Západočeská univerzita
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Bakalářská práce

Rozšíření funkčnosti validačního serveru a jeho testování

Plzeň 2016 Anežka Jáchymová

Prohlášení

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

V Plzni dne 22. června 2016

Poděkování

Ráda bych poděkovala doc. Ing. Pavlu Heroutovi, Ph.D. za odborné vedení a cenné rady v průběhu zpracování této práce.

Abstract

Hlavním cílem této práce je popsat tvorbu sady regresních testů pro vlastní akce, což jsou zásuvné moduly pro validační server používaný na Západočeské univerzitě v Plzni. Další důležitou součástí je vytvoření nové vlastní akce a úprava již existujících vlastních akcí. V první části je stručně popsán validační server, způsoby testování a nastavení nástrojů pro vývoj. Druhá část se zabývá implementací regresních testů vlastních akcí a popisuje tvorbu a úpravy vlastních akcí, které byly provedeny.

The main goal of this thesis is to describe the creation of regression tests for plugins for the validation server used by the University of West Bohemia in Pilsen. The next important part of this thesis is the creation of a new plugin and the modification of existing plugins. The first part briefly describes the validation server, testing methods and the setup of devepment tools. The second part explains the implementation of regression tests for plugins and describes the creation and modifications of plugins, which were done.

Obsah

1 Úvod **1**

2 Validační server **2**
2.1 Využití . . . 2
2.2 Princip fungování . . . 2
2.3 Adresářová struktura . . . 3
2.4 Testovací verze validačního serveru . . . 4

3 Testování **6**
3.1 Důvody k testování . . . 6
3.2 Bílá a černá skříňka . . . 6
3.3 Funkční testování . . . 7
3.4 Regresní testování . . . 7
3.5 Smoke testování . . . 7
3.6 Unit testování . . . 8
3.7 Automatizované testování . . . 8

4 Řízení vývoje **9**
4.1 Systém pro správu požadavků . . . 9
4.2 Systém pro správu verzí . . . 10

5 Domény a vlastní akce **11**
5.1 Vlastní akce . . . 11
5.2 Existující vlastní akce . . . 12
5.3 Validační domény . . . 16

6 Příprava a práce v prostředí pro vývoj **19**
6.1 Příprava Eclipse . . . 19
6.2 Příprava nástroje SVN . . . 19
6.3 Příprava nástroje Maven . . . 20

7 Testování validačního serveru **22**
7.1 Možnosti testování validátoru 22
7.2 Testovací skupiny 22

8 Implementace testů **25**
8.1 Před spuštěním 25
8.2 Vložení vstupního souboru 25
8.3 Vytvoření parametrů vlastní akce 26
8.4 Nastavení vlastností doméně 26
8.5 Získání poslední zprávy 27
8.6 Automatizace 27

9 Průběh JUnit testu **29**
9.1 *@BeforeClass* 29
9.2 *@Rule* 29
9.3 *@Before* 30
9.4 *@Test* 30
9.5 *@After* 31

10 Práce s vlastními akcemi **32**
10.1 Úprava vlastních akcí 32
10.2 Vytvoření nové vlastní akce 32
10.3 Opravení nalezených chyb 34

11 Závěr **36**

A Ukázka jedné testovací třídy **40**

B SmokeTest.java **45**

1 Úvod

Na Katedře informatiky a výpočetní techniky Fakulty aplikovaných věd na Západočeské univerzitě již osmým rokem funguje validační server, který slouží k automatizovanému vyhodnocování elektronicky odevzdávaných studentských prací. V současné době jsou validátorem standardně vyhodnocovány studentské projekty ze šesti předmětů vyučovaných na FAV a FEL.

Validační server od svého vytvoření prošel poměrně značným změnami. Na jeho vývoji se podílelo minimálně šest programátorek a programátorů. Postupem času vznikaly další a další požadavky na jeho funkčnost a to zejména na rozšíření funkčnosti tzv. vlastních akcí, což jsou de facto plug-in moduly validátoru. Mimo těchto rozšíření a vylepšení samotný validátor již dvakrát migroval na výkonnější server a několikrát na jinou (vyšší) verzi Javy.

Právě kvůli neustálému vylepšování a rozvoji funkčnosti validačního serveru, který bude probíhat i nadále, by bylo prospěšné mít možnost otestování, zda provedené změny neměly nežádoucí účinky na stávající funkčnost validačního serveru. Tuto možnost nám poskytnou regresní testy, které jsou hlavním cílem této práce.

Kromě tohoto hlavního cíle bude mít práce ještě dílčí cíle, kterými budou úpravy funkčnosti existujících vlastních akcí, oprava minoritních chyb a příprava nové vlastní akce. Během let užívání validačního serveru se totiž neustále zpřesňují požadavky na optimální funkčnost validačního serveru a pomocí menších oprav vlastních akcí se k tomuto stavu postupně přibližuje.

Celá tato práce by měla zlepšit použití validátoru v měnícím se prostředí jak hardwarovém, tak i softwarovém. Dále také zvýšit komfort práce s validačním serverem a zajistit jeho přesnější funkčnost.

2 Validační server

2.1 Využití

Ve vysokoškolském prostředí je kladen velký důraz na samostudium a vypracovávání semestrálních prací je jednou z důležitých forem samostudia. Studenti odevzdávají velké množství svých prací. Všechny tyto práce musí být zkontrolovány a objektivně vyhodnoceny podle stejných kritérií, což někdy znamená spoustu repetitivní činnosti pro vyučující.

Díky validátoru mohou tyto opakující se kontroly probíhat automaticky. To vede k úspoře času jak pro učitele, tak pro studenta. Student navíc dostane výsledky ihned a může okamžitě začít vytknuté chyby opravovat.

2.2 Princip fungování

Student odevzdá svojí práci přes portlet, který se nachází na univerzitních webových stránkách `portal.zcu.cz`. Odevzdaný soubor se předá validačnímu serveru společně s dalšími parametry. Jedním z parametrů je i název domény, podle které se má soubor validovat. Poté, co se určí správná doména, se zkontroluje, zda soubor odpovídá základním požadavkům, které je možno nastavit v konfiguraci jednotlivých domén a to např.: maximální velikost souboru.

Server pak připraví prostředí pro validaci. Je vytvořeno nové vlákno, ve kterém bude validace probíhat. Soubor je pro případ nutnosti pozdějšího dohledání uložen do složky *store* a je vytvořen nový dočasný pracovní adresář s jedinečným názvem.

Dále už probíhá vlastní validace, jejíž průběh je definován doménou. Během ní se všechny potřebné informace můžou přidávat do výsledku validace tj. instance třídy *FullValidationResult*. Výsledek validace je převeden do HTML podoby. Následuje promazání vyrovnávacích pamětí serveru a odstranění adresářů a souborů, které při validaci vznikly a jsou již nepotřebné. [Valenta(2012)]

Validační server vrátí HTML soubor, na který je poskytnut odkaz.

2.3 Adresářová struktura

Struktura validačního serveru je rozdělena do tří adresářů: *valid, validrun, valid-common*. Tyto adresáře jsou dělené do podadresářů a obsahují všechny potřebné soubory k fungování validačního serveru. Dále jsou zmíněny adresáře významné pro tuto práci.

V adresáři *valid* se nalézají soubory důležité pro samotný běh serveru a to především v podadresáři *lib*, který obsahuje např. knihovny serveru Apache Tomcat. V podadresáři *logs* se nacházejí logovací soubory a v podadresáři *webapps* jsou soubory definující webové rozhraní validačního serveru. [Duong Manh(2012)]

Pro tuto práci je nejpodstatnější adresář *valid-common*, ve kterém se nacházejí soubory přímo související s validací, a obsahuje následující podadresáře:

- ***data*** uchovává všechny datové soubory související s vlastní validací a obsahuje následující podadresáře:
  - ***domains***
    Adresář *domains* obsahuje soubory patřící k jednotlivým validačním doménám. Každá vytvořená doména má v tomto adresáři právě jeden podadresář, ve kterém jsou všechny potřebné soubory domény – konfigurace, skripty, validační data apod.
  - ***store***
    Do tohoto adresáře se před začátkem validace ukládají odevzdané soubory pro případné pozdější dohledání.
  - ***workdir***
    V adresáři *workdir* se při každé validaci vytvoří dočasný pracovní adresář. Jeho jméno je ve formátu *tmp_xxxxx*, kde *xxxxx* označují jedinečné pěticiferné číslo. Tyto adresáře se obvykle po ukončení validace automaticky mažou, pokud není mazání explicitně zakázáno v konfiguraci domény. Konfiguraci jde změnit „ručně“ v adresáři jednotlivých domén, editací souboru *domain.xml*, nebo přes webové rozhraní validátoru. V pracovním adresáři lze provádět operace s uživatelskými právy, která jsou nastavena v souboru *run.policy*

- ***lib*** obsahuje podadresář *plugins*, ve kterém se nacházejí JAR archivy jednotlivých vlastních akcí.
- ***www*** uchovává ve složce *examples* vstupní soubory k ukázkovým doménám. Ve složce *results* jsou uloženy výsledky validací jako soubory typu *html*. Tyto soubory jsou rozděleny do složek podle domén a v jednotlivých složkách jsou seřazeny podle data validace.

2.4 Testovací verze validačního serveru

Během semestru potřebují mít studenti možnost kdykoliv odevzdávat své semestrální práce a proto je důležité, aby byl validační server vždy funkční. Na druhou stranu je nutné rozšiřovat jeho funkčnost, opravovat nově odhalené problémy a připravovat nové domény. Při těchto činnostech by se ovšem mohly vyskytnout výpadky validačního serveru, které by ohrozily plynulý průběh validace.

Tyto rozporuplné potřeby řeší použití testovací verze validačního serveru. Na testovací verzi serveru se provedou všechny potřebné úpravy a když se změny osvědčí, jsou po určitém časovém období duplikovány na ostrou verzi validátoru. Takto se můžou bez problému provádět změny ohrožující funkčnost validátoru a přesto je ostrý validátor dostupný studentům téměř bez přestávky.

Rozdíly mezi ostrou a testovací verzí validačního serveru prakticky neexistují. Kdyby byly na testovací verzi dlouhodobé rozdíly v nastavení serveru, tak by to popíralo samotnou podstatu existence testovací verze.

Všechny vytvořené rozdíly existují pouze krátkodobě a po ověření, že se se zavedenými změnami nevyskytly nové problémy, se změny přenesou na ostrou verzi validačního serveru.

Vývojářům pro validační server je dán přístup na testovací verzi validačního serveru, kde si můžou odlaďovat svoje programy. Můžou se k němu připojit přes SSH klienta *PuTTy*.

Nejdříve je potřeba se připojit na školní server *eryx.zcu.cz* a z něho vytvořit SSH spojení k serveru s validačním serverem příkazem:

```
ssh -K validt@validator.zcu.cz
```

Poté se může použít pro lepší orientaci souborový správce Midnight Commander. Ten je již na serveru nainstalovaný a spouští se příkazem:

```
mc -a
```

Pak si vývojář může procházet složky, ke kterým má přístup (většinou *validt-common* a *validt*), obr.2.1. Validační server využívá operační systém *GNU/Linux* a vývojář tedy může využívat jakékoliv dostupné programové vybavení.

Obrázek 2.1: Zobrazení složek testovacího validačního serveru přes Midnight Commander

Správnost fungování provedených změn lze ověřit přes webové rozhraní testovacího validačního serveru, které je na adrese: `validator-test.zcu.cz`

3 Testování

3.1 Důvody k testování

Během programování se do kódu zanese určité množství chyb a proto je důležité mít nástroj k odhalování těchto chyb a to je právě testování.

Testování programů většinou spočívá v kontrole, zda sledované chování odpovídá specifikaci. Jinak řečeno se kontroluje, že program dělá to, co má a nedělá to, co nemá. Čím větší ale projekt je, tím náročnější je toto ověřit, proto je důležité začít s testováním už hned na začátku vývoje projektu.

Validační server využívá několik stovek studentů v semestru a odevzdávají několik desítek prací. Každá, i ta sebemenší, chyba funkčnosti je tak rychle objevena a stává se příčinou oprávněných stížností studentů.

3.2 Bílá a černá skříňka

Pojmy *černá* a *bílá skříňka* označují úroveň viditelnosti kódu při testování.

U *černé skříňky* testujeme aplikaci z pohledu uživatele. Známe očekávané vstupy a výstupy, ale neznáme vlastní implementaci kódu. Testování typu černá skříňka tedy simuluje reálné použití aplikace. Toto může mít i své nevýhody. Může totiž dojít k nadměrnému testování některých částí aplikace a nedostatečnému testování jiných. Je ovšem výhodou, že tester může být obyčejný uživatel, který nemá žádné speciální znalosti programování.

U testování typu *bílá skříňka* se předpokládá, že známe detailně implementaci kódu. Toto testování následně kontroluje chování aplikace v implementovaných cestách v kódu. U tohoto testování se neklade důraz na scénáře beroucí ohled na reálné použití aplikace. Díky tomu, že je známa implementace, se mohou části aplikace do hloubky otestovat, ale tester už musí mít jisté znalosti programování.

Něco mezi černou a bílou skříňkou je tzv. *šedá skříňka*. U té se navrhuje testování jako u černé skříňky, ale poté je využita částečná znalost implementace, například k ověření dostatečného pokrytí kódu.[Herout(2016)]

3.3 Funkční testování

Funkční testování je testování typu černé skříňky a zaměřuje se na fukční požadavky. Ty zadává zákazník a definuje je pomocí způsobů užití, které jsou většinou zahrnuty ve specifikaci. U tohoto testování se v podstatě zadá vstup a získaný výstup se porovná s očekávaným.

Funkční testování není zaměřeno na vnitřní strukturu aplikace. Netestují se tedy určité implementační části (jako například jednotlivé metody), ale ta část aplikace, která poskytuje určitou funkčnost.

3.4 Regresní testování

Regresní testy se využívají především při opětovném testování funkčnosti software po provedení změn jako je např. přidávání nových funkcí. Ověřují, zda provedené úpravy neměly nežádoucí vliv na stávající funkčnost, která měla zůstat nezměněna.

Regresní testy se často provádí automatizovaně, protože spočívají pouze v porovnávání výsledku před provedením změn a po něm. Tyto testy je vhodné provádět periodicky nebo po zásahu do aplikace. [Tahchiev et al.(2010)Tahchiev, Leme, Massol,, Gregory]

3.5 Smoke testování

Jako smoke testování označujeme testování, které kontroluje základní funkčnost aplikace. Pokud smoke test neprojde, je chyba v některé z nejdůležitějších částí funkčnosti a je pravděpodobné, že aplikace nebude fungovat správně.

Smoke test by měl proběhnou rychle a odhalit potencionální základní problémy s funkčností. Tyto testy jsou výhodné při rozhodování, zda má cenu začít aplikaci testovat více do hloubky či jako základní kontrola při spouštění aplikace.

3.6 Unit testování

Unit testování (neboli jednotkové testování) je založeno na předpokladu, že pokud postupně otestujeme funkčnost jednotlivých částí programu, vznikne nám správně fungující celek. Každý jednotkový test tedy kontroluje chování části testovaného programu v jednom speciálním případě.[Hunt – Thomas(2003)Hunt, Thomas]

Existuje hned několik frameworků pro programovací jazyk Java, z nichž nejpoužívanější jsou *JUnit* a *TestNG*.[Vogel(2007)] Framework *JUnit* bude použit i v této práci pro tvorbu jednotkových testů a to ve verzi 4.12.

3.7 Automatizované testování

Protože je nutné určité testování provádět opakovaně, je vhodné toto testování zautomatizovat.

Automatizací testování se rozumí použití nějakého nástroje na automatické spouštění a vyhodnocování testů. Vyhodnocování většinou probíhá jako kontrola získaného výsledku s očekávanou hodnotou, která je předem někde pro tyto účely uložena.

4 Řízení vývoje

4.1 Systém pro správu požadavků

Jak již bylo několikrát uvedeno, neustále přicházejí nové požadavky na rozvíjení a vylepšování validačního serveru. Tyto požadavky je vhodné uchovávat v přehledné formě. Právě za tímto účelem byl v projektu validačního serveru využit systém pro správu požadavků Redmine. Redmine je svobodný open source software, který umožňuje mimo jiné zadávat úkoly a sledovat jejich plnění (obr. 4.1).

Při zadávání nového úkolu se vyplní předmět a krátký popis problému, který má tento úkol řešit. Je možné zadat i časové údaje, například do kdy má být úkol uzavřen nebo jaká je odhadovaná doba plnění. Dále se zadají parametry úkolu:

- Typ (Bug, Enhancement, Task, Support, Feature)
- Stav (např. New, Assigned, Resolved, Verified, Invalid)
- Priorita (Low, Normal, High, Urgent)
- Přiřazeno – kdo má úkol plnit
- Severity (Small, Common, Big, Critical) – závažnost problému

Základní životní cyklus úkolu na Redmine začíná tím, že je úkol vytvořen a je mu přiřazen stav *New*. Pokud se řešitel rozhodne úkol přijmout, změní stav na *Assigned*. Poté řešitel pracuje na splnění úkolu a může navyšovat ukazatel, kolik procent je již hotových. Když se řešitel domnívá, že úkol splnil, změní stav na *Resolved*. Pokud je zadavatel spokojen s řešením úkolu, přiřadí úkolu stav *Verified*. [Duong Manh(2014)]

Redmine v sobě dále zahrnuje fórum, správu dokumentů a souborů, zobrazení novinek a nedávných aktivit a propojení s verzovacím nástrojem SVN. K projektu na Redmine také existuje vlastní *Wiki*, kde se uchovávají důležité informace a jsou zde vzorově vyřešeny problémy, které se mohou vyskytnout při vývoji. Při vypracovávání této práce byla Wiki projektu průběžně aktualizována.

Stránky Redmine projektu se nachází na adrese:

`students.kiv.zcu.cz:3443`.

✔	#	Fronta	Stav	Priorita	Předmět	Přiřazeno	Uzavřít do	Odhadovaná doba	% Hotovo
☐	3821	Enhancement	Resolved	Normal	Oprava OK výpisu Spuštění všech JUnit testů	Anežka Jáchymová		1.00	
☐	3800	Enhancement	Resolved	Normal	Oprava chybového hlášení porovnání obrázků	Anežka Jáchymová		1.00	
☐	3799	Enhancement	Resolved	Normal	oprava chybového hlášení JUnit testu	Anežka Jáchymová		1.00	
☐	3798	Enhancement	Resolved	Normal	Oprava chybového hlášení	Anežka Jáchymová		1.00	
☐	2898	Feature	Resolved	Low	Jak se zajistí, aby se před testováním nemusel zadávat vstupní soubor	Anežka Jáchymová		1.00	
☐	2830	Enhancement	Resolved	Low	Formát výpisu vlastní akce z PMD	Anežka Jáchymová		0.50	
☐	2818	Bug	Resolved	Low	Chybné kódování češtiny při výpisu chyby z JUnit	Anežka Jáchymová		1.00	
☐	3819	Enhancement	Resolved	High	Chybový výpis vlastní akce Spustit PMD	Anežka Jáchymová	2015-09-21	1.00	
☐	3820	Bug	Resolved	High	Opačný report chyb vlastní akce Spustit UML testování	Anežka Jáchymová	2015-09-30	1.00	

Obrázek 4.1: Ukázka mně přidělených úkolů na Redmine

4.2 Systém pro správu verzí

Změny ve zdrojových kódech vlastních akcí provádělo více lidí a bylo nutné zavést systém, který by tyto změny dokázal efektivně distribuovat a zároveň uchovávat historii provedených změn. Tyto požadavky byly uspokojeny použitím verzovacího nástroje Apache Subversion, zkráceně SVN.[Collins-Sussman et al.(2011)Collins-Sussman, Fitzpatrick,, Pilato]

Díky SVN je možné si zdrojové kódy stáhnout z centrálního úložiště, provést potřebné změny a nahrát novější verze zpět do úložiště. SVN úložiště projektu je na adrese `forge.kiv.zcu.cz/svn-validator`. Tím jsou veškeré změny spolehlivě zaevidovány a spolu s nimi je uložen čas nahrání, autor a komentář. Do komentáře se píše popis změny a číslo úkolu na Redmine, kvůli kterému byly změny provedeny.

Poznámka: Instalace a nastavení jednotlivých nástrojů bude podrobně popsána v kapitole 6.2, kam logicky patří, protože se jedná o konkrétní část vývojového procesu.

5 Domény a vlastní akce

5.1 Vlastní akce

Jednou ze základních jednotek funkčnosti validátoru je rozšiřitelný modul neboli vlastní akce. Ve své podstatě jsou to plug-in moduly do validačního serveru.

Vlastní akce jsou navzájem nezávislé a jejich funkčnost by se neměla překrývat. Jednotlivé vlastní akce nejsou samy o sobě významné. Jejich výhoda je patrná až po spojení do většího celku ve validační doméně (viz 5.3). Díky možnosti libovolně a jednoduše spojovat funkčně rozdílné akce, je dosaženo mnohem větší flexibity, než by bylo možné s pevně naprogramovanou strukturou.

Z implementačního hlediska je vlastní akce Java třída rozšiřující abstraktní třídu *AbstraktniVlastniAkce*. Samotné vlastní akce obsahují metodu:

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

Pro tento projekt jsou významné následující vstupní parametry:

- ***ValidationInfo info***

  Tento parametr v sobě zahrnuje informace o konkrétní instanci validace (např.: zdroj, pracovní adresář, cesty...)

- ***FullValidationResult result***

  Tento parametr obsahuje kompletní informace o průběhu a výsledku validace (např.: výsledek validace, zprávy, datum, čas...) Postupně se předává do všech kroků validace, kde může být průběžně doplňován.

- ***Collection<Parametr> parametry***

  Kolekce vstupních parametrů.

Během validace mohou právě vykonávané vlastní akce přidávat do výsledku validace (*FullValidationResult result*) svoje zprávy. Tyto zprávy mají dokumentovat průběh validace.

Možné typy zpráv:

- info – zprávy informačního charakteru; nemají vliv na validaci. Přidávají se příkazem `result.addInfo()`.
- warning – zprávy varující před nějakým menším problémem, který není kritický, ale měl by být opraven; nemají vliv na validaci. Přidávají se příkazem `result.addWarning()`.
- error – chybové zprávy, upozorňující na závažný problém; mají vliv na validaci. Přidávají se příkazem `result.addError()`.

5.2 Existující vlastní akce

Cíl této práce je mimo jiné implementovat testy pro 26 existujících vlastních akcí. Aby bylo možné vytvořit správné testy, je nutné nejdříve chápat, jakou mají jednotlivé vlastní akce plnit funkci. Proto je zde uveden základní přehled existujících vlastních akcí. Popisovaná funkčnost tak slouží i jako specifikace jejich požadovaného chování.

Všechny parametry zmiňované u jednotlivých akcí je možné zadat vlastním akcím přes webové rozhraní validačního serveru.

- ***generovanijavadoc*** – generuje Javadoc dokumentaci do pracovního adresáře validace. Je možné jí předat Javadoc parametry, které bychom normálně zadávali do příkazové řádky. Lze omezit dobu, kterou se má čekat na dokončení generování.
- ***kompilaceadresare*** – kompiluje soubory v zadaném adresáři. Na cílový adresář se může odkazovat pomocí existující JS proměnné (v tomto případě musí být její název uvozen znakem dolaru) nebo relativní cestou z pracovního adresáře. Pokud není vyplněn cílový adresář, bere se defaultně pracovní adresář.
- ***kompilacesouboru*** – slouží ke kompilaci jednoho souboru. Na soubor se může odkazovat pomocí existující JS proměnné nebo relativní cestou

z pracovního adresáře. Pokud není vyplněn soubor ke kompilaci, bere se defaultně vstupní soubor.

- ***kontrolajavadocdokumentace*** – kontroluje úplnost Javadoc dokumentace podle zadaných parametrů. Jako jeden z parametrů se zadává rodičovský adresář, ve kterém se nachází Javadoc dokumentace. Na tento adresář lze odkazovat pomocí existující JS proměnné nebo relativní cestou z pracovního adresáře. Pomocí checkboxů pak zaškrtneme, jaké prvky mají být v dokumentaci kontrolovány.

- ***kontrolanazvuodevzdavanehosouboru*** – kontroluje, zda se název odevzdávaného souboru shoduje se zadaným. Požadované jméno souboru můžeme zadat také jako regulární výraz.

- ***kontrolanepovolenychtypusouboru*** – slouží jako kontrola toho, že všechny soubory obsažené v cílové složce splňují požadavky na zadaný typ souboru. Vlastní akce v základním stavu poskytovala možnost zkontrolovat, zda se v cílovém adresáři nevyskytují zakázané typy souborů. Po rozšíření její funkčnosti je možné pomocí checkboxu využít inverzní kontrolu, tzn. kontrolu, zda cílový adresář obsahuje pouze soubory zadaného typu. Další checkbox umožňuje nastavit, zda chceme kontrolovat i podadresáře cílového adresáře. Na cílový adresář se může odkazovat pomocí existující JS proměnné nebo relativní cestou z pracovního adresáře.

- ***kontrolavelikostiodevzdavanehosouboru*** – kontroluje to, že odevzdávaný soubor nepřekročí mezní velikost. Požadovanou kontrolovanou velikost lze zadat jako přesnou velikost nebo jako interval. Interval se zadává pomocí znaků menší (<), větší (>) a rovná se (=).

- ***kopirovatdoworkdir*** – akce sloužící ke kopírování souborů ze složky domény do pracovního adresáře. Soubory ke zkopírování se mohou zadat jako seznam souborů oddělených středníkem nebo jako regulární výraz.

- ***kopirovatslozkydoworkdir*** – slouží ke zkopírování složek z doménové složky do pracovního adresáře. Složky ke zkopírování se zadávají jako seznam souborů oddělených středníkem.

- ***najdiadresare*** – vyhledá v cílovém adresáři všechny adresáře odpovídající zadanému regulárnímu výrazu. Na cílový adresář se může odkazovat pomocí existující JS proměnné nebo relativní cestou z pracovního adresáře. Pokud není vyplněn cílový adresář, bere se defaultně pracovní

adresář. Pomocí checkboxu je možné nastavit, zda se mají prohledávat i podadresáře cílového adresáře. Dále se jako parametry zadají dva názvy JS proměnných. Jedna bude obsahovat počet nalezených souborů odpovídajících filtru a druhý pole odkazů (indexované od nuly) na nalezené soubory.

- ***najdisoubory*** – vyhledá v cílovém adresáři všechny soubory (včetně adresářů) odpovídající zadanému regulárnímu výrazu a případně spočte jejich velikost. Na cílový adresář se může odkazovat pomocí existující JS proměnné nebo relativní cestou z pracovního adresáře. Pokud není vyplněn cílový adresář, bere se defaultně pracovní adresář. Pomocí checkboxu je možné nastavit, zda se mají prohledávat i podadresáře cílového adresáře. Dále se jako parametry zadají tři názvy JS proměnných. Jedna bude obsahovat počet nalezených souborů odpovídajících filtru, druhá pole odkazů (indexované od nuly) na nalezené soubory a třetí celkovou velikost nalezených souborů.

- ***porovnatpng*** – porovnává dva obrázky typu *.png*. Oba obrázky se musí nalézat v pracovním adresáři validace. Obrázky k porovnávání se mohou zadat pomocí existující JS proměnné nebo relativní cestou z pracovního adresáře. Pokud se nevyplní jméno porovnávaného obrázku, vezme se automaticky první nalezený soubor typu *.png* z pracovního adresáře.

- ***porovnejsoubory*** – slouží k porovnání obsahů dvou souborů, které se nacházejí v pracovním adresáři validace. Porovnávané soubory můžeme určit pomocí relativní cesty z pracovního adresáře (popř. pouze jménem, pokud jsou přímo v pracovním adresáři) nebo JS proměnnou. Pokud se soubory v něčem liší, vypíše se rozdíl do výstupu validace.

- ***porovnejstrukturuadresaru*** – kontroluje, zda mají dva různé adresáře shodnou strukturu souborů, tzn. zda obsahují soubory, které mají stejná jména. Velikost ani obsah souborů se nekontroluje. Pomocí checkboxu lze nastavit, zda mají být soubory, které jsou v jednom z adresářů navíc, považovány za chybu. Na adresáře k porovnání se lze odkazovat pomocí existující JS proměnné nebo relativní cestou z pracovního adresáře.

- ***rozbalzipjar*** – umožňuje rozbalit archiv typu *ZIP* nebo *JAR*. Na adresář k rozbalení se může odkazovat pomocí existující JS proměnné nebo relativní cestou z pracovního adresáře. Pokud není tento parametr vyplněn, bere se defaultně odevzdávaný soubor. Na rozbalený archiv se

můžeme odkazovat pomocí JS proměnné, jejíž jméno zadáme jako parametr.

- ***smazatsouboryadresare*** – smaže zadané soubory a adresáře. Soubory ke smazání se zadávají jako seznam, jehož prvky jsou oddělené středníkem. Prvky seznamu můžou být relativní cesty z pracovního adresáře nebo existující JS proměnné.

- ***spoctislovaastrany*** – slouží ke zjištění počtu slov a stran v dokumentech typu *doc* a *PDF*. Na soubor ke kontrole se může odkazovat pomocí existující JS proměnné nebo relativní cestou z pracovního adresáře. Pokud není vyplněno jméno souboru ke spuštění, bere se defaultně odevzdávaný soubor.

- ***spustenijarprogramu*** – slouží ke spuštění *JAR* programu. Na soubor ke spuštění se může odkazovat pomocí existující JS proměnné nebo relativní cestou z pracovního adresáře. Pokud není vyplněno jméno souboru ke spuštění, bere se defaultně odevzdávaný soubor. Jako další parametr můžeme zadat argumenty, které se mají programu předat. Argumenty mohou být zadány i pomocí JS proměnných.

- ***spustenijavaprogramu*** – slouží ke spuštění Java programu. Na soubor ke spuštění typu *.class* se může odkazovat pomocí existující JS proměnné nebo relativní cestou z pracovního adresáře. Pokud není vyplněno jméno souboru ke spuštění, bere se defaultně odevzdávaný soubor. Jako další parametr můžeme zadat argumenty, které se mají programu předat. Argumenty mohou být zadány i pomocí JS proměnných.

- ***spustenikonkretnihojunittestu*** – akce sloužící ke spuštění konkrétního *JUnit* testu. Před spuštěním této akce je nutné mít v pracovním adresáři přeloženy všechny soubory, které jsou potřebné k testování (tedy alespoň soubor s testem a testovaný soubor). Jako parametry zadáme jméno přeloženého souboru s testem a jméno testu, který chceme spustit. Oba parametry lze zadat i jako JS proměnnou.

- ***spustenivsechjunittestu*** – slouží ke spuštění všech *JUnit* testů v zadaném souboru. Před spuštěním této akce je nutné mít v pracovním adresáři přeloženy všechny soubory, které jsou potřebné k testování (tedy alespoň soubor s testy a testovaný soubor). Jako parametry zadáme jméno přeloženého souboru s testy, který chceme spustit. Oba parametry lze zadat i jako JS proměnnou.

- ***spustenipmd*** – zkontroluje zadaný zdrojový soubor pomocí *PMD* se zadanou sadou pravidel. Zdrojový soubor ke kontrole a soubor s *PMD* pravidly se může zadat pomocí existující JS proměnné nebo relativní cestou z pracovního adresáře.

- ***spusteniumltestovani*** – nejdříve prohledá pracovní adresář a první nalezený soubor s koncovnou *.uxf* použije jako vstupní argument pro spuštění zadaného *JAR* souboru. *JAR* soubor by měl obsahovat testy na *UML* diagramy. *JAR* soubor se může zadat pomocí existující JS proměnné nebo relativní cestou z pracovního adresáře.

- ***spustitducktesty*** – slouží ke spuštění *Duck* testů. Jako parametr zadáme odkaz na soubor s *Duck* testy, který se nachází v pracovním adresáři. Na tento soubor se může odkazovat pomocí existující JS proměnné nebo relativní cestou z pracovního adresáře. Vlastní akce soubor rozbalí do pracovního adresáře a postupně spustí všechny nalezené testy.

- ***vlozitscreenshot*** – prohledá pracovní adresář a pokud nalezne screenshot zadaného jména, vloží ho do výstupu validace. Jméno hledaného screeshotu lze zadat jako regulární výraz.

- ***vymazsouborypodlevzoru*** – smaže soubory v cílovém adresáři podle struktury vzorového adresáře. Akce porovnává strukturu cílového adresáře se vzorovým a pokud nalezne v obou adresářích stejně pojmenovaný soubor na stejném místě v adresářové struktuře, tak tento soubor smaže z cílového adresáře.

5.3 Validační domény

Vlastním akcím jsou funkčně nadřazené validační domény. Doména je definována pomocí *XML* souborů. Tyto soubory se jmenují *webmodule.xml*, který obsahuje sekvenci jednotlivých kroků (tj. vlastních akcí) validace, a *domain.xml*, kde je uloženo detailní nastavení domény. Oba soubory jsou uloženy v adresáři domény na validačním serveru (viz sekce 2.3). Dále je důležitý soubor *run.policy*, ve kterém jsou zadána práva pro přístup, čtení a zápis.

Poznámka: Ve starších verzích domén byl ještě soubor s názvem *process.xml* a adresář *nl* se souborem *texts.properties*. Právě soubor *process.xml* pak řídil validaci. Tyto soubory již v novější verzi domén nejsou potřeba.

Pro jednodušší a efektivnější používání validačního serveru bylo vytvořeno webové uživatelské rozhraní. Přes toto webové rozhraní se v současné době tvoří validační domény. Každý uživatel s přístupem k webovému rozhraní valičního serveru má na hlavní stránce výpis všech domén, u kterých je uveden jako správce. U těchto domén může měnit (tj. ubírat či přidávat) oprávnění a tím usměrňovat, kdo má dovoleno s danou doménou manipulovat.

Domény uživatele > Upravit doménu ukazkova_domena
ukazkova_domena
Domena ukazkova_domena
Název domény: ukazkova_domena
Popis domény: Domena ukazkova_domena
Seznam správců domény (oddělený čárkou): anezkaj
uložit
Detailní nastavení
Kroky validace
1. ukazkovy_krok_1 Uprav Zruš
2. ukazkovy_krok_2 Uprav Zruš
nový krok
Testování domény
Vyberte soubor, který chcete otestovat
Procházet... Soubor nevybrán.
Testovat
Zpět

Obrázek 5.1: Tvoření validační domény přes webové rozhraní validačního serveru

Po vytvoření prázdné domény se do ní mohou vkládat jednotlivé kroky. Každý krok má dvě části. V první části se nastavuje podmínka, která označuje, kdy se má krok provést (např. vždy nebo pokud validace (ne)obsahuje chybu). A ve druhé se může přidat buď vlastní akce, nebo JS skript.

V *detailním nastavení* je možné nastavit parametry platící pro celou validační doménu. Tyto parametry můžou být například, zda se má po skončení validace mazat pracovní adresář, maximální doba validace nebo kódování. Toto nastavení se kontroluje ještě před začátkem samotné validace.

Validace je vyhodnocena podle obsahu proměnné *result*. Možné výsledky (viz Tabulka 5.1) jsou reprezentovány číselným kódem, který je definován

VR_OK	Validace proběhla v pořádku
VR_SERVER_ERROR	Validace neuspěla kvůli interní chybě validačního serveru
VR_COMPILATION_ERROR	Chyba při překladu
VR_RUNTIME_ERROR	Chyba při běhu validovaného programu
VR_BAD_RESULTS	Špatné výstupy validovaného programu
VR_TIMEOUT	Vypršení času – validace či některá její část překročila max. povolenou dobu
VR_INVALID_DOMAIN	Neexistující validační doména
VR_INVALID_INPUT	Chybný vstup validace
VR_INVALID_LIBRARY_CALL	Použití nepovolené knihovny programovacího jazyka

Tabulka 5.1: Možné výsledky validace

rozhraním *ValidationResult*. Pokud *result* na konci validace obsahuje kód chyby, tak se validovaná úloha vyhodnotí jako nevyhovující.

V současné době existuje na testovacím validačním serveru přibližně 40 domén, z nichž většina je tzv. ukázkových domén. Ukázkové domény představují vzor pro tvoření vlastních domén a zároveň ukazují, jak jednotlivé vlastní akce fungují a jak se dají kombinovat, aby bylo dosaženo požadované funkčnosti.

Pro každý vyučovaný předmět, který využívá automatizované kontroly prací validačním serverem, existuje minimálně jedna validační doména. Počet domén na předmět není nikterak omezen a závisí pouze na počtu úloh, které se mají v daném předmětu kontrolovat. Efektivní využití validačního serveru je zřejmé například u předmětu KIV/OOP, u kterého existuje 17 validačních domén a kontrolují se tak všechny úkoly zadané v semestru.

6 Příprava a práce v prostředí pro vývoj

6.1 Příprava Eclipse

Aby se mohl používat Eclipse pro vývoj validačního serveru a jeho akcí, je nutné ho příslušně nastavit. Je také vhodné mít pro tento vývoj samostatný pracovní adresář. Všechny zmíněné soubory je možné stáhnout z *Redmine*.

Nejdříve je potřeba nastavit *Code formatter* a *Coding templates*. Ty přidáme do nastavení Eclipse následujícím způsobem:

V Redmine klikneme na Soubory -> stáhneme soubor *codeformatter.xml* -> v Eclipse zvolíme Window -> Preferences -> Java -> Code Style -> Formatter -> Import -> Zadáme cestu k souboru -> Apply -> OK

V Redmine klikneme na Soubory -> stáhneme soubor *codetemplates.xml* -> v Eclipse zvolíme Window -> Preferences -> Java -> Code Style -> Code Templates -> Import -> Zadáme cestu k souboru -> Apply -> Rozklikneme položku Code -> Klikneme na položku New Java Files -> Edit -> změníme email na svůj vlastní e-mail -> OK -> Apply -> OK [Cais(2015)]

Pokud chceme vyvíjet JUnit testy je nutné přidat použité knihovny a JAR soubory jednotlivých vlastních akcí do *Build path* projektu. To provedeme následujícím způsobem:

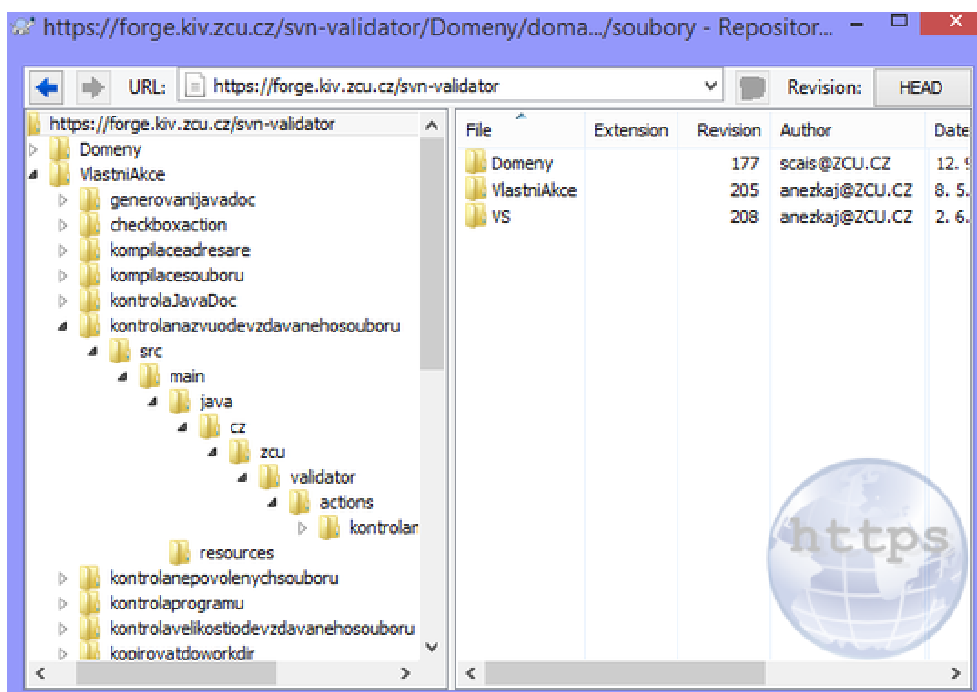
V Eclipse klikneme pravým tlačítkem na projekt s validačním serverem -> Build Path -> Configure Build Path -> Add External JARs -> vybereme všechny knihovny a vlastní akce, které chceme přidat -> Apply -> OK

6.2 Příprava nástroje SVN

Protože jsou všechny zdrojové kódy týkající se validátoru uloženy na SVN, je před začátkem vývoje nutné se připojit k SVN. To je možné udělat pomocí programu *TortoiseSVN*, což je vlastně klient pro nástroj *Apache Subversion*. TortioseSVN je samostatná aplikace [`https://tortoisesvn.net`].

Po nainstalování *TortoiseSVN* se tento program ovládá pomocí pravého tlačítka myši. Stačí tímto tlačítkem kliknout na soubor, který chceme nahrát na SVN, a objeví se možnosti použití. Příkladem jsou možnosti *commit*, který nahraje aktualizovaný soubor na SVN, *add*, který přidá nový soubor na SVN nebo *repo-browser*, kterým můžeme zobrazit všechny soubory uložené na SVN (obr. 6.1).

Pokud chceme propojit SVN s Eclipse, můžeme tak učinit následujícím způsobem: Otevřeme Eclipse -> Help -> Eclipse Marketplace -> najdeme *Subclipse* -> nainstalujeme *Subclipse* -> restartujeme Eclipse



Obrázek 6.1: Záloha validačního serveru na SVN zobrazena přes *TortoiseSVN*

6.3 Příprava nástroje Maven

Apache Maven je nástroj pro správu, řízení a automatizaci buildů aplikace, který se používá převážně pro Javu. Pro Eclipse je součástí instalačního balíku od verze *Kepler* pro *JEE*.

V Eclipse nastavíme *Maven* následovně:

V Redmine klikneme na Soubory -> stáhneme soubory *VS-1.0.jar*, *VS-SDK-1.0.jar* a *jdparser-1.0.0.jar* -> v Eclipse zvolíme File -> Import -> Maven -> Install or deploy an artifact to a Maven repository -> jako artifact file vybereme jeden ze stažených jar souborů -> zaškteneme Generate POM -> vyplníme Group Id, Artifact Id a Version následovně:

- Group Id: *cz.zcu.validator*, Artifact Id: *VS*, Version: *1.0*
- Group Id: *cz.zcu.validator*, Artifact Id: *VS-SDK*, Version: *1.0*
- Group Id: *cz.zcu.validator*, Artifact Id: *jdparser*, Version: *1.0.0*

-> Finish

7 Testování validačního serveru

7.1 Možnosti testování validátoru

V případě našeho validačního serveru se jako subjekt testování nabízí dvě hlavní jednotky funkčnosti a to domény a vlastní akce. Vlastní akce jsou nejmenší funkční jednotkou a proto je ze začátku nejdůležitější, aby bylo jisté, že fungují správně.

Testování domén je další možnost pro otestování funkčnosti validátoru. Je ovšem mnohem komplexnější a neodráží tak dobře vlastní funkčnost, jako spíše spolupráci mezi akcemi a správné předávání parametrů. Testování domén by tedy bylo možné použít k testování správného společného chování několika vlastních akcí a webového rozhraní validačního serveru.

7.2 Testovací skupiny

Jak je patrné z kapitoly 5.2 na validátoru existuje 26 vlastních akcí. Vymýšlet pro každou vlastní akci jedinečný typ testu by bylo ve výsledku nepřehledné a neefektivní. Proto byly vlastní akce rozděleny do tzv. *testovacích skupin.*

Každá skupina bude představovat množinu navzájem si podobných akcí, pro které se pak vytvoří prototyp testu. Tento prototyp se následně přizpůsobí jednotlivým vlastním akcím. Díky prototypu se docílí jednotného a přehledného zápisu testu.

Testovací skupiny se dají nejsnáze definovat podle koncového stavu po ukončení validace. Na tomto základě byly stanoveny tři testovací skupiny:

1. **Vypisující** – přidávají informační výpisy do výsledku validace
2. **Tvořící JS proměnné** – jejich výsledkem je nově vytvořená JS proměnná
3. **Měnící stav prostředí validace** – po jejich ukončení je zřetelně změněn stav prostředí validace (např. vznikne nový soubor)

Testování vypisujících akcí je nejjednodušší. Výpisy se můžou získat z parametru *FullValidationResult result* (viz 5.1) a tyto výpisy se můžou porovnat s očekávanými výstupy.

U vlastních akcí, které tvoří nové JS proměnné, je možné porovnat obsah JS proměnných s očekávanými hodnotami.

Testování akcí, které mění prostředí validace bude nejrozmanitější a je pravděpodobné, že pro každou z těchto akcí bude nutné použít jiný typ testování.

Vlastní akce byly rozděleny do testovacích skupin následovně:

1. **Vypisující**
   - kontrolajavadoc
   - kontrolanazvyodevzdavanehosouboru
   - kontrolanepovolenychsouboru
   - kontrolavelikostiodevzdavanehosouboru
   - kopirovatslozkydoworkdir
   - porovnatpng
   - porovnejsoubory
   - porovnejstrukturuadresaru
   - smazatsouboryadresare
   - spustenijarprogramu
   - spustenijavaprogramu
   - spustenikonkretnihojunittestu
   - spustenipmd
   - spusteniumltestovani
   - spustenivsechjunittestu
   - spustitducktesty
   - vlozitscreenshot
   - vymazsouborypodlevzoru
2. **Tvořící JS proměnné**
   - rozbalzipjar

- najdiadresare
- najdisoubory
- spoctislovaastrany

3. **Měnící stav systému**

- generovanijavadoc
- kompilaceadresare
- kompilacesouboru
- kopirovatdoworkdir

8 Implementace testů

8.1 Před spuštěním

Většina JUnit testů předpokládá definovaný stav před spuštěním validace. Ať už jsou to soubory v pracovním adresáři či existující JS proměnné. Tento stav můžeme nastavit v metodě

```
onBeforeValidation(ValidatorTestingContext
                                validatorTestingContext),
```

která je součástí třídy `ValidatorTestingCallback`. Tato třída se předá metodě `spustTestovaciKrokValidace()`.

V době, kdy je použita `onBeforeValidation()` už existuje pracovní a doménový adresář společně s JS prostředím a doménovým nastavením. Je tedy možné tyto atributy libovolně přizpůsobovat.

Díky metodě `onBeforeValidation()` je možné kopírovat potřebné soubory do pracovního a doménového adresáře. Tyto soubory pak slouží jako vstupní soubory pro jednotlivé JUnit testy. Tento postup je jednodušší než vkládání celých souborů jako vstupů.

Jak již bylo řečeno, je možné měnit i JS prostředí, tzn. přidávat JS proměnné podle potřeby. To nám umožňuje testovat vlastní akce na fungování s JS proměnnými.

8.2 Vložení vstupního souboru

U většiny akcí postačí zkopírovat soubor do pracovního adresáře pomocí již zmíněné metody `onBeforeValidation()`, ale u některých je nutné, aby byl vložen přímo vstupní soubor. K tomu slouží třída `ValidationInputImpl`.

Této třídě se v konstruktoru předá jméno, obsah a autor vstupního souboru pro validaci. Takto vytvořený objekt, který nahrazuje reálný vstupní

soubor, se předá metodě `spustTestovaciKrokValidace()`, která ho přijme jako vstupní soubor validace.

Pokud tuto možnost nechceme využívat, zadáme jako parametr metody `spustTestovaciKrokValidace()` hodnotu *null* místo proměnné typu *ValidationInputImpl*. V tom případě se jako vstup validace použije prázdný defaultní vstupní soubor.

8.3 Vytvoření parametrů vlastní akce

Každá vlastní akce má určitý počet parametrů, které můžeme nastavovat přes webové rozhraní validačního serveru a jsou popsané v souboru *resources/nazev_akce.properties*, který se nachází u jednotlivých vlastních akcí. U JUnit testů se musí tyto parametry vytvářet „ručně“.

Všechny parametry vlastních akcí jsou typu *Parametr* a tvoří se zavoláním kontruktoru `new Parametr(nazev, hodnota)`. Pokud je parametr checkbox a nemá být zaškrtnutý, tak se netvoří, v opačném případě může mít libovolnou hodnotu typu *String*, například: `parameters.add(new Parametr ("Cílový adresář", "cilovyAdresar"));`. Všechny parametry pro validaci jsou následně přidány do proměnné typu *ArrayList* a ta je předána metodě `spustTestovaciKrokValidace()`.

8.4 Nastavení vlastností doméně

Validační doména má svoje vlastní parametry, které můžeme nastavovat před začátkem validace (viz 8.1). V tomto případě je nejdůležitější parametr *core.do_cleanup*, který značí, zda se má po ukončení validace domény smazat pracovní adresář. Protože z pohledu JUnit testů probíhá celá validace v jednom nedělitelném kroku, je nutné mít u akcí, které nic nevypisují do výstupu validace, možnost zkontrolovat koncový stav pracovního adresáře. A proto jej nesmíme vymazat.

Nastavení tohoto parametru se provádí tak, že se těsně před spuštěním validace získá z *ValidationInfo* odkaz na doménu a té se pak nastaví parametr *core.do_cleanup* na hodnotu *0*.

```
validatorTestingContext.getValidationInfo()
            .getDomain().getDomainProperties()
                   .setProperty("core.do_cleanup", "0");
```

8.5 Získání poslední zprávy

Jak již bylo řečeno, adekvátní potvrzení funkčnosti vlastní akce v případě, že vypisuje do výstupu validace, je porovnat vypisované zprávy s očekávanými. Přičemž nezáleží na charakteru zprávy, tj. na tom, zda je typu *info, warning* či *error.*

Proto vznikla metoda `getLastMessage()`, která získá z výsledku validace poslední zprávu. Tato zpráva se následně použije při porovnávání metodou `assertEquals()` jako aktuální hodnota získaná z validace.

Takto se získá poslední zpráva ze seznamu zpráv:

```
List<ValidationResultItem> messageList =
        vtc.getFullValidationResult().getMessages();

String message =
        messageList.get(messageList.size()-1).getContent();
```

8.6 Automatizace

Protože bylo vytvořeno přes 100 JUnit testů vlastních akcí, je žádoucí, aby je bylo možné efektivně spouštět nezávisle na vývojovém prostředí. To znamená, že se může vyskytnout potřeba spustit celou existující množinu testů nebo naopak stačí spustit jen jejich vybranou část. *Test suite* umožňuje spojování jednotlivých testů do skupiny, která se pak spustí najednou. Skupina samozřejmě nemusí obsahovat všechny existující testy.

Jedním z cílů této bakalářské práce bylo vytvořit *smoke test. Smoke testy* jsou rychlé testy, které mají zkontrolovat, zda má testovaná aplikace stále své základní funkce. Tyto testy jsou důležité pro ověření funkčnosti aplikace po integraci nových prvků a budou významné zejména při dalším vývoji

funkčnosti validačního serveru a při přechodu na jiné verze software. Byla proto vytvořena jedna *test suite SmokeTest.java* (viz příloha B).

9 Průběh JUnit testu

Jednotkové testy vlastních akcí se provádějí podle stejného schématu. Jednotlivé části jejich JUnit testů se spouští v závislosti na notacích, které jsou v testovací třídě uvedeny. V této kapitole je ukázán průběh JUnit testu na jedné z vytvořených testovacích tříd pro vlastní akci *rozbalzipjar*.

9.1 *@BeforeClass*

Jako první se z testovací třídy spustí metoda s označená anotací *@BeforeClass*. Tato anotace říká, že se metoda provede pouze jednou a to před spuštěním všech testů z testovací třídy. V této metodě se připraví prostředí pro lokální testování.

```
@BeforeClass
public static void oneTimeSetUp() {
    ServerCore.getInstance().getValidatorTestingUtils()
                    .pripravLokalniTestovani();
}
```

9.2 *@Rule*

Anotace *@Rule* umožňuje přidání či redefinici chování všech testovacích metod najednou. `TestName` je jedno z předpřipravených pravidel a poskytuje jméno testovací metody, ve které je test uložen. [Herout(2016)]

```
@Rule
public TestName jmenoTestovaciMetody = new TestName();
```

9.3 *@Before*

Před každým testovacím případem se spustí metoda s anotací *@Before*. V této metodě se připraví vstupní parametry pro danou vlastní akci. Každý vstupní parametr je typu `Parametr`, který se skládá z jména a hodnoty. Tvoření parametrů je popsáno v sekci 8.3.

```
@Before
public void setUp() throws Exception {
    parameters = new ArrayList<Parametr>();
    parameters.add(new Parametr("ZIP archiv",
        "RozbalZipJarInput.zip"));
    parameters.add(new Parametr("Cilovy adresar",
        "cilovyAdresar"));
}
```

9.4 *@Test*

Anotace *@Test* uvozuje testovací metody. Ty jsou následně spuštěny pomocí *JUnit*. Na konci každého testovacího případu je zavolána metoda z třídy `Assert`. Tato metoda vyhodnocuje, zda se očekávaná hodnota shoduje s reálnou. V této práci je většinou porovnáván poslední výpis do výstupu validace, popřípadě hodnota *booleovské* proměnné, která značí, zda nastal očekávaný stav prostředí validace (např. existuje určitý soubor).

Nejdříve se zavolá metoda na spuštění validace (viz 8.1) a předají se jí vstupní parametry akce. Návratová hodnota metody `spustValidaci()` poskytuje odkaz na validační kontext, ze kterého můžeme získat informace o proběhlé validaci (například pracovní adresář, zprávy ve výstupu validace...) Řetězec `identifikace` obsahuje informační výpis v případě, že test neprošel.

```
@Test
public void testRozbalZipJar_OK() {
    ValidatorTestingContext vtc = spustValidaci(parameters,
        false);
    File cilovyAdresar = (File)
        vtc.getScope().get("cilovyAdresar", vtc.getScope());
    String identifikace = JMENO_TRIDY + ">" +
```

```
        jmenoTestovaciMetody.getMethodName();
    Boolean spravne = (new File(cilovyAdresar,
        "prvni.txt").exists()) && (new File(cilovyAdresar,
        "druhy.txt").exists());
    assertTrue(identifikace, spravne);
}
```

9.5 *@After*

Při testování některých vlastních akcí je nutné explicitně nastavit, aby se po ukončení akce nemazal pracovní adresář (viz 8.4). Ovšem po provedení kontroly stavu pracovního adresáře je vhodné ho vymazat. To zajistí použití metody s anotací *@After*, která se spouští po každém provedení testu.

```
@After
public void smazWorkdir() {
    if (workdir.exists()) {
        FileUtils.deleteDir(workdir);
    }
}
```

10 Práce s vlastními akcemi

10.1 Úprava vlastních akcí

Během seznamování se s vývojovým prostředím validátoru byly provedeny změny v souladu s úkoly zadanými na *Redmine*.

Byla provedena úprava nápověd k doménám a vlastním akcím, aby byla jednoznačnější a byly přidány příklady použití. Dále bylo potřeba změnit možnosti informačního výpisu vlastních akcí. Byly tedy implementovány přepínače, které mění frekvenci výpisů a některé výpisy byly upraveny podle zadaných požadavků.

V některých akcích byl problém s kódováním češtiny, což se pak projevovalo ve výpisech, které se zobrazovaly studentům. Tyto akce bylo nutné upravit tak, aby důsledně pracovaly s kódováním *UTF-8*. Tyto úpravy byly provedeny přibližně u deseti akcí.

Kromě poměrně jednoduchých úprav popsaných výše byla provedena větší úprava vlastní akce *kontrolovatnepovolenetypysouboru*. Ta kontroluje, zda se v zadané složce a jejích podsložkách nevyskytují soubory se zakázanými příponami. Bylo požadováno, aby šlo tuto akci použít i inverzně, tj. aby šlo kontrolovat, zda složka obsahuje pouze soubory zadaného typu. Filtr stávající akce byl změněn tak, aby toto umožňoval. Mezi kontrolami se přepíná přidaným přepínačem.

10.2 Vytvoření nové vlastní akce

Na validačním serveru se používají javascriptové proměnné, které představují jediný způsob, jak předat v doméně parametry z jedné vlastní akce do jiné. Protože bylo dosud možné vytvořit JS proměnné a naplnit je hodnotou pouze pomocí některých vlastních akcí (např.: *najdisoubory*), vznikla potřeba nové akce, která by umožňovala vytvořit vlastní JS proměnnou a zároveň jí přiřadit libovolnou hodnotou. Tato akce by pak značně zjednodušovala používání některých dalších akcí.

Pro vytvoření nové vlastní akce je nutné nejdříve založit nový *Maven*

projekt a nastavit jeho parametry následovně:

- Group Id: *cz.zcu.validator.actions*
- Artifact Id: *vytvorenijspromenne*
- Package: *cz.zcu.validator.actions.vytvorenijspromenne*
- Verze: *2.0.0*

Poté je nutné definovat *Project Object Model* projektu, což se dělá v XML dokumentu *pom.xml.* [mav(2016)] Z *Redmine* je možné stáhnout ukázkový soubor *pom.xml.* Ten pak stačí zkopírovat do souboru *pom.xml* nově vytvořeného projektu a upravit následující řádky:

- 6. řádku na <artifactId>*vytvorenijspromenne*</artifactId>
- 42. řádku na <AKCE>*cz.zcu.validator.actions.vytvorenijspromenne.*
  .VytvoreniJSPromenne</AKCE>
- 43. řádku na <Built-By>*Anezka Jachymova, anezkaj@students.zcu.cz*
  </Built-By>
- 52. řádku na *vytvorenijspromenne-$project.version*

Jako další krok je potřeba v adresáři *resources* vytvořit dva soubory: *VytvoreniJSPromenne-help.html* a *VytvoreniJSPromenne-text.properties.* První z těchto souborů obsahuje nápovědu k vlastní akci, která se zobrazí přes webové rozhraní. V druhém jsou základní informace o vlastní akci. Můžou se zde definovat proměnné, které umožňují předávat vstupy od uživatele přes webové rozhraní do samotné vlastní akce.

Webové rozhraní akce *vytvorenijspromenne* se skládá ze dvou textových polí, kam se zadává jméno a hodnota nové JS proměnné, a ze dvou checkboxů. První checkbox označuje, zda se má brát zadaná hodnota proměnné jako pole hodnot oddělené středníky. Druhý checkbox zase říká, zda se má finální hodnota brát jako absolutní cesta k souboru. Je totiž možné zadat pouze jméno souboru (např.: *pokus.txt*) a při zapnutí tohoto přepínače se hodnota JS proměnné doplní na *workdir* a jméno souboru, například */opt/valid–common/data/workdir/tmp_00001/pokus.txt.*

V adresáři *java* se nachází zdrojový soubor vlastní akce. V tomto případě se jmenuje *VytvoreniJSPromenne.java*. V něm je definována třída *VytvoreniJSPromenne*, která rozšiřuje abstraktní třídu *AbstraktniVlastniAkce*. V této třídě se nachází pouze metoda `execute()`, která je implementací abstraktní metody své nadřazené třídy.

V této metodě se nejdříve získají vstupy od uživatele, které zadal přes webové rozhraní. Tyto vstupy se do metody předají formálním parametrem `Collection<Parametr> parametry`. Z tohoto parametru se následně získají jejich hodnoty přes jména, která byla definována v dříve zmíněném souboru *VytvoreniJSPromenne-text.properties*.

Následně se zkontroluje, zda nejsou vstupní parametry prázdné a v případě, že jsou, se vyhodí výjimka. Zadaná hodnota nově tvořené JS proměnné může obsahovat již existující JS proměnné, proto se v dalších kroku nahradí všechny tyto existující proměnné jejich hodnotami.

Poté se podmínkami ošetří všechny možnosti, které mohou být navoleny checkboxy. Uživatel mohl ve webovém rozhraní zadat více hodnot oddělených středníky. Pak se tyto hodnoty se uloží do pole, kde se k jednotlivým prvkům přistupuje přes index od nuly. Může být také zadána cesta k souboru a pokud je zaškrnut odpovídající checkbox, tak je tato proměnná rovnou uložena jako typ *File* s hodnotou úplné cesty. A zaškrtnutím obou checkboxů je možné získat pole odkazů na soubory.

Když je uživatelem zadaná hodnota zpracována v souladu s zaškrtnutými checkboxy, je nově vzniklá JS proměnná přidána do proměnného prostředí domény. Pokud má být proměnná typu pole, musí se toto pole nejdříve vytvořit a to příkazem:

```
Scriptable arr = jsContext.newArray(scope, values);
```

Tato nově vytvořená vlastní akce je přístupná přes webové rozhraní testovacího validačního serveru (obr. 10.1).

10.3 Opravení nalezených chyb

I přesto, že vlastní akce byly manuálně testovány před nasazením na validační server, bylo díky nově vytvořeným komplexním JUnit testům odhaleno

Obrázek 10.1: Nastavení parametrů nové vlastní akce přes webové rozhraní validačního serveru.

několik nesrovnalostí. Největší chyba se nacházela ve vlastní akci *vymazsouborypodlevzoru*.

Tato akce by správně měla vzít dva adresáře, z nichž jeden má sloužit jako vzor a druhý jako cíl. Postupně se prochází soubory vzoru a pokud se stejné soubory nachází i v cíli, tak se z cíle smažou. Testování odhalilo, že se soubory nemazaly z cíle, ale ze vzoru.

Chyba byla následně opravena tak, aby fungování vlastní akce odpovídalo její specifikaci.

11 Závěr

V bakalářské práci se podařilo implementovat požadovanou vlastní akci vytvoření a nastavení javascriptové proměnné. Dále byla provedena oprava 10 drobnějších chyb dle úkolů zadaných na Redmine. Systém řízení práce pomocí Redmine se ukázal jako výhodný. Během celého procesu zpracování bakalářské práce byla průběžně opravována, doplňována a aktualizována stávající dokumentace.

Hlavní cíl práce byla příprava testů jednotlivých vlastních akcí. Celá práce byla poměrně složitá, protože vyžadovala detailní pochopení funkčnosti validačního serveru a systém řízení jeho vývoje. Bylo navrženo několik možných přístupů k testům a po konzultaci s vedoucím práce byl vybrán jeden z nich.

Následně bylo připraveno 26 testovacích tříd. Každá pro samostatnou vlastní akci. Počet jednotlivých testů se mění v závislosti na charakteru vlastní akce a pohybuje se v rozmezí od minimálně dvou testů po osm testů. Byly použity pozitivní i negativní testy. Dohromady bylo vytvořeno přes 100 testů. Testy jsou uloženy v 26 java souborech a mají rozsah přibližně 3900 řádů kódu.

Kromě těchto testů byl naprogramován i jejich spouštěč, který splňuje základní funkční scénář ověření funkčnosti všech vlastních akcí. Při vývoji testů pro vlastní akce byly odhaleny některé dosud skryté chyby ve vlastních akcích a tyto chyby byly opraveny.

Všechny body zadání byly splněny.

V současné době ještě nejsou testy uloženy na finální místo, neboť to vyžaduje nějaké drobnější úpravy, ke kterým dojde během letní profylaxe. Přiložené CD obsahuje programové prostředky pro lokální testování a soubor *README*, který vysvětluje, jak toto testování spustit.

Jako podstatné budoucí rozšíření se jeví koncept testů funkčnosti jednotlivých validačních domén, což by byla přirozená nadstavba nad testy vlastních akcí. Během přípravy této bakalářské práce bylo vykonáno množství přípravných aktivit (například příprava kompletních testovacích dat) pro toto rozšíření.

Seznam zkratek

FAV Fakulta aplikovaných věd na Západočeské univerzitě

FEL Fakulta elektrotechnická věd na Západočeské univerzitě

HTML HyperText Markup Language – značkovací jazyk

JAR Java Archive

JEE Java Enterprise Edition

JS Javascript

KIV Katedra informatiky a výpočetní techniky

KIV/OOP Předmět Objektově orientované programování

PMD Statický analyzátor zdrojového kódu

PNG Portable Network Graphics – grafický formát

SDK Software development kit

SSH Secure Shell – zabezpečený komunikační protokol

SVN Apache Subversion – verzovací systém

UML Unified Modeling Language

VS Validační server

XML Extensible Markup Language – rozšiřitelný značkovací jazyk

Literatura

[mav(2016)] *Maven Documentation* [online]. 2016. [cit. 2016/05/22]. Dostupné z: http://maven.apache.org/guides/.

[Cais(2015)] Cais, t. *Wiki projektu Validační server a jeho moduly* [online]. 2015. [cit. 2016/05/20]. Dostupné z: https://students.kiv.zcu.cz:3443/projects/validator/wiki.

[Collins-Sussman et al.(2011)Collins-Sussman, Fitzpatrick,, Pilato] Collins-Sussman, B. – Fitzpatrick, B. – Pilato, M. *Version Control with Subversion* [online]. 2011. [cit. 2016/05/27]. Dostupné z: http://svnbook.red-bean.com/en/1.7/svn-book.pdfg.

[Duong Manh(2012)] Duong Manh, H. *Zásuvné moduly a konfigurace domén validačního serveru.* Bakalářská práce na Fakultě aplikovaných věd Západočeská univerzity v Plzni, 2012.

[Duong Manh(2014)] Duong Manh, H. *Rozšiřující moduly validačního serveru.* Diplomová práce na Fakultě aplikovaných věd Západočeská univerzity v Plzni, 2014.

[Herout(2016)] Herout, P. *Záznamy přednášek z předmětu KIV/OKS* [online]. 2016. [cit. 2016/05/09]. Dostupné z: http://www.kiv.zcu.cz/~herout/vyuka/oks/prednasky/oks-1a4.pdf.

[Hunt – Thomas(2003)Hunt, Thomas] Hunt, A. – Thomas, D. *Pragmatic Unit Testing in Java with JUnit.* The Pragmatic Bookshelf, 2003. ISBN 0-9745140-1-2.

[Tahchiev et al.(2010)Tahchiev, Leme, Massol,, Gregory] Tahchiev, P. et al. *JUnit in Action, Second Edition.* Manning Publications Co., 2 edition, 2010. ISBN 978-1935182023.

[Valenta(2012)] VALENTA, L. *Validační server* [online]. 2012. [cit. 2016/05/23]. Dostupné z: `https://validator-test.zcu.cz/vs/auth/doc/`.

[Vogel(2007)] VOGEL, L. *Unit Testing with JUnit - Tutorial* [online]. 2007. [cit. 2016/05/20]. Dostupné z: `http://www.vogella.com/tutorials/JUnit/article.html#junittesting`.

A Ukázka jedné testovací třídy

```
package cz.zcu.validationserver.junit;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertTrue;

import java.io.File;
import java.util.ArrayList;
import java.util.List;
import java.util.Locale;

import org.junit.After;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Rule;
import org.junit.Test;
import org.junit.rules.TestName;
import org.mozilla.javascript.Scriptable;

import cz.zcu.validationserver.ValidationInputImpl;
import cz.zcu.validationserver.common.ServerCore;
import cz.zcu.validationserver.testing.ValidatorTestingCallback;
import cz.zcu.validationserver.testing.ValidatorTestingContext;
import cz.zcu.validationserver.utils.FileUtils;
import cz.zcu.validationserver.validation.ValidationResultItem;
import cz.zcu.validationserver.webmodule.data.Parametr;

public class RozbalZipJarTest {

    private static String JMENO_TRIDY;

    private static final int ZDROJ = 0;

    private static final int CIL = 1;

    private static final String inputPath =
        "inputs\RozbalZipJarInput.zip";

    private ArrayList<Parametr> parameters;

    private File workdir;
```

```
@Rule
public TestName jmenoTestovaciMetody = new TestName();

@BeforeClass
public static void oneTimeSetUp() {
    ServerCore.getInstance().getValidatorTestingUtils()
        .pripravLokalniTestovani();
}

@Before
public void setUp() throws Exception {
    parameters = new ArrayList<Parametr>();
    parameters.add(new Parametr("ZIP archiv",
        "RozbalZipJarInput.zip"));
    parameters.add(new Parametr("Cilovy adresar",
        "cilovyAdresar"));
}

@After
public void smazWorkdir() {
    if (workdir.exists()) {
        FileUtils.deleteDir(workdir);
    }
}

private ValidatorTestingContext
    spustValidaci(ArrayList<Parametr> parameters, boolean
    input) {
    ValidationInputImpl inputFile = null;
    if (input) {
        Locale loc = new Locale("cz");
        inputFile = new ValidationInputImpl(new File(inputPath),
            "testing-domain", "testing-author", loc, null);
    }
    ValidatorTestingContext validatorTestingContext =
        ServerCore.getInstance().getValidatorTestingUtils()
        .spustTestovaciKrokValidace("Rozbal_ZIP",
        parameters, inputFile, new ValidatorTestingCallback() {
```

```
            @Override
            public void onBeforeValidation(ValidatorTestingContext
                validatorTestingContext) throws Exception {
                validatorTestingContext.getValidationInfo()
                     .getDomain().getDomainProperties()
                     .setProperty("core.do_cleanup", "0");
                FileUtils.copyFile(new File(inputPath), new
                    File(validatorTestingContext
                     .getValidationInfo().getWorkDir(),
                     "RozbalZipJarInput.zip"));
                new File(validatorTestingContext.getValidationInfo()
                     .getWorkDir(), "koncovka.txt").createNewFile();
                File src = new File(inputPath);
                Scriptable scope =
                     validatorTestingContext.getScope();
                scope.put("src", scope, src);
            }
        });
        workdir =
            validatorTestingContext.getValidationInfo().getWorkDir();
        return validatorTestingContext;
    }

    private String getLastMessage(ValidatorTestingContext vtc) {
        List<ValidationResultItem> messageList =
            vtc.getFullValidationResult().getMessages();
        String message =
            messageList.get(messageList.size()-1).getContent();
        return message;
    }

    @Test
    public void testRozbalZipJar_OK() {
        ValidatorTestingContext vtc = spustValidaci(parameters,
            false);
        File cilovyAdresar = (File)
            vtc.getScope().get("cilovyAdresar", vtc.getScope());
        String identifikace = JMENO_TRIDY + ">" +
            jmenoTestovaciMetody.getMethodName();
        Boolean spravne = (new File(cilovyAdresar,
            "prvni.txt").exists()) && (new File(cilovyAdresar,
            "druhy.txt").exists());
        assertTrue(identifikace, spravne);
```

```
}

@Test
public void testRozbalZipJar_OK_JS() {
    parameters.get(ZDROJ).setValue("$src");
    ValidatorTestingContext vtc = spustValidaci(parameters,
        false);
    File cilovyAdresar = (File)
        vtc.getScope().get("cilovyAdresar", vtc.getScope());
    String identifikace = JMENO_TRIDY + ">" +
        jmenoTestovaciMetody.getMethodName();
    Boolean spravne = (new File(cilovyAdresar,
        "prvni.txt").exists()) && (new File(cilovyAdresar,
        "druhy.txt").exists());
    assertTrue(identifikace, spravne);
}

@Test
public void testRozbalZipJar_OK_InputFile() {
    parameters.get(ZDROJ).setValue("");
    ValidatorTestingContext vtc = spustValidaci(parameters,
        true);
    File cilovyAdresar = (File)
        vtc.getScope().get("cilovyAdresar", vtc.getScope());
    String identifikace = JMENO_TRIDY + ">" +
        jmenoTestovaciMetody.getMethodName();
    Boolean spravne = (new File(cilovyAdresar,
        "prvni.txt").exists()) && (new File(cilovyAdresar,
        "druhy.txt").exists());
    assertTrue(identifikace, spravne);
}

@Test
public void testRozbalZipJar_ChybiCilovyAdr() {
    parameters.get(CIL).setValue("");
    ValidatorTestingContext vtc = spustValidaci(parameters,
        false);
    String identifikace = JMENO_TRIDY + ">" +
        jmenoTestovaciMetody.getMethodName();
    assertEquals(identifikace, "Spatny parametr!Cilovy
        adresar", getLastMessage(vtc));
}
```

```
    @Test
    public void testRozbalZipJar_NeexistujeZipJar() {
        parameters.get(ZDROJ).setValue("spatne.txt");
        ValidatorTestingContext vtc = spustValidaci(parameters,
            false);
        String identifikace = JMENO_TRIDY + ">" +
            jmenoTestovaciMetody.getMethodName();
        assertEquals(identifikace, "Chybi resource:
            file.doesnt_exist", getLastMessage(vtc));
    }

    @Test
    public void testRozbalZipJar_SouborNeniTypuZipJar() {
        parameters.get(ZDROJ).setValue("koncovka.txt");
        ValidatorTestingContext vtc = spustValidaci(parameters,
            false);
        String identifikace = JMENO_TRIDY + ">" +
            jmenoTestovaciMetody.getMethodName();
        assertEquals(identifikace, "Chybi resource:
            file.is_not_archive", getLastMessage(vtc));
    }

}
```

B SmokeTest.java

```java
package cz.zcu.validationserver.junit;

import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;

@RunWith(Suite.class)
@SuiteClasses({ GenerovaniJavadocTest.class,
               KompilaceAdresareTest.class,
               KompilaceSouboruTest.class,
               KontrolaJavaDocTest.class,
               KontrolaNazvuOdevzdavanehoSouboruTest.class,
               KontrolaNepovolenychTypuSouboruTest.class,
               KontrolaVelikostiOdevzdavanehoSouboruTest.class,
               KopirovatDoWorkdirTest.class,
               KopirovatSlozkyDoWorkdirTest.class,
               NajdiAdresareTest.class,
               NajdiSouboryTest.class,
               PorovnatPNGTest.class,
               PorovnejSouboryTest.class,
               PorovnejStrukturuAdresaruTest.class,
               RozbalZipJarTest.class,
               SmazatSouboryAdresareTest.class,
               SpoctiSlovaAStranyTest.class,
               SpusteniJARProgramuTest.class,
               SpusteniJavaProgramuTest.class,
               SpusteniKonkretnihoJUnitTestuTest.class,
               SpusteniPMDTest.class,
               SpusteniUMLTestovaniTest.class,
               SpusteniVsechJUnitTestuTest.class,
               SpustitDuckTestyTest.class,
               VlozitScreenshotTest.class,
               VymazSouboryPodleVzoruTest.class })
public class SmokeTest {

}
```