České vysoké učení technické v Praze
Fakulta elektrotechnická

katedra počítačové grafiky a interakce

# ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student: **Luboš Černý**

Studijní program: Softwarové technologie a management
Obor: Web a multimedia

Název tématu: **Validátor jBPM 3 Workflow jako plugin pro Eclipse IDE**

Pokyny pro vypracování:

Oblíbenou platformou pro implementaci business procesů v enterprise aplikacích jsou jBPM Workflows. Lze je vytvářet např. zapsáním struktury orientovaného grafu jako XML souboru. Na uzly a hrany grafu se pak vážou programové akce v podobě zdrojového kódu v jazyce Java (BSh).
Při editaci XML souboru, jehož součástí je zdrojový kód, přijde ale vývojář o všechny výhody, které IDE poskytuje - analýzu syntaxe, obarvování kódu, atd.
Implementujte parser pro tyto XML soubory, pro definici JBoss jBPM v3. Parser bude Javovskému kódu ve workflow poskytovat všechny základní kontroly a podpůrné funkce, které IDE běžně poskytuje zdrojovým kódům v Javě. Dále bude validovat logickou strukturu grafu - izolované uzly, hrany bez konců, atd.

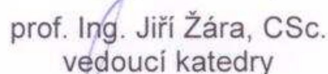
Parser realizujte jako plugin do Eclipse IDE a otestujte ho na sadě workflow pro CzechIdM, které dodá vedoucí práce. Testovací data zahrnují desítky workflow (až desítky uzlů v grafu, stovky řádků kódu na workflow).

Seznam odborné literatury:

JBoss jBPM - Workflow in Java: jBPM jPDL User Guide. [online]. [cit. 2013-08-22]. Dostupné z: http://docs.jboss.com/jbpm/v3.2/userguide/html_single/
CLAYBERG, Eric a Dan RUBEL. Eclipse plug-ins. 3rd ed. Upper Saddle River: Addison-Wesley, c2009, xlv, 878 s. The eclipse series. ISBN 978-0-321-55346-1.
GREGOR, Jan, Jaromír MLEJNEK, Jana MOUDRÁ a Vojtěch MATOCHA. BCV SOLUTIONS S.R.O. CzechIdM 1.1: Programátorská dokumentace. 2012.

Vedoucí: Ing. Jaromír Mlejnek

Platnost zadání: do konce zimního semestru 2014/2015



prof. Ing. Jiří Žára, CSc.
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 18. 9. 2013

České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra počítačové grafiky a interakce

Bakalářská práce

# Validátor jBPM 3 Workflow jako plugin pro Eclipse IDE

*Luboš Černý*

Vedoucí práce: Ing. Jaromír Mlejnek

Studijní program: Softwarové technologie a management, Bakalářský

Obor: Web a multimedia

Květen 2014

## Poděkování

Rád bych poděkoval všem, kteří mě v této práci podporovali. Děkuji Petře Doležalové za jazykové korekce práce, děkuji celé své rodině za podporu a děkuji panu Jaromíru Mlejnkovi, svému vedoucímu bakalářské práce, za podporu a konstruktivní připomínky v průběhu tvorby této práce. V neposlední řadě bych chtěl také poděkovat účastníkům závěrečného testování za čas, který mi věnovali.

## Prohlášení

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

V Praze dne 23. května 2014

..............................................

Luboš Černý

// Abstrakt

Práce se věnuje vývoji pluginu pro vývojové prostředí Eclipse IDE, jenž vzniká na základě požadavků vývojářů business procesů JBoss jBPM v3 Workflow na efektivnější vývojové prostředí. Začátek tohoto textu je věnován specifikaci překážek v průběhu vývoje Workflow. Na základě těchto poznatků jsou určeny požadavky na plugin a je proveden návrh struktury aplikace. Závěr práce je pak věnován výsledné realizaci prototypu a jeho testování.

## Klíčová slova

Eclipse IDE, plugin, jBPM 3 Workflow, validace.

# Abstract

The thesis deals with plugin development for Eclipse IDE, based on bussiness process developers' demand on more effective development environment dedicated to JBoss jBPM v3 Workflow. Workflow development obstacles are described in the first part. Based on information gathered, requirements and plugin structure are defined. The final part is dedicated to realization of prototype design and its testing.

## Keywords

Eclipse IDE, plugin, jBPM 3 Workflow, validation.

# Obsah

1. Úvod 1
1.1. Úvod . . . . . 1
1.2. Motivace . . . . . 1
1.3. Kapitoly . . . . . 1

2. Popis problému a specifikace cíle 3
2.1. Cíle práce . . . . . 3
2.2. Přínos pluginu . . . . . 3

3. Rozbor prvků Workflow 5
3.1. JBoss jBPM 3 a Workflow . . . . . 5
3.2. BeanShell . . . . . 7

4. Návrh pluginu 8
4.1. Obarvení kódu . . . . . 9
4.2. XML parser . . . . . 9
4.3. Validátor kódu . . . . . 10

5. Realizace 13
5.1. Vytvoření editoru . . . . . 13
5.2. Kontroler validace . . . . . 19
5.3. Parser . . . . . 21
5.4. Validátory obecně . . . . . 25
5.5. GraphValidator . . . . . 27
5.6. BeanShellValidator . . . . . 30
5.7. VariableValidator . . . . . 33

6. Testování 42
6.1. Test uživatelského rozhraní . . . . . 42
6.2. Test kompatibility . . . . . 43
6.3. Test výkonu . . . . . 43

7. Závěr 47
7.1. Závěr . . . . . 47
7.2. Další vývoj projektu . . . . . 47

Přílohy

A. Obsah přiloženého CD 49

B. Instalace 50
B.1. Update site . . . . . 50

B.2. Dropins . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50
B.3. Známé problémy při instalaci . . . . . . . . . . . . . . . . . . . . . . . . 51

**C. Použití** **52**

**D. Screener** **54**

**E. Předtestový dotazník** **56**

**F. Scénář uživatelského testování** **57**

**G. Potestový dotazník** **60**

**H. Výsledky** **61**

**I. Obrázky** **64**

**Literatura** **67**

**Seznam obrázků** **69**

**Seznam výpisů kódu** **70**

## Použité zkratky a označení

| | |
|---|---|
| API | Application programing interface: rozhraní knihovny, popisující kolekci metod, tříd, funkcí poskytovaných danou knihovnou, které lze v rámci vývoje aplikace využít. |
| BP | Business proces: posloupnost kroků potřebných pro dosažení daného cíle. |
| BPMN | Business Process Model and Notation: specifikuje způsob grafického popisu business procesu. Poskytuje převážně analytický pohled. |
| DOM | Document Object Model: pamětová reprezentace XML dokumentu, nebo také jeden z rošířených XML parserů. |
| DTD | Document Type Definition: jazyk pro popis struktury XML dokumentu. |
| GUI | Graphic User Interface: zkratka popisující grafické rozhraní aplikace |
| IDE | Integrated Development Environment: vývojové prostředí určené pro usnadnění vývoje v daném programovacím jazyce. |
| jBPM | Business Process Management: framework pro tvorbu a spouštění BP (BPMN). |
| JDE | Java Development Environment: prostředí určené pro vývoj v jazyce Java. |
| jPDL | Process Definition Language: proprietární značkovací jazyk založený na XML, vyvíjený JBoss týmem, jenž je součástí společnosti Red Hat. |
| SAX | Simple API for XML: velmi rychlý a úsporný XML parser. |
| XML | eXtensible Markup Language: značkovací jazyk určený pro jednoduchý přenos dat přes internet, vytvořený s ohledem na jednoduché vytvoření a zpracování dokumentu[14]. |
| XML Schema | Jazyk určený pro popis struktury XML dokumentu, jiř oproti DTD poskytuje bohatší jazyk pro popis obsahu elementů a zavádí jmenné prostory. |

# Kapitola 1.

# Úvod

## 1.1. Úvod

Práce se zabývá návrhem a implementací pluginu do IDE Eclipse pro vývoj JBoss jBPM 3 Workflow, dále jen Workflow. Uvedený plugin doplní chybějící podporu IDE Eclipse pro vývoj Workflow, který se používá pro popis modelového business procesu, dále jen BP.

## 1.2. Motivace

V současnosti existuje velké množství různých vývojových prostředí, dále jen IDE, usnadňujících programátorovi vývoj aplikací. Jako příklad lze uvést NetBeans IDE vyvíjené firmou Oracle, Eclipse vyvíjené Eclipse Foundation nebo Microsoft Visual Studio od firmy Microsoft. IDE pak účinně eliminuje syntaktické chyby vznikající většinou z nepozornosti programátora, jako jsou překlepy v názvech proměnných či metod nebo také chybějící či přebývající závorky a středníky, což ve výsledku zrychluje vývoj celé aplikace.

Při vývoji Workflow však programátor všechny standardní výhody IDE, jako jsou obarvení kódu nebo kontrola správného uzávorkování, nemá k dispozici. Žádný z aktuálně rozšířených IDE totiž neposkytuje implicitní podporu pro vývoj dokumentů Workflow. Programový kód umístěný uvnitř struktury dokumentu je považován za obyčejný text. Syntaktické chyby se tak projeví až v průběhu sestavení celého projektu, které v mnoha případech trvá i několik minut, a následný proces odstraňování chyb se ve výsledku stává časově velmi náročný a zdlouhavý.

## 1.3. Kapitoly

Práce se skládá ze dvou částí, teoretické a implementační. Celkem je rozdělena do sedmi kapitol, z nichž pět je teoretického charakteru a dvě se věnují praktické stránce projektu. Kapitoly obsahují následující témata:

**Kapitola** 2 se věnuje problému tvorby Workflow a identifikuje požadavky na funkčnosti pluginu. Nakonec specifikuje cíle, kterých má tento projekt dosáhnout.

**Kapitola** 3 se hlouběji věnuje problematice a součástem dokumentu Workflow, jeho tvorbě a skriptovacímu jazyku BeanShell.

**Kapitola** 4 se zabývá teoretickým návrhem aplikace a specifikuje požadavky na jednotlivé komponenty, ze kterých se projekt skládá.

**Kapitola** 5 popisuje implementaci jednotlivých komponent pluginu.

**Kapitola** 6 se věnuje testování použitelnosti, kompatibility a výkonových nároků projektu.

**Kapitola** 7 poskytuje shrnutí celého projektu a dosažených výsledků. Děle rozvádí diskuzi o dalším možném rozšíření projektu.

# Kapitola 2.

# Popis problému a specifikace cíle

V současné době existuje několik aplikací určených pro design a modelování business procesů popsaných ve Workflow, například JBoss jBPM GDP [8], jenž je součástí balíčku IDE Eclipse pluginů JBoss Tools, nebo také webová aplikace jBPM Designer 2.4 [9][10]. Naneštěstí žádný z uvedených nástrojů neposkytuje podpůrné funkce pro psaní programového kódu, který je nedílnou a důležitou součástí Workflow. Ani na poli moderních IDE včetně jejich pluginů dnes neexistuje nástroj umožňující efektivní vývoj programové logiky Workflow. Editace se tak ve výsledku stává operací s vysokou náročností na čas potřebný k provedení. Kombinace těchto faktorů tak dala vzniknout tématu této práce.

## 2.1. Cíle práce

Cílem práce je navrhnout a implementovat komplexní nástroj pro tvorbu a editaci Workflow v prostředí Eclipse IDE. Toto prostředí bylo zvoleno z důvodu množství dalších podpůrných pluginů a nástrojů pro vývoj enterprise aplikací. Dalším důvodem je rozsáhlá podpora ze strany JBoss komunity, například projekt JBoss Tools.

Požadavkem na výsledný plugin je zachování schopnosti zvýrazňování syntaxe XML, obohacené o zvýraznění syntaxe BeanShellu[1], čímž se dosáhne vyšší přehlednosti kódu. Plugin také poskytne základní funkce pokročilých IDE jako detekci chybějících importů, detekci nedeklarovaných proměnných a kontrolu správného pořadí závorek v kódu. Dále nabídne kontrolu logické struktury v rámci definice Workflow, konkrétně vyhledání izolovaných uzlů a neukončených hran.

## 2.2. Přínos pluginu

V závislosti na neexistenci ekvivalentu vyvíjeného pluginu lze očekávat výrazné zjednodušení a zrychlení vývoje Workflow. V důsledku eliminace syntaktických chyb v průběhu implementace business procesů a vylepšení přehlednosti vyvíjeného kódu dojde ke zvýšení výkonnosti programátora. Dalším pozitivem zvýšení přehlednosti kódu bude snížení psychické zátěže programátora, což opět pozitivně přispěje k jeho celkové

[1] BeanShell - interpretr skriptů, blíže v kapitole 3.

produktivitě. V závislosti na neexistenci ekvivalentu vyvíjeného pluginu lze očekávat výrazné zjednodušení a zrychlení vývoje Workflow. V důsledku eliminace syntaktických chyb v průběhu implementace business procesů a vylepšení přehlednosti vyvíjeného kódu dojde ke zvýšení výkonnosti programátora. Dalším pozitivem zvýšení přehlednosti kódu bude snížení psychické zátěže programátora, což opět pozitivně přispěje k jeho celkové produktivitě. Z firemního hlediska tento projekt představuje úsporu a možnost efektivnější alokace finančních prostředků, která vede ke zvýšení počtu zpracovaných projektů, a tedy ke zvýšení konkurenceschopnosti.

# Kapitola 3.

# Rozbor prvků Workflow

## 3.1. JBoss jBPM 3 a Workflow

Knihovna JBoss jBPM je rozsáhlý framework napsaný v jazyce Java určený pro správu BP implementovaných jako Workflow umožňující jejich následné spouštění[5].

Dokumenty Workflow jsou definovány v jazyce jPDL, což je ve své podstatě graf procesu, skládající se z uzlů a přechody mezi nimi, zapsaný v dokumentu XML pomocí elementů. Jména elementů uvnitř XML dokumentu pak odpovídají názvům uzlů. Takto vytvořený dokument musí být pojmenován processdefinition.xml[1].

Dokument Workflow se skládá z uzlů, přechodů mezi uzly a z akcí. Kořenový element Workflow dokumentu, který musí být obsažen v každém správně strukturovaném XML dokumentu, je pojmenován `process-definition`. Kořen grafu pak v sobě musí obsahovat elementy `start-state` a `end-state`. Vyjmenované elementy dokumentu XML představují minimální množinu, ze které již lze vytvořit zcela platný Workflow dokument. Dalšími možnými typy elementů pak jsou[1]:

**task-node**
: Reprezentuje jednu nebo více úloh prováděných uživatelem a čeká na interakci. V grafické reprezentaci znázorněn uzlem.

**state**
: Reprezentuje uzel, při které se čeká na explicitní vyvolání akce. Tato akce může být vyvolána externím systémem, jinou komponentou nebo případně uživatelem.

**decision**
: Reprezentuje rozdělení toku Workflow na základě uvedených podmínek. V grafické reprezentaci znázorněn uzlem.

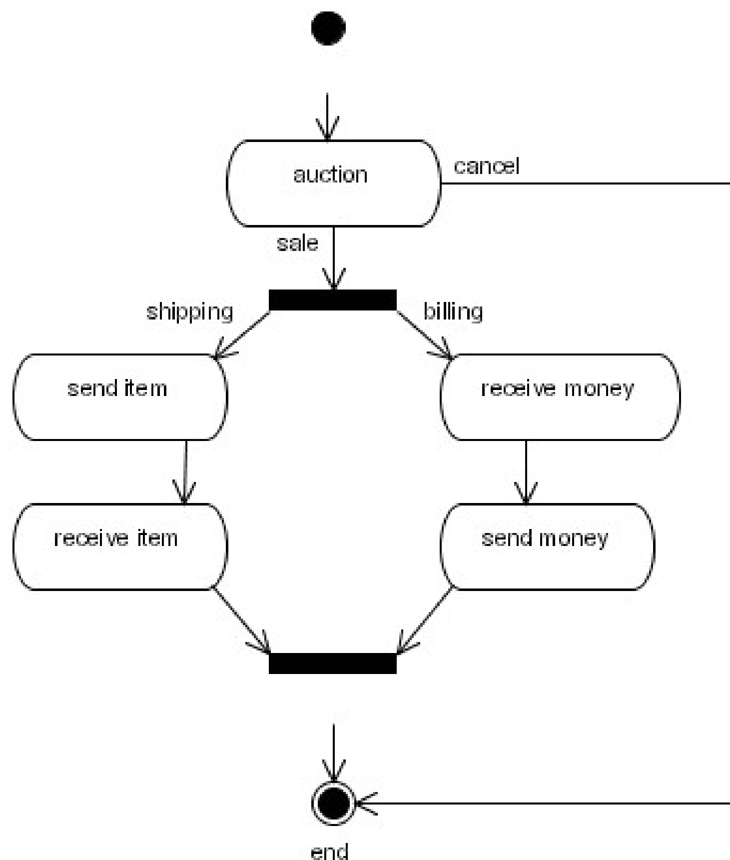
**fork** Uzel, jenž rozděluje tok zpracování Workflow na více souběžných úloh.

**join** Spojuje běh souběžných úloh, které vzešly ze stejného uzlu `fork`. Pokud není stanoveno jinak, zpracování Workflow čeká v tomto uzlu na dokončení běhu všech jeho aktivních vstupů.

**node**
: Uzel určený pro definici vlastního kódu v uzlu. Kód uvnitř je spuštěn při vstupu do uzlu v průběhu zpracování Workflow.

**transition**

Reprezentuje přechod mezi ostatními uzly. Samostatně nemůže existovat a musí být zapsán jako vnitřní element některého z výše uvedených uzlů, což určí výchozí uzel přechodu. Uzel povinně obsahuje atribut `to` určující cílový uzel. Počet uzlů `transition` uzavřených v jednom uzlu není nijak omezen.




**Obrázek 1.** Model BP aukce fiktivního aukčního domu[1]

Všechny uzly uvedené v předchozím seznamu, včetně uzlu `start-state`, musí obsahovat definici přechodu, v grafickém znázornění Workflow zobrazenou jako hranu mezi uzly. [1]

Následující výpis kódu 3.1 je příkladem, jak může Workflow vypadat. Zobrazuje strukturu grafu Workflow, jenž byl vytvořen na základě modelu BP z obrázku 1.

**Výpis kódu 3.1 Workflow BP znázorněného na obrázku 1.[1]**

```
<process-definition>
  <start-state>
    <transition to="auction" />
  </start-state>
  <state name="auction">
    <transition name="auction ends" to="salefork" />
    <transition name="cancel" to="end" />
```

```
    </state>
    <fork name="salefork">
      <transition name="shipping" to="send item" />
      <transition name="billing" to="receive money" />
    </fork>
    <state name="send item">
      <transition to="receive item" />
    </state>
    <state name="receive item">
      <transition to="salejoin" />
    </state>
    <state name="receive money">
      <transition to="send money" />
    </state>
    <state name="send money">
      <transition to="salejoin" />
    </state>
    <join name="salejoin">
      <transition to="end" />
    </join>
    <end-state name="end" />
  </process-definition>
```

## 3.2. BeanShell

Jedná se o malý, snadno použitelný interpretr skriptů napsaný v jazyce Java. Interpretr je schopný vyhodnotit standardní výrazy jazyka Java, navíc přidává funkčnosti běžných skriptovacích jazyků. Oproti běžným skriptovacím jazykům, které redukují datové typy na minimální množství, je možné využít silné typovosti jazyka Java. Ačkoli je ve skriptu BeanShell vše považováno za typ Object, lze dosáhnout typové kontroly explicitním přetypováním proměnné v těle skriptu[2, 13].

BeanShell byl vyvíjen Patem Niemeyerem, který se jeho první verzi rozhodl vydat v roce 1997 v návaznosti na přidání reflexe do verze 1.1 jazyka Java. Od té doby BeanShell získával pomalu na popularitě a stal se součástí mnoha vývojových prostředí jako například JDE textového editoru Emacs a také NetBeans IDE[2, 13].

V květnu roku 2007 vznikla odnož projektu pojmenovaná BeanShell2, která je aktivně vyvíjena v aktuální verzi 2.1.8. Důvodem pro tento krok byla již dlouhodobá stagnace původního projektu. [4]

# Kapitola 4.

# Návrh pluginu

Požadavky na plugin jsou rozděleny do tří částí, kterým se do hloubky věnují následující sekce.

První část se věnuje požadavku na obarvení textu, kdy je nutné specifikovat požadavky na volbu barev pro oba jazyky vyskytující se v dokumentu Workflow (XML a Java) a jakým způsobem od sebe odlišit literály typu String, které se u obou jazyků vyskytují.

Druhá část se věnuje problému parsování, což je proces analýzy informací, jehož účelem je získání syntaktické struktury analyzovaného objektu, v tomto případě XML dokumentu. Vhodný parser musí umožnit průchod stromem dokumentu pro potřeby jednotlivých validátorů. Uzly stromu musejí být schopny uchovat dodatečné informace pro validátory, jako například informaci o umístění v dokumentu.

V tuto chvíli není stanoven požadavek na schopnost parseru provést validaci struktury XML dokumentu oproti DTD, či XML Schema. Z uvedeného důvodu je při výběru parseru další podmínkou pouze schopnost validovat správnost struktury dokumentu v rámci standardu XML 1.0[14].

Poslední část je pak věnována samotné validaci Workflow dokumentu a jeho obsahu. Z důvodu, že je projekt stále v rané fázi a má potenciál se dále rozvíjet, lze očekávat, že požadavky na tuto část budou procházet změnami a budou se postupně rozšiřovat. Proto je nutné navrhnout strukturu validace tak, aby bylo co možná nejsnazší funkčnosti validátoru doplňovat a měnit. Zároveň je však třeba udržet co nejmenší závislost na paměťové reprezentaci dokumentu Workflow.

Plugin musí být schopen provést validaci logické struktury grafu Workflow. Dále je nutné validovat struktury přechodů mezi uzly v rámci business procesu, kontrolovat správnost syntaxe BeanShell kódů umístěných uvnitř uzlů skriptů a v neposlední řadě kontrolovat definované proměnné.

Ve výsledku je při návrhu nutné vzít v potaz náročnost jednotlivých kroků zpracování dokumentu. Z důvodu přímé závislosti kontroly na prováděných změnách v průběhu vývoje je nežádoucí provádět validaci při každé změně v textu. Je nezbytné co možná nejvíce zredukovat počet zpracování, což znamená, že změny musí být zpracovávány dávkově.

## 4.1. Obarvení kódu

Prvním krokem je odlišení řetězců znaků jazyka XML od řetězců jazyka Java. K tomuto účelu je v API Eclipse několik nástrojů, které jsou popsány v kapitole 5.1.

Dalším krokem je vhodná volba sady barev pro jednotlivé řetězce obou jazyků. Výběr barev tohoto projektu byl inspirován základním barevným rozložení vývojového prostředí NetBeans IDE. Toto rozhodnutí bylo motivováno subjektivním pocitem vyšší přehlednosti a kontrastu s ostatním textem než v případě základního barevného rozložení zvoleném v Eclipse IDE.

K problému dochází v případě barevného rozlišení literálů typu String. Ty se totiž vyskytují v kontextu obou jazyků a je nezbytně nutné oba výskyty dostatečně vzájemně odlišit. V souladu s myšlenkou zachování nejvyšší možné přehlednosti budou tyto literály odlišeny barevně, tak aby barvy byly buď komplementární nebo z velmi blízkého okolí barevné palety.

## 4.2. XML parser

V dosavadním textu byly na parser stanoveny základní požadavky na funkčnost, které jsou implementovaným pluginem očekávány. Výsledný parser tohoto projektu tedy musí splňovat následující:

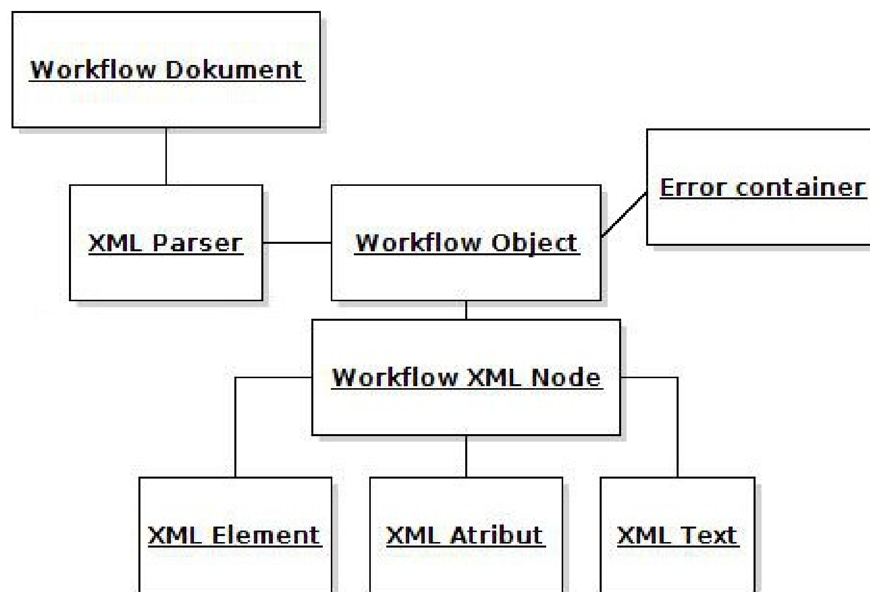
1. Schopnost ověřit, že obsah dokumentu splňuje požadavky na validitu XML[14].
2. Sestavení objektu reprezentujícího strukturu a data parsovaného XML dokumentu umožňujícího přístup validátorům.
3. Uchování pozice jednotlivých rozparsovaných entit (fragmentů) XML v původním textu pro potřeby označení nalezených chyb.

Dále je pak nutné určit požadavky na validitu XML. Ty jsou dány specifikací konsorcia W3C a jsou[14]:

1. Dokument má právě jeden kořenový element.
2. Element, jenž má otevírací tag umístěn v jednom elementu, bude mít ve stejném elementu umístěn uzavírací tag.

Součástí standardní knihovny Javy, balík `javax.xml.parsers`, jsou dva XML parsery. Jedná se o parsery DOM a SAX.[11]

Parser SAX je událostmi řízený mechanismus se sériovým přístupem k dokumentu XML. Jeho největšími výhodami jsou rychlost zpracování dokumentu a nízké požadavky na paměť. Tato kombinace splňuje část požadavků na parser pro tento plugin, avšak velkou nevýhodou a zároveň hlavním důvodem pro zamítnutí jeho použití je jeho bezestavovost. Parser totiž zpracovává pouze aktuálně předanou část dat, bez dalšího kontextu XML dokumentu. V případě validace kódu BeanShellu to znamená nemožnost




**Obrázek 2.** Analyticky model implementovaného parseru

přístupu k ostatním elementům obsahujícím důležité informace o jménech a hodnotách definovaných proměnných a tudíž znehodnocení celé validace. Pro potřeby validace je nutné mít okamžitý přístup k celému stromu. [7]

Požadavek na přístup k celému stromu XML dokumentu splňuje parser DOM. Ten oproti předchozímu parseru v paměti vytvoří stromovou strukturu odpovídající struktuře parsovaného dokumentu. Nevýhodou tohoto přístupu je vyšší paměťová náročnost. Díky své robustnosti by byl skvělou volbou. Požadavky na XML parser však tuto implementaci vylučují z důvodu úplné absence zpětného určení pozice elementu ve zpracovávaném dokumentu[12].

Z výše uvedených faktů vyplývá, že ani jeden z parserů neposkytuje funkčnost zpětně určit pozici prvku XML v dokumentu. Ta je však pro schopnost označení chyby v okně editoru zcela nezbytná. V důsledku těchto faktů bylo přistoupeno k rozhodnutí implementovat vlastní, pro projekt specifický, parser, který chybějící funkčnost poskytne. Z analýzy vyplývá, že výsledný parser bude postaven na podobném funkčním principu jako výše popsaný DOM, kdy je celá logická struktura dokumentu uchována v paměti ve formě grafové struktury. Model výše popsaného parseru je znázorněn na obrázku 2.

## 4.3. Validátor kódu

Mezi základní požadavký na funkčnost pluginu patří schopnost provedení validace vnitřní logické struktury dokumentu Workflow a validace BeanShell kódu. Význam validace struktury spočívá v ověření dostupnosti kroků business procesu definovaného ve vyvíjeném dokumentu. Druhou zmíněnou validaci lze dále rozdělit na dílčí validační procesy. Těmi jsou syntaktická kontrola BeanShell kódu, jejíž účelem je odstranit programátorské chyby (překlepy, apod.), a kontrola použitých proměnných a importů.

Účelem této validace je eliminace překlepů v názvech a částečná sémantická kontrola skriptu. Na základě takto specifikovaných požadavků budou vytvořeny tři samostatné validátory. `VariableValidator`, `BeanShellValidator` a `GraphValidator`.

### Validace grafu

Validátor grafu `GraphValidator` provede kontrolu vnitřní struktury Workflow, kdy vyhledá nedostupné (izolované) uzly a neukončené přechody. Pro validaci je použita metoda průchodu grafem do šířky a dvě množiny reprezentující všechny uzly grafu a všechny dosažitelné uzly.

Validace začne ve výchozím uzlu `start-state`. Aktuálně zpracovávaný uzel je přidán do množiny dosažitelných uzlů a provede porovnání uzlů připojených hranou s množinou všech existujících uzlů. Pokud jsou tyto uzly v množině nalezeny, přidá je proces validace do fronty zpracovávaných uzlů. Nejsou-li některé z uzlů nalezeny, byla identifikována definice hrany s neexistujícím cílem.

Po dokončení kontroly je zpracován další uzel ze začátku fronty. Pokud již ve frontě nejsou další uzly, došlo ke zpracování celého grafu. Následně je proveden rozdíl obou množin a v případě, že výslednou množinou je prázdná množina, nebyly nalezeny nedostupné uzly. V opačném případě výsledná množina obsahuje všechny nedostupné uzly.

### Validace syntaxe

Validace BeanShell kódu, prováděná validátorem `BeanShellValidator`, má za cíl odhalit syntaktické chyby a je založená na BeanShell interpreteru popsaném v kapitole 3.2, jehož knihovna je pojmenována `bsh`. Uvedená knihovna implementuje všechny nástroje potřebné pro použití BeanShell skriptů, včetně parseru skriptů, který validátor použije[4].

### Validace použití proměnných a importů

Poslední validátor, tedy `VariableValidator` je určen pro kontrolu používaných proměnných. Účelem je odhalení použití nedefinovaných proměnných, chybějících a duplicitních importů. Dále pak detekuje duplicitní definice proměnných a nesprávný přístup k nim (např. zápis do proměnné, která je XML elementem `variable` definována pouze pro čtení.).

Proces validace je nutné implementovat s ohledem na snadnou rozšiřitelnost a principy objektového paradigmatu. S ohledem na takto stanovené požadavky tak celý proces validace a parsování bude implementován jako samostatný kontroler.

Je vhodné, aby byly jednotlivé validátory z důvodu zachování snadné rozšiřitelnosti pluginu navrženy nezávislé na uzlech Workflow a celkově tak byly od zbytku programu izolovány. Z toho důvodu bude přístup k validátorům řízen managerem, který bude

vytvořen na základě návrhových vzorů `Factory` a `Adapter`[6]. Nezávislosti na uzlech Workflow bude kromě použití vzoru `Factory` pro manager dosaženo definicí společného interface validátorů, který bude navržen podle návrhového vzoru `Visitor`[6].

# Kapitola 5.

# Realizace

## 5.1. Vytvoření editoru

Pro vytvoření editoru, okna uvnitř Eclipse IDE obsahujícího upravovaný text, je nutno v průběhu inicializace předat instanci třídy sloužící pro načtení editovaného dokumentu a instanci třídy obsahující konfiguraci zobrazení a chování editoru.[15]

Třída umožňující přístup editoru ke zpracovávanému souboru se nazývá `WFDocumentProvider`[1] a je předána metodou `setDocumentProvider`. Konfigurace editoru `WFSrcViewConf`, která obsahuje kompletní soubor informací potřebných pro provedení obarvení textu načteného dokumentu, chování editoru při dvojitém kliknutí myší a dalších funkcí, je předána jako parametr metodou `setSourceViewerConfiguration`[17]. Výpis kódu 5.1 obsahuje popsané kroky pro vytvoření editoru.

**Výpis kódu 5.1 Vytvoření editoru**

```
//třída sloužící pro převod RGB na třídu Color a zároveň
//cache používaných barev
private WFColorManager colorManager;

@Override
protected void initializeEditor() {
  super.initializeEditor();
  colorManager = new WFColorManager();
  //nastavení konfigurace zobrazení
  setSourceViewerConfiguration(new WFSrcViewConf(colorManager));
  //nastavení přístupu k souboru
  setDocumentProvider(new WFDocumentProvider());
}
@Override
//uvolenění používaných zdrojů
public void dispose() {
  ((WFDocumentProvider)getDocumentProvider()).dispose();
  colorManager.dispose();
  super.dispose();
}
```

[1] Všechny třídy implementované v rámci pluginu jsou označeny prefixem WF.

Třída WFDocumentProvider voláním metody createDocument provede načtení editorem požadovaného dokumentu a následné prvotní zpracování textového obsahu. Cílem tohoto předzpracování je rozčlenění textu na související bloky, což usnadňuje další kroky pro obarvení obsahu.

Předzpracování je provedeno třídou FastPartitioner, která na základě souboru pravidel definovaných uvnitř třídy WFPartitionScanner rozčlení vnitřní prezentaci dokumentu na oddíly a v závislosti na rozpoznaném textu každému z nich přiřadí příslušný deskriptor (tag). V kódu 5.2 lze vidět postup nutný pro napojení instance sloužící k předzpracování obsahu.

Dokumentu je nakonec přiřazen listener WFValidationController, který předává informace o změnách ve vytvořeném okně editoru validační logice celého pluginu. Toho je docíleno použitím návrhového vzoru Observer[6], v jazyce Java pojmenovaném jako Listener. Této třídě se blíže věnuje kapitola 5.2.

**Výpis kódu 5.2 Přístup k dokumentu a přidání validátoru.**

```
@Override
  protected IDocument createDocument(Object element) throws CoreException{
    doc = super.createDocument(element);
    if (doc != null){
      //proces předzpracování textu otevíraného editorem
      IDocumentPartitioner partitioner =
          new FastPartitioner(new WFPartitionScanner(),
          ITokenDescriptors.Descriptors.getAllDescriptors());
      partitioner.connect(doc);
      //vytvoření kontroleru validace, argument this je instance
      //třídy pro zpracování načteného souboru, argument element
      //je obecný objekt načteného souboru
      documentListener = new WFValidationController(this, element);
      //napojení třídy pro předzpracování dokumentu
      doc.setDocumentPartitioner(partitioner);
      //napojení kontroleru validace
      doc.addDocumentListener(documentListener);
    }
    return doc;
  }
```

Třída WFPartitionScanner rozpoznává celkem tři typy obsahu, které jsou XML komentář, XML instrukce a XML tag. Soubor těchto pravidel je možno nalézt v kódu 5.3. Zbylý text, který nesouhlasí s žádným z uvedených pravidel, je označen tagem DEFAULT_CONTENT_TYPE z interface IDocument[17]. Pravidlo se skládá ze startovní a koncové sekvence znaků a unikátního tagu, který danému textu bude přiřazen. Problém ovšem nastává ve chvíli, kdy má editor odlišit XML element od zbylých dvou konstrukcí. Všechny totiž začínají znakem < a končí znakem > a pravidlo pro XML element by tak bylo aplikovatelné vždy. Toto řeší implementace vlastního pravidla WFTagRule, která provede kontrolu následujícího znaku po detekovaném znaku < a v případě, že odpovídá

znakům uvozujícím instrukci nebo komentář, vrátí již přečtený znak zpět do řetězce zpracovávaných znaků a skončí s negativním výsledkem. To má za následek, že blok nebude označen tagem pro XML element, ale tagem, který odpovídá specifičtějšímu pravidlu, tedy jako XML komentář nebo instrukce.

**Výpis kódu 5.3 Definice pravidel předzpracování dokumentu.**

```
import eu.bcvsolutions.plugin.workflow.ui.textscanners.IWFTokenDescriptors;

public WFPartitionScanner() {
    IToken xmlComment =Descriptors.XML_COMMENT.getToken();
    IToken tag = Descriptors.XML_TAG.getToken();
    IToken xmlInstr= Descriptors.XML_INSTRUCTION.getToken();

    IPredicateRule[] rules = new IPredicateRule[3];
    //jednotlivá pravidla detekující XML entity
    rules[0] = new MultiLineRule("<!--", "-->", xmlComment); //XML komentář
    rules[1] = new SingleLineRule("<?", "?>", xmlInstr); //XML instrukce
    rules[2] = new WFTagRule(tag); //XML element
    setPredicateRules(rules);
  }
  private class WFTagRule extends MultiLineRule {
    public WFTagRule(IToken token)
    { super("<", ">", token); }

    protected boolean sequenceDetected(ICharacterScanner scanner,
      char[] sequence, boolean eofAllowed) {
      int c = scanner.read();
      if (sequence[0] == '<') {
        if (c == '?') {
          // detekována XML instrukce
          scanner.unread(); //nutno vrátit znak zpět
          return false;
        }
        if (c == '!') {
          // detekován XML komentář
          scanner.unread();//nutno vrátit znak zpět
          return false;
        }
      } else if (sequence[0] == '>') {
        //konec detekované sekvence, vrátí znak z proměnné c zpět
        scanner.unread();
      }
      return super.sequenceDetected(scanner, sequence, eofAllowed);
    }
```

Třída `WFSrcViewConf` obsahuje kompletní konfiguraci celého editoru. Nastavení obarvení obsahu je dosaženo implementací metody `getPresentationReconciler` a metody `getConfiguredContentTypes`, která poskytuje informace o známých typech obsahu. Pro obarvení textu obsaženého v editoru je nutné již předzpracovanou

vnitřní reprezentaci textu opět rozčlenit. Toho je docíleno na základě komplexnějších filtrů připravených pro každý z definovaných typů obsahu viz. kód 5.3.

Jednoduchým příkladem tohoto filtru je třída `XMLInstructionScanner`, která zpracovává textový oddíl, v němž byla v procesu předzpracování rozpoznána instrukce XML. Třída opět obsahuje soubor pravidel pro filtraci textu, avšak tentokrát si s sebou pravidla již nesou informaci o přidělené barvě, jak je možné vidět v kódu 5.4

**Výpis kódu 5.4 Filtr pro obarvení XML instrukcí.**

```
public class XMLInstructionScanner extends RuleBasedScanner {

  public XMLInstructionScanner(WFColorManager manager) {
    //informace o formátování rozpoznaného textu
    IToken procInstr =
      new Token(
        new TextAttribute(
          manager.getColor(IColorConstants.XML_PROC_INSTR)));

    IRule[] rules = new IRule[2];
    //Pravidlo pro rozpoznání XML instrukce
    rules[0] = new SingleLineRule("<?", "?>", procInstr);
    // Pravidlo pro rozpoznání neviditelných znaků
    rules[1] = new WhitespaceRule(new WFWhitespaceDetector());
    setRules(rules);
  }
}
```

V kódu 5.5 lze nalézt zkrácený příklad implementace metody, jež má za úkol obarvení textu a jeho následnou údržbu, `getPresentationReconciler`. Uvedený kód provádí zpracování XML instrukce. Obdobným způsobem jsou přiřazeny filtry i pro ostatní typy textu, v tomto případě celkem čtyři. Filtry pro zpracování XML elementů a XML komentářů, které jsou v podstatě identické s již popsaným filtrem, zde uvádět nebudu.

**Výpis kódu 5.5 Nastavení obarvení textu XML instrukce.**

```
  public IPresentationReconciler getPresentationReconciler(ISourceViewer
        sourceViewer) {
    //údržba prezentace dokumentu v závislosti na provedených změnách
    PresentationReconciler reconciler = new PresentationReconciler();
    reconciler.setDocumentPartitioning(
    getConfiguredDocumentPartitioning(sourceViewer));

    DefaultDamagerRepairer dr;
    dr = new DefaultDamagerRepairer(getXMLInstructionScanner());
    // slouží k určení regionu v textu, který
    //bude nutno upravit v důsledku změny v textu
    reconciler.setDamager(dr,
      ITokenDescriptors.Descriptors.XML_INSTRUCTION.toString());
    // slouží k určení potřebných oprav v zobrazení textu
    reconciler.setRepairer(dr,
```

```
      ITokenDescriptors.Descriptors.XML_INSTRUCTION.toString());
    return reconciler;
  }
  // Filtr zpracovávající XML instrukce
  private XMLInstructionScanner getXMLInstructionScanner() {
    if (instructionScanner == null) {
      instructionScanner = new XMLInstructionScanner(cManager);
      //nastavení formátu a barvy textu, který nebyl filtrem rozpoznán
      instructionScanner.setDefaultReturnToken(new Token(
        new TextAttribute(cManager.getColor(
        IColorConstants.DEFAULT))));
    }
    return instructionScanner;
  }
```

Filtr `WFJavaCodeScanner` pro zpracování textového obsahu umístěného mezi XML elementy lze nalézt ve výpisu kódu 5.6. Každé klíčové slovo, datový typ a konstanta, které mají být nějakým způsobem obarveny či jinak zvýrazněny, je potřeba filtru samostatně specifikovat a předat. Z každého slova je následně vytvořeno samostatné pravidlo nesoucí informaci o jeho formátování. Například definice proměnné `keyReturn` kromě barvy také upravuje řez písma.

**Výpis kódu 5.6 Filtr pro obarvení programového kódu v jazyce Java.**

```
public class WFJavaCodeScanner extends RuleBasedScanner {
  //rezervovaná slova jazyka Java rozdělená do skupin pro potřeby obarvení
  private static final String[] KEYWORDS = {"abstract", "break", "case",
  "catch",  "class", "continue", "default", "do", "else", "extends",
  "final", "finally", "for", "if", "implements", "import", "instanceof",
  "interface", "native", "new", "package", "private", "protected",
  "public", "static", "super", "switch",  "synchronized", "this",
  "throw", "throws", "transient", "try", "volatile", "while" };
  private static final String[] KEYWORD_RETURN = { "return" };
  private static final String[] TYPES = { "void", "boolean", "char",
  "byte", "short", "int", "long", "float", "double" };
  private static final String[] CONSTANTS = { "false", "null", "true" };
  private static final String[] COMMENTS_TASKS = { "TODO", "TASK" };

  public WFJavaCodeScanner(WFColorManager manag) {
    //vytvoření instancí vlastností formátování textu pro
    //jednotlivé skupiny
    IToken keyword = new Token(new TextAttribute(manag.getColor(
    IColorConstants.JAVA_KEYWORD), null, SWT.BOLD));
    IToken keyReturn = new Token(new TextAttribute(manag.getColor(
    IColorConstants.JAVA_KEYWORD_RETURN), null, SWT.BOLD));
    IToken type = new Token(new TextAttribute(manag.getColor(
    IColorConstants.JAVA_TYPE)));
    IToken string = new Token(new TextAttribute(manag.getColor(
    IColorConstants.JAVA_STRING)));
    IToken comment = new Token(new TextAttribute(manag.getColor(
```

```
    IColorConstants.JAVA_SINGLE_LINE_COMMENT)));
    IToken other = new Token(new TextAttribute(manag.getColor(
    IColorConstants.DEFAULT)));
    IToken task = new Token(new TextAttribute(manag.getColor(
    IColorConstants.JAVA_TASK)));

    //kontejner pro pravidla
    List<IRule> rules = new ArrayList<>();
    //pravidlo pro rozpoznání komentáře
    rules.add(new EndOfLineRule("//", comment));
    //pravidla pro rozpoznání literálů typu char a String
    rules.add(new SingleLineRule("\"", "\"", string, '\\'));
    rules.add(new SingleLineRule("'", "'", string, '\\'));
    //pravidlo pro rozpoznání tzv. whitespace znaků
    rules.add(new WhitespaceRule(new WFWhitespaceDetector()));

    //pravidla pro rozpoznání klíčových slov, zákl. datových typů
    //a konstant.
    WordRule wordRule = new WordRule(new JavaWordDetector(), other);
    for (int i = 0; i < KEYWORD_RETURN.length; i++)
      wordRule.addWord(KEYWORD_RETURN[i], keyReturn);
    for (int i = 0; i < KEYWORDS.length; i++)
      wordRule.addWord(KEYWORDS[i], keyword);
    for (int i = 0; i < TYPES.length; i++)
      wordRule.addWord(TYPES[i], type);
    for (int i = 0; i < CONSTANTS.length; i++)
      wordRule.addWord(CONSTANTS[i], type);
    for (int i = 0; i < COMMENTS_TASKS.length; i++)
      wordRule.addWord(COMMENTS_TASKS[i], task);
    rules.add(wordRule);
    IRule[] result = new IRule[rules.size()];
    rules.toArray(result);
    setRules(result);
  }

  //realizuje detekci klíčových slov jazyka Java
  protected class WFJavaWordDetector implements IWordDetector {
    public boolean isWordPart(char character) {
      return Character.isJavaIdentifierPart(character);
    }
    public boolean isWordStart(char character) {
      return Character.isJavaIdentifierStart(character);
    }
  }
}
```

## 5.2. Kontroler validace

Třída `WFValidationController`, jen řídí proces validace, je potomkem třídy `Thread`, která je součástí standardní knihovny jazyka Java, což z třídy vytvoří samostatné programové vlákno a umožní běh nezávisle na ostatních součástech pluginu.[3]

Třída obsahuje dvě důležité privátní konstanty, které řídí její běh a určují frekvenci zpracování upravovaného kódu v editoru. Těmi jsou `SLEEP_TIME`, jejíž hodnota určuje periodu kontroly, zda došlo v dokumentu ke změně, a `WAIT_TIME`, jenž určuje minimální dobu, která musela uběhnout od poslední provedené změny v dokumentu. Kombinace uvedených faktorů umožnila odstranit nutnost reagovat na každou změnu obsahu editoru, což umožňuje tyto změny kumulovat a zpracovávat dávkově.

**Výpis kódu 5.7 Spuštění validace.**

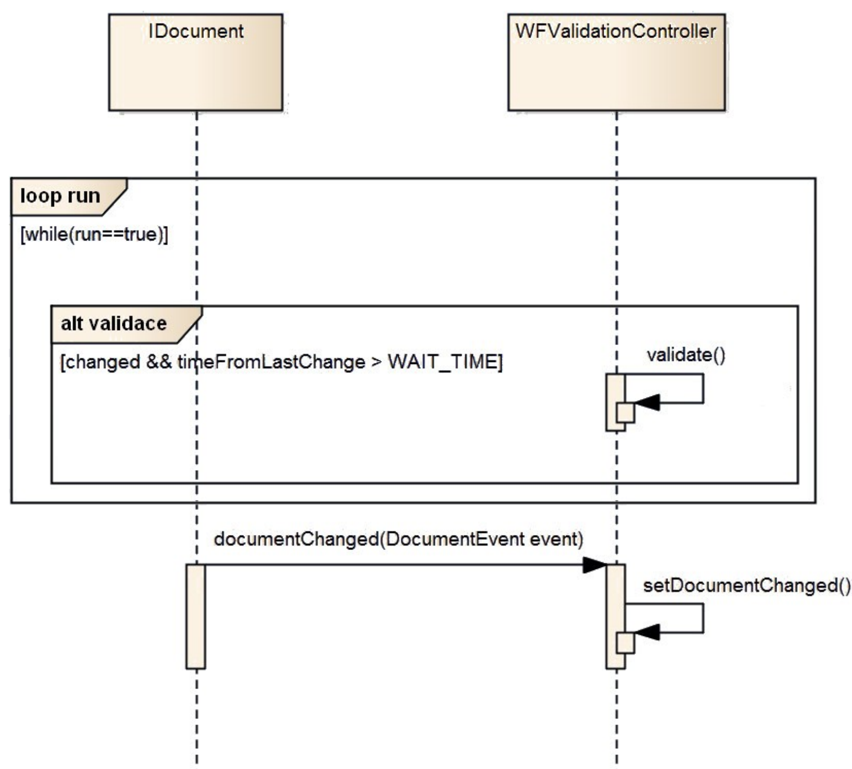
```
private void validate() throws BadLocationException, CoreException {
    //odstraní notace chyb v GUI editoru
    clearAnnotations(documentProvider, element);

    //vytvoří vnitřní reprezentaci dokumentu a provede její validaci
    WFObject wf = null;
    wf = this.parser.parse(this.document.get());
    wf.validate(validatorFactory);

    //pokud byly nalezeny chyby, jsou předány editoru k zobrazení
    if (wf.getErrors().size() > 0) {
      try {
        for (IWFError e : wf.getErrors()) {
          addAnnotation(
            this.getMarker(this.element, e.getMessage(), e.getPosition()),
            e.getPosition(),
            this.documentProvider,
            this.element);
        }
      } catch (BadLocationException ex) {
        System.err.println(ex.getLocalizedMessage());
        ex.printStackTrace();
      } catch (WFException ex) {
        System.err.println(ex.getLocalizedMessage());
        ex.printStackTrace();
      } catch (CoreException ex) {
        System.err.println(ex.getLocalizedMessage());
        ex.printStackTrace();
      }
    }
    //zamezí validaci do doby než bude obsah opět změněn
    changed = false;
  }
```

Proces kontroly změn, který je vyobrazen na obr. 3, je implementován jako nekonečný cyklus. V závislosti na hodnotách proměnných `changed` a `timeOfLastChange` pak buď spustí nebo přeskočí validaci dokumentu. Následně běh vlákna uspí, čímž umožní kumulaci většího množství změn.

Hodnota obou proměnných je nastavena v metodě `setDocumentChanged`, což je implementace metody z interface `IDocumentListener`. Problémem se ukázalo být vhodné nastavení hodnot již zmíněných konstant `SLEEP_TIME` a `WAIT_TIME`. Při jejich nízkých hodnotách byl uživatel obtěžován častým spouštěním validace. Naopak při vysokých hodnotách plugin neposkytoval uživateli dostatečnou zpětnou vazbu. Jako optimum se v praxi ukázala hodnota 200 ms pro konstantu `SLEEP_TIME` a 750 ms pro konstantu `WAIT_TIME`.

Spuštění validace je docíleno zavoláním metody `validate`. Ve výpisu 5.7 lze vidět, že její zavolání má v první řadě za následek odstranění všech upozornění na chyby v GUI editoru. Poté je provedeno rozparsování obsahu editoru, což má za následek získání paměťové reprezentace daného XML dokumentu, instanci třídy `WFObject`. Nad tímto objektem je spuštěn proces validace a v případě, že byly nalezeny chyby, jsou metodou `addAnnotation` předány editoru, viz výpis kódu 5.8.

Atribut metody typu `WFPosition` je speciální třída, jenž udržuje informaci o pozici prvků, ze kterých je složena třída `WFObject`. Pozice je určena dvěma hodnotami,




**Obrázek 3.** Cyklus spuštění validace

offsetem a délkou původního textu. Nakonec jsou editoru postupně předány všechny nalezené chyby. Oběma třídám a procesu parsování se hlouběji věnuje následující kapitola 5.3.

**Výpis kódu 5.8 Předání správce chyb editoru.**

```
private void addAnnotation(IMarker marker, WFPosition position,
    IDocumentProvider documentProvider, Object element) {
    //Získá model spravující anotace
    IAnnotationModel am =
      documentProvider.getAnnotationModel(element);
    //Opětovné napojení na dokument, přidání chyby a následné odpojení
    am.connect(document);
    SimpleMarkerAnnotation san = new SimpleMarkerAnnotation(
      "eu.bcvsolutions.plugin.workflow.ui.errannotation",
      marker);
    san.setText(marker.getAttribute(IMarker.MESSAGE, "..."));
    am.addAnnotation(san, position);
    am.disconnect(document);
  }
```

## 5.3. Parser

Zpracování dokumentu je provedeno třídou `WFXMLParser`, která v průběhu zpracování validuje XML dokument podle požadavků stanovených v kapitole 4.2. Zavoláním metody `parse`, která obdrží obsah dokumentu jako parametr typu `String`, je provedeno rozčlenění obsahu a vytvoření jeho paměťové reprezentace, velmi podobné jako například u parseru DOM. Tato reprezentace má podobu stromového grafu složeného z uzlů, kdy každý uzel obsahuje reference na své potomky a svého předka. Jedinou výjimkou je kořenový uzel, který je speciální a žádného předka nemá.

V kódu 5.9 je možno vidět proměnné `start` a `end`, které slouží jako zarážky a k vyčlenění částí textu uzavřených uvnitř XML elementu. Hodnota obou proměnných má ještě jeden velmi důležitý význam – určení pozice XML elementu v dokumentu. Tu si každý uzel grafu, jenž reprezentuje element v paměti, uchovává pro potřeby označení nalezených chyb v GUI editoru.

V první řadě je vytvořena instance kořenového elementu `WFXMLRootNode`, který je okamžitě přiřazen do proměnné `element`. Tato proměnná v programu reprezentuje poslední zpracovaný element, jinými slovy předchůdce dále zpracovávaných elementů a textových polí.

**Výpis kódu 5.9 Parsování dokumentu.**

```
public WFObject parse(String document) {
    builder = new WFObjectBuilder();
    int start = 0;
    int end = 0;
    //vytvoření instance kořenového elementu pamět'ové reprezentace
    //Workflow dokumentu
    WFXMLRootNode root = new WFXMLRootNode();
    WFXMLAbstrElement element = root;
    //zpracování XML instrukcí uvedených na začátku dokumentu
    end = parseInstructions(document, root, builder);
    do {
    ...

    //kód zpracovávající XML dokument je příliš dlouhý a
    //proto zde není uveden celý je možné ho nalézt na
    // přiloženém optickém médiu ve zdrojových kódech

    ...
    //cyklus je ukončen jakmile dorazí při hledání XML elementů
    //na konec dokumentu
    } while (start < document.length() &&
      end < document.length() &&
      start >= 0 && end >= 0);

    builder.setWorkflow(root);
    return builder.getBuildedWorkflow();
}
```

Následně jsou zpracovány XML instrukce, které jsou nepovinnou součástí dokumentu. I když s tímto typem obsahu plugin žádným způsobem nepracuje, je zpracován a následně vložen do datové struktury vnitřní reprezentace dokumentu pro případ, že by v budoucnu byly vzneseny požadavky na validaci tohoto obsahu. Po zpracování instrukcí, které se mohou vyskytovat na úplném začátku XML dokumentu, přejde instance parseru ke zpracování obsahu XML dokumentu Workflow.

Výpis kódu 5.10 názorně zobrazuje výše popsaný princip zpracování textové prezentace XML elementu parserem. V textu detekuje pozici otevírací sekvence znaků, v tomto případě `<?` a následně v cyklu prochází text a hledá uzavírací sekvenci `?>`. V případě, že není uzavírací sekvence nalezena, je vytvořena instance třídy `WFError` reprezentující nalezený problém. Chyba je pak popsána pozicí, typem chyby a krátkým textovým popisem. Následně je vrácena hodnota `-1`, což dalšímu běhu programu sdělí, že při zpracování bylo dosaženo konce dokumentu a ukončí se proces parsování. V případě, že je vše v pořádku, nastaví se hodnota start na pozici další hledané sekvence v řadě a opět proběhne popsaný proces. V případě, že hledaná startovní sekvence neodpovídá pozici nejbližšího znaku <, je vrácená hodnota pozice poslední uzavírací sekvence navýšena o délku této sekvence.

**Výpis kódu 5.10 Parsování instrukce.**

```
private int parseInstructions(String document,
    WFXMLRootNode root, WFObjectBuilder builder) {
    //získání pozice otevírací sekvence znaků
    int start = document.indexOf("<?"), end = 0;

    while (start >= 0) {
      //nalezení pozice uzavírací sekvence znaků
      end = document.indexOf("?>", start);
      if (end < 0) {
      //pokud nebyla uzavírací sekvence nalezena je vygenerována chyba
      //a uložena pro pozdější zpracování
        builder.addError(new WFParseError(IWFError.Type.XML,
          new WFPosition(start, 2),
          "Missing XML instruction closing tag!"));
        return -1;
      }
      try {
        //přidání instrukce do kořenového elementu
        root.addInstruction(new WFXMLInstruction(root));
      } catch (ParserException ex) {
        builder.addError(new WFParseError(new WFPosition(start, end),
          ex.getMessage()));
      }
      //posunutí zarážky start k další otevírací sekvenci
      start = document.indexOf("<?", end);
      //pokud již další sekvence nebyla nalezena je vrácena pozice
      //uzavírací sekvence posunuté o délku této sekvence
      if (start != document.indexOf('<', end)) {
        return end + 2;
      }
    }
    return end + 2;
  }
```

Stejným způsobem jsou pak parsovány elementy a komentáře XML. Text umístěný mezi uzavírací sekvencí znaků a sekvencí otevírací je přiřazen do XML elementu, který je aktuální hodnotou proměnné `element`. Proměnná element obsahuje nadřazený XML element, ve kterém je text obsažen, v kontextu programu je to pak reprezentace elementu, ke kterému zatím nebyl detekován uzavírací tag.

Výsledná instance třídy WFObject, která slouží jako obálka stromu reprezentujícího dokument, v sobě udržuje informace o nalezených chybách a o podobě zpracovaného dokumentu. Poskytuje unifikovaný přístup pro získání nalezených chyb a pro spuštění validace. Ve výpisu kódu 5.11 je možné vidět, že parametru `validator` je nastaven kontejner pro nalezené chyby, který však již může obsahovat chyby z procesu parsování.

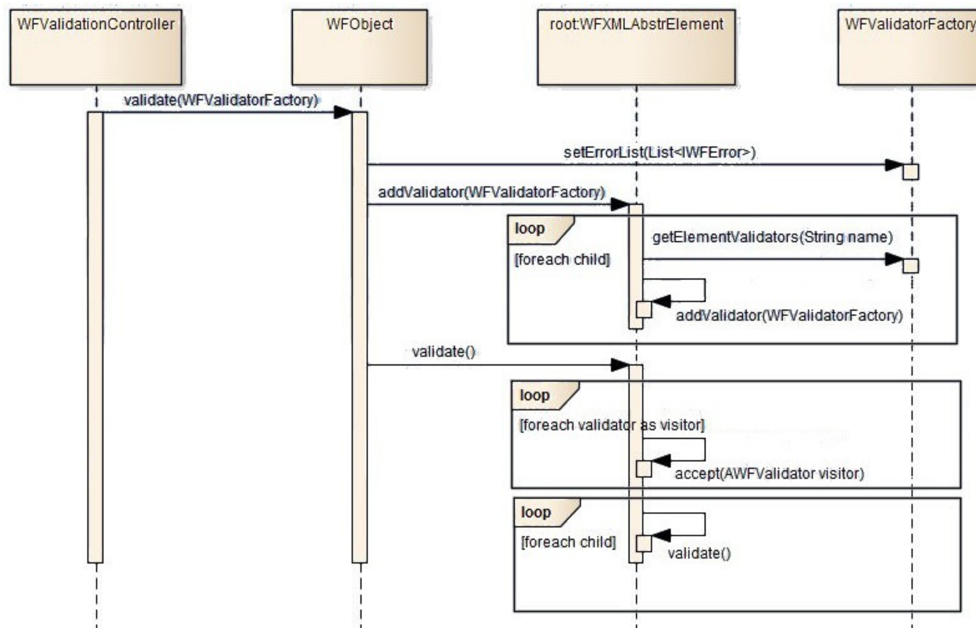
**Výpis kódu 5.11 Spuštění validace nad daty.**

```
public void validate(WFValidatorFactory validator){
    //nastavení společného kontejneru chyb
    validator.setErrorList(errorList);
    if(root == null) return;
    //start procesu získání validátorů
    root.addValidator(validator);
    //spuštění validace
    root.validate();
  }
```

Poté je na kořenovém elementu provedeno přiřazení příslušných instancí validátoru, jehož průběh popisuje kód 5.12. Uzel stromu, v tomto případě přímo kořenový element, získá z předaného parametru `factory` seznam validátorů určených přímo pro element s daným jménem a poté rekurzivně provede stejný proces na všech svých potomcích.

**Výpis kódu 5.12 Přiřazní validátorů.**

```
@Override
public void addValidator(WFValidatorFactory factory) {
  //získá seznam validátoru a přiřadí ho instanci
  this.addValidator(factory.getElementValidators(name));
  //rekurzivní volání této metody nad všemi potomky elementu
  for(WFXMLAbstrElement el: elementList ){
    el.addValidator(factory);
  }
}
//uchování získaných validátorů
protected void addValidator(
  List<AWFValidator<WFXMLAbstrElement>> validator) {
  for (AWFValidator<WFXMLAbstrElement> v : validator) {
    validators.add(v);
  }
}
//metoda třídy WFValidatorFactory poskytující požadované validátory
public List<AWFValidator<WFXMLAbstrElement>> getElementValidators(
  String elementName){
  List<AWFValidator<WFXMLAbstrElement>> list = new LinkedList<>();

  list.addAll(genericElementValidators);
  if(elementValidators.containsKey(elementName)) {
    Set<AWFValidator<WFXMLAbstrElement>> s;
    s=elementValidators.get(elementName);
    list.addAll(elementValidators.get(elementName));
  }
  return list;
}
```




**Obrázek 4.** Proces přidání validátoru

Jakmile jsou všechny prvky připraveny, dojde ke spuštění samotné validace, která opět probíhá rekurzivně nad celou stromovou strukturou. Popsaný proces validace lze vidět na obrázku 4. Po ukončení validace je na instanci `WFObject` zavolána metoda `getErrors`, jejíž návratovou hodnotou je kolekce nalezených chyb. Následně je přes tyto chyby iterováno a v každé iteraci je chyba upravena do formátu API Eclipse. Poté je předána editoru k zobrazení.

**Výpis kódu 5.13 Popisek.**

```
@Override
public void accept(Visitor<WFXMLAbstrElement> visitor) {
  visitor.visit(this);
}
@Override
public void validate() {
  for (AWFValidator<WFXMLAbstrElement> visitor: validators ){
    this.accept(visitor);
  }
  for(WFXMLAbstrElement elem: getElementNodes()){
    elem.validate();
  }
}
```

## 5.4. Validátory obecně

Všechny validátory jsou dostupné pomocí tovární třídy `WFValidatorFactory`, která byla navržena zjednodušením návrhového vzoru `Abstract Factory`. Jak je možné

vidět v kódu 5.14, třída značně ulehčuje přidání dalšího validátoru. Stačí v metodě `init` přidat do kontejneru validátoru nový prvek ve tvaru klíčové slovo, označující XML element, ke kterému se validátor vztahuje, a instanci nového validátoru. Třída `WFValidatorFactory` také poskytuje interface `IWFErrorContainer` a figuruje jako kontejner pro jednotlivé validátory, kteří do ní ukládají všechny nalezené problémy.

**Výpis kódu 5.14 Přidání nového validátoru.**

```
//parametr elementName označuje jméno elementu jemuž
//validátor přísluší
public void addElementValidator(
  String elementName,
  AWFValidator<WFXMLAbstrElement> validator){
    //pokud neexistuje množina validátorů pro daný element
    //dojde k jejímu vytvoření
    if(!elementValidators.containsKey(elementName)){
    elementValidators.put(
      elementName,
      new HashSet<AWFValidator<WFXMLAbstrElement>>());
  }
  //přidání elementu do příslušné množiny
  elementValidators.get(elementName).add(validator);
}
```

Validátory mají společného předka, jímž je abstraktní třída `AWFValidator`, kterou lze nalézt v kódu 5.15. Tato třída implementuje interface `Visitor`, čímž je dosaženo snadné rozšířitelnosti pro potřeby dalšího vývoje. Dále definuje základní konstruktor, jenž nastavuje jméno daného validátoru a kontejner `IWFErrorContainer`, do kterého jsou v průběhu celé validace ukládány nalezené chyby.

**Výpis kódu 5.15 Abstraktní předek všech validátoru.**

```
public interface IWFVisitor<T extends WFNode> {
  public void visit(T node);
}
public abstract class AWFValidator<T extends WFNode>
    implements IWFVisitor<T> {
  private IWFErrorContainer ec;
  private String name;

  public AWFValidator (String name, IWFErrorContainer ec){
    this.ec= ec;
    this.name = name;
  }
  //metoda poskytující přístup k obsahu uzlu logické reprezentace
  //zpracovaného dokumentu XML
  public final void visit(T node){
    try{
      //volání metody implementované potomky
      makeVisit(node);
```

```
      //odchytí případné chyby v implementaci validátoru
      //což zachová funkčnost ostatních součástí pluginu
      } catch (final Throwable ex){
        ex.printStackTrace();
        this.addError(new IWFError() {
          @Override
          public Type getType() {
            return Type.UNSPECIFIED;
          }
          @Override
          public WFPosition getPosition() {
            //vrátí pozici 0
            return new WFPosition();
          }
          @Override
          public String getMessage() {
            StringWriter sw = new StringWriter();
            ex.printStackTrace(new PrintWriter(sw, true));

            BufferedReader br = new BufferedReader(
                new StringReader(sw.toString()));
            String message="";
            try {
              message= br.readLine();
            } catch (IOException e) {
              e.printStackTrace();
            }
            return name+" occured unexpected error and is unavailable.\n"
                +message;
          }
        });
      }
    }
    protected void addError(IWFError error){
      this.ec.addError(error);
    }
    //abstraktní metoda, kterou implementují potomci
    protected abstract void makeVisit(T node);
  }
```

## 5.5. GraphValidator

Validace logické struktury Workflow je navázána na uzel `process-definition` a je možné ji rozdělit do tří kroků. Prvním krokem je průchod celým stromem instance `WFObject`, jenž má za cíl vytvořit množinu uzlů obsažených v daném workflow a separovat startovací a cílový uzel. Uvedený postup lze nalézt v kódu 5.16.

**Výpis kódu 5.16 Vytvoření množiny existujících uzlů.**

```
public class GraphValidator extends AWFValidator<WFXMLAbstrElement> {
  //název elementu, ke kterému je validátor primárně vázán
  //není nezbytnou součástí třídy
  public static final String BOUND_TO = "process-definition";

  public GraphValidator(IWFErrorContainer ec) {
    super("GraphValidator", ec);
  }
  //spuštění validace
  @Override
  protected void makeVisit(WFXMLAbstrElement node)  {
    assert node.getName().toLowerCase().equals(BOUND_TO);

    //třída provádějící porovnání existujících uzlů grafu a
    //dostupných uzlů grafu
    GraphBuilder gb = new GraphBuilder();

    for (WFXMLAbstrElement element : node.getElementNodes()) {
      //přidání uzlu do množiny existujících uzlů
      if(element.getName().matches(
      "\\s*((node)||(decision)||(state)||(join)||(fork)||(task-node))\\s*"
      )){
        gb.addNode(element);
      }
      //detekce a přidání startovního uzlu
      if(element.getName().matches("\\s*start-state\\s*")){
        gb.addStartNode(element);
      }
      //detekce a přidání koncového uzlu
      if(element.getName().matches("\\s*end-state\\s*")){
        gb.addEndNode(element);
      }
    }
    //spuštění validace nad předpřipravenými daty
    gb.validate();
    for(IWFError err:gb.getErrors()){
      this.addError(err);
    }
  }
}
```

Následujícím krokem je kontrola existence startovního uzlu a průchod logické struktury Workflow. Z důvodu, že logická struktura je grafem, bylo možné zvolit algoritmus průchodu grafem do šířky nebo do hloubky. Mezi oběma možnostmi není pro potřeby pluginu většího rozdílu. Zvolena byla metoda průchodu grafem do šířky, tedy algoritmus realizovaný pomocí fronty, s počátkem v uzlu `start-state`, neboť implementace byla o něco málo jednodušší.

**Výpis kódu 5.17 Validace vnitřní struktury Workflow**

```
public void validate() {
    this.nodesQueue = new LinkedBlockingQueue<>();

    //kontrola zda proměnná sn obsahuje startovní uzel
    if (sn != null){
      //start průchodu grafem
      walkThrough(sn);
    }else{
      //přidání chyby obsahující informace o nenalezení
      //startovního uzlu
      this.ec.add(new ValidationError(new WFPosition(),
          "Undefined start-state node."));
    }
    //kontrola existence koncového uzlu
    if (en == null){
      this.ec.add(new ValidationError(new WFPosition(),
          "Undefined end-state node."));
    }
    //odstranění všech dosažitelných uzlů z množiny
    //všech existujících uzlů
    removeAllAttainableNodes();
    if (!nodes.isEmpty()) {
      //iterace přes uzly, které zustaly v množině všech uzlů
      //po odstranění dosažitelných uzlů
      for (BuilderNode n : nodes.values()) {
        this.addError("Unreachable node: " + n.getName(),
          n.getWFXMLNode());
      }
    }
    //vyprázdnění všech použitých kontejnerů
    clear();
  }
  //průchod datovou strukturou
  private void walkThrough(BuilderNode bn) {
    //přidání dosažitelného uzlu
    this.attainableNodes.add(bn.getGraphNode());
    //iterace přes všechny hrany vedoucí z uzlu
    for (Transition transition : bn.getTransitions()) {
      //pokud je nalezen uzel v množině existujících uzlů je v
      //případě, že ještě nebyl navštíven přidán do množiny
      //dostupných uzlů a zařazen ke zpracování
      if (nodes.get(transition.to) != null) {
        if (!this.attainableNodes.contains(
                nodes.get(transition.to).getGraphNode())) {
          this.nodesQueue.add(nodes.get(transition.to));
        }
      } else {
        //v případě, že uzel není přítomen v množině všech existujících
        //uzlů, byla detekována hrana grafu k neexistujícímu uzlu
```

```
          this.addError("Unknown transition node: " + transition.to,
              transition.position);
        }
      }
      //pokud není fronta uzlů čekající na zpracování prázdná
      //zpracuje další uzel
      if (!this.nodesQueue.isEmpty()) {
        walkThrough(this.nodesQueue.poll());
      }
    }
```

Každý navštívený uzel je v případě, že se tak ještě nestalo, vložen do množiny aktuálně dosažitelných uzlů. V množině všech uzlů jsou pak vyhledány uzly odpovídající hodnotě atributu `to` všech elementů `transition`, které aktuální uzel obsahuje. Získané uzly jsou následně zkontrolovány, zda již byly navštíveny, a pokud ne, jsou zařazeny na konec fronty. Posledním krokem je porovnání množiny dosažitelných uzlů s množinou všech uzlů.

Uzly, které nejsou obsaženy v obou uvedených množinách, jsou následně označeny jako nedosažitelné. Pro uvedené označení je vytvořena instance `ValidationError` s příslušnou pozicí v dokumentu a textovou informací o daném problému, která je přidána do seznamu nalezených chyb. Kód 5.17 zobrazuje popsaný proces validace.

## 5.6. BeanShellValidator

Tento validátor provádí kontrolu syntaktické správnosti BeanShell skriptu. Pro kontrolu je použita knihovna `bsh`, konkrétně v aktuální verzi 2.1.18, jenž poskytuje potřebnou podporu pro zpracování skriptů.[2] Validátor předpokládá přímý přístup k textovému obsahu uzlu, a z toho důvodu je spouštěn přímo nad uzly `expression`, které jej uzavírají.

**Výpis kódu 5.18 Validace BeanShell skriptu a úprava pozice nalezené chyby v kontextu dokumentu Workflow**

```
import bsh.ParseException;
import bsh.Parser;

  public void makeVisit(WFXMLAbstrElement node) {
    assert node.getName().toLowerCase().equals(BOUND_TO);

    StringBuilder jbsh = new StringBuilder();

    //v případě, že by byl kontrolovaný text rozdělen nějakým XML
    //elementem na více částí dojde k jeho spojení
    int i = 0;
    for (WFXMLText txt : node.getTextNodes()) {
      jbsh.append(txt.getText());
      ++i;
```

```
    }
    String str = jbsh.toString();
    //detekce prázdného výrazu v XML elementu expression
    if (str.trim().equals("")) {
      addError(new ValidationError(
        new WFPosition(node.getPosition().getOffset()
        + node.getPosition().length, 1), "Expression is empty"));
      return;
    }
    //nahrazení XML entint znaky které zastupují
    for (Entry<String, String> entry : xmlReplaceEntities.entrySet()) {
      str = str.replaceAll(entry.getKey(), entry.getValue());
    }
    //přidání nalezeného problému, kdy je text v XML elementu
    //neočekávaně rozdělen dalšími XML elementy
    if (i != 1) System.err.println(
      "ERROR, text split into parts by other element");
    parse(str, node.getTextNodes().get(0).getPosition());
  }
  private void parse(String jbsh, Position pos) {
    //vytvoření instance bsh Parseru
    Parser parser = new Parser(new StringReader(jbsh));
    try {
      do {
        try {
          //pokud nalezene chybu vyvolá výjimku ParseException
          if (parser.Line()) break;
        } catch (ParseException e) {
          //rozparsování chybové zprávy a určení správné pozice v
          //kontextu celého dokumentu
          WFPosition p = parseErrorOffset(jbsh, e.getMessage());
          p.setOffset(pos.getOffset() + p.getOffset());
          this.addError(
            new ValidationError(IWFError.Type.JAVA, p,
                "Unexpected BeanShell code! "
                + e.getMessage().substring(e.getMessage().indexOf(
                "Encountered:"))));
          return;
        }
      } while (true);
    } catch (Throwable e) {
      System.err.println(e.getMessage());
      WFPosition p = parseGeneralErrorOffset(jbsh, e.getMessage());
      p.setOffset(p.getOffset() + pos.getOffset());
      this.addError(new ValidationError(IWFError.Type.JAVA, p,
          "Unexpected BeanShell code! "
          + e.getMessage().substring(e.getMessage().indexOf(
          "Encountered:"))));
    }
  }
```

```
//určení pozice nalezené chyby v BeanShell skriptu vzhledem k
//celému dokumentu
final private WFPosition parseErrorOffset(
    String text, String errorMessage) {

  String[] tokens = errorMessage.split(" ");
  //získání čísla řádku ve skriptu
  int line = Integer.parseInt(
    tokens[4].substring(0, tokens[4].length() - 1));
  //slovo či znak označující problémové místo
  String encWord = tokens[9];
  BufferedReader br = new BufferedReader(new StringReader(text));

  int offset = 0; //pořadí počátečního znaku v kontextu celého dokumentu
  int length = 0; //délka problémového výrazu
  try {
    String s = null;
    //získání pořadí počátečního znaku v kontextu skriptu
    for (int i = 0; i < line - 1; i++) {
      s = br.readLine();
      offset += s.length();
      ++offset;
    }
    s = br.readLine();
    //pokud bylo dosaženo konce skriptu je délka problémového výrazu
    //nastavena na hodnotu délky celého skriptu jinak je nastavena
    //na délku problematického slova
    if (encWord.equals("<EOF>")) {
      length = s.length();
    } else {
      int i = s.indexOf(encWord);
      //úprava pozice na začátek problematického výrazu
      offset += i;
      String nstr="";
      if (i >= 0 ) {
        nstr = s.substring(i).replaceAll("\\s*//.*\$", "");
        length = nstr.length();
      }
    }
  } catch (IOException ex) {
    ex.printStackTrace();
    System.out.println(ex.getMessage());
  }
  return new WFPosition(offset, length);
}
```

Textový obsah uzlu je nejdříve nutné upravit, jak lze vyčíst z výpisu kódu 5.18. To je z důvodu, že XML dokument přikládá některým znakům speciální význam. Pro tyto znaky existují v kontextu XML předdefinované entity. Tabulka 1 obsahuje uvedené znaky a jim přiřazené entity.[14]

| Znak | Předdefinovaná entita |
| --- | --- |
| < | < |
| > | > |
| & | > |
| ’ | ' |
| " | " |

**Tabulka 1.** Tabulka XML předdefinovaných entit

Takto připravený text je předán instanci třídy `Parser` z knihovny Bsh (viz kód 5.18). Následně je v cyklu volána metoda `Line`, která předaný text po řádcích zpracuje a v případě, že je nalezena chyba, vytvoří výjimku typu `ParseException`. Tato výjimka je validátorem zachycena a následně je z ní získána informace o místě výskytu problému a informace o problému samotném. Tuto informaci je dále nutné upravit do kontextu celého dokumentu Workflow, protože získaná informace se vztahuje pouze na text uzavřený v XML elementu `expression`. Po úpravě je informace přidána do kontejneru chyb nalezených validátorem pro pozdější zpracování.

## 5.7. VariableValidator

Kontrola proměnných je spouštěna nad uzly `script` a validuje se, zda jsou použité proměnné a datové typy definovány a pokud je to možné, zda jsou správně použity. Důvodem pro přístup už k elementu `script` a ne elementu `expression` je potřeba přístupu k názvům proměnných definovaných v elementu `variable` a i k BeanShell skriptu v elementu `expression`. Validátor při spuštění provede přípravu proměnných v již zmiňovaném elementu `variable` tak, že je vloží do kontejneru pro existující proměnné, a přípravu základních datových typů jakou jsou např. `String`, `int`, `double` a dalších, vložením do kontejneru pro známé datové typy, viz kód 5.19

**Výpis kódu 5.19 Příprava proměnných definovaných v elementu `variable` a základních datových typů.**

```
private static final String[] TYPES = { "void", "boolean", "char",
  "byte", "short", "int", "long", "float", "double", "String",
  "Object", "Boolean", "Byte", "Short", "Integer", "Long", "Float",
  "Double", "Character", "Exception", "RuntimeException", "Math",
  "System" };

private final void initTypes() {
    for (int i = 0; i < TYPES.length; i++) {
      dataTypesSet.add(TYPES[i]);
    }
  }
private void prepareVariableSet(List<WFXMLAbstrElement> el) {
    for (WFXMLAbstrElement e : el) {
      if (e.getName().equals("variable")) {
      //třída PairBuilder vytvoří v závislosti na získaných
```

```
//informacích třídu Variable obsahující její správný název.
  PairBuilder pb = new PairBuilder();
  for (WFXMLAttribute a : e.getAttributeNodes()) {
    String tmp = a.getName();
    if (tmp.equals("name")) {
      pb.setName(a.getValue().replaceAll("\"", ""));
      continue;
    }
    //atribut mapped-name má větší význam než atribut name
    if (tmp.equals("mapped-name")) {
      pb.setMappedName(a.getValue().replaceAll("\"", ""));
      continue;
    }
    //proměnná určující povolený přístup k def. proměnné
    if (tmp.equals("access")) {
      tmp = a.getValue();
      if (tmp.matches(".*read.*"))
        pb.setAccess(R);
      if (tmp.matches(".*write.*"))
        pb.setAccess(W);
    }
  }
  //získání proměnné se správným názvem
  Variable p = pb.getPair();

  //kontrola zda je proměnná se stejným jménem již definována
  if (variablesMap.containsKey(p.name)) {
    //získání již existující proměnné se stejným jménem
    Variable tmp = variablesMap.get(p.name);
    //pokud mají obě proměnné stejný modifikátor přístupu
    //je vygenerována chyba a ta je vložena do kontejneru
    if ((tmp.access & p.access) != 0) {
      this.addError(new ValidationError(
        IWFError.Type.JAVA,
        e.getPosition(),
        "Variable of the same name already defined!"));
    } else {
      //již existující proměnné je upraven přístup tak,
      //že si svůj původní zachová a je obohacen o nový
      p.access |= tmp.access;
      variablesMap.put(p.name, p);
    }
  } else {
    //vložení nové proměnné
    variablesMap.put(p.name, p);
  }
} else if (e.getName().equals("expression")) {
  //pokud je nalezený název elementu expression a již byl element
  //se stejným názvem nalezen dojde k vygenerování a uložení chyby
  if (expression != null)
```

```
          this.addError(new ValidationError(
            IWFError.Type.JAVA,
            e.getPosition(),
            "Unexpected number of expression tags!"));
        expression = e;
      }
    }
    //modifikátor zanoření proměnné do bloků kódu určující
    //její lokální platnost
    ++varDepth;
  }
```

Dalším krokem je již samotná validace BeanShell skriptu, který je uzavřen v elementu `expression`. Jak je možno vidět v kódu 5.20, tak předaný text je čten po znacích, kdy je přečtený znak porovnáván s jednou ze čtyř možností znaků, která následně určí, jak bude s doposud přečtenými znaky zacházeno. Tyto znaky jsou:

- `\n`
- `}`
- `{`
- `;`

V případě, že přečtený znak žádné ze čtyř možností neodpovídá, je přidán do proměnné `token`, která je instancí typu `StringBuilder`. Prvním kontrolovaným znakem je `\n`. V tomto případě je provedena kontrola, zda přečtený řetězec není komentářem. Pokud řetězec komentáři odpovídá, je zahozen, a proces validace pokračuje s prázdným řetězcem v proměnné `token`.

Dalším hledaným znakem je `}`. Pokud je nalezen tento znak, dojde ke snížení hodnoty proměnné `varDepth` udržující množství zanořených bloků kódu. Následně dojde k zavolání metody starající se o proces správy proměnných. Ta projde množinu známých proměnných a odstraní všechny, u nichž je hloubka zanoření, v níž byly definovány, větší než aktuální hloubka zanoření. Informace o hloubce zanoření konkrétní proměnné je držena v atributu `depth`. Doposud nashromážděný řetězec znaků zůstane nezměněn.

**Výpis kódu 5.20 Separace výrazů pro validaci.**

```
private void validateText(String code) {
    StringBuilder token = new StringBuilder(100);
    //proměnná minorOffset udržuje pozici znaku v kontextu
    //zpracovávaného BeanShell skriptu
    for (minorOffset = 0; minorOffset < code.length(); minorOffset++) {
      char c = code.charAt(minorOffset);
      switch (c) {
      case '\n':
```

```
          token.append(c);
          //pokud je obsah rozpoznána jako komentář dojde k
          //jeho vyprázdnění a pokračuje se dalším znakem
          if (isComment(token.toString())) {
            token = new StringBuilder(100);
          }
          continue;
        case '}':
          //nalezen znak ukončující blok kódu
          --varDepth; //snížení hloubky zanoření
          //odstranění proměnných, které byly definovány
          //v opuštěném bloku a již neplatí
          manageVariables();
          //opuštěn cyklus do-while a proto je nutné provést
          //kontrolu pravdivostního výrazu
          if (doFlag) doExpected = true;
          continue;
        case '{':
          //nalezen znak otevírající blok kódu
          ++varDepth; //navýšení hloubky zanoření
          //pokud neodpovídá tvaru definice pole provede se validace
          //výrazu uvádějícího blok kódu
          if (!token.toString().matches(
              "\\s*\\w+\\s*\\[.*\\]\\s*[\\w\\d]+\\s*(=.*)?")) {
            validateBlockStart(token.toString(), varDepth);
            token = new StringBuilder(100);
          }
          continue;
        case ';':
          //nalezen znak ukončující výraz
          if (isIfOrCycle(token.toString())) {
            token.append(';'); //pokud nalezený výraz odpovídá
                //cyklu for doplni se nalezený znak a pokračuje se
                //ve čtení znaků skriptu
            continue;
          }
          //pokud je očekáván konec do-while bloku provede jeho validaci
          if (doExpected && checkWhileContent(token.toString())){
            continue;
          }
          //validace získaného výrazu
          validateToken(token.toString(), varDepth);
          token = new StringBuilder(100);
          continue;
        }
        token.append(c);
      }
    }
```

Dalším v řadě je znak `{`, který označuje start bloku kódu. V tomto případě dojde k navýšení hodnoty proměnné `varDepth`. Následně je porovnán řetězec znaků v proměnné `token` s regulárním výrazem, který reprezentuje definici pole. V případě, že je výsledek porovnání pozitivní, neprovádí se žádná validace do doby, než je nalezen znak `;`.

V případě negativního výsledku dojde k zavolání metody `validateBlockStart`, která pokračuje ve zpracování BeanShell skriptu. Uvedená metoda provede potřebné kroky pro validaci blokových výrazů, jako jsou například `if`, `while`, `for`, nebo `catch`. Po ukončení je proměnná token vyprázdněna a postupně opět plněna do doby, než je nalezen jeden ze čtyř výše uvedených znaků.

Posledním je pak znak `;`, který označuje konec výrazu. Prvním krokem je kontrola, zda nashromážděný řetězec znaků neodpovídá syntaxi `if` nebo některému cyklu. Blok `if` je z dříve uvedeného důvodu nutné zahrnout také, jelikož je potřeba předdefinované entity pro znaky `&`, < a > nahradit původními znaky (viz. tabulka 1).

Tyto znaky totiž nebyly předem odstraněny, protože by docházelo k potížím při přesném určení místa chyby. V případě, že řetězec znaků neodpovídá výsledku, je zavolána metoda `validateToken`.

**Výpis kódu 5.21 Určení typu výrazu pro validaci.**

```
private void validateToken(String token, int depth) {
  token = jbshTrim(token);
  if (token.equals(""))
    return;
  //Porovnání s regulárním výrazem pro import třídy
  // "\\s*import\\s+(\\w+\\.)+\\w+"
  if (isTypeDefinition(token)) {
    parseTypeDefinition(token);
  //Porovnání s regulárním výrazem pro definici proměnné
  //"\\s*([\\w\\.])*\\w+\\s*\\[?\\]?\\s+[\\w\\d]+\\s*(=.*)?"
  } else if (isVariableDefinition(token)) {
    parseVariableDefinition(token);
  //Porovnání s reg. výrazem pro části blokového výrazu switch
  //"\\s*((case\\s+[\\S&&[^\\:]]*)||(default))\\s*:.*"
  } else if (isPartOfSwitch(token)) {
  //Porovnání s reg výrazem pro vyvolání výjimky
  //"\\s*throw\\s+new\\s+[\\w]+\\(.*\\);?
  } else if (isExceptionThrow(token)) {
  } else {
    //neodpovídá žádnému z výše uvedených možností, je tedy
    //provedena validace výrazu
    validateExpression(token);
  }
}
```

Metoda `validateToken`, uvedená ve výpisu kódu 5.21 následně předaný řetězec porovná s regulárním výrazem, který odpovídá struktuře použití klíčového slova `import`.

V případě, že je porovnání úspěšné, dojde k přidání nového datového typu do množiny známých typů.

Další v řadě je porovnání řetězce s regulárním výrazem popisujícím definici proměnné. Pokud řetězec odpovídá, je přidána nová proměnná, viz kód 5.22. Zároveň dochází ke kontrole, zda je definovaný typ známý a zda proměnná se stejným jménem již existuje. V případě nalezení chyby je vytvořen objekt `ValidationError`, obsahující pozici chyby, a průvodní text popisující nalezenou chybu.

**Výpis kódu 5.22 Zpracování definice proměnné.**

```
private void parseVariableDefinition(String token) {
  //kontrola, zda výraz odpovídá poli
  if (token.matches("\\s*\\w+\\s*\\[.*\\]\\s*[\\w\\d]+\\s*(=.*)?")) {
    parseArrayDefinition(token);
    return;
  }
  //rozdělení na části, s rozdělovačem nastaveným na
  //libovolný počet bílých znaků
  String[] subTokens = token.split("\\s+");
  //získání typu proměnné
  String dType = subTokens[0];

  //pokud typ obsahuje tečky, je za název považován
  //řetězec od poslední tečky do prava
  int i;
  if ((i = dType.indexOf('.')) != -1)
    dType = dType.substring(0, i);
  //pokud nebyl daný datový typ nalezen mezi známými
  //typy dojde k vytvoření a uložení chyby
  if (!dataTypesSet.contains(dType)) {
    errorFound("Unknown data type: " + dType, token.length());
  }
  String var = subTokens[1];
  //pokud je obsaženo rovnítko jsou z řetězce použity pouze znaky
  //vlevo od rovnítka
  if(var.contains("=")) var= var.substring(0, var.indexOf('='));

  //vytvoření nové proměnné
  Variable p = new Variable();
  p.name = var;
  p.access = RW;
  p.depth = varDepth;
  if (this.variablesMap.containsKey(var)) {
    Variable v = this.variablesMap.get(var);
    //upravení proměnné definované v XML elementu variable
    if (v.depth == 0) {
      v.depth = p.depth;
    } else {
      //nalezena již jednou definované proměnná
      errorFound("Variable already defined", token.length());
```

```
        }
      } else {
        this.variablesMap.put(p.name, p);
      }
      if (token.contains("=")) {
        validateRightSide(token.substring(token.indexOf('=') + 1));
      }
    }
```

V případě, že je proměnné přiřazena hodnota, dojde ke kontrole výrazu na pravé straně. To je provedeno metodou `validateRightSide`, která bude blíže specifikována v následujícím odstavci. Pokud řetězec neodpovídal žádné z uvedených možností, je zavolána metoda `validateExpression`, která v případě, že výraz obsahuje znak =, který není součásti literálu typu `String`, provede kontrolu existence proměnné na levé straně. Na pravou stranu, případně na celý řetězec, pokud znak = obsažen nebyl, je zavolána metoda `validateRightSide`, viz kód 5.23.

**Výpis kódu 5.23 Validace pravé strany výrazu.**

```
private void validateRightSide(String token) {
  int length = token.length();
  //regulární výraz detekující přetypování hodnoty
  if (token.matches("^\\s*\\(\\s*[\\w\\.]+\\s*\\).*")) {
    String type = token.substring(
        token.indexOf('(') + 1, token.indexOf(')'));
    //kontrola existence datového typu použitého pro přetypování
    if (!this.dataTypesSet.contains(type)) {
      errorFound("Unknown data type " + type, length);
    }
    //odstranění řetězce přetypování
    token = token.substring(token.indexOf(')') + 1);
  }
  //kontrola zda řetězec odpovídá předdefinovaným konstantám
  //jazyka Java
  if (isJavaConstant(token)) {
    return;
  }
  //kontrola zda řetězec odpovídá regulárnímu výrazu pro novou instanci
  //"^\\s*new\\s+.*"
  if (isNewInstance(token)) {
    //odstranění klíč slova new
    token = token.replaceFirst("^\\s*new\\s+", "");
    try {
      int openingBracket = token.indexOf('(');
      //získání názvu pro datový typ
      String type = token.substring(0, openingBracket);

      if (!dataTypesSet.contains(type)) {
        errorFound("Unknown data type: " + type, length);
      }
```

```
          return;
        //zachycení výjimky kdy mezi klíč. slovem new a závorkou (
        //nebylo specifikováno jméno třídy
        } catch (Exception e) {
          //nalezen neočekávaný řetězec pro definici pole
          if (token.trim().matches("\\{(\".*\"\\s*,\\s*)+\".*\"}\\s*")) {
            errorFound(
            "UNABLE TO VALIDATE! Array definition occured: " + token,
            length);
            return;
          }
        }
      }
      int i;
      if ((i = token.indexOf('.')) == -1) {
        //odstranění hranatých závorek u výrazů, kde je použito pole
        String var = token.replaceAll("\\[.*\\]", "").trim();
        //odstranění znaků inkrementace
        if (var.contains("++")) {
          var = var.replaceFirst("\\+\\+", "").trim();
        } else if (var.contains("--")) {
          var = var.replaceFirst("\\-\\-", "").trim();
        }
        //kontrola zda získaný řetězec odpovídá nějaké
        //definované proměnné
        if (!variablesMap.containsKey(var))
          errorFound("Unrecognized expression: " + var, length);
      } else {
        //výraz obsahuje tečku, jde tedy o volání metody nad třídou
        //nebo proměnnou
        String var = token.substring(0, i).trim();
        if (!variablesMap.containsKey(var) && !dataTypesSet.contains(var)) {
          errorFound("Unrecognized token: " + var, length);
        }
        return;
      }
    }
```

Tato metoda předaný řetězec porovná s regulárním výrazem, který reprezentuje přetypování. Provede kontrolu, zda je daný datový typ známý, a řetězec přetypování odstraní. Dále provede porovnání zbylého textu se známým konstantami a literály jazyka Java. V případě, že porovnání proběhlo úspěšně, dojde k ukončení běhu metody.

Další je pak kontrola definice nové instance, která v případě úspěšného testu provede odstranění klíčového slova `new` a porovná, zda je typ vytvářené instance známý a ukončí běh metody. V případě, že ani jedna z možností nebyla úspěšná a řetězec neobsahuje znak `.`, jsou z textu odstraněny řetězce ++ a – a text je porovnán se známými proměnnými. Pokud nebyla nalezena shoda, je vytvořen objekt `ValidationError` s informací o neúspěšném rozeznání výrazu.

Poslední možností je případ, kdy textový řetězec tečky obsahuje. Část textu do první tečky je následně porovnána s množinami definovaných proměnných a definovaných datových typů. Pokud jsou obě porovnání neúspěšná, text se v množinách nevyskytuje, je opět vytvořen objekt `ValidationErrors` s informací o neúspěšné identifikaci výrazu.

# Kapitola 6.

# Testování

## 6.1. Test uživatelského rozhraní

Test má za cíl ověřit míru srozumitelnosti popisu a označení chyb, a dále schopnost vybrané množiny uživatelů provést instalaci pluginu. Pro testování byla zvolena metoda přímého testování s uživatelem, která poskytuje možnost interakce přímo s cílovou skupinou.

Dalšími možnostmi pak byly Heuristická evaluace, která v tomto případě není nejlepší volbou, jelikož nezíská informace přímo od cílových uživatelů. Metoda je založena na otestování kvality návrhu grafického rozhraní odborníky. Uživatelské rozhraní je však z naprosté většiny součástí API Eclipse IDE a nedá se příliš ovlivnit.

Další v řadě je pak Kognitivní průchod, který by bylo možné použít, avšak tato metoda vyžaduje poměrně přesně identifikovat slabá místa v návrhu. Metoda testuje návrh pomocí série úkolů, kdy je každý úkol rozdělen do kroků. V každém kroku pak tester musí odpovědět na tři otázky:

1. Je správná akce pro uživatele zjevná?

2. Dokáže uživatel za pomoci popisků dosáhnout svého cíle?

3. Je zpětná vazba pro uživatele dostatečná?

Proto je spíše vhodná pro fázi vývoje, kdy již byly problémy identifikovány, a pracuje se na jejich odstranění. Toto testování je opět prováděno odborníky, což s sebou přináší ještě jeden problém. Tímto problémem je případ, kdy odborník neodhadne správně potřeby cílové skupiny. Z důvodu možnosti přímé interakce s uživatelem, bylo proto zvoleno již zmiňované testování s uživatelem.

Prvním krokem byl výběr vhodných participantů. Pro výběr byl navržen screener, což je krátký dotazník se spíše uzavřenými otázkami navrženými tak, aby identifikovaly participanty s požadovanými vlastnostmi či schopnostmi, viz Příloha D. Požadovaný počet participantů byl stanoven na tři. Participant byl definován jako aktivní vývojář, s dlouhodobou zkušeností s vývojem v Eclipse IDE, jenž je zaměřen na implementaci a vývoj BP

Takto zvolený participant byl po příchodu do testovací místnosti krátce seznámen s průběhem testování a podstoupil předtestový dotazník. Dotazník se zaměřuje na získání informací o zvycích participanta v oblasti vývoje BP (viz. Příloha E). Časová dotace na provedení testu byla stanovena přibližně na půl hodiny.

Z důvodu časové vytíženosti participantů probíhalo testování v jimi zvoleném prostředí. Pro test byla uvolněna kancelář, což poskytlo dostatečné soukromí. V místnosti byl k dispozici stůl s širokoúhlým monitorem, notebookem a klávesnicí s myší. Participant byl uveden ke stolu a obdržel zadání testu (viz. F), který se skládal ze tří úkonů.

Po ukončení testování participant podstoupil potestový dotazník. Ten měl za úkol upřesnit informace získané v průběhu testu. Dalším cílem závěrečného dotazníku bylo zjistit ochotu cílové skupiny podstoupit změnu ve stylu vývoje výměnou za funkce poskytované pluginem. Celkové testování jednoho uživatele trvalo v průměru 27 minut.

Analýzou získaných dat, která lze nalézt v příloze H, byly zjištěny nedostatky v návrhu uživatelského rozhraní. Nedostatečné prpojení informací o nalezených chybách, které jsou zobrazeny v okně editoru, a seznamu chyb pod oknem editoru v konzoli vedlo ke zmatení účastníka. Další vytýkanou vlastností bylo nelogické řazení výpisu chyb a nepřesné označování chyb v syntaxi skriptů BeanShell. Nalezené chyby budou odstraněny v dalších verzích pluginu.

## 6.2. Test kompatibility

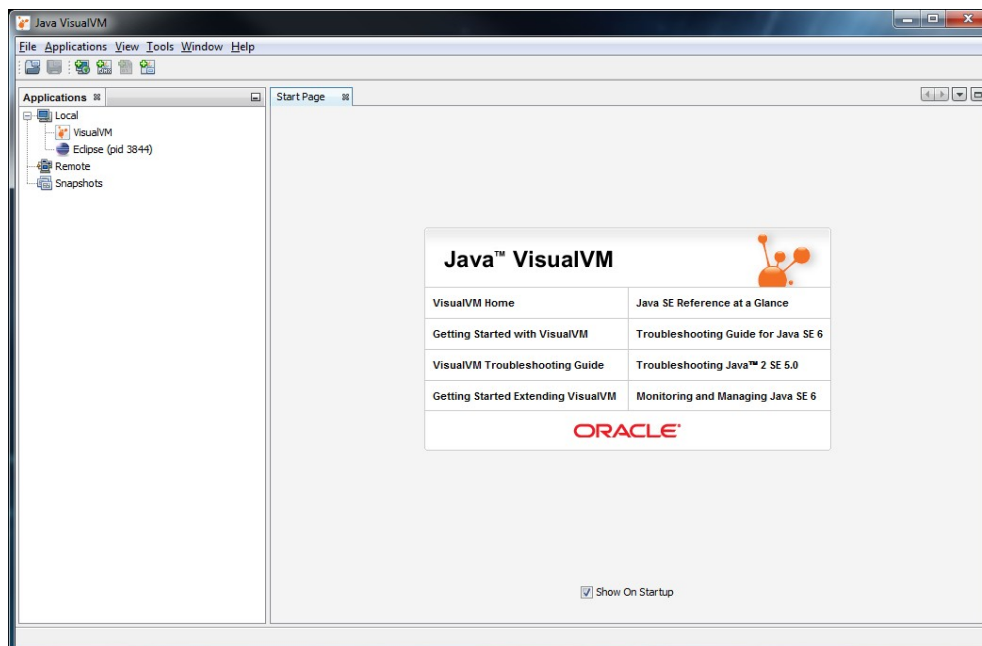
Cílem testu bylo potvrdit bezproblémovou instalaci a použití pluginu na množině zvolených operačních systémů. Jako testovací platformy byly zvoleny operační systémy Windows 7 Professional 64-bit a Linux. V rámci Linux se pak jednalo o distribuce Ubuntu 12.04 LTS, Centos 6.5 x86_64 a OpenSuse 12.1. Všechny testovací platformy byly dále připraveny v konfiguraci Oracle JDK 1.7, s připraveným Eclipse IDE for Java EE Developers. Na takto připravených platformách byl následně proveden test, rozdělený do dvou kroků.

Prvním krokem byla instalace pluginu. Ta probíhala dle postupu uvedeného v příloze B a proběhla zcela bez problémů na všech uvedených OS. Druhý krok testu, použití pluginu, proběhl až na distribuci OpenSuse 12.1 také bez problému.

Problémem, jenž se v Eclipse IDE v OpenSuse vyskytl, byla neschopnost IDE plugin načíst. Důvodem této nekompatibility byl startovací skript uvedeného IDE, který pro spuštění využíval JVM ve verzi 1.6. Návod k odstranění tohoto problému lze nalézt v příloze B.3.

## 6.3. Test výkonu

Testovací sestavou pro tento test byl notebook v hardwarové konfiguraci Intel i5 2.5GHz, 6GB RAM a s 2GB swapovacího prostoru. Pracovní prostředí instalované na uvedeném stroji bylo Linux OpenSUSE 12.1 x86(pae) a JDK 1.7.0_u22. Pro test samotný byl



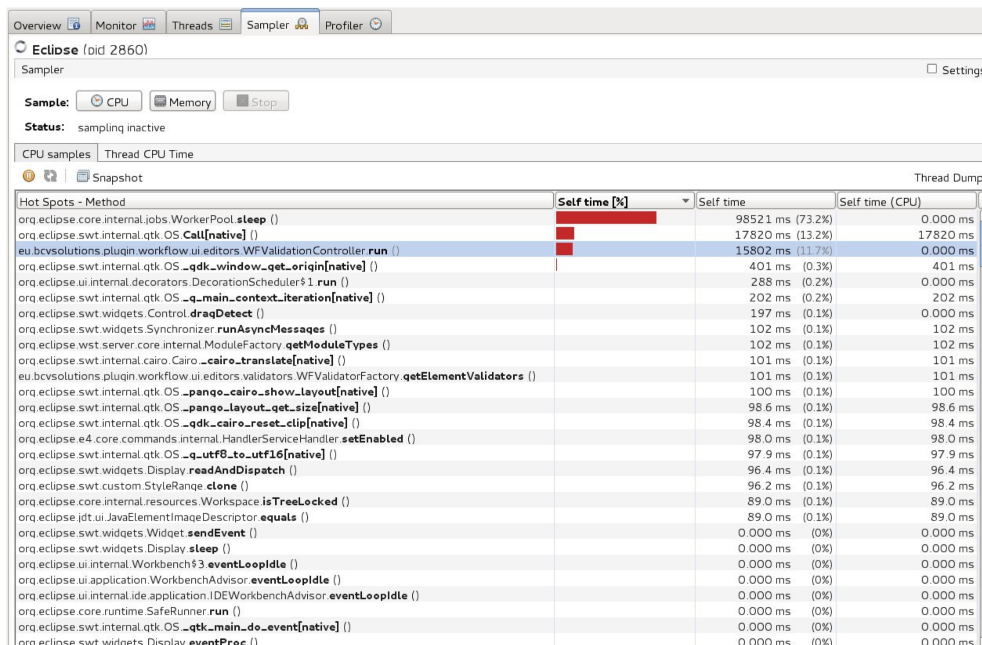

**Obrázek 5.** Testovací nástroj určený pro vizualizaci stavu Java VM

použit nástroj jvisualvm (viz. obrázek 5), jenž je standardní součástí JDK.

Cílem tohoto testu bylo určit nároky projektu na hardware uživatele, konkrétně požadavky na výkon procesoru a množství operační paměti. Analýzou získaných hodnot bylo zjištěno, že projekt má zcela zanedbatelné hardwarové požadavky. Minimální požadavky pro použití pluginu jsou tedy shodné s požadavky Eclipse IDE.

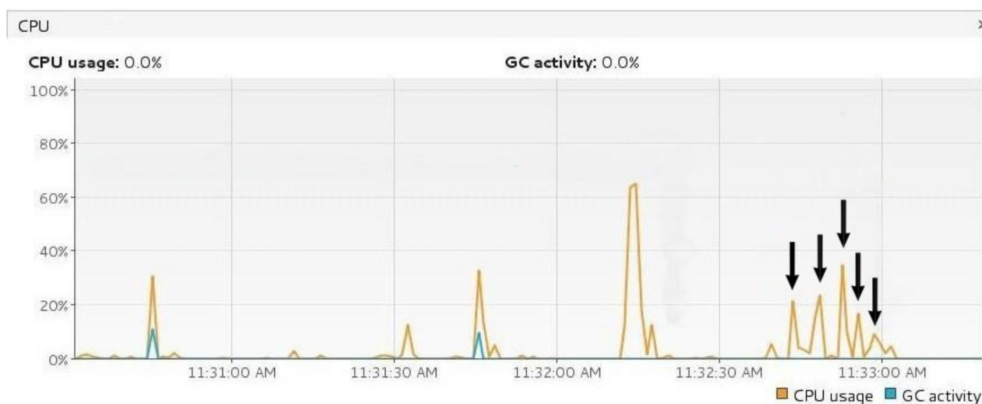
Z následujících informací na obrázcích 6 a 9 lze vyčíst, že největší nároky pluginu na výpočetní čas procesoru pochází od třídy `WFValidationController`, což je očekávaný výsledek vzhledem k tomu, že řídí celou logiku validace. Největší požadavky na paměť pak pochází od parseru XML, kde se jedná o třídy `WFPosition`, `WFElement` a `WFAttribute`, které udržují informace o původním dokumentu, a třídy knihovny BeanShellu, konkrétně třídy `bsh.Node` a `bsh.BSHAmbiguousName`, které pracují s velkým množstvím textových dat.

Na obrázku 7 jsou zvýrazněny nároky pluginu při spuštění validace. Šipky, které zvýrazňují výkonové špičky v grafu, odpovídají krátkodobě zvýšeným nárokům při vyvolání validace dokumentu Workflow. Při porovnání časové linie z obrázku 7 s obrázkem 8 je možné vidět, že spuštění validace neznamená znatelné výkyvy ani pro využití paměti Eclipse IDE. Uvedené obrázky je možno nalézt ve větším formátu v příloze I.




| Hot Spots - Method | Self time [%] | Self time | Self time (CPU) |
|---|---|---|---|
| org.eclipse.core.internal.jobs.WorkerPool.**sleep** () | | 98521 ms (73.2%) | 0.000 ms |
| org.eclipse.swt.internal.gtk.OS.**Call[native]** () | | 17820 ms (13.2%) | 17820 ms |
| eu.bcvsolutions.plugin.workflow.ui.editors.WFValidationController.**run** () | | 15802 ms (11.7%) | 0.000 ms |
| org.eclipse.swt.internal.gtk.OS.**_gdk_window_get_origin[native]** () | | 401 ms (0.3%) | 401 ms |
| org.eclipse.ui.internal.decorators.DecorationScheduler$1.**run** () | | 288 ms (0.2%) | 0.000 ms |
| org.eclipse.swt.internal.gtk.OS.**_g_main_context_iteration[native]** () | | 202 ms (0.2%) | 202 ms |
| org.eclipse.swt.widgets.Control.**dragDetect** () | | 197 ms (0.1%) | 0.000 ms |
| org.eclipse.swt.widgets.Synchronizer.**runAsyncMessages** () | | 102 ms (0.1%) | 102 ms |
| org.eclipse.wst.server.core.internal.ModuleFactory.**getModuleTypes** () | | 102 ms (0.1%) | 102 ms |
| org.eclipse.swt.internal.cairo.Cairo.**_cairo_translate[native]** () | | 101 ms (0.1%) | 101 ms |
| eu.bcvsolutions.plugin.workflow.ui.editors.validators.WFValidatorFactory.**getElementValidators** () | | 101 ms (0.1%) | 101 ms |
| org.eclipse.swt.internal.gtk.OS.**_pango_cairo_show_layout[native]** () | | 100 ms (0.1%) | 100 ms |
| org.eclipse.swt.internal.gtk.OS.**_pango_layout_get_size[native]** () | | 98.6 ms (0.1%) | 98.6 ms |
| org.eclipse.swt.internal.gtk.OS.**_gdk_cairo_reset_clip[native]** () | | 98.4 ms (0.1%) | 98.4 ms |
| org.eclipse.e4.core.commands.internal.HandlerServiceHandler.**setEnabled** () | | 98.0 ms (0.1%) | 98.0 ms |
| org.eclipse.swt.internal.gtk.OS.**_g_utf8_to_utf16[native]** () | | 97.9 ms (0.1%) | 97.9 ms |
| org.eclipse.swt.widgets.Display.**readAndDispatch** () | | 96.4 ms (0.1%) | 96.4 ms |
| org.eclipse.swt.custom.StyleRange.**clone** () | | 96.2 ms (0.1%) | 96.2 ms |
| org.eclipse.core.internal.resources.Workspace.**isTreeLocked** () | | 89.0 ms (0.1%) | 89.0 ms |
| org.eclipse.jdt.ui.JavaElementImageDescriptor.**equals** () | | 89.0 ms (0.1%) | 89.0 ms |
| org.eclipse.swt.widgets.Widget.**sendEvent** () | | 0.000 ms (0%) | 0.000 ms |
| org.eclipse.swt.widgets.Display.**sleep** () | | 0.000 ms (0%) | 0.000 ms |
| org.eclipse.ui.internal.Workbench$3.**eventLoopIdle** () | | 0.000 ms (0%) | 0.000 ms |
| org.eclipse.ui.application.WorkbenchAdvisor.**eventLoopIdle** () | | 0.000 ms (0%) | 0.000 ms |
| org.eclipse.ui.internal.ide.application.IDEWorkbenchAdvisor.**eventLoopIdle** () | | 0.000 ms (0%) | 0.000 ms |
| org.eclipse.core.runtime.SafeRunner.**run** () | | 0.000 ms (0%) | 0.000 ms |
| org.eclipse.swt.internal.gtk.OS.**_gtk_main_do_event[native]** () | | 0.000 ms (0%) | 0.000 ms |
| org.eclipse.swt.widgets.Display.**eventProc** () | | 0.000 ms (0%) | 0.000 ms |

**Obrázek 6.** Procentuální využití procesorového času jednotlivými vlákny




**Obrázek 7.** Graf využití procesoru Java VM

**Obrázek 8.** Graf využití paměti Java VM. Šipky označují nároky při spuštění validace obsahu.

Overview | Monitor | Threads | Sampler | Profiler

Eclipse (pid 2860)

Sampler | Settings

**Sample:** CPU | Memory | Stop

**Status:** sampling inactive

Heap histogram | PermGen histogram | Per thread allocations

Deltas | Snapshot | Peform GC | Heap Dump

**Classes:** 7,224 **Instances:** 4,423,284 **Bytes:** 181,786,640

| Class Name | Bytes | | Instances | |
|---|---|---|---|---|
| bsh.**BSHAmbiguousName** | 38,800 | (0.0%) | 970 | (0.0%) |
| bsh.**BSHArguments** | 10,880 | (0.0%) | 340 | (0.0%) |
| bsh.**BSHArrayInitializer** | 192 | (0.0%) | 6 | (0.0%) |
| bsh.**BSHAssignment** | 2,000 | (0.0%) | 50 | (0.0%) |
| bsh.**BSHBinaryExpression** | 2,320 | (0.0%) | 58 | (0.0%) |
| bsh.**BSHBlock** | 3,760 | (0.0%) | 94 | (0.0%) |
| bsh.**BSHCastExpression** | 192 | (0.0%) | 6 | (0.0%) |
| bsh.**BSHForStatement** | 1,120 | (0.0%) | 20 | (0.0%) |
| bsh.**BSHFormalParameter** | 240 | (0.0%) | 6 | (0.0%) |
| bsh.**BSHIfStatement** | 1,536 | (0.0%) | 48 | (0.0%) |
| bsh.**BSHImportDeclaration** | 7,680 | (0.0%) | 192 | (0.0%) |
| bsh.**BSHLiteral** | 11,840 | (0.0%) | 296 | (0.0%) |
| bsh.**BSHMethodInvocation** | 8,576 | (0.0%) | 268 | (0.0%) |
| bsh.**BSHPrimaryExpression** | 28,096 | (0.0%) | 878 | (0.0%) |
| bsh.**BSHPrimarySuffix** | 1,824 | (0.0%) | 38 | (0.0%) |
| bsh.**BSHPrimitiveType** | 2,320 | (0.0%) | 58 | (0.0%) |
| bsh.**BSHReturnStatement** | 160 | (0.0%) | 4 | (0.0%) |
| bsh.**BSHStatementExpressionList** | 640 | (0.0%) | 20 | (0.0%) |
| bsh.**BSHTryStatement** | 128 | (0.0%) | 4 | (0.0%) |
| bsh.**BSHType** | 12,288 | (0.0%) | 256 | (0.0%) |
| bsh.**BSHTypedVariableDeclaration** | 9,760 | (0.0%) | 244 | (0.0%) |
| bsh.**BSHUnaryExpression** | 1,040 | (0.0%) | 26 | (0.0%) |
| bsh.**BSHVariableDeclarator** | 9,760 | (0.0%) | 244 | (0.0%) |
| bsh.**JJTParserState** | 1,216 | (0.0%) | 38 | (0.0%) |
| bsh.**JavaCharStream** | 2,736 | (0.0%) | 38 | (0.0%) |
| bsh.**Node[]** | 51,152 | (0.0%) | 2,742 | (0.0%) |
| eu.bcvsolutions.plugin.workflow.tagprovider.**TagRule** | 48 | (0.0%) | 1 | (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.**Activator** | 48 | (0.0%) | 1 | (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.editors.**IWFError$Type** | 48 | (0.0%) | 3 | (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.editors.**IWFError$Type[]** | 24 | (0.0%) | 1 | (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.editors.**WFDocumentProvider** | 40 | (0.0%) | 1 | (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.editors.**WFEditor** | 432 | (0.0%) | 1 | (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.editors.**WFObject** | 48 | (0.0%) | 2 | (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.editors.**WFObjectBuilder** | 32 | (0.0%) | 2 | (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.editors.**WFPartitionScanner** | 72 | (0.0%) | 1 | (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.editors.**WFPosition** | 49,032 | (0.0%) | 2,043 | (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.editors.**WFSrcViewConf** | 32 | (0.0%) | 1 | (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.editors.**WFValidationController** | 136 | (0.0%) | 1 | (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.editors.**WhitespaceDetector** | 32 | (0.0%) | 4 | (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.editors.colors.**ColorManager** | 16 | (0.0%) | 1 | (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.editors.parsers.**WFNode$Type** | 112 | (0.0%) | 7 | (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.editors.parsers.**WFNode$Type[]** | 40 | (0.0%) | 1 | (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.editors.parsers.**WFXMLAttribute** | 20,032 | (0.0%) | 626 | (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.editors.parsers.**WFXMLComment** | 48 | (0.0%) | 2 | (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.editors.parsers.**WFXMLElement** | 21,728 | (0.0%) | 388 | (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.editors.parsers.**WFXMLInstruction** | 32 | (0.0%) | 2 | (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.editors.parsers.**WFXMLParser** | 8 | (0.0%) | 1 | (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.editors.parsers.**WFXMLParser$Parser** | 64 | (0.0%) | 2 | (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.editors.parsers.**WFXMLRootNode** | 80 | (0.0%) | 2 | (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.editors.parsers.**WFXMLText** | 1,728 | (0.0%) | 54 | (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.editors.validators.**BeanShellValidator** | 16 | (0.0%) | 1 | (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.editors.validators.**GraphValidator** | 16 | (0.0%) | 1 | (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.editors.validators.**ValidationError** | 336 | (0.0%) | 14 | (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.editors.validators.**VariableValidator** | 40 | (0.0%) | 1 | (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.editors.validators.**VariableValidator$PairBuilder** | 3,312 | (0.0%) | 138 | (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.editors.validators.**VariableValidator$Variable** | 8,832 | (0.0%) | 368 | (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.editors.validators.**WFValidatorFactory** | 24 | (0.0%) | 1 | (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.editors.validators.graph.**GraphBuilder** | 64 | (0.0%) | 2 | (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.editors.validators.graph.**GraphBuilder$BuilderEndNode** | 64 | (0.0%) | 2 | (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.editors.validators.graph.**GraphBuilder$BuilderNode** | 960 | (0.0%) | 40 | (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.editors.validators.graph.**GraphBuilder$BuilderStartNode** | 64 | (0.0%) | 2 | (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.editors.validators.graph.**GraphBuilder$Transition** | 1,872 | (0.0%) | 78 | (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.editors.validators.graph.**GraphNode** | 5,424 | (0.0%) | 226 | (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.textscanners.**ITokenDescriptors$Descriptors** | 144 | (0.0%) | 6 | (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.textscanners.**ITokenDescriptors$Descriptors[]** | 240 | (0.0%) | 6 | (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.textscanners.**JavaCodeScanner** | 40 | (0.0%) | 1 | (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.textscanners.**JavaCodeScanner$JavaWordDetector** | 16 | (0.0%) | 1 | (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.textscanners.**StringScanner** | 40 | (0.0%) | 1 | (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.textscanners.**XMLCommentScanner** | 40 | (0.0%) | 1 | (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.textscanners.**XMLInstructionScanner** | 40 | (0.0%) | 1 | (0.0%) |

**Obrázek 9.** Procentuální využití paměti jednotlivými třídami

# Kapitola 7.

# Závěr

## 7.1. Závěr

Cílem této bakalářské práce bylo vytvořit nástroj, jenž usnadní vývoj a implementaci workflow business procesů. Na základě analýzy problému a stanovených potřeb vývojářů byl navržen a realizován plugin pro vývojové prostředí Eclipse. Výsledek je možné vidět na obrázku 10. Plugin usnadňuje orientaci ve vyvíjeném business procesu zvýrazněním syntaxe jazyka Java a zvýrazněním chyb, čímž fakticky zkracuje dobu nutnou pro proces debugování a pro následné opravy.

Vyvinutý plugin je schopný validace použití deklarovaných proměnných, definovaných datových typů na základě jejich importů, základní syntaktické kontroly skriptovacího jazyka BeanShell a validace logické struktury business procesu popsaného ve Workflow.

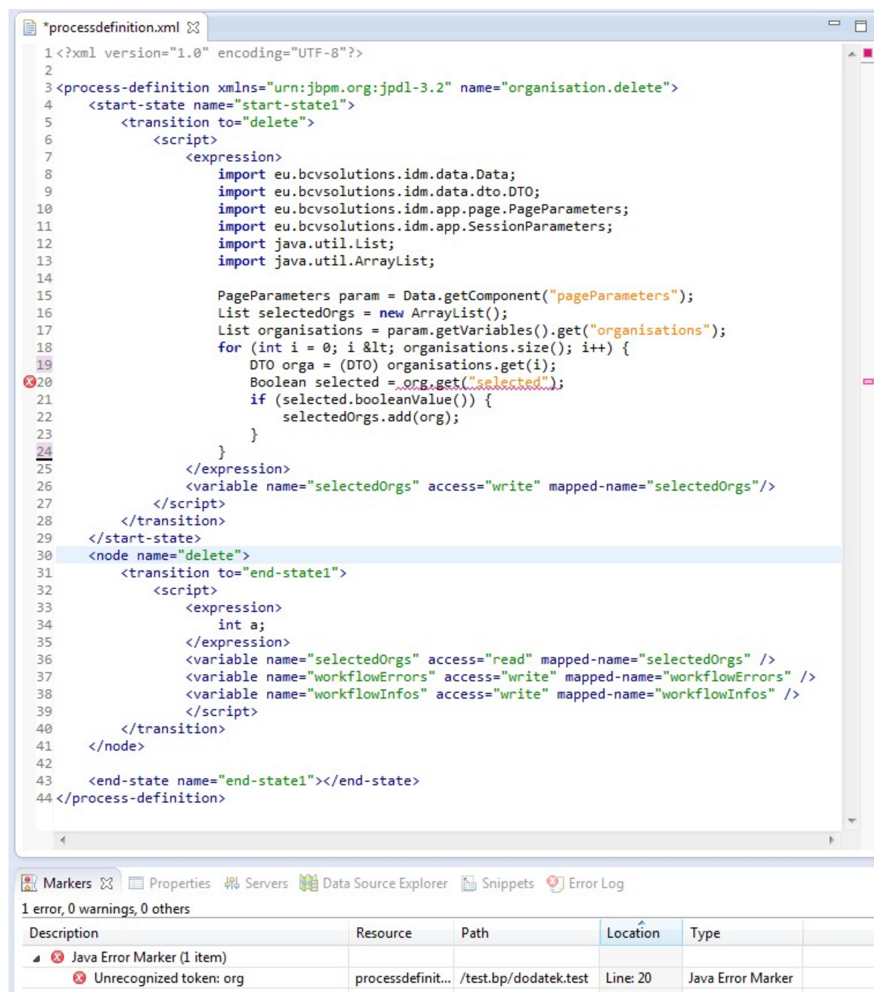
V průběhu testování byly zjištěny nedostatky uživatelského rozhraní v propojení chyb zobrazených v seznamu a chyb zaznamenaných uvnitř editoru, které budou v průběhu dalšího vývoje v následujících verzích odstraněny. Přes uvedené nedostatky je plugin již aktivně využíván při vývoji Workflow a ukázal se být velkým přínosem a efektivním nástrojem.

## 7.2. Další vývoj projektu

Stávající funkčnosti pluginu budou do budoucna dále rozšiřovány, a to ve třech etapách.

V první etapě dojde k vylepšení zpětné vazby editoru poskytované vývojáři, zlepšením propojení nástrojů pro označení nalezené chyby. Dále vylepšení XML parseru přesnější detekcí chyb ve struktuře XML a jejich následném označení v editoru, kdy v případě, že chybí uzavírací element, dojde k falešné detekci i u nadřazených elementů, které uzavřeny jsou. Posledním krokem etapy bude přidání kontroly existence importovaných tříd, což znamená validaci tzv. classpath, a vylepšení validace o znalost struktury těchto importovaných tříd.

V druhé etapě bude doplněna typová kontrola při práci s proměnnými a dojde k omezení validace pouze na oblasti textu v dokumentu, kde došlo k přímé změně. Tím dojde ke zrychlení celého procesu validace a zpětná interakce editoru bude více plynulá.




**Obrázek 10.** Dokument Workflow zobrazený pluginem.

V poslední etapě pak bude plugin obohacen o interaktivní nápovědu při najetí myší nad metodu či třídu a o dynamický asistent obsahu. Asistent dále zrychlí vývoj Workflow a usnadní vývojáři práci, tím že sníží nároky na množství informací, kterými musí vývojář nezbytně disponovat. V případě změn v používaných knihovnách tak bude možné získat popis interface třídy přímo z editoru.

# Příloha A.

# Obsah přiloženého CD

- **bin** -obsahuje .jar verzi pluginu
- **projekt_Eclipse_IDE** -obsahuje celé adresáře projektů souvisejících s pluginem
  - eu.bcvsolutions.plugin.workflow.feature -sdružuje skupinu pluginů
  - eu.bcvsolutions.plugin.workflow.ui - projekt se samotným pluginem
  - eu.bcvsolutions.plugin.workflow.updatesite -update site slouží k zveřejnění feature
- **projekt_TeXnicCenter** -kompletní projekt editoru LaTeX souborů
- **sada_testovacich_workflow** -sada vzorových Workflow s již vytvořenými chybami
- **src** -zdrojové soubory pluginu
- **src_text** -zdrojové soubory textu
- **cernylu5_2014bach.pdf** -pdf verze textu bakalářské práce

# Příloha B.

# Instalace

Instalaci lze provést dvěma metodami, které jsou uvedené níže. Doporučuji k instalaci využít první metody, Update site, jelikož poskytuje v průběhu instalace zpětnou vazbu a následně usnadňuje a automatizuje aktualizace pluginů.

## B.1. Update site

Metoda update site umožňuje snadnou správu a následnou údržbu instalovaných pluginů.

1. V rolovacím menu zvolit `Help`→ `Install new software...`
2. Stisknutím na tlačítko `Add` otevřte okno pro přidání repozitáře.
3. Do kolonky `Name:` vložte libovolné jméno, např. `workflow-plugin`.
4. Do kolonky `Location:` vložte url `http://lubos.superhosting.cz/plugin`.
5. Stisknutím `OK` dojde k uložení repozitáře.
6. Nyní přidaný repozitář vyberte v select menu s popiskem `Work with:` a zobrazený plugin zaškrtněte.
7. Restartujte IDE
8. Plugin je možné okamžitě používat. Pokud byly před restartem některé dokumenty `processdefinition.xml` v editoru otevřeny, bude nutné je zavřít a znovu otevřít.

## B.2. Dropins

Vložením souboru jar obsahující plugin do adresáře v souborové struktuře vývojového prostředí Eclipse IDE dojde k jeho přidání.

1. Stáhnout soubor `.jar` pluginu z url:
   `http://lubos.superhosting.cz/plugin-jar`
2. Otevřít adresář **`*cesta/k/adresari/eclipse*`**`/dropins`.
3. Přesunout do adresáře stažený `.jar` soubor s pluginem.

4. V případě, že bylo prostředí Eclipse IDE otevřeno je nutné ho restartovat.
5. Plugin je možné okamžitě používat. Pokud byly před restartem některé dokumenty `processdefinition.xml` v editoru otevřeny, bude nutné je zavřít a znovu otevřít.

Další informace k možnostem instalace pluginu pomocí adresáře dropins lze nalézt ve zdroji [16].

## B.3. Známé problémy při instalaci

### Plugin se po instalaci nezobrazí

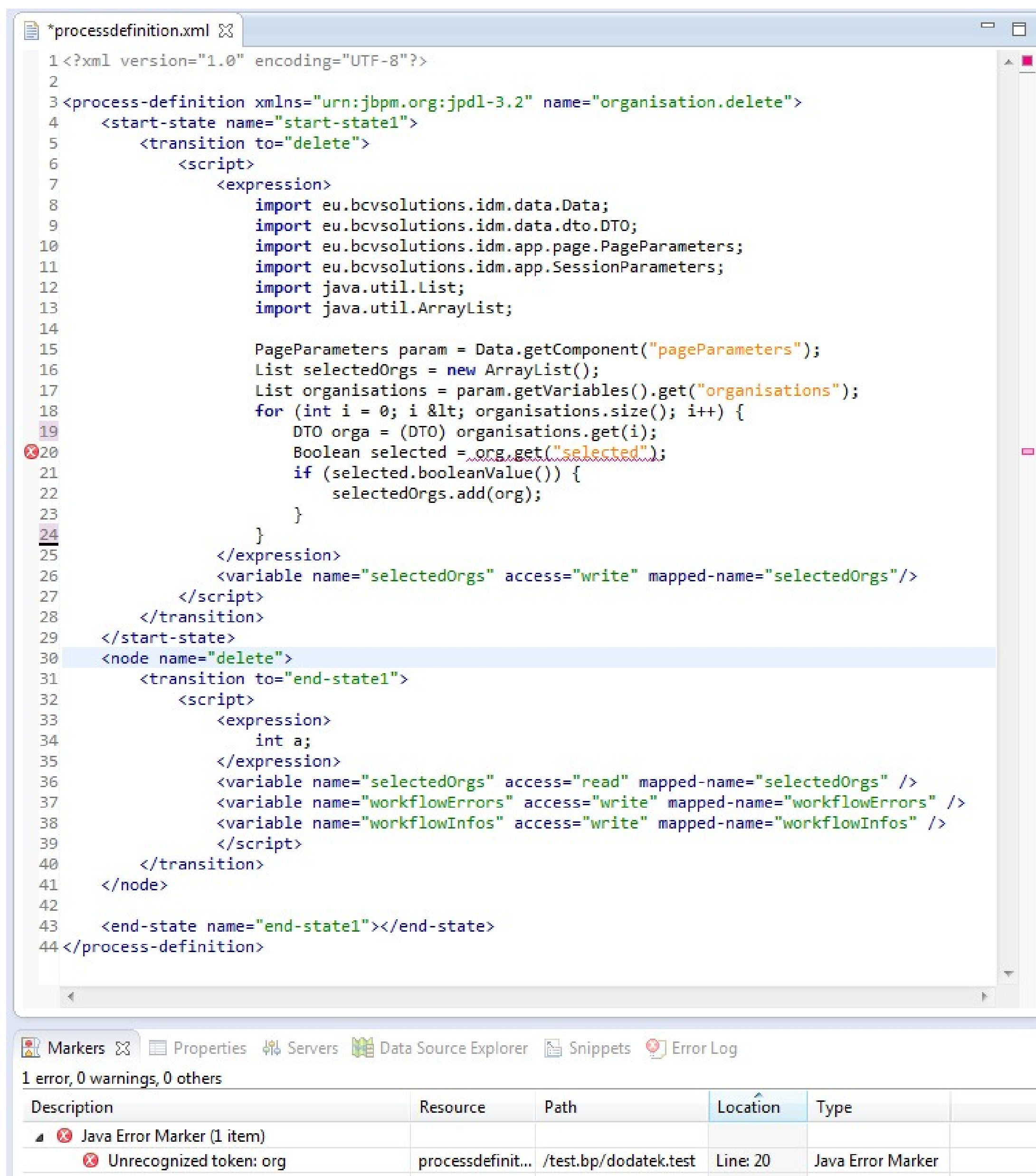
Tento problém je způsoben nastavením systému, kdy je Eclipse IDE spouštěno virtuálním strojem JVM verze 1.6.x a nižší. Tento problém je možné obejít přidáním parametru `-vm` **`cesta-k-JDK`** při spuštění Eclipse IDE.

# Příloha C.

# Použití

Pro použití pluginu není po instalaci potřeba v Eclipse IDE provádět žádné změny nastavení. Plugin se automaticky aktivuje pouze pro soubory jejichž název je `processdefinition.xml`.

Jedinou podmínkou pro správnou funkčnost je, že soubory musí být otevřeny z prostředí Eclipse IDE a jejich umístění musí být v existujícím projektu v aktuálním použitém `Workspace`. Důsledkem této podmínky je, že v případě přetažení souboru do okna Eclipse IDE nedojde k jeho otevření, ale tento proces skončí chybou. V případě, že vše proběhlo v pořádku, bude po otevření souboru s dokumentem Workflow vypadat vývojové prostředí jako na následujícím obrázku 11.




**Obrázek 11.** Zobrazení dokumentu processdefinition.xml po úspěšné instalaci.

# Příloha D.

# Screener

Čísla v závorkách udávají minimální požadovaný počet participantů.

## 1) Vyberte oblast specializace v oboru IT:

- ☐ Programátor (7)
- ☐ Manager (0)
- ☐ Analytik (0)
- ☐ Mimo IT (0)
- ☐ Jiná (0)

## 2) Zvolte vývojová prostředí, s nimiž pravidelně pracujete více než jeden rok:

- ☐ NetBeans (0)
- ☐ Eclipse IDE (7)
- ☐ Microsoft (0)
- ☐ Jiné (0)

## 3) Kolik hodin týdně věnujete vývoji:

- ☐ 0 - 5 hodin týdně (0)
- ☐ 5 - 10 hodin týdně (0)
- ☐ 10 - 20 hodin týdně (2)
- ☐ Více než dvacet hodin (5)

## 4) Vývoji jakých aplikací se věnujete:

- ☐ Webové aplikace (2)
- ☐ Implementace business procesů (3)
- ☐ Enterprise aplikace (2)
- ☐ Desktopové aplikace (0)
- ☐ Mobilní aplikace (0)

# Příloha E.

# Předtestový dotazník

1. Používáte grafický designer na workflow? Jaký a proč?
2. Kolik hodin týdně strávíte tvorbou workflow?
3. Jak často se vám stává, že uděláte překlep (chybějící středník, závorka, apod.)?
4. Kontrolujete po sobě napsaný kód, nebo ho jen zkusíte zkompilovat/spustit a až pak opravujete?

# Příloha F.

# Scénář uživatelského testování

## Časový harmonogram

1. Příchod participanta, předtestový dotazník (5 minut)
2. Seznámení se zadáním testu, vypracování testu (20 minut)
3. Potestový dotazník (5 minut)

## Zadání hlavního testu

1. Z url *http://lubos.superhosting.cz/plugin* nainstalujte plugin pro editaci Workflow. Po dokončení instalace restartujte vývojové prostředí.
2. Do nového nebo již existujícího projektu přidejte nový soubor, který pojmenujte *processdefinition.xml*.
3. Do nově vytvořeného souboru zkopírujte šablonu Workflow a dle níže uvedeného vzoru doplňte kód BeanShell skriptu.

**Výpis kódu F.1 Příloha zadání testu**

```
<?xml version="1.0" encoding="UTF-8"?>

<process-definition xmlns="urn:jbpm.org:jpdl-3.2"
    name="organisation.delete">

<start-state name="start-state1">
  <transition to="delete">
    <script>
      <expression>
        import eu.bcvsolutions.idm.data.Data;
        import eu.bcvsolutions.idm.data.dto.DT;
        import eu.bcvsolutions.idm.app.page.PageParameters;
        import eu.bcvsolutions.idm.app.SessionParameters;

        PageParameters param = Data.getComponent("pageParameters");

        List selectedOrgs = new ArrayList();
```

```
        List organisations = params.getVariables().get("organisations");

        for (int i = 0; i &lt; organisations.size(); i++) {
          DTO org = (DTO) organisations.get(i);
          Boolean selected = org.get("selected"){
          if (selected.booleanValue()) {
            selectedOrgs.add(org);
          }
        }
      </expression>
      <variable name="selectedOrgs" access="write"
          mapped-name="selectedOrgs"/>
    </script>
  </transition>
</start-state>

<node name="delete">
  <transition to="end-state">
    <script>
      <expression>
        import eu.bcvsolutions.idm.data.Data;
        import eu.bcvsolutions.idm.data.dto.DTO;
        import java.util.List;
        import java.util.ArrayList
        import eu.bcvsolutions.idm.data.view.View;
        import eu.bcvsolutions.idm.app.workflow.WorkflowMessage;
        import eu.bcvsolutions.idm.data.Data;
        import java.util.List;

        List workflowErrors = new ArrayList();
        List workflowInfos = new ArrayList();

        boolean lastDeleteResults;

        for (int i = 0; i &lt; selectedOrgs.size(); i++) {
          DTO org  (DTO) selectedOrgs.get(i);

          try {
            lastDeleteResult = Data.delete(
                View.Type.ORGANISATION, org.getId());
          } catch (Exception e) {
            lastDeleteResult = false;
          }
          Object [] params = {org.get("name")};

          if (lastDeleteResult) {
            workflowInfos.add(
              new WorkflowMessage("app_organisation_delete_success",
                params));
          } else {
```

```
            workflowErrors.add(
              new WorkflowMessage("app_organisation_delete_fail",
                params));
          }
        }
      </expression>
      <variable name="selectedOrgs" access="read"
          mapped-name="selectedOrgs" />
      <variable name="workflowErrors" access="write"
          mapped-name="workflowErrors" />
      <variable name="workflowInfos" access="write"
          mapped-name="workflowInfos" />
    </script>
  </transition>
  </node>

<end-state name="end-state1"></end-state>

</process-definition>
```

# Příloha G.

# Potestový dotazník

1. Jaký je váš názor na workflow plugin?
2. Pokud byste používal jiné IDE (NetBeans, IntelliJ, ...), byl byste ochotný kvůli workflow pluginu přejít na Eclipse?
3. Jakou byste očekával úsporu času při vývoji, pokud byste plugin používal?
4. Zajímají Vás systémové nároky pluginu jako takového? Co říkáte na nutnost mít JDK1.7?

# Příloha H.

# Výsledky

## Participant 1

### Předtestový dotazník:

1. Nepoužívám, více mi vyhovuje textové vývojové prostředí.
2. Mezi 15 a 23 hodinami.
3. Poměrně běžně.
4. Většinou ho celý znovu neprocházím.

### Uživatelský test:

9:04 Začátek testu.
9:06 Part. opravil nedostupnost uzlu `end-state`.
9:09 Part. napsal `PateParameters` namísto `PageParameters`.
9:09 Překlep opraven.
9:10 Doplněn import datového typu `List` a `ArrayList`.
9:13 Participant špatně uzavřel literál typu `String`: `"selected'`
9:15 Chyba odhalena a opravena (participant měl problém s nalezením umístění chyby).
9:15 Opraveno chybné jméno proměnné `params` na `param`.
9:17 Opraven import `import eu.bcvsolutions.idm.data.dto.DT;`, doplněno chybějící písmeno.
9:17 Opravena syntaktická chyba. Znak { nahrazen znakem ;.
9:19 Opraven chybějící středník u importu typu.
9:19 Opraveny duplicitní importy datových typů.
9:20 Konec testu.

### Potestový dotazník:

1. Vývoj je podstatně pohodlnější. V případě zapomenuté závorky je poměrně obtížné dohledat označenou chybu.
2. Pokud bych nenalezl alternativy aktuálně používaných pluginů, tak bych o tom možná přemýšlel.

3. Asi někde mezi 20 a 30 procenty.
4. Pokud po nainstalování nepocítím výraznou změnu výkonu, pak spíš ne. JDK 1.7 již používám, takže v tom problém nevidím.

### Participant 2

#### Předtestový dotazník:

1. Používal jsem, ale teď již ne. Stejně bylo nutné kód napsat ručně.
2. Cca 2-3 dny v týdnu.
3. Skoro pořád.
4. Kontroluji tak napůl.

#### Hlavní test:

9:43 Začátek testu.
9:45 Opravena nedostupnost uzlu `end-state`.
9:46 Participant napsal chybně import
`eu.bcvsolutions.idm.app.page.PageParaneters`.
9:47 Doplněn import datového typu `List` a `ArrayList`.
9:47 Opraven překlep v importu.
9:48 Participant napsal místo závorky ), vzápětí opraveno.
9:50 Participant nenapsal středník.
9:51 Opraven chybějící středník. (Nalezení místa chyby trvalo trochu déle)
9:52 Opraveno chybné jméno proměnné `params` na `param`.
9:54 Upraven `import eu.bcvsolutions.idm.data.dto.DT;` doplněno chybějící písmeno, avšak chybně.
9:54 Opraven `import eu.bcvsolutions.idm.data.dto.DT;` doplněno správné písmeno.
9:55 Opravena syntaktická chyba. Znak { nahrazen znakem ;.
9:57 Opraven chybějící středník u importu typu. (detekce chyby trvala déle)
9:57 Opraveny duplicitní importy datových typů.
10:00 Doplněno rovnítko u definice proměnné `DTO org`
10:01 Upraven název definice proměnné `lastDeleteResults` na `lastDeleteResult`.
10:03 Konec testu.

#### Potestový dotazník:

1. Vypadá to hezky, ale chtělo by to našeptávač.
2. Nevím, nejspíše ne.
3. Cca pár hodin týdně.

4. Je mi to jedno.

### Participant 3

#### Předtestový dotazník:

1. Chvíli jsem je zkoušel, ale u všech mi přišla implementace pomalejší, než když jsem ji psal ručne.
2. Řekl bych víc než 25 hodin týdně.
3. Jak kdy, ale dalo by se to hodnotit jako poměrně často.
4. Kontrolovat se ho snažím, ale nemůžu říct, že bych to dělal pokaždé.

#### Hlavní test:

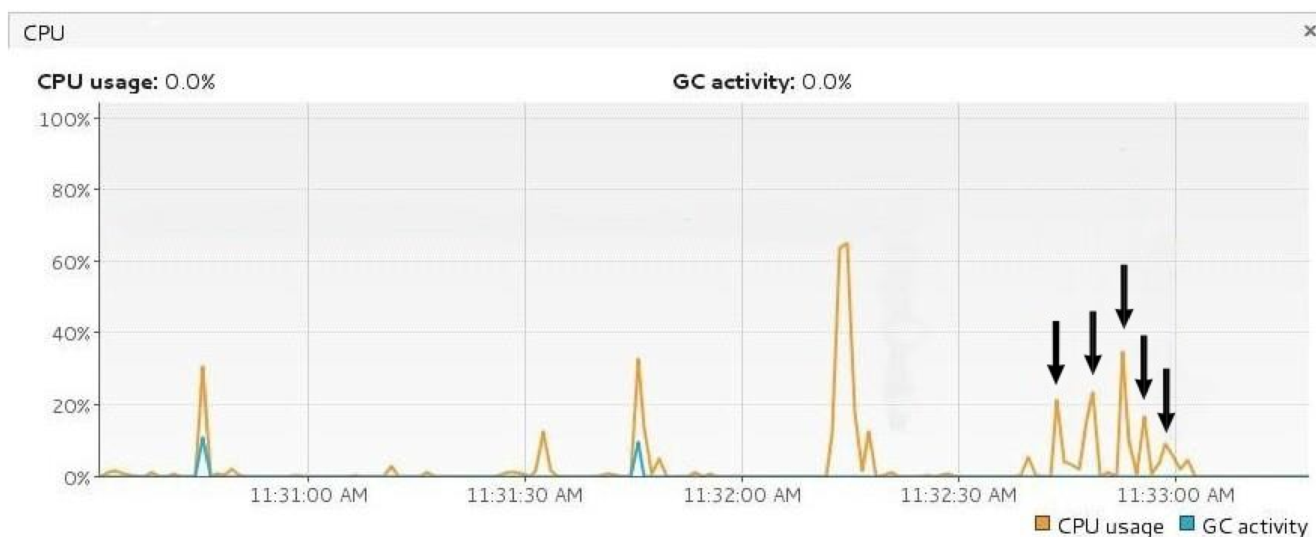
10:14 Začátek testu.
10:18 Doplněn import datového typu `List` a `ArrayList`.
10:19 Opravena nedostupnost uzlu `end-state`.
10:21 Opraveno chybné jméno proměnné params na param
10:22 Opraven `import eu.bcvsolutions.idm.data.dto.DT`
10:25 Opravena syntaktická chyba. Znak { nahrazen znakem ;. (problém s detekcí chyby)
10:27 Opraven chybějící středník u importu typu. (opět problém s detekcí chyby)
10:28 Opraveny duplicitní importy datových typů.
10:29 Opraven název proměnné `lastDeleteResult` a doplněno rovnítko u definice proměnné
texttttDTO org.
10:31 Konec testu.

#### Potestový dotazník:

1. Je to příjemná pomůcka, ale bylo by fajn, kdyby při kliknutí na chybu v seznamu byl kurzor přesunut na uvedené místo v editoru.
2. Eclipse již využívám, ale pravděpodobně by se mi vývojové prostředí měnit nechtělo.
3. Těžko říct, ale věřím, že by byla znatelná.
4. Ani ne. Již používám.

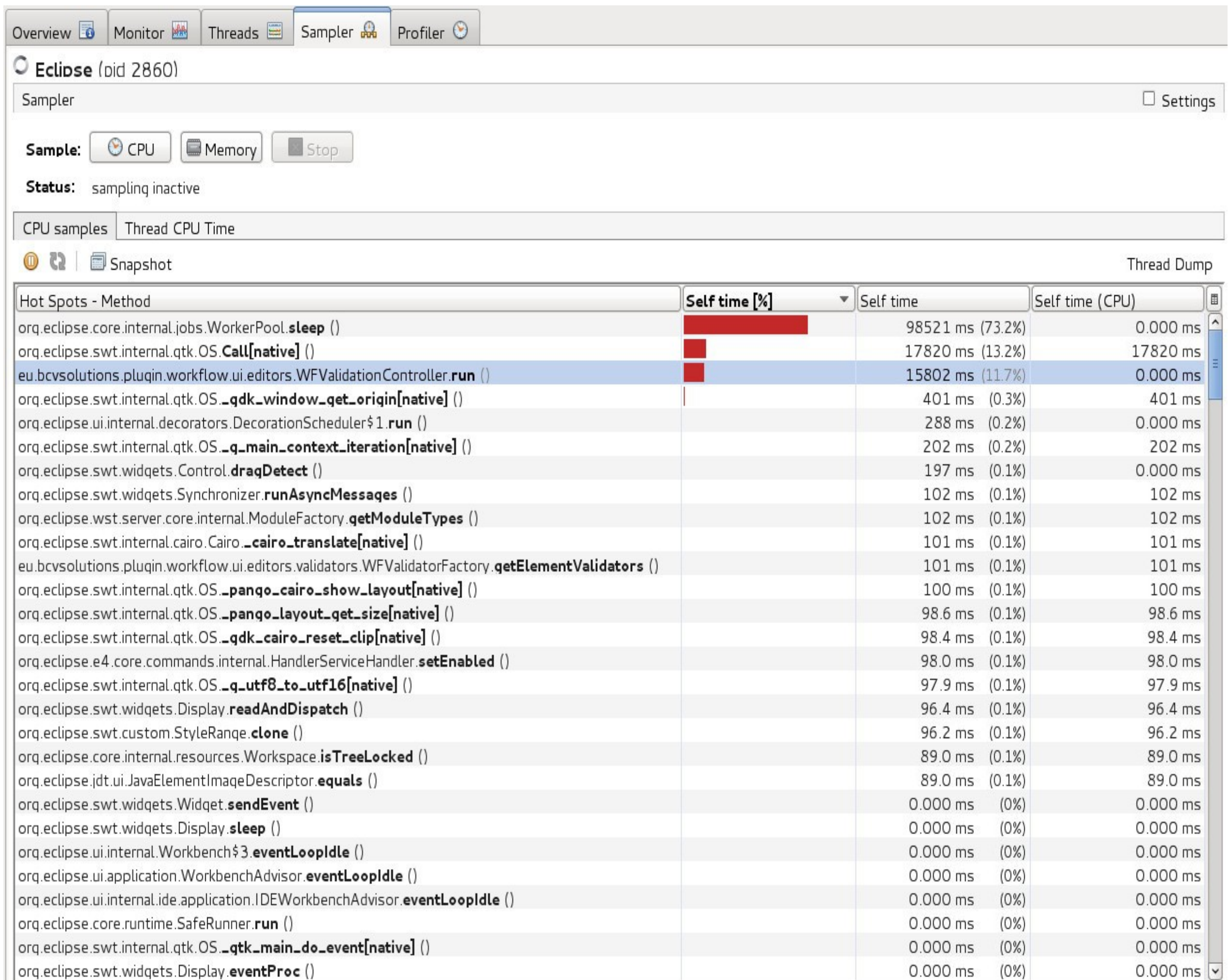
# Příloha I.

# Obrázky




**Obrázek 12.** Graf využití procesoru Java VM

**Obrázek 13.** Graf využití paměti Java VM. Šipky označují zvýšené nároky při spuštění validace obsahu.




**Obrázek 14.** Procentuální využití procesorového času jednotlivými vlákny

Overview | Monitor | Threads | Sampler | Profiler

Eclipse (pid 2860)

Sampler — Settings

**Sample:** CPU | Memory | Stop

**Status:** sampling inactive

Heap histogram | PermGen histogram | Per thread allocations

Deltas | Snapshot — Peform GC | Heap Dump

**Classes:** 7,224 **Instances:** 4,423,284 **Bytes:** 181,786,640

| Class Name | Bytes | Instances |
|---|---|---|
| bsh.**BSHAmbiguousName** | 38,800 (0.0%) | 970 (0.0%) |
| bsh.**BSHArguments** | 10,880 (0.0%) | 340 (0.0%) |
| bsh.**BSHArrayInitializer** | 192 (0.0%) | 6 (0.0%) |
| bsh.**BSHAssignment** | 2,000 (0.0%) | 50 (0.0%) |
| bsh.**BSHBinaryExpression** | 2,320 (0.0%) | 58 (0.0%) |
| bsh.**BSHBlock** | 3,760 (0.0%) | 94 (0.0%) |
| bsh.**BSHCastExpression** | 192 (0.0%) | 6 (0.0%) |
| bsh.**BSHForStatement** | 1,120 (0.0%) | 20 (0.0%) |
| bsh.**BSHFormalParameter** | 240 (0.0%) | 6 (0.0%) |
| bsh.**BSHIfStatement** | 1,536 (0.0%) | 48 (0.0%) |
| bsh.**BSHImportDeclaration** | 7,680 (0.0%) | 192 (0.0%) |
| bsh.**BSHLiteral** | 11,840 (0.0%) | 296 (0.0%) |
| bsh.**BSHMethodInvocation** | 8,576 (0.0%) | 268 (0.0%) |
| bsh.**BSHPrimaryExpression** | 28,096 (0.0%) | 878 (0.0%) |
| bsh.**BSHPrimarySuffix** | 1,824 (0.0%) | 38 (0.0%) |
| bsh.**BSHPrimitiveType** | 2,320 (0.0%) | 58 (0.0%) |
| bsh.**BSHReturnStatement** | 160 (0.0%) | 4 (0.0%) |
| bsh.**BSHStatementExpressionList** | 640 (0.0%) | 20 (0.0%) |
| bsh.**BSHTryStatement** | 128 (0.0%) | 4 (0.0%) |
| bsh.**BSHType** | 12,288 (0.0%) | 256 (0.0%) |
| bsh.**BSHTypedVariableDeclaration** | 9,760 (0.0%) | 244 (0.0%) |
| bsh.**BSHUnaryExpression** | 1,040 (0.0%) | 26 (0.0%) |
| bsh.**BSHVariableDeclarator** | 9,760 (0.0%) | 244 (0.0%) |
| bsh.**JJTParserState** | 1,216 (0.0%) | 38 (0.0%) |
| bsh.**JavaCharStream** | 2,736 (0.0%) | 38 (0.0%) |
| bsh.**Node[]** | 51,152 (0.0%) | 2,742 (0.0%) |
| eu.bcvsolutions.plugin.workflow.tagprovider.**TagRule** | 48 (0.0%) | 1 (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.**Activator** | 48 (0.0%) | 1 (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.editors.**IWFError$Type** | 48 (0.0%) | 3 (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.editors.**IWFError$Type[]** | 24 (0.0%) | 1 (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.editors.**WFDocumentProvider** | 40 (0.0%) | 1 (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.editors.**WFEditor** | 432 (0.0%) | 1 (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.editors.**WFObject** | 48 (0.0%) | 2 (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.editors.**WFObjectBuilder** | 32 (0.0%) | 2 (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.editors.**WFPartitionScanner** | 72 (0.0%) | 1 (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.editors.**WFPosition** | 49,032 (0.0%) | 2,043 (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.editors.**WFSrcViewConf** | 32 (0.0%) | 1 (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.editors.**WFValidationController** | 136 (0.0%) | 1 (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.editors.**WhitespaceDetector** | 32 (0.0%) | 4 (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.editors.colors.**ColorManager** | 16 (0.0%) | 1 (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.editors.parsers.**WFNode$Type** | 112 (0.0%) | 7 (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.editors.parsers.**WFNode$Type[]** | 40 (0.0%) | 1 (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.editors.parsers.**WFXMLAttribute** | 20,032 (0.0%) | 626 (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.editors.parsers.**WFXMLComment** | 48 (0.0%) | 2 (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.editors.parsers.**WFXMLElement** | 21,728 (0.0%) | 388 (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.editors.parsers.**WFXMLInstruction** | 32 (0.0%) | 2 (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.editors.parsers.**WFXMLParser** | 8 (0.0%) | 1 (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.editors.parsers.**WFXMLParser$Parser** | 64 (0.0%) | 2 (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.editors.parsers.**WFXMLRootNode** | 80 (0.0%) | 2 (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.editors.parsers.**WFXMLText** | 1,728 (0.0%) | 54 (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.editors.validators.**BeanShellValidator** | 16 (0.0%) | 1 (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.editors.validators.**GraphValidator** | 16 (0.0%) | 1 (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.editors.validators.**ValidationError** | 336 (0.0%) | 14 (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.editors.validators.**VariableValidator** | 40 (0.0%) | 1 (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.editors.validators.**VariableValidator$PairBuilder** | 3,312 (0.0%) | 138 (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.editors.validators.**VariableValidator$Variable** | 8,832 (0.0%) | 368 (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.editors.validators.**WFValidatorFactory** | 24 (0.0%) | 1 (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.editors.validators.graph.**GraphBuilder** | 64 (0.0%) | 2 (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.editors.validators.graph.**GraphBuilder$BuilderEndNode** | 64 (0.0%) | 2 (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.editors.validators.graph.**GraphBuilder$BuilderNode** | 960 (0.0%) | 40 (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.editors.validators.graph.**GraphBuilder$BuilderStartNode** | 64 (0.0%) | 2 (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.editors.validators.graph.**GraphBuilder$Transition** | 1,872 (0.0%) | 78 (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.editors.validators.graph.**GraphNode** | 5,424 (0.0%) | 226 (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.textscanners.**ITokenDescriptors$Descriptors** | 144 (0.0%) | 6 (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.textscanners.**ITokenDescriptors$Descriptors[]** | 240 (0.0%) | 6 (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.textscanners.**JavaCodeScanner** | 40 (0.0%) | 1 (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.textscanners.**JavaCodeScanner$JavaWordDetector** | 16 (0.0%) | 1 (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.textscanners.**StringScanner** | 40 (0.0%) | 1 (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.textscanners.**XMLCommentScanner** | 40 (0.0%) | 1 (0.0%) |
| eu.bcvsolutions.plugin.workflow.ui.textscanners.**XMLInstructionScanner** | 40 (0.0%) | 1 (0.0%) |

**Obrázek 15.** Procentuální využití paměti jednotlivými třídami

# Literatura


[1] JBoss: Community driven open source middleware. *JBoss jBPM - Workflow in Java: jBPM jPDL User Guide 3.2.7[online]*. 2013. URL: http://docs.jboss.com/jbpm/v3.2/userguide/html_single/.

[2] BeanShell. *Lightweight Scripting for Java [online]*. 2000. URL: http://www.beanshell.org/.

[3] Oracle and/or its affiliates. *Java$^{TM}$ Platform, Standard Edition 7 API Specification [online]*. 2014. URL: http://docs.oracle.com/javase/7/docs/api/index.html.

[4] BeanShell2. *BeanShell2 fork of BeanShell - Google Project Hosting [online]*. 2014. URL: http://code.google.com/p/beanshell2/.

[5] JBoss: Community driven open source middleware. *jBPM - JBoss Community [online]*. 2014. URL: https://www.jboss.org/jbpm/.

[6] Rudolf Pecinovský. *Návrhové vzory*. První vydání. cPress, 2007, s. 527. ISBN: 978-80-251-10582-4.

[7] SAX. *About SAX [online]*. 2004. URL: http://www.saxproject.org/.

[8] JBoss: Community. *JBoss jBPM Graphical Process Designer [online]*. 2014. URL: http://docs.jboss.com/jbpm/v3/gpd/.

[9] JBoss: Community. *jBPM Designer [online]*. 2014. URL: http://jbpm.jboss.org/components/designer.

[10] JBoss: Community. *jBPM Designer 2.4.0 [online]*. 2014. URL: http://sourceforge.net/projects/jbpm/files/designer/designer-2.4/.

[11] Oracle. *javax.xml.parsers (Java Platform SE 7 ) [online]*. 2013. URL: http://docs.oracle.com/javase/7/docs/api/javax/xml/parsers/package-summary.html.

[12] W3C. *Document Object Model (DOM) [online]*. 2009. URL: http://www.w3.org/DOM/.

[13] Pat. NIEMEYER. *BeanShell User's Manual: Simple Java Scripting version 1.3. [online]*. 2002. URL: http://www.beanshell.org/manual/bshmanual.html.

[14] W3C. *Extensible Markup Language (XML) 1.0 (Fifth Edition) [online]*. W3C Recommendation. 2008. URL: http://www.w3.org/TR/xml/.

[15] The Eclipse Foundation. *The Official Eclipse FAQs [online]*. 2014. URL: http://wiki.eclipse.org/index.php/Eclipse_FAQs.

[16] The Eclipse Foundation. *The dropins folder and supported file layouts [online]*. 2014. URL: http://help.eclipse.org/kepler/index.jsp?topic=/org.eclipse.platform.doc.isv/reference/misc/p2_dropins_format.html.

[17] The Eclipse Foundation. *Eclipse documentation [online]*. 2014. URL: http://help.eclipse.org/indigo/index.jsp?topic=/org.eclipse.platform.doc.isv/reference/api/overview-summary.html.

# Seznam obrázků


1. Model BP aukce fiktivního aukčního domu[1] . . . . . . . . . . . . . . . . 6
2. Analyticky model implementovaného parseru . . . . . . . . . . . . . . . . 10
3. Cyklus spuštění validace . . . . . . . . . . . . . . . . . . . . . . . . . . 20
4. Proces přidání validátoru . . . . . . . . . . . . . . . . . . . . . . . . . 25
5. Testovací nástroj určený pro vizualizaci stavu Java VM . . . . . . . . . . 44
6. Procentuální využití procesorového času jednotlivými vlákny . . . . . . 45
7. Graf využití procesoru Java VM . . . . . . . . . . . . . . . . . . . . . . 45
8. Graf využití paměti Java VM. Šipky označují nároky při spuštění validace obsahu. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45
9. Procentuální využití paměti jednotlivými třídami . . . . . . . . . . . . . 46
10. Dokument Workflow zobrazený pluginem. . . . . . . . . . . . . . . . . . 48
11. Zobrazení dokumentu processdefinition.xml po úspěšné instalaci. . . . . 53

# Seznam výpisů kódu

3.1. Workflow BP znázorněného na obrázku 1.[1] . . . . . . . . . . . . . . . . 6

5.1. Vytvoření editoru . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
5.2. Přístup k dokumentu a přidání validátoru. . . . . . . . . . . . . . . . . 14
5.3. Definice pravidel předzpracování dokumentu. . . . . . . . . . . . . . . . 15
5.4. Filtr pro obarvení XML instrukcí. . . . . . . . . . . . . . . . . . . . . . 16
5.5. Nastavení obarvení textu XML instrukce. . . . . . . . . . . . . . . . . . 16
5.6. Filtr pro obarvení programového kódu v jazyce Java. . . . . . . . . . . . 17
5.7. Spuštění validace. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
5.8. Předání správce chyb editoru. . . . . . . . . . . . . . . . . . . . . . . . 21
5.9. Parsování dokumentu. . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
5.10. Parsování instrukce. . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
5.11. Spuštění validace nad daty. . . . . . . . . . . . . . . . . . . . . . . . 24
5.12. Přiřazní validátorů. . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
5.13. Popisek. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
5.14. Přidání nového validátoru. . . . . . . . . . . . . . . . . . . . . . . . . 26
5.15. Abstraktní předek všech validátoru. . . . . . . . . . . . . . . . . . . . 26
5.16. Vytvoření množiny existujících uzlů. . . . . . . . . . . . . . . . . . . . 28
5.17. Validace vnitřní struktury Workflow . . . . . . . . . . . . . . . . . . . 29
5.18. Validace BeanShell skriptu a úprava pozice nalezené chyby v kontextu dokumentu Workflow . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
5.19. Příprava proměnných definovaných v elementu `variable` a základních datových typů. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
5.20. Separace výrazů pro validaci. . . . . . . . . . . . . . . . . . . . . . . . 35
5.21. Určení typu výrazu pro validaci. . . . . . . . . . . . . . . . . . . . . . 37
5.22. Zpracování definice proměnné. . . . . . . . . . . . . . . . . . . . . . . 38
5.23. Validace pravé strany výrazu. . . . . . . . . . . . . . . . . . . . . . . . 39

F.1. Příloha zadání testu . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57