Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Bakalářská práce

Zásuvné moduly a konfigurace domén validačního serveru

Plzeň 2012 Hung Duong Manh

Prohlášení

Prohlašuji, že jsem bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 6. května 2012

Hung Duong Manh

Abstrakt

Tato práce popisuje, jak vytvářet zásuvné moduly a validační domény na validační server používaným Západočeskou univerzitou v Plzni. V první části je stručně popsán validační server, jeho webové rozhraní, studentské úlohy KIV/OOP a jak lze vytvářet pravidla do PMD. Druhá část se zabývá samotným vytvářením externích modulů a poslední část popisuje, jak jsou nakonfigurovány validační domény KIV/OOP ve webovém rozhraní.

Abstract

This thesis describes how to create plugins and validation domains for the validation server used by University of West Bohemia. The first part briefly addresses the validation server, its web interface, student assignments for the course KIV/OOP and how to write a new PMD rule. The second part is a guide on how to create validation server plugins and the last part describes how KIV/OOP validation domains are configured in the web interface.

Poděkování

Rád bych poděkoval doc. Ing. Pavlu Heroutovi, Ph.D. za směrování mé činnosti a Ing. Lukášovi Valentovi a Bc. Veronice Dudové za ochotu mi s čímkoliv pomoci a poradit.

Obsah

1 Úvod **1**

2 Teoretický úvod **2**
2.1 Validační server 2
2.1.1 Přístup na validační server 3
2.1.2 Adresářová struktura validačního serveru 3
2.1.3 Restartování validačního serveru 4
2.2 Webové rozhraní validační serveru 5
2.2.1 Adresářová struktura webového rozhraní 5
2.2.2 Restartování webového rozhraní 6
2.3 Testovací kopie validačního serveru s webovým rozhraním . . . 6
2.4 KIV/OOP – Předmět objektově orientované programování . . 7
2.4.1 JUnit testy 8
2.4.2 Porovnávání obrázků 8
2.4.3 Kontrola UML diagramu tříd 9
2.5 PMD – statická analýza kódu 11
2.5.1 Vytvoření pravidla Java třídou 12
2.5.2 Vytvoření pravidla pomocí XPath 15

3 Zásuvné moduly pro validátor **17**
3.1 Custom validace 17
3.1.1 Vytvoření custom validace v Eclipse 18
3.1.2 Zprovoznění custom validace 20
3.2 Externí vlastní akce 23
3.2.1 Vytvoření externí vlastní akce 23
3.2.2 Zprovoznění externí vlastní akce 26
3.3 Zásuvné moduly pro KIV/OOP 28
3.3.1 Porovnání dvou obrázků 28
3.3.2 Použití custom validace 30
3.4 Knihovna pro kontrolu UML diagramů 31

4 Validační domény pro KIV/OOP **35**
4.1 Klíčové úlohy 35
4.2 Úlohy s kolekcemi 38
4.3 JUnit úloha 39

5 Závěr **40**

Literatura **42**

Webové zdroje **43**

A Ukázka custom validace **44**

B Ukázka externí vlastní akce **47**

C CD Readme **52**

1 Úvod

Na Katedře informatiky a výpočetní techniky Fakulty aplikovaných věd Západočeské univerzity byl vytvořen komplexní systém umožňující automatizované, či poloautomatizované vyhodnocování studentských samostatných prací.

Principem je, že práce odevzdané přes Portál ZČU kontroluje tzv. validační server. Tento validační server postupuje podle naprogramovaných kroků a pokud odevzdaný soubor neprojde validací, není studentská práce přijata.

Doposud si validační server vystačil s jednoduchou metodou kontroly – porovnáním výstupu programu se vzorovým textovým souborem. Avšak s ohledem na možnosti validačního serveru a možným budoucím využitím je třeba řešit i další metody kontroly. Takové kontroly mohou například zahrnovat spouštění jednotkových testů, porovnávání grafického výstupu či kontrolu UML diagramů, případně obecně jakoukoliv programovatelnou učitelskou kontrolu.

Pro validační server bylo také nově vytvořeno webové uživatelské rozhraní, které mělo za úkol zjednodušit práci s konfigurací validátoru. Žádný aktivní předmět však dosud neměl svojí validační doménu vytvořenou v tomto webovém rozhraní.

Cílem práce je vytvoření validačních domén pro předmět OOP ve webovém rozhraní, což bude bráno jako pilotní projekt pro budoucí využití konfigurace validačního serveru v dalších předmětech. Práce dále předpokládá vytvoření několika nových zásuvných modulů přímo použitelných pro kontrolu výstupů studentských prací. K tomu je třeba využít všech již existujících možností validačního serveru.

2 Teoretický úvod

2.1 Validační server

Jakýkoliv studijní předmět, ve kterém je třeba odevzdat samostatnou práci elektronicky, může využít možnost odevzdávání přes Portál ZČU. Tento způsob odevzdávání samostatných prací má mnoho výhod, nás však hlavně zajímá možnost kontrolovat odevzdávané soubory přes validační server.

Tím, že se soubory kontrolují validačním serverem, se zajistí téměř okamžitá odpověď studentovi, zda jeho práce vyhovuje požadavkům či nikoliv. Student nemusí čekat na výsledky s tím, že jeho práce není správně a musí ještě něco opravit. Vyučující naopak nemusí podrobně kontrolovat každou práci a to, zda práce vyhovuje či nikoliv, je stanoveno dle objektivního kritéria, které je pro všechny práce stejné.

Validační server funguje jako volitelná nadstavba odevzdávacího portletu, viz [11]. Portlet obdrží odevzdaný soubor a ten přepošle validačnímu serveru. Validační server nejprve provede úvodní doménové kontroly – např. maximální možná velikost souboru, povolené přípony atd. – a pokud soubor projde, spustí se samotný validační proces. Pro každý validační proces je vytvořen výstup, do kterého může autor validačního procesu generovat informativní, varovné či chybové zprávy. Pokud dojde k přidání chybové zprávy do výstupu, neprojde soubor validací jako takovou. Po dokončení validace je pak její výstup převeden do HTML stránky a studentovi je na ni poskytnut odkaz. Výstup nemusí být ale převeden jen do HTML stránky pro studenta, další formy výstupu mohou být například uložení do databáze nebo `XML` souboru.

HTML výstupy se generují na adresu:

http://vs.kiv.zcu.cz/validator/

Přesná cesta je určena v souboru `domain.xml` a podle ní se vytvoří adresářová struktura pro HTML výstupy. Podrobnější informace o validačním serveru viz [2], [10].

2.1.1 Přístup na validační server

Adresa validačního serveru je:

validator.kiv.zcu.cz

Pokud se na tuto adresu připojíme pomocí protokolu HTTPS, dostaneme se na webové rozhraní validačního serveru (viz 2.2).

Pokud se na tuto adresu připojíme pomocí protokolu SSH přes port 22, dostaneme se na samotný souborový systém validačního serveru.

SVN repozitář se zdrojovými kódy se nachází na adrese:

svn+ssh://forge.kiv.zcu.cz/home/svn/validator/ValidacniServer/trunk

Údaje pro přístup na validační server a na SVN repozitář poskytuje a spravuje Ing. Lukáš Valenta (lvalenta@civ.zcu.cz).

2.1.2 Adresářová struktura validačního serveru

```
/home/validator/validator
|
| VS.jar
| restart
| start
| stop
|    [lib]
|    [data]
|    |    [domains]
|    |    |    [common]
|    |    |    [ppa1]
|    |    |    |  ...
|    |    |    | domain.xml
|    |    |    | run.policy
|    |    |    [ssp07]
|    |    |     ...
|    |    [html]
|    |    |  vs.css
|    |    [log]
|    |    |  vs.log
|    |    [store]
|    |    [workdir]
```

Ukázka 2.1: Adresářová struktura validačního serveru

Pro učely vytváření zásuvných modulů je třeba seznámit se s adresářovou strukturou a důležitými soubory validačního serveru. Ukázka 2.1 je převzata z [10].

Protože účty pro přístup na validační server mohou mít jinak nastaveny domovské adresáře po přihlášení, je v ukázce 2.1 na první řádce uvedena absolutní cesta k ostrému validačnímu serveru v rámci souborového systému.

Název	**Použití**
`VS.jar`	hlavní spustitelný `JAR` serveru
`restart`	skript na restartování validátoru
`lib`	složka pro pomocné knihovny
`domain.xml`	nastavení validační domény
`run.policy`	nastavení práv ve validační doméně
`workDir`	dočasné pracovní složky validací, důležité zejména pro ladění

Tabulka 2.1: Popis důležitých částí validačního serveru

2.1.3 Restartování validačního serveru

Občas je po provedení nějakých změn nutno validační server restartovat, aby se změny projevily. Restart trvá přibližně dvě a půl sekundy, tudíž studenti by neměli ani postřehnout, že server neběží. Přesto je ale vhodnější provádět restart v době, kdy se neočekává, že by studenti odevzdávali své práce. Může se totiž stát, že změny, které jsme provedli, budou vykonávat jinou činnost, než jakou jsme požadovali, a server budeme muset restartovat vícekrát. Při rozsáhlejších změnách je vhodné použít nejdříve testovací server (viz 2.3) a na něm úpravy odladit.

Restart validačního serveru se provádí tak, že se pomocí programu Putty připojíme na validační server přes protokol SSH:

validator.kiv.zcu.cz:22

a tam, v kořenové složce validátoru, spustíme skript `restart` (viz 2.1.2).

```
./restart
```

2.2 Webové rozhraní validační serveru

V původním validačním serveru bez webového uživatelského rozhraní byl celý průběh validace řízen skriptem `process.xml`, který obsahoval kombinaci `XML` elementů s JavaScriptovým programem (Princip fungování validačního serveru – [10]). Pokud vyučující nebyl detailně seznámen s možnostmi konfigurace, bylo vytváření validačních domén obtížné.

Z tohoto důvodu vzniklo webové uživatelské rozhraní. Uživatelské rozhraní poskytuje srozumitelnější způsob, jak nastavit validační domény. Je vytvořeno pomocí servletů a JSP stránek a využívá webový sever Tomcat [2]. Samotný validační proces je rozdělen do posloupnosti kroků, které mají jednoznačnou funkci, a podmínku, za jaké se spustí. Autor validace může navíc vytvářet JavaScriptové proměnné, které jsou viditelné ze všech kroků. Pomocí těchto proměnných tak může předávat data z jednoho kroků do dalších.

Kroky validace

1.	rozbaleni	Uprav	Zruš
2.	hledej_testOsoby	Uprav	Zruš
3.	nenalezeno_testOsoby	Uprav	Zruš
4.	konec_chyba	Uprav	Zruš
5.	spusteni_junit_testu	Uprav	Zruš
6.	porovnani_vystupu	Uprav	Zruš

nový krok

Obrázek 2.1: Ukázka kroků validační domény

2.2.1 Adresářová struktura webového rozhraní

```
/home/validator/tomcat/webapps/ROOT
|
| domenaNastaveni.jsp
| domenaSmaz.jsp
|  ...
|     [css]
|     [imgs]
|     [js]
|     [META-INF]
|     [WEB-INF]
|     |      [classes]
|     |      [data]
|     |      [libg
```

Ukázka 2.2: Adresářová struktura webového rozhraní

Název	**Použití**
`data`	jsou zde uloženy jednotlivé kroky námi vytvořených domén
`lib`	složka pro knihovny

Tabulka 2.2: Popis důležitých částí webového rozhraní

Někdy potřebujeme překopírovat určité kroky z jedné domény do druhé nebo potřebujeme vytvořit větší počet kroků najednou. V takovém případě by bylo klikání přes webové rozhraní pracné. Řešením je modifikovat náš soubor s kroky ve složce `WEB-INF/data`. Tento soubor, který je pojmenovaný podle našeho Orion loginu, obsahuje všechny domény a jejich kroky, které jsme v rámci webového rozhraní vytvořili. Pro projevení změn v souboru stačí restartovat webové rozhraní, samotný validační server se restartovat nemusí.

2.2.2 Restartování webového rozhraní

Abychom restartovali webové rozhraní, musíme restartovat webový server Tomcat. To se provede podobným způsobem jako restartování validačního serveru (viz 2.1.3). Skripty na spuštění a vypnutí Tomcatu jsou ale v:

```
/home/validator/tomcat/bin
```

Samostatný skript pro restart tu není, musíme Tomcat vypnout a znovu zapnout skripty `shutdown.sh` a `startup.sh`.

```
./shutdown.sh
./startup.sh
```

2.3 Testovací kopie validačního serveru s webovým rozhraním

Pro účely testování a ladění validačních domén a externích modulů byl vytvořen testovací validační server, který je téměř identický s ostrým, pouze

má jiné cesty. Autoři validačních domén zde mohou testovat své konfigurace, aniž by při tom narušovali běh ostrého serveru. Protože se testovací validační server používá pro vývoj a ozkoušení externích modulů, poskytuje funkce, které ostrý server dosud nevyužívá. Do ostrého serveru se pak tyto moduly zavedou až bude nutné.

Adresa webového rozhraní testovacího validátoru je:

https://validator.kiv.zcu.cz/val-test/

Pro přístup k souborovému systému testovacího validátoru je adresa stejná jako pro ostrý, viz 2.1.1. Samotný testovací validátor je pak umístěn ve složce:

```
\home\valtest\val-test
```

2.4 KIV/OOP – Předmět objektově orientované programování

Předmět si klade za cíl naučit studenty objektově orientovaný styl programování spolu s dobrými programátorskými návyky. Studenti přitom mimo jiné pracují s JUnit testy, UML diagramy či statickou kontrolou zdrojových kódů.[3] Samostatné projekty jsou vytvořeny tak, aby byly jednoduše strojově ověřitelné. Tento předmět je tedy velmi vhodný pro kontrolu úloh pomocí validátoru.

Studenti v rámci předmětu pracují na celkem až jedenácti samostatných úlohách. Prvních osm úloh je klíčových (tj. povinných) a týkají se objektově orientovaného designu, psaní JavaDoc dokumentace a vytváření UML diagramů. Další dvě úlohy jsou na téma kolekce v Javě a poslední úloha se zabírá vytvářením JUnit testů.

Jedním z požadavků této práce bylo vytvořit validační domény pro předmět KIV/OOP. Validační domény jako takové už byly předtím vytvořené, pouze bylo třeba je převést do formátu webového rozhraní. Některé domény byly následně doplněné o další kroky, aby byla kontrola přísnější, jiné bylo nutno kompletně předělat a vytvořit k nim zásuvné moduly, aby je bylo možno spustit z webového rozhraní.

2.4.1 JUnit testy

Prvních osm úloh se kontroluje pomocí JUnit testů. JUnit je open source nástroj na psaní a spouštění jednotkových testů pro programovací jazyk Java [7]. Jednotkové testy fungují tak, že si programátor napíše určitou sadu testů, které porovnávají očekávanou a skutečnou návratovou hodnotu metod, a pokud testy proběhnou v pořádku, lze předpokládat, že metody pracují správně. Jednou z výhod jednotkových testů je, že po jejich vytvoření může probíhat testování zcela automaticky.

S jednotkovými testy se také pojí programovací přístup, kdy nejdříve se napíšou jednotkové testy a až potom se napíše vlastní kód. Tomuto přístupu se říká „programování řízené testy".[1] V samostatných úlohách předmětu KIV/OOP si studenti zkoušejí oba způsoby práce s jednotkovými testy, tedy jak situaci, kdy mají k dispozici pouze jednotkové testy a musejí vytvořit třídy a metody, tak situaci, kdy mají ověřit funkčnost třídy pomocí testů.

Studenti každou samostatnou klíčovou úlohu odevzdávají jako celý projekt se zdrojovými soubory, v pracovní složce na validačním serveru se pak vybrané soubory projektu přepíšou vzorovými a celý projekt se zkompiluje. Po zkompilování se spustí JUnit testy a pokud testy neohlásí chybu, je projekt přijat. Společně s testy se také pořídí snímek obrazovky (studenti pracují s vykreslovacím plátnem).

2.4.2 Porovnávání obrázků

Na rozdíl od zbylých sedmi klíčových úloh odevzdávají studenti v první úloze pouze jeden soubor s testy a přípravkem (sada objektů, s nimiž budou testy pracovat). Studenti mají na začátku k dispozici pouze jednotkové testy a jejich úkolem je vykreslit na plátno pomocí geometrických objektů jednoznačně definovaný obrázek.

Jak už bylo řečeno, validace pro klíčové projekty předmětu KIV/OOP spoléhá výhradně na průběh JUnit testů. Bylo by proto možné, aby si studenti napsali svůj přípravek a testy si pozměnili, aby vyhovovaly jejich přípravku. To mělo za následek, že by testy na straně serveru prošly, ačkoliv by správné nebyly. Vzhledem k tomu, že studenti odevzdávají pouze jeden soubor, není jednoduše možné jej přepsat vzorovými testy. Vyvstala tedy nutnost najít jiný způsob, jak ověřit správnost odevzdané práce.

Ze zmíněného důvodu se do procesu validace začlenilo i pořizování snímku obrazovky, protože obrázek lze porovnat vůči vzoru. Samotné porovnávání dvou obrázků ale vůbec není triviální záležitost.

Předpokládejme, že bychom porovnávali dva obrázky ve formátu JPEG. Vlivem komprese by obrázky nemusely být binárně totožné, i kdyby byly vygenerovány stejným programem a se stejným nastavením.

Naštěstí lze pořizovat snímek plátna do bezztrátového formátu PNG. Dále máme jistotu, že všechny pořízené snímky budou mít stejné rozlišení, protože studenti pracují se stejným frameworkem.

Porovnávání obrázků se nám tudíž zjednodušilo na problém, kdy stačí pouze porovnávat pixel po pixelu první obrázek s druhým.

2.4.3 Kontrola UML diagramu tříd

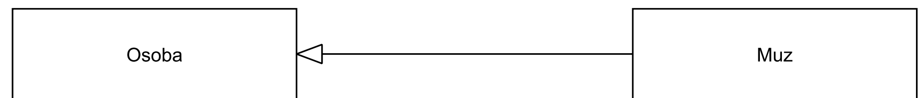
Od páté klíčové úlohy předmětu KIV/OOP až do osmé odevzdávají studenti kromě samotných zdrojových kódů také UML digramy tříd.[5] Tyto UML diagramy tříd vytvářejí v jednoduchém programu zvaném UMLet, který umí diagramy exportovat do formátu PNG. Doposud se tyto diagramy tříd musely kontrolovat ručně jako obrázky.

Pro zautomatizování kontroly se zdánlivě nabízí možnost kontrolovat diagramy stejně jako v 2.4.2. Tento přístup je ale nepoužitelný, protože ačkoliv studenti použijí stejné třídy jako ve vzorovém obrázku, každý obrázek bude mít jiné rozložení.

Naštěstí však UMLet dovoluje ukládat diagramy jako `XML` soubory, tudíž lze snadno programově získat strukturu diagramu. Ukázka diagramu s dvěma třídami a jednou relací:

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<diagram program="umlet" version="11.3">
  <zoom_level>10</zoom_level>
  <element>
    <type>com.umlet.element.Class</type>
    <coordinates>
      <x>20</x>
      <y>20</y>
      <w>100</w>
      <h>30</h>
    </coordinates>
    <panel_attributes>Osoba</panel_attributes>
    <additional_attributes/>
  </element>
  <element>
    <type>com.umlet.element.Class</type>
    <coordinates>
      <x>250</x>
      <y>20</y>
      <w>100</w>
      <h>30</h>
    </coordinates>
    <panel_attributes>Muz</panel_attributes>
    <additional_attributes/>
  </element>
  <element>
    <type>com.umlet.element.Relation</type>
    <coordinates>
      <x>90</x>
      <y>0</y>
      <w>180</w>
      <h>50</h>
    </coordinates>
    <panel_attributes>lt=&lt;&lt;-</panel_attributes>
    <additional_attributes>30;30;160;30</additional_attributes>
  </element>
</diagram>
```

Ukázka 2.3: Diagram uložený v UMLet



Obrázek 2.2: Výsledek ukázky 2.3

Za zmínku stojí dvě skutečnosti. Element `type` se už nijak dále nedělí a to jak pro třídy, tak i pro relace. Když chceme u třídy specifikovat, že se jedná například o rozhraní, tak to lze provést pouze tak, že se napíše typ třídy jako text do obdélníku (element `panel_attributes`) a je jenom na autorovi diagramu, aby používal konzistentní značení pro daný typ třídy (v tomto případě <<`interface`>>).

Směr relace se určuje pomocí hodnot v elementu `additional_attributes`. Hodnoty jsou vždy po dvojicích a platí, že první dvojice zprava se vztahuje na počáteční bod relace a poslední dvojice nalevo na koncový bod relace. Dvojice jsou ve tvaru X, Y, kdy po přičtení těchto hodnot k souřadnicím X, Y relace získáme výsledné souřadnice bodu. V naší ukázce je to (relace vede od `Muz` k `Osoba`):

- počáteční bod:
  - $X = 90 + 160 = 250$
  - $Y = 0 + 30 = 30$
- konečný bod:
  - $X = 90 + 30 = 120$
  - $Y = 0 + 30 = 30$

Pro zpracování `XML` souboru s diagramem bylo možné použít technologii Simple API for XML (SAX) nebo Document Object Model (DOM). V podstatě bylo jedno jakou technologii zvolím, protože na UMLet souborech by se neprojevila žádná z výhod jednotlivých technologií. DOM vytvoří strom uzlů, který je po celou dobu v paměti a ve kterém lze přistupovat k jakémukoliv uzlu. Jenže vzhledem k tomu, že jednotlivé prvky diagramu budu stejně muset převést na Java objekty, tak si tím nijak nepomůžu. Nakonec jsem se rozhodl pro SAX, protože i když je zpracování pouze sekvenční, tak je rychlejší a méně náročné na paměť než u DOM [4], [9].

2.5 PMD – statická analýza kódu

PMD [8] je opensource nástroj, který slouží k analýze zdrojových souborů napsaných v jazyce Java. V předmětech KIV/PPA1, KIV/PT či KIV/JXT se používá především pro výuku správných programovacích praktik, ale je možno ho použít i pro jiné účely, např. odhalování bugů (prázdné try/catch příkazy) nebo hledání duplicitního kódu. Chyby odhalené pomocí PMD nejsou chyby v pravém slova smyslu (tj. funkčnosti), ale spíše chyby v kvalitě zdrojového kódu.

Používání PMD je ve výše uvedených předmětech volitelné. Pro předmět KIV/OOP by se ale PMD mohlo stát přirozenou součástí validačního

procesu. V minulosti už byly pokusy začlenit PMD do validačního serveru, bohužel ale nebyly dotaženy do zdárného konce. Navíc v té době nebylo k dispozici ani webové rozhraní.

Spouštění PMD se provádí pomocí `JAR` souboru, kterému předáme jako parametry soubory, které chceme zkontrolovat, typ výstupu a pravidla, která chceme použít. Rozhodně by tedy nebyl problém vytvořit externí modul do validátoru, který by spouštěl PMD nad odevzdanými soubory, a výsledek kontroly by přesměroval do celkového výsledku validace. V případě, že by se nepoužilo příliš mnoho složitých pravidel, by PMD mohlo být pro studenty užitečné. Míru složitosti stanoví snadno vyučující.

Nová pravidla do PMD lze vytvářet jako Java třídy nebo jako XPath výrazy.

2.5.1 Vytvoření pravidla Java třídou

PMD nezpracovává samotné zdrojové soubory, pouze z nich parsuje *Abstract Syntax Tree* (AST) a s tímto stromem výrazů pak dále pracuje.

```
public class Trida {
  int a;
}
```

Ukázka 2.4: Jednoduchá třída

```
TypeDeclaration
  ClassOrInterfaceDeclaration:Trida
    ClassOrInterfaceBody
      ClassOrInterfaceBodyDeclaration
        FieldDeclaration
          Type
            PrimitiveType:int
          VariableDeclarator
            VariableDeclaratorId:a
```

Ukázka 2.5: Odpovídající AST

Pravidla se píší tak, že reagujeme na výskyt, či nevýskyt určitého uzlu. Chceme-li vytvořit nové pravidlo, musíme jako první vědět, na jaký uzel reagujeme. To zjistíme tak, že použijeme program `designer.bat`, který je součástí distribuce PMD. Tímto programem zpracujeme zdrojový kód v Javě a dostaneme vygenerovaný AST.

Dejme tomu, že bychom chtěli vyvolat porušení pravidla při jednoznakovém pojmenování instanční proměnné typu `int`. Bude nás tedy zajímat uzel `VariableDeclaratorId`.

Když už víme, na jaký uzel budeme reagovat, tak si můžeme vytvořit Java třídu:

```
import net.sourceforge.pmd.*;           // Z knihovny "pmd-(cisloVerze).jar"
import net.sourceforge.pmd.ast.*;

public class OurRule extends AbstractJavaRule {

  public Object visit(ASTFieldDeclaration node, Object data) {
    Node type = node.jjtGetChild(0);
    Node varDeclarator = node.jjtGetChild(1);

    if (isInt(type)) {
      if (isOneCharLong(varDeclarator)) {
        addViolation(data, node);
      }
    }
    return super.visit(node,data);
  }

  private boolean isInt(Node type) {
    SimpleNode primitiveType = (SimpleNode)type.jjtGetChild(0);
    return (primitiveType instanceof ASTPrimitiveType &&
        primitiveType.getImage().equals("int"));
  }

  private boolean isOneCharLong(Node varDeclarator) {
    SimpleNode varDeclaratorId = (SimpleNode)varDeclarator.jjtGetChild(0);
    return (varDeclaratorId instanceof ASTVariableDeclaratorId &&
        varDeclaratorId.getImage().length() == 1);
  }
}
```

Ukázka 2.6: Porušení pravidla při jednoznakové instanční proměnné typu `int`

V metodě `visit()` musíme jako první argument uvést přímo nadřazený uzel pro `VariableDeclaratorId`. Kdybychom chtěli, aby pravidlo platilo pro všechny názvy proměnných, tak bychom si vystačili s `VariableDeclarator`. My ale chceme konkrétně pouze instanční proměnné typu `int`, a proto musíme vybrat `FieldDeclaration (ASTFieldDeclaration)`.

Metodou `jjtGetChild()` získáme potomky uzlu `FieldDeclaration`, v našem případě to jsou `Type` a `VariableDeclarator`.

V konkrétních podmínkách spojených s nějakým uzlem bychom se vždy měli přesvědčit, že se jedná o požadovaný typ uzlu operátorem `instanceof`.

Kdyby nám totiž chyběla tato podmínka v metodě `isInt()`, tak by nám vznikla chyba při proměnné typu `String`. Uzel pro `String` není `ASTPrimitiveType` a tudíž nemá žádný parametr `int`.

Ještě je třeba vytvořit tzv. ruleset neboli sadu pravidel, který využije naši vytvořenou třídu. Vytvoříme si libovolně pojmenovaný `XML` soubor (například `mycustomrules.xml`), ve kterém bude následující:

```
<?xml version="1.0"?>
<ruleset name="Moje pravidla"
  xmlns="http://pmd.sourceforge.net/ruleset/2.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://pmd.sourceforge.net/ruleset/2.0.0 http://pmd.sourceforge.net/
       ruleset_2_0_0.xsd"
  xsi:noNamespaceSchemaLocation="http://pmd.sourceforge.net/ruleset_2_0_0.xsd">

  <description>
Ukazka noveho pravidla
  </description>
  <rule name="OurRule"
    message="Instancni promenne typu int nesmi mit jeden znak!"
    class="OurRule">
  <description>
Instancni promenne typu int nesmi mit jeden znak!
  </description>

  <example>
<![CDATA[
public class Trida {
  int a;          // jednoznakovy identifikator neni povolen
}

]]>
  </example>
  </rule>
</ruleset>
```

Ukázka 2.7: Ruleset

V PMD pak použijeme tento ruleset například takto:

```
java -cp 1;2;3;4 net.sourceforge.pmd.PMD analyzovanySoubor.java text mycustomrules.xml

1: cesta k souboru jaxen-(verze).jar  // ve slozce lib
2: cesta k souboru asm-(verze).jar    // stazeneho distribucniho
3: cesta k souboru pmd-(verze).jar    // baliku
4: slozka s class souborem nasi Java tridy
text: textovy vystup
```

2.5.2 Vytvoření pravidla pomocí XPath

XPath je jazyk určený pro navigování nad `XML` dokumenty a protože PMD pracuje se strukturou zdrojového kódu jako se stromem, lze XPath použít i zde. Kromě samotného výběru uzlů a atributů umožňuje XPath také jednoduché výpočty (viz [6]).

Vytvoření nového pravidla pomocí XPath začíná stejným způsobem jako u Java třídy – identifikujeme potřebný uzel.

Mějme tutéž třídu jako v ukázce 2.4 v sekci 2.5.1. Nástroj `designer.bat` umí kromě generování AST také vyhodnocovat XPath výrazy nad daným AST. Například výraz, který od kořene hledá všechny deklarované proměnné s názvem „a“:

```
//VariableDeclaratorId[@Image="a"]
```

by nám vypsal:

```
ASTVariableDeclaratorId at line 2 column 5
```

Tentokrát budeme chtít vytvořit pravidlo, aby název jakékoliv proměnné nebyl kratší než stanovené minimum (v ukázce je minimum 3). Pravidlo se bude psát přímo do rulesetu, například `mycustomrules2.xml`, do elementu `value`.

```
<?xml version="1.0"?>
<ruleset name="Moje pravidla"
  xmlns="http://pmd.sf.net/ruleset/1.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://pmd.sf.net/ruleset/1.0.0 http://pmd.sf.net/ruleset_xml_schema.xsd"
  xsi:noNamespaceSchemaLocation="http://pmd.sf.net/ruleset_xml_schema.xsd">

  <rule name="ShortVariable"
  message="Promenna musi mit alespon 3 znaky"
  class="net.sourceforge.pmd.rules.XPathRule">

  <description>
    Detects when a field, formal or local variable is declared with a short name.
  </description>

  <priority>3</priority>
  <properties>
    <property name="minimum" description="The variable length reporting threshold" value="3"/>
    <property name="xpath" pluginname="true">
    <value>
```

```
      <![CDATA[
//VariableDeclaratorId[string-length(@Image) < $minimum]
      ]]>
    </value>
    </property>
  </properties>
  <example>
  <![CDATA[
public class Something {
  int aa = -5;      // Moc kratky nazev promenne!
}
]]>
  </example>
  </rule>
</ruleset>
```

Použití rulesetu je podobné jako v 2.5.1, pouze není třeba přidávat do classpath složku s `class` souborem.

3 Zásuvné moduly pro validátor

S nasazením webového uživatelského rozhraní je k dispozici mnohem intuitivnější způsob ovládání. Celá validace zůstala pod validační doménou a pomocí webového rozhraní jsou nyní jasně patrné kroky, které se mají provést. Webové rozhraní toho ale samo o sobě moc neumí, pouze poskytuje přehlednější prostředí. Funkčnost získává až pomocí zásuvných modulů.

Zásuvné moduly se používají jako akce v jednotlivých krocích. Lze je tedy označovat jako „vlastní akce". V textu budou dále oba tyto termíny zaměňovány. Tyto vlastní akce by se měly programovat co možná nejvíc modulárně.

V době psaní tohoto textu je na ostrém serveru k dispozici jen několik základních vlastních akcí jako jsou *Rozbalení archivu*, *Najdi soubory* či *Kontrola programu* (akce zejména určené pro předmět PPA1). Na testovacím validačním serveru jsou však k dispozici další vlastní akce, z nichž jedna je pro dále popisované postupy podmiňující. Jedná se o vlastní akci *Spuštění custom validace*. Očekává se, že ostrý validační server bude ještě tento rok (2012) aktualizován na verzi, kde bude i tato vlastní akce.

Zásuvné moduly pro validační server lze vytvořit dvěma způsoby: jako custom validaci (název vychází ze jména hlavního rozhraní), nebo jako externí vlastní akci do webového rozhraní.

Všechny dále uvedené postupy jsou uvedeny pro operační systém Windows 7 a popsány podrobněji, aby podle nich bylo možné v budoucnu validace připravovat.

3.1 Custom validace

Požadavky:

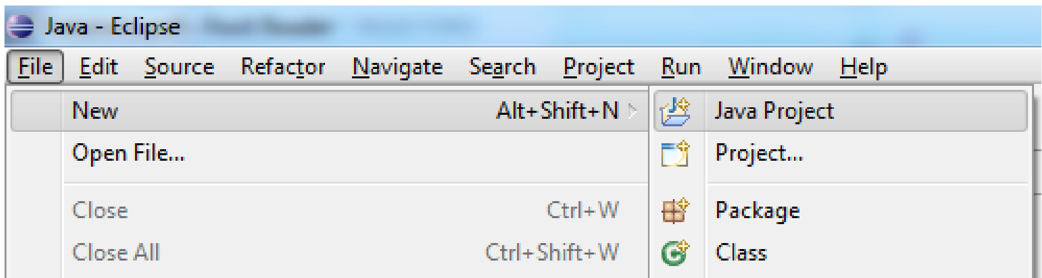
- Java SE SDK 6u30,
- Eclipse SDK (Indigo SR2),
- WinSCP,
- přístup k validačnímu serveru.

Custom validace je starší způsob vytváření zásuvných modulů. V porovnání s druhým způsobem nepodporuje JavaScriptové proměnné webového rozhraní a pokud potřebujeme měnit parametry akce, je to obtížnější (do `domain.xml` se dopíše vlastnost a ta se pak v kódu získává). Výhodou je však snadnější zprovoznění nově vytvořené akce a ladění. Z toho všeho vyplývá, že custom validace se spíše hodí pro specializované a jednorázové akce nebo pokud potřebujeme rychle něco vyzkoušet.

3.1.1 Vytvoření custom validace v Eclipse

Pro vytvoření custom validace je třeba mít knihovnu `VS-API.jar`, která je součástí distribuce validátoru (k dispozici také na přiloženém CD).

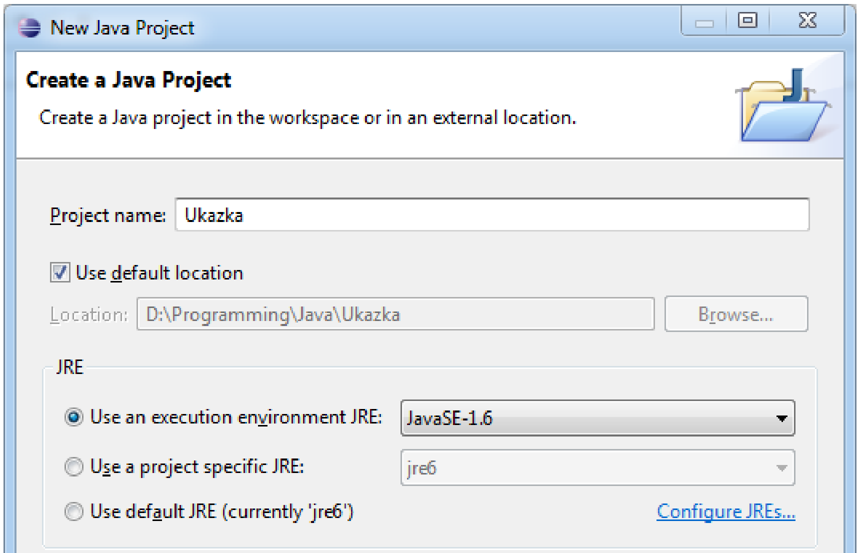
1. **Vytvoření nového projektu**
   File → New → Java Project



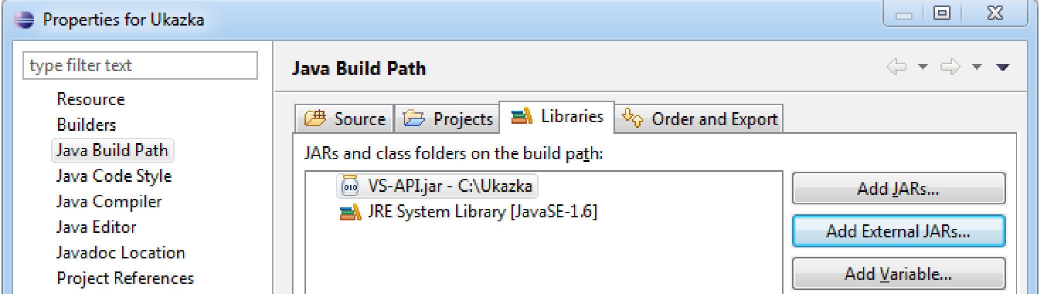
Obrázek 3.1: Vytvoření nového projektu

2. **Pojmenování projektu a nastavení verze prostředí**
   V dalším okně zadáme název projektu a nastavíme verzi prostředí. Validační server ještě nepodporuje Javu 1.7, proto vybereme verzi 1.6. Nakonec klikneme na tlačítko *Finish*

3. **Přidání knihovny `VS-API.jar` do projektu**
   Knihovnu `VS-API.jar` nakopírujeme do libovolné složky (např. do složky projektu). V Eclipse v okně *Package Explorer* klikneme na náš projekt pravým tlačítkem a zvolíme *Properties*. Najdeme vlastnost *Java Build Path* a poté zvolíme záložku *Libraries*. Zde klikneme na tlačítko *Add External JARs...* a zadáme místo, kam jsme nakopírovali knihovnu `VS-API.jar`. Po vybrání knihovny by se nám měla zobrazit v seznamu knihoven. Nic dalšího zde už nepotřebujeme, proto dáme *OK*.

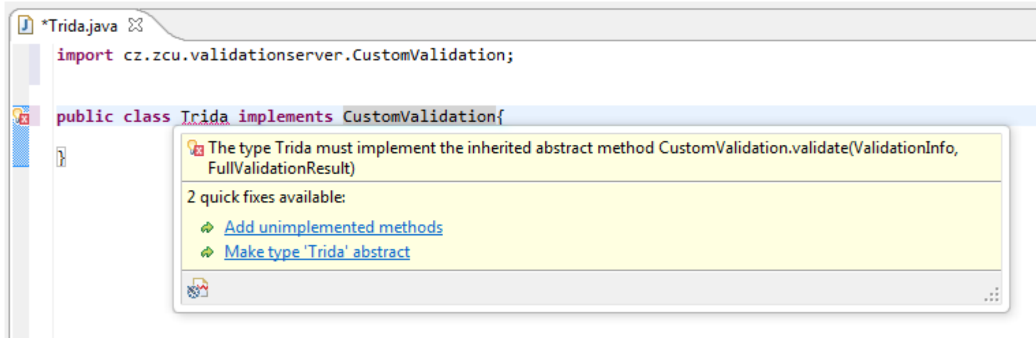


Obrázek 3.2: Zadání názvu projektu a verze prostředí



Obrázek 3.3: Přidání knihovny `VS-API.jar` do projektu

4. **Vytvoření třídy s metodou** `validate()`
   Vytvoříme v Eclipse v našem projektu novou třídu (nemusí mít balíček). V této třídě stačí už jen implementovat rozhraní `CustomValidation` dostupné díky knihovně `VS-API.jar`, vytvořit metodu `validate()` a můžeme začít psát samotný kód validace. Celou hlavičku metody `validate()` získáme buď v Eclipse, kdy najedeme myší na podtržený název třídy a vybereme *Add unimplemented methods*, nebo z JavaDoc dokumentace validátoru.



Obrázek 3.4: Doplnění metody `validate()` pomocí funkce *quick fix*

Vše, co se má během validace provést, se píše do metody `validate()`. Metoda `validate()` má dva parametry a to `ValidationInfo` a `FullValidationResult`. `ValidationInfo` obsahuje data, které nám předá validační server např. serverem přijatý soubor, atributy domény, absolutní cestu k dočasnému pracovnímu adresáři apod. Do `FullValidationResult` se naopak generují informace, varování a chyby, které se následně vypíší do výstupní HTML stránky. Dostupné metody těchto předaných objektů lze opět získat pomocí doplňování kódu v Eclipse nebo v JavaDoc dokumentaci validátoru.

```
public class ValidationOOP11 implements CustomValidation {

  @Override
  public void validate(ValidationInfo validationInfo, FullValidationResult
       fullValidationResult) {

    String filename = validationInfo.getInputFile().getName();
    if (!filename.equals("SpravnyNazev.java")) {
      fullValidationResult.addError("Odevzdavany soubor se nejmenuje
          'SpravnyNazev.java'!");
    }
  }
}
```

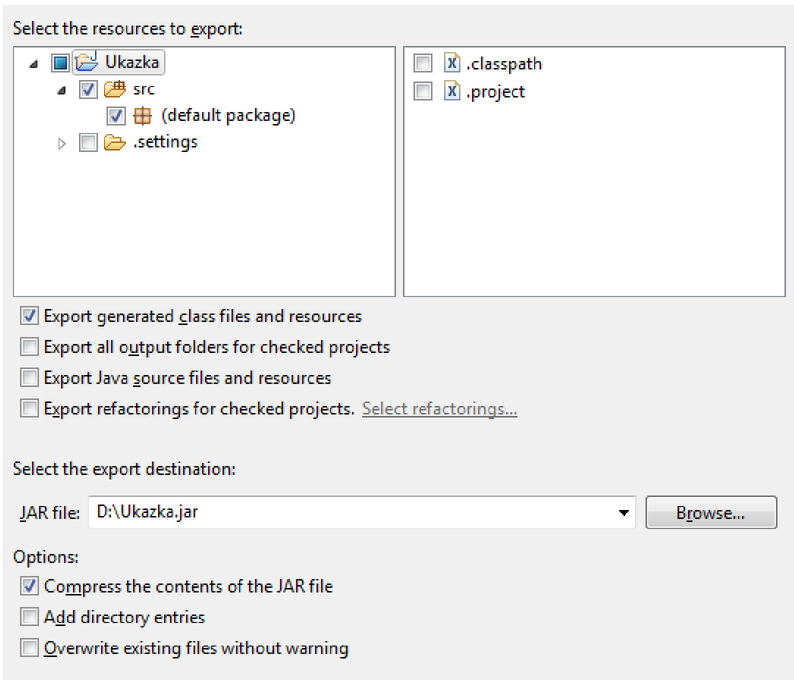
Ukázka 3.1: Příklad custom validace

3.1.2 Zprovoznění custom validace

Po naprogramování třídy s metodou `validate()` je třeba tuto třídu (a další třídy, pokud byly vytvořeny) umístit jako `JAR` soubor na validační server a

příslušně upravit validační doménu. V případě, že ještě nemáme validační doménu, lze ji vytvořit ve webovém rozhraní.

1. **Exportování projektu do `JAR` souboru**
   V Eclipse v okně *Package Manager* klikneme na náš projekt pravým tlačítkem a vybereme *Export*. Ze skupiny *Java* zvolíme *JAR file* a klikneme na *Next*. Ve výsledném `JAR` souboru jsou nutné pouze `class` soubory tříd, proto můžeme soubory jako `.classpath` a `.project` odškrtnout. Důležité je, aby byla zaškrtnutá možnost *Export generated class files and resources*. Tlačítkem *Browse* vybereme, kam chceme `JAR` soubor uložit a klikneme na tlačítko *Finish*.



Obrázek 3.5: Volby při exportování do `JAR` souboru

2. **Zkopírování vyexportovaného souboru na validační server**
   Pomocí programu WinSCP se připojíme na validační server. Ve složce `data/domains` (viz 2.1.2, strana 3) najdeme název domény, ve které budeme chtít vytvořenou custom validaci použít, a do této domény nakopírujeme vyexportovaný `JAR` soubor.

3. **Úprava souboru `domain.xml`**
   Do souboru `domain.xml` ve složce domény doplníme název zkopírova-

ného `JAR` souboru jako hodnotu záznamu `domain.classpath`. Záznam tedy bude vypadat přibližně takto:

```
<entry key="domain.classpath">Ukazka.jar</entry>
```

Pokud používáme více knihoven, oddělíme je středníkem:

```
<entry key="domain.classpath">Foo.jar; Bar.jar</entry>
```

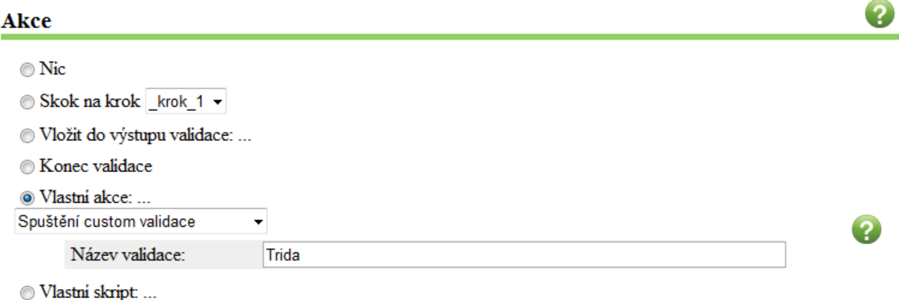
4. **Úprava souboru `run.policy`**
   Provádíme-li během custom validace zápis, musíme nastavit práva pro dočasnou pracovní složku (v dočasné pracovní složce bychom měli provádět veškeré operace spojené se soubory). Pod povolení číst ze složky domény si proto přidáme povolení zapisovat v pracovní složce:

```
permission java.io.FilePermission "\${validator.domaindir}/−", "read";
permission java.io.FilePermission "\${validator.workdir}/−", "write";
```

5. **Vytvoření kroku ve webovém rozhraní**
   Přihlásíme se do webového rozhraní na adrese:

   https://validator.kiv.zcu.cz

   a u vybrané domény klikneme na *Uprav*. Poté v sekci *Kroky validace* zvolíme *nový krok*. Zadáme název kroku a jako typ akce vybereme *Vlastní akce* → *Spuštění custom validace*. Do pole *Název validace* napíšeme celý název třídy (tj. název i s balíčky), na konci nesmí být žádná přípona. Upozorňuji, že název třídy a název `JAR` souboru jsou rozdílné věci.



Obrázek 3.6: Krok s custom validací

3.2 Externí vlastní akce

Požadavky:

- Java SE SDK 6u30,
- Eclipse SDK (Indigo SR2),
- WinSCP,
- Putty,
- přístup na validační server,
- přístup na SVN se zdrojovými kódy validačního serveru nebo samotné zdrojové kódy validačního serveru

Externí vlastní akce podporují JavaScriptové proměnné a umožňují napsat nápovědu, která se uživateli zobrazí ve webovém rozhraní. Dále nabízejí snadnější způsob, jak měnit dynamické parametry validace. Všechny vlastní akce, které jsou dostupné, se pak zobrazují ve webovém rozhraní při úpravě kroku pod volbou *Vlastní akce*. Nevýhodou vlastních akcí je fakt, že při jejich zprovozňování je třeba rekompilovat validační server (`VS.jar`) a následně pak novou verzi nahrát na server. Z mého pohledu jsou externí vlastní akce spíše vhodnější pro obecnější činnosti, které budou potencionálně využívány i jinými uživateli.

3.2.1 Vytvoření externí vlastní akce

Pro vytvoření externí vlastní akce je třeba mít knihovny `externi.jar` a `js.jar` (na přiloženém CD). Knihovna `externi.jar` obsahuje třídy z validačního serveru potřebné pro naprogramování validace a knihovna `js.jar` slouží pro práci se vstupními parametry `Context` a `Scriptable` [2].

1. **Vytvoření nového projektu**
   Viz krok číslo 1 v sekci 3.1.1 na straně 18.
2. **Pojmenování projektu a nastavení verze prostředí**
   Viz krok číslo 2 v sekci 3.1.1 na straně 18.

3. **Přidání knihoven `externi.jar` a `js.jar` do projektu**
   Analogický postup jako v kroku číslo 3 v sekci 3.1.1 na straně 18.

4. **Vytvoření třídy implementující rozhraní `VlastniAkce`**

```
public class Trida2 implements VlastniAkce {}
```

5. **Implementace nutných metod**
   Metody, které je nutné implementovat, nám vytvoří sám Eclipse po najetí kurzoru myši na název třídy a kliknutí volby *Add unimplemented methods* (podobně jako na obrázku 3.4 na stránce 20). Jsou to metody:

```
public String getHelp()
public String getId()
public String getKategorii()
public String getNazev()
public List<ParametrAkce> getParametry()
public String getPopis()
public void execute(ValidationInfo arg0, FullValidationResult arg1,
  Scriptable arg2, Context arg3, Collection<Parametr> arg4)
```

Metoda `getHelp()` vrací řetězec s nápovědou. Veškerý text nápovědy musí být v tomto jediném řetězci, přičemž text musí být napsán pomocí HTML značek.

```
public String getHelp() {
  String help = "<h4>Nadpis</h4>";
  help += "<p>Text napovedy</p>";
  return help;
}
```

Ukázka 3.2: Metoda `getHelp()`

Metoda `getId()` vrací identifikátor, podle kterého se vlastní akce odliší od ostatních. Stačí zadat jméno třídy.

```
public String getId() {
  return "Trida2";
}
```

Ukázka 3.3: Metoda `getId()`

Metoda `getKategorii()` vrací kategorii, do které vlastní akce spadá.

```
public String getId() {
  return KAT_SOUBOR;          // Prace se soubory
}
```

Ukázka 3.4: Metoda `getKategorii()`

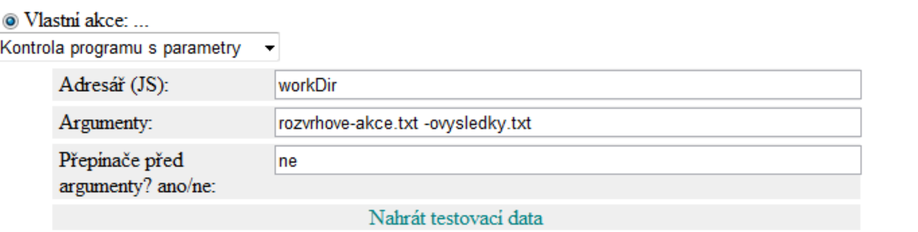
Kromě výše uvedené kategorie existuje ještě kategorie `KAT_PROGRAM` pro práci s programy a `KAT_DOC` pro práci s dokumenty.

Metoda `getNazev()` vrací název, který se bude zobrazovat se seznamu vlastních akcí.

```
public String getNazev() {
  return "Trida 2 – ukazkova akce";
}
```

Ukázka 3.5: Metoda `getNazev()`

Metoda `getParametry()` vrací seznam parametrů. Parametrem je myšleno například tlačítko na nahrání dat na server nebo různé přepínače (viz obr. 3.7) V následujícím příkladu je řešeno přidání dvou parametrů: název souboru a ovládací prvek pro nahrání dat na server. Při vytvoření objektu `ParametrAkce` se zadávají čtyři argumenty – id, název, popis a typ. Id se definuje jako třídní konstanta. Název a popis jsou informace pro zobrazení ve webovém rozhraní a typ se udává jenom ve speciálních případech.



Obrázek 3.7: Příklad parametrů vlastní akce

```
public class Trida2 implements VlastniAkce {
public static final String PARAM_NAZEV_SOUBORU = "Nazev souboru";

...

public List<ParametrAkce> getParametry() {
  ArrayList<ParametrAkce> list = new ArrayList<ParametrAkce>();
  list.add(new ParametrAkce(PARAM_NAZEV_SOUBORU, "Nazev souboru", "JavaScriptova
        promenna pro ulozeni nazvu souboru", ""));
  list.add(new ParametrAkce("kontrola-programu/nahraj", "Nahrat testovaci data", "Nahrajte
        testovaci data vytvorena podle navodu", "anchor"));
  return list;
}
```

Ukázka 3.6: Metoda `getParametry()`

Metoda `getPopis()` vrací řetězec s popisem vlastní akce.

```
public String getPopis() {
  return "Akce slouzi jako ukazka externi  vlastni  akce.";
}
```

Ukázka 3.7: Metoda `getPopis()`

Metoda `execute()` vykonává samotnou validaci. Z parametrů metody jsou pro nás nejdůležitější `ValidationInfo` (informace o validaci), `FullValidationResult` (výsledek validace) a `Scriptable` (JavaScriptové proměnné). Příklad níže vypíše do výsledku validace absolutní cestu k odevzdanému souboru.

```
public void execute(ValidationInfo info, FullValidationResult result,
  Scriptable scope, Context jsContext, Collection<Parametr> param) {

  String path = info.getInputFile().getAbsolutePath();
  result.addInfo(path);
}
```

Ukázka 3.8: Metoda `execute()`

3.2.2 Zprovoznění externí vlastní akce

Zprovoznění bude tentokrát vyžadovat znovusestavení a restart validátoru.

1. **Exportování projektu do `JAR` souboru**
   Viz krok číslo 1 v sekci 3.1.2 na straně 21

2. **Úprava souboru `MANIFEST.MF`**
   Ve složce, kam jsme vygenerovali `JAR` soubor, si vytvoříme textový soubor (např. `manifest.txt`). Tento textový soubor bude obsahovat pouze jednu řádku:

   ```
   AKCE: nazevVlastniAkce
   ```

   kde název musí být jedinečný přes všechny akce. Může se například použít název třídy implementující rozhraní `VlastniAkce`. V té samé složce si otevřeme příkazovou řádku a zadáme:

   ```
   jar umf manifest.txt nazevJARSouboru.jar
   ```

   Tento příkaz zajistí, že se do `MANIFEST.MF` doplní řádka ze souboru `manifest.txt`. Alternativně lze též použít archivovací program, otevřít náš `JAR` soubor a `MANIFEST.FM` upravit ručně.

3. **Zkopírování vyexportovaného souboru na validační server**
   Postupujeme jako v kroku číslo 2 v sekci 3.1.2 na straně 21. Tentokrát však budeme kopírovat do jiných složek. `JAR` soubor musíme zkopírovat do složkek:

   - `lib`
   - `tomcat/webapps/ROOT/WEB-INF/lib`

   Přičemž tyto cesty platí pro ostrý validační server.

4. **Úprava souboru `build.properties`**
   Soubor `build.properties` se nachází v SVN repozitáři se zdrojovými kódy validačního serveru. Tento soubor je třeba upravit tak, aby řádky `libraries` a `libraries.spaced` obsahovaly cestu k `JAR` souboru, který jsme zkopírovali do složky `lib`.

   ```
   libraries          = lib/commons-vfs-1.1.jar;...;lib/Ukazka2.jar
   libraries.spaced   = lib/commons-vfs-1.1.jar ... lib/Ukazka2.jar
   ```

5. **Spuštění `build.xml` (ant)**
   Provedeme sestavení pomocí souboru `build.xml`, který se nachází v SVN repozitáři. Do složky `out/JARs` by se nám měl vygenerovat soubor `VS.jar`.

6. **Nahrání nové verze `VS.jar` na server**
   Přes WinSCP nahrajeme `VS.jar` do kořenového adresáře validačního serveru a přepíšeme starou verzi.

7. **Restartování validačního serveru**
   Viz sekce 2.1.3 na straně 4.

8. **Restartování webového rozhraní**
   Viz sekce 2.2.2 na straně 6.

3.3 Zásuvné moduly pro KIV/OOP

3.3.1 Porovnání dvou obrázků

Pro vytvoření tohoto modulu stačila jediná třída. Modul navazuje na vlastní akci *Kontrola programu OOP*, která pro každý soubor (z odevzdávaného `JAR` souboru), jehož název končí slovem `Test`, spouští JUnit testy a generuje při tom snímek obrazovky. Tento snímek pojmenovaný `screenshot.png` ukládá do pracovní složky validace.

Metoda `validate()` načte vygenerovaný obrázek `screenshot.png` a vzorový obrázek `vzor.png`, který musíme nahrát společně s testovacími daty. Oba tyto obrázky jsou pak převedeny na pole pixelů.

```
public class PorovnatPNG implements VlastniAkce {
  public void execute(ValidationInfo info, FullValidationResult result,
          Scriptable scope, Context jsContext, Collection<Parametr> parametry) {

   ... // Kontrola existence potrebnych souboru

  String sampleImage = info.getDomain().getDomainProperties().getProperty(
      "image.sample_name");    // Vlastnost, ktera se musi dopsat do domain.xml
                               // rika, jak se ma jmenovat vzorovy soubor

  File validationImage = new File(workDir, sampleImage);

   ...
  // Nacteni obrazku
  sample = ImageIO.read(validationImage);

   ...
  // Prevedeni obrazku na pixely
  PixelGrabber grab1 = new PixelGrabber(sample, 0, 0, -1, -1, false);

   ...
  // Vytvoreni pole s pixely
```

```
int []  data1 = null;
 ...
int width = grab1.getWidth();
int height = grab1.getHeight();
data1 = new int[width * height];
data1 = (int[])grab1.getPixels();
```

Pixely jsou uložené jako `int` hodnota ve tvaru `0xAARRGGBB`, kde `AA` je alpha složka barvy a zbytek jsou RGB složky.

Pokud jsou pole totožná, nemusí se nic řešit a přidáme pouze informativní zprávu, že je výstup v pořádku. Pro porovnávání jsem použil metodu `Arrays.equals()`, která vrací `true`, pokud jsou obě pole stejně velká a každý prvek prvního pole je stejný jako odpovídající prvek druhého pole.

V případě, že pole stejná nejsou, musí se cyklem projít všechny prvky. Aby bylo možné studentovi ukázat, kde přesně se jeho obrázek liší od vzorového, je nutné vytvořit jakýsi rozdílový obrázek. Rozdílový obrázek se vytvoří zesvětlením pixelů, které jsou stejné v obou polích, a pixelem s červenou barvou v místě, kde se vytvořený obrázek liší od vzorového.

```
if (data1[i + j*width] == data2[i + j*width]) {
  pixel = data1[i + j*width];
  diffData[i + j*width] = brighten(pixel);
}
else {
  diffData[i + j*width] = markDifference();
}
```

Zesvětlení RGB hodnoty se ukázalo být mírně obtížnější, protože RGB model se zesvětlováním nebo ztmavováním bez změny barevného odstínu nepočítá. Z RGB modelu proto musíme vytvořit HSB (Hue, Saturation, Brightness) model a změnit požadované složky.

```
float []  hsbVals = Color.RGBtoHSB(red, green, blue, null);
// Zmena saturace a svetlosti
Color brighter = Color.getHSBColor(hsbVals[0], 0.2f * hsbVals[1], 0.5f * (1f + hsbVals[2]));
```

Po vygenerování rozdílového obrázku jej uložíme a překopírujeme z pracovní složky na server s výsledky validací (vs.kiv.zcu.cz/validator). Při kopírování musíme dbát na to, aby snímek obrazovky i rozdílový obrázek měly unikátní název, protože všechny obrázky se ukládají do jedné složky pro celou validační doménu. K tomu použijeme metodu `System.currentTimeMillis()`.

```
String destinationPath = info.getDomain().getDomainProperties().getProperty(
    "image.destination_path");        // Musi se dopsat do domain.xml
String diffImage = String.valueOf("Diff_" +System.currentTimeMillis()) + ".png";
ImageIO.write(bufferedImage, "png", new File(destinationPath + File.separator + diffImage));
```

Nakonec ještě vložíme odkaz na obrázek do výstupu validace.

```
result.addInfo("<a href=\"" +imageUrl +"/" +diffImage +"\"\>Rozdily oproti spravnemu obrazku
    .</a>");
```

Celý zdrojový kód je v příloze B.

3.3.2 Použití custom validace

V jedné z úloh studenti mají za úkol připravit sadu testovacích případů, které odhalí problémy v jim poskytnuté knihovně, konkrétně v souboru `Osoba.java`. Odevzdává se jediný soubor `OsobaTest.java`. Dále následujou okomentované části zdrojového kódu custom validace. Celý zdrojový kód je příloze A.

```
public class ValidationOOP11 implements CustomValidation {

  final String JUNIT_JAR = "/home/valtest/val-test/lib/junit-4.8.2.jar";
// final String JUNIT_JAR = "/home/validator/validator/lib/junit-4.8.2.jar";
```

Cesta k JUnit knihovně je dána napevno v kódu, protože jsem nepřišel na to, jak zjistit tuto cestu nezávisle na struktuře serveru. Při použití na ostrém serveru bude proto třeba přehodit cesty.

Abychom mohli testy spustit, musíme nejdříve `OsobaTest.java` zkompilovat, abychom získali `OsobaTest.class`. Pokud se podaří soubor zkompilovat, přesuneme jej do podsložky `testy`. Tato podsložka je nutná z důvodu použití balíčku. Do classpath pak přidáme celou pracovní složku.

```
JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();
compilationResult = compiler.run(null, null, errorOutput, "-classpath", JUNIT_JAR +":" +
    osobaPath, "-encoding", "UTF-8", inputFile.getAbsolutePath());

...
File classFile = new File(testyDir.getAbsolutePath() + File.separator + "OsobaTest.class");
File oldClassFile = new File(inputFile.getAbsolutePath().replace("java", "class"));
oldClassFile.renameTo(classFile);
```

Spustí se JUnit testy z `OsobaTest` a všechny by měly odhalit konkrétní chyby. Chyby (respektive jejich popis) musí být identické se vzorovými, které studenti mají v zadání.

```
for (Failure fail: result.getFailures()) {
  String chyba = fail.getDescription().getMethodName() + ": " +
      fail.getMessage();
  if (mnozinaHlaseni.remove(chyba) == true)
  {
    fullValidationResult.addInfo("Nalezeno: " +chyba);
  }
```

Aby byl tedy odevzdaný soubor `OsobaTest.java` považován za správný, musí být nalezeny všechny chyby.

3.4 Knihovna pro kontrolu UML diagramů

Tuto knihovnu jsem vytvořil pro použití v custom validaci. Celá knihovna se skládá z celkem devíti tříd. Čtyři třídy jsou v balíčku `data`, který zajišťuje vytvoření objektů z `XML` souboru, a zbylých pět tříd je v balíčku `logic`, který pracuje s vytvořenými objekty v paměti.

```
[data]
  Handler
  IUMLReader
  UMLElement
  UMLReaderSax

[logic]
  UMLClass
  UMLDiagram
  UMLRelation
  UMLRelationType
  UMLUtils
```

Třída `UMLReaderSax` implementuje rozhraní `IUMLReader` a slouží k načtení všech prvků UMLet souboru (přípona `uxf`) do paměti. Jak už z názvu vyplývá, je použita technologie SAX (viz 2.4.3). `UMLReaderSax` využívá pro samotné parsování třídu `Handler`, která přesně říká, jakým způsobem se mají získat hodnoty z `XML` souboru. Pro každý zpracovaný prvek `element` se vytvoří nový objekt v Javě typu `UMLElement`, což je v podstatě reprezentace prvku `element` v paměti. Seznam všech `UMLElement` objektů se pak předává dál metodou `getAllElements()`.

Nejpodstatnější třídou je třída `UMLDiagram`, která z objektů `UMLElement` vytváří objekty diagramu a poskytuje nad nimi dotazy. Při vytváření objektů `UMLClass` a `UMLRelation` se musí z `UMLElement` nějakým způsobem získat, o jaký typ třídy nebo relace se jedná. Potíž je v tom, že klasifikace je pouze textová a vytváří ji autor diagramu.

Zjišťování typu třídy (třída, rozhraní, abstraktní třída atd., případně návrhové vzory) je tedy napevno podle výskytu nějakého znaku nebo řádky. Z ukázkového UML diagramu, který jsem měl k dispozici, jsem vypozoroval čtyři možnosti, jak může být třída popsána.

- pouze název třídy,
- název třídy, barva obdélníku,
- typ třídy uvnitř znaků „<<“ a „>>“, název třídy,
- typ třídy uvnitř znaků „<<“ a „>>“, název třídy, barva obdélníku.

```
<panel_attributes>&lt;&lt;singleton&gt;&gt;
Seznamka
bg=red</panel_attributes>
```

Ukázka 3.9: Třída typu `singleton`; názvem `Seznamka` a červenou barvou

Samotná klasifikace probíhá tak, že kontroluji, zda se v popisu vyskytuje řetězec „>>“ nebo znak pro novou řádku. Tato metoda je silně závislá na tom, že studenti budou mít jasně stanovené stejné značení a budou se ho držet. Dále je nutné si uvědomit, že i typ třídy není omezen a může být jakýkoliv.

Relace jsou na tom podobně. Tvar šipek se autoři UMLet rozhodli ukládat jako první řádku v popisu elementu. Viz ukázka 2.2 na straně 10. Pro určení typu relace musí být typ spojený s tvarem šipky. Komplikace ale opět nastávají, pokud se autor diagramu rozhodne dát popis relace nad/pod šipku a na tvar šipky nebude dbát. Tyto situace neřeším, protože by to bylo zbytečně komplikované. Klasifikace relací je tedy pouze na základě tvaru šipky (viz tabulka 3.1).

Zda je relace mezi konkrétními dvěma třídami, se stanoví prohledáním všech tříd a kontrolou, zda počáteční a koncový bod relace leží uvnitř obdélníku některé z tříd. Všechny objekty typu `UMLRelation` proto mají atributy

Značení	**Typ**	**Podoba**
`<-` nebo `-`	asociace	
`<.`	závislost	
`<<-`	dědění	
`<<.`	implementace rozhraní	
`<<<-`	agregace	
`<<<<-`	kompozice	

Tabulka 3.1: Druhy relací v UMLet

`from` a `to` typu `UMLClass`. Pokud je nějaká relace obousměrná, pak se musí vytvořit dvě relace, které budou mít prohozené `from` a `to`.

V momentální podobě se dotazy vykonávají nad dvěma seznamy – seznam s `UMLClass` a `UMLRelation`. Seznamy nejsou nijak logicky propojené, protože mě nenapadl případ, kde by to bylo potřebné. Vzhledem k tomu, že modul ještě nebyl vyzkoušen pro vytvoření validace a není k dispozici žádná zpětná vazba, lze očekávat, že modul se bude v budoucnu ještě upravovat.

K dispozici jsou nyní tyto dotazy:

```
public boolean noteExists()
public boolean classExists(String name)
public boolean interfaceExists(String name)
public int checkNumberOfRelationsOf(String name)
public boolean checkRelationFromTo(String from, String to)
public boolean checkRelationTypeFromTo(UMLRelationType type, String from, String to)
```

Použití je pak takové, že v třídě pro custom validaci se vytvoří objekt `UML-Diagram`, kterému se jako parametr předá cesta k odevzdávanému diagramu a objekt typu `FullValidationResult`. Aby autor validace nemusel stále psát podmínky pro porovnávání očekávané a navrácené hodnoty, vytvořil jsem v třídě `UMLUtils` statickou metodu `assertEquals()`, která porovnává předané

hodnoty a v případě nerovnosti přidá do výsledku validace chybu s textem `errorMessage`:

```
public static void assertEquals(String errorMessage, boolean expected, boolean returned)
```

Pro příklad 2.3 ze strany 10 mohou testy vypadat takto:

```
import java.io.File;

import logic.UMLDiagram;
import logic.UMLRelationType;
import logic.UMLUtils;

import cz.zcu.validationserver.CustomValidation;
import cz.zcu.validationserver.validation.FullValidationResult;
import cz.zcu.validationserver.validation.ValidationInfo;


public class Test implements CustomValidation {

  @Override
  public void validate(ValidationInfo validationInfo, FullValidationResult
      fullValidationResult) {

    File inputFile = validationInfo.getInputFile();
    UMLDiagram diagram = new UMLDiagram(inputFile.getAbsolutePath(), fullValidationResult);

    UMLUtils.assertEquals("Trida 'Osoba' neexistuje.", true,  diagram.classExists("Osoba"));
    UMLUtils.assertEquals("Trida 'Muz' neexistuje.", true, diagram.classExists("Muz"));
    UMLUtils.assertEquals("Mezi tridou 'Muz' a 'Osoba' neni dedicnost", true,
        diagram.checkRelationTypeFromTo(UMLRelationType.INHERITANCE, "Muz", "Osoba"));
  }
}
```

Ukázka 3.10: Použití `assertEquals()`

4 Validační domény pro KIV/OOP

Všechny vytvořené domény byly úspěšně otestovány na odevzdaných úlohách z let 2011/2012 a 2010/2011.

4.1 Klíčové úlohy

Validační domény pro klíčové úlohy mají téměř stejný obsah. Na začátku je vždy akce *Rozbal ZIP*. U této akce musí být ještě vyplněný parametr *Cílový adresář*, který bude použit v další fázi při kontrole existence konkrétních souborů (obr. 4.2).

Kroky validace

1.	rozbaleni	Uprav	Zruš
2.	hledej_doc	Uprav	Zruš
3.	nenalezeno_doc	Uprav	Zruš
4.	hledej_rozmer	Uprav	Zruš
5.	nenalezeno_rozmer	Uprav	Zruš
6.	hledej_osoba	Uprav	Zruš
7.	nenalezeno_osoba	Uprav	Zruš
8.	hledej_pohlavi	Uprav	Zruš
9.	nenalezeno_pohlavi	Uprav	Zruš
10.	konec_chyba	Uprav	Zruš
11.	spusteni_junit	Uprav	Zruš

nový krok

Obrázek 4.1: Kroky pro validační doménu úlohy 04

Obrázek 4.2: Detail kroku `rozbaleni`

Následuje kontrola existence povinných souborů a složek. Každá klíčová úloha má jiné povinné soubory a typicky souborů s číslem úlohy přibývá. Nevytvářel jsem kontrolu existence pro úplně všechny soubory, pouze ty nejdůležitější, protože mi přišlo nepřehledné mít ve validační doméně desítky

kroků (jsou to soubory, které mají studenti jednoznačně uvedené v zadání). Pro vyhledávání souborů je použita akce *Najdi soubory* (obr. 4.3), u které je třeba uvést v jaké složce se má hledat (výše zmiňovaný *Cílový adresář*), co se má hledat (regulární výraz) a JavaScriptovou proměnnou, do které bude uložen počet nalezených souborů/složek. Pokud akce žádný vyhovující soubor nenajde, bude tato JavaScriptová proměnná rovna nule.

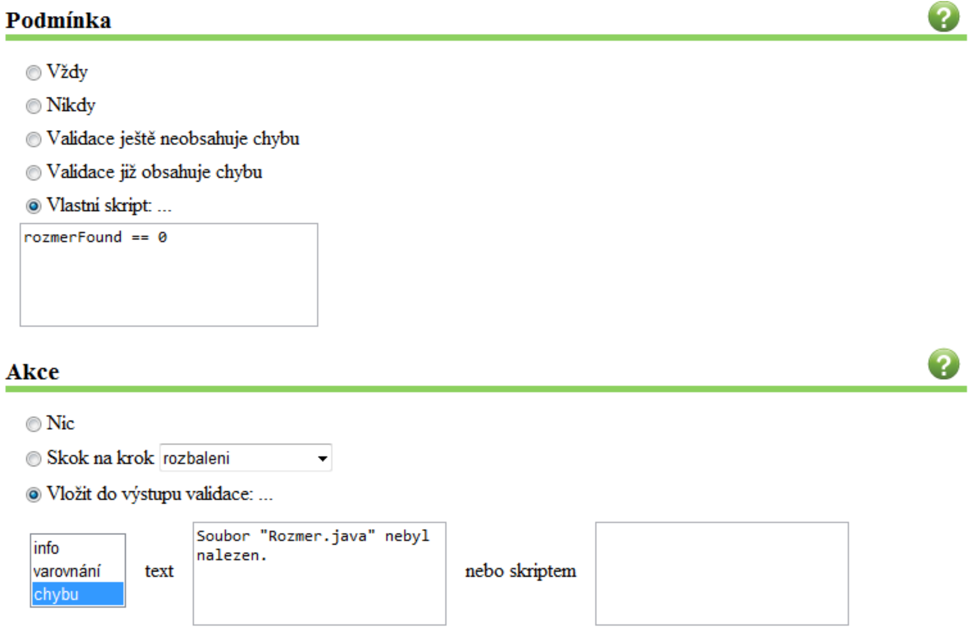
Vlastní akce: ...
Najdi soubory

Rodičivský adresář (JS):	workDir
Filtr - regex:	Rozmer.java
I v podadresářich? ano/ne:	
Nalezené soubory (JS):	
Počet souborů (JS):	rozmerFound
Velikost souborů (JS):	

Vlastní skript: ...

Obrázek 4.3: Detail kroku `hledej_rozmer`

Další krok je vložení chyby do výsledku validace, pokud nebude soubor či složka nalezena (obr. 4.4). Tyto dva kroky, hledání souboru a případné vložení chyby při nenalezení, se opakují pro všechny povinné soubory/složky.



Obrázek 4.4: Detail kroku `nenalezeno_rozmer`

Na konci fáze hledání povinných souborů/složek je pak krok, který ukončí validaci, v případě, že se ve výsledku validace již vyskytuje chyba (obr. 4.5).

Validace již obsahuje chybu
Vlastní skript: ...

Akce

Nic
Skok na krok rozbaleni
Vložit do výstupu validace: ...
Konec validace

Obrázek 4.5: Detail kroku `konec_chyba`

Posledním krokem je akce *Kontrola programu OOP*. Zde je nutné vysvětlit, jak se nahrávají testovací data. Testovací data můžeme nahrát buď přes webové rozhraní tlačítkem *Nahrát testovací data* nebo přes WinSCP.

Přes webové rozhraní se testovací data vždy nahrávají jako `zip` archiv (koncovka musí být malými písmeny). Archiv se rozbalí do složky `validation_data` v dané doméně na serveru a pokud nebude archiv obsahovat složky, rozbalí se do `validation_data` rovnou. Akce *Kontrola programu OOP* byla původně vytvořena tak, že se počítalo s jednou validační doménou pro všechny úlohy a nahráním všech testovacích dat najednou, neboli že složka `validation_data` obsahovala podsložky pojmenované číslem úlohy. Když tedy chceme mít pro každou úlohu vlastní validační doménu, musíme mít ve složce `validation_data` podsložku s číslem úlohy a až v ní testovací data pro danou doménu. Při nahrání archivu, který obsahuje zabalenou složku, bychom ale zjistili, že se do `validation_data` rozbalily pouze soubory a složka se nevytvořila. Je proto zapotřebí zabalit do archivu složku dvakrát vnořeně.

```
01.zip
  [01]
    [01]
      Barva.java
      Elipsa.java
       ...
```

Použijeme-li WinSCP, tento problém nemusíme řesit. Připojíme se na validační server, vytvoříme složku `validation_data`, v ní vytvoříme složku s číslem úlohy a překopírujeme do ní testovací data.

U první úlohy je navíc krok s porovnáváním vygenerovaného snímku obrazovky se vzorovým snímkem. Akce nemá žádné parametry, ale je třeba počítat s tím, že vzorový obrázek musí být ve složce `validation_data` a do `domain.xml` se musí dopsat název vzorového obrázku:

```
<entry key="image.sample_name">vzor.png</entry>
```

První úloha má ještě jednu zvláštnost. I když se neodevzdává archiv ale soubor `OsobaTest.java`, přesto je prvním krokem rozbalení. Je to z důvodu, aby bylo možno použít akci *Najdi soubory*, která vyžaduje uvést adresář, ve kterém se má hledat. To má za následek, že se při odevzdávání vypíše, že se nejedná o archiv a celá validace skončí. Aby validace pokračovala, musí se použít trik. Do souboru `nl/text.properties` ve složce domény se dopíše vlastnost `file.is_not_archive` a její hodnota se ponechá prázdná.

```
file.is_not_archive =
```

Klíčové úlohy používají svůj `run.policy`, ve kterém jsou práva pro knihovny a pro vytváření snímků (jsou k dispozici na přiloženém CD).

V implicitním `domain.xml`, který se vytvoří s novou doménou, se většinou akorát musí změnit položky `html_output_dir`, `html_output_url` a `accept_extensions`, popřípadě ještě `domain.classpath`. Položky `html_output_dir` a `html_output_url` určují, kam se budou generovat výstupní HTML soubory.

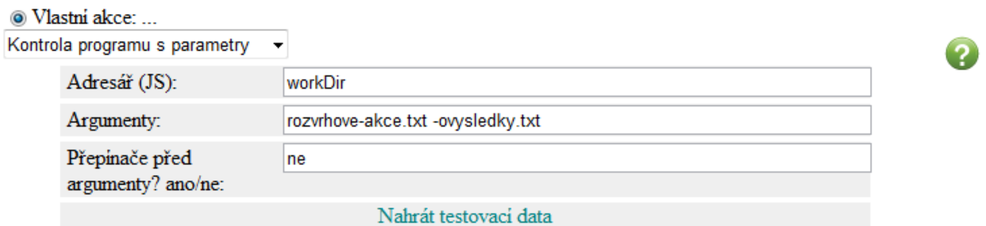
```
<entry key="html_output_url">/var/www/validator/val-test/oop01</entry>
<entry key="html_output_dir">/var/www/validator/val-test/oop01</entry>
```

Pokud bychom se chtěli přes webový prohlížeč kouknout do složky, kam se generují HTML soubory, stačí nahradit `/var/www/` za `vs.kiv.zcu.cz/`.

4.2 Úlohy s kolekcemi

Úlohy s kolekcemi maji podobné kroky jako klíčové úlohy, pouze místo poslední akce *Kontrola programu OOP* mají *Kontrola programu s parametry* (obr. 4.6). Studenti odevzdávají `JAR` soubor, který jde spustit a který potře-

buje zadat argumenty. Do pole *Argumenty* této akce se zapisují argumenty, které bychom bežně zadávali do příkazové řadky při spuštění. Pouze před argument, který označuje výstupní soubor, se musí napsat `-o`, aby se tento výstupní soubor dal vytvořit.



Obrázek 4.6: Vlastní akce *Kontrola programu s parametry*

Testovací data pro tuto akci musí mít ve `validation_data` dvě složky: `input_data` a `vysledky`. Do `input_data` se dávají soubory, které budeme potřebovat jako vstup, a do `vysledky` se dá vzorový textový soubor, vůči kterému budeme porovnávat výstup studentovy úlohy. Porovnání se provede metodou `compareFiles()` třídy `StreamComparator`.

Dojde-li k nějaké chybě při spouštění programu, může se stát, že ve výsledku validace bude chyba, ale nebudou uvedeny žádné další podrobnosti. V této situaci je vhodné podívat se do dočasného pracovního adresáře, kam vlastní akce *Kontrola programu s parametry* vygenerovává chybový textový soubor.

4.3 JUnit úloha

Jedenáctá úloha má pouze jeden krok. V tomto kroku se použije akce *Spuštění custom validace* pro spuštění `ValidationOOP11`. Viz sekce 3.3.2 na straně 30.

5 Závěr

Celá práce měla dva hlavní cíle. Prvním z nich bylo využití již existujícího webového rozhraní pro uživatelsky příjemnou konfiguraci validačních domén (jeden předmět typicky obsahuje několik validačních domén). Popisovaný úkol byl proveden na 11 doménách pro předmět KIV/OOP. Toto lze považovat za úspěšný pilotní projekt reálného nasazení webového konfigurátoru. Celý postup konfigurace domén byl důsledně popsán, takže se podle něj budou moci v budoucnu příslušní vyučující jiných předmětů řídit. Protože existuje konfigurace 11 domén, lze tento úkol považovat za splněný.

Druhý hlavní úkol představoval vytvoření tří kompletně nových zásuvných modulů. Tyto moduly umožňují principiálně zcela odlišné způsoby validace studentských úloh. Dříve bylo možné porovnávat jen textový výstup, v současné době mnou vytvořené moduly umožňují porovnávat grafický výstup a též kontrolu UML diagramů. Navíc byl splněn požadavek na ověření možnosti libovolné programátorské kontroly, což bylo třetím zmíněným modulem splněno. Všechny tři zásuvné moduly jsou zcela funkční a jejich zdrojový kód odpovídá současným požadavkům (dokumentace apod.). Všechny tři zmíněné moduly jsou již součástí konfigurace výše uvedených validačních domén.

V dokumentu je také navíc popsáno, jak rozšířit pravidla pro statickou kontrolu zdrojového kódu pomocí PMD. Kontrola pomocí PMD se tak bude moci stát volitelnou součástí validačního procesu.

Všechny body zadání byly splněny a validační domény jsou připraveny k nasazení.

Celou práci na této problematice však nepovažuji za dokončenou. Hodlám zde v dalším období dále spolupracovat a to v rámci probíhajícího rozvojového projektu, na jehož řešení se podílím. Lze totiž předpokládat, že v budoucnu – při nasazení v ostrém provozu – vzniknou požadavky na další rozšiřování funkčnosti jak zásuvných modulů, tak i samotných validačních domén.

Seznam zkratek

AST Abstract Syntax Tree - stromová reprezentace syntaktické struktury zdrojového kódu

DOM Document Object Model - multiplatformní a jazykově nezávislé rozhraní pro navigaci v XML dokumentech

HTML Hypertext Markup Language - značkovací jazyk webových stránek

HTTP Hypertext Transfer Protocol - internetový protokol

JPEG Joint Photographic Experts Group - formát pro ukládání bitmapových obrázků ztrátovou kompresí

PNG Portable Network Graphics - formát pro ukládání bitmapových obrázku bezztrátovou kompresí

SAX Simple API for XML - jednoduché rozhraní pro parsování XML dokumentů

SSH Secure Shell - bezpečný síťový protokol

SVN Apache Subversion - systém pro správu a verzování zdrojových kódů

UML Unified Modeling Languange - standardizovaný jazyk pro modelování programových systémů

XML Extensible Markup Language - značkovací jazyk

Literatura

[1] BECK, K.: *Test Driven Development: By Example.* Addison-Wesley Professional, 2003, ISBN 0321146530.

[2] DUDOVÁ, V.: *Webová konfigurace validačního serveru.* Bakalářská práce, Západočeská univerzita v Plzni, Fakulta aplikovaných věd, 2010.

[3] HYNAR, M.: *Java nástroje.* Neocortex s.r.o., 2004, ISBN 08-86330-16-8.

[4] MCLAUGHLIN, B.: *Java & XML.* O'Reilly, 2001, ISBN 0-596-00197-5.

[5] SCHMULLER, J.: *Myslíme v jazyku UML.* Grada publishing, 2001, ISBN 80-247-0029-8.

Webové zdroje

[6] *XML Path Language (XPath) 2.0 (Second Edition)* [Online]. Poslední aktualizace v lednu 2011 [cit. 6. května 2012], dostupné z: http://www.w3.org/TR/xpath20/.

[7] CLARK, M.: *JUnit FAQ* [Online]. Poslední aktualizace v únoru 2006 [cit. 6. května 2012], dostupné z: http://junit.sourceforge.net/doc/faq/faq.htm.

[8] JENSEN, J.: *PMD - Welcome to PMD* [Online]. Poslední aktualizace v lednu 2012 [cit. 6. května 2012], dostupné z: http://pmd.sourceforge.net/.

[9] MEGGINSON, D.: *Megginson Technologies: Simple API for XML* [Online]. ©2012 [cit. 6. května 2012], dostupné z: http://www.megginson.com/downloads/SAX/.

[10] VALENTA, L.: *Validační server* [Online]. Publikováno v září 2007 [cit. 6. května 2012], dostupné z: http://vs.kiv.zcu.cz/doc/index.html.

[11] VALENTA, L.: *Podpora výuky - Uživatelská příručka* [Online]. Poslední aktualizace v červnu 2008 [cit. 6. května 2012], dostupné z: http://portal.zcu.cz/zdroje/podporavyuky/help/.

A Ukázka custom validace

```
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.OutputStream;
import java.net.MalformedURLException;
import java.net.URL;
import java.net.URLClassLoader;
import java.util.Arrays;
import java.util.HashSet;
import java.util.Set;

import javax.tools.JavaCompiler;
import javax.tools.ToolProvider;

import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

import cz.zcu.validationserver.CustomValidation;
import cz.zcu.validationserver.ValidationResult;
import cz.zcu.validationserver.validation.FullValidationResult;
import cz.zcu.validationserver.validation.ValidationInfo;

/**
 * Trida pro kontrolu 11.ulohy z OOP pomoci CustomValidation.
 *
 * @author Duong Manh Hung, zer0@students.zcu.cz
 */
public class ValidationOOP11 implements CustomValidation {

  // NUTNO ZMENIT na ostrem validatoru !!!
  final String JUNIT_JAR = "/home/validator/validator/lib/junit-4.8.2.jar";

  // final String JUNIT_JAR = "/home/valtest/val-test/lib/junit-4.8.2.jar";

  static String[] hlaseni = {
    "testSetJmeno: Chybne jmeno: expected:<K[]arel> but was:<K[A]arel>",
    "testSetVyskaDolniRozsah: Chybna vyska: expected:<50> but was:<51>",
    "testSetVyskaHorniRozsah: Chybna vyska: expected:<250> but was:<251>",
    "testSetVahaVyjimka: Unexpected exception, expected<java.lang.IllegalArgumentException> but
         was<java.lang.IllegalAccessError>",
    "testSetVahaDolu: Chybna vaha: expected:<91.0> but was:<92.0>"
    };

  @Override
  public void validate(ValidationInfo validationInfo, FullValidationResult
       fullValidationResult) {

    String osobaPath = validationInfo.getDomain().getDomainDir().getAbsolutePath() + File.
         separator + "Osoba.jar";
    File errorFile = new java.io.File(validationInfo.getWorkDir(), "error.txt");
    File inputFile = validationInfo.getInputFile();
    OutputStream errorOutput = null;
    try {
      errorOutput = new FileOutputStream(errorFile);
    } catch (FileNotFoundException e1) {
      // TODO Auto-generated catch block
```

```
  e1.printStackTrace();
}

if (!"OsobaTest.java".equals(inputFile.getName())) {
  // vlozeni chyby do vysledku
  fullValidationResult.addError("Soubor se nejmenuje OsobaTest.java!",
      ValidationResult.VR_INVALID_INPUT);
  return;
}

if (inputFile.length() == 0) {
  // vlozim varovani do vysledku
  fullValidationResult.addWarning("Odevzdany soubor je prazdny!");
  return;
}

JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();

// Zkompilovani poslaneho souboru
// od 4.argumentu metody run jsou argumenty pro zkompilovani
int compilationResult = -1;
if (errorOutput != null) {
  compilationResult = compiler.run(
      null, null, errorOutput, "-classpath", JUNIT_JAR +":" +osobaPath, "-encoding", "UTF-8",
          inputFile.getAbsolutePath());
}
else {
  compilationResult = compiler.run(
      null, null, null, "-classpath", JUNIT_JAR +":" +osobaPath, "-encoding", "UTF-8",
          inputFile.getAbsolutePath());
}
if (compilationResult != 0) {
  fullValidationResult.addError("Chyba pri kompilaci souboru OsobaTest.java");
}

// Vsechny cinnosti se souborem jsou na serveru provadeny v tmp slozce ve workDir
String workDirPath = inputFile.getAbsolutePath().replace(inputFile.getName(), "");

File testyDir = new File(workDirPath + "testy");
testyDir.mkdir();

// Presunuti zkompilovaneho .class souboru do adresare testy (kvuli classpath testy.OsobaTest)
File classFile = new File(testyDir.getAbsolutePath() + File.separator + "OsobaTest.class");
File oldClassFile = new File(inputFile.getAbsolutePath().replace("java", "class"));
oldClassFile.renameTo(classFile);

// Nahrani .class souboru do classpath
File workDir = new File(workDirPath);
File osobaJar = new File(osobaPath);
URL url1 = null;
URL url2 = null;
try {
  url1 = workDir.toURI().toURL();
  url2 = osobaJar.toURI().toURL();
} catch (MalformedURLException e) {
  // TODO Auto-generated catch block
  e.printStackTrace();
  return;
}
URL[] urls = new URL[]{url1, url2};

ClassLoader cl = new URLClassLoader(urls);
```

```
    Class<?> cls = null;
    try {
      cls = cl.loadClass("testy.OsobaTest");
    } catch (ClassNotFoundException e) {
      fullValidationResult.addError("Trida nebyla nalezena");
      e.printStackTrace();
      return;
    }

    Set<String> mnozinaHlaseni =
        new HashSet<String>(Arrays.asList(hlaseni));

    // Spusteni tridy pod JUnit
    Result result = org.junit.runner.JUnitCore.runClasses(cls);

    fullValidationResult.addInfo("Probehlo celkem: " +result.getRunCount() +" testu.");

    for (Failure fail: result.getFailures()) {
      String chyba = fail.getDescription().getMethodName() + ": "
          + fail.getMessage();

      String tmp = chyba.replace("<", "&lt;");
      tmp = tmp.replace(">", "&gt;");
      if (mnozinaHlaseni.remove(chyba) == true)
      {
        fullValidationResult.addInfo("Nalezeno: " +tmp);
      }
    }

    int pocet = hlaseni.length - mnozinaHlaseni.size();
    if (pocet < hlaseni.length) {
      fullValidationResult.addError("Odhaleno pouze " + pocet + " chyb z " + hlaseni.length +".
           Zbyvaji chyby:");
      System.out.println();
      for (String hlaseni : mnozinaHlaseni) {
        // Nahrazeni <> entitami pro zobrazeni v HTML
        String resultMessage = hlaseni.replace("<", "&lt;");
        resultMessage = resultMessage.replace(">", "&gt;");
        fullValidationResult.addError(resultMessage);
      }
    }
    else {
      fullValidationResult.addInfo("Odhaleny vsechny chyby.");
      return;
    }
  }

}
```

B Ukázka externí vlastní akce

```
import java.awt.Color;
import java.awt.Graphics2D;
import java.awt.Image;
import java.awt.Toolkit;
import java.awt.image.BufferedImage;
import java.awt.image.MemoryImageSource;
import java.awt.image.PixelGrabber;
import java.io.File;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collection;
import java.util.List;

import javax.imageio.ImageIO;

import org.mozilla.javascript.Context;
import org.mozilla.javascript.Scriptable;

import cz.zcu.validationserver.gui.actions.ParametrAkce;
import cz.zcu.validationserver.gui.actions.VlastniAkce;
import cz.zcu.validationserver.gui.data.Parametr;
import cz.zcu.validationserver.validation.FullValidationResult;
import cz.zcu.validationserver.validation.ValidationInfo;

/**
 * Vlastni akce do weboveho rozhrani validatoru
 * Porovnava zdrojovy PNG obrazek (nahrany v domene) s prvnim PNG obrazkem
 * v pracovnim adresari momentalni validace.
 * (Pokud se odevzdava .jar, predpoklada se, ze se v drivejsim kroku rozbali)
 * Vytvoreno pro predmet KIV/OOP
 *
 * @author Duong Manh Hung, zer0@students.zcu.cz
 */
public class PorovnatPNG implements VlastniAkce {

  public void execute(ValidationInfo info, FullValidationResult result,
      Scriptable scope, Context jsContext, Collection<Parametr> parametry) {

    File workDir = info.getWorkDir();
    String sampleImage = info.getDomain().getDomainProperties().getProperty("image.sample_name"
        );

    if (sampleImage.isEmpty()) {
      result.addError("Neni definovana vlastnost image.sample_name.");
    }

    // Existuje vzorovy obrazek?
    File validationImage = new File(workDir, sampleImage);

    if (!validationImage.exists()) {
      // vzorovy obrazek neexistuje!
      result.addError("Vzorovy áobrzek pro validaci nebyl nalezen.");
      return;
    }
```

```
File screenshot = new File(workDir, "screenshot.png");

if (!screenshot.exists()) {
  // png obrazek neni v odevzdavanem jaru
  result.addError("Nebyl nalezen vygenerovay screenshot.");
  return;
}

Image sample = null;
try {
  sample = ImageIO.read(validationImage);
} catch (IOException e) {
  result.addError("Pri nacitani vzoroveho obrazku nastala chyba.");
  return;
}

Image comparedImage = null;
try {
  comparedImage = ImageIO.read(screenshot);
} catch (IOException e) {
  result.addError("Pri nacitani odevzdavaneho obrazku nastala chyba.");
  return;
}

PixelGrabber grab1 = new PixelGrabber(sample, 0, 0, -1, -1, false);
PixelGrabber grab2 = new PixelGrabber(comparedImage, 0, 0, -1, -1, false);

// Prevod obrazku na pole pixelu
int[] data1 = null;
int[] data2 = null;
try {
  if (grab1.grabPixels()) {
    int width = grab1.getWidth();
    int height = grab1.getHeight();

    data1 = new int[width * height];
    data1 = (int[])grab1.getPixels();
  }
} catch (InterruptedException e) {
  e.printStackTrace();
  return;
}

try {
  if (grab2.grabPixels()) {
    int width = grab2.getWidth();
    int height = grab2.getHeight();

    data2 = new int[width * height];
    data2 = (int[])grab2.getPixels();
  }
} catch (InterruptedException e) {
  e.printStackTrace();
  return;
}

if (Arrays.equals(data1, data2)) {
  result.addInfo("Graficky vystup je v poradku.");
}
else {
  result.addError("Graficky vystup neodpovida zadani.");
```

```
    int width = grab1.getWidth();
    int height = grab1.getHeight();
    int i = 0, j = 0;
    int [] diffData = new int[width * height];
    int pixel;

    for (j = 0; j < height; j++) {
      for (i = 0; i < width; i++) {

        if (data1[i + j*width] == data2[i + j*width]) {
          pixel = data1[i + j*width];
          diffData[i + j*width] = brighten(pixel);
        }
        else {
          diffData[i + j*width] = markDifference();
        }

      }
    }

    MemoryImageSource mis = new MemoryImageSource(width, height, diffData, 0, width);
    Toolkit tk = Toolkit.getDefaultToolkit();
    Image img = tk.createImage(mis);

    BufferedImage bufferedImage = new BufferedImage(img.getWidth(null),
        img.getHeight(null), BufferedImage.TYPE_INT_ARGB);

    Graphics2D g2 = bufferedImage.createGraphics();
    g2.drawImage(img, null, null);

    String destinationPath = info.getDomain().getDomainProperties().getProperty("
          html_output_dir");
    destinationPath += "/screenshots";
    File imagesFolder = new File(destinationPath);
    imagesFolder.mkdir();

    String diffImage = String.valueOf("Diff_" +System.currentTimeMillis()) + ".png";
    try {
      ImageIO.write(bufferedImage, "png", new File(destinationPath + File.separator +
            diffImage));
    } catch (IOException e) {
      result.addError("Obrazek s rozdily se nepodarilo ulozit.");
      return;
    }

    String imageUrl = destinationPath.replace("/var/www", "http://vs.kiv.zcu.cz");
    result.addInfo("<a href=\"" +imageUrl +"/" +diffImage +"\">Rozdily oproti spravnemu
          obrazku.</a>");
  }
}


/**
 * Urcuje jak bude zvyrazneny odlisny pixel.
 *
 * @return ARGB hodnoty zvyraznovaci barvy.
 */
private static int markDifference() {
  int pixel = 0xff000000;
  int red = 244;
  int green = 34;
  int blue = 58;
```

```
    red = red << 16;
    green = green << 8;

    pixel = pixel | red | green | blue;
    return pixel;
  }

  /**
   * Zesvetleni pixelu.
   *
   * @param pixel Momentalne zpracovavany pixel.
   * @return ARGB hodnoty zesvetlenyho pixelu.
   */
  static int brighten(int pixel) {
    int red = (pixel >> 16) & 0xff;
    int green = (pixel >> 8) & 0xff;
    int blue = (pixel) & 0xff;

    float [] hsbVals = Color.RGBtoHSB(red, green, blue, null);
    // Zmena saturace a svetlosti
    Color brighter = Color.getHSBColor(hsbVals[0], 0.2f * hsbVals[1], 0.5f * (1f + hsbVals[2]));

    int redModified = brighter.getRed();
    int greenModified = brighter.getGreen();
    int blueModified = brighter.getBlue();

    redModified = redModified << 16;
    greenModified = greenModified << 8;

    int pixelModified = pixel & 0xff000000;
    pixelModified = pixelModified | redModified | blueModified | greenModified;

    return pixelModified;
  }

  public String getNazev() {
    return "Porovnat PNG (KIV/OOP)";
  }

  public List<ParametrAkce> getParametry() {
    ArrayList<ParametrAkce> list = new ArrayList<ParametrAkce>();
    return list;
  }

  public String getPopis() {
    return "Akce ááporovnv ývygenerovan screenshot se ývzorovm áobrzkem.";
  }

  public String getHelp() {
    String help = "<h4>Kontextova napoveda: Vlastni akce \"Porovnat PNG (KIV/OOP)\"</h4>";
    help += "<p>Akce slouzi k porovani studentova obrazku se vzorovym.</p>";
    help += "<p>Pokud se odevzdava JAR soubor, musi byt nejdrive rozbalen pomoci \"Rozbal ZIP
          \".</p>";
    help += "<p>Aby tato akce fungovala, je treba do \"domain.xml\" teto domeny dopsat vlastnost (
          pres WinSCP):</p>";
    help += "<p>&lt;entry key=\"image.sample_name\"&gt;&lt;/entry&gt;, kde hodnota je nazev
          vzoroveho obrazku i s priponou. ";
    help += "Napr. vzor.png </p>";
    help += "<p>Vzorovy obrazek se nahraje pres \"Nahrat vzorovy obrazek\". (Na server se nahrava
          .zip soubor, kde je slozka,";
    help += "ve ktere je samotny vzorovy obrazek.) </p>";
```

```
    return help;
  }

  public String getId() {
    return "PorovnatPNG";
  }

  public String getKategorii() {
    return KAT_SOUBOR;
  }


}
```

C CD Readme

[Dokumentace]

- Složka obsahuje pdf verzi bakalářské práce a zdrojové tex soubory.

[Domény KIV-OOP]

- Obsahuje jednotlivé konfigurace validačních domén KIV-OOP. V této složce jsou také odevzdané úlohy studentů z let 2011/2012 a 2010/2011, které sloužily jako testovací data.
- Pro nasazení na ostrý server stačí překopírovat domény ze souboru `Uzivatel.xml` do souboru webového rozhraní s našimi jednotlivými doménami

```
/home/validator/tomcat/webapps/ROOT/WEB-INF/data/nasOrionLogin.xml
```

  a překopírovat domény z CD na server

```
/home/validator/validator/data/domains
```

- V `domain.xml` je pak ještě nutno změnit položky:

```
<entry key="html_output_dir"></entry>
<entry key="html_output_url"></entry>
```

  aby odpovídaly cestám na ostrý validační serveru.

[Moduly]

- Ve složce jsou potřebné knihovny pro vytvoření modulů a mnou vytvořené moduly včetně jejich zdrojových kódů.

[Ostatní nástroje]

- Putty, WinSCP, PMD, UMLet