
8.2 View

V tomto balíčku se nacházejí všechny třídy související s grafickou stránkou editoru. Strom se v editoru skládá z uzlů (třída NodeRepresentation) a spojovacích hran (třída TreeLine). Strom je umístněn na ploše implementované třídou ScrollView.



Obr 8.2.1: Okno editoru

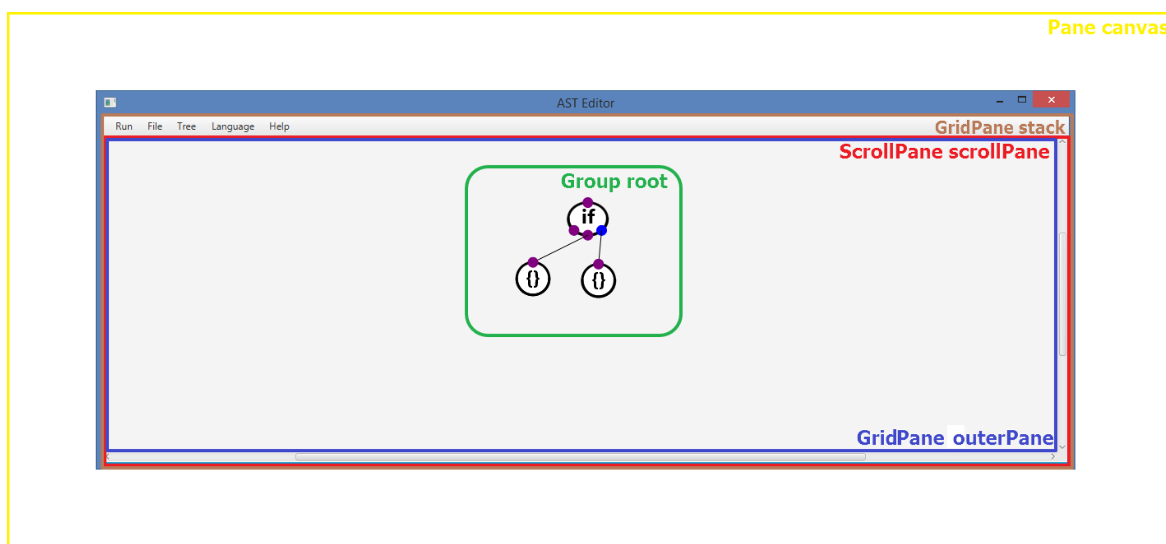
8.2.1 Třída ScrollView



Obr 8.2.1.1: Třída ScrollView

Třída ScrollView je centrální třídou tohoto balíčku. Obsahuje pět do sebe vnořených ploch (objektů Pane). Plocha na nejnižší úrovni – Pane *canvas*, tedy „plátno“ – představuje samotnou pracovní plochu. Její velikost přesahuje velikost okna editoru a navíc je měnitelná. Plátno je vnořeno do objektu třídy ScrollPane, který na něj nabízí částečný náhled (viewport). V plátnu je dále vnořen objekt třídy Group, což je volné seskupení grafických objektů.

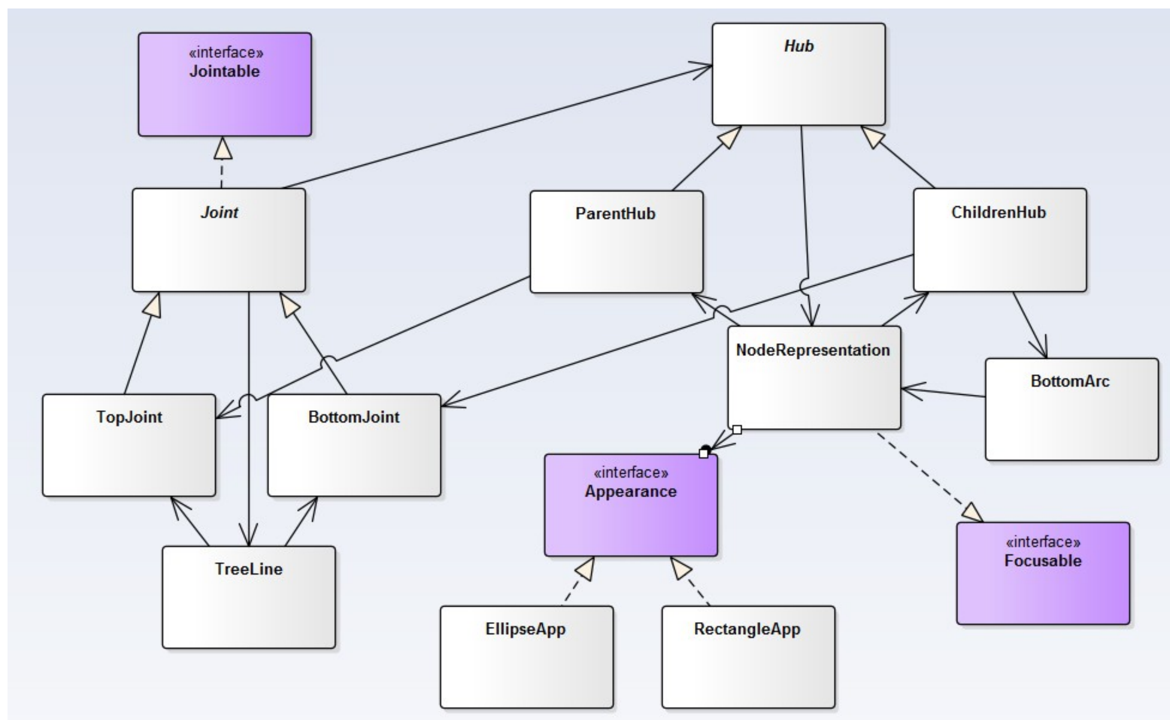
Při představení JavaFX bylo zmíněno, že tento framework staví z grafických prvků vlastní strom. Objekt stack třídy GridPane je v tomto případě jeho kořenem. Má dva potomky: objekt menu třídy MenuBar a instanci třídy GridPane outerPane. Níže ve stromu je pak právě seskupení Group. Do něj Editor přidává jednotlivé uzly JavaFX stromu (v našem případě se jedná o instance NodeRepresentation a TreeLine, příležitostně dialogy). AST strom tedy neodpovídá JavaFX stromu. Například na obrázku 8.2.1.2 je uzel „{}“ potomkem uzlu „if“, v rámci grafické reprezentace jsou to ale uzly na stejné úrovni stromu.



Obr 8.2.1.2: Kompozice třídy ScrollView

8.2.2 Basic

Tento balíček obsahuje všechny třídy tvořící NodeRepresentation (a navíc tovární třídu pro NodeRepresentation). Konkrétní kompozice bude popsána v příslušné části. Nachází se zde taky rozhraní Focusable nutné pro získávání zaměření (focus) v editoru.



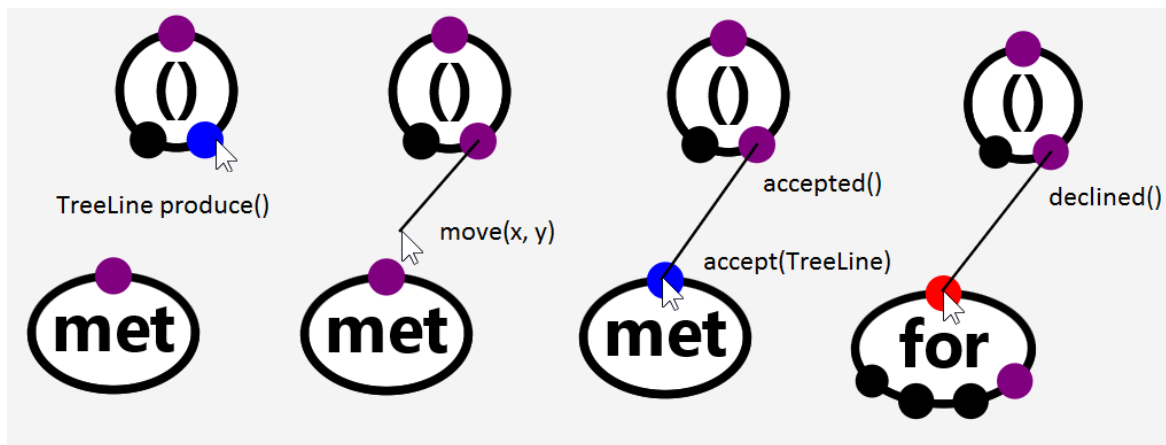
Obr 8.2.2.1: Diagram tříd v balíčku basic

8.2.3 Rozhraní Jointable a třídy Joint

Spojování uzlů grafu (přes objekty TreeLine) se uskutečňuje pomocí tzv. spojů.
Spoj (třída Joint) je malý kruh, který na sebe dokáže navázat hranu. Musí splňovat rozhraní Jointable, obsahující následující metody:

```
public interface Jointable {
    public void declined();
    public void accepted();
    public void accept(TreeLine t);
    public TreeLine produce();
    public void move(double x, double y);
    public boolean hasLine();
    public boolean shouldAccept(Joint j);
}
```

Po zahájení tahu myší se zmáčknutým tlačítkem (drag) ze spoje je vyprodukována hrana spojena s daným objektem Joint. To umožňuje metoda ***produce***. Tímto tahem můžeme hranu připojit na jiný spoj (drop) – implementováno pomocí metody ***accept***.



Obr 8.2.3.1: Volání metod rozhraní Jointable v průběhu práce se spojem

Metody ***declined*** a ***accepted*** jsou volány na spoji který hranu vyprodukoval po rozhodnutí zda ji přijmou nebo ne. Rozhodnutí se děje mimo jiné na základě pravidel, které budou popsány v části 8.4.1 a činí jej metoda ***shouldAccept***. Dalším kritériem pro rozhodnutí je, zda je spoj již obsazen jinou hranou (metoda ***hasLine***).

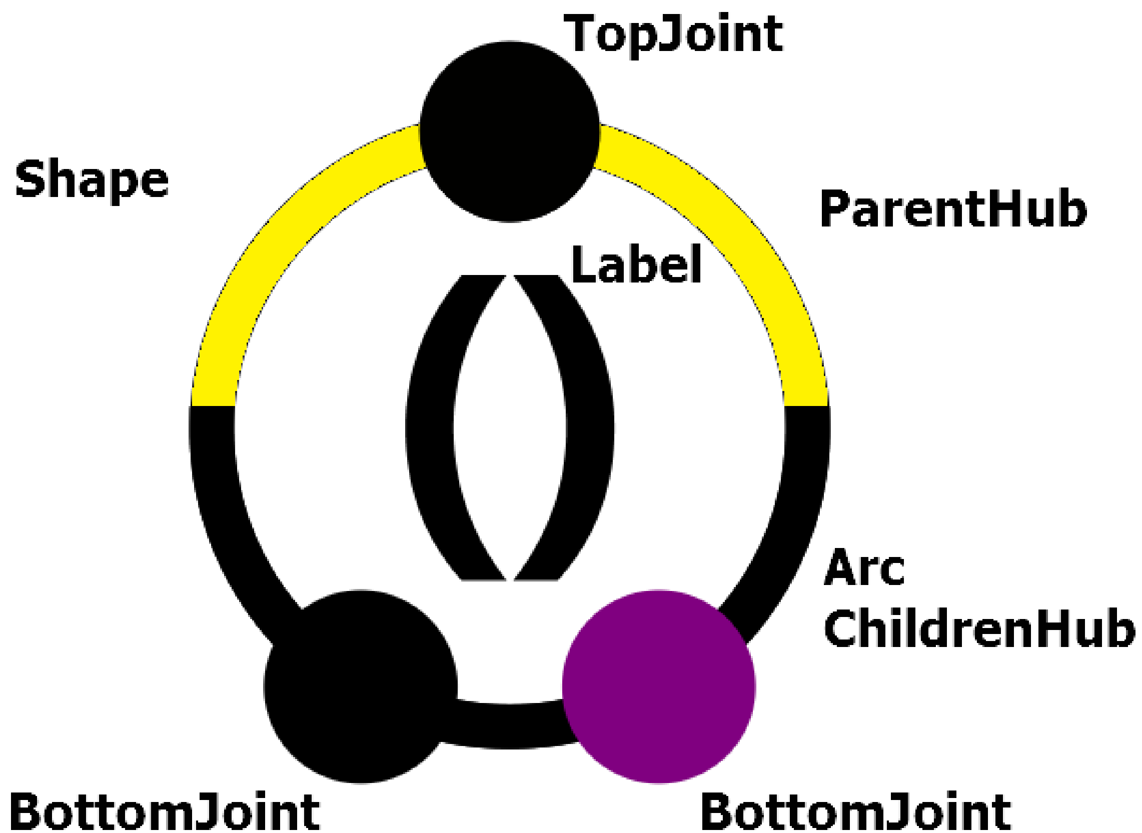
Většinu funkcionality implementuje již abstraktní třída Joint. Z ní dědí dvě třídy pro spoj na horní a spodní části uzlu. Platí přitom, že spojit se můžou jenom rozdílné spoje (BottomJoint s TopJoint).

8.2.4 Hub, ParentHub a ChildrenHub

Spoje nejsou vlastněny přímo uzlem, nýbrž instancemi třídy Hub. Ta je zapouzdřuje a stará se o jejich pohyb při přesouvaní uzlu, vykreslování a reakci na události. ChildrenHub ještě vlastní speciální objekt Arc, což je půlkružnice reagující na události spojené s působením myši. Některé uzly totiž mají dynamické přidávání spojů, je proto potřeba poslouchat i události na obvodu kružnice, ne jenom na spojích samotných.

8.2.5 NodeRepresentation

Třída pro grafickou reprezentaci uzlu AST. Struktura tříd byla naznačena v části 8.2.2, zde se podíváme na grafickou kompozici:



Obr 8.2.5.1: Uzel a jeho kompozice

NodeRepresentation je potomkem třídy Group, což je třída JavaFX sloužící pro seskupení více grafických prvků, nemá tedy sám o sobě žádný vizuál. Ten mu dodává instance další JavaFX třídy: Shape, lépe řečeno její potomek. V normálním stavu je to elipsa (kružnice), ve sbalené formě obdélník (čtverec).

Horní a dolní část jsou (v normálním stavu) překryty ChildrenHubem a ParentHubem popsanými výše. Kolem středu se nachází Label, popisek uzlu. Ten je u většiny uzlů konstantní, u některých jej lze měnit.

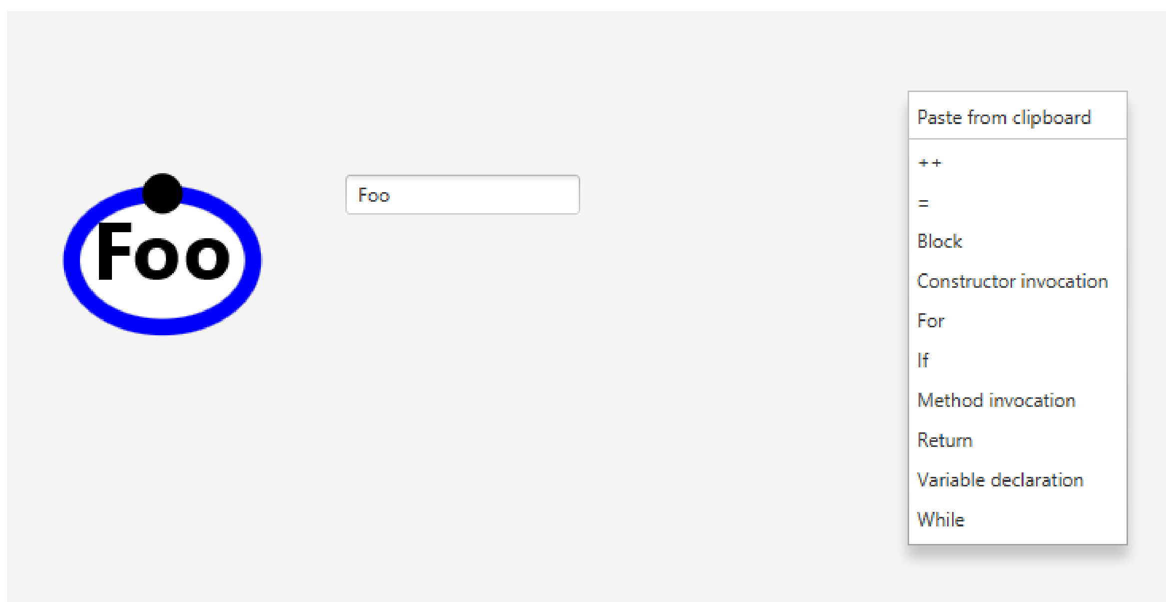
NodeRepresentation dále vlastní objekt implementující rozhraní MNode z balíčku model. Z toho získává jak Label, tak pravidla pro připojování spojů a případnou dynamickou tvorbu dalších spojů. Zatímco NodeRepresentation je společnou formou pro všechny druhy uzlů, jejich konkrétní chování je determinováno právě tímto objektem.
NodeRepresentation také implementuje rozhraní Visitable a Inspectable z balíčku controller.tree, které umožňují procházení stromovou strukturou pomocí metod ***acceptVisitor(Visitor v)*** *a* ***acceptInspector(Inspector i)*** v duchu podobném klasickému návrhovému vzoru Visitor. Více viz 8.3.2 a 8.3.3.

8.2.6 Dialogy

V balíčku view.popups se nacházejí rozhraní a třídy pro realizaci vyskakovacích dialogů pro komunikaci s uživatelem. Editor používá dva vlastní dialogy. Implementují rozhraní Popupable předepisující pouze dvě metody:

```
public interface Popupable {
	public void dispose();
	public void reveal(Node anchor, double screenX, double screenY);
}
```

Pomocí těchto metod jsou dialogy přístupné pro Controller který řídí jejich životní cyklus. Prvním z dialogů (ač je možná termín dialog nepříslušný) je DropMenu nabízející vytvoření nového uzlu. Rozšiřuje třídu ContextMenu frameworku JavaFX. Druhý typ dialogu se používá pro zadávání textového vstupu od uživatele při editaci uzlu. Graficky připomíná obyčejný TextField, ve skutečnosti se jedná o instanci třídy Stage.



Obr 8.2.6.1: Dva druhy vlastních dialogů editoru

8.3 Controller

Třídy v tomto balíčku provádí většinu řízení editoru. Dále provádí překlady ze stromové reprezentace do zdrojového kódu a naopak.

8.3.1 Překlad do zdrojového kódu

Překlad se provádí prohledáváním stromu do hloubky algoritmem podobným návrhovému vzoru *Visitor*, a to ve dvou fázích. V první fázi se provádí inspekce – kontroluje se, zda jsou splněny podmínky pro překlad. V druhé fázi se opět prohledává strom do hloubky a tentokrát už dochází k vytvoření stromu pomocí konstruktů Eclipse JDT.

8.3.2 Inspekce stromu

NodeRepresentation implenetuje rozhraní Inspectable:

```
public interface Inspectable {
        public void acceptInspector(Inspector ins);
        public String evaluate();
}
```

Implementace metody ***acceptInspector*** vypadá následovně:

```
public String acceptInspector(Inspector ir) {
        NodeRepresentation prev = ir.getCurrent();
        ir.setCurrent(this);
        String status = this.astNode.acceptInspector(ir);
        if (status != null && !status.equals("")) {...}
        ir.setCurrent(prev);
        return null;
}
```

Ve skutečnosti tedy v samotném NodeRepresentation k ničemu důležitému nedochází. Je to proto, že on sám o sobě je pouhou grafickou reprezentací, která umožňuje spojení uzlů, ale nenese žádné informace o významu. Ty jsou uloženy v objektu implementujícím rozhraní MNode z balíčku model – zde je to objekt *this.astNode*. Jak je vidět, na něm je taktéž volána metoda ***acceptInspector(Inspector)****,* má totiž vlastní implementaci rozhraní Inspectable. Implementace této metody v NodeRepresentation tak hlavně zajišťuje, že v každou chvíli bude možno přistoupit k aktuálnímu uzlu ve stromu, a hlavně k jeho potomkům. NodeRepresentation je totiž tím objektem, který spojování zajišťuje pomocí spojů. Zde se taky nachází část logiky pro reakci na neúspěšnou inspekci.

Další metoda tohoto rozhraní, evaluate, umožňuje definovat další podmínky pro správnost uzlu, nicméně v této třídě není použita.

Inspector sám o sobě je taky rozhraní:

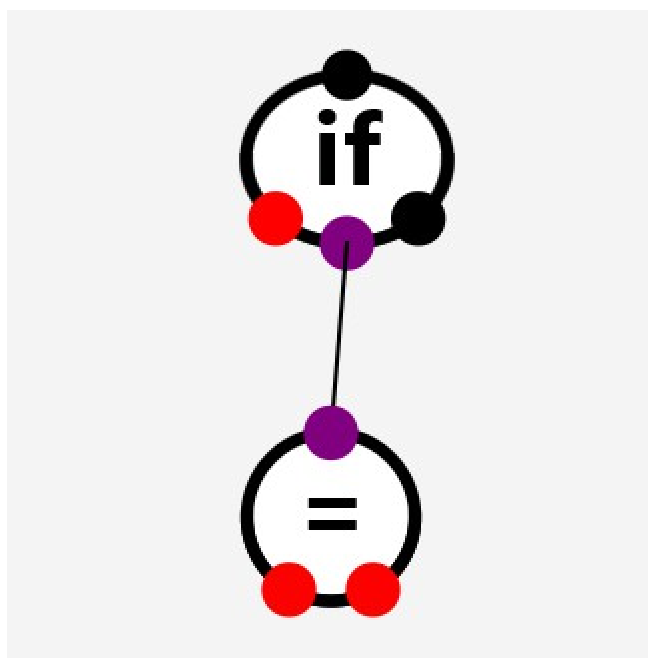
```
public interface Inspector {
        void setCurrent(NodeRepresentation nodeRepresentation);
        void  inspectChild(JointRules jr);
        void inspectChildren(JointRules jr);
        NodeRepresentation getCurrent();
}
```

A jeho implementátorem je třída TreeInspector. Nejzajímavější je metoda ***inspectChild***:

```
public void inspectChild(JointRules jr) {
        NodeRepresentation next =this.currentNR.findJointedChild(jr.name);
                if (next == null) {
                        if (jr.isMandatory) {
                                currentNR.warnJoint(jr.name);
                                 Controller.getInstance().setInspection(false);
                  }
```

```
                } else {
                        currentNR.okayJoint(jr.name);
                        next.acceptInspector(this);
                }
}
```

JointRules je objekt, který nese mimo jiné informace o tom, zda je daný podstrom pro uzel povinný. Taktéž obsahuje pojmenování podstromu/spoje. Pomocí něj si TreeInspector nechá vyhledat potomka NodeRepresentation připojeného na dané místo (zde je vidět proč musí mít instanci na aktuální uzel). Pokud zůstává daný spoj neobsazen, a je povinen (isMandatory), TreeInspector nechá daný spoj upozornit a zároveň nastaví ve třídě Controller ukazatel inspekce na false. Pokud je po inspekci ukazatel nastaven na false, Controller nepokračuje v překladu. V opačném případě postupuje do potomka.



Obr 8.3.2.1: Spoje po neúspěšné inspekci

Metoda ***inspectChildren*** vykonává stejnou funkcionalitu, ale na spojích které jsou rozšířitelné. Může jich tak být větší počet, ale všechny jsou identifikované jediným jménem. Proto se s nimi pracuje jako s kolekcí.

8.3.3 Budování stromu

Po úspěšné inspekci přistupuje Controller k dalšímu průchodu stromem, tentokrát již dochází k překladu do stromové struktury frameworku Eclipse JDT. To vyžaduje od NodeRepresentation implementaci rozhraní *Visitable*:

```
public interface Visitable {
        public ASTNode acceptVisitor(Visitor tr);
}
```

Velice podobné rozhraní Inspectable. Zatímco ale to mělo za úkol pouze upozorňovat spoje a nastavit ukazatel, metoda ***acceptVisitor*** již vyprodukuje instanci třídy ASTNode. To je třída Eclipse JDT která reprezentuje uzel se všemi jeho potomky (dá se na něj tedy dívat jako na podstrom). Pokud je tedy NodeRepresentation navštíven objektem Visitor, výsledkem je jeho překlad do ASTNode. Implementace je opět téměř totožná s implementací metody ***acceptInspector***

```
public ASTNode acceptVisitor(Visitor tr) {
        NodeRepresentation prev = tr.getCurrent();
        tr.setCurrent(this);
        ASTNode ret = this.astNode.acceptVisitor(tr);
        tr.setCurrent(prev);
        return ret;
}
```

Rozhraní Visitor vypadá následovně:

```
public interface Visitor {
        public void setCurrent(NodeRepresentation nodeRepresentation);
        public ASTNode getChild(String name);
        public List<ASTNode> getChildren(String name);
        public NodeRepresentation getCurrent();
        public void registerConstructor(MethodDeclaration md);
        public List<MethodDeclaration> getConstructors();
}
```

Toto rozhraní implementuje třída TreeBuilder. Klíčovou metodou, obdobou ***inspectChild*** z TreeInspectora, je metoda ***getChild***. Na rozdíl od ní má ale jednodušší logiku, protože nemusí nic signalizovat, pouze zprostředkuje přechod na potomka, existuje-li.

```
public ASTNode getChild(String name) {
        NodeRepresentation next = this.currentNR.findJointedChild(name);
        if (next == null) {
                return null;
        }
        return next.acceptVisitor(this);
}
```

Jak je vidět, tento algoritmus se liší od návrhového vzoru *Visitor.* U něj by totiž Visitor musel mít speciální metody pro každý element, který by navštívil. To by vyžadovalo při přidání nového typu uzlu editaci rozhraní i třídy. Navíc by část logiky pracující z Eclipse JDT byla vykonávaná v kontroléru, čemuž jsem se chtěl vyhnout. Proto Visitor pouze zprostředkuje průchod stromem, zatímco „sémantika“ se vykonává v třídách modelu.

Metody ***registerConstructor*** a ***getConstructors*** jsou zde proto, aby objekt, který spouští překlad, mohl při překladu třídy přejmenovat deklarace konstruktorů dle

obsahující třídy. S deklaracemi konstruktorů je jinak zacházeno jako s metodami s názvem ***init***.

8.3.4 NodeService

V podobném duchu, jako v předchozím případě, jsem se snažil, aby kontrolér nebyl závislý na konkrétní struktuře tříd v modelu ani jinde. Další moment, ve kterém by to mohlo být problematické, je přidávání nového uzlu. Jak bylo ukázáno výše, lze toho dosáhnout pomocí DropMenu dvěma způsoby – DropMenu se všemi existujícími druhy uzlu, nebo s možnými potomky daného uzlu. V obou případech musí mít Controller přehled o všech existujících možnostech – tedy všech potomcích implementujících rozhraní MNode. Pokud by ale neměl mít nějaký seznam, který by vyžadoval od programátora registraci všech možných druhů uzlů, bylo potřeba, aby byla tato struktura rozřešena při běhu. K tomu by se intuitivně nabízela technologie javy - reflexe. Ta ale bohužel neumožňuje zjištění všech podtypů daného typu. Naštěstí existují řešení třetích stran, pomocí kterých to je dosažitelné. Zvolil jsem knihovnu reflections [16]. Je to nástroj umožňující běhovou analýzu metadat. Pracuje tak, že oskenuje classpath, naindexuje metadata a pak umožňuje se nad nimi dotazovat při běhu. Mezi její možnosti patří získat podtypy daného typu v daném adresáři. Použití je velice intuitivní a jednoduché:

```
 Reflections reflections = new Reflections("my.project.prefix");
Set<Class<? extends SomeType>> subTypes =
        reflections.getSubTypesOf(SomeType.class);
```

Toho využívá třída NodeService. Nabízí metodu ***getSubclasses*** s argumentem třídním objektem, jehož potomky chceme vrátit. Vrací pouze neabstraktní třídy a případně i samotnou dotazovanou třídu.

```
public static Set<Class<? extends MNode>> getSubclasses(
                Class<? extends MNode> nodeType) {
        Reflections reflections = new Reflections(path);
        Set<?> allClasses = reflections.getSubTypesOf(nodeType);
        Set<Class<? extends MNode>> classes = (Set<Class<? extends MNode>>)
                        allClasses;
        classes.add(nodeType);
        return (Set<Class<? extends MNode>>) classes.stream()
                        .filter(n -> !Modifier.isAbstract(n.getModifiers()))
                        .collect(Collectors.toSet());
}
```

To ale není vše, k čemu tato třída slouží. Zajišťuje taky lokalizaci názvů uzlů. Ten je definován v objektu implementujícím rozhraní MNode. Třída NodeService je v určitou chvíli pobídnuta, aby pomocí reflexe získala lokalizované názvy všech tříd implementujících toto rozhraní a následně aktualizuje lokalizační balíčky.

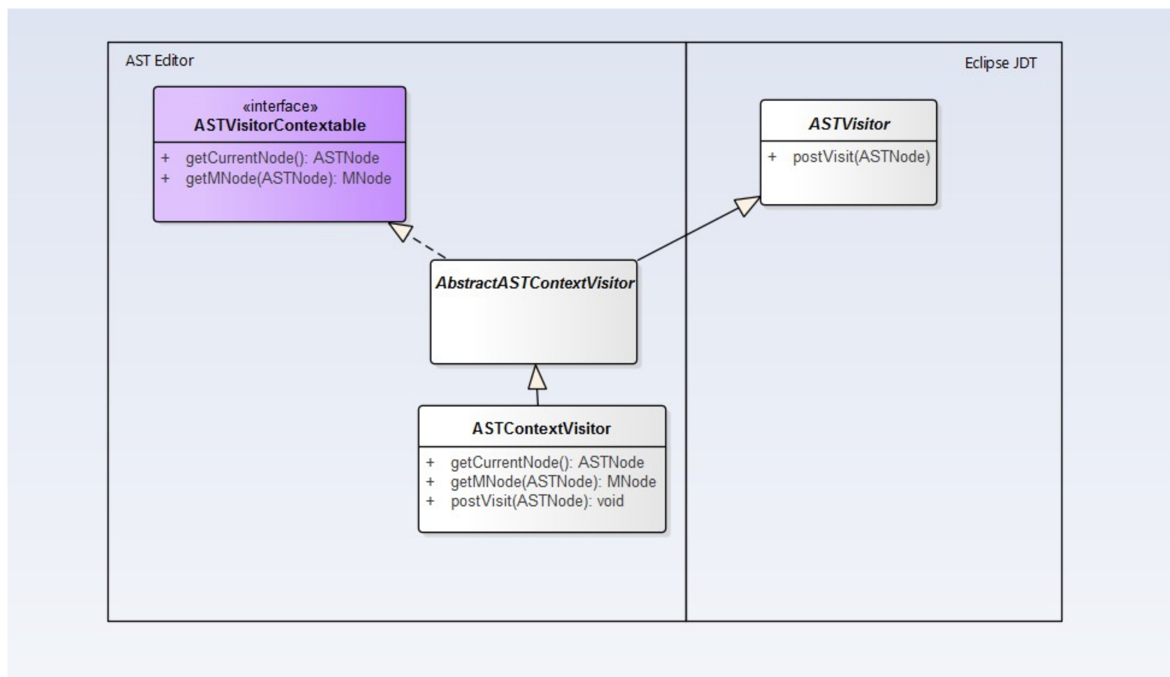
Dále je tato třída instrumentální při překladu ze zdrojového kódu do AST a tento proces si přiblížíme v následující části.

8.3.5 Překlad ze zdrojového kódu

Zde popíšu proces opačný k překladu do zdrojového kódu, tedy proces, kdy je na vstupu zdrojový kód a editor z něj vystaví grafickou reprezentaci AST.

Pro čtení zdrojového kódu editor využívá opět frameworku Eclipse JDT. Ten umí velice jednoduše získat vlastní AST reprezentaci daného kódu. Důležité pro tento proces je, že uzly tohoto stromu jsou pak implementátory návrhového vzoru Visitor. Lze je díky tomu navštívit vlastním Visitorem. Ten musí být potomkem abstraktní třídy ASTVisitor, která předepisuje dvojici metod, ***visit*** a **end*Visit*,** pro každý možný druh uzlu (tedy klasický vzor Visitor). Dále ale taky předepisuje dvojici metod ***preVisit*** a ***postVisit*** společnou pro všechny typy uzlů. Právě tento druhý přístup jsem zvolil pro implementaci logiky překladu. Při volbě první možnosti by totiž bylo nutné implementovat vlastní metodu pro každý typ uzlu, což je přesně to, čemu jsem se chtěl vyhnout. Při volbě druhé možnosti je logika překladu dodána z příslušných tříd v balíčku model**.**

Konkrétní Visitor, objekt třídy ASTContextVisitor, navíc implementuje rozhraní ASTVisitorContextable. Toto rozhraní, podobně jako rozhraní Inspector a Visitor, zabezpečuje, že pro uživatele budou v každou chvílí přístupné instance aktuálně navštíveného uzlu ve stromu. Druhá metoda tohoto rozhraní, ***getMNode***, zase zpřístupňuje dříve přeložené uzly.

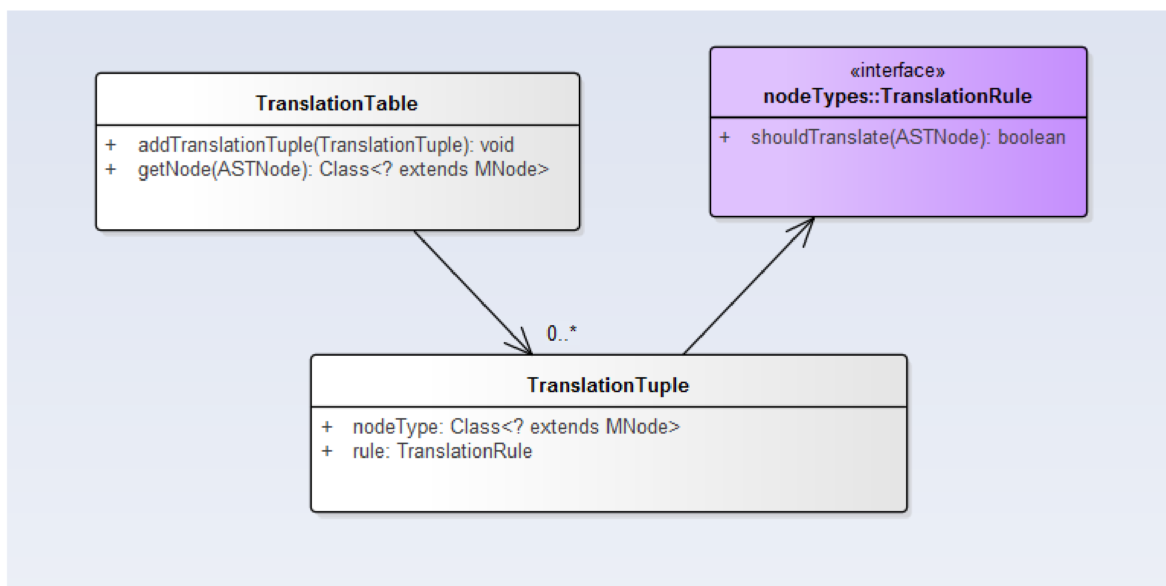


Obr 8.3.5.1: Visitor pro překlad ze zdrojového kódu

Jak vlastně tento ASTContextVisitor pracuje? Jeho úkolem je přeložit strom z konstruktů Eclipse JDT frameworku (tzn. strom skládající se z uzlů typu ASTNode) do stromu konstruktů editoru (strom skládající se z uzlů typu MNode). K tomu potřebuje znát pravidla, určující na jaký MNode má přeložit daný ASTNode. Tyto pravidla jsou opět internalizovány právě v potomcích MNode, a přístupně jsou skrze třídu NodeService. NodeService při inicializaci posbírá všechna tyto pravidla, podobně jako například lokalizované názvy uzlů, a ASTContextVisitor se ně kdykoliv může dotázat. Po vytvoření odpovídajícího potomka MNode tomuto předá odpovědnost za správný překlad a uloží jej

do mapy vytvořených uzlů. V té je objekt ASTNode použit jako klíč a objekt MNode jako hodnota. Právě z této mapy jsou dříve přeložené uzly přístupné pomocí metody ***getMNode*** dalším uzlům (pro připojení jako potomka).

8.3.6 Třídy TranslationTable, TranslationTuple a TranslationRule



Obr 8.3.6.1: Trojice tříd pro překlad z ASTNode

Tato trojice tříd implementuje aparát pro užívání pravidel překladu z ASTNode do MNode. Rozhraní TranslationRule má jedinou metodu, která určuje, zda daná podtřída ASTNode je přeložitelná na asociovanou podtřídu MNode. Implementace tohoto rozhraní jsou definované v jednotlivých podtřídách MNode a jsou shromážděny třídou NodeService při inicializaci. Asociace ASTNode-MNode je realizovaná v objektu typu TranslationTuple. Metoda ***getNode*** v TranslationTable pak jednoduše iteruje přes všechny objekty TranslationTuple, a pokud nalezne ten, jenž odpovídá danému objektu ASTNode, vrátí asociovaný třídní objekt MNode.

```
public Class<? extends MNode> getNode(ASTNode astn){
    for(TranslationTuple t:table){
        if(t.rule.shouldTranslate(astn)){
            return t.nodeType;
        }
    }
    return null;
}
```

8.3.7 Třída ConnectionCommand

S pomocí tříd představených v předešlé části je kontrolér schopen vytvořit všechny uzly AST reprezentované pomocí podtříd MNode, zatím je ale není schopen spojit do stromu. Je to proto, že ke spojení dochází jenom na úrovni grafické reprezentace a rozhraní MNode spojení nijak neřeší. Pro tento účel existuje třída ConnectionCommand. Tato třída určuje

kód spoje, jenž má byt použit pro napojení, a instanci potomka, který má být na daný spoj připojen. Metoda zodpovědná za překlad ASTNode do MNode, kromě vykonání vlastní překladové logiky nad uzlem, také vrátí pole instancí třídy ConnectionCommand, které plně determinuje napojení všech potomků. Pomocí těchto informací je pak třída Controller schopna vygenerovat příslušné hrany grafu.

8.4 Model

Funkcionalita v balíčku model je v podstatě prostředníkem mezi doménou grafického JavaFX editoru a světem Eclipse JDT frameworku. Ve skutečnosti je to jediné místo, kde se používají konstrukty z tohoto frameworku, můžeme se na něj dívat jako na „překladač" z jazyka JavaFX do jazyka Eclipse JDT a naopak. Zatímco NodeRepresentation je grafickou reprezentací společnou pro všechny uzly, význam mu dodávají třídy z toho balíčku.

Tato práce obsahuje jenom podmnožinu všech možných konstruktů jazyka Java, a pokud by bylo potřeba přidat další, jediné, co je potřeba udělat, je přidat další třídu vyhovující určitému kontraktu (rozhraní MNode) do příslušného balíčku, která zpřístupní grafickému editoru příslušné nástroje Eclipse JDT. To je umožněno třídou NodeService představenou výše.

8.4.1 JointRules

Nejdříve se podíváme na objekty, které definují vlastnosti spojů. Jedná se o instance třídy JointRules, která vypadá následovně:

```
public class JointRules {
        public String name;
        public boolean isMandatory;
        public Class<? extends MNode> nodeType;
        …}
```

Jak je vidět, třída je velice jednoduchá. Pro každý spoj definuje tři vlastnosti:

- jméno – používá se pro identifikaci daného podstromu;
- povinnost – určuje, zda spoj musí být při překladu obsazen, nebo může být vynechán;
- třídu potomka – tedy jaký uzel je možné na daný joint připojit.

8.4.2 Rozhraní MNode

Toto velice důležité rozhraní odhaluje zbytku aplikace „sémantiku" uzlů, proto se na něj podíváme podrobněji.

```
public interface MNode extends Visitable, Inspectable{

        public final int NON_EXTENDABLE=-1;
        public final int FULLY_EXTENDABLE=0;
        public final int END_EXTENDABLE=1;

        String getLabel();
        int isExtendable();
        JointRules getExtendableRule();
        List<JointRules> getJointRules();
        TranslationRule getTranslationRule();
        List<ConnectionCommand<ASTNode>> translate(AbstractASTContextVisitor
                                                                        av);
        String getEN();
        String getCZ();
        String evaluate();
        MNode cloneNode();
        @Override
        default public String acceptInspector(Inspector ins) {
                for(JointRules jr:getJointRules()){
                        ins.inspectChild(jr);
                }
                if(isExtendable() == MNode.END_EXTENDABLE || isExtendable() ==
                                        MNode.FULLY_EXTENDABLE){
                        ins.inspectChildren(getExtendableRule());
                        }
        return evaluate();
}
```

První důležitá věc je, že rozhraní rozšiřuje dvojici rozhraní Visitable a Inspectable. Ty jsou důležité pro procházení stromu, jak bylo demonstrováno dříve. Rozhraní Inspectable přímo implementuje, protože inspekce má společnou část logiky pro všechny implementace rozhraní MNode. Defaultní implementace metod je možná od Javy 8. Jediná metoda rozhraní Inspectable je ***acceptInspector***. Tato metoda nejdříve přesměrovává inspektora na příslušné potomky. Pokud má uzel dynamicky rozšířitelný spoj, použije speciální metodu. Po vyřízení potomků ještě zavolá metodu ***evaluate,*** kde můžou jednotliví implementátoři definovat specifickou logiku pro kontrolu validity uzlu (typicky platnost identifikátoru). Zda má uzel dynamicky rozšířitelné spoj je signalizováno metodou ***isExtendable***, jehož návratovou hodnotou je int. Přípustné hodnoty jsou definované jako konstanty rozhraní a určují stupeň možnosti rozšíření. Jsou tři:

- *NON_EXTENDABLE=-1* – žádný spoj není rozšířitelný, uzel má statický počet potomků (například uzel Přiřazení má levou a pravou stranu);
- *FULLY_EXTENDABLE=0;* - uzel má kdekoliv rozšířitelný spoj (například uzel Blok může mít libovolný počet potomků);
- *END_EXTENDABLE=1;* - uzel má rozšířitelný spoj na poslední pozici (například Volání metody).

Dalším členem rozhraní jsou metody zpřístupňující pravidla pro chování spojů – ve formě objektů JointRules. Jsou dvě, první ***getJointRules*** vrací kolekci instancí pro statické spoje. Má-li tedy například uzel Přiřazení dva potomky typu Výraz, tato metoda vrátí dva objekty JointRules s příslušným nastavením. Druhá metoda, ***getExtendableRule*** definuje pravidla pro rozšířitelný spoj. Měla by být vždy volána až po ověření pomocí metody isExtendable. Dalším členem rozhraní je metoda ***getLabel***, která dodává grafický popisek uzlu. Ten může být u různých uzlů dynamický nebo statický. Metoda ***getTranslationRule*** vrací objekt implementující rozhraní TranslationRule pro určení toho, jaký typ uzlu z frameworku Eclipse JDT má byt překládán právě na danou implementaci rozhraní MNode (viz část *8.3.6).* Konkrétní překlad provádí metoda ***translate*,** jejíž návratovou hodnotou je list objektů ConnectionCommand pro připojení potomků (viz 8.3.7) Metody ***getEN*** a ***getCZ*** vrací lokalizované názvy uzlů. Metoda ***cloneNode*** provádí hlubokou kopii.

8.4.3 Rozhraní MEditable

Ty uzly, které mají dynamicky editovatelný popisek, musí implementovat rozhraní MEditable:

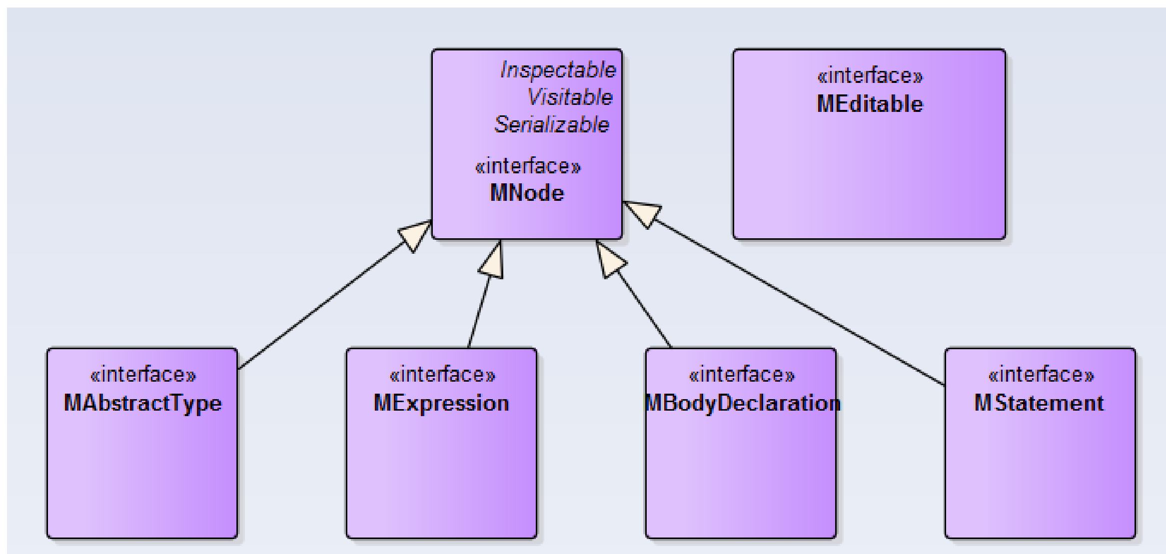
```
public interface MEditable {
        public void setValue(String s);
}
```

8.4.4 Abstraktní třída MAbstractNode

Protože každý potomek MNode musí implementovat metody ***getJointRules*** a ***getExtendableRule*** stejným způsobem, patří tato implementace do předka. Nelze jí ale definovat v rozhraních, protože pracuje s instančním proměnnými. Proto všechny uzly dědí od společné třídy MAbstractNode. Ta splňuje část kontraktu, jeho potomci musí jenom těmto instančním proměnným přiřadit příslušné objekty.

```
public abstract class MAbstractNode implements Inspectable{
        protected List<JointRules> rules = new ArrayList<JointRules>();
        protected JointRules extendableRule;
        …
}
```

8.4.5 Struktura rozhraní v balíčku nodeTypes

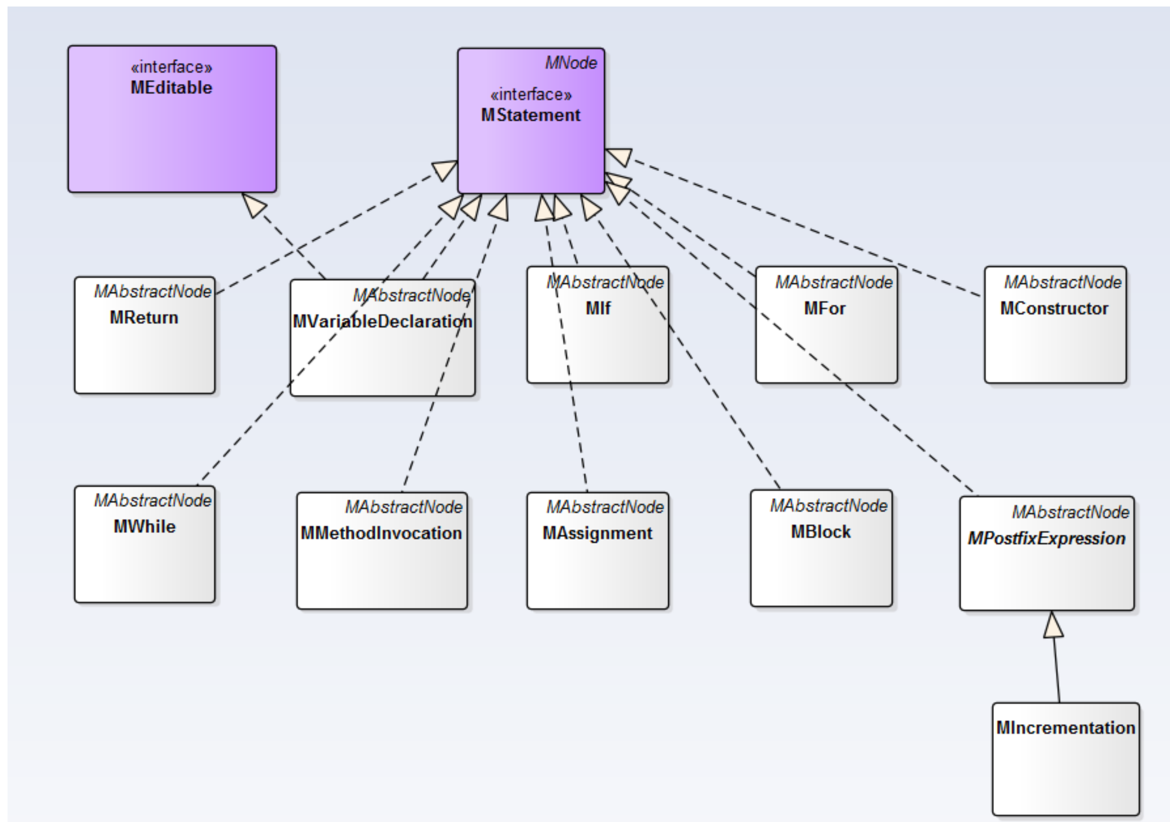


Obr 8.4.5.1: Diagram některých rozhraní v balíčku nodeTypes

Rozhraní MNode rozšiřují čtyři další rozhraní. Tento model zhruba odpovídá modelu Eclipse JDT, jak byl představen v části 6 (první písmeno všech tříd a rozhraní v tomto balíčku slouží právě k rozlišení od tříd Eclipse JDT). Rozhraní jsou prázdná a slouží pouze k deklaraci struktury.

8.4.6 Statement

Struktura tříd implementující rozhraní Statement:



Obr 8.4.6.1: Diagram tříd implementujících rozhraní MStatement

8.4.7 Třída MIf

Modeluje chování příkazu If.

```
public class MIf extends MAbstractNode implements MStatement {

    private static String IF_CODE = "if";
    private static String THEN_CODE = "then";
    private static String ELSE_CODE = "else";

    public MIf() {
        rules.add(new JointRules("if", true, MExpression.class));
        rules.add(new JointRules("then", true, MStatement.class));
        rules.add(new JointRules("else", false, MStatement.class));
    }

    public String getLabel() {
        return "if";}

    public int isExtendable() {
        return MNode.NON_EXTENDABLE;
    }
```

```
public ASTNode acceptVisitor(Visitor tr) {
        IfStatement is = ASTService.getIf();
        is.setExpression(ASTService.getExpression(tr.getChild(IF_CODE)));
        is.setThenStatement(ASTService.getStatement(tr.getChild(THEN_CODE)));
        is.setElseStatement(ASTService.getStatement(tr.getChild(ELSE_CODE)));
        return is;
}
public TranslationRule getTranslationRule() {
        return new TranslationRule() {
                @Override
                public boolean shouldTranslate(ASTNode a) {
                        return (a.getNodeType() == ASTNode.IF_STATEMENT);}
        };
}

public List<ConnectionCommand<ASTNode>> translate(AbstractASTContextVisitor
                                                                        v) {
        List<ConnectionCommand<ASTNode>> l= new ArrayList<>();
        IfStatement anode = ((IfStatement)v.getCurrentNode());
        l.add(new ConnectionCommand<ASTNode>(this.IF_CODE,
                                (Expression)anode.getExpression()));
        l.add(new ConnectionCommand<ASTNode>(this.THEN_CODE,
                                (Statement)anode.getThenStatement()));
        l.add(new ConnectionCommand<ASTNode>(this.ELSE_CODE,
                                (Statement)anode.getElseStatement()));
        return l;
}
public String evaluate() {
        return null;
}
}
```

Rozšiřuje třídu MAbstractNode, takže zdědil implementaci metod ***getJointRules*** a ***getExtendableRule***. Pravidla mu tak stačí definovat v konstruktoru. Jsou tři, protože očekává tři potomky. Ty jsou pomocí statických konstant pojmenovány „if“, „then“ a „else“. V potomku „if“ je výraz definující větvící podmínku a je povinný. Potomek „then“ definuje větev při splnění větvící podmínky, a je opět povinen. Potomek „else“ již povinen není, protože to javovská syntaxe nevyžaduje. Obě očekávají potomka typu Statement.
Metoda ***isExtendable*** je implementována pomocí konstant rozhraní MNode, je zde implementována taky metoda ***getLabel***.

V implementaci metody ***acceptVisitor*** dochází k samotnému vytváření uzlu IfStatement z Eclipse JDT příslušnými metodami na ASTService. Poté se postupně nastaví potomci získaní pomocí objektu TreeBuilder – ten má referenci na aktuální objekt NodeRepresentation a potomka nalezne podle jména. Zde je tedy potřeba znát API Eclipse JDT a zde taky dochází k samotnému překladu. Vytvořený objekt je vracen a v rekurzívním stylu s ním může pracovat jeho předek v AST.

Metoda ***getTranslationRule*** vrací vlastní implementaci rozhraní TranslationRule

(ve formě anonymní třídy). Ta jednoduše říká, že tento typ uzlu odpovídá typu *ASTNode.IF_STATEMENT* frameworku Eclipse JDT. Metoda ***translate*** provádí překlad z objektu ASTNode do objektu MIf. Ten v tomto případě spočívá jenom v definici potomků, kteří mají být připojeni na dané pozice. Metoda ***evaluate*** nespecifikuje žádnou další podmínku inspekce, proto vrací null. V případě chyby by vracela chybový kód.

8.4.8 Třída MBlock

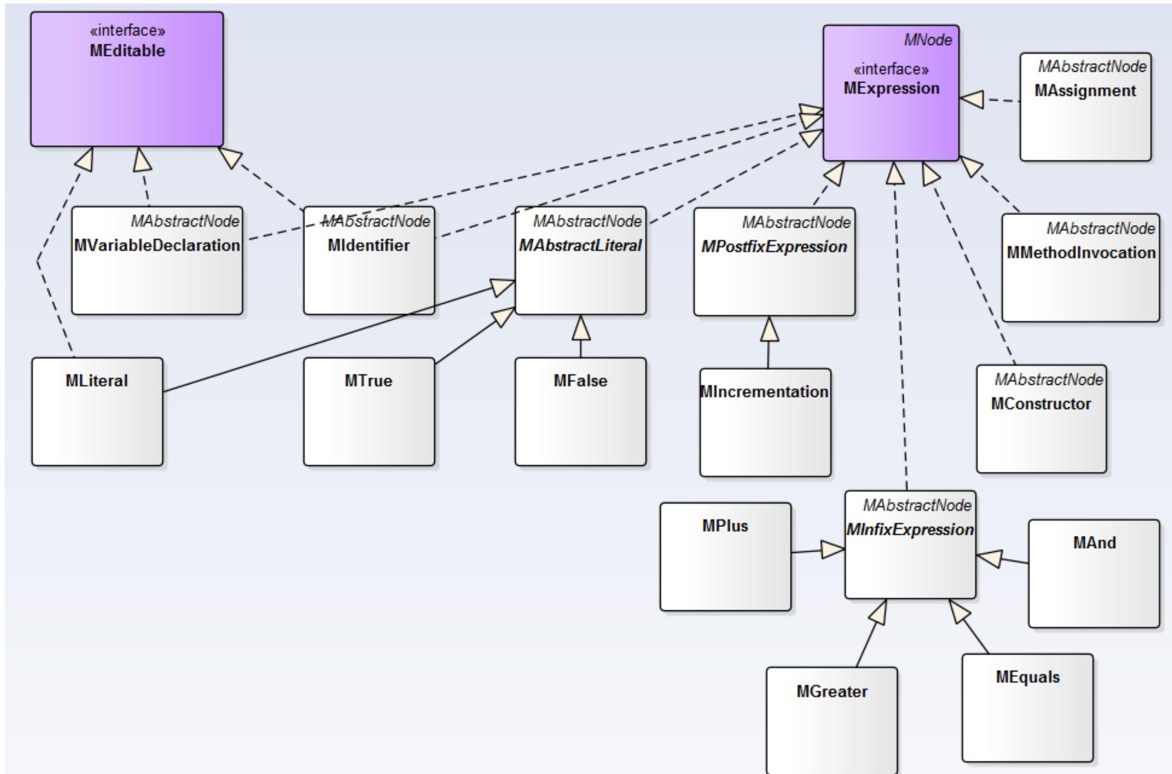
Blok je příkaz, který má libovolný počet potomků, na této třídě demonstrujeme práci s rozšířitelným spojem.

```
public class MBlock extends MAbstractNode implements MStatement {

    private static String STATEMENT_CODE = "stmnt";

    public MBlock() {
        extendableRule = new JointRules(STATEMENT_CODE, false,
                                MStatement.class);
    }

    …

    public int isExtendable() {
        return MNode.FULLY_EXTENDABLE;
    }

    public ASTNode acceptVisitor(Visitor tr) {
        Block bs = ASTService.getBlock();
        for (ASTNode an : tr.getChildren(STATEMENT_CODE)) {
            bs.statements().add(ASTService.getStatement(an));
        }
        return bs;
    }
}
```

Metoda ***isExtendable*** vrací konstantu ***MNode.FULLY_EXTENDABLE***. Pravidlo je definované v konstruktoru. Potomci jsou potom získání v kolekci a postupně zpracování v iteraci.

8.4.9 Expression

Struktura tříd implementující rozhraní Expression:



Obr 8.4.9.1: Diagram tříd implementujících rozhraní MExpression

8.4.10 Třída MLiteral

Reprezentuje javovský literál. Protože ten může nabývat různých hodnot, implementuje rozhraní MEditable.

```
public class MLiteral extends MAbstractLiteral MEditable {

    protected String text;

    ...

    public MLiteral(String val) {
        this.text=val;
    }

    public void setValue(String str){
        text=str;
    }

    public String getLabel() {
        return this.text;
    }
```

```
        ...
        public ASTNode acceptVisitor(Visitor tr) {
                Expression lit = ASTService.getLiteral(this.text);
                return lit;
        }
}
```

Má vlastní členskou proměnnou, kterou lze nastavit pomocí metody ***setValue*** z rozhraní MEditable. Právě pomocí ní k hodnotě přistupuje třída Controller při změně jména. Tato proměnná je pak vracena jako popisek uzlu metodou ***getValue***. Literál je přiřazen do reference na abstraktní třídu Expression. Tento uzel totiž reprezentuje všechny (část) literálů jazyka. Podíváme-li se na metodu ***getLiteral*** třídy ASTService... :

```
public static Expression getLiteral(String value) {
        Expression exp;
        try {
                Double.parseDouble(value);
                exp = ast.newNumberLiteral(value);
        } catch (NumberFormatException ex) {
                exp = ast.newStringLiteral();
                ((StringLiteral) exp).setLiteralValue(value);
        }
        return exp;
}
```

...vidíme, že je vytvořen buď literál typu double, lze-li hodnotu přetypovat, jinak String. Toto samozřejmě není vyčerpávající řešení, ale to ani není cílem této práce.

8.4.11 Abstraktní třída MInfixExpression a její potomci

Infix výraz je vlastně operátor mezi dvěma operandy. V této práci jsou implementovány čtyři a pro svůj společný charakter mají společného předka.

```
public abstract class MInfixExpression extends MAbstractNode implements
                                                        MExpression {

        private static String LEFT_CODE = "left";
        private static String RIGHT_CODE = "right";

        public MInfixExpression() {
                rules.add(new JointRules(LEFT_CODE, true, MExpression.class));
                rules.add(new JointRules(RIGHT_CODE, true, MExpression.class));
        }

        ...
```

8.4.11 Abstraktní třída MInfixExpression a její potomci


```
        public ASTNode acceptVisitor(Visitor tr) {
                InfixExpression ie = ASTService.getInfix();
                        ie.setLeftOperand(ASTService.getExpression(
                                                tr.getChild(LEFT_CODE)));
                        ie.setRightOperand(ASTService.getExpression(
                                                tr.getChild(RIGHT_CODE)));
        return ie;
        }
}
```

Každý potomek třídy bude mít dva statické spoje, které si nastaví pomocí metody předka.

```
public class MAnd extends MInfixExpression {

        public String getLabel() {
                return "&&";
        }

        public ASTNode acceptVisitor(Visitor tr) {
                InfixExpression ie = (InfixExpression) super.acceptVisitor(tr);
                ie.setOperator(InfixExpression.Operator.CONDITIONAL_AND);
                return ie;
        }
}
```

A sám již jenom určí typ operandu, kterým je. Eclipse jDT API tohle řeší pomocí statických konstant vnořené třídy.

8.4.12 Třída MMethodInvocation

Uzel pro volání metody kombinuje statické a dynamické spoje.

```
public class MMethodInvocation extends MAbstractNode implements
                                        MExpression, MStatement {

        private static String TARGET_CODE = "target";
        private static String NAME_CODE = "name";
        private static String ARGUMENTS_CODE = "args";

        public MMethodInvocation() {
                rules.add(new JointRules(TARGET_CODE, false, MIdentifier.class));
                rules.add(new JointRules(NAME_CODE, true, MIdentifier.class));
        }

...
```

```
		public int isExtendable() {
			return MNode.END_EXTENDABLE;
		}

		public JointRules getExtendableRule() {
			return new JointRules(ARGUMENTS_CODE, false,
							MExpression.class);
		}
		public ASTNode acceptVisitor(Visitor tr) {
			MethodInvocation mi = ASTService.getMethodInvocation();
		mi.setExpression((Name)ASTService.getExpression(
							tr.getChild(TARGET_CODE)));
		mi.setName((SimpleName)ASTService.getExpression(
							tr.getChild(NAME_CODE)));
			for (ASTNode an : tr.getChildren(ARGUMENTS_CODE)) {
				mi.arguments().add(ASTService.getExpression(an));
			}
		return mi;
		}
		...
		public List<ConnectionCommand<ASTNode>>
					translate(AbstractASTContextVisitor  v) {
			MethodInvocation anode = (MethodInvocation) v.getCurrentNode();
			List<ConnectionCommand<ASTNode>> l= new ArrayList<>();
			l.add(new ConnectionCommand<ASTNode>(this.TARGET_CODE,
							anode.getExpression()));
			l.add(new ConnectionCommand<ASTNode>(this.NAME_CODE,
							anode.getName()));
			for(Expression ex:(List<Expression>)anode.arguments()){
				l.add(new ConnectionCommand<ASTNode>(
						this.ARGUMENTS_CODE, ex));
				}
			return l;
		}
		...
	}
```

Má tři spoje, z toho jeden je dynamický rozšiřitelný. Pravidla pro statické spoje jsou definovány v konstruktoru, pro dynamický spoj v metodě ***getExtendableRule***. Metody ***acceptVisitor*** a ***translate*** kombinuje práci s kolekcí a práci se statickými jointy.

8.4.13 Třída MVariableDeclaration

Třída pro deklaraci proměnné. Používá se pro definici (nebo deklaraci) lokální proměnné v metodách, nebo v inicializační části příkazu if. Nepoužívá se k deklaraci členských proměnných tříd, k tomu slouží třída MFieldDeclaration.

```
public class MVariableDeclaration extends MAbstractNode implements
                                        MExpression, MStatement, MEditable {

        private String text;
        private static String TYPE_CODE = "type";
        private static String EXP_CODE = "expression";

        public MVariableDeclaration(){
                this.rules.add(new JointRules(TYPE_CODE, true,
                                                MAbstractType.class));
                this.rules.add(new JointRules(EXP_CODE, false, MExpression.class));
        }

        ...

        public ASTNode acceptVisitor(Visitor tr) {
                VariableDeclarationFragment vdf =
                                ASTService.getVariableDeclarationFragment();
                vdf.setName(ASTService.getName(this.getLabel()));
                vdf.setInitializer(ASTService.getExpression(tr.getChild(EXP_CODE)));
                VariableDeclarationExpression vde =
                                        ASTService.getVariableDeclaration(vdf);
                vde.setType(ASTService.getType(tr.getChild(TYPE_CODE)));
                return vde;
        }
}
```

Deklarace proměnné (VariableDecalrationExpression) v jazyku Java má formát:

Type VariableDeclarationFragment
{ , VariableDeclarationFragment }

kde VariableDeclarationFragment je:

Identifier { [] } [= Expression]

Například i = 6 nebo d = getD().

Takže VariableDeclarationExpression může být

int i = 6;

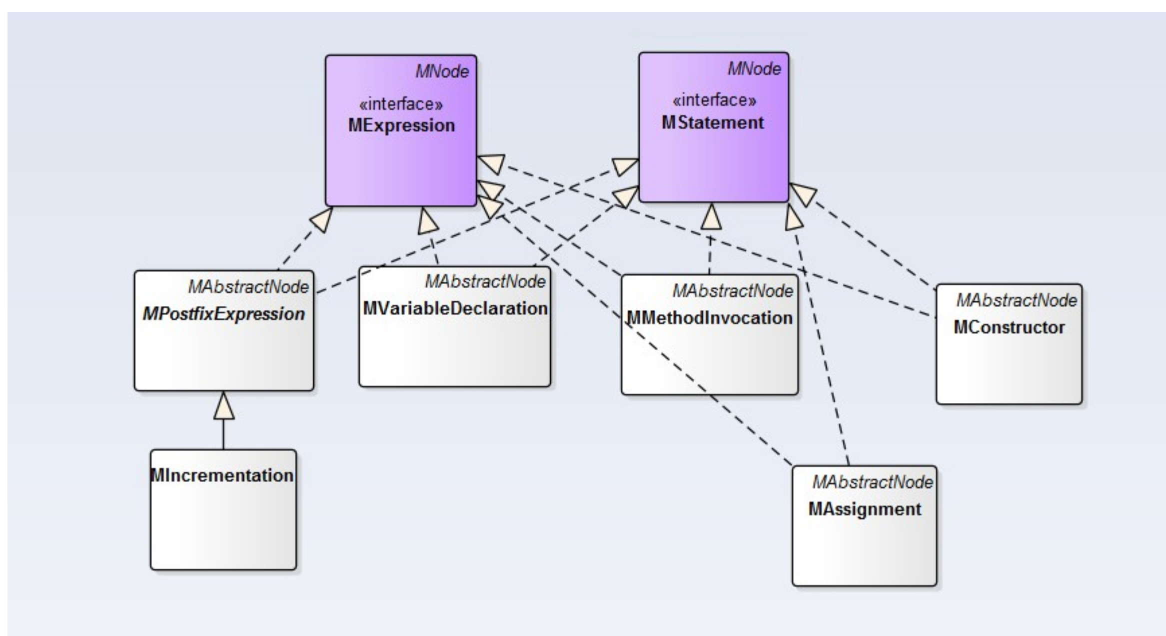
nebo

int i = 6, j = 7, k = 8;

Počet „fragmentů“ je libovolný. Jak je ale vidět v implementaci metody ***aceptVisitor***, uzel MVariableDeclaration očekává na pozici jediného potomka, a druhou definici tak nelze provést.

8.4.14 ExpressionStatement

Dvě poslední třídy implementovaly jak rozhraní MExpression, tak rozhraní MStatement. V části 6 bylo upozorněno na problém výrazů, které lze použit jako příkaz. Eclipse JDT API proto nabízí zaobalení výrazu do příkazu, které v praxi znamená přidání mezistupně do stromu. Já jsem místo toho zvolil cestu vícenásobné dědičnosti. Uzly, které mají tuto aspiraci, proto musí implementovat obě rozhraní (v Javě vícenásobná dědičnost není možná, ale lze jí nasimulovat právě pomocí rozhraní – to je důvod proč výše v hierarchii typů nejsou abstraktní třídy).



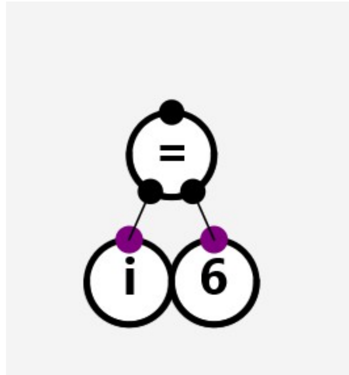
Obr 8.4.14.1: Diagram tříd implementujících rozhraní MStatement a MExpression

Použití si ukážeme na třídě MAssignment pro přiřazení. Ta má následující hlavičku:

public class MAssignment extends MAbstractNode implements MExpression, MStatement

Uzel tedy půjde připojit jak k spojům, které očekávají MExpression, tak k těm, které očekávají Mstatement.

Zapojení...:

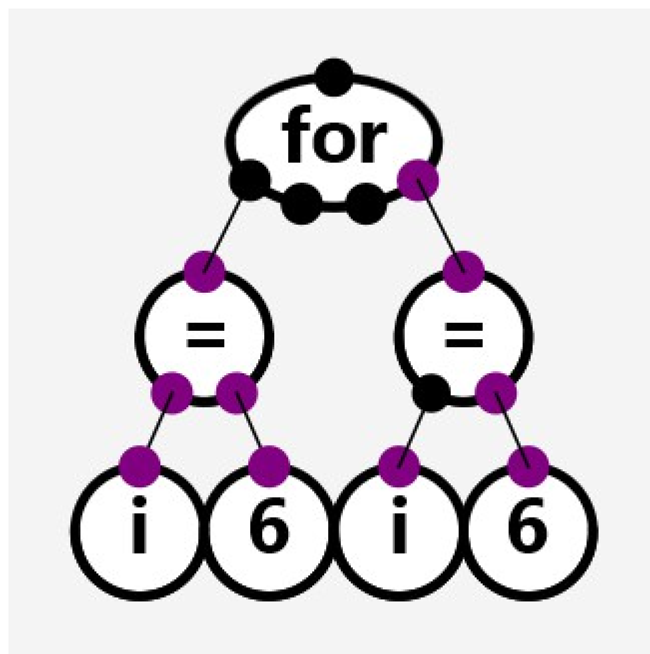


Obr 8.4.14.2: Zapojení přiřazení

...se přeloží na:

i=6

Tohle přiřazení teď připojíme k uzlu for a to na dvou místech:



Obr 8.4.14.3: Zapojení uzlu for s potomky uzly přiřazení

Pravidla pro MFor vypadají následovně:

rules.add(new JointRules(INIT_CODE, false, MExpression.class));
rules.add(new JointRules(EXPRESSION_CODE, false, MExpression.class));
rules.add(new JointRules(UPDATE_CODE, false, MExpression.class));

rules.add(new JointRules(BODY_CODE, true, MStatement.class));
K prvnímu spoji je tedy připojena instance MExpression, k poslednímu MStatement.
Metoda ***acceptVisitor*** třídy MFor vypadá následovně:

```
public ASTNode acceptVisitor(Visitor tr) {
        ForStatement fs = ASTService.getFor();
        Expression ex = ASTService.getExpression(tr.getChild(INIT_CODE));
        if (ex != null) {
                fs.initializers().add(ex);
        }
        fs.setExpression(ASTService.getExpression(
                tr.getChild(EXPRESSION_CODE)));
        ex = ASTService.getExpression(tr.getChild(UPDATE_CODE));
                if (ex != null) {
                        fs.updaters().add(ex);
                }
        Statement st = ASTService.getStatement(tr.getChild(BODY_CODE));
        fs.setBody(st);
        return fs;
}
```

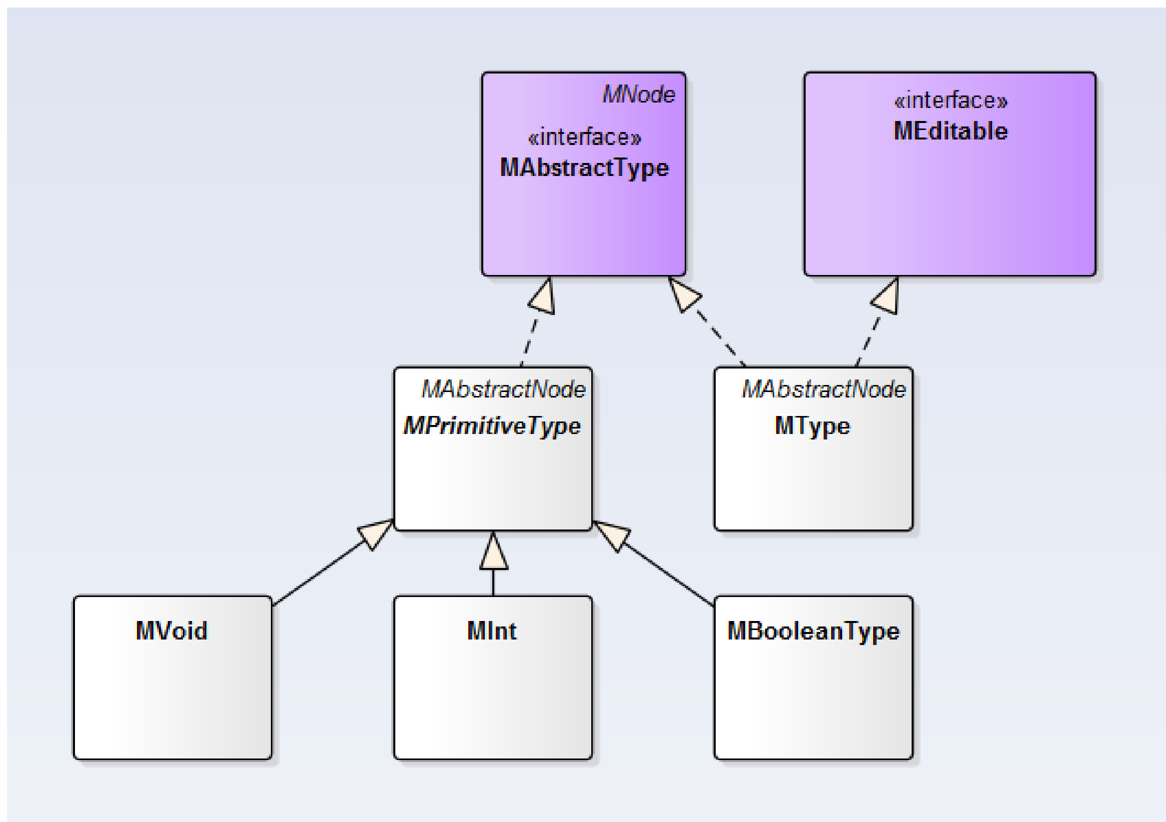
V prvním případě si třídou ASTService potomka nechá přetypovat na Expression, ve druhém na Statement. Ve druhém případě je před přetypováním ověřeno, zda se nejedná o Expression, a pokud ano, zaobalí se do pomocného objektu ExpressionStatement API Eclipse JDT. Uživatel tohoto editoru je od potřeby práce s ním odstíněn.

```
public static Statement getStatement(ASTNode child) {
        if (child instanceof Expression) {
                return ast.newExpressionStatement((Expression) child);
        } else {
                return (Statement) child;
        }
}
```

Výsledný kód:

```
for (i=6; ; )    i=6;
```

8.4.15 Rozhraní MAbstractType



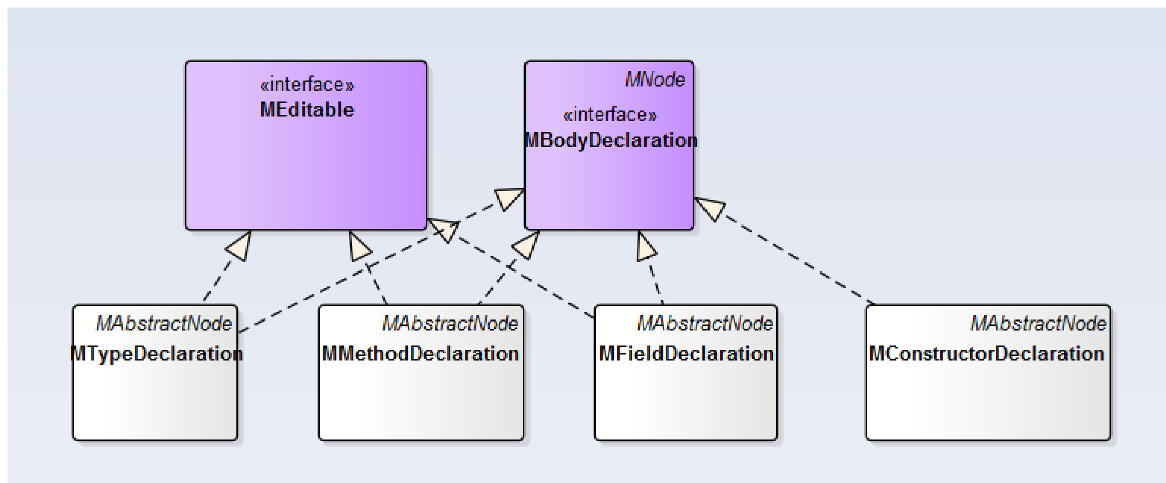
Obr 8.4.15.1: Diagram tříd implementujících rozhraní MAbstractType

Třídy v tomto balíčků slouží k definici typů. Ty můžou být primitivní, nebo odvozené. Java zná několik primitivních typů, zde jsou implementovány tři: int, boolean a void(void ve skutečnosti není v Javě typem, ale pro účely AST se na něj tak můžeme dívat). Zdrojové kódy jsou vcelku triviální, proto je zde nebudu uvádět. Uvedu pouze metody třídy ASTService pro získání Eclipse JDT API ekvivalentů:

```
public static PrimitiveType getPrimitiveType(PrimitiveType.Code code) {
        return ast.newPrimitiveType(code);
}

public static SimpleType getSimpleType(String typeName) {
        return ast.newSimpleType(ast.newName(typeName));
}
```

Jak je vidět, jedinou zajímavou vlastností třídy MPrimitiveType je statická konstanta definující typ, u odvozeného typu jeho jméno. Tyto třídy se používají například pro deklaraci typu návratových hodnot metod, jak uvidíme v další části.

8.4.16 Rozhraní MBodyDeclaration

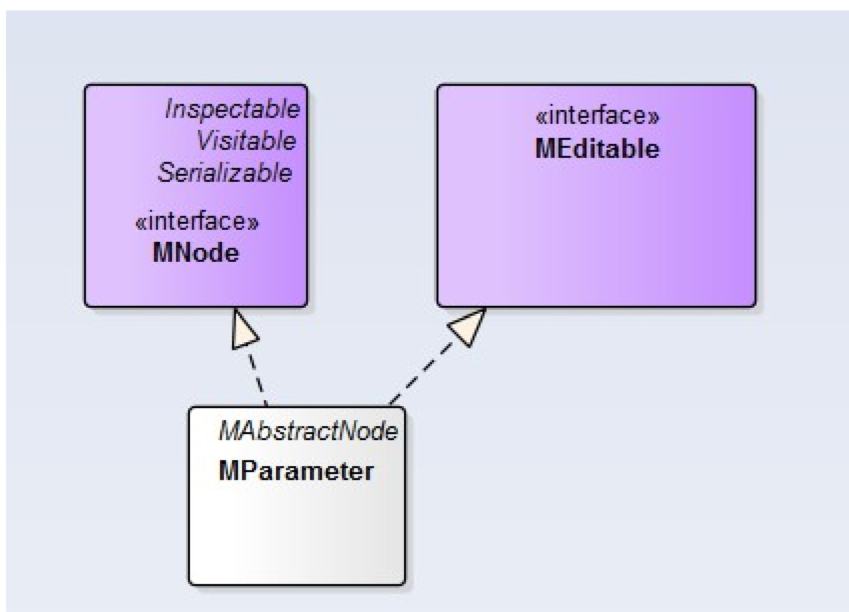


Obr 8.4.16.1: Diagram tříd implementujících rozhraní MBodyDeclaration

Třídy implementující rozhraní MBodyDeclaration slouží k definici typů (tříd), metod a konstruktorů a členských proměnných.

8.4.17 Třída MParameter

Speciální třída pro reprezentaci parametru. Používá se pouze na dvou místech: při deklaraci parametrů metod a konstruktorů. Dědí přímo z rozhraní MNode.



Obr 8.4.17.1: Třída MParameter a její rozhraní

9 Testování

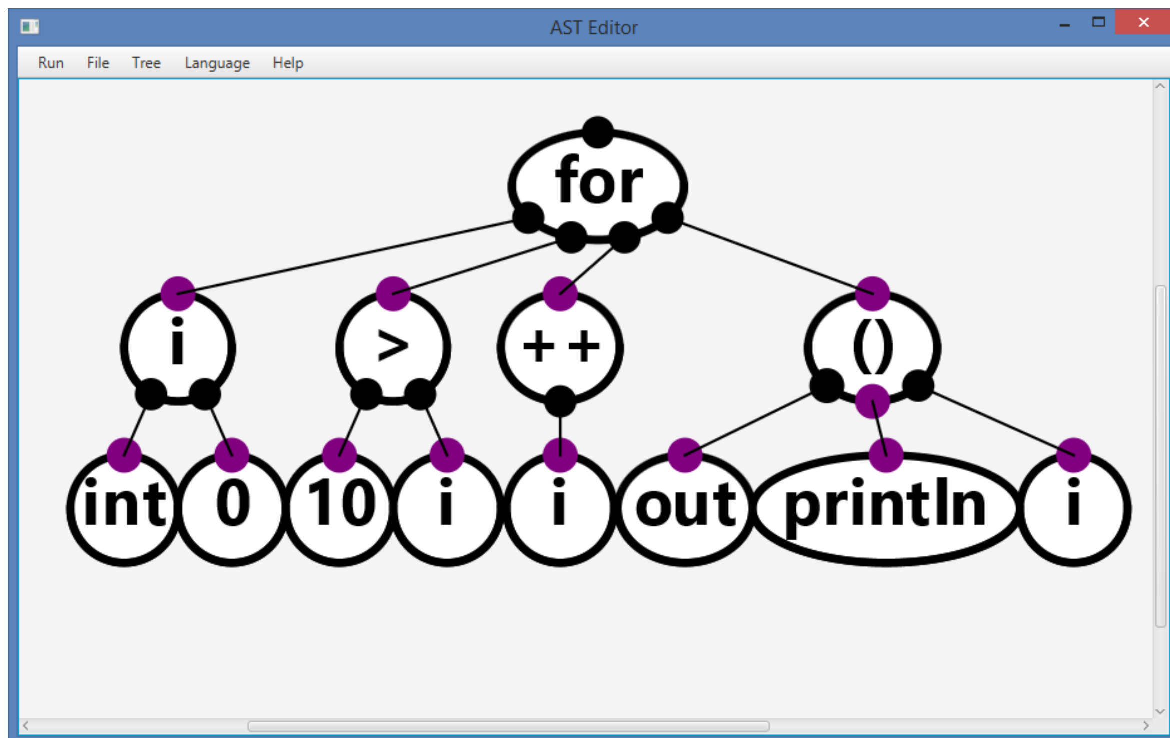
V této části se pokusím zjistit, jak si tento grafický editor jako nástroj pro vývoj aplikací vede v porovnání s klasickým psaním kódu. V další části bude demonstrována možnost spuštění vytvořeného programu.

9.1 Test

Test spočívá ve vyřešení dvojice úloh pomocí dvou nástrojů - grafického editoru AST a poznámkového bloku(NP). Čas vývoje byl měřen a porovnán. Testu se zúčastnilo pět osob. Programátor č. 1 (P1) byl autorem této práce a byl tak nejvíc zvyklý na práci s editorem. Programátor P2 byl z ostatních testovaných subjektů jediný, který se s editorem setkal již v minulosti. Programátor P3 znal obecně ideu AST. Programátoři P4 a P5 byli neznalí jak editoru, tak AST.

Úloha 1:
Sestavení řídící struktury

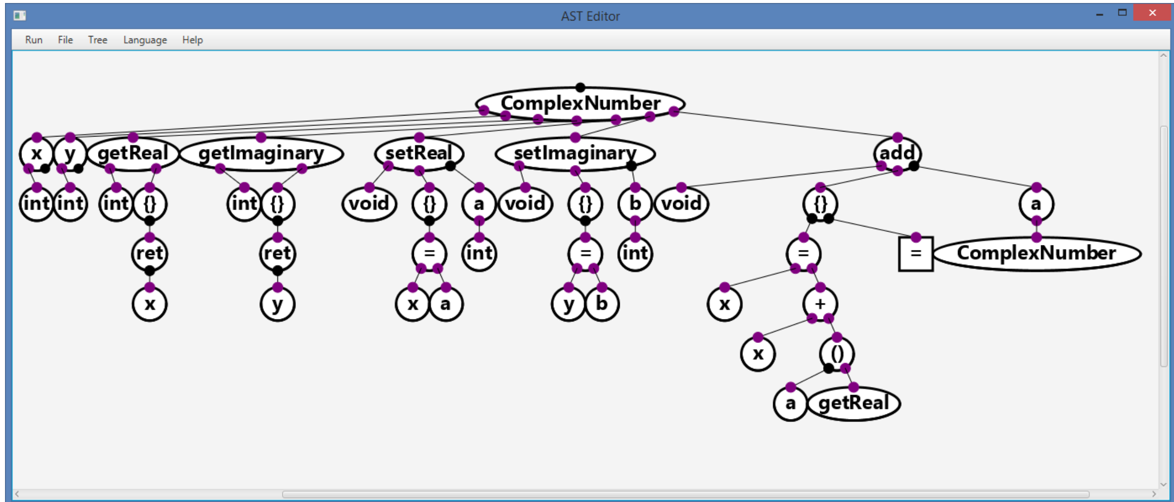
```
for (int i = 0; i<10; i++){
	out.println(i);
}
```



Obr 9.1.1: Testovací zapojení 1

Úloha 2:
Konstrukce třídy

```
Class ComplexNumber{
        int x;
        int y;

ComplexNumber(int a, int b){
        x=a;
        y=b;
}
ind getReal(){
        return x;
}
void setReal(int a){
        x=a;
}
int getImaginary(){
        return y;
}
void setImaginary(int b){
y=b;
}
void add(ComplexNumber b){
        x=x+b.getReal();
        y=y+b.getImaginary}}
```



Obr 9.1.2: Testovací zapojení 2

9.2 Výsledky

Úloha 1:

	NP	AST
P1	00:21	00:39
P2	00:48	01:38
P3	00:41	02:12
P4	00:39	02:41
P5	00:29	02:37

Obr 9.2.1: Tabulka naměřených výsledků pro úlohu 1

Úloha 2:

	NP	AST
P1	01:55	02:57
P2	01:53	03:56
P3	02:09	03:53
P4	02:15	04:12
P5	02:35	04:59

Obr 9.2.2: Tabulka naměřených výsledků pro úlohu 2

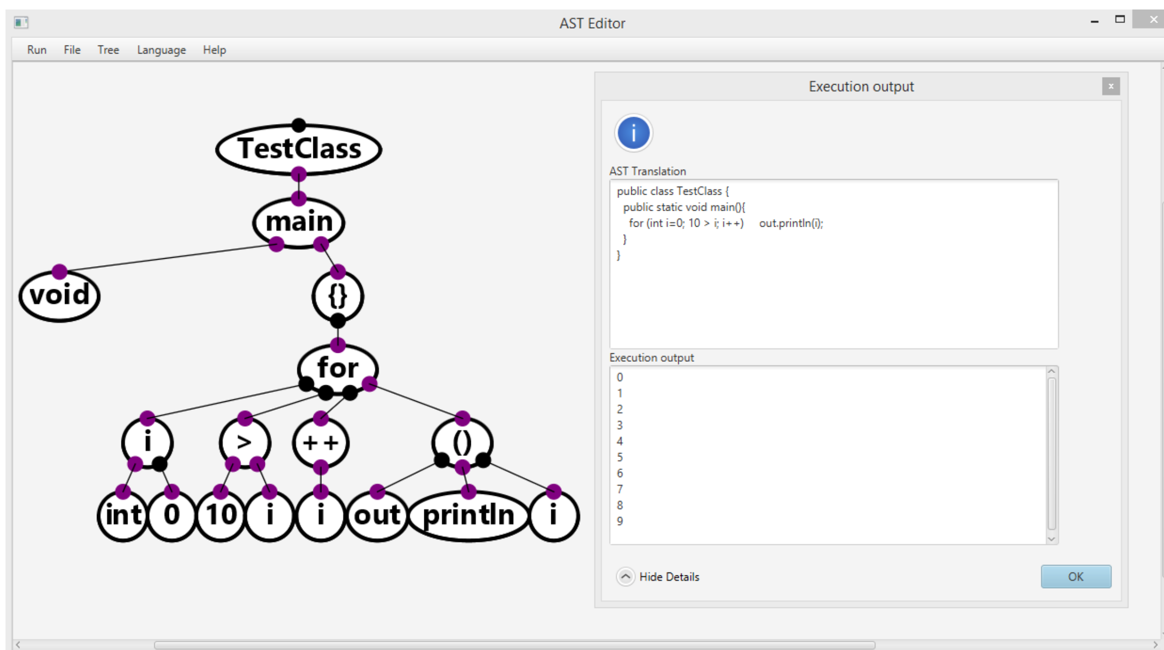
9.3 Diskuse

Jak je vidět, ve všech instancích bylo psaní kódu textově rychlejší než grafická tvorba AST. To je do velké míry samozřejmě dáno tím, že všechny testované subjekty měli více než pětiletou zkušenost s klasickým programováním. P1 byl naproti tomu s technologií seznámen relativně dobře (i když je diskutabilní, zda lze během několika týdnů vývoje získat zručnost srovnatelnou s léty praxe), i přesto však časy výrazně zaostávaly.

Subjektivně mezi výhody patří absence nutnosti psát některé strukturální znaky jako závorky nebo středníky. Nevýhodou je zejména nutnost vyvolávat myší vytvoření nového uzlu, většinou následováno výběrem z menu. Zatímco v textové podobě někdy stačí napsat „i“ , při tvorbě stromu je potřeba vyvolat menu, vybrat uzel a navíc napsat „i“. Lepších výsledků by nejspíše šlo dosáhnout vylepšením GUI a lepší ergonomií, to už je ale oblast do značné míry překrývající se s návrhem IDE.

9.4 Spuštění vytvořeného programu

Výstupem tvorby AST je zdrojový kód. V části 7.3.7 bylo představeno, jak lze tento vytvořený program vykonat. Vykonání zahrnuje kompilaci, a pokud je úspěšná, tak volání metody ***main***. Řídící konstrukt z úlohy 2 jsem pro účely této demonstrace zahrnul právě do metody ***main*** testovací třídy. Vykonání vypadá následovně:



Obr 9.4.1: Vykonání cyklu

10 Závěr

V souladu se zadáním byla v této práci představena koncepce grafické tvorby abstraktního syntaktického stromu. Byly prozkoumané možnosti použití prostředků třetích stran k dosažení cíle, navržen a implementován grafický editor. Ten teď umožňuje uživateli tvorbu AST, jeho vykonání a ukládání.

Klíčovým stavebním kamenem je technologie Eclipse JDT, která umožňuje programátorskou editaci abstraktního syntaktického stromu. Pro grafickou část editoru je použita technologie Java FX.

Výkon editoru jako nástroje pro tvorbu a vývoj programů byl otestován a porovnán s konvenčním nástrojem. Následně byla demonstrována možnost program samotný spustit a vykonat. Editor v porovnání s konvenčním nástrojem zaostával, zejména u uživatelů pro které testování představovalo první setkání s editací AST. Nicméně nejspíš nešlo od editoru očekávat srovnatelných výsledků s postupy, které jsou dnes běžnou praxí. V tomto směru by mohlo velice pomoct další vylepšení ergonomie programu.

Architektura aplikace byla navržena tak, aby její případné rozšíření bylo co možná nejintuitivnější a málo pracné. Programátor může v případě zájmu rozšířit množinu použitelných konstruktů jazyka Java pouhou implementací jediného rozhraní.

Díky oddělení řídící a grafické části je možné pro editor případně navrhnout jiné grafické rozhraní. Dalším možným vylepšením by mohl být interaktivní překlad v reálném čase z a do zdrojového kódu, jehož implementace by díky architektuře neměla být problém.

Literatura

[1]: "The Common Language Runtime (CLR) and Java Runtime Environment (JRE)", Kashif Manzoor. Dostupné z <http://www.codeproject.com/Articles/1825/The-Common-Language-Runtime-CLR-and-Java-Runtime-E >.

[2]: "Rooted Trees", Dennis L. Frey . Dostupné z <http://www.csee.umbc.edu/courses/undergraduate/341/fall98/frey/ClassNotes/Class14/trees.html >.

[3]: "Abstract and Concrete Syntax", Aarne Ranta. Dostupné z <http://www.cse.chalmers.se/edu/year/2012/course/DAT150/lectures/proglang-02.html >.

[4]: "Abstract syntax tree", . Dostupné z <http://en.wikipedia.org/wiki/Abstract_syntax_tree >.

[5]: "JRefactory", . Dostupné z <http://jrefactory.sourceforge.net >.

[6]: "CodeModelProject", . Dostupné z <https://codemodel.java.net >.

[7]: "javaparser", . Dostupné z <https://code.google.com/p/javaparser >.

[8]: "Eclipse Java development tools (JDT)", . Dostupné z <https://eclipse.org/jdt/ >.

[9]: "JavaFX", . Dostupné z <http://en.wikipedia.org/wiki/JavaFX >.

[10]: "JavaFX Architecture", . Dostupné z <https://docs.oracle.com/javafx/2/architecture/jfxpub-architecture.htm >.

[11]: "JDT Plug-in Developer Guide", . Dostupné z <http://help.eclipse.org/kepler/index.jsp?nav=%2F3 >.

[12]: "Java model", . Dostupné z <http://help.eclipse.org/luna/index.jsp?topic=%2Forg.eclipse.jdt.doc.isv%2Fguide%2Fjdt_int_model.htm&cp=3_0_0_0 >.

[13]: "Abstract Syntax Tree", Thomas Kuhn, Eye Media GmbHOlivier Thomann, IBM Ottawa Lab. Dostupné z <http://eclipse.org/articles/Article-JavaCodeManipulation_AST/index.html >.

[14]: "Insert/Add Statements to Java Source Code by using Eclipse JDT ASTRewrite", X Wang. Dostupné z <http://www.programcreek.com/2012/06/insertadd-statements-to-java-source-code-by-using-eclipse-jdt-astrewrite/ >.

[15]: "Eclipse JDT API Specification", . Dostupné z <http://help.eclipse.org/juno/index.jsp?topic=%2Forg.eclipse.jdt.doc.isv%2Freference%2Fapi%2Findex.html&org/eclipse/jdt/core/dom/package-summary.html >.

[16]: "reflections", . Dostupné z <https://code.google.com/p/reflections/ >.

A Uživatelské informace

Pro spuštění jsou nezbytné následující knihovny:

- guava-18.0.jar
- javassist.jar
- org.eclipse.core.contenttype_3.4.200.v20130326-1255.jar
- org.eclipse.core.jobs_3.5.300.v20130429-1813.jar
- org.eclipse.core.resources_3.8.101.v20130717-0806.jar
- org.eclipse.core.runtime_3.9.100.v20131218-1515.jar
- org.eclipse.equinox.preferences_3.5.100.v20130422-1538.jar
- org.eclipse.equinox.common_3.6.200.v20130402-1505.jar
- org.eclipse.osgi_3.9.1.v20140110-1610.jar
- org.eclipse.jdt.core_3.9.2.v20140114-1555.jar
- org.osgi-3.0.0.jar
- reflections.jar
- reflections-0.9.9-RC1-uberjar.jar

Všechny se nacházejí v adresáři AST a jsou přibalené taky ve spustitelném AST.jar souboru.

Projekt vyžaduje verzi Javy 1.8. Pro plnou funkcionalitu však nestačí JRE, ale je potřebné JDK. Pro kompilaci a následné spuštění metody ***main*** je totiž potřeba nástrojů JDK.
Je doporučeno nastavit proměnnou prostředí Path na patřičné umístnění JDK a následně spustit z příkazového řádku

java -jar ast.jar

z umístnění kde se jar soubor nachází.

Je také potřeba počítat s tím, že při kompilaci dojde na disku k vytvoření dvou souborů (.java a .class) se zdrojovým kódem a bytekódem. Soubory budou vytvořeny v systémovém adresáři pro dočasné soubory ("java.io.tmpdir") a při terminaci JVM budou smazány.

B Obsah přiloženého CD

CD obsahuje tuto práci jak ve zdrojovém formátu .odt, tak ve formátu pdf. Ve složce figs se nachází veškeré použité obrázky a schemata.

CD dále obsahuje kompletní Eclipse projekt připravený k importu (a taky jeho achiv). Tento projekt obsahuje několik nezbytných knihoven pro spuštění a správnou funkčnost.

Navíc se zde nachází spustitelný jar archiv připravený ke spuštění dle instrukcí v příloze A.

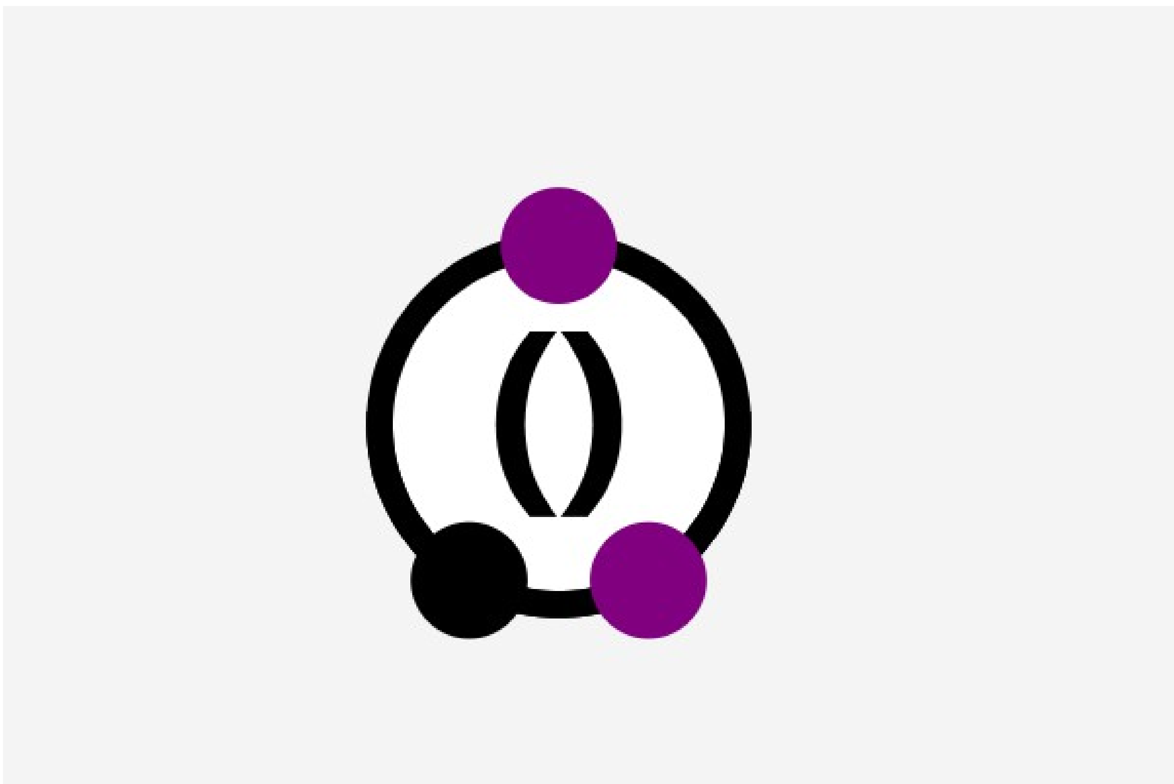
C Uživatelská příručka

C.1 Úvod

Tento dokument poskytuje návod k práci s Editorem abstrakních syntaktických stromů a představuje veškerou jeho funkcionalitu.

C.2 Uzel

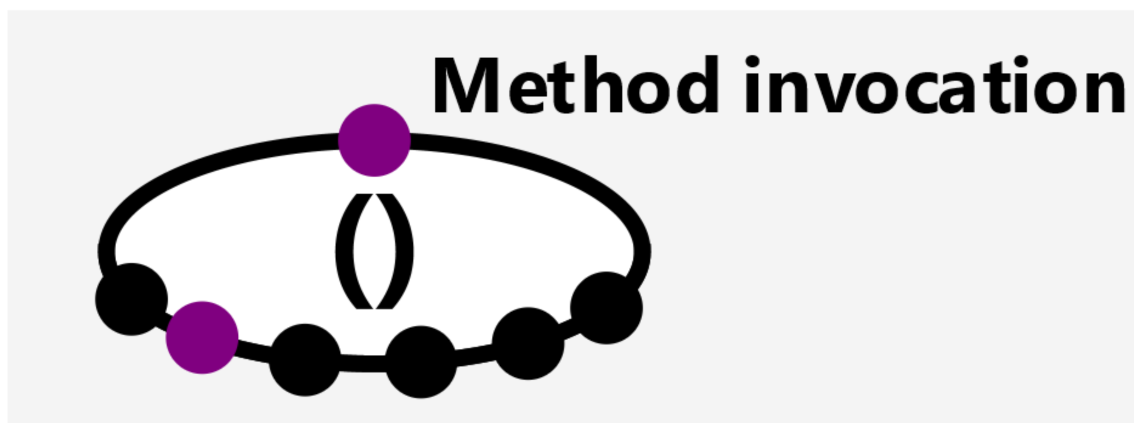
Uzel abstraktního syntaktického stromu je reprezentován grafickým prvkem – kružnicí (nebo elipsou – dle velikosti obsahu) s několika dalšími součastmi (dále jen uzel).



Obr C.1: Uzel typu Volání metody

Uzel obsahuje popisek, který buď charakterizuje jeho typ (například na obrázku typ „volání metody“), nebo jeho hodnotu (například číselný literál). Po obvodu uzlu se nacházejí kruhy. Jedná se o spoje, které umožňují navázat spojení s jinými uzly. V horní

části je právě jeden univerzální spoj pro spojení s rodičovským uzlem. Ve spodní částí se nachází dle typu uzlu specifická konfigurace spojů. Jejich konfigurace spočívá v počtu, očekávaném druhu potomka a případně dynamickém přidávání nových spojů (například počet argumentů metody není konstantní). Po najetí myší na uzel se zobrazí popisek s názvem typem uzlu.

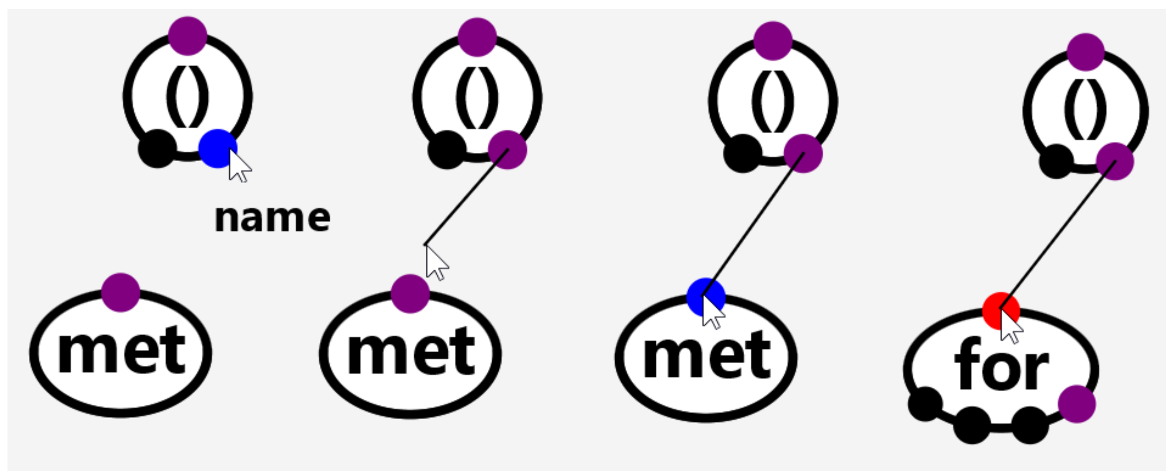


Obr C.2: Uzel typu Volání metody s dynamicky přidanými spojy a popiskem

C.3 Spoje

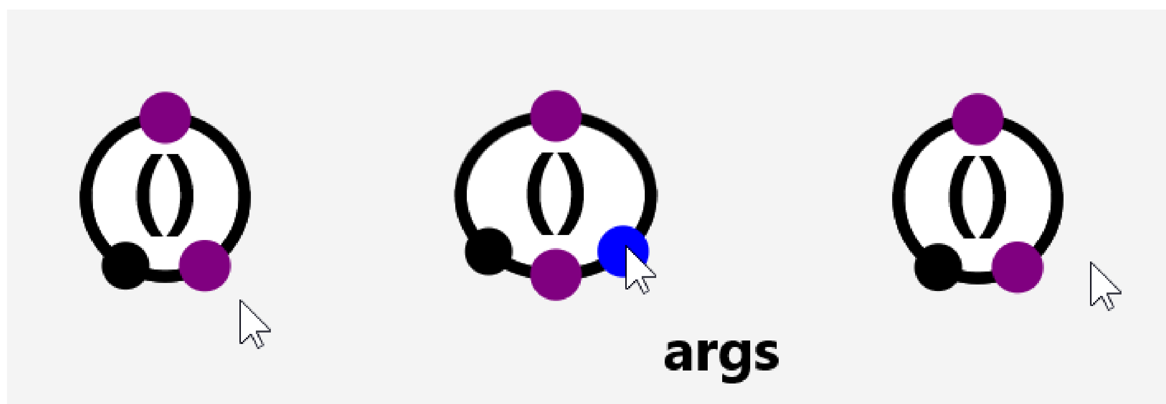
Najetím myší na spoj se vyvolá zobrazení popisu spoje – jedná se o kód naznačující jaký podstrom má být připojen na daný spoj.

Po zahájení tahu myší se zmáčknutým tlačítkem (drag) ze spoje je vyprodukována hrana spojena s daným spojem. Dále tahem můžeme hranu připojit na jiný spoj (drop). Protože spoje nesou informaci o tom, jaké uzly jsou pro ně přijatelné pro připojení, pokus o připojení nepřijatelného typu uzlu je odmítnut. Jak je vidět na obrázku 9.4.1, nelze jako identifikátor metody připojit příkaz For.



Obr C.3: Připojení uzlu definující název metody a odmítnutí připojení nevhodného typu uzlu

Pokud má uzel povolenou dynamickou tvorbu spojů, stačí najet na spodní část uzlu a dočasný spoj se vytvoří automaticky. Dočasný spoj se stane trvalým, jenom pokud se jej podaří úspěšně připojit k jinému uzlu, jinak je opět zrušen. Spoje se můžou dynamicky tvořit buď za statickými spoji (bráno zleva doprava – v případě uzlu Volání metody jsou dva statické spoje), nebo kdekoliv na uzlu (například uzel Blok).



Obr C.4: Dynamicky vytvořený spoj pro argumenty metody

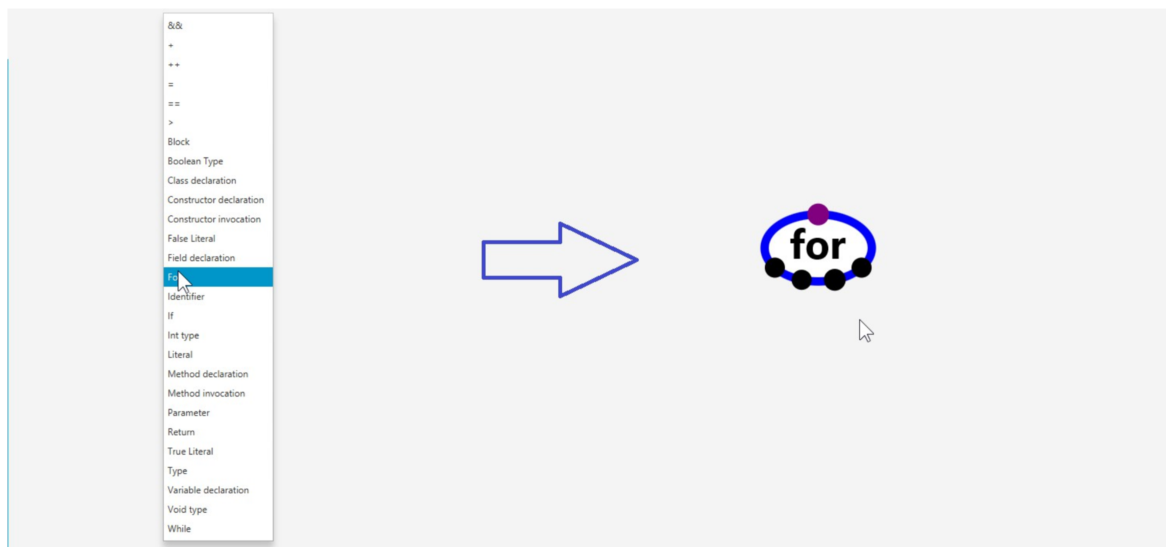
C.4 Práce s uzlem

Nejzákladnější operace s uzlem je pohyb po ploše. Ten se provádí velice intuitivně: stiskem primárního tlačítka myši nad uzlem a následným pohybem se stisknutým tlašítkem.

C.4.1 Vytvoření nového uzlu

Uzel lze vytvořit dvojím způsobem.

a) Kliknutí sekundárním tlačítkem myši na plochu editoru vyvolá rozbalovací menu nabízející všechny typy uzlů.



Obr C.5: Vytvoření nového uzlu z kompletní nabídky

b) vytvoření uzlu z konkrétního spoje. Protože spoj přijímá pouze něteré uzly, jsou v menu nabídnuty pouze příslušné možnosti. Pokud spoj přijímá jediný typ uzlu, je tento vytvořen automaticky bez zobrazení nabídky.

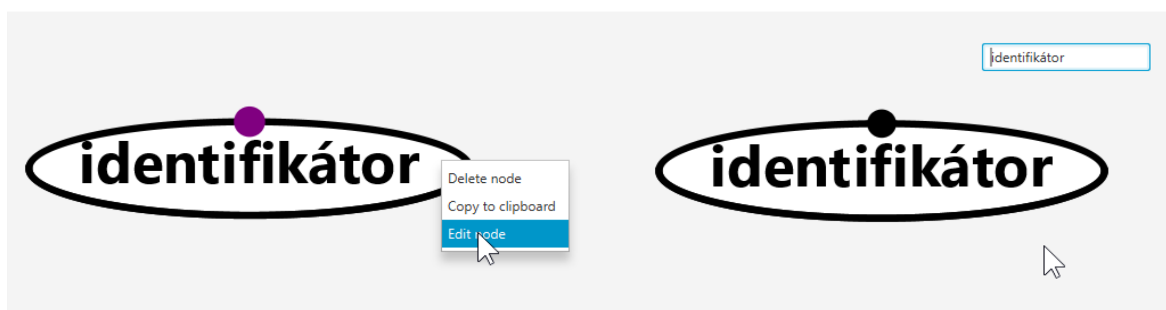
Obr C.6: Vytvoření nového uzlu z nabídky specifické pro spoj rodiče

C.4.2 Kontextová nabídka

Kliknutí pravým tlačítkem myši na uzlu vyvolá kontextovou nabídku. Ta nabízí smazání uzlu, jeho editaci (je-li uzel editovatelný) a kopírování podstromu.

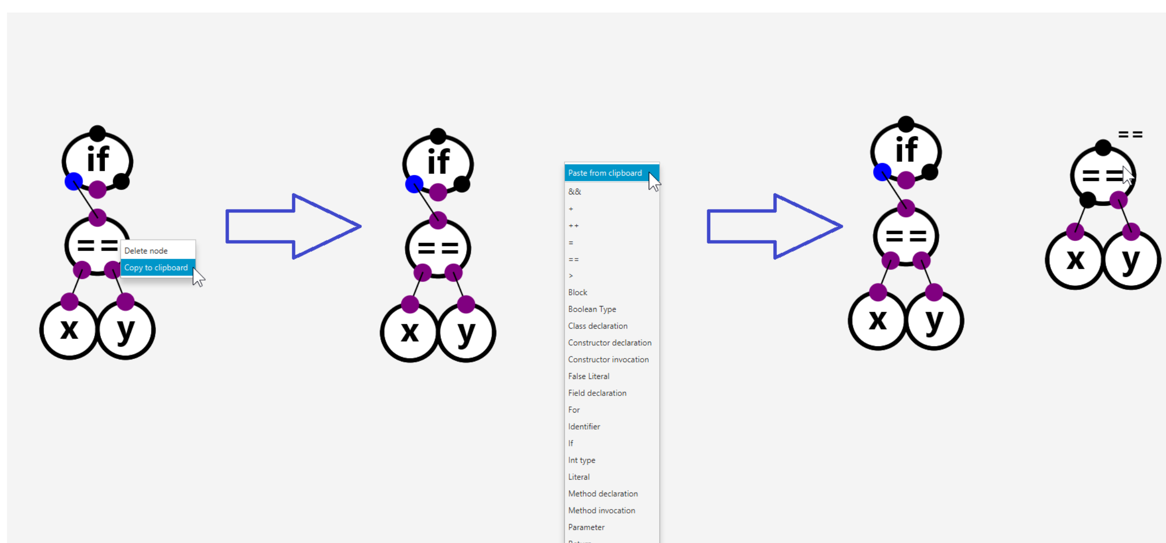
Editace uzlu

Některé uzly (například Identifikátor nebo Literál) mají proměnnou hodnotu. Ta se zadává pomocí dialogu, který je vyvolán při vytvoření uzlu a lze jej opět vyvolat kontextovou nabídkou.



Obr C.7: Editace uzlu

Kopírování podstromu
Výběr možnosti kopírování uloží kopii celého podstromu (s daným uzlem jako kořenem) do schránky. Vložit podstrom ze schránky na plochu lze pomocí kompletní nabídky. Kopie je přístupná taky v specifické nabídce, pokud kořen stromu vyhovuje podmínce.

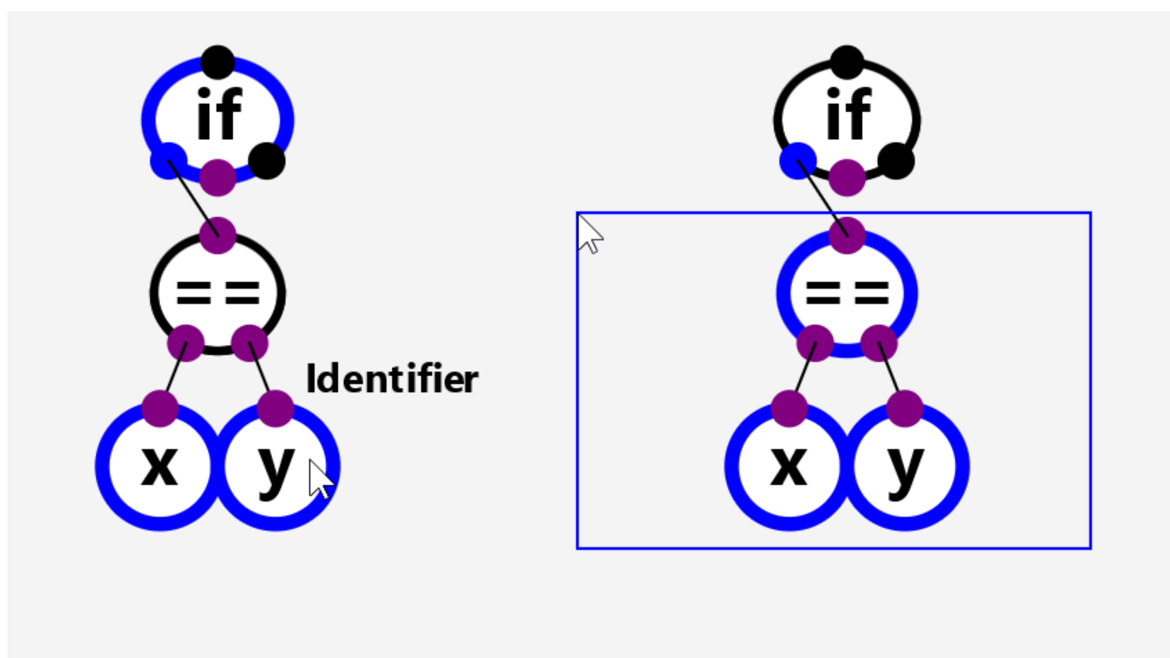


Obr C.8: Kopírování podstromu

C.4.3 Mazání a zaměření

Uzel lze smazat třemi způsoby: pomocí kontextové nabídky, zmáčknutím kolečka myši nad uzlem, nebo zmáčknutím klávesy DELETE. To vyvolá smazání uzlů které mají aktuálně zaměření (focus). To je signalizováno modrým zbarvením uzlu.

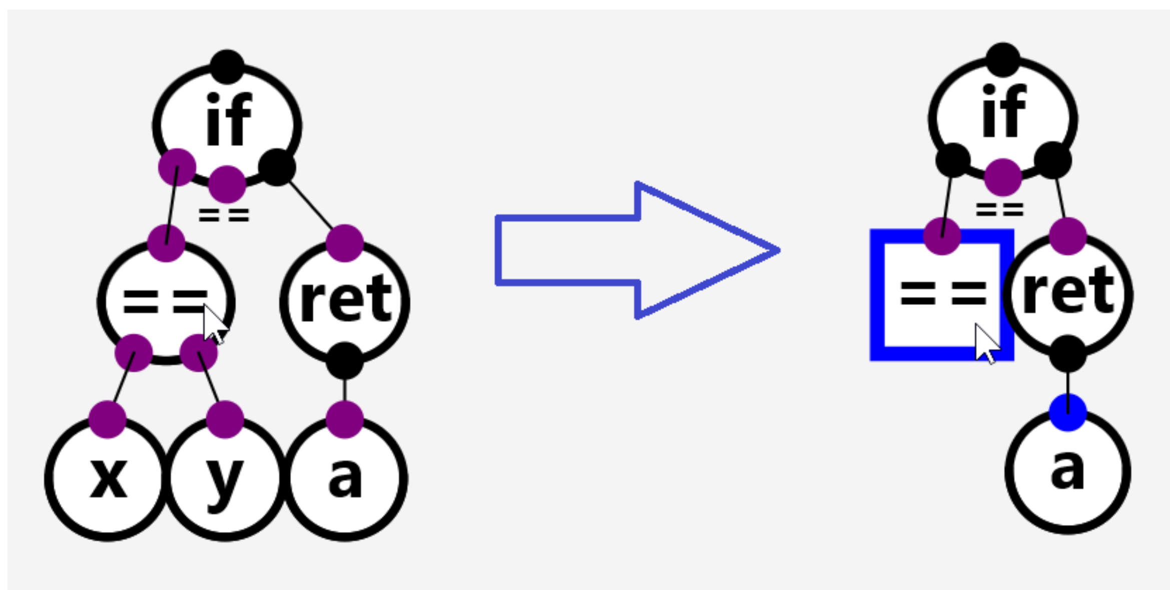
Zaměření uzel získá stiskem primátního tlačítka. Tohle zaměření může držet v jednu chvílí pouze jeden uzel, je však možné i hromadné zaměření: buď přidržením klávesy CONTROL při výběru, nebo použitím selekčního boxu. Ten se vyvolá pohybem myši po ploše se zmáčknutým primárním tlačítkem. Dvojklik na plochu primárním tlačítkem zbaví všechny uzly zaměření.



Obr C.9: Hromadný výběr uzlu

C.4.4 Sbalení uzlu

Dvojitým kliknutím primárním tlačítkem na uzlu způsobí zabalení uzlu. Uzel změní svlj tvar na čtverec a celý podstrom je schován. Rozbalení se vyvolá stejným způsobem.



Obr C.10: Sbalení podstromu

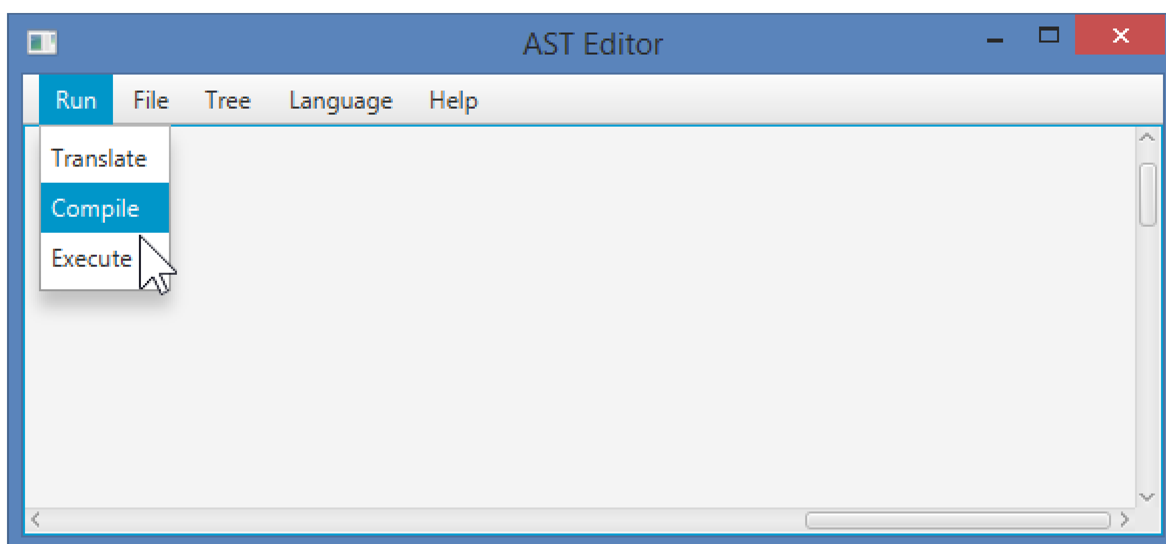
C.5 Práce s plochou

Pracovní plochu editoru lze přizpůsobit trojím způsobem.

Rolování – pomocí rolovacích proužků (scrollbar) na okrajích okna se lze posouvat po ploše.
Zrychlený pohyb – posunem po ploše se zmáčknutým kolečkem myši se lze zrychleně pohybovat po ploše.
Změna velikosti plochy – pokud je plocha nedostatečná, lze její velikost měnit rolováním kolečka myši.

C.6 Menu

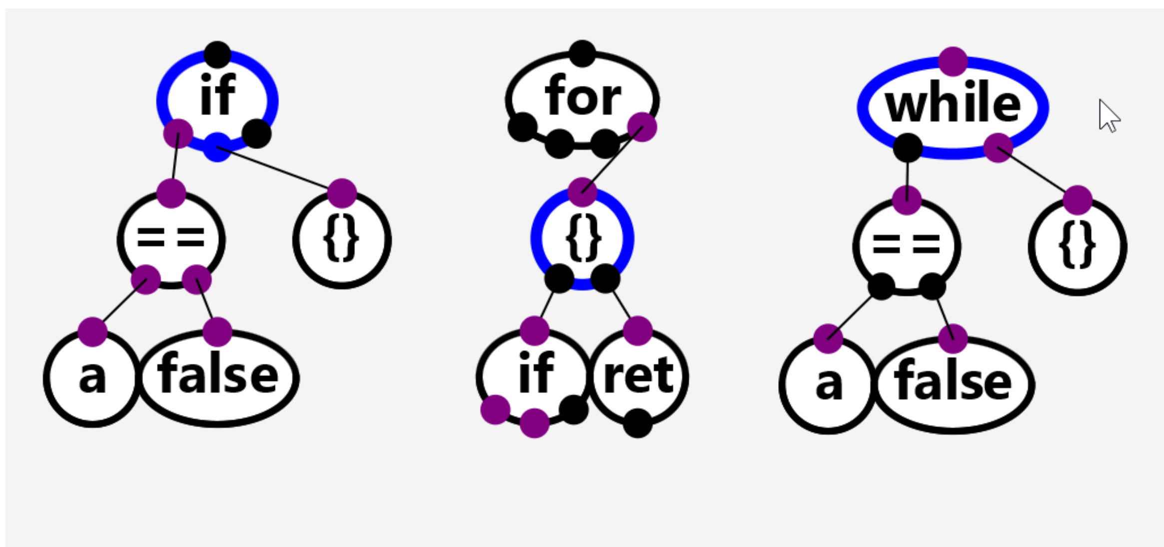
Okno editoru obsahuje menu s několika položkami. Ty nabízí mimo jiné operace nad stromy. Protože v editoru jich v danou chvíli může být několik, je potřeba určit, nad kterými bude daná operace vykonána.



Obr C.11: Okno editoru s menu

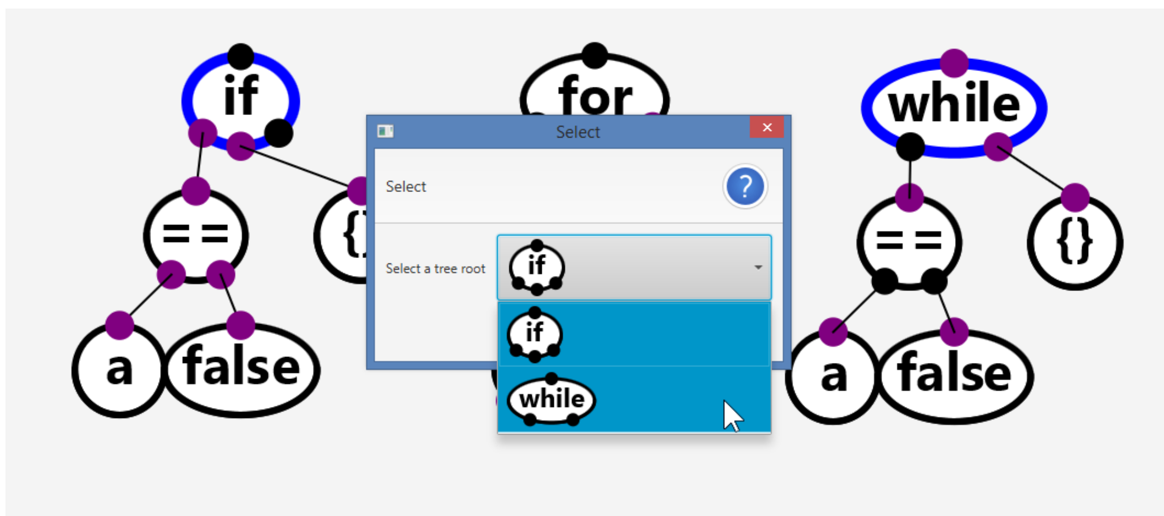
C.6.1 Výběr stromu pro operaci

Výběr cílového stromu závisí na dané operaci. Některé operace (konkrétně z podnabídky Tree/Strom) lze provést nad všemi stromy. To se také stane, pokud není explicitně určena podmnožina stromů, které mají být cílem operace. Výběr konkrétních stromů se provede tím, že jejich kořeny mají zaměření (focus). Na obrázku 9.7.7.1 by byla operace vykonána nad prvním a třetím stromem, protože jejich uzly mají zaměření. Zaměření nekořenových uzlů nehraje roli.



Obr C.12: Množina možných stromů pro operaci

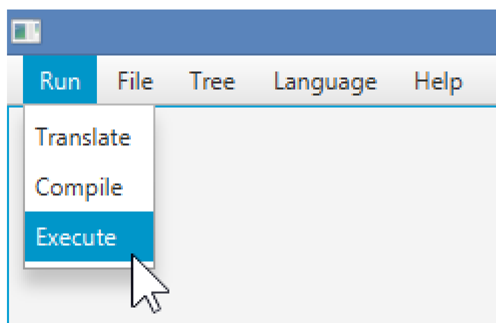
Další operace lze spustit pouze nad jedním stromem. Pokud je konkrétní strom určen jednoznačně (je jediný, nebo jeho kořen je jediný se zaměřením), operace se vykoná ihned. Jinak je uživateli nabídnuto výběrové okno pro selekci stromu. U případu z obrázku C.12 by okno vypadalo následovně:



Obr C.13: Výběr stromu pro operaci

Strom s kořenem For není v nabídce, protože jeho kořen nemá zaměření. Pokud by jej měl, nebo by bez zaměření byly všechny kořeny, nabídka by jej obsahovala.

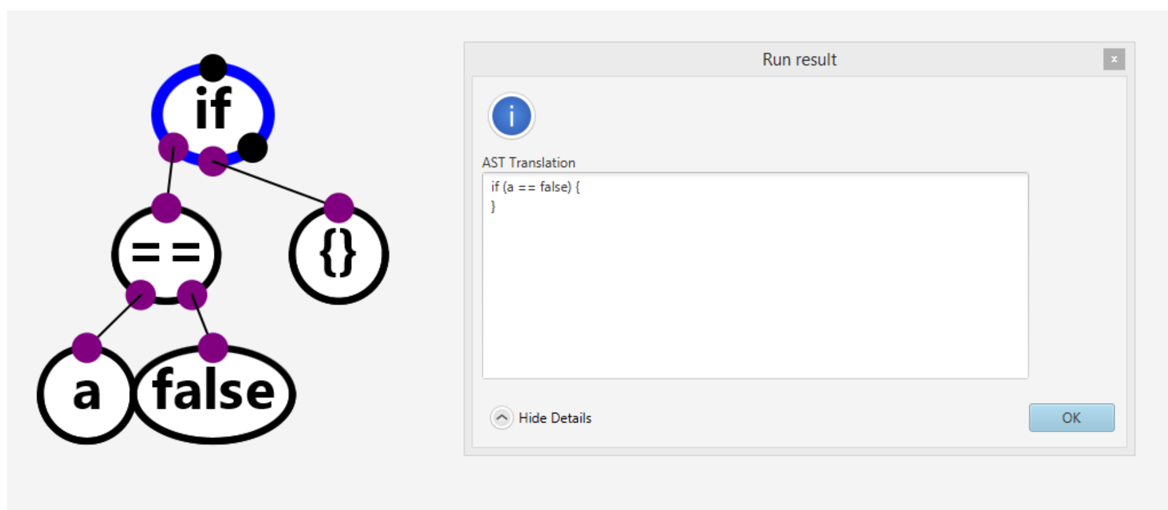
C.6.2 Položka menu Run



Obr C.14: Run submenu

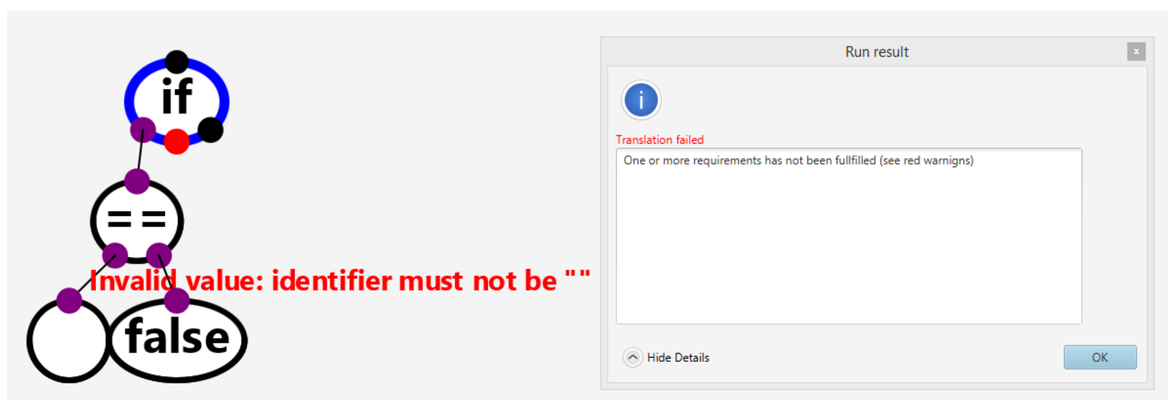
Tato podnabídka nabízí tři možnosti pro spuštění běhu.

Translate/Přeložit – vyvolá „přeložení“ vytvořeného abstraktního syntaktického stromu do zdrojového kódu. V případě úspšchu je zdrojový kód zobrazen v dialogovém okně.



Obr C.15: Přeložení jedoduchého AST do zdrojového kódu

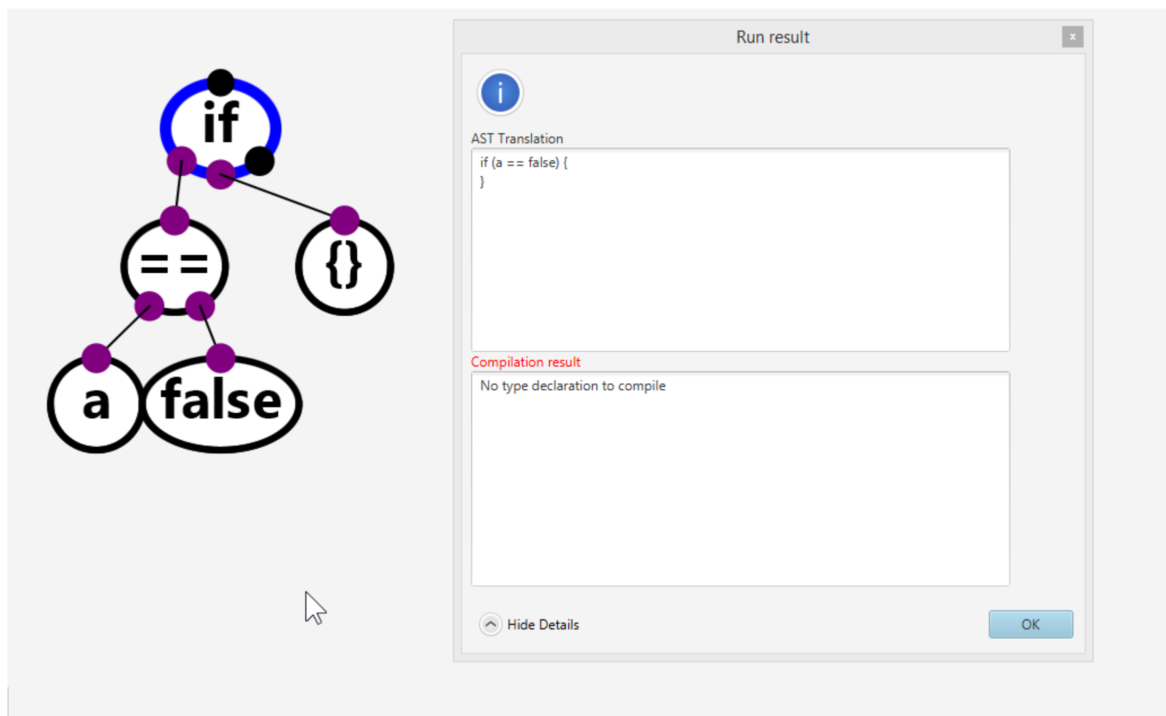
Pokud překlad selže kvůli nesplnění některé z podmínek, zobrazí se chyboví dialog a chyby jsou signalizovány na stromu:



Obr C.16: Selhání překladu

V tomto případě se jedná o nesplnění připojení povinného podstromu “then“ a neplatný identifikátor.

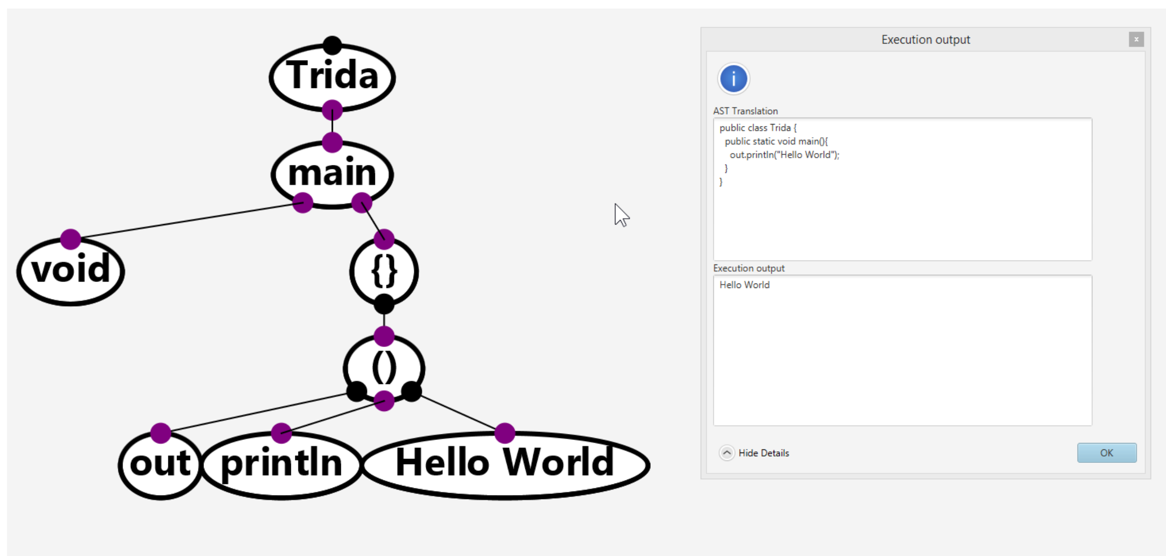
Compile/Kompilovat – v tomto případě je překlad do zdrojového kódu následován pokusem o kompilaci. Dialogové okno kromě přeloženého zdrojového kódu obsahuje výsledek kompilace. V následujícím příkladě nastane chyba, protože Java kompilátor kompiluje pouze deklarace typů.



Obr C.17: Neúspěšný pokus o kompilaci stromu

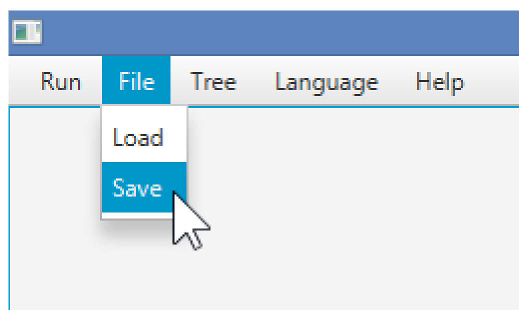
Při kompilaci dochází k uložení zdrojového kódu do souboru v systémovém adresáři pro dočasné soubory a případný soubor obsahující byte kód je taky uložen v tomto umístnění. Soubory jsou dočasné a při terminaci JVM jsou odstraněny.
Protože editor neumí pracovat s importy a modifikátory přístupu, jsou do zdrojového kódu připsany automaticky.

Execute/Vykonat – tahle operace se ve zkompilovaném souboru snaží nalézt metodu main a následně ji vykoná. Standardní a chybový výstup jsou převedeny a vypsány do dialogového okna. Pro výpis na standardní výstup je při kompilaci do zdrojového kódu přidán statický import objektu *System.out,* takže stačí zavolat na objektu *out* metodu *println.*



Obr C.18: Vykonání "Hello World" programu

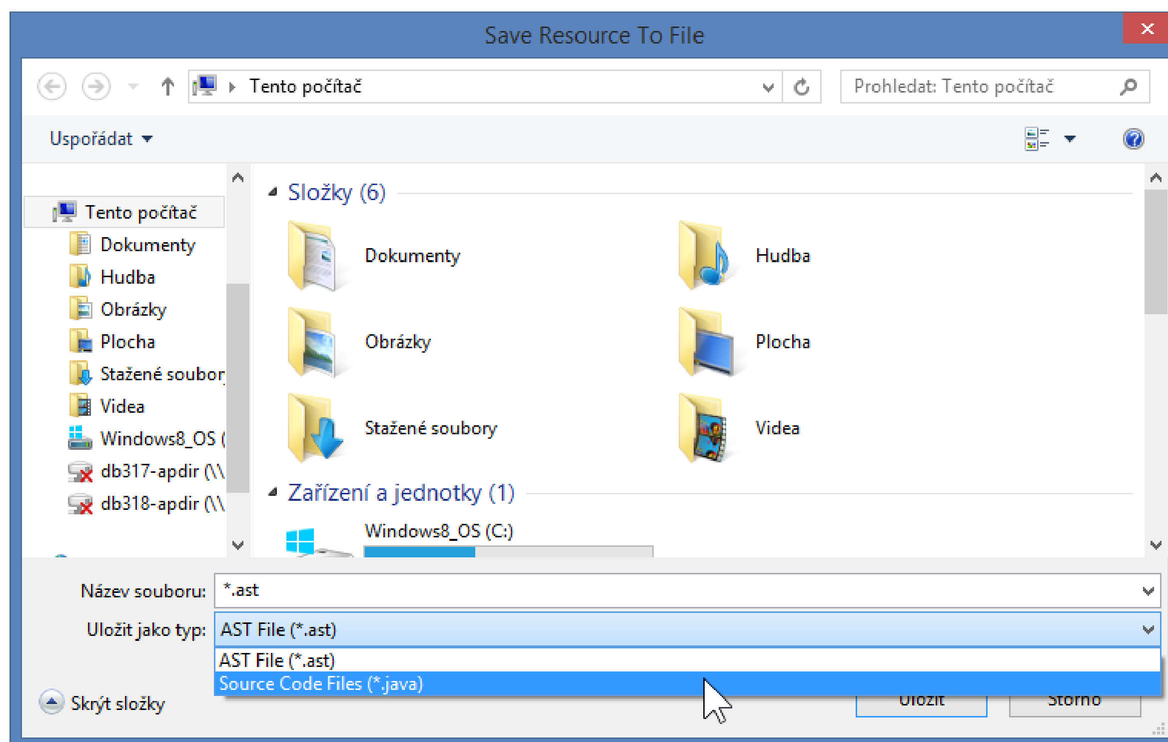
C.6.3 Položka menu File



Obr C.19: File submenu

Tato podnabídka nabízí načtení a ukládání souborů.Editor umí pracovat se soubory ve dvou formátech. Jednak s textovými soubory (typu .txt a .java) za kterých čte/ukládá zdrojový kód. Druhým formátem je vlastní formát editoru .ast.

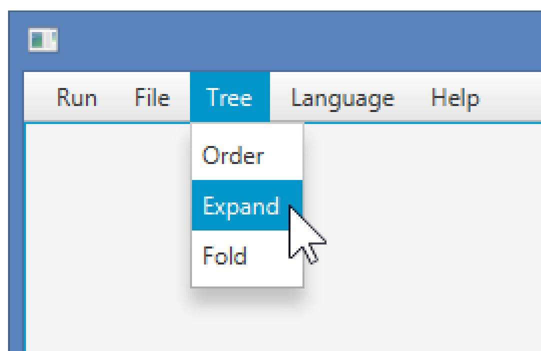
Save/Uložit – uložení začíná výběrem stromu k uložení dle pravidel popsaných v části 9.7.1. Následuje standardní dialog pro výběr souboru včetně volby formátu souboru. V případě, že se ukládá Deklarace třídy ve formátu .java, dochází k přidání deklarací balíčku, importu a přidání modifikátoru přístupu podobně jako při kompilaci.



Obr C.20: Výběr formátu uložení stromu

Load/Načíst – ve formátu .java je podporováno pouze čtení stromu Deklarace třídy. Protože zdrojový kód může obsahovat konstrukty neznáme editoru, je možné, že strom nebude načten kompletně.

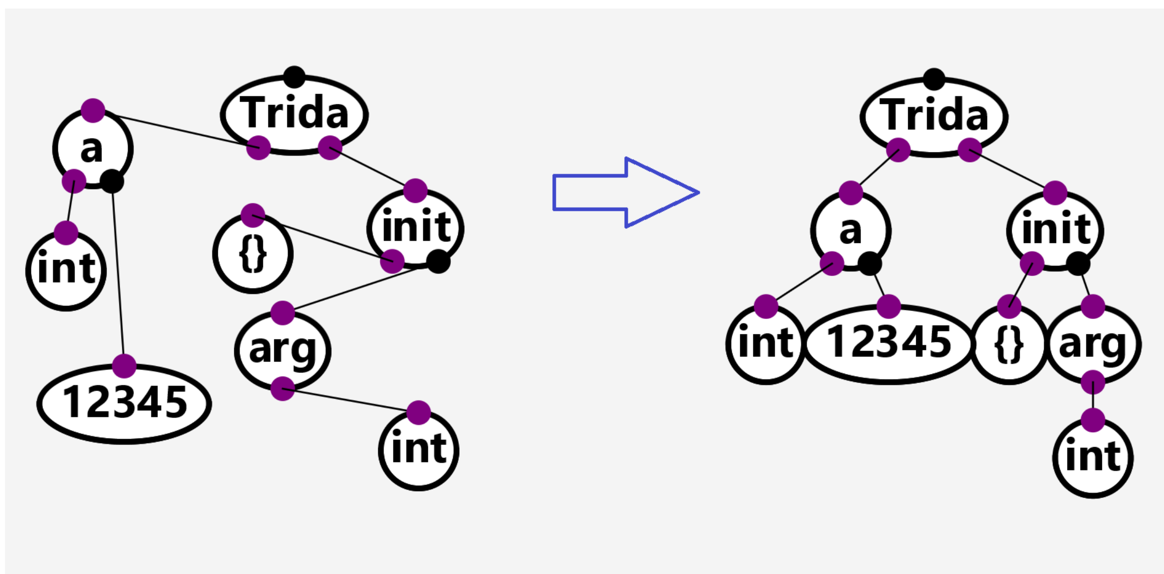
C.6.4 Položka menu Tree



Obr C.21: Tree submenu

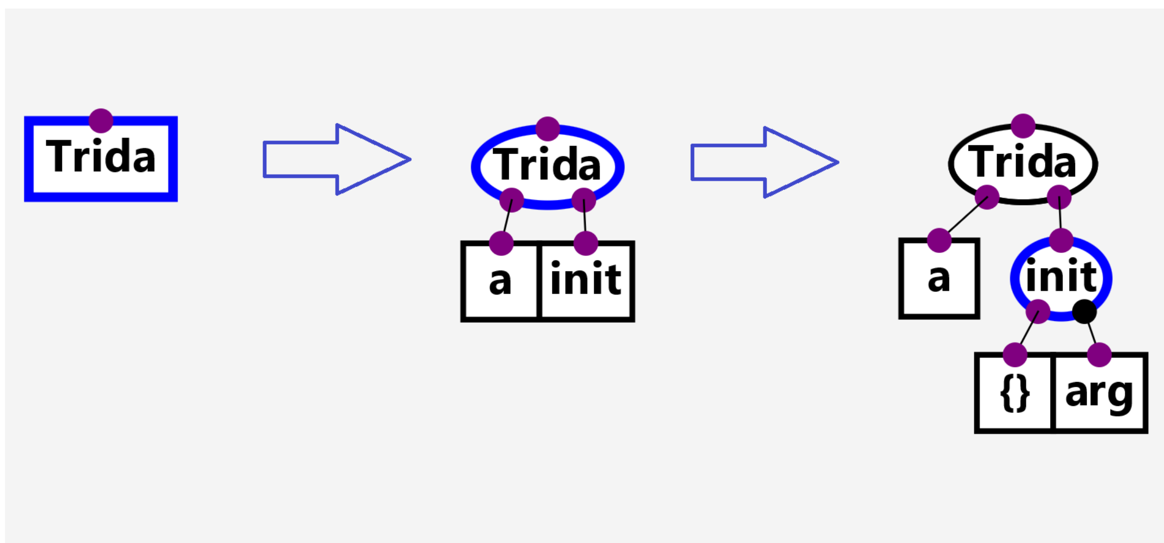
Nabízí několik operací úpravy stromů. Operace se vykonají nad všemi stromy v editoru, pokud není explicitně určeno jinak (viz část C.6.1).

Order/Uspořádat – převede strom do uspořádaně pravidelné struktury



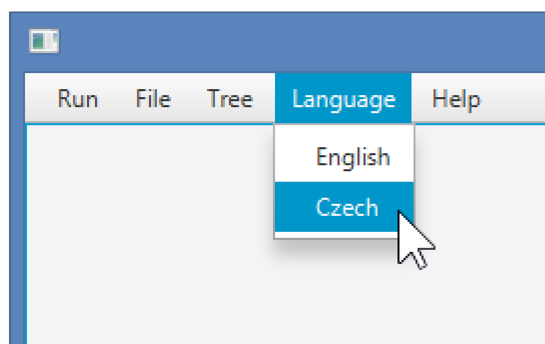
Obr C.22: Uspořádání stromu

Rozbalit-Sbalit/Expand-Fold – sbalení uzlu bylo popsáno v části 9.5.5. Při volbě této operace dojde ke kompletnímu sbalení všech uzlů ve zvolených stromech. Po rozbalení kořene zůstavají potomci sbalení. Pro kompletní rozbalení všech uzlů slouží právě zbývající položka této podnabídky.



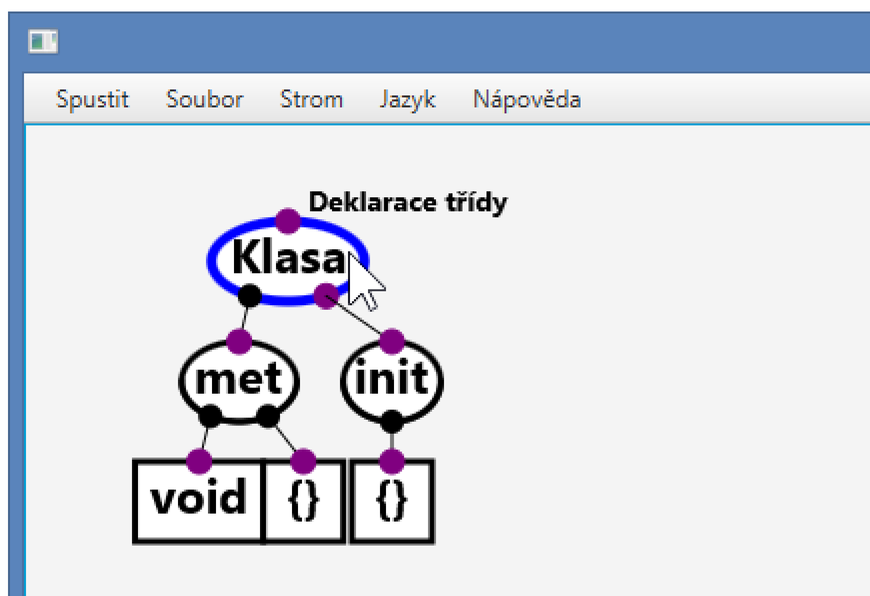
Obr C.23: Postupné rozbalování kompletně sbaleného stormu

C.6.5 Položka menu Language



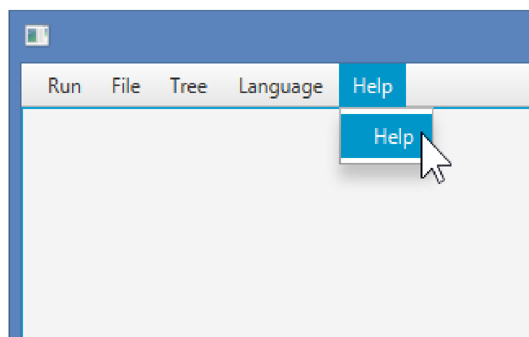
Obr C.24: Tree submenu

Nabízí možnost změny jazyka. V editoru jsou podporovány jazyky čeština a angličtina.



Obr C.25: Česká lokalizace

C.6.6 Nápověda



Obr C.26: Help submenu

C.7 Gesta

Pro svižnější práci s editorem jsou k dispozici tzv. Gesta. Jedná se o koncept jehož inspirací je webový prohlížeč Opera. Konkrétně se jedná o pohyb myší nad plochou se zmáčknutým sekundárním tlačítkem v určitém směru.
Každé gesto vyvolá jednu z operací přístupných i skrze menu, konkrétně:

Gesto **LEFT** – operace Přeložit
Gesto **RIGHT** – operace Vykonat
Gesto **DOWN** – operace Načíst
Gesto **UP** – operace Uspořádat