MASARYKOVA UNIVERZITA

FAKULTA INFORMATIKY

Vývojové prostředí pro webový šablonovací systém

Diplomová práce

Brno, 2008 Bc. Martin Lazar

Prohlášení

Prohlašuji, že tato diplomová práce je mým původním autorským dílem, které jsem vypracoval samostatně. Všechny zdroje a literaturu, které jsem při vypracování používal nebo z nich čerpal, v práci řádně cituji s uvedením úplného odkazu na příslušný zdroj.

.....

Martin Lazar

Poděkování

Rád bych tímto poděkoval RNDr. Radku Ošlejškovi Ph.D., vedoucímu této diplomové práce, za ochotu a čas, který mi věnoval. Můj dík také patří pracovníkům společnosti oXy Online, zejména řediteli vývoje Ing. Pavlu Klobásovi, za vstřícný přístup, věnovaný čas a podnětné připomínky a rady. Dále bych rád poděkoval Toru Norbyemu, autorovi modul GSF, za jeho ochotné zodpovězení několika mých dotazů, bez čehož bych si při vývoji aplikace zřejmě neporadil.

Samozřejmě můj dík patří také rodině a přátelům, a také všem lidem, kteří mne v průběhu vytváření diplomové práce podporovali, ať už odborným, či jakýmkoli jiným způsobem.

Shrnutí

Cílem práce bylo navrhnout a implementovat vývojové prostředí, které umožní snadný vývoj webových stránek pomocí šablon existujícího šablonovacího systému společnosti oXy Online. Vývojové prostředí mělo podporovat zvýraznění a kontrolu syntaxe, makra šablonovacího systému a automatické formátování, a zároveň mělo být snadno rozšiřitelné o další funkce. Jako základ byla použita platforma NetBeans a v kombinaci s nástrojem ANTLR byl vyvinut plugin do vývojového prostředí NetBeans IDE poskytující požadovanou funkcionalitu.

Klíčová slova

IDE, NetBeans, modul, ANTLR, oXy Online, šablonovací systém, gramatika, lexer, parser

Obsah

1 Úvod

Počítače patří neodmyslitelně k našemu každodennímu životu. Řídí činnosti různých zařízení, vykonávají nejrůznější úlohy a nacházejí se všude kolem nás - v mobilních telefonech, průmyslových robotech, letadlech, autech, nebo třeba automatických pračkách. Nejedná se tedy jen o osobní počítače, ale obecně je počítač stroj, který podle předem připravených instrukcí zpracovává data, tzn. vykonává určitý program. Počítačový program je posloupnost operací typicky zapsaná v nějakém programovacím jazyce a popisující realizaci určité úlohy.

1.1 Programovací jazyky

Původně byly programy realizovány buď přímo mechanickým strojem, elektrickými obvody, nebo zápisem strojového kódu přímo pro danou architekturu. S rostoucím potenciálem počítačů rostla také složitost řešených úloh a tím pádem také požadavky na produktivitu programátorů. Přestávaly proto stačit tyto prostředky zápisu programů a začaly se vyvíjet speciální jazyky usnadňující programátorům jejich práci. Nejdříve se jednalo o jazyk symbolických adres (assembler), který vlastně pouze poskytoval abstrakci adres instrukcí a umožňoval jejich zápis ve srozumitelném tvaru. Postupně se začaly vyvíjet tzv. vyšší programovací jazyky s vyšší úrovní abstrakce, často zaměřené na konkrétní problémovou doménu. To přinášelo jednak lepší srozumitelnost zdrojových kódů a také umožňovalo programátorům oprostit se od rutinních úkolů a zaměřit se pouze na řešení problému. Algoritmus je zapisován jazykovými prostředky daného programovacího jazyka a vzniká tak zdrojový kód programu. Ten je potom přeložen pomocí překladače. Podle způsobu překladu a spuštění rozlišujeme tyto druhy programovacích jazyků:

- **kompilované** – zdrojový kód je přeložen přímo do binární podoby závislé na architektuře a ta se potom distribuuje
- **interpretované** – distribuuje se zdrojový kód a jeho vykonání je zajištěno interpretem
- **hybridní** – zdrojový kód je přeložen do mezikódu nezávislém na cílové platformě, ten potom není vykonán přímo procesorem, ale virtuálním strojem

Toto rozdělení není absolutní a některé programovací jazyky není možné přesně zařadit. Mohou existovat v různých implementacích spadajících do odlišných kategorií. Existují také tzv. makrojazyky, což jsou programovací jazyky, které překládají své zdrojové kódy pouze do zdrojového kódu jiného programovacího jazyka. Ke kompilaci a spuštění kódu programu psaného v makrojazyce potom využijeme kompilátor nebo interpret cílového jazyka. Můžeme tak pohodlně řetězit programovací jazyky, k čemuž v praxi skutečně dochází (např. JSP → Java). Často také dochází ke kombinaci různých programových jazyků tak, že v určitých místech kódu hostujícího jazyka se může vyskytovat kus kódu jiného vloženého jazyka (např. HTML a JavaScript). Tato technika se nazývá *language embedding* (prokládání jazyků) a umožňuje delegování zpracování vložených jazyků na již existující nástroje pro tento jazyk.

1.2 Integrovaná vývojová prostředí

Od doby co se přesunulo programování od kleští na děrné štítky směrem k terminálům, začaly se spolu s programovacími jazyky vyvíjet také vývojová prostředí. Jejich úkolem bylo usnadňovat práci

s programovacími jazyky a jejich učení a osvojení, což mělo za následek zvýšení produktivity práce programátorů. Integrované vývojové prostředí (IDE[1]) je uživatelské rozhraní poskytující funkce pro vytváření, úpravu, kompilaci nebo interpretaci a často i distribuci a ladění programů. Nejprve se jednalo o prostředí řízené příkazy, ale postupně s vývojem grafických uživatelských rozhraní rostla i intuitivita vývojových prostředí. Moderní IDE obsahují celou řadu funkcí a nástrojů usnadňujících vývoj aplikací, jako například zvýrazňování syntaxe a sémantiky, automatické formátování, refaktorování, a nebo systém pro rychlý vývoj aplikací (RAD[2]) usnadňující vizuální návrh grafického uživatelského rozhraní. Často je poskytována podpora pro více fází vývoje než jen vlastní kódování a některá IDE také podporují více programovacích jazyků. Typická je modulární architektura umožňující snadnou rozšiřitelnost o další funkce.

1.3 Šablonovací systém společnosti oXy Online

Společnost oXy Online se zabývá zejména aplikacemi pro internetové obchodování a publikování na internetu. Je provozovatelem několika úspěšných internetových médií a cestovní kanceláře a také autorem redakčního systému oXyMedia a systému pro podporu internetového obchodování oXySHOP. Produkty založené na těchto řešeních jsou držiteli několika prestižních ocenění. Při vývoji systémů je využíváno osvědčených technologií, ale také probíhá vývoj technologií vlastních jako je například vlastní webový šablonovací systém. Tento šablonovací systém je proprietárním řešením firmy, a proto je tato diplomová práce psána tak, aby neodhalila podstatné části know-how, ale zároveň aby nastínila principy jak tento systém funguje.

Šablonovací systémy byly navrženy proto, aby umožnily efektivní oddělení aplikační logiky od prezentační vrstvy. Přirozeně by tyto vrstvy měly být odděleny, protože každá má za úkol něco jiného. Aplikační vrstva se stará zejména o získání dat a práci s nimi, zatímco prezentační vrstva slouží k zobrazení těchto dat uživateli. S takovýmto rozdělením by se mělo počítat už ve fázi návrhu aplikace a vlastní implementace by se potom tohoto návrhu měla držet. V praxi však bez použití šablonovacích systémů často docházelo k míchání těchto dvou vrstev dohromady, což s sebou neslo celou řadu problémů. Výsledný kód byl těžko (nebo vůbec ne) znovupoužitelný a díky tomu také těžko udržovatelný a náchylný k chybám. Použitím šablonovacích systémů je možné se takovýmto problémům vyhnout.

1.4 Cíle práce a technologie použité k dosažení těchto cílů

Cílem práce bylo vyvinout vývojové prostředí pro proprietární webový šablonovací systém společnosti oXy Online. IDE by mělo podporovat zvýraznění a kontrolu syntaxe, makra šablonovacího systému, automatické formátování a mělo by být navrženo tak, aby bylo snadno rozšiřitelné o další funkce. Jedním z hlavních požadavků byla také intuitivnost ovládání.

Jako prostředek k dosažení cíle byla zvolena rychle se rozvíjející platforma NetBeans Platform, na jejímž základu již několik vývojových prostředí existuje. Sama platforma vznikla důslednou modularizací vývojového prostředí NetBeans DeveloperX2 a na tomto základě byly v průběhu času vyvíjeny i další verze NetBeans IDE. S novými požadavky které s sebou vývoj těchto aplikací nesl, byla rozvíjena také sama platforma. A právě proto, že platforma je původně navržena a po celou dobu své existence konstruována jako základ vývojového prostředí, je zřejmé, že poskytuje prostředky k

1 Zkratka anglického výrazu „Integrated Development Environment“
2 Zkratka anglického výrazu *Rapid Application Development*

tvorbě integrovaných vývojových prostředí. Přesněji řečeno existují moduly do této platformy, které jsou součástí vývojového prostředí NetBeans IDE, a které je možné při tvorbě vývojového prostředí využít. V posledním roce se rozšiřování funkcionality NetBeans IDE mimo jiné soustředilo na podporu vytváření editorů pro nové programovací jazyky, čehož jsem ve své práci využil.

Dalším nástrojem, který byl při vývoji aplikace využit je generátor analyzátorů ANTLR. Tento nástroj umožňuje automaticky generovat zdrojový kód lexikálního a syntaktického analyzátoru podle bezkontextové vstupní gramatiky. Vstupní gramatika je zapisována ve speciálním jazyce, který je svou syntaxí podobný EBNF[3]. Současně je možné přímo do zdrojového souboru gramatiky vpisovat kusy kódu v jazyce Java a rozšířit tak funkce generovaných analyzátorů. Tímto způsobem je například možné nechat si ze vstupní gramatiky generovat syntaktický analyzátor, který zároveň plní funkci překladače.

Součástí práce je také modul, který zprostředkovává spolupráci obou výše zmíněných technologií a umožňuje aby jazykové analyzátory vytvořené za pomocí nástroje ANTLR mohly být použity pro podporu funkcí editoru daného jazyka implementovaného s využitím existujících modulů NetBeans IDE.

1.5 Poznámky k obsahu práce

Úvodní kapitola zasazuje tuto práci do širšího kontextu a poskytuje čtenáři základní přehled o tématech, kterých se tato práce týká, co je cílem této práce a jaké technologie byly použity k dosažení těchto cílů. Zároveň je zde také stručně nastíněn obsah dalších kapitol.

Kapitola 2 představuje nástroj ANTLR. Nejprve jsou v sekci 2.1 stručně zopakovány některé základní pojmy, které vytvářejí aparát nutný k popisu charakteristických vlastností ANTLR. Popisu těchto vlastností se dále postupně věnují sekce 2.2, 2.3 a 2.4 této kapitoly.

Kapitola 3 vysvětluje co je to šablonovací systém, a kde šablonovací systémy nachází své uplatnění. V sekci 3.1 jsou tyto systémy rozděleny do několika kategorií podle různých vlastností a popsány výhody a nevýhody jednotlivých typů systémů. Sekce 3.2 se věnuje popisu architektury MVC a použití šablonovacích systémů v této architektuře. Vlastnosti webového šablonovacího systému společnosti oXy online jsou stručně popsány v sekci 3.3, stejně jako důvody motivující ke vzniku tohoto systému.

Kapitola 4 je věnována popisu modulární platformy NetBeans. V sekci 4.1 je popsáno běhové prostředí, které je základem každé modulární aplikace založené na této platformě. Vlastnosti typické pro moduly jsou rozebrány v sekci 4.1.1 a jednotlivých podsekcích. Komunikace mezi moduly a další funkce, které běhové prostředí nabízí, jsou popsány v sekcích 4.1.2 a 4.1.3. Kromě základních modulů, které poskytuje běhové prostředí, je dále v sekci 4.2 popsáno také několik dalších užitečných modulů, které platforma poskytuje. Sekce 4.3 stručně zmiňuje možnost a důvody využití moduláních aplikací jako platforem pro vývoj dalších aplikací.

Kapitola 5 popisuje vlastní vytvoření vývojového prostředí. Sekce 5.1 se věnuje návrhu pluginu do NetBeans IDE, který přidává podporu pro jazyk šablon. Plugin se skládá z několika modulů a dva z těchto modulů jsou podrobně popsány v dalších sekcích páté kapitoly. Sekce 5.2 popisuje modul *AntlrGsfBridge*. Tento modul umožňuje využít analyzátory generované nástrojem ANTLR k podpoře

3 Zkratka anglického výrazu *Extended Backus-Naur Form* – notace používaná k standardizovanému zápisu bezkontextových gramatik

rozšířených funkcí editoru v NetBeans IDE a vytváří tak most mezi těmito dvěma technologiemi. Sekce 5.3 popisuje implementaci jednotlivých funkcí vývojového prostředí s využitím modulů, které poskytuje NetBeans IDE, a analyzátorů generovaných nástrojem ANTLR.

Poslední kapitola shrnuje stručně použitý postup a dosažené výsledky, a zároveň popisuje možnosti budoucího rozšíření vytvořeného pluginu.

2 ANTLR – ANother Tool for Language Recognition

Jak už název kapitoly napovídá ANTLR je nástroj pro rozpoznávání jazyků. Konkrétně se jedná o Javovskou implementaci nástroje pro generování lexikálních a syntaktických analyzátorů z bezkontextových gramatik. Zároveň je možné přímo k jednotlivým pravidlům vstupních gramatik přidávat *akce*, což jsou kusy zdrojového kódu jazyka Java. Tyto *akce* mohou jednak ovlivňovat běh a funkci generovaných analyzátorů, ale také mohou být využity k rozšíření funkcionality analyzátorů, například na překladače nebo interprety. Přestože samotný ANTLR je implementován v jazyce Java, je k dispozici možnost generovat analyzátory i pro jiné programovací jazyky. V současné době existují generátory pro jazyky Java, C, C++, C#, Python a JavaScript a je k dispozici API pro vytváření generátorů pro další jazyky. V této práci jsem pracoval s verzí ANTLR 3.0, která oproti verzím 2.x používá poněkud jinou syntaxi vstupních gramatik a nabízí užitečné funkce jako LL(*) analýza. Velkou výhodou verze 3 je také, že k vývoji gramatik je možné použít vývojové prostředí ANTLRWorks, které kromě editoru nabízí také interpret a debugger gramatik. Aktuálně je k dispozici ANTLR verze 3.1 a ANTLRWorks verze 1.2.2. Více informací je možno nalézt v dokumentaci[1].

Trocha teorie na úvod velice stručně připomene čtenáři základní pojmy. Znalost těchto pojmů je nezbytná k pochopení možností a omezení, které souvisí s použitím nástroje ANTLR. Pokud je čtenář seznámen s teorií formálních jazyků, může následující kapitolu přeskočit a pokračovat další kapitolou. Pokud je naopak text v následující kapitole příliš stručný a hutný, nezbývá než odkázat čtenáře na některý z podrobnějších zdrojů[2].

2.1 Formální jazyky a Chomského hierarchie tříd gramatik

Formální jazyk L je množina konečných slov (řetězců) nad určitou abecedou. Abeceda je konečná množina symbolů, ze kterých se mohou slova jazyka skládat. Existuje také speciální symbol, který není součástí abecedy, ale může být součástí jazyka. Tento symbol se většinou označuje jako ϵ a nazývá se prázdné slovo – tzn. slovo s nulovou délkou. Délka slova je počet symbolů abecedy, ze kterých se dané slovo skládá. Přestože abeceda i slova jsou konečné množiny, jazyk být konečný nemusí, protože délka slov nemusí být shora omezena. Proto není možné některé jazyky zapsat pouhým výčtem prvků jazyka, ale používají se k jejich konečné reprezentaci jiné metody. Mezi tyto metody patří definice jazyka pomocí následujících forem:

- Jazyk je množina všech slov, které je možné definovat danou formální gramatikou.
- Jazyk je množina všech slov, které vyhovují danému regulárnímu výrazu.
- Jazyk je množina všech slov akceptovaných daným automatem.

Síla těchto definic není stejná – konkrétně regulární výraz je nejslabší formou definice, protože je schopen rozpoznat pouze slova regulárního jazyka. Regulární jazyk je takový jazyk, který je možné zapsat pomocí regulární gramatiky. Přívlastek „regulární“ označuje množinu určitých vlastností daného typu gramatik (viz dále).

Gramatika G je struktura, která popisuje formální jazyk. Skládá se z množiny pravidel, pomocí nichž je možné předepsaným způsobem generovat všechna slova daného jazyka z předem daného

počátečního symbolu. Pravidla se mohou skládat z terminálních symbolů (symbolů abecedy) a neterminálních symbolů (jiných pravidel). Obecně jediná podmínka kladená na tvar pravidel je, že levá strana musí obsahovat alespoň jeden neterminál. Pravá strana pak obsahuje libovolný počet neterminálů a terminálů (tzn. třeba i žádné), nebo prázdné slovo. Generování slova jazyka probíhá tak, že na počáteční symbol je aplikováno jedno z pravidel gramatiky, čímž vznikne určitý řetězec. Tento řetězec může být složen z neterminálů i terminálů, a na tento řetězec může být dále opět aplikováno jedno z pravidel. Celý postup se opakuje tak dlouho, dokud není vygenerováno požadované slovo (slovo jazyka je složeno jen ze symbolů abecedy – tzn. vygenerovaný řetězec už neobsahuje žádný neterminální symbol). Tento postup se dá graficky znázornit pomocí tzv. derivačního stromu. Pokud existuje více možností jak generovat stejné slovo jazyka, nazývá se gramatika nejednoznačná. Následuje ukázka jednoduché gramatiky popisující jazyk obsahující všechna slova, která mají stejný počet písmen *a* a *b*:

G = ({S, A, B}, {a, b}, P, S), kde P je množina obsahující následující pravidla:

$S \rightarrow ABS$
$S \rightarrow \epsilon$
$AB \rightarrow BA$
$BA \rightarrow AB$
$A \rightarrow a$
$B \rightarrow b$

Jak je z ukázky patrné, gramatiku zapisujeme jako uspořádanou čtveřici s následujícími prvky:

- množina neterminálních symbolů
- množina terminálních symbolů (abeceda)
- množina pravidel
- počáteční symbol

Podle specifických vlastností je možné gramatiky rozdělit do 4 základních tříd (viz obrázek 1). Toto rozdělení se nazývá Chomského hierarchie. Jak je z obrázku patrné jednotlivé třídy nejsou diskrétně odděleny, ale naopak se jedná o soubor podmnožin. Je tedy zřejmé, že spadá-li gramatika do určité podtřídy, je zároveň také prvkem všech nadtříd této třídy. To znamená, že všechny gramatiky jsou typu 0, ale některé z nich jsou navíc kontextové. Některé kontextové gramatiky jsou zároveň gramatikami bezkontextovými, a některé bezkontextové dokonce gramatikami regulárními. Aby gramatika spadala do určité třídy, musí splňovat podmínky, které jsou pro gramatiky tohoto typu stanoveny. Rozdělení gramatik do těchto tříd není samoúčelné, ale udává výpočetní sílu daného typu gramatiky. Následuje výčet jednotlivých tříd gramatik, omezení, která jsou kladena na pravidla gramatik daného typu, a specifické vlastnosti těchto gramatik:

Obrázek 1: Chomského hierarchie tříd jazyků

- **Typ 0 (frázové gramatiky)** – každá gramatika je gramatikou typu 0. Na tvar pravidel gramatiky se nekladou žádná omezení.
- **Typ 1 (kontextové gramatiky)** – pro všechna pravidla $\alpha \rightarrow \beta$ tohoto typu gramatik musí platit, že $|\alpha| \leq |\beta|$. Výjimku tvoří pravidlo $S \rightarrow \epsilon$, které se v gramatikách toho to typu může také vyskytovat, ale pouze za podmínky že se neterminál S nevyskytuje na pravé straně žádného z pravidel.
- **Typ 2 (bezkontextové gramatiky)** – všechna pravidla tohoto typu gramatik musí být ve tvaru $A \rightarrow \alpha$, kde $|\alpha| \geq 1$. Eventuálně se opět může vyskytovat pravidlo $S \rightarrow \epsilon$ za stejné

podmínky jako u gramatik typu 1. Ke gramatikám tohoto typu je možné sestrojit ekvivalentní zásobníkový automat[4].

- **Typ 3 (regulární gramatiky)** – gramatiky tohoto typu mohou obsahovat pouze pravidla tvaru $A \rightarrow aB$ a $A \rightarrow a$ (tzv. levé lineární gramatiky) nebo $A \rightarrow Ba$ a $A \rightarrow a$ (tzv. pravé lineární gramatiky). Ke gramatikám tohoto typu je možné sestrojit ekvivalentní konečný automat[5].

K této hierarchii existuje také příslušná hierarchie jazyků – tzn. regulární jazyk je možné zapsat pomocí regulární gramatiky a podobně. Pro úplnost ještě uveďme, že existují také jazyky, které nejsou žádného z uvedených typů a tyto jazyky není možné zapsat pomocí gramatiky.

2.2 Analýza jazyka pomocí ANTLR

Jak již bylo řečeno v úvodní části této kapitoly, ANTLR je nástroj pro rozpoznávání bezkontextových jazyků. Vlastní rozpoznávání probíhá v několika fázích, jak je znázorněno na obrázku , a provádí jej analyzátory jazyka, které jsou automaticky generovány ze vstupní gramatiky. Gramatika se skládá z jednoho nebo více fyzických souborů a je možné ji rozdělit na několik logických částí, z nichž každá odpovídá jednomu analyzátoru. Následuje stručný popis vstupů, výstupů a funkcí jednotlivých analyzátorů:

- **Lexikální analyzátor** bývá též nazýván *lexer* nebo *scanner*. Vstupem tohoto analyzátoru je proud znaků ze vstupního souboru. Vstupní znaky soustřeďuje podle pravidel předepsaných gramatikou pro tento analyzátor do jednotlivých skupin (*lexémy*) a tyto skupiny podle jejich významu označuje nálepkami (*tokeny*). Zároveň může provádět kontrolu lexikální správnosti vstupního souboru, případně i odfiltrování a zahození nerelevantních nebo chybných znaků. Výstupem úspěšné analýzy je proud tokenů.
- **Syntaktický analyzátor** na vstup dostane proud tokenů a ty považuje za terminály (tzn. atomické jednotky). Jinými slovy pravidla gramatiky tohoto analyzátoru pracují s tokeny a právě tokeny se vyskytují v listech derivačního stromu. Úkolem tohoto analyzátoru je ověřit, jestli sekvence tokenů na vstupu odpovídá struktuře, kterou definuje gramatika – tzn. syntaktickou správnost. Výstupem úspěšné analýzy bývá abstraktní syntaktický strom, což je derivační strom vzniklý při analýze (může být i různým způsobem upravený, k čemuž jazyk

Obrázek 2: Analýzy jazyka pomocí ANTLR

4 Přesnou definici pojmu *zásobníkový automat* je možné nalézt např. v [2] str. 76
5 Přesnou definici pojmu *konečný automat* je možné najít např. v [2] str. 9

gramatiky poskytuje prostředky). Jiný běžně používaný název syntaktického analyzátoru je *parser*[6].

- **Analyzátor AST**[7] dostává na vstup abstraktní syntaktický strom a jeho hlavním úkolem je procházet tento strom a různě jej podle předepsaných pravidel transformovat nebo spouštět *akce* v uzlech stromu. Výstupem potom může být jiná stromová reprezentace.

Z uvedeného popisu je patrné, že všechny popsané analyzátory vykonávají obdobnou funkcionalitu, jen na různé úrovni abstrakce. Analýza je typu shora-dolů (*LL(*)* viz dále), tzn. začíná se u počátečního symbolu a pomocí pravidel gramatiky se pokoušíme vygenerovat vstupní text.

U syntaktického analyzátoru a analyzátoru AST nemusí být důležitý výstupní strom, ale v některých případech stačí fakt, jestli analýza proběhla úspěšně či nikoli. V některých případech také nemusí nutně probíhat všechny tři fáze analýzy, ale může stačit jen výsledek některé z nich, což je na obrázku znázorněno čárkovanými šipkami. Lexikální analýza je provedena vždy, přestože v některých případech může být triviální (znaky nejsou shlukovány do skupin, ale každý znak má svůj lexém). ANTLR díky svým *akcím* také nabízí dost silný aparát k tomu, aby byly analyzátory různým způsobem upravovány a rozšiřována jejich funkcionalita. Je proto možné, že u některých generovaných analyzátorů nejsou hranice jejich působnosti tak striktně odděleny jak je popsáno výše.

2.3 LL(*) analýza

Klasické analyzátory shora-dolů bývají typu *LL(1)* nebo *LL(k)*, kde *k* bývá malé celé číslo udávající velikost *lookahead bufferu* analyzátoru. Tento buffer má načteno vždy právě *k* symbolů dopředu – tzn. symbolů které se teprve v následujících *k* krocích dostanou na čtecí hlavu analyzátoru. Pokud analyzátor při konstrukci derivačního stromu narazí na místo, ve kterém není jednoznačné jaké pravidlo pro derivaci použít, má možnost nahlédnout do tohoto bufferu, a ten by měl poskytnout dostatek informací ke správnému rozhodnutí. V praxi bývá toto rozhodování realizováno pomocí konečných stavových automatů. Klasické *LL(k)* analyzátory používají acyklické automaty (jejich graf neobsahuje kružnici), kde *k* je nejdelší možná cesta od počátečního do koncového stavu automatu. ANTLR však pro rozhodování používá automat, který ve svém grafu může obsahovat kružnici. Délka nejdelší cesty proto není fixně daná, tzn. *k* může mít neomezenou hodnotu. Proto se tyto analyzátory označují jako *LL(*)*.

Rozdíl mezi *LL(k)* a *LL(*)* analyzátory si můžeme představit personifikací analyzátoru do jakéhosi cestovatele, který došel na rozcestí a rozhoduje se kterou cestou půjde dál. *LL(k)* cestovatel vytáhne dalekohled a podívá se, kam cesty vedou. Nikdy však nedohlédne za horizont (*k*) a může se tak stát, že nepozná správnou cestu (*k* je moc malé). Naproti tomu *LL(*)* cestovatel je chytřejší a místo dalekohledu vyšle každou cestou průzkumníka. Průzkumníci zajdou i za horizont a tak mají větší šanci rozpoznat správnou cestu.

2.4 Omezení a problémy

ANTLR je nástroj, který pracuje s bezkontextovými gramatikami. Jazyk, který chceme analyzovat, ale nemusí být bezkontextový, a v takovém případě se můžeme dostat do problémů. Bezkontextová gramatika potom nenabízí dostatečný aparát k popisu takového jazyka. V praxi se tento

6 Z anglického *to parse* - analyzovat

7 Zkratka anglického výrazu *Abstract Syntax Tree*

problém řeší tak, že se vytvoří analyzátor, který rozpozná nadmnožinu zkoumaného jazyka – tzn. přijme i některé slova, která do jazyka nepatří. Tato slova jsou potom odstraněna při následné další analýze (např. založené na procházení syntaktického stromu).

Další problémy mohou nastat u nejednoznačných gramatik. Jak již bylo řečeno, některé jazyky bohužel nelze jednoznačně zapsat pomocí bezkontextových gramatik. ANTLR sice umožňuje rozpoznávat i nejednoznačné gramatiky, ale nese to s sebou určité riziko. Pro každé problematické místo gramatiky je totiž vytvořen zvláštní automat, který určuje správné rozhodnutí. To má za následek nejen to, že se zdrojové kódy generovaných analyzátorů nepříjemně zvětší, ale hlavně to, že metody obsahující přechodové funkce složitých automatů mohou přesáhnout pevné limity dané specifikací virtuálního stroje Java[3]. Je proto vhodné omezit počet míst, kde je gramatika nejednoznačná pokud možno na minimum, což může být někdy na úkor přehlednosti a čitelnosti.

Další problém nastává při použití rekurzivních pravidel u nejednoznačných gramatik – konečný automat totiž rozpoznává pouze regulární gramatiky, což je v rozporu s rekurzí pravidel. Toto omezení je možné obejít pomocí tzv. sémantického predikátu. Sémantický predikát je zvláštní druh *akce*, která předchází problematickým pravidlům. Jedná se vlastně o kus kódu v jazyce Java, který provede rozhodování místo zmíněného automatu (jenž by byl v tomto případě nepoužitelný).

3 Šablonovací systémy

Šablonovací systém je nástroj pracující se šablonami. Šablona je specifický typ dokumentu (zdrojový kód) obsahující speciální značky, které jsou určitým způsobem vyhodnocovány a nahrazovány šablonovacím systémem. Výsledkem je výstupní dokument. Jak je patrné ze schématu (obrázek 3), šablonovací systém je tedy jakýsi druh překladače, který s využitím dodaných dat provádí překlady šablon na výstupní dokumenty.

Obrázek 3: Schéma funkce šablonovacího systému

Tím je tedy dáno oddělení prezentační vrstvy, která je realizována šablonou, od aplikační vrstvy, která je zprostředkovatelem dat. Striktnost tohoto rozdělení je dána přímo konkrétním šablonovacím systémem a u různých systémů se liší. Je potřeba si uvědomit, že i v prezentační vrstvě se může vyskytovat programový kód, ovšem pouze pro prezentační účely – tzv. prezentační logika. Typickým příkladem je barvení sudých a lichých řádků v tabulkách různými barvami. Tento problém řeší různé systémy různě. Existují velice jednoduché systémy (jako například P. E. T.), které v šablonách nepodporují prakticky žádnou prezentační logiku. U takovýchto systémů musí být prezentační logika realizována v aplikaci. V takovém případě je vhodné prezentační logiku nějakým způsobem oddělit od aplikační, což je ovšem mimo rámec šablonovacího systému. Na druhou stranu existují také rozsáhlé a funkčně bohaté systémy, které podporují podmínky a vyhodnocování výrazů, a umožňují tak řešit prezentační logiku přímo v šablonách a plně tak realizovat prezentační vrstvu pomocí šablon. Nevýhodou tohoto řešení je možnost zneužít těchto funkcí a tak vložit do šablony i kus kódu, který nepatří do prezentační logiky (například nějaký výpočet nad daty). Obecně tedy nelze říct, že by použití šablonovacího systému vyřešilo oddělení prezentační a aplikační vrstvy za nás, ale výrazným způsobem nám tento úkol usnadní.

Další věcí, kterou použití šablonovacího systému usnadní, je vymezení rolí při vývoji a správě projektu. To do značné míry souvisí s předchozím rozdělením aplikace do vrstev. U aplikací kde tyto vrstvy nejsou odděleny se totiž často stává, že designér a programátor musí editovat stejný soubor, přičemž každý upravuje jen jemu příslušející část souboru. Vzniká tak riziko zavlečení nechtěných chyb při náhodné editaci špatné části kódu. Použití šablonovacího systému usnadňuje separaci těchto

rolí. O vývoj a správu šablon se starají výhradně designéři, zatímco o aplikační vrstvu výhradně programátoři. Je tak umožněna efektivní spolupráce mezi těmito dvěma rolemi bez obav z toho, že by si navzájem doslova „lezly do zelí“.

Oddělení vrstev s sebou přináší ještě několik dalších výhod jako je snížení počtu duplicit v kódu a s tím související jednodušší a levnější údržbu, nebo třeba snadnou lokalizaci, protože lokalizace a internacionalizace je záležitostí pouze prezentační vrstvy.

Většina šablonovacích systémů také nějakým způsobem řeší automatické escapování speciálních a řídících znaků. To má za následek nejen správné zobrazení těchto znaků, ale také zamezení XSS[8] útokům.

3.1 Typy šablonovacích systémů

Pro dělení šablonovacích systémů není dán žádný všeobecný předpis nebo norma. Existuje však několik specifických vlastností, podle kterých by se tyto nástroje kategorizovat daly.

Podle způsobu předávání dat šablonám můžeme dělit šablonovací systémy na dva modely:

- **Push model** – všechna data jsou předána šabloně před jejím vypsáním Šablona neprovádí žádné vyhodnocování – veškeré potřebné vyhodnocování je provedeno apriori před předáním dat šabloně.
- **Pull model** – sama šablona si určí jaká data potřebuje a ta si ve vhodnou chvíli „natáhne“ z modelu.

Podle zařazení prezentační logiky:

- **Prezentační logika v aplikaci** – šablony nenabízejí způsob jak implementovat prezentační logiku, ta proto musí být implementována v aplikaci. Šablony tak nepokrývají veškerou funkcionalitu, která je v prezentační vrstvě potřebná a část prezentační vrstvy je v aplikaci. Aby bylo zachováno oddělení vrstev, je proto dobré prezentační logiku od aplikace nějakým způsobem oddělit (vlastní API, pomocné objekty a podobně).
- **Prezentační logika v šablonách** – vyjadřovací schopnost šablon je natolik silná, že veškerá prezentační logika může být implementována přímo v šablonách. Je však potřeba dbát na to, aby šablony nebyly zneužívány k jiným než prezentačním účelům.

Podle API šablonovacího systému:

- **„Pinhole“ API** – celá šablona je reprezentována jako jeden jediný objekt.
- **DOM API** – šablona je analyzována DOM parserem a je reprezentována jako strom objektů reprezentujících jednotlivé elementy šablony. S tímto stromem je možné manipulovat pomocí DOM API.
- **Komponentová API** – každá značka, která se nachází v kódu šablony, je reprezentována vlastním objektem (komponentou).

Podle jazyka šablon:

8 *Cross Site Scripting* – metoda narušení internetových stránek využitím bezpečnostních děr ve skriptech stránek

- **Šablony využívají čistě již existující jazyk** – například jazyk (X)HTML nabízí dost prostředků k tomu, aby šablony byly zároveň validní (X)HTML dokumenty a zároveň poskytovaly dost prostoru pro manipulaci. Pro vyhodnocování je možné využívat například hodnot atributu *id* jednotlivých elementů, využívat vlastní jmenný prostor s vlastními atributy nebo elementy, nebo zapisovat speciální značky do komentářů. Výhodou je možnost použití existujících vývojových nástrojů a možnost okamžitého náhledu šablony (třeba i s ukázkovým obsahem) v prohlížeči.
- **Šablony využívají existující jazyk a rozšiřují jej o nové značky** – například jazyky JSP nebo PHP, které využívají jazyk (X)HTML, a rozšiřují jeho syntaxi o speciální skriptovací značky. Výhodou je, že tvůrci šablon zpravidla existující jazyk už znají a tak se musí učit jen rozšiřující prvky jazyka nového.
- **Šablony mají definován vlastní jazyk** – šablony nejsou založeny na žádném existujícím jazyce, ale mají definován jazyk vlastní. Výhodou je, že tento jazyk bývá přesně šitý na míru potřebám a účelu šablon.

Další vlastnosti bývají specifické pro jednotlivé šablonovací systémy. Některé bývají implementovány jako samostatné knihovny, nebo jako pluginy pro servery a distribuovány jako již zkompilovaný binární kód. To s sebou většinou nese výhodu vysokého výkonu těchto systémů při nižších hardwarových nárocích. Takovéto systémy mohou nabízet API i v jiném jazyce než jsou samy implementovány a některé také nabízejí API pro více různých programovacích jazyků. Nevýhodou je ovšem závislost na platformě a někdy také na použitém serveru nebo provozovateli (jestli je zde daný plugin k dispozici, případně jestli je možné jej doinstalovat). Jindy je systém implementován a distribuován jen jako kus skriptu, který je potřeba nějakým způsobem importovat do našeho projektu. V tomto případě může být limitující závislost na stejném programovacím jazyce, ve kterém je šablonovací systém implementován, a také potencionálně nižší výkon, případně zvýšená systémová zátěž.

Každé řešení má svá pro a proti a není možné obecně říci, že by některé bylo lepší nebo horší. Každý projekt má totiž své specifické nároky a pro různé projekty může být vhodné použít různý typ šablonovacího systému. Je tedy jen na uživateli, jestli si z bohaté nabídky hotových šablonovacích systémů vybral ten, který jeho potřebám nejlépe vyhovuje, nebo jestli si vytvoří vlastní systém přesně šitý na míru svým potřebám, jako to udělala společnost oXy Online.

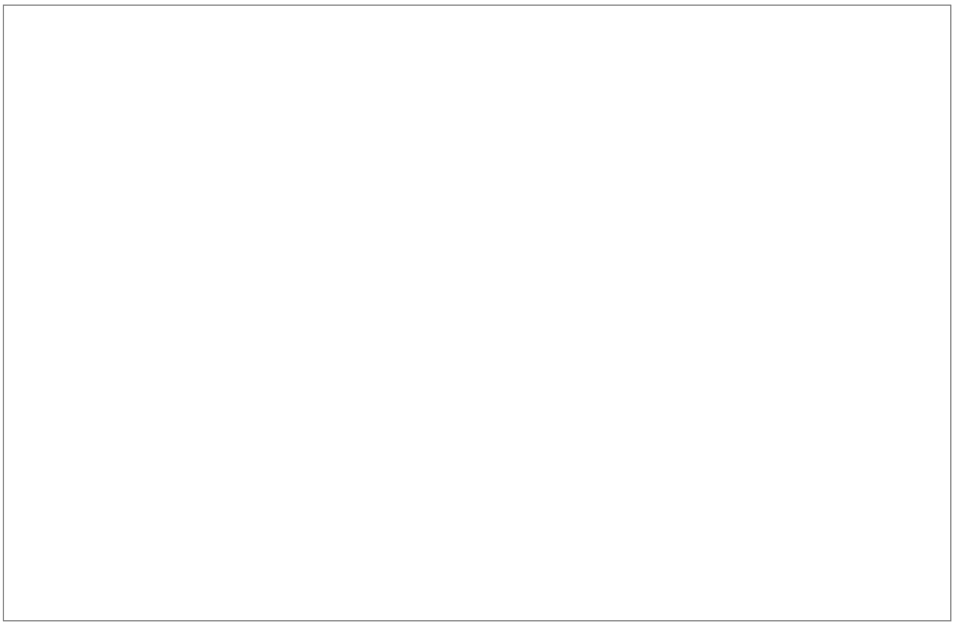
3.2 Agregační návrhový vzor Model-View-Controller a šablonovací systémy

Návrhový vzor Model-View-Controller rozděluje datový model, uživatelské rozhraní a řídící logiku aplikace do tří nezávislých komponent (vrstev). Tyto komponenty spolu navzájem spolupracují, ale modifikace kterékoliv z nich má minimální vliv na funkci ostatních komponent. MVC (Model-View-Controller) bývá chápán nejen jako návrhový vzor, ale také jako vzor architektonický, protože ovlivňuje návrh aplikace v mnohem globálnějším měřítku než klasické návrhové vzory. Pojem *agregační* vyjadřuje, že samotný návrhový vzor je složen z několika různých jednodušších návrhových vzorů, jak bude popsáno dále. Jednotlivé výrazy *Model*, *View* a *Controller* vyjadřují činnost kterou má daná část systému na starosti a do češtiny by se daly přeložit jako *model*, *pohled* a *řadič* (nebo spíše *správce*), v dalším textu této práce však považuji tyto výrazy za terminus technicus a nepoužívám jejich český překlad. Pokud si představíme aplikaci jako jednoduchý systém složený z procesů *Vstup*

→ *Zpracování* → *Výstup*, pak těmto procesům odpovídají komponenty *Controller* → *Model* → *View*. Právě díky tomuto faktu může být tento návrhový vzor aplikován i rekurzivně pro návrh jednotlivých subsystémů. MVC bývá někdy označován jako tzv. *Model-2*. Číslovka 2 v názvu napovídá, že existuje také vzor *Model-1*, který je jednodušší a rozděluje aplikaci pouze na dvě vrstvy – datový model (1. vrstva) a uživatelské rozhraní s řídící logikou (2. vrstva). Následuje stručný popis jednotlivých komponent MVC:

- **Model** – patří do aplikační vrstvy a reprezentuje datovou logiku systému. V některých systémech zapouzdřuje také aplikační logiku. Často se skládá z menších znovupoužitelných komponent, je proto dobré jej od zbytku systému oddělit ucelenou sadou rozhraní. K tomuto účelu bývá využíván návrhový vzor *facade* (*fasáda*).
- **View** – realizuje prezentační vrstvu a obsahuje metody, jak zobrazit data, která reprezentuje model danému uživateli. Často také umožňuje jeden model interpretovat uživateli různými způsoby. Bývají zde využívány návrhové vzory *value object (hodnota)* a *view helper (pomocník)*.
- **Controller** – tvoří pojítko mezi *modelem* a *view* a zajišťuje jak spolupráci těchto dvou vrstev, tak zároveň redukuje závislosti mezi nimi. Hlavním úkolem je zpracování událostí, které vznikají v prezentační vrstvě a následné vyvolání odpovídajících akcí v aplikační vrstvě a naplnění modelu daty. K vyvolání akcí bývá využíván návrhový vzor *factory method (tovární metoda)* a samotné akce bývají aplikací návrhového vzoru *command* (*požadavek*).

Důležitým použitým návrhovým vzorem je také *observer (pozorovatel)* – *model* představuje pozorovaný objekt a *view* a *controller* jsou pozorovatelé tohoto objektu. Když potom nastane změna v modelu, pozorovatelé jsou o této změně informováni.



Obrázek 4: Schéma komponent Model-View-Controller

Co se týče použití šablonovacích systémů v architektuře MVC, jejich operační pole je právě komponenta *view*. Jak ukazuje obrázek 3, šablonovací systém zpracovává poskytnutá data a podle

určité šablony z nich vytvoří výstupní dokument. Funkci takového systému si potom můžeme představit podle následujícího scénáře. Uživatel provede nějakou akci a tu dostane ke zpracování *controller*. Ten provede aktualizaci *modelu* a rozhodne, kterou šablonu použít a tyto informace předá do *view*. Komponenta *view* je realizována pomocí šablonovacího systému. Šablonovací systém tedy dostane na vstup šablonu, kterou má použít, a data reprezentovaná *modelem* a může tak vytvořit výstupní dokument.

V praxi se konceptu MVC využívá zejména u webových aplikací a podpora tohoto modelu bývá implementována v různých aplikačních frameworcích.

3.3 Webový šablonovací systém společnosti oXy Online

Motivací k vytvoření tohoto šablonovacího systému bylo vyhovění několika požadavkům, kterým žádný existující šablonovací systém neodpovídal. Jednotlivým dílčím požadavků sice některé existující systémy vyhovovaly, neexistoval ale systém, který by vyhovoval všem těchto požadavků zároveň. Jedním z požadavků bylo použití JavaScriptu běžícího na serveru, protože je známý široké komunitě lidí, ověřený praxí, a je distribuován společně s Javou, a Java je jeho kontejnerem. Dále byla požadovány makra a inkluze a syntaxe podobná jazyku JSP.

S ohledem na rozdělení popsané v předchozí kapitole, by se dalo šablonovací systém společnosti oXy Online popsat jako systém založený na kombinaci *push* i *pull modelu* s podporou prezentační logiky v šablonách. Systém je implementován v jazyce Java a k dispozici jako JAR archiv s *pinhole API* pro jazyk Java. Šablony mají definován vlastní jazyk, který je svou syntaxí podobný jazyku HTML, respektive JSP, a rozšiřuje tuto syntaxi o další značky. Gramatika specifikující jazyk šablon je popsána v kapitole 3.2.2. Samotný šablonovací systém pracuje ve 2 fázích, kde první je překlad šablony a druhá provedení šablony s daným modelem. Princip funkce překladače je nastíněn v kapitole 3.2.3 a výstupem překladače je zkompilovaná šablona, což je objekt implementující rozhraní *Template*. Toto rozhraní předepisuje metodu *execute*, které je jako parametr předán model a jejímž voláním je možné šablonu provést. Implementace šablonovacího systému je majetkem společnosti oXy Online a není předmětem této práce.

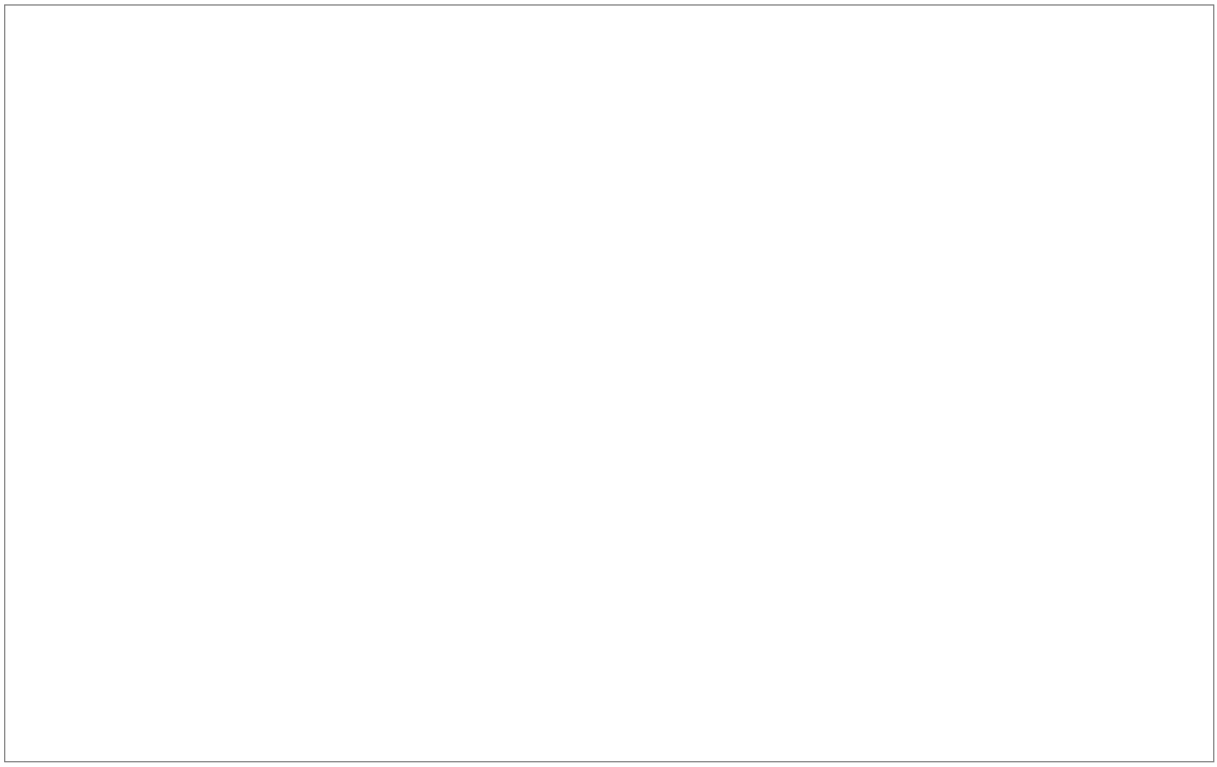
3.3.1 Specifikace jazyka šablon

Jak již bylo řečeno v úvodu této kapitoly, jazyk šablon je založen na jazyce HTML a rozšiřuje jeho syntaxi o další značky. Jedním dechem je ale potřeba dodat, že se nejedná o značkovací jazyk založený na bázi SGML[9], přestože je svou syntaxí těmto jazykům velice podobný. Na rozdíl od jazyků rodiny SGML tedy gramatika není popsána pomocí DTD a není možné dokumenty analyzovat pomocí dostupných SGML parserů. Jazyk šablon je popsán pomocí gramatiky pro nástroj ANTLR a s pomocí tohoto nástroje je možné potřebné analyzátory generovat (viz kapitola 2.2). Veškeré jazykové konstrukce stručně představuje následující seznam a názorně prezentuje obrázek 5.

- **HTML** – do šablon je možné psát libovolný HTML kód a vše, co k tomu patří, tedy například vložené CSS a klientské skripty. Jediné omezení v tomto směru je to, že nejde použít jmenný prostor *„m“*, protože tento jmenný prostor je vyhrazen pouze pro makra šablonovacího systému.

9 Zkratka anglického výrazu *Standard Generalized Markup Language* – univerzální značkovací metajazyk umožňující definovat značkovací jazyky jako své vlastní podmnožiny

- **Direktivy** – direktivy nevytváří žádný viditelný výstup, ale jsou to instrukce pro překladač šablony. Zapisují se mezi značky „*<%@*“ a „*%>*“. Název direktivy je složen z písmen, číslic nebo některých speciálních znaků, a nesmí obsahovat mezery. Direktiva může mít jeden nebo více parametrů uzavřených do uvozovek.
- **Skriptlety** – jsou to kusy JavaScriptového kódu a zapisují se mezi značky „*<%*“ a „*%>*“. Jejich použitím je možné realizovat prezentační logiku.
- **Výrazy** – jsou taky kusy JavaScriptového kódu, ale na rozdíl od skriptletů se výsledek jejich vyhodnocení přímo vloží na patřičné místo šablony, respektive na odpovídající místo výstupního dokumentu, který vznikne ze šablony.
- **Makra** – jsou speciální značky, které umožňují budovat šablonu pomocí předem připravených kusů kódu nebo jiných šablon deklarativním způsobem. Zapisují se podobným způsobem jako SGML element v jmenném prostoru „*m*“, tzn. jako značka začínající „*m:*“. Oproti syntaxi SGML značek se však liší v tom, že hodnoty atributů maker se mohou zapisovat pouze mezi uvozovky. V těchto hodnotách se navíc neescapují speciální znaky a nesmí se vyskytovat znak uvozovka. Název makra se může stejně jako název direktivy skládat z písmen, číslic a některých speciálních znaků. Stejně jako u SGML značek, také makra se mohou vyskytovat jako značka s obsahem (např. „*<m:mojeMakro> obsah </m:mojeMakro>*“) nebo jako prázdná značka (např. „*<m:mojeMakro/>*“).



Obrázek 5: Ukázka kódu šablony a nové prvky jazyka šablon

Gramatika jazyka šablon se skládá ze dvou souborů – první popisuje lexikální konstrukty jazyka a je podle něj vytvořen lexikální analyzátor, druhý popisuje syntaktickou stránku jazyka a je podle něj

vytvořen syntaktický analyzátor jazyka. Konkrétní implementace těchto gramatik jsou navrženy tak, aby umožňovaly vytvoření překladače rozšířením gramatiky syntaktického analyzátoru pomocí akcí (viz kapitola 3.2.3), a také aby analyzovaly jazyk na vhodné elementy a poskytovaly tak dobrou podporu pro zvýrazňování kódu v editoru (viz kapitola 5.3.3).

Lexikální analýza je v případě jazyka šablon tou složitější částí analýzy. V kódu šablon se totiž mohou vyskytovat na různých místech stejné znaky, ale s různým významem. Například znak „<“ může být porovnávací operátor v JavaScriptovém kódu, nebo může být součástí tokenu, který značí začátek skriptu, nebo může být součástí HTML tagu. K jakému tokenu tento symbol patří je proto potřeba určit podle kontextu, ve kterém se takovýto problematický znak vyskytuje. Jak již ale bylo řečeno dříve, ANTLR umožňuje pracovat pouze s bezkontextovými gramatikami. Tento problém je možné vyřešit vhodným použitím akcí a sémantických predikátů, a tak si chybějící informaci o kontextu nahradit. Následuje zkrácená ukázka gramatiky lexikálního analyzátoru (kde je toto řešení použito) a stručný popis jednotlivých sekcí gramatiky. Kompletní gramatika lexikálního analyzátoru je k dispozici na přiloženém CD. Více informací o jazyku pro zápis gramatik je možné získat z dokumentace – viz [1] sekce „Grammars“.

```
lexer grammar OxyTemplateLexer;
...
@members {
public boolean inMacro = false;
...
public boolean isMacroStartAhead() {
    if (input.LA(1) == '<') {
        if (input.LA(2) == 'm') {
            return (input.LA(3) == ':');
        } else if (input.LA(2) == '/') {
            return ((input.LA(3) == 'm') && (input.LA(4) ==
':'));
        }
    }
    return false;
}
...
LT  :
    {isMacroStartAhead()}?=> '<' {inMacro = true;};
GT :
    {inMacro}?=> '>' {inMacro = false;};
...
HOST_LANGUAGE :
    ({!inMacro && !inBlock && !inDirective && !inExpression
&& !isMacroStartAhead() && !isBlockStartAhead() }?=> .)+;
```

Klíčové slovo ***lexer*** udává, že se jedná o gramatiku pro lexikální analyzátor.
V sekci ***@members{...}*** je možné deklarovat vlastní proměnné a metody v jazyce Java.
Konstrukce ***{...}?***=> se nazývá *sémantický predikát* a jedná se o kód v jazyce Java, který rozhodne o tom, jestli se následující pravidlo provede, či nikoliv.
Konstrukce ***{...}*** zapsaná na konci pravidla se nazývá *akce* a jedná se o kód v jazyce Java, který se vykoná po provedení daného pravidla. Akce je možné psát i na jiná místa gramatiky.

Z předchozí ukázky je patrné, že pokud se v kódu šablony vyskytuje řetězec „*<m:*“ nebo „*</m:*“, pak je znak „<“ rozpoznán jako token *LT*. Pro ilustraci je také uvedeno pravidlo rozpoznávající hostitelský jazyk šablony (HTML). Toto pravidlo se skládá z anonymního subpravidla uzavřeného do závorek. Toto subpravidlo nejprve ověří, jestli nebyl dříve rozpoznán a nastaven žádný speciální kontext, a potom zkonzumuje libovolný jeden znak reprezentovaný tečkou (tedy i znak „<“). Znak plus udává, že se toto subpravidlo může opakovat vícekrát, nejméně však jednou. Všechny zkonzumované znaky jsou nakonec společně označeny jako token *HOST_LANGUAGE*. Aplikací

tohoto postupu je tedy možné stejné znaky v různém kontextu rozpoznávat jako různé tokeny. Pro úplnost je nutno dodat, že toto není jediný způsob jak je možné takovouto situaci řešit – v některých jednoduchých případech stačí zkoumat „dopředný“ kontext, k čemuž poslouží dostatečný *lookahead* nebo *LL(*)* analýza (viz kapitola 2.3). Místo sémantického predikátu je taky možné použít slabší syntaktický predikát. Opět se jedná o jakousi rozhodovací metodu, ale v tomto případě se nezapisuje v jazyce Java, ale stejně jako pravá strana pravidla gramatiky a je uvedena konstrukcí *(...)*?=>. ANTLR potom při zpracování gramatiky pro každý syntaktický predikát vytvoří stavový automat, který dané pravidlo rozpoznává (což jsme u sémantického predikátu de facto udělali ručně např. v metodě *isMacroStartAhead* v předchozí ukázce). Použití syntaktického predikátu s sebou však nese rizika popsaná v kapitole 2.4. Podle mého názoru je navíc přehlednější a čitelnější použití sémantických predikátů a metod s výmluvnými názvy, obzvláště pak u složitých podmínek.

Gramatika popisující syntaktickou stránku jazyka je o mnoho jednodušší a díky dobře připraveným tokenům z lexikální analýzy si vystačí jen s jednoduchými pravidly bez predikátů a akcí. Díky *LL(*)* analýze kterou ANTLR poskytuje je navíc možné zapisovat pravidla velice čitelně jako je tomu v následující ukázce. Protože se subpravidlo *(S+ macroAttribute)* může v pravidlech *macroStart* a *emptyMacro* opakovat vícekrát, nebylo by při *LL(k)* analýze s fixní hodnotou *lookahead* možné rozeznat, o které z těchto dvou pravidel se jedná (viz kapitola 2.3). Kompletní gramatika syntaktického analyzátoru je k dispozici na přiloženém CD.

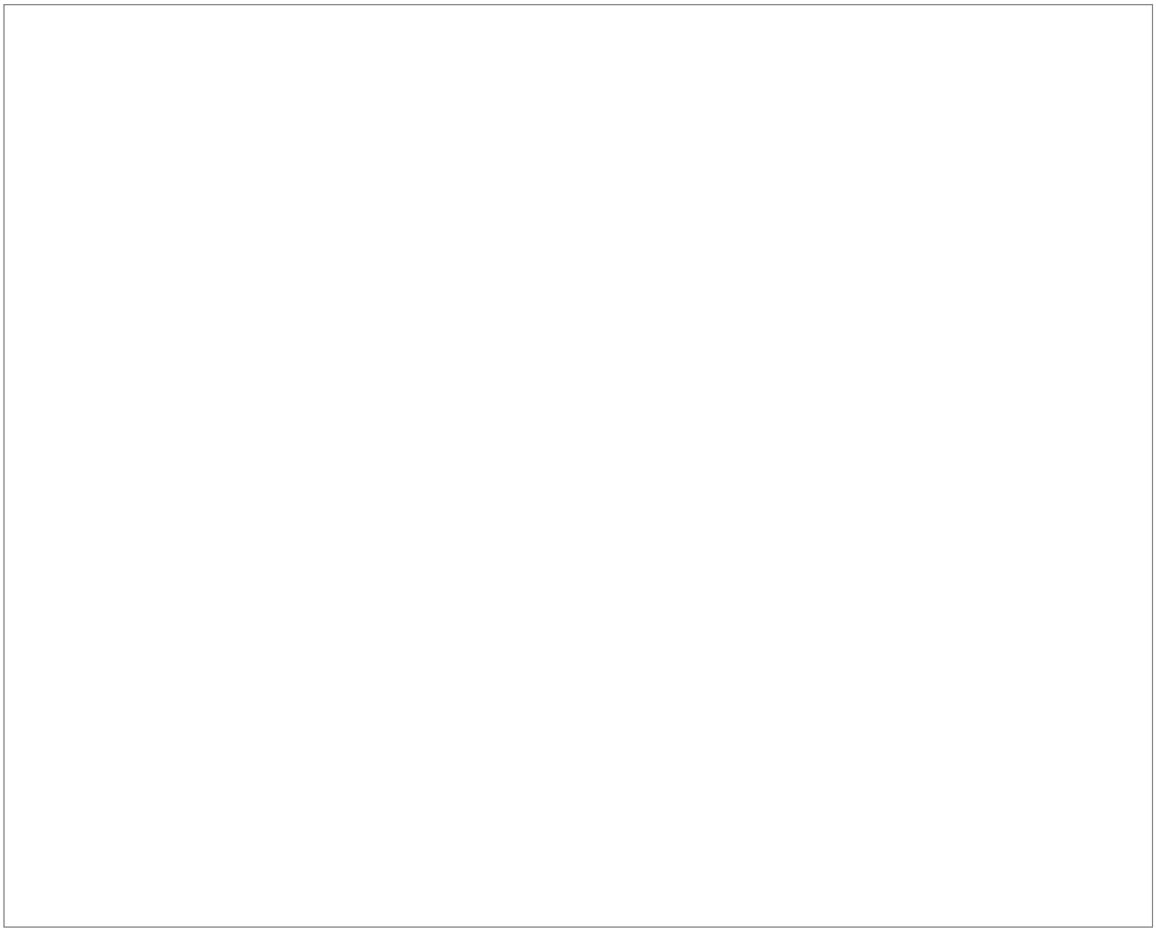
```
parser grammar OxyTemplateParser;

options {
    tokenVocab=OxyTemplateLexer;
}

template:
    ( macroStart
    | emptyMacro
    ...
    )* EOF;

macroStart:
    LT macroId (S+ macroAttribute)* S* GT;

emptyMacro:
    LT macroId (S+ macroAttribute)* S* SLASH GT;

macroId:
    CHARS COLON (CHARS | DOT)+;

macroAttribute:
    CHARS S? EQ S? QUOTE QUOTED? QUOTE;
...
```

3.3.2 Překladač

> *„Překladač (též kompilátor, anglicky compiler z to compile – sestavit, zpracovat) je všeobecně stroj, respektive program, provádějící překlad z nějakého vstupního*

jazyka do jazyka výstupního. Nejčastěji bývá tento pojem používán v programování, kde označuje nástroje překládající algoritmus zapsaný ve vyšším programovacím jazyce do jazyka strojového, nebo spíše do strojového kódu.“[4]

Překladač bývá obvykle rozdělen na několik částí. Tzv. front-end je závislý na zdrojovém jazyce a provádí překlad vstupního dokumentu ze zdrojového jazyka do mezikódu. Back-end je závislý na cílovém jazyce a provádí překlad z mezikódu do cílového jazyka. Mezikód tvoří rozhraní mezi front-endem a back-endem a je možné na něm například provádět optimalizace. Díky tomuto rozhraní je také možné použít jeden front-end s několika různými back-endy, nebo naopak (viz obrázek 6)



Obrázek 6: Schéma funkce překladače

Při použití nástroje ANTLR ke generování překladače, můžeme za front-end považovat vstupní gramatiku, protože z této gramatiky budou automaticky vygenerovány lexikální a syntaktický analyzátor. Back-end je realizován kusy kódu jazyka Java vloženými na patřičná místa této gramatiky. To znamená, že generovaný syntaktický analyzátor potom zároveň vykonává i funkci překladače. Alternativně je možné back-end realizovat formou AST analyzátoru (viz obrázek kapitola 2.2). Abstraktní syntaktický strom, který je výstupem syntaktického analyzátoru, potom plní funkci mezikódu (rozhraní) a díky tomu je možné skutečně oddělit front-end a back-end překladače.

Front-end překladače šablonovacího systému je tedy v našem případě tvořen dvěma gramatikami, které jsou popsány v předchozí kapitole. Mezikód je reprezentován následujícím způsobem. Pomocí akcí vložených na vhodná místa syntaktické gramatiky, jsou mapována jednotlivá pravidla na objekty, reprezentující jazykové konstrukty jazyka (objekty implementující rozhraní *Chunk*). Toto je názorně vidět na následující ukázce:

```
macroEnd returns [Chunk value]:
    t=LT SLASH macroId S* GT { $value = new MacroEndChunk($macroId.name, $t.line); };
```

Šablona je tedy po skončení syntaktické analýzy reprezentována jako sekvence takovýchto objektů. Jak se s těmito objekty naloží už je záležitostí back-endu. Rozhraní *Chunk* by například mohlo předepisovat metodu *compile*, která by vracela řetězec reprezentující daný objekt v cílovém jazyce. Back-end by potom jen postupně zavolal tuto metodu na všech objektech. Konkrétní implementace back-endu není předmětem této práce a skutečný back-end překladače společnosti oXy Online sice používá rozhraní s názvem *Chunk*, ale toto rozhraní nedefinuje žádnou metodu *compile* a back-end pracuje s těmito objekty jiným způsobem.

4 Platforma NetBeans

Jak již bylo řečeno v úvodu, platforma NetBeans je modulární systém, který je vhodným základem pro vývoj rozsáhlých desktopových aplikací. Při vývoji aplikací se vývojářům často stává, že se nemohou soustředit na vývoj samotné aplikace, ale věnují hodně úsilí nezbytné práci na infrastruktuře aplikace. Části aplikací však bývají často velice podobné – životní cyklus desktopových aplikací, systém správy oken apod. Proto vznikly tzv. aplikační rámce (frameworks), které obstarají tuto běžnou funkcionalitu a vývojář se potom může soustředit na vlastní podstatu problému. NetBeans Platform je aplikačním rámcem typu *Rich Client Platform*[10].

Modularita je velice důležitou vlastností této platformy. Modulární systémy umožňují vyvíjet aplikace pomocí modulů. Moduly jsou kompozitní části, ze kterých se staví modulární aplikace, a které mohou být vyvíjeny a používány nezávisle a vždy zapouzdřují určitou ucelenou funkcionalitu, kterou zpřístupňují okolnímu světu definovaným rozhraním. Moduly ale také mohou využívat služeb, které nabízejí jiné moduly a vznikají tak závislosti mezi nimi. Architekturu aplikace je potom možné vidět jako graf závislostí mezi moduly. Modulární architektura s sebou nese výhody jako znovupoužitelnost modulů, pružnější návrh aplikace, snadná rozšiřitelnost a škálovatelnost. Jak se aplikace vyvíjí a roste, modularita zajistí, že aplikace bude stále dobře uspořádaná a koherentní, a nebude zároveň s aplikací růst i entropie.

4.1 NetBeans runtime container

Nutným a postačujícím základem každé aplikace postavené na platformě NetBeans je r*untime container*, což je běhové prostředí poskytující základní funkce a služby. Skládá se z 5 modulů, jak je vidět na obrázku 7. Toto běhové prostředí je základem modulového systému NetBeans.

Obrázek 7: NetBeans runtime container[6]

10 Platforma k vytváření aplikací typu Rich Client Aplication, což jsou aplikace, jejichž podstatná část běží na uživatelově lokálním systému. Víceméně se jedná o jiné označení desktopových aplikací.[5]

Modul *Bootstrap* je zodpovědný za nahrání aplikace, definuje co je to modul, a umožňuje, aby moduly byly nalezeny a zacházelo se s nimi jako s aplikací. Modul *Startup* má na starosti start aplikace – tzn. nemusíme psát metodu *main*, protože tato metoda je obsažena v modulu *Startup*. Ostatní moduly jsou popsány dále.

4.1.1 Module system API

Tento modul poskytuje abstraktní nadtřídu všech NetBeans modulů – tzn. specifikuje základní vlastnosti, které jsou společné pro všechny moduly – a také se stará o správu závislostí mezi moduly. V prostředí platformy NetBeans jsou moduly speciálním typem JAR archivů, které se ukládají s příponou NBM a kromě běžného obsahu JAR archivů navíc obsahují konfigurační soubor *manifest.mf* ve speciálním tvaru. V tomto souboru se vyskytují specifické klíče, které obsahují doplňující informace pro *runtime container,* a podle kterých *runtime container* pozná, že se jedná o NetBeans modul. Důležité vlastnosti NetBeans modulů jsou rozebrány v následujících podkapitolách.

4.1.1.1 Veřejná API modulů

U NetBeans modulů je možné definovat veřejná API modulů (pomocí kterých mohou ostatní moduly využívat jejich služeb) tak, že se explicitně vyjmenuje seznam balíků, které budou viditelné z ostatních modulů. Obsah balíků, které nejsou vyjmenované, zůstane ostatním modulům skryt. Přestože při vývoji můžeme věnovat značné úsilí správnému návrhu API, časem se mohou požadavky na modul změnit a může být potřeba toto rozhraní upravit. Proto se u NetBeans modulů API označuje verzí. Existují následující 3 typy verzí:

- **Mejor Release Version** – tento typ se používá k označování veřejných API. Jestliže se výraznějším způsobem změní veřejné API, mělo by se změnit také číslo verze tohoto typu.
- **Specification Version** – tento typ se používá při menších změnách veřejných API. Při těchto změnách je zachována zpětná kompatibilita a pouze je například přidána nová metoda apod.
- **Implementation Version** – tento typ se používá pro označení implementačních (interních) změn v modulu, přičemž veřejné rozhraní zůstává netknuté. Pro ostatní moduly, které využívají služeb tohoto modulu, není toto číslo důležité.

Může také nastat případ, kdy vyvíjený modul ještě není příliš stabilní a je možné nebo pravděpodobné, že API tohoto modulu se bude ještě měnit. Kolegové, kteří pracují na stejném projektu, už ale potřebují s vaším modulem pracovat. V takovém případě může autor modulu uvést seznam *přátelských modulů*, kterým bude veřejné API k dispozici. Moduly, které na tomto seznamu nejsou, nemají k API přístup – tzn přestože modul definuje seznam veřejných balíků, můžou mít k těmto balíkům přístup jen některé vybrané moduly.

4.1.1.2 Závislosti mezi moduly

U modulárních systémů je důležitým aspektem správa závislostí mezi moduly. V prostředí platformy NetBeans je spolupráce mezi moduly realizována tak, že jeden modul explicitně definuje své veřejné API, a druhý modul explicitně definuje závislost na prvním modulu. Potom druhý modul může využívat služeb, které poskytuje první modul pomocí svého veřejného API. Závislost je potřeba

definovat nejen na konkrétním modulu, ale také přímo na konkrétní verzi modulu (z důvodů, které byly popsány v předchozí kapitole). Typicky se definuje závislost na specifikační verzi modulu.

Ve výjimečných případech je možné přistupovat také k veřejným třídám a rozhraním balíků modulů, které nejsou v modulu definované jako veřejné API, nebo nejsou pro váš modul viditelné, protože váš modul není uveden na seznamu *přátelských modulů*. V tomto případě se jedná o tzv. implementační závislost – tj. závislost na implementační verzi modulu.

Je důležité dbát na to, aby mezi moduly nevznikaly cyklické závislosti. Pokud by například modul A byl závislý na modulu B, a zároveň modul B opět na modulu A, potom by nastal problém při pokusu inicializovat jeden z těchto modulů. Modul A by při své inicializaci požadoval, aby byl nejprve inicializován modul B, a modul B by zase požadoval, aby byl nejprve inicializován modul A. Pokud se potýkáme s tímto problémem, je to nejspíše způsobeno špatným návrhem aplikace a řešení nabízí zavedení třetího modulu C, který by obsahoval společnou funkcionalitu, a moduly A i B by potom byly závislé na tomto modulu C.

Dále je potřeba si uvědomit, že závislosti mezi moduly nejsou tranzitivní – každá závislost mezi moduly musí být uvedena explicitně. Pokud je tedy modul A závislý na modulu B a modul B závislý na modulu C, modul A nemá přístup k veřejnému rozhraní modulu C, protože mezi A a C není explicitně definována závislost.

Runtime container prozkoumává moduly, ze kterých se aplikace skládá, a uvádí v platnost závislosti mezi moduly za běhu. Nejprve vyřeší závislosti mezi moduly a pokud se toto podaří, pak vytvoří aplikaci.

4.1.1.3 Izolovanost modulů

Moduly jsou od sebe navzájem izolované. Každý modul totiž používá svůj vlastní *classloader*[11]. Proto pokud se ve svém modulu snažíme použít třídu z jiného modulu, a pokud není mezi těmito moduly explicitně definována závislost, aplikace vyhodí výjimku *ClassNotFoundException*. Díky tomuto faktu se nám nemůže stát, že bychom náhodou použili špatnou třídu ze špatného modulu. Moduly jsou zapouzdřené a máme přístup jen ke třídám ve veřejných balících modulů, ke kterým explicitně definujeme závislost.

Další výhodou této vlastnosti je, že aplikace může klidně používat dvě různé verze stejné knihovny, za předpokladu, že jsou tyto knihovny každá ve svém zvláštním modulu.

4.1.1.4 Životní cyklus modulů

Jak již bylo řečeno dříve, moduly můžeme považovat za samostatné aplikace. Proto každý modul má svůj životní cyklus. Jednotlivé fáze životního cyklu reprezentují důležité události, které během života modulu mohou nastat. Přístup k těmto událostem nám poskytuje právě *Module system API*. Následuje stručný popis možných událostí:

- **Validace** – jedná se o první fázi životního cyklu. Probíhá zde například ověření závislostí.
- I**nstalace / Odinstalace** – událost nastávající při první (od)instalaci modulu do (z) aplikace.

11 *Classloader* je součást virtuálního stroje Java, která má na starosti vyhledávání a nahrávání souborů tříd v reálném čase. Ve skutečnosti nemá každý modul svůj vlastní *classloader*, ale je vytvářena hierarchie *clasloaderů* odpovídající závislostem mezi moduly.

- **Aktivace / Deaktivace** – událost nastávající při použití modulu (inicializace).
- **Update** – událost nastávající pokud je při instalaci modulu do aplikace zjištěna novější verze než při předchozí instalaci.

Ve většině případů není nutné do životního cyklu modulů nijak zasahovat.

4.1.1.5 Typy modulů podle inicializace

Pokud máme rozsáhlou aplikaci skládající se ze stovek modulů a při startu aplikace se musí všechny moduly inicializovat, může start takové aplikace trvat poměrně dlouhou dobu. Je proto dobré na to při vývoji modulu pamatovat a snažit se, aby modul při své inicializaci prováděl jen nezbytné minimum operací a zabíral tak minimální množství inicializačního času. Nejrychleji se samozřejmě inicializuje ten modul, který se neinicializuje vůbec, proto tvůrci platformy implementovali také mechanismus líné inicializace – tzn. moduly se nemusí inicializovat při startu aplikace, ale až v okamžiku, kdy jsou skutečně potřeba. Tyto moduly jsou při startu aplikace pouze registrovány do systému deklarativním přístupem. Platforma NetBeans proto nabízí tyto tři typy modulů:

- **Regular modules** – jsou inicializovány při startu aplikace. Je proto velice důležité aby tyto moduly zabíraly co možná nejmenší inicializační čas, protože celkový čas startu aplikace nemůže být nikdy menší než součet inicializačních časů všech *regular* modulů, které aplikace obsahuje.
- **Autoload modules** – jsou moduly, které jsou inicializovány líně – tzn. až za běhu aplikace v okamžiku, kdy jsou poprvé skutečně potřeba. Tento typ se většinou používá u modulů, které obsahují knihovny (není potřeba, aby byla nahrána knihovna dříve než se použije a zbytečně tak zabírala startovní čas a paměť).
- **Eager Modules** – modul tohoto typu je inicializovány teprve v okamžiku, kdy už jsou inicializovány a k dispozici všechny moduly na kterých je závislý. Kdy bude tento modul inicializován tedy není pod přímou kontrolou uživatele.

4.1.2 File system API

Modul *File system API* poskytuje virtuální souborový systém a API pro práci s tímto systémem. Souborový systém obecně je místo, kde můžeme ukládat, organizovat a získávat složky, soubory a data. Aplikace při instalaci ukládají své soubory na určitá místa souborového systému, který jim poskytuje operační systém. Udržet si přehled v instalovaných aplikacích je potom jednoduché právě díky hierarchické struktuře, kterou souborový systém poskytuje. Jak již bylo několikrát řečeno, moduly si můžeme představit jako samostatné aplikace. Potom stejně jako při instalaci skutečné aplikace, také při instalaci modulu do modulárního systému se vytvářejí na určitých místech souborového systému soubory a složky. V tomto případě se však jedná právě o virtuální souborový systém poskytovaný modulem *File systém API*.

Moduly provádějí instalace do souborového systému jen pokud to skutečně potřebují. Například pokud chceme přidat novou položku do menu aplikace, potom náš modul musí přidat soubor do složky *„Menu“* v souborovém systému. Tuto složku vytváří a spravuje modul, který má na starosti konstrukci a správu aplikačních menu. Ten jednoduše vezme všechny soubory, které se vyskytují v této složce a vytvoří z každého z nich položku v menu aplikace. Souborový systém tedy umožňuje modulům jistou formu komunikace.

Moduly mohou pracovat se souborovým systémem také deklarativním způsobem – pomocí speciálního konfiguračního XML souboru *layer.xml* někdy také vývojáři nazývaného *layer(vrstva)* nebo *sandwich(sendvič)*. Tento soubor někdy modul nemusí mít vůbec (pokud nepotřebuje pracovat se systémem souborů), ale nikdy nesmí mít víc než jeden tento soubor. Jedná se o druhý nejdůležitější konfigurační soubor modulu (po *manifest.mf*) a cesta k tomuto souboru bývá deklarována v souboru manifestu. Struktura souboru je velice jednoduchá a je dána pomocí DTD (http://www.netbeans.org/dtds/filesystem-1_1.dtd).

Jak už název konfiguračního souboru napovídá, jedná se o vrstvu souborového systému. Každý modul si může vytvořit svou vlastní vrstvu souborového systému a registrovat si zde své objekty, soubory, konfigurační data, nebo co je potřeba. Při startu aplikace založené na platformě NetBeans *runtime container* prozkoumává všechny moduly, ze kterých se aplikace skládá. Při tomto průzkumu najde všechny *vrstvy* souborových systémů všech modulů a sloučí je do jednoho globálního souborového systému a ten je potom přístupný aplikaci. Proto si může například každý modul definovat vlastní položky menu a modul, který je zodpovědný za vytváření aplikačního menu, je potom pohodlně všechny najde ve složce *„Menu“* globálního souborového systému.

Runtime container navíc také pracuje s adresářem na disku (ve skutečném souborovém systému operačního systému), ve kterém perzistentně ukládá nastavení aplikace. Pokud tedy modul nastavuje pomocí své *vrstvy* nějaké implicitní hodnoty a uživatel je při práci s aplikací změní, *runtime container* se postará o to, aby tyto hodnoty byly uloženy, a při dalším startu aplikace potom opět použity namísto hodnot implicitních.

Souborový systém bývá většinou moduly využíván k registraci prvků (nebo celkově vzato škálování) uživatelského rozhraní, ke konfiguraci implicitních hodnot různých nastavení nebo k registraci různých externích zdrojů jako jsou obrázky, soubory nápovědy apod. Mechanismus souborového systému je také možné využít k registraci rozšiřujících bodů modulu – poskytovatelů služeb (podporovaných od JDK6[12]). Je také možné dynamicky za běhu aplikace nahrávat různé vrstvy souborového systému například podle toho, který uživatel s aplikací zrovna pracuje. Navíc souborový systém umožňuje registraci Java objektů, neboli mapování souborů na Java objekty – v souborovém systému je tedy uložen soubor s názvem ve tvaru *nazev-meho-baliku.MojeTrida.instance*. *Runtime container* potom ve chvíli, kdy je potřeba daný objekt, automaticky vytvoří instanci třídy dané souborem.

4.1.2.1 Objekt FileObject

Pomocí *File system API* můžeme přistupovat k souborům jednotným způsobem. Každý soubor je reprezentován jako instance třídy *FileObject* bez ohledu na to, jestli se jedná o skutečný soubor nebo složku ve skutečném souborovém systému operačního systému, nebo o soubor ve virtuálním souborovém systému NetBeans, nebo o soubor uložený například v naší vlastní implementaci souborového systému. Všechny souborové systémy přístupné platformě NetBeans jsou registrovány v registru souborových systémů reprezentovaném objektem *Repository*. *FileObject* tvoří položky souborových systémů, a tak je možné tyto objekty z tohoto registru získávat. *FileObject* je zapouzdřením třídy *java.io.File* a je navržen tak, aby vývojáři neměli potřebu pracovat s touto klasickou třídou. Následuje stručný popis hlavních rozdílů mezi objekty typu *File* a *FileObject*:

12 Poskytovatele služeb (service providers) je možno registrovat v souboru *layer.xml* alternativně ke klasické registraci v adresáři *META-INF/services*, která je používána v JDK.[7]

- objekty typu *FileObject* nereprezentují něco co neexistuje, na rozdíl od *File*, kde toto bylo možné voláním konstruktoru s neexistující cestou k souboru.
- objekty typu *FileObject* získáváme z souborového systému a ne voláním konstruktoru jako u objektů typu *File*.
- na objektech typu *FileObject* můžeme hlídat změny (včetně změn na nižších hierarchických úrovních složky).
- objekty typu *FileObject* nemusí nutně reprezentovat jen skutečné soubory na disku.
- objekty typu *FileObject* mohou mít atributy (dvojice klíč–hodnota), které jsou asociovány se soubory.
- oddělovačem cest je u objektů *FileObject* vždy znak „/“ (ne *File.separator* jako u *File*).

Objekty typu *FileObject* nabízejí jednotný přístup ke všem typům souborů také proto, že vidí soubory jen jako proud bytů. Pokud chceme se soubory pracovat na vyšší úrovni, slouží k tomu *Datasystems API* poskytující objekt *DataObject*, který zapouzdřuje *FileObject* a *Nodes API*, které poskytuje objekty typu *Node* sloužící k reprezentaci objektů *DataObject* na prezentační vrstvě. Tuto hierarchii je možné vidět na obrázku 8.

Obrázek 8: Hierarchie tříd pro reprezentaci souborů v NetBeans API[8]

4.1.3 Utilities API

Modul *Utilities API*, jak už jeho název napovídá, nabízí různé užitečné pomůcky, mezi než patří:

- **Lookup** – je podrobně popsán v následující podkapitole.
- **NbBundle** – poskytuje vylepšenou podporu pro lokalizaci a nahrazuje klasický *ResourceBundle*.
- **Task a RequestProcessor** – nabízí způsob jak vytvářet a spravovat vlákna a asynchronní výpočty.
- **HelpCtx** – slouží k vytváření nápovědy k jednotlivým funkcím aplikace.
- **Utilities** – poskytuje různé blíže nezařazené užitečné statické metody.
- **Enumerations** – zavádí vylepšenou podpora výčtových typů. Umožňuje vytvářet, modifikovat a slučovat výčtové typy, filtrovat jejich obsah a pracuje s nimi líným způsobem.

Tyto pomůcky samozřejmě slouží k různým účelům, a proto při vývoji určitých modulů některé z nich vůbec nebudou potřeba. Bez pomůcky *Lookup* se však jen těžko obejdeme, protože ta je jedním ze základních pilířů platformy.

4.1.3.1 Lookup

Modulární aplikace se zpravidla skládají z několika nezávislých modulů. Tyto moduly jsou nezávislé v tom smyslu, že jsou použitelné jeden bez druhého, bývají vyvíjeny odděleně a podobně. Navzájem spolu tyto moduly ale potřebují komunikovat a spolupracovat – některé moduly poskytují určité služby pomocí svého veřejného rozhraní a jiné moduly těchto služeb následně využívají. Při stavbě aplikace z modulů tak mezi moduly vytváříme závislosti neboli vazby. Dobrou vlastností těchto vazeb je jejich volnost. Pojem volnost vazeb může v různých prostředích nabývat různých významů. V prostředí platformy NetBeans nám postačí představit si volnou vazbu tak, že modul využívající určitou službu (zákazník) deklaruje závislost pouze na službě (SPI[13]) a ne na konkrétním poskytovateli služby (implementaci tohoto rozhraní). Je potom tedy možné snadno nahrazovat nebo přidávat nové poskytovatele služeb, a tak měnit funkcionalitu aplikace bez nutnosti zasahovat do modulů, ze kterých se už aplikace skládá.

Obrázek 9: Různé implementace (poskytovatelé) služby (rozhraní)

Modulární systém pak stojí před problémem jak zařídit, aby mohly různé komponenty registrovat do systému služby, které poskytují, a jak potom mohou jiné komponenty tyto dostupné služby vyhledat a získat k nim přístup. V prostředí platformy NetBeans tento problém řeší právě pomůcka *Lookup*.

Lookup je obecný registr, umožňující „zákazníkům“ najít a získat poskytovatele dané služby, kteří jsou zavedeni v tomto registru. Registraci poskytovatele služeb je možné provést dvěma způsoby:

- stejně jako v JDK, vytvořením souboru se jménem shodným s plným kvalifikovaným názvem rozhraní služby ve složce *META-INF/services* modulu. Tento soubor se skládá z řádků obsahujících plné kvalifikované jméno třídy implementující toto rozhraní (třída poskytující danou službu).

13 SPI je zkratkou anglického výrazu *Service Provider Interface* – tedy rozhraní služby, které poskytovatelé implementují

- vytvořením instance poskytovatele služby jako souboru *.instance* ve složce *Services* v systémovém souborovém systému.

Na první pohled tedy *Lookup* zastává podobnou funkci jako *ServiceLoader* JDK6 a navíc nabízí ještě druhý způsob registrace poskytovatelů služeb. Toto druhé řešení je však výhodnější. Instanci můžeme totiž vytvořit několika způsoby – voláním bezparametrického konstruktoru, parametrického konstruktoru, nebo pomocí statické tovární metody. Navíc jako atributy souboru instance můžeme poskytnout doplňující informace jako je seznam rozhraní a nadtříd, které daná instance implementuje a rozšiřuje. Díky tomu potom *Lookup* nemusí při prohledávání registrovaných poskytovatelů služeb vytvářet instanci tohoto objektu, ale může ověřovat jestli se jedná o požadovaného poskytovatele pomocí těchto doplňujících informací. Instance je potom vytvořena líně, až když je skutečně potřeba, a tím se šetří čas a paměť.

Další výhodou *Lookup*u je, že navíc je možné poskytovatele služeb také ze systému odebírat a tím například nahrazovat poskytovatele které registrovaly jiné moduly vlastními implementacemi. Je také možné udávat řazení poskytovatelů služeb.

Lookup tak jak byl popisován doteď je tzv. *Default Lookup* neboli *implicitní globální Lookup* a působí na úrovni aplikace. Registrovaní poskytovatelé služeb jsou přístupní libovolné části aplikace. Často se jedná o objekty podle návrhového vzoru *singleton* u kterých stačí jediná instance pro celou aplikaci jako například *ErrorManager*, který slouží k logování chyb a výjimek.

Některé objekty ale ještě navíc nabízejí vlastní *lokální Lookup*. Ten potom umožňuje získávat instance tříd, které skutečně vykonávají určitou práci pro objekt poskytující tento *Lookup*. Typický příklad použití *lokálního Lookupu* je uveden v následujícím odstavci.

Užitečnou funkcí je také možnost sledování *Lookup*u pomocí *listenerů* – tímto způsobem funguje například ukládání souboru v Netbeans IDE. Modul spravující menu a tlačítkovou lištu hlídá, jestli je v lokálním *Lookup*u souboru (objekt *DataObject*), který je zrovna otevřený v editoru, k dispozici instance implementace rozhraní *SaveCookie*[14]. Pokud není, tlačítko pro uložení je zakázané. Když se ale obsah souboru v editoru změní, modul editoru zaregistruje implementaci rozhraní *SaveCookie* do *Lookup*u příslušného *DataObject*u. Modul spravující menu a tlačítkovou lištu to zaznamená a povolí tlačítko pro uložení. Při stisku tlačítka je vykonána akce, kterou služba *SaveCookie* poskytuje. Tlačítko tedy neví, jak se soubor uloží, ví jen, že je možné ho uložit pomocí služby *SaveCookie*, kterou poskytuje daný datový objekt, a to při svém stisku také provede.

4.2 Další užitečné moduly platformy NetBeans

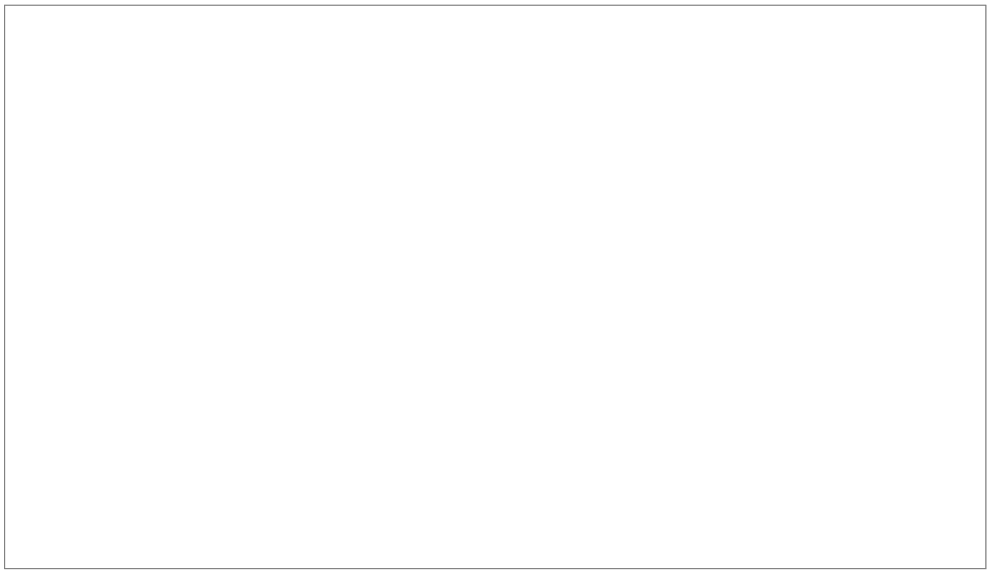
Jak bylo vysvětleno v minulé kapitole, *runtime container* je základem každé aplikace založené na platformě NetBeans a poskytuje aplikaci modulární systém, prostředky jak s tímto systémem zacházet a pár dalších užitečných pomůcek. Neposkytuje však aplikaci žádné grafické uživatelské rozhraní nebo jiné podobné užitečné nástroje, které často bývají součástí aplikací. To ale také není potřeba – tato další funkcionalita je implementována a zapouzdřena v dalších modulech, které platforma NetBeans nabízí. A díky faktu, že je naše aplikace modulární, můžeme v ní tyto již hotové moduly s výhodou použít a ušetřit si tak spoustu práce. Následující seznam vyjmenovává některé moduly, které bývají součástí většiny aplikací založených na platformě NetBeans:

14 Objekty *Cookie* jsou návrhovým vzorem platformy NetBeans a reprezentují určitou schopnost nějakého objektu. Tedy například *SaveCookie* znamená schopnost *DataObject*u uložit se a jedná se o rozhraní služby. Konkrétní implementace říkající jak se má soubor uložit je získána pomocí *Lookupu*.

- **Datasystems API** – umožňuje reprezentovat soubor jako datový objekt, jak již bylo zmíněno v kapitole 4.1.2.1.
- **Nodes API** – poskytuje mechanismy pro reprezentaci datových objektů na prezentační vrstvě.
- **Window System API** – poskytuje základní GUI aplikace, bohatě škálovatelný dokovací systém a metody jak spravovat okna a pracovat s nimi. Základními komponentami jsou objekty *TopComponent*, což jsou rozšíření třídy *javax.swing.JComponent*, která jsou spravována dokovacím systémem.
- **Actions API** – uživatelé mohou využívat služeb modulů pomocí volání akcí přes menu, nástrojové lišty nebo klávesové zkratky. Akce implementují rozhraní *javax.swing.Action* a navíc je možné, aby akce byly k dispozici jen pro konkrétní objekty, aby se změnila implementace akce podle toho která komponenta má fokus apod.
- **UI Utilities API** – poskytuje užitečné pomůcky týkající se různých komponent uživatelského rozhraní.

4.3 Platformy založené na platformě NetBeans

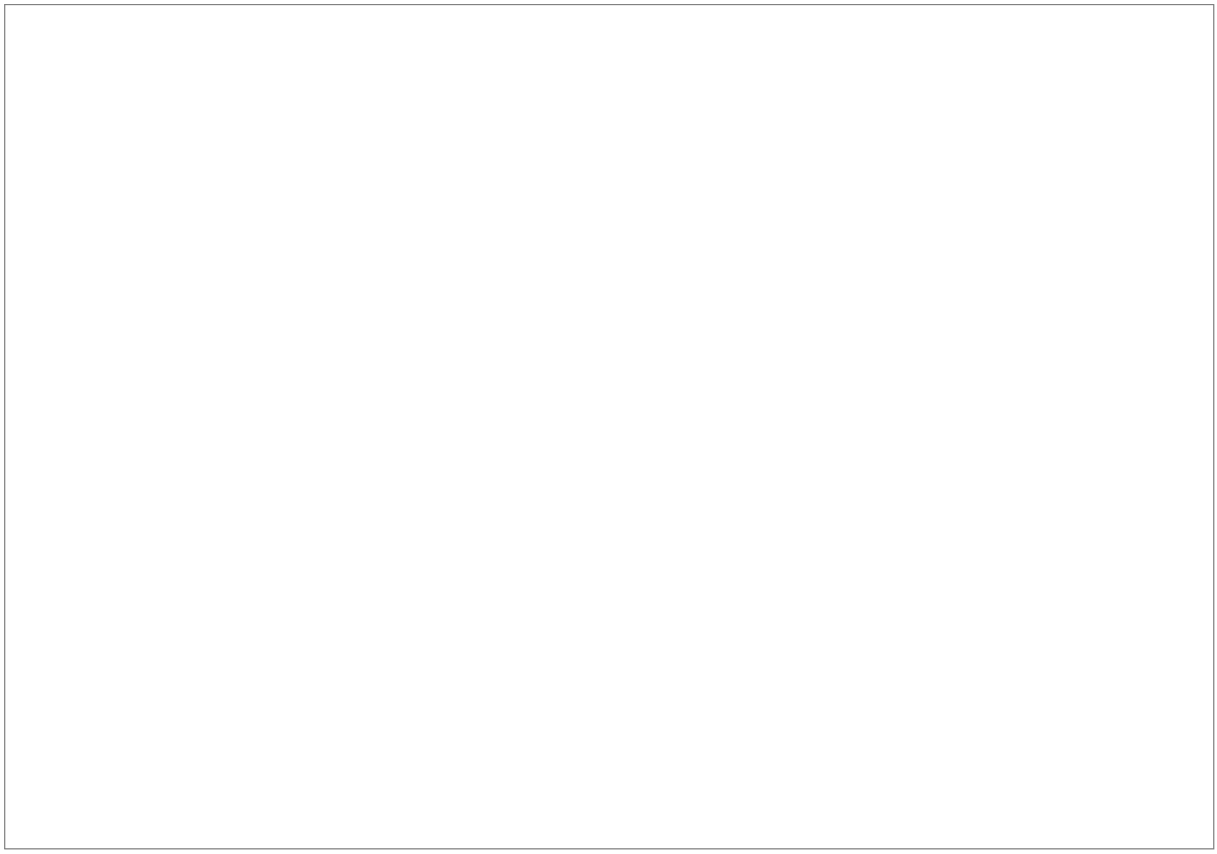
Jak již bylo řečeno v úvodu, platforma NetBeans je vyvíjena souběžně s integrovaným vývojovým prostředím NetBeans IDE, které je na této platformě založeno. Samotné NetBeans IDE však vlastně také tvoří platformu, protože díky modulárnímu systému můžeme použít platformu NetBeans a moduly NetBeans IDE jako základ a pouze přidat své vlastní moduly a vytvořit tak svou aplikaci, respektive svou platformu. Takovéto řetězení platforem skutečně má svůj význam, protože je potřeba si uvědomit, že závislosti mezi moduly je možné vytvářet pouze mezi platformou a pluginem do této platformy, nebo jen v rámci jednoho pluginu, jak ukazuje obrázek 10, kde je problémová vazba zvýrazněna silněji. Pojem *plugin* se v prostředí NetBeans objevil teprve nedávno a vyjadřuje skupinu modulů, které spolu sémanticky souvisí. Jedná se vlastně o vyšší úroveň abstrakce než je modul.



Obrázek 10: Závislosti mezi moduly pluginů a platformy

5 Implementace vývojového prostředí

Úvodem je potřeba říct, jaké funkce bude vytvořené vývojové prostředí podporovat. Zřejmě nejdůležitější částí vývojového prostředí je textový editor, proto začněme s popisem funkcí textového editoru. Nebudeme zmiňovat funkce běžných textových editorů, jako je psaní a upravování textu, práce se schránkou, vyhledávání a podobně, ale zaměříme se jen na funkce, které mají navíc editory ve vývojových prostředích.



Obrázek 11: Vývojového prostředí usnadňující vývoj šablon

Patrně nejdůležitější funkcí u textového editoru vývojového prostředí je barevné zvýrazňování syntaxe zdrojového kódu dokumentu. I kdyby toto byla jediná funkce, kterou by editor podporoval, přesto by tím byl programátorům výrazně zvýšen komfort práce. Další příjemnou funkcí, kterou od editoru očekáváme je zvýrazňování syntaktických chyb. Většinou je taky dobré, aby zvýrazněna chyba byla vybavena popiskem s informací proč byla zvýrazněna. Uživatel tak má možnost zjistit nejen kde se chyba nachází, ale také o jaký druh chyby se jedná a má tak dostatek informací k tomu, aby mohl chybu odstranit. Kromě chyb se ve zdrojovém kódu mohou vyskytovat také jiná problémová místa, která nejsou přímo chybami, ale za určitých okolností mohou k chybám vézt. Takováto místa je možné označit varováním. Užitečnou funkcí editoru je také automatické formátování kódu, které pomáhá programátorům udržovat kód v čitelnější formě. Konkrétně se jedná o automatické odsazování nových

řádků na správnou pozici podle kontextu, a o možnost přeformátovat část dokumentu (nebo celý dokument) podle předepsaných konvencí. Další pomůckou kterou textový editor nabízí je tzv. *code folding* neboli skládání kódu. Tato pomůcka zobrazuje skládací značky u určitých bloků kódu a pomocí těchto značek je možné daný blok sbalit a schovat. Programátor potom místo tohoto kódu vidí jen zkratku, která reprezentuje schovaný blok. Délka zdrojového kódu se tak opticky zkrátí, což může pomoct čitelnosti. V případě potřeby je samozřejmě možné zkratku opět rozbalit. Poslední z důležitých funkcí editoru je automatické doplňování kódu a kontextová nápověda.

Kromě funkcí, které nabízí samotný textový editor, ale vývojové prostředí nabízí ještě další užitečné pomůcky, jako je například navigátor. Navigátor je zobrazen v okně odděleném od editoru a jak už jeho název napovídá, nabízí zjednodušený pohled na dokument usnadňující navigaci v tomto dokumentu. Navigátor tedy zobrazuje hierarchický seznam důležitých bloků dokumentu a kliknutím na položku v tomto seznamu dojde k přesunutí kurzoru v editoru na patřičné místo dokumentu. Na druhou stranu také při přesunu kurzoru v editoru je vždy zvýrazněna odpovídající položka v navigátoru a programátor tak má přehled, v které části dokumentu se nachází, přestože v obrazovce editoru vidí jen krátkou část kódu.

5.1 Návrh pluginu

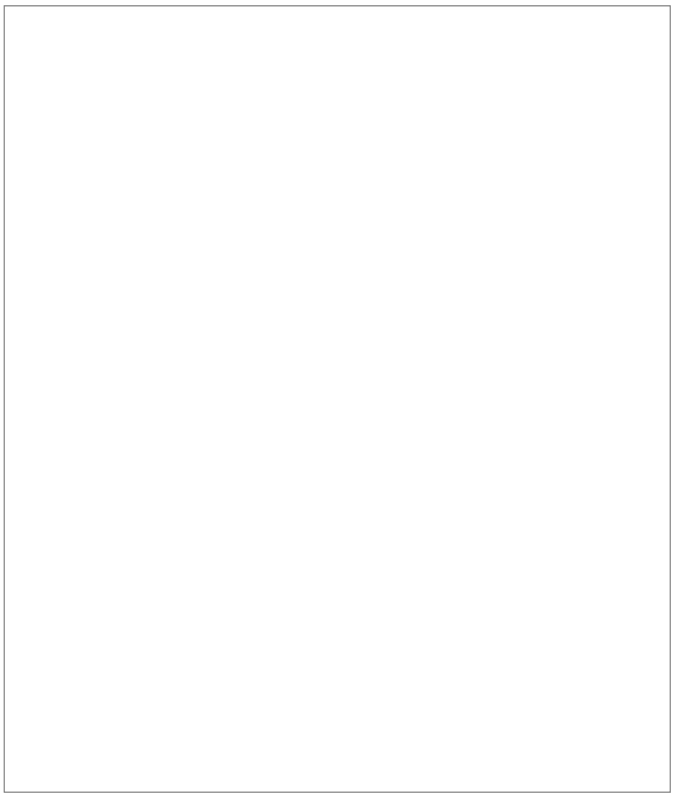
Aby bylo možné implementovat funkce jako je zvýrazňování syntaxe, skládání kódu nebo navigátor, je potřeba, aby tyto funkce věděly, které části kódu mají zabarvit, ke kterým dát skládací značku, a které abstrahovat do položky navigátoru. Je tedy zřejmé, že základem těchto funkcí je analýza dokumentu, kterou bude zřejmě provádět nějaký analyzátor.

Vývojové prostředí pro vývoj šablon webového šablonovacího systému společnosti oXy Online je implementováno jako plugin do NetBeans IDE. Platforma NetBeans IDE podporuje několik způsobů jak analyzovat strukturu dokumentu a vizualizovat výsledky analýzy:

- **Rozšířením tříd *Syntax* a *TokenContext*** – toto je nejstarší způsob, analyzátor a spoustu dalších tříd je potřeba implementovat ručně. Při prokládání jazyků je potřeba implementovat třídu *MultiSyntax*, analyzátor je velice složitý a dlouhý a tím pádem také náchylný na chyby.
- **Použitím modulu Lexer** – tento modul nabízí možnost definovat gramatiku podporovaného jazyka deklarativním způsobem, o analýzu podle této gramatiky se už postará modul. Jedná se však pouze o lexikální a syntaktickou analýzu, k sémantické analýze nenabízí modul žádné prostředky. Je možné také definovat elementární pravidla pro skládání kódu a navigátor. Situace kolem prokládání jazyků je zde vyřešena mnohem lépe – bloky vloženého jazyka jsou označeny jako zvláštní tokeny. Analýza těchto tokenů je potom delegována na analyzátor příslušného vloženého jazyka. Takto vzniká hierarchie jazyků.
- **Použitím modulu Generic Languages Framework** – modul je zatím ve stádiu vývoje, nicméně už je použitelný a je na jeho základě vytvořena podpora pro několik programovacích jazyků. Stejně jako u modulu lexer, také tento modul má za úkol zjednodušit vytváření podpory nového programovacího jazyka pro NetBeans IDE, ale v tomto případě jiným způsobem. Modul definuje rozhraní pro jednotlivé analyzátory a abstrahuje jednotlivé komponenty IDE, které pracují s výsledky těchto analyzátorů. Zjednodušeně řečeno vývojář se postará o analýzu jazyka a modul se postará o IDE. Vývojář tedy musí poskytnout jednotlivé analyzátory, což ovšem neznamená, že tyto

analyzátory skutečně musí implementovat. K některým jazykům už jsou totiž k dispozici analyzátory hotové, a stačí je proto pouze adaptovat na rozhraní, která modul poskytuje.

Protože v našem případě máme již hotové analyzátory k dispozici (viz kapitola 3.3.1), je zřejmé, že je použita třetí možnost – tedy modul *Generic Languages Framework*. Původním název modulu byl *Generic Scritping Framework*, odtud vznikla zkratka GSF. Pokud se tato zkratka objeví v dalším textu práce, vězte, že zastupuje právě tento modul. Protože modul je stále ve stádiu vývoje, bylo k dispozici jen velmi malé množství dokumentace, a to prakticky pouze v podobě JavaDocových komentářů. Často však i tento komentář chyběl a proto bylo nutné přistoupit až k prohlídce zdrojových kódů modulu. Dalším pramenem ze kterého se dalo čerpat byly zdrojové kódy několika modulů NetBeans IDE, které také využívají modul GSF k podpoře jiných jazyků. Na několik dotazů mi také ochotně emailem odpověděl sám autor modulu Tor Norbye, za což mu patří můj dík.



Obrázek 12: Architektura pluginu

Plugin se skládá ze 4 základních modulů a jeho architektura je vidět na obrázku 12. Následuje stručný popis těchto modulů:

- **AntlrWrapper** – jedná se o tzv. *wrapper* modul. Tento typ modulu se používá pro zapouzdření již existujících knihoven do NetBeans modulu. Tyto knihovny je samozřejmě možné přidat také přímo do modulu, který je bude využívat. V našem případě ale tyto knihovny využívají 2 různé moduly, je proto vhodnější knihovny zapouzdřit do wrapper modulu a potom jen deklarovat závislosti na tento modul. Vytvoření wrapper modulu je díky průvodci v NetBeans IDE záležitostí několika kliknutí.

- **AntlrGeneratedParser** – tento modul obsahuje analyzátory, které jsme vygenerovali pomocí ANTLR. Tyto analyzátory jsou zároveň zaregistrovány jako poskytovatelé služeb (viz kapitola 4.1.3.1).
- **AntlrGsfBridge** – modul obstarávající adaptování analyzátorů, které generuje ANTLR, do formy, kterou akceptuje GSF. Tento modul bude podrobněji rozebrán v kapitole 5.2.
- **JstEditor** – modul realizující základ vývojového prostředí. K tomuto účelu využívá zejména API, které poskytuje GSF, a analyzátory generované nástrojem ANTLR adaptované pomocí modulu *AntlrGsfBridge*. Modul bude podrobněji popsán v kapitole 5.3.

Jak je možné vidět na obrázku 12, moduly mezi sebou deklarují závislosti. Pojďme si ve stručnosti vysvětlit, co tyto závislosti mezi moduly znamenají, a jak probíhá vzájemná spolupráce modulů. Čísla závislostí v textu korespondují s čísly v obrázku 12.

Moduly *JstEditor* a *AntlrGsfBridge* deklarují závislosti (1 a 2) na modulech platformy NetBeans IDE. Protože je ve skutečnosti větší množství modulů platformy, na kterých je zejména modul *JstEditor* závislý, jsou pro schématičnost obrázku tyto závislosti zobrazeny jedinou šipkou a podrobně jsou popsány v příslušných kapitolách. Modul *AntlrWrapper* obsahuje knihovny ANTLR a mimo jiné také rozhraní analyzátorů – třídy *Lexer* a *Parser*. Tato rozhraní tvoří SPI modulu *AntlrGsfBridge*. Modul *AntlrGeneratedParser* obsahuje implementace analyzátorů a ty jsou zaregistrovány jako poskytovatelé služeb. Modul *AntlrGsfBridge* je uživatelem těchto služeb. Vazby číslo 3 a 4 tedy reprezentují volnou vazbu mezi modulem *AntlrGsfBridge* a dodanými implementacemi analyzátorů. Vazba číslo 5 bude vysvětlena v kapitole 5.2.1. Modul *AntlrGsfBridge* adaptuje analyzátory tak, aby vyhovovaly rozhraní, které poskytuje GSF. Takto adaptované analyzátory potom využívá modul *JstEditor* pro podporu svých funkcí – vazba číslo 6.

V následujících dvou kapitolách budou podrobně rozebrány moduly *AntlrGsfBridge* a *JstEditor*. Při návrhu a vývoji modulů bylo využíváno několika návrhových vzorů, jak bude vždy na příslušném místě uvedeno v textu. Principy použitých návrhových vzorů je možné nalézt v některém z těchto zdrojů [9], [10].

5.2 Modul AntlrGsfBridge

Tento modul provádí adaptaci analyzátorů vytvořených pomocí ANTLR do formy, kterou akceptuje GSF. Díky tomuto modulu je možné používat gramatiku pro nástroj ANTLR k podpoře funkcí editoru v NetBeans IDE. Díky tomu, že adaptování analyzátorů je abstrahováno a zapouzdřeno v tomto modulu, zatímco skutečné implementace analyzátorů jsou v jiném modulu, je možné jednoduše vyměnit implementace analyzátorů za jiné, aniž by bylo nutné sáhnout do zdrojového kódu libovolného modulu (samozřejmě za předpokladu, že výstupy analyzátorů zůstanou stejné).

Tento modul umí adaptovat dva druhy analyzátorů – lexikální (lexer) a syntaktický (parser). Způsob, jakým se adaptují tyto analyzátory, bude dále podrobně popsán ve zvláštní kapitole pro každý analyzátor. Zde jen v kostce shrňme princip.

Rozhraní ANTLR analyzátorů je dáno třídami *org.antlr.runtime.Lexer* a *org.antlr.runtime.Parser* (třídy jsou součástí modulu *AntlrWrapper*). Implementace těchto analyzátorů jsou získány pomocí lookupu. Modul *AntlrGsfBridge* poskytuje třídy *GeneratedLexer* a *GeneratedParser*, které implementují rozhraní daná GSF a vhodným způsobem delegují činnost analyzátorů na implementace

získané z lookupu. Modul *JstEditor* potom s výhodou použije potomky tříd *GeneratedLexer* a *GeneratedParser* a nemusí se starat o implementaci ani adaptaci analyzátorů.

Modul je implementován v balících *cz.muni.fi.antlrgsfbridge*, *cz.muni.fi.antlrgsfbridge.lexer* a *cz.muni.fi.antlrgsfbridge.parser*. Poslední dva jmenované balíky obsahují třídy realizující adaptaci analyzátorů, jak už názvy těchto balíků napovídají. Balík *cz.muni.fi.antlrgsfbridge* obsahuje třídu pomůcek *AntlrGsfBridgeUtils* a důležitou konfigurační třídu *LanguageConfiguration*. Aby bylo možné provézt adaptování analyzátorů, je potřeba kromě konkrétní implementace analyzátorů znát také název kořenového pravidla syntaktické gramatiky a mime typ, který je přiřazen souborům analyzovaného jazyka. Právě tyto informace jsou modulu poskytnuty pomocí atributů třídy *LanguageConfiguration*. Třída je také zodpovědná za získání implementací ANLTR analyzátorů z lookupu. Pokud je v lookupu k dispozici implementací jednoho z analyzátorů více, je brána první z nich (pořadí implementací je možné určit při registraci poskytovatele do lookupu). Alternativně je také možné předat do konstruktoru objektu *LanguageConfiguration* jako parametry plně kvalifikovaná jména tříd obsahujících implementace analyzátorů.

5.2.1 Adaptace lexeru – balík cz.muni.fi.antlrgsfbridge.lexer

Na úvod této kapitoly je potřeba si vysvětlit jaké jsou rozdíly mezi lexikálními analyzátory v ANTLR a v GSF (přesněji řečeno v platformě NetBeans IDE, protože Lexer je samostatný modul, který do GSF nepatří a GSF jej jen využívá). Oba tyto analyzátory sice provádějí lexikální analýzu nějakého vstupního dokumentu, ale každý za jiných podmínek a s jiným primárním účelem, proto jsou na tyto analyzátory také kladeny jiné požadavky.

Lexikální analyzátory generované nástrojem ANTLR pracují se vstupním souborem dávkově – tzn. dostanou na vstup soubor s neměnným obsahem, ten od začátku do konce analyzují, a jejich výstupem je výsledek této analýzy. Výsledkem analýzy je zpravidla seznam tokenů reprezentujících důležité části dokumentu a ten bývá většinou vstupem pro další analyzátory. Nedůležité části dokumentu, jako jsou například bílé znaky, můžou být při analýze bez problémů zahozeny a ve výsledném seznamu tokenů nemusí být součástí žádného tokenu. Stejným způsobem je možné zacházet také s řetězci, které do analyzovaného jazyka nepatří – analyzátor je jednoduše může přeskočit a zahodit. Toto chování analyzátoru nemusí být na škodu, protože další analyzátory, které dostanou seznam tokenů na svůj vstup se už nemusí zabývat nerelevantními nebo chybnými částmi analyzovaného dokumentu, protože lexer pro ně tyto části jednoduše odfiltruje.

Naproti tomu hlavním úkolem lexikálního analyzátoru platformy NetBeans je poskytnout informace potřebné k obarvení zdrojového kódu vstupního dokumentu. Tím je myšleno celého zdrojového kódu, tzn. včetně případných chyb, nebo řetězců, které do jazyka nepatří. Se vstupním dokumentem se totiž zrovna pracuje v editoru, a je proto pravděpodobné, že některé chyby jsou jen dosud nedokončené části slov jazyka. Je proto důležité, aby žádný znak nebyl přeskočen a každý znak vstupního dokumentu byl součástí právě jednoho tokenu. Při každé úpravě vstupního dokumentu je potřeba znovu obarvit zdrojový kód dokumentu, protože slovo po úpravě už může patřit jinému tokenu. Je proto zřejmé, že z hlediska výkonu není vhodné analyzovat znovu celý dokument, ale pouze části dokumentu, které mohla provedená změna ovlivnit. Aby bylo možné použít tento postup, analyzátor musí umět vyjádřit svůj vnitřní stav v každém bodu analýzy. Infrastruktura NetBeans potom ve chvíli, kdy nastane změna, použije tento stav k tomu, aby mohla restartovat lexer, a to ne od začátku dokumentu, ale až od místa, kde změna nastala, nebo přesněji řečeno od místa, které změna ovlivnila.

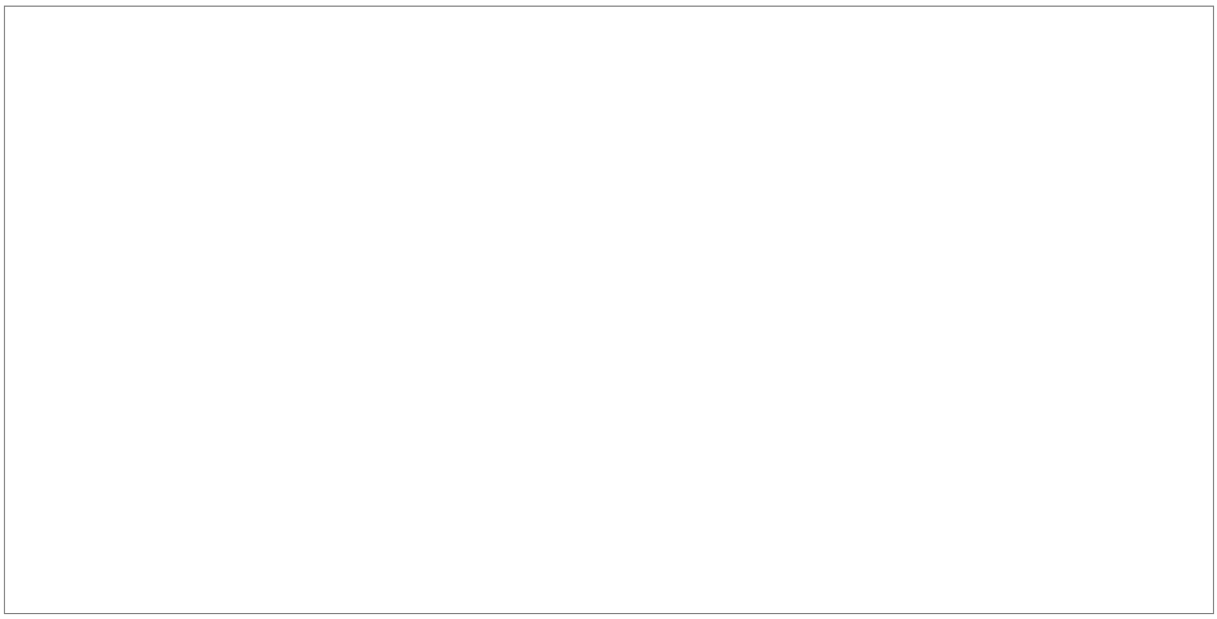
Všechny lexikální analyzátory generované pomocí ANTLR jsou potomky abstraktní třídy *org.anltr.runtime.Lexer*, která zapouzdřuje základní funkcionalitu každého ANTLR lexeru. Jak již bylo řečeno výše, každý ANTLR lexer však nemusí vyhovovat podmínkám, které si na lexikální analyzátor klade NetBeans. Proto modul *AntlrGsfBridge* poskytuje třídu *ErrorRecoveringLexer*, která je potomkem třídy *org.antlr.runtime.Lexer* a upravuje její funkcionalitu tak, aby vyhovovala podmínkám NetBeans. V gramatice ANTLR pro lexikální analyzátor je možné nastavit rodičovskou třídu, jejíž potomkem bude generovaný analyzátor. Pokud zde tedy nastavíme třídu *ErrorRecoveringLexer*, generovaný analyzátor bude vhodný pro použití v NetBeans IDE prostřednictvím modulu *AntlrGsfBridge*. Toto je slibované vysvětlení vazby číslo 5 z obrázku 12. Následuje shrnutí toho, jaké inovace třída *ErrorRecoveringLexer* představuje.

Třída *org.antlr.runtime.Lexer* obsahuje atribut *state*, který udržuje stav, v němž se analyzátor nachází. Tento atribut má však přístupová práva typu *protected*, tzn. je přístupný pouze třídám ve stejném balíku, nebo potomkům této třídy. Zároveň k tomuto atributu neexistuje žádná veřejná getmetoda, která by jej zpřístupňovala. Proto tuto metodu zavádí třída *ErrorRecoveringLexer* a díky ní tak mohou získat stav lexeru i jiné objekty. Set-metoda není potřeba, protože stav je možné nastavit při vytváření instance parametrem konstruktoru. Jak si ukážeme dále, analýza bude delegována na *ErrorRecoveringLexer* a právě díky tomu, že zpřístupňuje svůj stav ostatním objektům, bude možné provádět analýzu části dokumentu, jak bylo popsáno v předminulém odstavci. Další úpravou, kterou třída *ErrorRecoveringLexer* představuje je zpracování nerozpoznaných znaků. Analýza probíhá opakovaným voláním metody *nextToken*, která přečte jeden nebo více znaků ze vstupu, a vrátí token odpovídající těmto znakům. Pokud se při analýze narazí na znak nebo sérii znaků, které není možné přiřadit žádnému tokenu, je v původní implementaci lexeru vyhozena výjimka. Záleží potom na autorovi analyzátoru, jak se s těmito výjimkami naloží – jestli se analýza zastaví, jestli se tyto znaky přeskočí a pokračuje se v analýze dále, nebo jestli se stane něco jiného. V implementaci této metody ve třídě *ErrorRecoveringLexer* jsou tyto výjimky rovnou odchyceny a problematické znaky jsou přiřazeny tokenu typu *INVALID_TOKEN_TYPE*. Díky tomu je zajištěno, že všechny znaky vstupního dokumentu budou tokenizovány, a není potřeba aby toto zajišťoval autor analyzátoru.

Další důležitou třídou tohoto balíku je třída *GeneratedLexer*. Tato třída je aplikací návrhového vzoru adaptér – implementuje rozhraní *org.netbeans.spi.lexer.Lexer* NetBeans IDE a zapouzdřuje v sobě delegování analýzy na generovaný lexer – potomka třídy *ErrorRecoveringLexer*. Třída také řeší adaptování[15] tokenů, které jsou v ANTLR reprezentovány objekty typu *org.antlrruntime.Token* do podoby, ve které jsou reprezentovány v NetBeans, tj. jako objekty typu *org.netbeans.api.lexer.TokenId*. Právě potomky třídy *GeneratedLexer* je možné použít jako implementaci lexeru pro platformu NetBeans IDE (viz kapitola 5.3.3).

Další aplikací tohoto návrhového vzoru je třída *LexerInputReader*. Tato třída je potomkem abstraktní třídy *java.io.Reader* a adaptuje vstup z podoby, v jaké jej dostane lexer NetBeans (tj. objektu *org.netbeans.spi.lexer.LexerInput*), tak, aby se s tímto vstupem dalo pracovat právě jako s potomkem *java.io.Reader*, což je jedna z forem vstupu kterou akceptují analyzátory generované pomocí ANTLR.

15 Tato adaptace je poměrně přímočará záležitost, kdy pro každou reprezentaci typu tokenu v ANTLR je vytvořena odpovídající reprezentace tohoto typu tokenu v NetBeans. Praktický jediný rozdíl je v tom, že ANTLR vytváří zvláštní token EOF značící konec vstupního souboru, zatímco NetBeans v tomto případě používá hodnotu *null*.



Obrázek 13: Adaptace lexikálního analyzátoru

Na závěr neuškodí krátké shrnutí. Balík *cz.muni.fi.antlrgsfbridge.lexer* zprostředkovává adaptování lexikálního analyzátoru vygenerovaného pomocí nástroje ANTLR tak, aby jej bylo možné použít v NetBeans IDE. K tomuto účelu modul zavádí několik nových tříd. Tyto třídy jsou na obrázku 13 označeny pouze svým jménem (bez názvu balíku). Třídy, které jsou součástí ANTLR, nebo NetBeans IDE, jsou označeny plně kvalifikovaným jménem. Adaptace lexeru je rozdělena do dvou fází. Nejprve je potřeba zajistit, aby dodaný analyzátor splňoval určité podmínky, čehož je dosaženo děděním funkcionality třídy *ErrorRecoveringLexer*. V druhé fázi adaptace je potom takovýto vyhovující analyzátor zapouzdřen do třídy *GeneratedLexer*. Tato třída implementuje rozhraní dané NetBeans IDE a její potomky je proto možné použít jako implementaci lexeru pro platformu NetBeans IDE.

5.2.2 Adaptace parseru – balík cz.muni.fi.antlrgsfbridge.parser

Balík *cz.muni.fi.antlrgsfbridge.parser* obsahuje třídy potřebné k adaptování syntaktických analyzátorů generovaných pomocí ANTLR do formy, kterou je možné použít v NetBeans IDE. Na rozdíl od lexikálních analyzátorů, pro syntaktické analyzátory neklade platforma NetBeans IDE prakticky žádná omezení. Samozřejmě tyto analyzátory musí implementovat určité rozhraní a právě na toto rozhraní budeme ANTLR analyzátor adaptovat. V tomto případě se jedná o třídu *org.netbeans.modules.gsf.api.Parser*, která je součástí modulu GSF. Rozhraní definuje třídu *Job* a dále předepisuje dvě metody. Položky rozhraní jsou popsány v následujícím seznamu:

- **public final class Job** – třída zapouzdřuje parametry, které jsou předány analyzátoru. Celkem obsahuje následující 4 atributy:
  - **List<ParserFile> files** – seznam souborů k analyzování. Objekt typu *ParserFile* obsahuje metody potřebné pro práci se vstupním souborem.

- **ParseListener listener** – listener[16] hlídající události analyzátoru.
- **SourceFileReader reader** – objekt umožňující čtení obsahu vstupního souboru podobným způsobem jako *java.io.Reader*.
- **TranslatedSource translatedSource** – objekt reprezentující virtuální zdrojový kód[17] dokumentu v určitém jazyce.

- **void parseFiles(@NonNull Job request)** – metoda provede analýzu množiny souborů. Anotaci *NonNull* zavádí GSF. Cíl této anotace (ať už se jedná o parametr, proměnnou, atribut nebo návratovou hodnotu metody) nesmí mít hodnotu *null*. Odpadá proto nutnost kontrolovat tuto hodnotu na *null*. V průběhu analýzy mohou nastávat různé události. Vždy při úspěšném dokončení analýzy by měla nastat událost *finished*. Výsledek analýzy je připojen k vyvolané události jako objekt typu *ParserResult* (bude popsán dále).
- **@NonNull PositionManager getPositionManager()** – metoda vrací instanci třídy *PositionManager*. Tento objekt je zodpovědný za mapování objektů, které jsou výsledkem analýzy, na pozice ve skutečném zdrojovém kódu dokumentu (pokud se pracuje s virtuálními zdrojovými kódy, musí se přepočítat pozice ve skutečném dokumentu).

Výsledky analýzy jsou předány jako potomek abstraktní třídy *ParserResult*. Tato třída poskytuje základní informace o výsledku analýzy – tzn. informace jestli je výsledek validní, seznam chyb a podobně. V jaké formě bude předán samotný výsledek analýzy však žádné rozhraní ani třída nepředepisuje. Je tedy jen na autorovi parseru, v jaké formě se rozhodne výsledky analýzy poskytnout. Třída *ParserResult* sice definuje abstraktní metodu *getAst*, která vrací abstraktní syntaktický strom, tato metoda však slouží pouze pro ladící účely (zobrazení AST v NetBeans IDE), a není proto bezpodmínečně nutné poskytnout její smysluplnou implementaci.

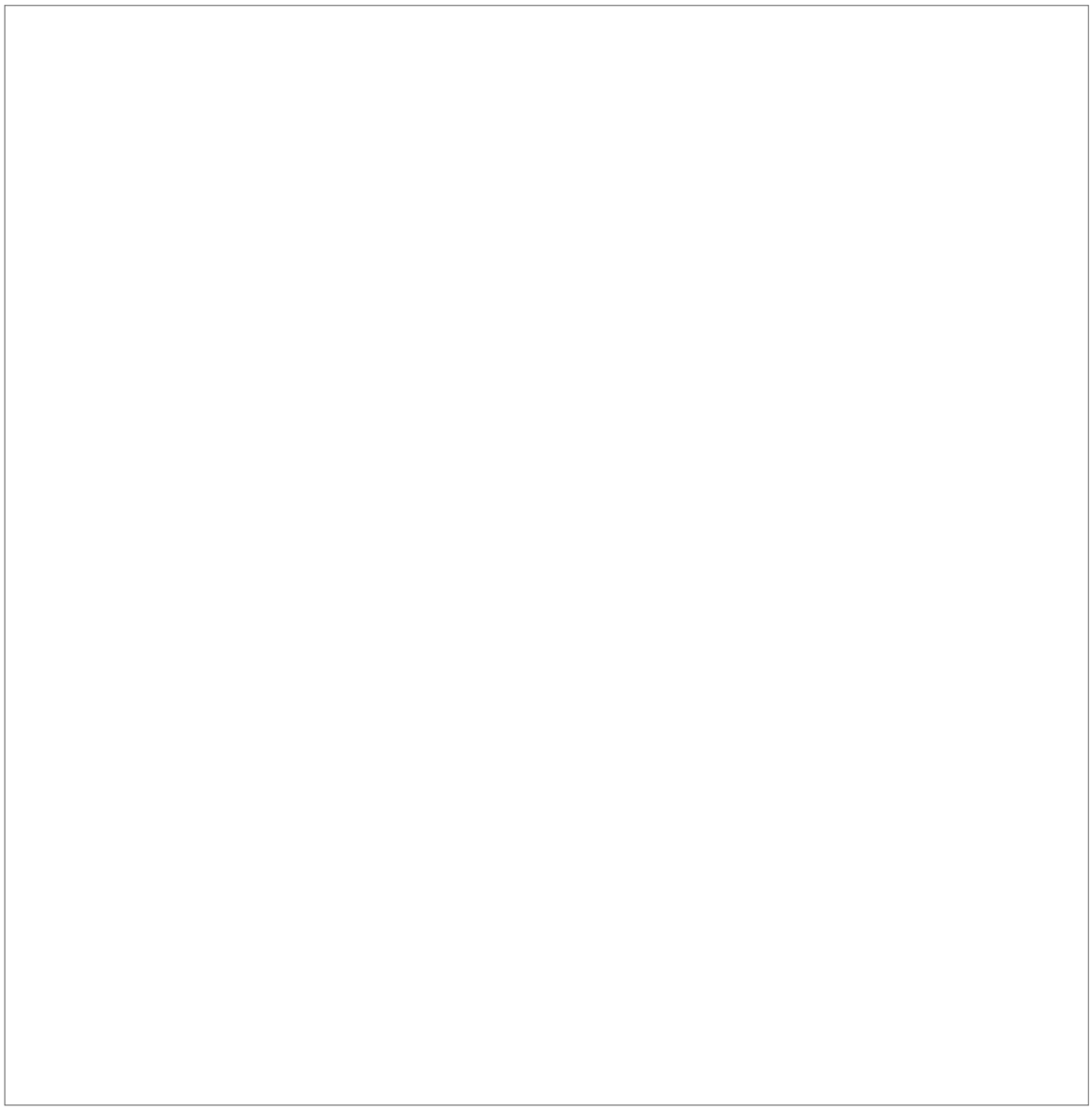
Všechny parsery generované pomocí ANTLR jsou potomky třídy *org.antlr.runtime.Parser*. Výstupy analýzy mohou vypadat různě podle účelu, ke kterému mají být použity. Jedno však mají všechny tyto parsery společné, a to že v průběhu analýzy je vytvářen derivační strom. Také ANTLR umožňuje registrovat k parseru listener, který bude poslouchat události nastávající v průběhu analýzy. K tomuto účelu však analyzátor musí být potomkem třídy *DebugParser* ze stejného balíku. K takovému analyzátoru je možné zaregistrovat listener implementující rozhraní *DebugEventListener*. Toto rozhraní definuje metody postihující všechny důležité události, které v průběhu analýzy mohou nastat. ANTLR také poskytuje několik implementací tohoto rozhraní, z nichž zmíníme jen jednu pro nás nejdůležitější – třídu *ParseTreeBuilder*. Jak název této třídy napovídá jedná se o implementaci listeneru, která podle událostí, jenž při analýze nastávají, buduje derivační strom a vrací tento strom jako instanci třídy *ParseTree*. Pro reprezentaci stromů ANLTR zavádí rozhraní *Tree* předepisující užitečné metody pro práci se stromy a *ParseTree* toto rozhraní implementuje.

V předchozích dvou odstavcích byla představena rozhraní obou systémů. Nyní tedy zbývá popsat způsob jakým modul *AntlrGsfBridge* provádí adaptaci parseru generovaného pomocí ANTLR tak, aby vyhovoval rozhraní modulu GSF. K tomuto účelu zavádí modul několik nových tříd. Stejně jako v

16 *Listener* je jiný název pro aplikaci návrhového vzoru průzkumník (observer) -tzn. umožňuje šíření událostí od objektu, ve kterém událost nastala, k libovolnému počtu objektů-posluchačů (listener).

17 Pokud se v dokumentu prokládají zdrojové kódy několika různých jazyků, jako například CSS v HTML, je možné vytvořit virtuální zdrojový kód, například pouze pro CSS, spojením bloků tohoto jazyka a zahozením bloků ostatních jazyků, nebo podobným postupem. Na takovémto virtuálním zdrojovém kódu je potom možné jednodušeji provézt například sémantickou analýzu.

minulé kapitole, také na obrázku 2 jsou tyto třídy označeny pouze svým jménem bez názvu balíku, zatímco třídy, které jsou součástí ANTLR nebo modulu GSF, jsou označeny plně kvalifikovaným jménem. Následuje popis tříd balíku *cz.muni.fi.antlrgsfbridge.parser*.



Obrázek 14: Adaptace syntaktického analyzátoru

Obdobně jako u adaptace lexeru, také zde existuje třída implementující rozhraní NetBeans a delegující analýzu na ANTLR analyzátor – třída *GeneradetParser*. Také v tomto případě se jedná o aplikaci návrhového vzoru adaptér. Při zavolání metody *parseFiles* je nejprve vytvořen ANTLR lexer stejným způsobem jako v minulé kapitole. Proud tokenů, které lexer vytvoří, je předán na vstup instanci ANTLR parseru, která je potomkem třídy *DebugParser*. K této instanci je přiřazen listener generující derivační strom, jak bylo popsáno výše. Následně je pomocí reflexe na této instanci

zavolána metoda provádějící analýzu podle kořenového pravidla gramatiky. V průběhu analýzy je vytvořen derivační strom a ten je následně předán jako parametr metodě *createParserResult*. Tato metoda se postará o vytvoření objektu zapouzdřujícího výsledek analýzy – potomka třídy *GeneratedParserResult*. Tento objekt je nakonec připojen k události finished, která je vyvolána při skončení analýzy (viz výše). Metoda *createParserResult* je aplikací návrhového vzoru template method – tzn. tato metoda je abstraktní, má za úkol vytvoření objektu *GeneratedParserResult*, ale jak je toho dosaženo neřeší a nechává to na starost konkrétní implementaci *GeneratedParseru*.

Třída *GeneratedParserResult* je potomkem třídy *ParserResult* a přináší s sebou dvě nové věci. Jednou z nich je, že poskytuje prázdnou implementaci abstraktní metody *getAst* a není proto nutné, aby byla tato třída abstraktní. Druhou novinkou pak je zavedení metody *getParseTree*, která vrací derivační strom vzniklý při analýze reprezentovaný objektem *ParseTreeNode*.

ParseTreeNode je další novou třídou kterou modul zavádí. Jak již bylo uvedeno v minulém odstavci, tato třída slouží k reprezentaci derivačního stromu, konkrétně jednoho uzlu stromu. Dříve v této kapitole bylo představeno rozhraní *Tree* a implementující třída *ParseTree*, které poskytuje ANTLR ke stejnému účelu. Použití těchto tříd však není vhodné z následujícího důvodu. Rozhraní *Tree* je navrženo přímo pro potřeby ANTLR analyzátorů. Listy stromů reprezentovaných implementacemi tohoto rozhraní obsahují přímo tokeny (objekty *org.antlr.runtime.Token*) nebo v případě chyb vyvolané výjimky. Při práci s takovýmto stromem v NetBeans by bylo potřeba znát větší množství tříd ANTLR a mít k těmto třídám přístup – tzn. na obrázku 12 by musela přibýt závislost mezi moduly *JstEditor* a *AntlrWrapper*. Proto modul *AntlrGsfBridge* zavádí vlastní reprezentaci derivačního stromu navrženou tak, aby ostatní moduly využívající tento modul byly oproštěny od přímé závislosti na ANTLR. Následuje popis této nové reprezentace.

Každý uzel *ParseTreeNode* obsahuje seznam potomků tohoto uzlu, proto je možné celý strom reprezentovat uzlem, který je kořenem tohoto stromu. Ke všem ostatním uzlům tohoto stromu se poté můžeme dostat rekurzivním procházením potomků, protože každý potomek je zároveň kořenem určitého podstromu. Kromě seznamu potomků je možné z každého uzlu (kromě kořene) získat také jeho rodiče. Kořen stromu samozřejmě rodiče nemá, proto má jeho rodič hodnotu *null*. Dále každý uzel obsahuje informace týkající se analýzy. Vnitřní uzly reprezentují jednotlivá pravidla gramatiky, podle kterých analýza probíhala. Listy stromu pak reprezentují konkrétní obsah tokenů – tzn. části textu ve vstupním dokumentu a ne tokeny samotné. Název pravidla, nebo obsah tokenu je v uzlu uložen jako textový řetězec v atributu *type*. Jestli se jedná o název pravidla, nebo o obsah tokenu je potom možné zjistit pomocí metody *isLeaf*, která vrací jestli je uzel listem, nebo vnitřním uzlem stromu. Každý uzel také obsahuje atributy *startOffset* a *endOffset*, které udávají začátek a konec části vstupního dokumentu, kterou pokrývá daný uzel nebo podstrom tohoto uzlu. Tento kus vstupního dokumentu je možné získat pomocí metody *getText*. Dále pak třída *ParseTreeNode* obsahuje ještě několik metod pro práci s podstromy uzlů a také informace o chybách, které při analýze nastaly. Pokud se ve vstupním dokumentu vyskytují chyby, nemusí být derivační strom vytvořen pro celý dokument, ale pouze pro jeho bezchybné části. Chování v případě chyb ve vstupním dokumentu záleží na konkrétní implementaci analyzátoru. V každém případě by však informace o chybách měly být součástí výsledků analýzy. Objekty *ParseTreeNode* obsahují atribut, který udržuje seznam informací o chybách, jenž se vyskytly při analýze potomků tohoto uzlu. K reprezentaci těchto informací slouží instance třídy *ErrorInfo*, která je také součástí modulu *AntlrGsfBridge*. Třída obsahuje atributy nesoucí tyto informace:

- začátek a konec chybné části dokumentu.

- číslo řádku a pozici na řádku, kde se chyba vyskytla.
- typ chyby – zároveň třída definuje 3 typy chyb – chybějící token, neočekávaný token a předčasné dosažení konce souboru.
- seznam parametrů – záleží na konkrétním použití – například pokud je chyba neočekávaný token, může být jako parametr chyby předán seznam povolených tokenů a podobně.

Na závěr této kapitoly zbývá představit ještě jednu třídu tohoto balíku – třídu *AntlrParseTreeAdapter*. Jak už její název napovídá, třída má na starosti adaptaci derivačního stromu reprezentovaného objekty *org.antlr.runtim.tree.ParseTree* do reprezentace pomocí objektů *ParseTreeNode*. Adaptace vnitřních uzlů je triviální záležitost. Při adaptaci listů ve kterých jsou uloženy tokeny jsou z těchto tokenů extrahovány pozice začátku a konce textu, který token pokrývá. Poté je s využitím těchto informací vytvořen nový list *ParseTreeNode*. Při vytvoření nového listu jsou ověřeny a v případě potřeby aktualizovány informace o začátku a konci pokrytého textu také u rodičovského uzlu (a rekurzivně dále u všech rodičů až po kořen stromu). Při adaptaci listů ANTLR stromu, ve kterých je obsažena výjimka vyvolaná při chybě, jsou z této výjimky získány informace o chybě a na základě těchto informací vytvořen objekt *ErrorInfo* a ten přiřazen rodičovskému uzlu *ParseTreeNode*. Podle druhy výjimky je nastaveny vhodná hodnota atributu *type* objektu *ErrorInfo* a do seznamu parametrů je předán buď jen název typu tokenu, který chybu způsobil, nebo ještě také název typu tokenu, který byl očekáván (podle druhu výjimky). Tyto informace je možné využít k podrobnějšímu popisu chyby uživateli.

5.3 Modul JstEditor

Tento modul realizuje základ vývojového prostředí. Jak bylo uvedeno v kapitole 5.1, modul implementuje celou řadu rozhraní NetBeans IDE a k tomu také mimo jiné využívá služeb modulu *AntlrGsfBridge*. Třídy modulu jsou organizovány do několika balíků podle funkcionality kterou zajišťují. V průběhu vývoje jsem narazil na dilema, jestli některé balíky neoddělit do samostatných modulů, protože funkcionalita, kterou realizují je jasně vymezena. Nakonec jsem se rozhodl ponechat všechny balíky v tomto jednom modulu, protože jimi poskytovaná funkcionalita je skutečným základem vývojového prostředí a proto je vhodné, aby byl tento základ zapouzdřen v jediném modulu. Jednotlivým balíkům budou postupně věnovány další podkapitoly.

Názvy všech balíků začínají stejným prefixem *org.netbeans.modules.jstlexer*. Pro zjednodušení bude v dalším textu tento prefix v názvech balíků vynecháván. Implementace některých tříd jsou triviální, nebo například byly vytvořeny pomocí průvodce, kterého poskytuje NetBeans IDE, proto také popis těchto tříd bude velice stručný a víceméně bude pouze popisováno API, které tyto třídy implementují. Implementace jednotlivých API jsou převážně registrovány do lookupu pomocí souboru *layer.xml*, proto relevantní části tohoto souboru budou postupně popisovány v příslušných podkapitolách. Téměř všechny balíky obsahují třídy implementující nějaké rozhraní definované modulem GSF, proto budou v jednotlivých podkapitolách také stručně popsána tato rozhraní.

5.3.1 Balík filetype

Tento balík obsahuje třídy potřebné k tomu, aby platforma NetBeans byla schopna rozpoznat nový typ souboru a uměla s tímto typem souboru pracovat. S výhodou můžeme využít průvodce, kterého nabízí NetBeans IDE a nechat si tak tyto třídy a soubory automaticky vygenerovat. Průvodce také automaticky provede registraci v konfiguračním souboru *layer.xml*. Hierarchie tříd pro práci se

soubory byla zobrazena na obrázku 8 v kapitole 4.1.2.1 a jednotlivé generované soubory pracují na různých úrovních této hierarchie. Následuje krátký popis vytvořených souborů:

- **JstDataObject.java** – datový objekt zapouzdřující objekty typu *FileObject* reprezentující skutečné soubory na disku. *FileObject* reprezentoval soubor jako proud bytů. *DataObject* reprezentuje soubor na vyšší úrovni. Operace, které je možné se souborem provádět, je možné k tomuto datovému objektu přiřadit pomocí lookupu. Generovaná třída je upravena tak, že implementuje rozhraní *CookieSet.Factory*. Pomocí mechanismu cookies (viz kapitola 4.1.3.1) je k tomuto datovému objektu přiřazeno několik schopností – uložení, uložení jako, schopnost práce s dokumentem v editoru, pozorovatelnost (v tomto případě zajištěna implementací rozhraní *EditorCookie.Observable* – návrhový vzor *observer* neboli *pozorovatel* neboli *listener*). Akce využívají schopností, které poskytují jednotlivé cookies.
- **JstDataLoader.java** – data loader má na starosti rozpoznání souboru reprezentovaného jako objekt *FileObject* a vytvoření odpovídajícího objektu *DataObject*. Je možné rozpoznávat skupinu spolu souvisejících souborů jako jeden datový objekt (implementováním abstraktní třídy *MultiFileLoader*). V našem případě stačí rozpoznávat vždy jen jeden soubor jako jeden datový objekt, proto je implementována třída *UniFileLoader*. Instance loaderu je registrována v souboru *layer.xml* ve složce *Loader/MIME-TYP*[18]*/Factories*.
- **JstDataLoaderBeanInfo.java** – doplňující informace pro data loader. V tomto případě slouží k nastavení ikony použité pro daný typ souboru.
- **JstDataNode.java** – reprezentace datového objektu na prezentační vrstvě. K objektům *DataNode* je možné registrovat akce, které může uživatel s tímto objektem (potažmo s datovým objektem) provádět. Akce jsou implementace abstraktní třídy *SystemAction* a jejich instance jsou registrovány v souboru *layer.xml* ve složce *Loaders/MIME-TYP/Actions*. Tato složka je nastavena v *DataLoader*u a vrací ji metoda *actionsContext*. NetBeans IDE poskytuje celou řadu již hotových implementací běžně používaných akcí. Nové akce není nutné registrovat do souboru *layer.xml* ručně, ale je opět možné využít průvodce, kterého nabízí NetBeans IDE.
- **JstResolver.xml** – resolver má za úkol zjistit MIME typ příslušící k určitému *FileObjectu*. Je možné registrovat jej do lookupu jako instanci implementace abstraktní třídy *MIMEResolver*, nebo pomocí takovéhoto jednoduchého xml souboru, který je zaregistrován v souboru *layer.xml* ve složce *Services/MIMEResolver*. Soubor obsahuje informace o rozeznávaných příponách a název MIME typu, který k souborům s těmito příponami patří. NetBeans nejprve interpretuje resolvery registrované jako xml, potom hledá vhodný resolver v lookupu, a rozpoznávání končí ve chvíli, kdy některý resolver úspěšně rozpozná MIME typ, nebo pokud jsou všechny dostupné resolvery neúspěšně vyčerpány.
- **JstTemplate.jst** – šablona, která bude použita pokud uživatel bude chtít vytvořit nový JST soubor. Šablony jsou registrovány v souboru *layer.xml* ve složce *Templates*, případně v podsložce této složky (podle kategorie do které by měla šablona spadat). Je možno registrovat více šablon připravených pro různé účely.

18 Řetězec *MIME-TYP* použitý v textu je zástupný symbol pro skutečný MIME typ. Pokud se v názvu MIME typu objevuje znak „/“, je v souboru *layer.xml* potřeba rozdělit název do podsložek podle toho, jak jej lomítko rozděluje. V našem případě například se jedná o MIME typ „text/x-ool-jst-template“, proto je použita složka „text“ obsahující podsložku „x-ool-jst-template“.

Dále se v tomto balíku nachází ještě jedna důležitá třída, která už však průvodcem vygenerována nebyla. Jedná se o třídu *JstEditor*, potomka třídy *DataEditorSupport*. *DataEditorSupport* umožňuje přiřadit editor a *javax.swing.text.Document* k datovému objektu. Tato třída také implementuje několik metod různých cookies, ale přitom neimplementuje rozhraní těchto cookies. Záleží tedy na potomkovi této třídy, rozhraní kterých cookies se rozhodne skutečně implementovat. *JstEditor* implementuje několik následujících rozhraní a přidává tak datovému objektu následující schopnosti:

- **OpenCookie** – umožňuje datovému objektu, aby mohl být otevřen.
- **EditorCookie** – definuje standardní operace s textovým dokumentem a s editorem, který tento dokument zobrazuje (otevření souboru, zavření editoru, nahrávání na pozadí, uložení, upozornění na změny).
- **EditCookie** – umožňuje datovému objektu, aby mohl být editován.
- **EditorCookie.Observable** – rozšiřuje *EditorCookie* o pozorovatelnost – přidává možnost přiřadit datovému objektu *PropertyChangeListener*.

JstEditor také implementuje metody *notifyModified* a *notifyUnmodified*, které jsou volány pokud je dokument modifikován (respektive pokud jsou modifikace vráceny zpět), a v těchto metodách je datovému objektu přidána nebo odebírána schopnost uložit se (*SaveCookie*).

5.3.2 Balík editor

Balík *editor* se oproti ostatním balíků trochu vymyká tím, že nezapouzdřuje nějakou ucelenou část funkcionality, ale spíše obsahuje třídy, které mají spíše konfigurační charakter a jsou využívány ostatními balíky a moduly. Konkrétně se jedná tři třídy – *JstKit*, *JstLanguage* a *JstLanguageConfiguration*.

Třída *JstKit* je rozšířením třídy *javax.swing.text.EditorKit*. *EditorKit* určuje množinu věcí, které potřebuje obyčejná textová komponenta k tomu, aby mohla sloužit jako editor nějakého druhu textového souboru. Je to vlastně jakási sada tříd, kterou využívá objekt *JEditorPane* k tomu, aby se mohl přeměnit v komponentu editoru pro určitý MIME typ. Pokud otevřeme soubor jiného typu, změní se přiřazený *EditorKit* za jinou implementaci (pro daný typ souboru) a rázem se *JEditorPane* promění v editor tohoto typu souboru. *EditorKit* implementuje rozhraní *Serializable* a *Cloneable*, což znamená, že tyto objekty je možné uložit do binární formy, a že je možné vytvářet nové instance kopírováním již existujících instancí. Nové instance se většinou vytvářejí klonováním prototypu určitého *EditorKit*u. Existuje několik rozšiřujících tříd, které poskytuje přímo Swing, a potom také několik rozšíření, které poskytují moduly platformy NetBeans.

Protože jazyk šablon je z části podobný jazyku HTML, je *JstKit* potomkem *HTMLKit*u, který je součástí modulu *HTML Editor*. Díky tomu dědí veškerou funkcionalitu, kterou *HTMLKit* poskytuje. V průběhu vývoje byla část funkcí přeimplementována a je pravděpodobné, že v dalším vývoji budou vytvořeny nové implementace tolika funkcí, že už nebude potřeba aby *JstEditor* byl potomkem třídy *HTMLKit*. Navíc GSF nabízí vlastní implementaci *EditorKit*u, který nabízí funkce obecně používané v editorech programovacích jazyků, proto možná bude v budoucnu vhodné aby *JstKit* byl spíše potomkem této třídy. Situace se má totiž tak, že v dřívějších verzích NetBeans, byla převážná část funkcí a chování editoru implementována právě v *EditorKit*ech. Postupem času se ale v NetBeans objevovaly nové postupy jak různé funkce implementovat a vznikaly nová API. Jedním z nejdůležitějších počinů v tomto směru byly právě moduly Lexer a GSF (viz kapitola 5.1), které mimo

jiné představují vlastní implementace *EditorKit*u. Tyto implementace abstrahují velkou část funkcí, které se běžně v editorech programovacích jazyků vyskytují a zároveň u těchto funkcí nabízejí buď přímo jejich implementace (např. automaticky vytvořené na základě poskytnutých analyzátorů), nebo pro tyto funkce zavádějí API. Vývojář potom vytvoří třídu poskytující určitou novou službu (například implementuje rozhraní *BracketCompletion*, které poskytuje metody pro automatické doplňování a hledání párových závorek a znaků v textu) a potom tuto třídu jen zaregistruje do lookupu a nemusí vůbec stávající *EditorKit* editovat ani rozšiřovat.

Vlastní implementace *EditorKit*u se registrují v souboru *layer.xml* ve složce *Editors/MIME-TYP*. Při použití vlastního *EditorKit*u je navíc potřeba v tomto souboru v konfigurační sekci modulu GSF nastavit atribut *useCustomEditorKit* na hodnotu *true*, jak bude vidět na následující ukázce. Podle toho GSF pozná, že nemá vytvářet vlastní *EditorKit*, ale použít dodaný. Tím získáme podporu funkcí implementovaných v daném *EditorKit*u, ale ztratíme podporu funkcí, které poskytuje GSF (které by byly automaticky volány z *GsfEditorKit*u). To se dá naštěstí napravit tak, že jednotlivé služby GSF, které poskytují tyto funkce, zaregistrujeme v souboru *layer.xml* sami (bude popsáno v následujících kapitolách). Aby modul GSF mohl vůbec nějaké služby poskytovat, tak ale také potřebuje určitých služeb využívat – máme na mysli implementace analyzátorů.

V současné době existují dva způsoby jak registrovat služby, které modul GSF využívá. První způsob je registrace služeb v konfigurační sekci modulu GSF v souboru *layer.xml*. Druhá možnost je implementováním rozhraní *GsfLanguage*. Přestože si můžeme vybrat, kterou z těchto možností budeme preferovat, žádné z nich se zároveň nemůžeme vyhnout. Třída *JstLanguage* implementuje rozhraní *GsfLanguage* a její instance musí být registrována v souboru *layer.xml* jak je vidět na následující ukázce:

```
<folder name="GsfPlugins">
   <folder name="text">
      <folder name="x-ool-jst-template">
         <attr name="useCustomEditorKit" boolvalue="true"/>
         <file name="language.instance">
            <attr name="instanceOf" stringvalue="org.netbeans.modules.gsf.api.GsfLanguage"/>
            <attr name="instanceClass"
stringvalue="org.netbeans.modules.jstlexer.editor.JstLanguage"/>
         </file>
         <file name="parser.instance">
            <attr name="instanceOf" stringvalue="org.netbeans.modules.gsf.api.Parser"/>
            <attr name="instanceClass"
stringvalue="org.netbeans.modules.jstlexer.parser.JstGSFParser"/>
         </file>
         <file name="structure.instance">
            <attr name="instanceOf" stringvalue="org.netbeans.modules.gsf.api.StructureScanner"/>
            <attr name="instanceClass"
stringvalue="org.netbeans.modules.jstlexer.structure.JstStructureScanner"/>
         </file>
      </folder>
   </folder>
</folder>
```

Dále v souboru *layer.xml* musí být registrována implementace rozhraní *StructureScanner* (pokud ji poskytujeme) a také dříve zmiňovaný atribut *useCustomEditorKit* (pokud používáme vlastní implementaci *EditorKit*u). K tomuto existují implementační důvody (viz dokumentace modulu GSF).

Ostatní služby je možné registrovat buď také touto cestou, nebo implementací metod konfigurační třídy *JstLanguage*. Alternativní způsob jak registrovat parser je možné vidět na následující ukázce:

```
@Override
public Parser getParser() {
    return new JstGSFParser();
}
```

Pokud chceme, aby modul GSF mohl využívat implementaci lexeru, musíme ji zaregistrovat právě implementováním metody třídy *JstLanguage*:

```
@Override
public Language getLexerLanguage() {
    return JstLexerLanguage.getLanguage();
}
```

Toto je zase důvod, proč se nelze vyhnout použití této konfigurační třídy. Registrace lexeru je zvláštní v tom, že je potřeba jej zároveň registrovat i v souboru *layer.xml*. Tentokrát ale v sekci *Editors/MIME-TYP* a to proto, že služeb lexeru nevyužívá jen modul GSF, ale také jiné moduly NetBeans. Tato redundance, stejně jako ostatní nejasnosti kolem registrace služeb pro modul GSF, by měla být později v konečné verzi tohoto modulu nějakým způsobem vyřešena. Ve verzi, která byla k dispozici v době psaní této práce, se však vyskytovaly výše popsané zvláštnosti.

Třetí a poslední třídou balíku *editor* je třída *JstLanguageConfiguration*. Tato třída obsahuje statickou metodu *getLanguageConfiguration* vracející instanci *LanguageConfiguration*, což je konfigurační třída modulu *AntlrGsfBridge*. Jedná se o aplikaci návrhového vzoru singleton, protože stačí jedna instance této konfigurační třídy pro celou aplikaci. Implementace konfigurační třídy poskytuje informaci o MIME typu souborů se kterými budou adaptované analyzátory pracovat, název kořenového pravidla gramatiky podle které byl generován parser a poskytuje instance generovaných analyzátorů, které jsou získány pomocí lookupu.

5.3.3 Balík lexer

Balík *lexer* obsahuje třídy poskytující NetBeans lexikální analyzátor a informace o jazyku, který lexikální analyzátor rozpoznává. Na základě těchto informací potom NetBeans (respektive modul Lexer) poskytuje barvení syntaxe zdrojového kódu zobrazeného v editoru[19]. Jak bude vysvětleno dále, je zde také popsáno jakým způsobem se má vytvářet hierarchie jazyků pokud jsou jazyky prokládány. Znalost hierarchie jazyků je důležitá také pro jiné funkce, které budou popsány v dalších kapitolách.

První třídou balíku je třída *JstLexer*, která je potomkem třídy *GeneratedLexer* z modulu *AntlrGsfBridge*. Analýzu provádí analyzátor generovaný pomocí ANTLR a *GeneratedLexer* v sobě zapouzdřuje delegování povinností na tento generovaný analyzátor (viz kapitola 5.2.1). Jediným

19 K problematice barvení syntaxe zdrojového kódu existuje několik návodů [11], [12] K plnému pochopení možností modulu Lexer doporučuji si tyto návody projít, protože text kapitoly popisuje spíše konkrétní použití v realizovaném pluginu.

úkolem třídy *JstLexer* tak je předat svému předkovi potřebné konfigurační informace. Toto je realizováno pomocí konstruktoru:

```
public JstLexer(LexerRestartInfo<TokenId> info) {
   super(JstLanguageConfiguration.getConfiguration(), info);
}
```

Další třídou tohoto balíku je třída *JstLexerLanguage*. Tato třída poskytuje statickou metodu *getLanguage*, která vrací objekt typu *Language* získaný z hierarchie jazyků – objektu *LanguageHierarchy*. Pokud se totiž ve zdrojovém kódu vstupního souboru vyskytuje několik jazyků, je potřeba popsat hierarchii těchto jazyků – tzn. na kterých místech a jakým způsobem se mohou tyto jazyky navzájem prokládat. Hierarchie jazyků tak popisuje jazyk, lexer pro tento jazyk a vložené jazyky. Při analýze dokumentu jsou potom povinnosti delegovány na analyzátory jednotlivých jazyků právě podle této hierarchie jazyků. Každý jazyk (objekt *Language*) obsahuje množinu typů tokenů, které se mohou v daném jazyce vyskytovat a MIME typ odpovídající tomuto jazyku. Množinu typů tokenů a MIME typ nám poskytuje konfigurační objekt *JstLanguageConfiguration*. Lexer je reprezentován třídou *JstLexer*. Takže jediné co zbývá v této třídě vytvořit je hierarchie prokládání jazyků – implementace metody *embedding*, která je vidět na následující ukázce:

```
protected LanguageEmbedding<TokenId> embedding(Token<TokenId> token,
                                LanguagePath languagePath,
                                InputAttributes inputAttributes) {

        String mimeType = null;
        switch (token.id().ordinal()) {
           case HTML_TOKEN_ID :
              mimeType=HTML_MIME_TYPE;
              break;
           case SCRIPT_TOKEN_ID :
              mimeType=SCRIPT_MIME_TYPE;
              break;
        }

        if(mimeType != null) {
           Language lang = Language.find(mimeType);
           if(lang == null) {
              return null; //no language found
           } else {
              return LanguageEmbedding.create(lang, 0, 0, true);
           }
        }

        return null;
     }
```

Z této ukázky je patrné, že místa, kde se vytváří další vrstva hierarchie, jsou tokeny typů *HTML_TOKEN_ID* a *SCRIPT_TOKEN_ID*. Pro tyto tokeny je vložen do hierarchie jazyk odpovídající danému MIME typu. Proto náš lexer nemusí rozeznávat jednotlivé tokeny těchto vložených jazyků, ale stačí, když rozpozná celý blok vloženého jazyka jako jediný token určitého typu a k tomuto typu tokenu je poté v hierarchii přiřazen vložený jazyk.

Pro práci s jazykem je potřeba znát nejen konkrétní lexer, ale také doplňující informace jako je přiřazený MIME typ a hierarchie jazyka. Proto se registruje objekt *Language* získaný z dané hierarchie jazyka, který všechny tyto potřebné informace poskytuje. Registrace pro modul GSF probíhá implementováním metody *getLexerLanguage* rozhraní *GsfLanguage* a je popsána v předchozí kapitole. V této kapitole také bylo uvedeno, že jazyk je potřeba registrovat také pro jiné moduly NetBeans a to pomocí následujícího kódu vloženého do složky *Editors/MIME-TYP* v souboru *layer.xml*:

```
<file name="language.instance">
   <attr name="instanceCreate"
         methodvalue="org.netbeans.modules.jstlexer.lexer.JstLexerLanguage.getLanguage"/>
   <attr name="instanceOf" stringvalue="org.netbeans.api.lexer.Language"/>
</file>
```

Ve stejné složce souboru *layer.xml* se také vyskytuje složka *FontColors*, která obsahuje konfiguraci barev pro jednotlivé typy tokenů. Přesněji řečeno v této složce je uveden odkaz na jiný xml soubor[20] obsahující tuto konfiguraci. Hodnoty barev je možné změnit v nastavení aplikace. Názvy typů tokenů zobrazené v okně nastavení jsou lokalizovány pomocí properties souboru registrovaného ve stejné složce. V tomto nastavení je také vidět náhled zdrojového kódu v příslušných barvách. Soubor s vzorovým zdrojovým kódem pro náhled je také registrován v souboru *layer.xml* ve složce *OptionsDialog/PreviewExamples*.

5.3.4 Balík coloring

Jak název balíku napovídá, tento balík obsahuje třídy zajišťující barvení zdrojového kódu v editoru. Barvení syntaxe zdrojového kódu poskytuje modul Lexer a třídy k tomuto účelu potřebné byly popsány v předchozí kapitole. Třídy v tomto balíku však ještě navíc umožňují zabarvit pozadí vložených jazyků.

K ovlivňování toho, jak bude zdrojový kód v editoru vypadat, poskytuje NetBeans rozhraní Highlighting SPI[13]. Toto rozhraní mimochodem využívá ke zvýrazňování syntaxe také modul Lexer. Hlavní myšlenkou je, že dokument v editoru je vykreslován jako výsledek projekce několika nezávislých vrstev. Části dokumentu, jejichž vykreslení vrstva upravuje, se nazývají zvýraznění (highlights). Každá vrstva poskytuje libovolný počet nepřekrývajících se zvýraznění. Různé moduly potom poskytují libovolný počet různých vrstev. NetBeans nakonec vezme všechny vrstvy registrované k danému typu souboru a složí dohromady všechna zvýraznění, která tyto vrstvy poskytují. Výsledek určuje konečnou podobu, v jaké bude dokument zobrazen v editoru. Samozřejmě záleží na pořadí v jakém budou vrstvy skládány, protože zvýraznění v jednotlivých vrstvách se mohou navzájem překrývat.

Vrstva je reprezentována pomocí třídy *HighlightsLayer* a její instance jsou vytvářeny tovární metodou *create*, kterou poskytují implementace rozhraní *HighlightsLayerFactory* - *JstHighlightsLayerFactory*. Jeden z parametrů, které jsou předávány této metodě je instance objektu *ZOrder* určující pořadí vrstvy, nebo-li výšku ve které se vrstva nachází. Tento objekt zavádí několik „škatulek“, do kterých je možné vrstvy řadit podle účelu zvýraznění, jež poskytují. Potom stačí číselně určit pozici vrstvy ve škatulce – čím vyšší číslo, tím výše bude i vrstva. Jednotlivé škatulky jsou ve

20 Soubor pro konfiguraci barev je definován pomocí tohoto DTD: http://www.netbeans.org/dtds/EditorFontsColors-1_1.dtd

skutečnosti pouze číselné konstanty, ke kterým se toto číslo nakonec přičte. Dalším důležitým parametrem předávaným metodě je implementace rozhraní *HighlightsContainer* – *JstHighlighstContainer*. Tento objekt nese informace o zvýrazněných oblastech.

Třída *JstHighlightsContainer* navíc implementuje také rozhraní *TokenHierarchyListener*, a je registrována jako pozorovatel hierarchie tokenů dokumentu. To znamená, že pokud nastane změna v dokumentu, a podle hierarchie jazyků je lexery vytvořena nová hierarchie tokenů, pak je mimo jiné také zavolána metoda *tokenHierarchyChanged* této třídy. Díky tomu jsou vytvořena nová zvýraznění vždy když je potřeba (viz komentář ve zdrojovém kódu třídy). Obsah tohoto kontejneru vrací metoda *getHighlights* jako sekvenci zvýraznění – objekt *HighlightsSequence*. Implementace této metody nejprve z hierarchie tokenů získá sekvenci tokenů jazyka šablon. Poté prochází jednotlivé tokeny v této sekvenci, a když narazí na typ tokenu, který zastupuje blok vloženého jazyka HTML nebo JavaScript, tak vytvoří nové zvýraznění. Počáteční a koncová pozice zvýraznění odpovídají počátku a konci tokenu. Atributy zvýraznění jsou získány z konfigurace aplikace (složka *FontColors* v souboru *layer.xml* – viz minulá kapitola).

Registrace vlastních vrstev se provádí přidáním instance tovární třídy, která tyto vrstvy vytváří, do složky *Editors/MIME-TYP* souboru *layer.xml*.

5.3.5 Balík parser a balík ast

Balík *parser* poskytuje NetBeans (respektive modulu GSF) syntaktický analyzátor a několik dalších tříd týkajících se syntaktické analýzy. Na základě výsledků syntaktické analýzy potom může vývojové prostředí poskytovat další funkce jako je skládání kódu, navigátor a podobně. Třídy tohoto balíku využívají tříd z balíku *ast*, který bude pouze v kostce shrnut v následujícím odstavci.

Balík *ast* obsahuje třídy reprezentující jednotlivé uzly abstraktního syntaktického stromu. Všechny uzly jsou potomky třídy *AstNode*, která obsahuje metody nutné pro práci s uzly jako se stromovou strukturou (obdoba třídy *ParseTreeNode* popsané v kapitole 5.2.2). K provádění operací na uzlech stromu je definováno rozhraní *AstNodeVisitor* (aplikace návrhového vzoru visitor – návštěvník). Dále balík obsahuje rozhraní *AstTreeBuilder*, jehož implementace zajišťují vytvoření abstraktního syntaktického stromu z derivačního stromu. Další třídou je *AstPath*, která reprezentuje cestu mezi dvěma uzly abstraktního syntaktického stromu a díky její metodě *equals* je možné tyto cesty jednoduše porovnávat. Poslední třídou balíku je třída *AstUtils* poskytující pomocné metody usnadňujících práci s abstraktním syntaktickým stromem.

Patrně nejdůležitější třídou balíku *parser* je třída *JstGSFParser*. Tato třída je implementací abstraktní třídy *GeneratedParser* z modulu *AntlrGsfBridge*. Analýzu opět provádí analyzátor generovaný pomocí ANTLR a *GeneratedParser* v sobě zapouzdřuje delegování povinností na tento generovaný analyzátor. Stejně jako tomu bylo u lexeru, také tato třída předává v konstruktoru svému předkovi potřebné konfigurační informace. *JstGSFParser* navíc implementuje abstraktní metodu *getPositionManager* vracející instanci *JstPositionManager* (viz dále) a metodu *createParserResult*, která vrací novou instanci objektu zapouzdřujícího výsledek analýzy. V tomto případě se jedná o instanci objektu *JstParserResult*, který je potomkem *GeneratedParserResult*.

JstParserResult dědí po svém předkovi metodu *getParseTree* vracející výsledky analýzy ve formě derivačního stromu. Tato třída však navíc poskytuje metodu *getAstRoot*, která vrací výsledky analýzy ve formě abstraktního syntaktického stromu reprezentovaného kořenovým uzlem *AstNode*. Metoda využívá služeb třídy *JstAstTreeBuilder*, která je implementací rozhraní *AstTreeBuilder*, což znamená,

že umí vytvořit abstraktní syntaktický strom podle dodaného derivačního stromu. Třída navíc nabízí novou implementaci metody *getAst*. Tato metoda slouží pouze pro ladící účely a vrací abstraktní syntaktický strom reprezentovaný objekty *AstTreeNode*. Modul GSF potom umožňuje zobrazit strom získaný pomocí této metody v samostatném okně vývojového prostředí. Adaptaci stromu do této reprezentace zajišťuje třída *AstTreeNodeAdapter*. Jak již bylo řečeno v kapitole 5.2.2, pokud při syntaktické analýze vstupního souboru nastanou chyby, jsou informace o těchto chybách dostupné v příslušných uzlech derivačního stromu jako seznam objektů *ErrorInfo*. Objekty *ParserResult* ale nesou informace o chybách jako seznam objektů *Error.* Proto *JstParserResult* při svém vytvoření prochází derivační strom, a pro jednotlivé objekty *ErrorInfo* na které narazí, vytváří odpovídající objekty *JstError* a naplňuje jimi tento svůj seznam. Na základě tohoto seznamu potom modul GSF zobrazuje informace o chybách v editoru. Procházení stromu je opět realizováno pomocí návrhového vzoru návštěvník – implementací rozhraní *ParseTreeNodeVisitor*.

Protože generovaný syntaktický analyzátor je bezkontextový, nemůže zajistit hlídání párování otevíracích a zavíracích makro značek. *JstAstTreeBuilder* ale při zpracovávání derivačního stromu a vytváření abstraktního syntaktického stromu využívá zásobník, a díky tomu může snadno zjistit, jestli se otevírací a zavírací značky vyskytují na stejných úrovních a jestli k sobě patří. Pro neúspěšně uzavřenou makro značku je vytvořen jiný typ uzlu v abstraktním syntaktickém stromu, než pro makro uzavřené korektně. Poté, co je ve třídě *JstParserResult* vytvořen abstraktní syntaktický strom, jsou pro všechny nesprávně uzavřená makra vytvořeny nové instance *UnmatchedTagWarning* (potomek třídy *Error*) a přidány do seznamu chyb. Rozdíl mezi *JstError* a *UnmatchedTagWarning* je v tom, že *JstError* slouží k reprezentaci chyb získaných z generovaného syntaktického analyzátoru. Tyto chyby zpravidla obsahují také určité doplňkové informace o problému který nastal a problémy jsou označeny jako závažné. Naproti tomu *UnmatchedTagWarning* neobsahuje žádné doplňující informace, pouze označuje potenciální problémové místo varováním.

V jednom z předchozích odstavců byla zmíněna třída *JstPositionManager*, zbývá ale vysvětlit k čemu tato třída slouží. Jedná se o implementaci rozhraní *PositionManager* a úkolem tohoto objektu je mapování uzlů stromu, který je výsledkem analýzy, na odpovídající pozice v dokumentu. Mapování provádí implementace metody *getOffsetRange*. Metoda vrací objekt typu *OffsetRange*, který reprezentuje příslušnou část dokumentu. Objekt uchovává dva atributy – začátek (znak na této pozici je součástí oblasti) a konec (znak na této pozici je první znak, který už není zahrnut do oblasti). Metoda přijímá dva parametry. Prvním z nich je výsledek analýzy a druhým je objekt typu *ElementHandle* – *JstElementHandle*. Jak už název objektu napovídá, jedná se o jakýsi úchyt, který používá GSF k práci s částmi dokumentu. Modul GSF nedefinuje žádné rozhraní, které by popisovalo elementy jazyka, ze kterých by se měly dokumenty skládat. Toto záleží čistě na implementaci analyzátoru a objektu *ParserResult*. Zároveň ale GSF potřebuje umět s těmito elementy nějakým způsobem pracovat a znát o nich určité informace, aby mohlo poskytovat některé funkce (navigátor, sémantické zvýrazňování a podobně). Proto modul GSF definuje rozhraní *ElementHandle*, jehož implementace zapouzdřují uzly abstraktního syntaktického stromu a umožňují tak GSF s těmito uzly pracovat. *JstElementHandle* zapouzdřuje objekt *AstNode* a poskytuje pouze přístup k atributům tohoto objektu pomocí rozhraní *ElementHandle*.

Registraci implementace parseru je možné provést oběma způsoby – jak implementováním metody v konfigurační třídě *JstLanguage*, tak pomocí souboru *layer.xml*, jak bylo popsána a ukázáno v kapitole 5.3.2.

5.3.6 Balík structure

Balík *structure* obsahuje pouze dvě třídy – implementace rozhraní *StructureScanner* a *StructureItem* – *JstStructureScanner* a *JstStructureItem*. Jak název napovídá, jedná se o analyzátor struktury dokumentu, respektive o jednotku (položku) této struktury. Na základě struktury dokumentu nabízí vývojové prostředí dvě funkce:

- **skládání kódu** – umožňuje některé strukturální bloky schovávat a zobrazit jen jejich zkratku. Části kódu, které je možné složit zjišťuje metoda *folds* a vrací je jako mapu, kde klíč je druh složitelného bloku, a hodnota je seznam *OffsetRange* částí kódu, které odpovídají tomuto druhu složitelného bloku. Existuje několik druhů složitelných bloků – komentáře, bloky kódu, importy a podobně. V konfiguraci aplikace je potom možné nastavit, které druhy budou implicitně rozložené a které budou implicitně složené.
- **navigátor** – zvláštní okno zobrazující dokument jako seznam strukturálních jednotek, čímž je uživateli usnadněna orientace v širším kontextu dokumentu. Uživatel může pomocí položek navigátoru jednoduše přeskakovat do různých částí dokumentu. Seznam strukturálních jednotek vrací metoda *scan*.

Strukturální analyzátor dostane na vstup výsledek syntaktické analýzy, jehož součástí je derivační nebo abstraktní syntaktický strom. Následně je analyzována struktura tohoto stromu a je podle něj vytvořen seznam strukturálních jednotek. Každá strukturální jednotka může obsahovat další vnořené jednotky. Strukturální jednotky jsou založeny na objektech *ElementHandle* (viz minulá kapitola) a navíc obsahují několik nových informací:

- **ikonu** – bude použita pro zobrazení v navigátoru.
- **řadící řetězec** – strukturální jednotky jsou podle tohoto řetězce řazeny v navigátoru. Zpravidla je tento řetězec shodný s názvem jednotky, ale v případě potřeby je možné před název jednotky například přidat určitou konstantu a jednoduše tak změnit řazení, například podle kategorií, které tato konstanta reprezentuje. Pokud chceme řadit položky ve stejném pořadí jako se objevují v dokumentu, stačí do tohoto řetězce nastavit pozici jednotky v dokumentu. Alternativně je také možno poskytnout strukturálnímu skeneru jinou implementaci filtru, která bude provádět řazení podle potřeby.
- **formátovaný popis** – popis jednotky, který bude zobrazen v navigátoru. K jeho vytvoření se využívá objektu *HtmlFormatter*, který nabízí metody usnadňující popis metod, parametrů atd. jednotným stylem.

Třída *JstStructureScanner* implementuje rozhraní *StructureScanner*. Implementace metody *scan* pouze zapouzdřuje objekty *JstElementHandle* do objektů *JstStructureItem*, které implementují rozhraní *StructureItem*. Zde se nabízí otázka jestli by nebylo vhodnější aby třída *JstElementHandle* rovnou také implementovala rozhraní *StructureItem* a odpadla by nutnost další adaptace. V současném stavu aplikace by se toto mohlo jevit jako vhodné řešení, protože objekty *JstElementHandle* jsou využívány pouze pro podporu funkcí navigátoru. V dalším vývoji aplikace ale pravděpodobně budou přidány další funkce využívající objekt *JstElementHandle* také k jiným účelům, jako je například doplňování kódu, kde bude tento objekt opět použit jako základ jiného objektu s jiným rozhraním. Proto je s ohledem na budoucí vývoj vhodnější *JstElementHandle* a *JstStructureItem* oddělit do dvou samostatných objektů.

Implementace metody *fold* prochází uzly abstraktního syntaktického stromu (pomocí návštěvníka) a pokud se jedná o uzel reprezentující makro, direktivu, výraz, nebo blok JavaScriptu a zároveň zabírá kus kódu reprezentovaný tímto uzlem více řádků, pak je tato oblast kódu přidána do seznamu složitelných bloků.

Registraci strukturálního analyzátoru je stejně jako registraci parseru možno provést oběma způsoby – jak implementováním metody v konfigurační třídě *JstLanguage*, tak pomocí souboru *layer.xml*. Protože používáme vlastní implementaci *EditorKit*u, musíme navíc ručně registrovat služby GSF, které chceme využít, v souboru *layer.xml*. Konkrétně se jedná o zobrazení a obsluhu skládacích sekcí. Relevantní část souboru *layer.xml* je vidět na následující ukázce:

```
<folder name="GsfPlugins">
   <folder name="text">
      <folder name="x-ool-jst-template">
...
         <file name="structure.instance">
            <attr name="instanceOf" stringvalue="org.netbeans.modules.gsf.api.StructureScanner"/>
            <attr name="instanceClass"
stringvalue="org.netbeans.modules.jstlexer.structure.JstStructureScanner"/>
         </file>
      </folder>
   </folder>
</folder>
<folder name="Editors">
   <folder name="text">
      <folder name="x-ool-jst-template">
         <folder name="FoldManager">
            <file name="org-netbeans-modules-gsfret-editor-fold-GsfFoldManagerFactory.instance"/>
         </folder>
         <folder name="SideBar">
            <file name="org-netbeans-modules-editor-gsfret-
GsfCodeFoldingSideBarFactory.instance">
               <attr name="position" intvalue="1200"/>
            </file>
         </folder>
      </folder>
   </folder>
</folder>
```

5.3.7 Balík formatting

Balík *formatting* obsahuje několik tříd, které mají na starosti automatické formátování dokumentu. Automatické formátování provádí zejména správné odsazování řádků podle určitých konvencí a přichází ke slovu v několika situacích. Jednak je možné nechat si zformátovat dokument (nebo jen označenou část dokumentu) na požádání zavoláním akce *format* z menu aplikace. Vývojové prostředí však také využívá automatického formátování aniž byste o to explicitně požádali. Konkrétně se jedná o situaci, kdy v editoru zmáčknete klávesu enter a kurzor na novém řádku se vám automaticky odsadí na správnou pozici.

Situace kolem formátování dokumentů, ve kterých se prokládá několik různých jazyků, je poměrně složitá, protože ke každému jazyku existují jiné formátovací konvence. Proto by ke každému jazyku měla existovat jiná implementace rozhraní *Formatter*, která by zajišťovala úpravu části kódu v daném jazyce podle příslušných konvencí. Podle hierarchie jazyků by potom tyto implementace měly rozděleno pole své působnosti. Každý *Formatter* by tak formátoval jen jemu příslušející část dokumentu. *Formatter*y příslušející jazykům, které jsou na vyšší úrovni hierarchie jazyků, by měly dostat již zformátované části dokumentu pro jazyky na nižších úrovních hierarchie, a toto formátování pouze upravit například zvětšením odsazení celého bloku vloženého jazyka. Takto by se postupovalo až k *Formatter*u jazyka, který je na první úrovni hierarchie jazyků, a tím by bylo formátování dokumentu dokončeno.

Do jazyka šablon mohou být vloženy kusy jazyka JavaScript nebo HTML. Pro tyto jazyky už implementace *Formatter*u existují. Aby bylo zajištěno formátování celého dokumentu, měly by *Formatter* pro jazyk šablon (*JstFormatter*), pro JavaScript, a pro HTML spolupracovat podle scénáře popsaného v předchozím odstavci. Bohužel tomu tak není. Moduly pro podporu různých jazyků byly vytvořeny dříve než existovaly moduly GSF a Lexer. Prokládaní jazyků bylo řešeno různě a nejinak tomu bylo u formátování. Se zavedením modulu Lexer se začaly podpory některých jazyků přepisovat do systému, který zaváděl modul Lexer, a kde bylo prokládání jazyků řešeno jednotně. S příchodem modulu GSF se situace opakovala a některé funkce modulů podporujících různé jazyky se začaly přepisovat do systému, který zavádí modul GSF. Podpora jazyka HTML je zatím bohužel tak na půl cesty mezi původní implementací a GSF, díky čemuž někdy vznikají problémy pokud je jedním z vkládaných jazyků právě HTML. Jedním z těchto problémů je například to, že HTML je formátováno vždy tak, jako by bylo na nejvyšší úrovni hierarchie jazyků, tzn. jako poslední. Toto chování je v našem případě nevyhovující. Na nejvyšší úrovni je jazyk šablon, proto je potřeba aby nejprve byly zformátovány všechny vložené jazyky (včetně HTML) a teprve potom tyto zformátované bloky zpracoval *JstFormatter*. Tohoto chování bylo dosaženo následujícím trikem.

Třída *JstFormatter* implementuje rozhraní *Formatter* modulu GSF. To znamená, že tato třída poskytuje metody, jejichž voláním je možné zformátovat dokument jazyka šablon. Jak tyto metody pracují bude popsáno později. Důležité ale je, že tato implementace není registrována jako poskytovatel služby, a proto o ní modul GSF neví. Formátování dokumentu tak není v režii modulu GSF, protože tento modul jednoduše neví jak na to. K formátování dokumentu je použit způsob, který se používal před zavedením modulu GSF a to je implementace rozhraní NetBeans Indentation SPI[14]. Zde se o formátování stará formátovací úloha – implementace rozhraní *ReformatTask*. Do systému je registrována tovární třída – implementace rozhraní *ReformatTask.Factory* – vytvářející instance konkrétní formátovací úlohy. Registrace se provádí pomocí souboru *layer.xml*, kde je tovární třída registrována k příslušnému MIME typu. Trik spočívá v tom, že jsou registrovány dvě formátovací úlohy:

- **JstReformatTask** – je registrována pro MIME typ jazyka šablon *text/x-ool-jst-template*. Tato formátovací úloha pouze vymaže odsazení všech řádků, které mají být zformátovány a tím připraví půdu pro formátování HTML.
- **JstHtmlIndentTask** – je registrována pro MIME typ HTML vloženého do jazyka šablon *text/x-ool-jst-template/text/html*. Tato formátovací úloha nejprve provede formátování vybrané části dokumentu pomocí objektu *HtmlLexerFormatter*, který je součástí modulu pro podporu jazyka HTML a má na starosti formátování. Poté je delegována činnost na *JstFormatter*. Díky tomu je jazyk šablon formátován jako poslední až už jsou všechny vnořené jazyky zformátovány.

Díky tomuto triku funguje formátování dle očekávání. Navíc protože toto řešení používá třídu *JstFormatter* implementující rozhraní *Formatter* modulu GSF, bude následný přechod na systém modulu GSF velice jednoduchý a měl by spočívat pouze ve změnách v konfiguračním souboru *layer.xml*.

Nyní zbývá jen popsat jak vypadá a pracuje třída *JstFormatter*. Rozhraní *Formatter* definuje následující metody:

- **reformat** – provádí formátování vybrané části dokumentu. V této implementaci je jediným úkolem formátování správné odsazení, proto je řízení delegováno na metodu *reindent*. Obecně může formátování provádět větší úpravy v dokumentu jako je například přidávání mezer na vhodná místa (kolem operátorů a podobně).
- **reindent** – provádí odsazení vybraných řádků dokumentu na správnou pozici. Metoda je také volána při stisku klávesy enter a přechodu na nový řádek.
- **needsParserResult** – udává jestli metoda *reformat* potřebuje ke své činnosti výsledek syntaktické analýzy. Pokud nepotřebuje (jako v našem případě), nemusí být potřeba syntaktickou analýzu provádět, což může pozitivně ovlivnit rychlost aplikace.
- **indentSize** – velikost odsazení jako celé číslo vyjadřující počet mezer. Hodnota je získána z konfigurace aplikace za pomoci pomocných tříd *CodeStyle* a *FormattingOptions*.
- **hangingIndentSize** – velikost odsazení řádku, který je pokračováním řádku předchozího. Hodnota je získána z konfigurace aplikace za pomoci pomocných tříd *CodeStyle* a *FormattingOptions*.

Metoda *reindent* pracuje následujícím způsobem. Nejprve je zjištěno výchozí odsazení. V případě, že formátujeme jenom část dokumentu, bereme jako výchozí odsazení posledního řádku, který je před formátovanou částí. Celý vybraný blok bude odsazován relativně k tomuto řádku bez ohledu na to, jestli tento řádek sám je odsazen správně nebo ne. Následně je vytvořen seznam doporučených odsazení pro jednotlivé řádky výběru, a nakonec je podle tohoto seznamu provedena modifikace dokumentu. Modifikace probíhá jako atomická operace, tzn. buď proběhne celá a úspěšně, nebo neproběhne vůbec. Nemůže tedy nastat situace, kdy uživatel kvůli chybě při práci s dokumentem, dostane dokument „v horším stavu“ než před formátováním.

Správná hodnota odsazení je určována podle obsahu řádku. Pro každý řádek se počítá tzv. váha, která určuje, jestli následující řádek bude odsazen více, méně nebo stejně, případně o kolik jednotek *indentSize* se bude odsazení dalšího řádku lišit. Při počítání váhy jsou analyzovány nejen všechny tokeny nacházející se na řádku, ale také obsah těchto tokenů, podle následujícího postupu:

- **zvýšení váhy:**
  - pokud token znamená začátek bloku JavaScriptu, výrazu nebo direktivy.
  - pokud tokeny značí začátek makra.
  - pokud se v tokenu objevuje větší počet levých závorek než pravých, nebo pokud se objevuje levá závorka na konci tokenu.
- **snížení váhy:**
  - pokud token znamená konec bloku JavaScriptu, výrazu nebo direktivy.

- pokud tokeny značí konec makra.
- pokud se v tokenu objevuje více pravých závorek než levých, nebo pokud token začíná pravou závorkou.

Každý řádek je tedy odsazen podle váhy, která byla spočítána analýzou řádku předchozího.

Speciální zacházení vyžadují víceřádkové tokeny. Pokud se při analýze narazí na token, který zabírá více řádků, počítá se váha jen pro první řádek tohoto tokenu. Tento první řádek je odsazen jako každý jiný řádek podle váhy spočítané z předchozího řádku. Pro další řádky tokenu váha počítána není, protože na těchto řádcích se vyskytuje pořád pouze jeden a tentýž token, který už byl analyzován. Protože se na řádku nevyskytuje žádný jiný token, není možné, aby se váha změnila. Proto jsou tyto řádky odsazeny stejně jako první řádek, na kterém se víceřádkový token vyskytuje. Výjimka nastává u posledního řádku víceřádkového tokenu. Tento řádek je konečně odsazen podle váhy spočítané u prvního řádku. A protože se na tomto řádku již vyskytují i tokeny, které doposud nebyly analyzovány, je opět spočítána váha pro následující řádek a vše se vrací do starých kolejí.

Díky tomuto zacházení s víceřádkovými tokeny jsou korektně odsazeny vždy celé bloky vložených jazyků. Zároveň díky tomu, že při počítání váhy je analyzován i obsah jednotlivých tokenů, jsou bloky vložených jazyků odsazeny na správnou úroveň.

6 Závěr

6.1 Shrnutí

Cílem práce bylo vytvořit vývojové prostředí, které by usnadňovalo vývoj šablon pro webový šablonovací systém společnosti oXy Online. Vývojové prostředí mělo poskytovat hlavně rozšířené funkce editoru, jako je zvýrazňování syntaxe, upozorňování na chyby a automatické formátování, a mělo být navrženo tak, aby bylo v budoucnu rozšiřitelné o další funkce. Po prostudování dostupných platforem usnadňujících vývoj desktopových aplikací byla jako základ zvolena platforma NetBeans a vývojové prostředí bylo realizováno jako plugin do této platformy. Díky modularitě platformy je zajištěna možnost snadného rozšíření o další funkce, a díky použití množství již existujících modulů byla podstatně usnadněna práce při vývoji.

Pro podporu rozšířených funkcí editoru je potřeba, aby editor uměl provádět jazykovou analýzu zdrojových souborů. K tomuto účelu jsou využity analyzátory generované nástrojem ANTLR podle bezkontextové gramatiky jazyka. Dodaná gramatika, která byla původně využita k vytvoření překladače šablonovacího systému, byla upravena tak, aby popisovala jazyk podrobněji, a aby elementy, pomocí kterých jazyk popisuje, mohly být použity pro podporu funkcí editoru. Součástí práce je univerzální modul platformy NetBeans – *AntlrGsfBridge*. Tento modul abstrahuje adaptaci lexikálních a syntaktických analyzátorů generovaných nástrojem ANTLR tak, aby vyhovovaly požadavkům, které jsou moduly NetBeans IDE kladeny na analyzátory pro podporu rozšířených funkcí editoru. Díky tomuto modulu je možné při případné změně gramatiky jednoduše vygenerovat nové analyzátory a ty potom vložit do aplikace pouhou změnou konfigurace. Modul je také možné použít pro vytvoření podpory libovolného jiného jazyka založeného na ANTLR gramatice.

Základem vývojového prostředí je modul *JstEditor*. Tento modul implementuje a využívá rozhraní různých modulů NetBeans IDE, a tak vytváří editor podporující soubory šablon. Editor podporuje lexikální a syntaktické zvýrazňování zdrojového kódu, zvýrazňování chyb ve zdrojovém kódu včetně informací o chybách, navigátor, skládání kódu a automatické formátování. Navíc jsou podporovány také některé funkce zděděné po editoru HTML, jako je doplňování párových značek, automatické doplňování kódu a kontextová nápověda, pro vložené kusy HTML kódu.

6.2 Možnosti dalšího rozšíření

Díky modularitě je zajištěna možnost snadného rozšiřování a škálování aplikace přidáváním dalších modulů. Realizovaný plugin v současné době poskytuje jen několik málo funkcí, které od vývojového prostředí očekáváme, a proto je jeho další rozšíření nasnadě. První na řadě by určitě mělo být automatické doplňování kódu a kontextová nápověda, protože tyto funkce jsou mezi uživateli jedny z nejoblíbenějších, pro značnou dávku pohodlí, které uživatelům poskytují. Mezi další rozšíření editoru by měly patřit funkce jako je automatické doplňování a zvýrazňování párových závorek, uvozovek a značek maker, automatické nabízení řešení některých chyb v kódu, sémantická analýza a různé refaktorovací funkce jako například rychlé přejmenování maker, nebo zabalení části kódu do makra. Kromě rozšířených funkcí editoru, ale může vývojové prostředí nabízet spoustu dalších nástrojů jako je například náhled v prohlížeči nebo debugger. Ke všem zmíněným funkcím a nástrojům existují moduly NetBeans IDE, kterých je možné při dalším vývoji s výhodou využít.

6.2 Možnosti dalšího rozšíření

Další možnosti rozšíření se nabízejí také u modulu *AntlrGsfBridge*. Modul byl původně vyvinut k tomu, aby abstrahoval adaptaci analyzátorů, a aby bylo možné laborovat s gramatikou jazyka bez nutnosti upravovat zbytek aplikace (kromě generovaných implementací analyzátorů). Zároveň by však tento modul mohl najít využití i u širší komunity lidí při vytváření podpory jiných jazyků. Přestože byl modul navržen pro univerzální použití, mohou se samozřejmě objevit problémy, které se při dosavadním použití neprojevily. V současné době modul podporuje adaptaci lexikálních analyzátorů a syntaktických analyzátorů vytvářejících derivační stromy. ANTLR však také umožňuje, aby analyzátory vytvářely abstraktní syntaktické stromy, nebo dokonce vytvářet analyzátory těchto stromů. Tyto analyzátory by mohly najít uplatnění například při sémantické analýze. Adaptace těchto druhů analyzátorů zatím není v modulu žádným způsobem řešena a zůstává prostorem pro budoucí rozšíření.

Seznam použité literatury

[1] Parr, Terence, ANTLR v3 documentation, 2007, http://www.antlr.org/wiki/display/ANTLR3/ANTLR+v3+documentation
[2] Černá, Ivana; Křetínský, Mojmír; Kučera, Antonín, Automaty a formální jazyky I, , 159s, 2002,
[3] Bodeček, Miroslav, Zpracování dokumentů specifikovaných jazykem Texy!, 2008
[4] Kolektiv autorů, Otevřená encyklopedie Wikipedia, 2002, http://en.wikipedia.org/wiki/Main_Page
[5] Boudreau, Tim; Tulach, Jaroslav; Wielenga, Gertjan, Rich client programming : plugging into the NetBeans platform, Prentice Hall, 614s, 2007, 0-13-235480-2
[6] Wielenga, Geertjan, Top 10 NetBeans APIs (slajdy), 2008, http://platform.netbeans.org/tutorials/nbm-10-top-apis.html
[7] Epple, Toni, How Do NetBeans Extension Points Work?, 2008, http://netbeans.dzone.com/news/netbeans-extension-points
[8] Jakubička, Martin, Aplikace pro správu fotografií naplatformě NetBeans, 2007
[9] Pavlíček, Luboš; Dvořák, Miloš, Návrhové vzory (design patterns), 2005, http://objekty.vse.cz/Objekty/Vzory
[10] Lasater, Christopher G., Design Patterns, Wordware Publishing, 296s, 2007, 1-59822-031-4
[11] Kolektiv autorů, NetBeans Platform Learning Trail, 2008, http://platform.netbeans.org/tutorials/
[12] Wielenga, Geertjan, Lexer Migration Guide, 2007, http://platform.netbeans.org/tutorials/nbm-mfsyntax_migrate_lexer.html
[13] Metelka, Miloslav; Stejskal Vítězslav, http://bits.netbeans.org/dev/javadoc/org-netbeans-modules-editor-lib2/org/netbeans/spi/editor/highlighting/package-summary.html, 2006,
[14] Kolektiv autorů, Editor Indentation SPI, 2007, http://bits.netbeans.org/dev/javadoc/org-netbeans-modules-editor-indent/org/netbeans/modules/editor/indent/spi/package-summary.html