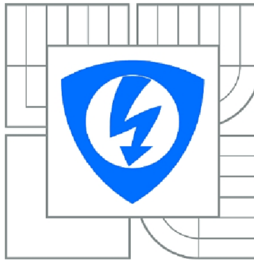
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
ÚSTAV ELEKTROTECHNOLOGIE

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION
DEPARTMENT OF ELECTRICAL AND ELECTRONIC TECHNOLOGY

NÁVRH FRAMEWORKU PRO AUTOMATIZOVANÉ TESTOVÁNÍ WEBOVÝCH STRÁNEK

FRAMEWORK DESIGN FOR AUTOMATED WEB PAGES TESTING

DIPLOMOVÁ PRÁCE
MASTER THESIS

AUTOR PRÁCE
AUTHOR
Bc. LUKÁŠ SIEBER

VEDOUCÍ PRÁCE
SUPERVISOR
Ing. MARTIN FRK, Ph.D.

BRNO 2014

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky a komunikačních technologií

Ústav elektrotechnologie

Diplomová práce

magisterský navazující studijní obor
Elektrotechnická výroba a management

Student:	Bc. Lukáš Sieber	***ID:***	119603
Ročník:	2	***Akademický rok:***	2013/2014

NÁZEV TÉMATU:

Návrh frameworku pro automatické testování webových stránek

POKYNY PRO VYPRACOVÁNÍ:

S využitím odborné literatury prostudujte možnosti volně dostupného nástroje Selenium WebDriver pro automatizaci testování webových aplikací.
Vytvořte tzv. "Proof of concept" a integrujte nástroj Selenium WebDriver do připravovaného frameworku tak, aby vznikl ucelený nástroj umožňující tvorbu automatizovaného testu i bez hlubších znalostí programování. Za účelem dosažení nezávislosti programového popisu webových stránek a tříd testů, tj. i snadné udržitelnosti i modifikovatelnosti celého projektu, klaďte důraz na vhodné rozdělení projektu na dílčí samostatné části, tak aby programový popis stránek byl zcela oddělen od samostatných tříd popisujících testy. Pro doplňkovou funkcionalitu frameworku, jako je např. přístup do databáze a vkládání závislostí čtení xml souborů, využijte běžně dostupné, prověřené a široce používané nástroje (Google Guice, Hibernate, apod.). Navrhněte vhodný formát zápisu výsledků testů, který bude možné uchovávat a poslouží pro pozdější analýzu. Celé řešení realizujte v programovacím jazyce Java vzhledem k jeho multiplatformnosti. Praktickou funkčnost frameworku demonstrujte na konkrétních ukázkách testů vybraných webových stránek.

DOPORUČENÁ LITERATURA:

Podle pokynů vedoucího práce.

Termín zadání:	10.2.2014	***Termín odevzdání:***	29.5.2014

Vedoucí práce: Ing. Martin Frk, Ph.D.
Konzultanti diplomové práce:

doc. Ing. Petr Bača, Ph.D.
Předseda oborové rady

Abstrakt

Následující práce se zabývá problematikou automatizovaného testování webových aplikací s využitím volně dostupného nástroje vyvíjeného pod Google Code. Tímto nástrojem je Selenium webdriver, jež se v poslední době stal oblíbeným nástrojem využívaným právě k automatizaci testování webových stránek. Nespornou výhodou tohoto řešení jsou nízké vstupní náklady, široká základna uživatelů a samozřejmě živá komunita kolem tohoto nástroje poskytující cenné informace. Výsledným cílem práce je vytvoření frameworku napsaného s pomocí jazyka Java, jež bude v sobě integrovat právě jmenovaný nástroj s umožněním efektivní tvorby funkčních testů webových stránek a to vše při snadné udržitelnosti testovacích scénářů. Z tohoto hlediska je kladen důraz na oddělení popisné části "webových objektů" a samotného popisu testovacího scénáře.

Klíčová slova

Selenium, Webdriver, Java, Hibernate, Oracle, Grid, Node, Automatizace, Testování, Webové aplikace

Abstract

This master's thesis is based on automated web applications testing with freeware instrument utilization developed under Google code project. It is called Selenium webdriver and become very popular during last years. Selenium webdriver is frequently used for web pages automation. Indisputable advantages of this solution are low entry cost, selenium is broadly used and also live community around that instrument with many useful informations. Main goal of this thesis is creation of automation framework created by Java programming language. This final framework will integrate Selenium webdriver and guarantee easy creation of testing scenarios. Because of this goal is placed mainly on separation of web page programmatic description and description of testing scenario.

Keywords

Selenium, Webdriver, Java, Hibernate, Oracle, Grid, Node, Automation, Automatizace, Web applications, Testing

Bibliografická citace:

- SIEBER, L. *Návrh frameworku pro automatické testování webových stránek*. Brno: Vysoké učení technické v Brně, Fakulta elektrotechnologie, 2014. 85 s. Vedoucí diplomové práce Ing. Martin Frk, Ph.D.

Prohlášení

- Prohlašuji, že svou diplomovou práci na téma automatizace testování webových stránek s využitím nástroje Selenium, jsem vypracoval samostatně pod vedením akademického vedoucího a externího vedoucího diplomové práce a také s použitím odborné literatury včetně dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce. Jako autor uvedené semestrální práce dále prohlašuji, že v souvislosti s vytvořením této semestrální práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

V Brně dne: **10. května 2014** …………………………

podpis autora

Poděkování

- Děkuji vedoucímu své diplomové práce Ing. Martinu Frkovi, Ph.D. a také externímu vedoucímu Bc. Zdeňku Doležalovi za účinnou metodickou, pedagogickou a odbornou pomoc a další cenné rady při zpracování mé diplomové práce.

V Brně dne: ....

podpis autora

Obsah

Úvod 1

1 Dělení funkčního testování 2

1.1 Rozdělení na základě znalosti testovaného systému 2

1.1.1 Black box systém 2

1.1.2 White box systém 3

1.1.3 Gray box systém 3

1.2 Rozdělení na základě šíře testování 4

1.2.1 Funkční testování 4

1.2.2 Unit testování (jednotkové testování) 5

1.2.3 Integrační testování (SIT - System integration test) 5

1.2.4 Systémové testování 5

1.2.5 Systémové integrační testování 5

1.2.6 Akceptační testování (UAT – User acceptance test) 6

1.2.7 Regresní testování (Regression test) 6

1.2.8 Nefunkční testování 6

2 Dokumentace v testování 7

2.1 Norma ISO/IEC 9126 7

3 Service oriented architecture 9

4 Programovací jazyk java 10

5 Selenium 12

5.1 Selenium WebDriver 13

5.2 Selenium Grid 15

6 Page object pattern 17

7 Využívané nástroje 21

7.1 Integrované vývojové prostředí Eclipse 21

7.2 Nástroj Apache Maven 22

7.3 TestNG 26

7.4 Google Guice 28

7.5 Hibernate 32

7.6 AspectJ 37

7.7 Relační databáze MySQL 39

8 Tvorba frameworku – jádro 40

8.1 Stanovení jmenné konvence 40

8.2 Dědičnost ... 42
8.3 Pomocné servisní třídy ... 43
8.4 Shrnutí obsahu jádra ... 43
9 Tvorba frameworku – testy ... 45
9.1 Stanovení pravidel pro tvorbu testů ... 45
9.2 Javadoc komentáře a Fluent interface ... 46
9.3 Spuštění testu ... 47
9.4 Vytvoření závislosti mezi testy ... 48
9.5 Vyhodnocení výsledků spuštěného testu ... 50
10 Tvorba frameworku – reporting ... 52
10.1 Tvorba tříd popisujících reporting ... 52
10.2 Konfigurace AspectJ frameworku ... 56
10.2.1 Vytvoření vstupních bodů pro logování ... 56
10.2.2 Vytvoření konfiguračního souboru pro AspectJ ... 59
10.3 Pohled do reportingové databáze ... 60
10.4 Tvorba užitečných SQL příkazů ... 62
11 Využití jiných prohlížečů a parametrizace ... 63
11.1 Parametrizace ... 63
11.2 Spuštění testu v prohlížeči Internet Explorer ... 64
12 Selenium Grid prakticky ... 67
13 Návrh na zlepšení: Jenkins a SVN ... 70
13.1 Apache Subversion ... 70
13.2 Jenkins ... 70
14 Praktické ověření funkčnosti ... 72
14.1 Popis vytvořených testů ... 72
14.1.1 VUTNavigation ... 72
14.1.2 VUTLogin ... 72
14.1.3 VUTCommonInformation ... 73
14.1.4 VUTMessages ... 73
14.1.5 VUTMicroTransaction ... 73
14.1.6 VUTPersonalSchedule ... 73
14.1.7 VUTSchedule ... 73
14.2 Porovnání trvání testů v rámci IE a FF ... 74
14.3 Vytvoření testovací suity ... 74
14.4 Rozklad vybraných scénářů na jednotlivé kroky ... 75

14.4.1 VUTCommonInformation 75
14.4.2 VUTMessages 75
14.4.3 VUTNavigation 76
Závěr 78
Bibliografie 79
Seznam obrázků 81
Seznam tabulek 84
Dodatek A 85
Soupis potřebného software pro korektní funkci frameworku 85

ÚVOD

V dnešním uspěchaném světě, kdy je kladen stále větší a větší důraz na kvalitu a rychlost odvedené práce, nabývají nástroje pro hromadnou automatizaci testování stále markantnějšího významu. Doby manuálního testování v oblastech umožňujícím hromadné nasazení automatizace pomalu končí. A to nejen z důvodu rychlosti, či přesnosti, ale převážně ve jménu kvality odvedené práce, kdy monotónnost takového počínání značně ovlivňuje soustředění testera. Vytváří tak tedy prostor pro chyby z nepozornosti. Přesně těmto nedostatkům, vzniklých v důsledku jmenované nepozornosti způsobené stejnorodostí práce, se úspěšně daří předcházet automatizovanému testování.

Jmenovaná výhoda tkvící v „neúnavnosti" automatizovaného nástroje a tím tedy v homogenitě prováděných testů, není pouze jedinou výhodou. Dalším aspektem, který zde zmíníme, je rychlost takovéhoto testu. Samozřejmostí je také možnost spuštění více stejných, případně rozdílných testů a to zcela bez nutnosti nepřiměřeného zvyšování personálních kapacit u firmy, jež testování provádí. Je zřejmé, že v určitých oblastech se automatizované testování stává silně rentabilní činností, která šetří nemalé náklady na lidské zdroje.

Bylo by však nerozumné poukazovat pouze na výhody automatizovaného testování, jelikož manuální testování má své nezastupitelné místo. Například u testování úplně nových funkčností, při tzn. „free testech", či jednoduše řečeno díky zatím nemožné algoritmizaci lidské intuice. Je tedy jasné, že do kvalitních automatizovaných testů vchází obrovská dávka lidských zkušeností.

Cílem této diplomové práce je vytvoření funkčního frameworku, který usnadní tvorbu automatizovaných testů, přičemž je do budoucna kladen důraz na snadnou udržitelnost a rozšiřitelnost, která ve výsledku maximálně možně usnadní tvorbu automatických testů. A to hlavně z pohledu snadného převodu zkušeností softwarového testera do oblasti automatizovaného testu, v ideálním případě testerem samotným.

1 DĚLENÍ FUNKČNÍHO TESTOVÁNÍ

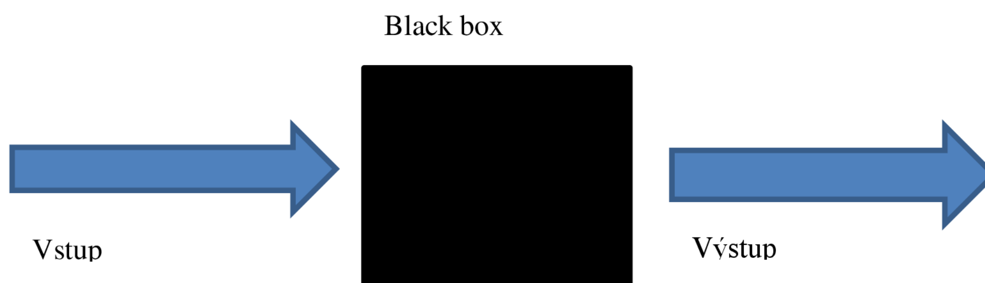
Následující kapitola se zabývá pohledem na rozdělení testování na základě různých kritérií. Posléze se zaměřuje na různé fáze (druhy) testování a snaží se přinést jednoduchý přehled pro lepší orientaci v problematice testování. Více [1] [2] [3]

1.1 Rozdělení na základě znalosti testovaného systému

V první řadě můžeme testování daného software (webové stránky) rozdělit na dva zcela odlišné směry. Toto dělení je založeno na znalostech fungování dané (webové) aplikace. Softwarový tester může mít hluboké znalosti o vnitřním fungování aplikace anebo naopak tyto znalosti postrádá. Na základě tohoto faktu můžeme přístup k testování rozdělit na tzv. „Black box“ testování, či „White box“ testování.

1.1.1 Black box systém

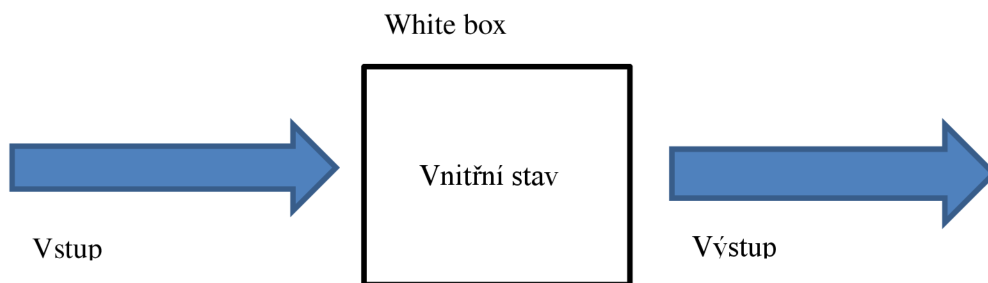
V případě testování založeném na technice „Black box“, nemá tester žádné (případně minimální) znalosti o vnitřním fungování aplikace. Testování v tomto případě sestává ze zadání vstupních dat a sledování očekávaného výsledku. Velikou výhodou tohoto přístupu k testování je diferenciace způsobu myšlení testera a softwarového vývojáře, který vytváří aplikaci. Je zde vysoká pravděpodobnost objevení chyb testerem, které vývojáři zcela unikají a to právě v důsledku naprosto rozdílného způsobu uvažování. Na druhou stranu se vystavujeme riziku nízké šíře testování systému a to právě v důsledku jeho absolutní neznalosti. Jinak řečeno tester si příliš neuvědomuje, kde se mohou nacházet rizika. Jako příklad uveďme konverze data a času, zadávání speciálních znaků do textových polí apod. Více [4]



Obrázek 1 – Reprezentace black box systému

1.1.2 White box systém

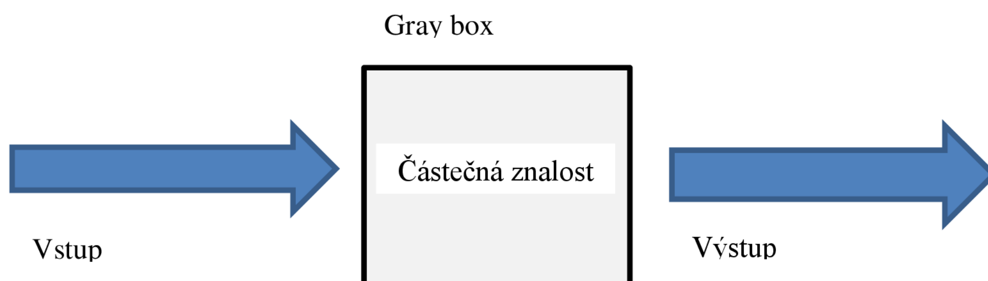
Přístup testování, který nese název „White box“, je zcela opačně od výše jmenované techniky založen na kvalitní a hluboké znalosti vnitřního fungování testovaného systému. Tester je obeznámen s principy fungování, programovacím jazykem, funkčním zapojením celého systému do globálního celku. Můžeme tedy říci, že člověk provádějící test je specializací daleko za možnostmi běžného testera. Testování touto metodou často spočívá ve spouštění určitých částí kódu s předem definovanými vstupy, podobně jako u předchozího přístupu, nicméně mimo výstupů jsou zde kontrolovány i hodnoty vnitřních proměnných a dalších sledovaných entit. Více [5]



Obrázek 2 – Reprezentace white box systému

1.1.3 Gray box systém

V případě testování aplikací, u nichž má tester pouze částečnou znalost o vnitřním uspořádání (fungování systému), hovoříme o tzv. „Gray box“. Krásným příkladem jsou webové aplikace, kdy je dostupný zdrojový kód webové stránky. Znalost této informace nám však v dnešním světě dynamických webových stránek poskytne pouze značně omezenou informaci o jejím fungování.



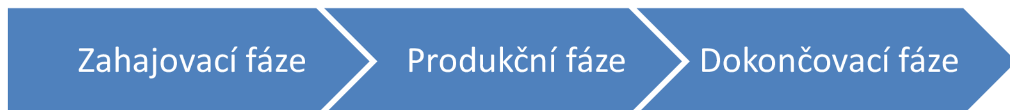
Obrázek 3 – reprezentace gray box systému

1.2 Rozdělení na základě šíře testování

Následující podkapitola sleduje dělení testů na základě šíře (hloubky) zapojení systému jako celku. Od testů, které provádí samotný vývojář, přes testy jednotlivých částí systémů, až po testy, při kterých je zapojen již systém jako celek.

Jednotlivé úrovně testování spadají do různých částí vývojové fáze daného softwarového projektu:

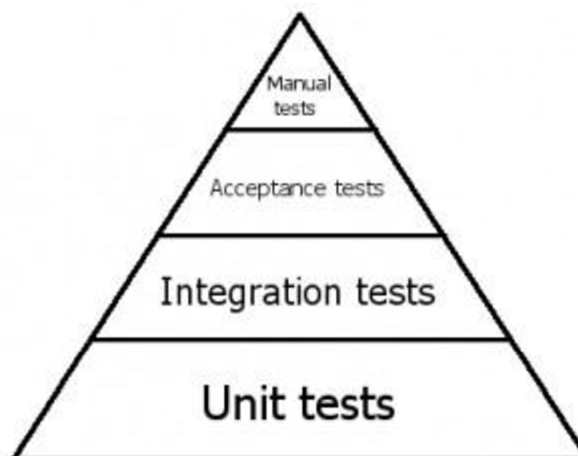
- **Zahajovací fáze** – čtení dokumentace, neexistuje funkční kód
- **Produkční fáze** – testování nových funkcí, testování celku dle funkční specifikace
- **Dokončovací fáze** – testování systému jako celku, nasazení software do produkce, příprava k prodeji



Obrázek 4 – jednotlivé fáze vývoje software z pohledu testování

1.2.1 Funkční testování

Jaké činnosti lze s vyvíjenou aplikací provádět nám určuje funkční specifikace z angl. „Function specification". A právě tyto činnosti podléhají důkladnému testování na základě přiložených funkčních specifikací. Výsledkem je porovnání požadované funkcionality a funkcionality skutečně implementované. V ideálním případě jsou funkce vyvíjeného softwaru přesně odpovídající svým požadovaným specifikacím.



Obrázek 5 – pyramida postupné posloupnosti testů převzato z [6]

1.2.2 Unit testování (jednotkové testování)

V rámci tohoto druhu testů jsou testovány velice malé části kódu, v případě vývoje s využitím objektově orientovaných jazyků (Java, C++, Delphi) se jedná o třídy. Pokud vývoj probíhá s užitím procedurálních jazyků (C, Basic), považuje se za nejmenší jednotku funkce. Výhodnost těchto testů je převážně v tom, že chybný kód je odhalen okamžitě u zdroje (vývojáře) a následně opraven před zpřístupněním kódu pro ostatní. Avšak funkčnost nejmenších částí kódu nám nezaručuje funkčnost systému jako celku. Existuje řada nástrojů, jež mírou usnadnění přímo nabádá k tvorbě unit testů. Samozřejmostí je vysoká míra jejich automatizace.

1.2.3 Integrační testování (SIT - System integration test)

Vzhledem k faktu, že je dnes většina webových systémů navržena s pomocí tzv. SOA (Service oriented architecture), jednotlivé bloky systémů společně zjednodušeně komunikují skrze zasílání zpráv a odpovědí na ně (tzn. request – response). Je potřebné provádět testování jednotlivých systémů z globálního pohledu. V případě integračních testů provádíme převážně ověřování funkčnosti komunikačních rozhraní jednotlivých bloků celku. Jednotlivé moduly globálního systému můžeme postupně přidávat a systém testovat postupně, nebo můžeme zvolit globální přístup a testovat kompletně propojený systém. Výhoda přístupu postupného přidávání zapojených systémů je hlavně v jednodušší lokalizaci komplikace. I v tomto případě se jedná o testy prováděné převážně vývojáři s vysokou mírou automatizace.

1.2.4 Systémové testování

Jakmile jsou dokončeny veškeré komponenty celku a dojde k jejich propojení, vzniká komplexní systém. U systémového testování hledáme odpověď na dvě zásadní otázky, a sice otázky validace a verifikace. Jedná se o hledání odpovědi na otázku: „Vyrábíme správný produkt?“ a následně na otázku: „Vyrábíme správný produkt správně?“. Jde tedy o závěrečné testování dokazující správnost funkce implementovaných možností systému, ale také o ověření, že systém obsahuje veškeré funkce z pohledu požadavků.

1.2.5 Systémové integrační testování

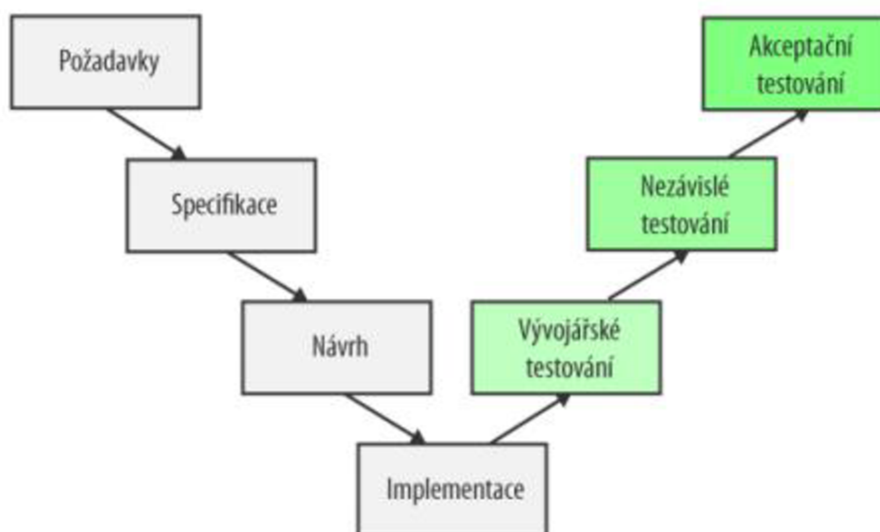
Jedná se o testování propojení aplikace s externími systémy, se kterými bude v průběhu své činnosti komunikovat.

1.2.6 Akceptační testování (UAT – User acceptance test)

Úkolem akceptačního testování je porovnat kvality dodaného softwaru oproti stanoveným požadavkům. Výsledkem akceptačního kola testů je rozhodnutí o přijetí, či nepřijetí výsledné verze dodaného systému. Tento druh testování probíhá již v prostředí zákazníka a nikoliv vývojové firmy. Výstupem z tohoto kola testů může být taktéž požadavek na přepracování. Podcenění této velmi důležité části testů, může mít fatální důsledek na fungování celého dodaného softwaru.

1.2.7 Regresní testování (Regression test)

Hlavním cílem regresního testování, jež stojí na úrovni unit, integračních nebo systémových testů je ověření, že v rámci změn aplikace nedošlo k žádným nežádoucím dopadům na ostatní funkčnosti softwaru. Jedná se převážně o stálé prověřování stěžejních funkčností vzhledem k prováděným změnám na softwaru jako celku. A to již bez ohledu na to, zdali se jedná o implementaci zcela nové funkčnosti, či jen úpravu stávající.



Obrázek 6 – pohled na testovací proces v širším záběru převzato z [7]

1.2.8 Nefunkční testování

Jak již název napovídá, u nefunkčního testování se zaměřujeme hlavně na oblast, která přímo nesouvisí s funkčností softwaru z pohledu funkční specifikace. Můžeme sem zařadit různé bezpečnostní testy, rychlost komunikace mezi databází a danou komponentou systému, zátěžové testování aplikace, grafické zpracování uživatelského rozhraní a mnoho dalších oblastí, které nebývají striktně definovány funkční specifikací. Více [8]

2 DOKUMENTACE V TESTOVÁNÍ

Testovací proces vytváří velké množství dokumentace sám o sobě, ať se již jedná o grafické znázornění výsledků, či popis jednotlivých testovacích scénářů. Tato kapitola si klade za cíl nám alespoň okrajově přiblížit dokumenty využívané k popisu testovacích scénářů. Mnohdy je firmou využíván specializovaný software umožňující správu, exekuci (spuštění) a statistické vyhodnocení jednotlivých testů.

-
- Testovací plán (Test plan) – Popisuje celý proces testování. Od rozsahu prováděného testování, přes jednotlivé alokace kapacit testerů, až po druh závěrečné prezentace výsledků. Testovací plán nám umožňuje držet se harmonogramu a průběžně reportovat postup, či provádět opatření v zájmu plánovaného ukončení testovacího kola.

- Testovací případ (Test case) – Jedná se o popis komunikace se systémem a to s přesně stanovenými vstupními daty i následným očekávaným výsledkem. V rámci testovacího případu je možné testovací kroky (test steps) provádět ručně, či plně automatizovat.

- Testovací scénář (Test scenario) – Je soubor testovacích případů, které se týkají společného tématu. Například testování jedné konkrétní funkce systému více způsoby.

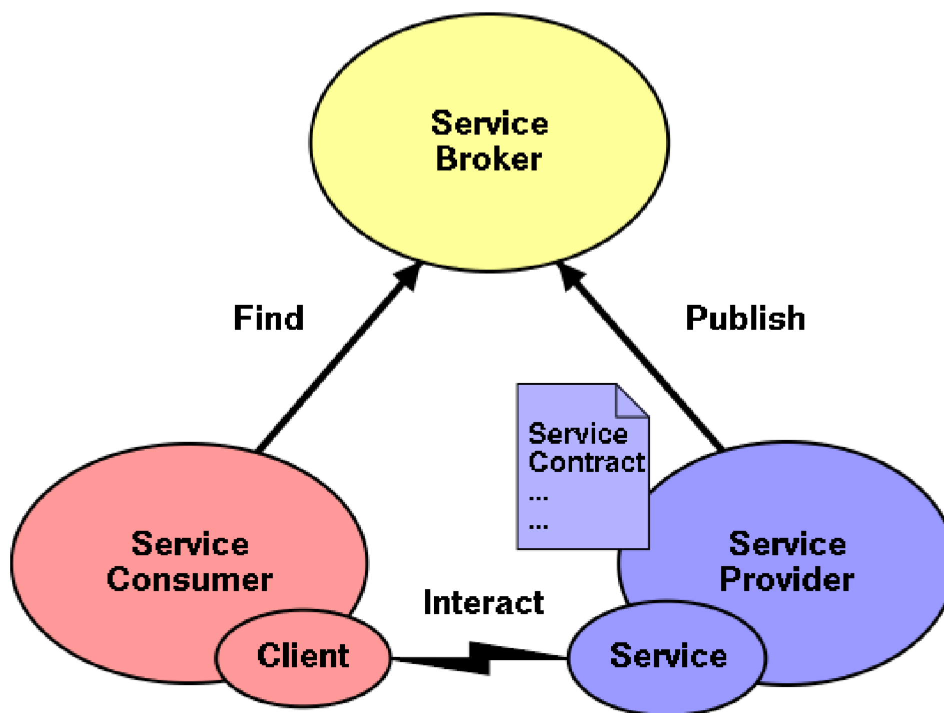
2.1 Norma ISO/IEC 9126

Jedná se komplexní pojetí sledování kvality softwaru, jeho neustálé hodnocení a samozřejmě zlepšování. V praxi se však dodržování této normy týká převážně softwaru vyvíjeného jako stěžejní produkt dané společnosti. S tímto přístupem se můžeme často setkat u vlajkových projektů softwarových gigantů, jako jsou například Oracle, Microsoft, či SAP. Avšak vnitřně vyvíjený software pro interní potřeby firem nezřídka tyto požadavky zdaleka neplní. Jsou sledovány převážně tato kritéria:

- **Funkčnost (Functionality)** – Je dodrženo chování systému dle funkční specifikace?
- **Použitelnost (Usability)** – Je systém uživatelsky přívětivý? Jak se zachová pro špatná vstupní data?
- **Spolehlivost (Reliability)** – Chová se systém stejně i při přetížení?
- **Výkon (Performance)** – Je systém dostatečně rychlý?

- **Podporovatelnost (Supportability)** – Jak složitá je instalace? Jaké jsou hardwarové nároky?
- **Lokalizovatelnost** – Lze systém snadno přeložit do jiného jazyka?
- **Kompatibilita** – Lze systém provozovat společně s jiným softwarem?
- **Bezpečnost** – Jsou uživatelská data dobře zabezpečena?
- **Přenositelnost** – Je aplikace schopná fungovat na jiném hardwaru?
- **Integrovatelnost** – Lze aplikaci začlenit do jiného systému?

Na tyto a mnohé další otázky je nutné se ptát pro vyhovění požadavkům jmenované normy. Cílem je vytvoření robustního softwaru, jež nám do budoucna zajistí nemalé úspory na reklamacích.



Obrázek 7 – ilustrace SOA architektury převzato z [9]

3 SERVICE ORIENTED ARCHITECTURE

Servisně orientovaná architektura je filozofie silně závislá na vytváření služeb, jež plní jednotlivé úkoly. V analogii k softwaru si zde službu můžeme představit jako samostatný blok komplexního systému, který zabezpečuje určitou činnost. K názorné ilustraci nám dobře poslouží tisková fronta. Představme si, že do takové tiskové fronty pošleme předepsaným způsobem (dle předpisu definovaného SOA architekturou) nový dokument určený k tisku. Avšak zanedlouho poté zjistíme, že nám tato komponenta tiskové fronty nevyhovuje a chceme ji vyměnit za jinou. Právě v tomto případě se setkáme s výhodou tohoto řešení. Komponenta se chová jako samostatná část, která komunikuje dle nastavených pravidel SOA architektury. Provedeme jedinou věc a tou je výměna nechtěné za novou komponentu. Je tedy zřejmé, že obrovská výhoda je zde zastoupena modularitou systému a možností dělení komplexního systému na menší samostatné bloky propojené pomocí předepsaného způsobu zasílání zpráv.

Mezi hlavní principy SOA architektury patří:

- **Standardizovaný kontrakt služby** – Již při návrhu je známo, co bude daná služba přijímat a poskytovat svému okolí.
- **Slabé vazby mezi komponentami** – Je snahou dosáhnout nejmenší možnou závislost jednotlivých služeb na sobě.
- **Princip abstrakce** – Skrývání implementačních detailů, přístup ke službě jako k černé krabici se známým rozhraním.
- **Znovupoužití** – Služby se nevytváří specializovaně jen pro daný projekt.
- **Nezávislost** – Služba má vnitřní logiku pro zajištění potřebných zdrojů.
- **Bezestavovost** – Služba nedrží žádný vnitřní stav.
- **Princip skládání** – Komplexní služba vzniká skládáním menších služeb.

Na obrázku 7 vidíme grafické zpracování architektury. Vystupuje zde několik entit. V první řadě je to entita „konzumenta služby“, jedná se například o uživatele požadujícího vytištění dokumentu (pro zachování celistvosti s výše uváděným příkladem). Další entitou je tzv. prostředník. Zde je důležité říci, že tato komponenta není nutná pro správnou funkci architektury, avšak velice usnadňuje vyhledávání chyb. Velmi jednoduše řečeno se jedná o „chytrý program“, přes který proudí všechny zprávy odesílané a přijímané jednotlivými komponentami. A konečně poslední komponentou je zde tzv. poskytovatel služby. V našem konkrétním příkladu je to tisková fronta, jež vystavuje své rozhraní v podobě servisního kontraktu.

4 PROGRAMOVACÍ JAZYK JAVA

Jedním z nejvíce oblíbených jazyků na světě, je právě programovací jazyk Java. Jeho užití se rozpíná přes čipové karty (platforma Java Card), mobilní telefony (platforma Java Mobile Edition), osobní počítače (platforma Java Standard Edition) až po korporátní projekty (platforma Java Enterprise Edition). Jazyk Java vznikl v roce 1995 a jeho autorem je James Gosling, tehdy pracující v Sun Microsystems (dnes koupeno firmou Oracle). V roce 2007 byly uvolněny zdrojové kódy jazyka s myšlenkou jeho dalšího vývoje jako open source projektu. Více [10] [11]

Obrázek 8 – logo programovacího jazyka Java

Základní vlastnosti jazyka:

- **Jednoduchost** – Syntaxe jazyka je zjednodušenou verzí syntaxe z C a C++.
- **Objektová orientovanost** – Mimo primitivní datové typy jsou veškeré ostatní reprezentovány objekty.
- **Distribuovatelnost** – Navržen pro podporu síťových aplikací.
- **Interpretovatelnost** – Není vytvářen přímo strojový kód, ale tzv. byte kód, čímž se dosahuje nezávislosti na platformě.
- **Robustnost** – Nejsou umožněny příkazy typu *goto*, *pointery* apod.
- **Generační správa paměti** – Paměť prohledává garbage collector a automaticky uvolňuje již nepotřebné objekty.
- **Přenositelnost** – Je předem určena lehká převoditelností mezi platformami.
- **Výkonnost** – Just In Time kompilace, HotSpot kompilace
- **Více vláknovost** – Vestavěná podpora pro tvorbu multi threadových aplikací.
- **Dynamičnost** – Knihovny funkcí mohou být doplňovány o nové funkce za běhu programu.

Právě díky své robustnosti a přenositelnosti byl tento programovací jazyk zvolen pro implementaci vytvářeného frameworku. Cílem tohoto počinu je maximální využití dostupných informací rozsáhlé komunity a také fakt, že samotný framework Selenium je především vyvíjen v jazyce Java. Pro zajímavost ještě dodejme, že k logu jazyka Java na obrázku 8 se váže úsměvná poznámka často vyslovována programátory jazyka C++ „Symbol znamená, že máte čas si udělat kávu, než vůbec dojde ke spuštění programu“.

```
public class HelloWorld {
    public static void main(String[] args) {
        Pozdrav pozdrav = new Pozdrav("Ahoj světe!");
        pozdrav.print();
    }
}

class Pozdrav {
    private String text;

    public Pozdrav(String pozdrav) {
        text = pozdrav;
    }

    private String getText() {
        return text;
    }

    public void print() {
        System.out.println(getText());
    }
}
```

Obrázek 9 – ukázka Java kódu

5 SELENIUM

Jedná se o soubor nástrojů pro automatizaci webového prohlížeče. Celý tento balík je vyvíjen pod Apache 2.0 licencí, což je velice výhodné, protože umožňuje komerční využití bez nutnosti za daný software platit. Avšak je nepsaným pravidlem, že pokud (korporátní) firmy takový software využívají, je slušností přinejmenším podpořit jeho vývoj, případně jiným způsobem přispět ke zvýšení kvality daného softwaru.

Historie Selenia se datuje do roku 2004, kdy Jason Huggins vytvořil jádro projektu „Java Script Test Runner" ve firmě ThoughWorks. Jason postupně ukazoval svůj projekt kolegům, které tím naprosto nadchnul. Jejich nadšení způsobil hlavně potenciál pro zlepšování dané aplikace, které již v té době představovala přidanou hodnotu pro testování webových aplikací. Toto nadšení otevřelo otázku vytvoření projektu Selenia jako open source. Později začíná veliké nadšení pro Selenium být sdíleno i mimo firmu ThoughWorks. V první řadě vzniká Selenium RC, což je samostatná serverová aplikace ovládající internetový prohlížeč, která vykonává akce požadované klientem. Nespornou výhodou tohoto přístupu je multiplatformnost. Později vzniká v Japonsku projekt Selenium-IDE šířený jako doplněk prohlížeče FireFox. Výhodou této implementace je možnost nahrávat operace v prohlížeči a následně je opakovaně spouštět. Nevýhodou je závislost na konkrétním prohlížeči. Posledním krokem vývoje bylo vytvoření Selenium Webdriveru zaměstnanci společnosti Google, kdy došlo ke kompletnímu přepsání kódu Selenia a dnes se těší stále větší oblibě. Do Webdriveru byla implementována zpětná kompatibilita se Seleniem RC a navíc přidána komponenta zvaná Selenium Grid, jež umožňuje vzdálenou distribuci exekuce testů. Více [12]

Obsah balíku pod názvem Selenium:

- **Selenium-IDE** – Doplněk do prohlížeče FireFox
- **Selenium-RC** – Serverová aplikace pro ovládání prohlížeče skrze příkazy od klienta
- **Selenium-WebDriver** – Modernizovaná verze aplikace pro automatizaci webového prohlížeče
- **Selenium-Grid** – Aplikace zajišťující distribuci spouštěných testů na vzdálené servery

Vzhledem k současným trendům vývoje se práce především soustřeďuje na široce využívané komponenty, které jsou obsaženy v balíku Selenium a to přesně Selenium WebDriver, popřípadě Selenium Grid pro pozdější realizaci paralelizace testování.

5.1 Selenium WebDriver

S příchodem WebDriveru se nám do rukou dostává oproti Selenium RC řada nesporných výhod. V dřívějším pojetí bylo nutné nejdříve spustit serverovou aplikaci Selenium RC, které následně fungovala jako proxy server (Komunikace byla nejdříve směřována přes tento server a teprve do prohlížeče.) internetového prohlížeče, který samostatně spouštěla a vypínala. Jinak řečeno komunikace probíhala přes GET a POST metody jazyka HTML. Jasnou výhodou byla nezávislost na platformě, protože ovládat tuto serverovou aplikaci bylo možné z mnoha programovacích jazyků, pro něž byly dostupné knihovny funkcí. Avšak zásadní nevýhodou byla určitá obecnost specifikace v zájmu zachování funkčnosti v nejběžněji používaných prohlížečích. Nebylo tak například možné operovat s „add-on“ prvky FireFoxu v zájmu zachování kompatibility s Internetem Explorerem.

```
class VUTPage
{
 static void Main(string[] args)
 {
 WebDriver driver = new FirefoxDriver();

 driver.Navigate().GoToUrl("http://www.vutbr.cz/");

 WebElement query = driver.FindElement(By.Name("login"));

 query.SendKeys("xsiebe03");

 query.Submit();
 }
```

Obrázek 10 – ukázka zdrojového kódu s užitím Selenium WebDriver

Tento nedostatek řeší novější nástroj Selenium WebDriver. Již není potřebné spouštět samostatnou serverovou aplikaci, ale pouze vytvořit entitu driveru pro požadovaný prohlížeč. Další nespornou výhodou je oddělenost takto vzniklých entit řídících daný prohlížeč oproti starší verzi, kde existoval pouze jediný server řídící veškeré prohlížeče. Cenou za tuto specializaci je nutnost upravovat každou entitu ovládající konkrétní prohlížeč zvlášť. Poslední větu si lze představit tak, že je nutné vytvořit objekt popisující funkcionalitu prohlížeče FireFox, Chrome, nebo například Internet Explorer.

Obrázek výše zachycuje syntaxi kódu v Javě využívanou v případě užití aplikace Selenium WebDriver. Můžeme si všimnout, že nejdříve dojde k vytvoření instance driveru, zde se konkrétně jedná o instanci pro FireFox. V dalším kroku již můžeme takto vytvořenou instanci využít pro ovládání prohlížeče. Povšimněme si také intuitivnosti implementovaných metod, například volání *driver.Navigate().GoToUrl(...)* je velmi sebe popisující. Interface WebDriveru nás tedy vede v možnostech, které lze s danými objekty provádět. Nyní si představme situaci, kdy chceme stejný skript spustit v jiném prohlížeči. Postačí nám k tomu velice snadná změna inicializace instance driveru, tedy místo *FirefoxDriver()* vytvoříme instanci *InternetExplorerDriver()* a v tom je celé kouzlo úspěchu.

```
class VUTPageLogin
{
 static void Main(string[] args)
 {
 WebDriver driver = new FirefoxDriver();

 driver.Navigate().GoToUrl("http://www.vutbr.cz/login/");

 WebElement user = driver.FindElement(By.id("login7"));
 WebElement pass = driver.FindElement(By.id("passwd7"));
 WebElement Subm = driver.FindElement(By.name("submit"));

 user.SendKeys("xsiebe03");
 pass.SendKeys("123456");
 Subm.click();
 }
```

Obrázek 11 – přihlášení na www.vutbr.cz

Na obrázku 11 můžeme vidět funkční skript pro přihlášení do informačního systému VUT. I když zdrojový kód čte člověk programováním nepoznamenaný, velmi pravděpodobně dojde k závěru, že zde figuruje internetový prohlížeč, jsou vyhledávány prvky na stránce, odesílány stisknutí kláves a následné kliknutí na tlačítko. Je příjemným zjištěním, že v případě WebDriveru se podařilo vytvořit uživatelsky přívětivé API (Application Interface), jinými slovy rozhraní, které je viditelné pro uživatele. Více [13]

5.2 Selenium Grid

Již jsme nastínili možnosti Selenium WebDriveru v oblasti tvorby testů pro webové aplikace. Avšak pokud takový test budeme schopni spouštět jen na lokálním počítači, bude naše snaha o tvorbu robustního frameworku podobná příslovečnému „Nošení dříví do lesa“. Požadavky pro úspěšnou automatizaci jsou jasně dány a patří mezi ně především schopnost testovat na různých platformách, verzích prohlížeče, či jednoduše potřeba rychlého spuštění většího množství testů. A právě v podobě Selenium Gridu se nám dostává „out of box“ řešení pro tyto účely.

Tento nástroj umožňuje spuštění serveru (hubu) zajišťující distribuci testů, které jsou požadovány od klientů na vzdálené stroje (nody). Jsme tedy schopni vytvořit testovací infrastrukturu pro hromadné spouštění testů. Vzdálené testovací stroje mohou díky implementaci Gridu v jazyce Java běžet na libovolném operačním systému, pro který je dostupná tzv. JVM (Java virtual machine), neboli prostředí nutné pro běh programu vytvořeného v jazyce Java. Seleniu Grid jde však ještě o pár kroků dál. Jsme schopni Gridovému serveru říci, že zde běží konkrétní verze prohlížeče a operačního systému. Pokud tedy přijde požadavek na spuštění testu za těchto podmínek, server automaticky distribuuje testy na tento vzdálený stroj (nod).

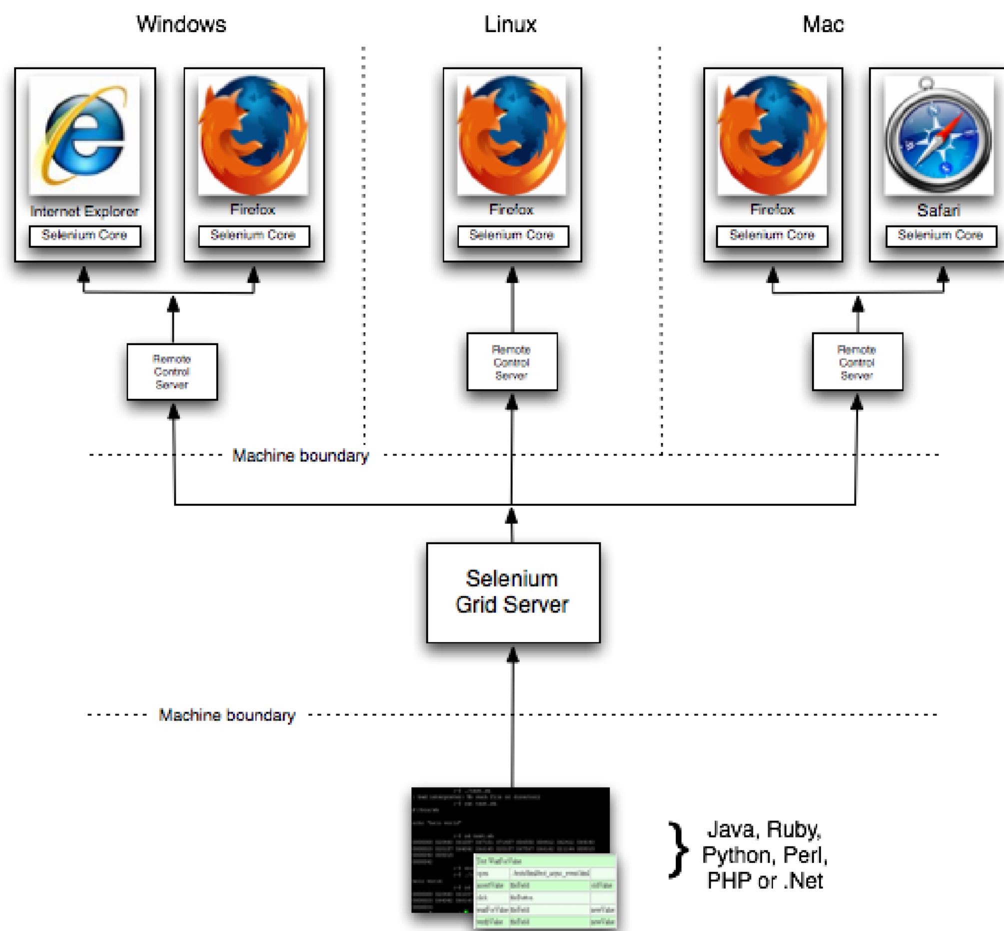
```
java -jar selenium-server-standalone-2.14.0.jar -role hub
```

Obrázek 12 – příkaz pro spuštění Selenium Gridu v módu serveru (hubu)

```
java -jar selenium-server-standalone-2.14.0.jar -role node  -hub
http://localhost:4444/grid/register
```

Obrázek 13 – spuštění Selenium Gridu v módu nodu na vzdáleném stroji (nodu)

Z obou obrázků výše je patrné, že při vývoji balíku Selenium je kladen velký důraz na jednoduchost. V prvním případě totiž spouštíme centrální server (hub), který se bude starat o distribuci obdržených požadavků na testy na vzdálené stroje (nody). Obrázek 13 potom zachycuje jakým způsobem se spouští aplikace ovládající prohlížeč na vzdáleném počítači (nodu), který fyzicky test vykonává. A to nejlepší na závěr, v naší aplikaci ovládající průběh testu na obrázku 11, provedeme pouze změnu instance driveru na *RemoteDriver(„http://localhost:4444/grid/“).*



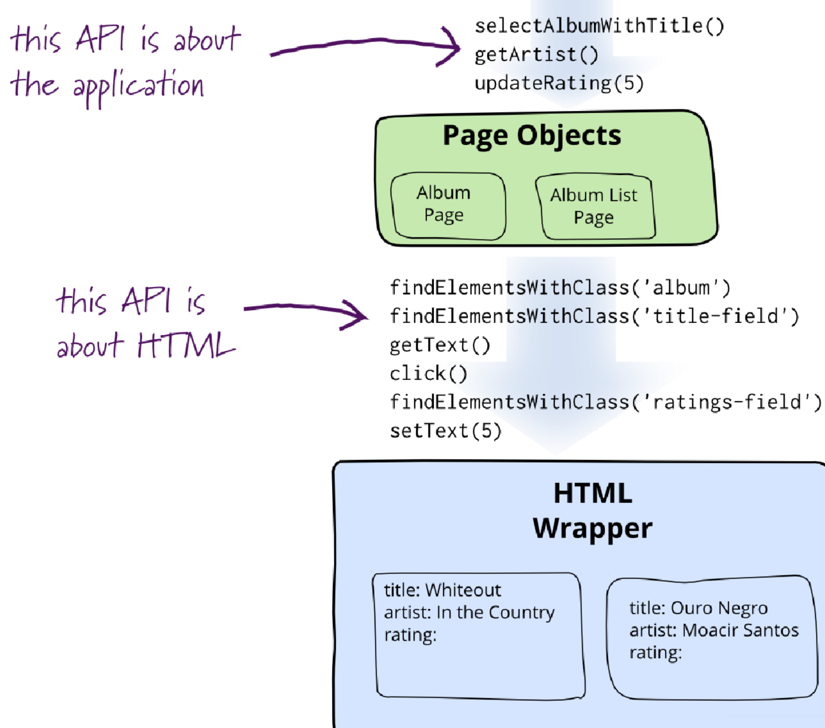
Obrázek 14 – grafické znázornění možného využití Selenium Gridu převzato z [12]

Po prohlédnutí obrázku 14, je jasné v čem spočívá přidaná hodnota tohoto řešení. Tvůrce testů nemusí řešit, jak daný test spustit na konkrétním stroji, on pouze vytvoří požadavky a Selenium Grid se pokusí takový stroj najít a test spustit. Mimo to je schopný koordinovat spouštění většího množství testů a to dokonce napříč platformami, nebo různými verzemi prohlížečů.

6 PAGE OBJECT PATTERN

Ve snaze o oddělení části popisu webových stránek převedeného do programovacího jazyka a vlastního popisu testovacího scénáře je nutné hledat optimální řešení. Vzhledem k použitému programovacímu jazyku je zřejmé, že webovou stránku bude reprezentovat objekt vystavující navenek proveditelné akce (metody). Abychom dosáhli znovu použitelnosti kódu, musíme objekt udržet bezestavový. V opačném případě by bylo nutné vždy vytvářet objekt nový a připravili bychom tím půdu pro vznik komplikací v případě multi-threadových užití našeho kódu.

Jedním ze zajímavých přístupů je vytváření tzv. Page Objektů. Tento způsob si lze představit jako zabalení funkčnosti webové stránky do třídy s následným vystavením proveditelných funkcí (možných interakcí), navenek v podobě metod třídy. Hlavní myšlenkou tohoto přístupu je vytvoření rozhraní třídy, které umožňuje se stránkou dělat přesně to, co může uživatel. Nenechejme se však zmást slovem „Page“ v názvu. Ne vždy se musí jednat o popis celé stránky, mnohdy se jedná pouze o popis její části. Využitím této metody se dostáváme o jednu abstraktní vrstvu výše, než jsou samotné příkazy WebDriveru. Pokud se do budoucna změní nějaká funkčnost WebDriveru jsme nuceni provést změnu pouze na jednom místě a veškeré testy využívající náš Page Object jsou tímto krokem opraveny. V této vědomosti tkví největší výhoda jmenovaného přístupu. Více [14]



Obrázek 15 – znázornění míry abstrakce při užití page objectu převzato z [14]

Nyní společně vytvoříme náš první jednoduchý Page object na základě výřezu stránky umožňující vyplnění a následné přihlášení se do systému VUT. Na obrázku níže je grafická reprezentace tohoto výřezu.

ABSOLVENTI | ECTS | E-PŘIHLÁŠKA | QS TOPUNIVERSITIES | ČASOPIS UDÁLOSTI | KNIHOVNY | INTRAPORTÁL

Úvod > Login

PŘIHLÁŠENÍ DO INFORMAČNÍHO SYSTÉMU VUT

Login (nápověda)

VUTheslo (nápověda)

Přihlásit se

Obrázek 16 – výřez stránky www.vutbr.cz/login/ pro účely vytvoření page objectu

Plně se soustředíme pouze na přihlašovací dialog a prezentuje na něm výhody reprezentace stránek za pomocí uvedené techniky. Z pohledu uživatele máme k dispozici následující činnosti:

- Vyplnění textboxu pro přihlašovací jméno
- Vyplnění textboxu pro přihlašovací heslo
- Potvrzení těchto údajů a vstup do systému tlačítkem „Přihlásit se"

Následujícím pseudokódem se pokusíme vytvořit programový popis page objectu, který bude ve výsledku sloužit pro pohodlnější tvorbu testů a zároveň jako abstraktní úroveň oddělující příkazy WebDriveru od programového popisu testu.

```
Public interface VUTLoginPage {

 Public VUTLoginPage fillLogin(String login);

 Public VUTLoginPage fillPassword(String password);

 Public VUTISMainPage submitLogin();

}
```

Obrázek 17 – implementace rozhraní page objectu pro VUT přihlašovací stránku

Interface (popis dostupných metod třídy) máme úspěšně dokončený. Říká nám, že pokud vytvoříme instanci třídy s tímto interfacem, bude možné volat právě veřejné (public) metody popsané na obrázku 17. Postupme však o krok dále a vytvořme společně i výslednou implementaci funkčnosti třídy. Předpokládejme, že funkce *getDriver()* navrací dříve vytvořenou instanci driveru.

```
Public class VUTLoginPageImpl implements VUTLoginPage {

 Public VUTLoginPage fillLogin(String login) {
WebElement loginBox =      getDriver().findElement(By.id("login"));
     loginBox.sendKeys(login);
     return this;
 }

 Public VUTLoginPage fillPassword(String password) {
WebElement passBox =      getDriver().findElement(By.id("pass"));
     passBox.sendKeys(password);
     return this;
 }
 Public VUTISMainPage submitLogin() {
WebElement passBox =
getDriver().findElement(By.id("submit"));
return new VUTISMainPage();
 }

}
```

Obrázek 18 – implementace funkčnosti page objektu

S úspěchem jsme vytvořili i implementaci funkčnosti našeho page objektu, který popisuje funkcionalitu stránky z pohledu uživatele. Čtenáře znalé programování může zajímat z jakého důvodu je v případě prvních dvou funkcí návratovou hodnotou objekt sám. Je to z důvodu užití další z výhod, které nám nabízí užití page objectu. Mluvíme zde o tzv. „fluent interface“, jehož přínos si prezentuje na ukázce kódu vlastního testu.

```
class VUTPageLogin
{
 static void Main(string[] args)
 {

 VUTLoginPage vutLoginPage = new VUTLoginPage ();

 vutLoginPage.fillLogin("xsiebe03")
             .fillPassword("123456")
             .submitLogin();
 }
```

Obrázek 19 – testovací scénář vytvořený s využitím page objectu

Jmenovaný „fluent interface" nemusí být na první pohled patrný. Pro úplnost prozraďme, že v důsledku navrácení sebe sama, či jiného page objectu, je možné psát kód bez přerušení. Jednoduše řečeno při psaní kódu uděláme jen tečku za názvem objektu (zde vutLoginPage) a vývojové prostředí nám samo nabídne z dostupných metod. A protože daná metoda vrací stejný (nebo jiný) page object, je možné jen vybírat z nabízených možností a dokončit testovací scénář tímto způsobem. Mimo jmenované výhody vidíme i značné zkrácení zápisu testu.

7 VYUŽÍVANÉ NÁSTROJE

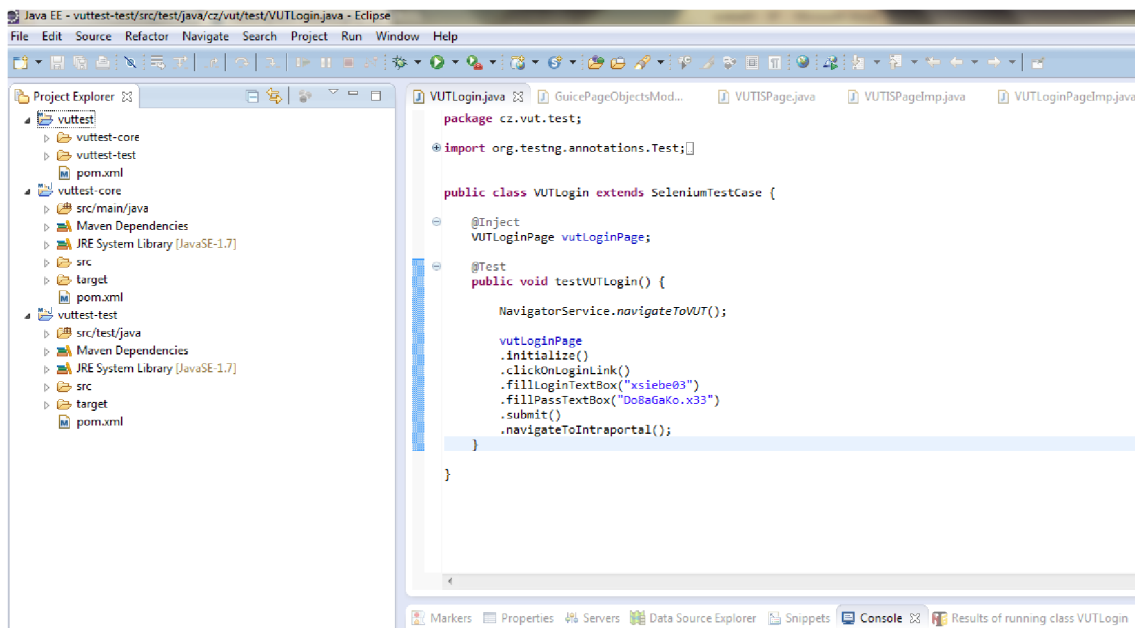
Následující části práce nás seznámí s celkovým návrhem automatizačního frameworku, propojením jeho jednotlivých komponent a návrhem reportingového rozhraní. Kladou si za cíl provést čtenáře postupnými kroky kompletního návrhu s důrazem na pochopení smyslu a principů napojení jednotlivých komponent vytvářeného celku.

7.1 Integrované vývojové prostředí Eclipse

První verze tohoto IDE (integrovaného vývojového prostředí) spatřila světlo světa uvolněním projektu od společnosti IBM. Tento počin byl později ohodnocen hodnotou 40 miliónů dolarů, avšak právě tento čin dal vzniknout kvalitnímu open source vývojovému prostředí. Hlavní filozofií Eclipse oproti konkurenčním nástrojům, je dodatečná rozšiřitelnost za pomocí zásuvných modulů (tzv. plug-inů). V základní verzi obsahuje prostředí jen to nejnutnější pro vývoj v programovacím jazyce Java, avšak po doplnění příslušných rozšiřujících modulů, se prostředí mění k nepoznání. Jsme následně schopni ovládat aplikační servery jako je například TomCat, vytvářet UML diagramy s pomocí grafického editoru, programovat v jazyku C++ a mnoho dalších možností. Od verze 3.0 implementoval vývojový team normu OSGi do filozofie vytváření doplňků, čímž učinil z tohoto vývojového prostředí atraktivní platformu. Hlavní výhodou architektury OSGi je nahrávání zásuvných modulů dynamicky až v čase využití. Je tedy patrné, že dosáhneme větší rychlosti prostředí, jelikož nejsou nahrávány nevyužívané doplňky. Více [15]

Volbu tohoto prostředí provázely následující požadavky:

- Open source prostředí s minimálními náklady na pořízení
- Dostatečně velká komunita uživatelů potvrzující kvalitu prostředí
- Množství dostupných příkladu na internetových fórech
- Množství dostupných doplňků
- Multiplatformní prostředí pro více OS
- Stabilita



Obrázek 20 – ukázka hlavního okna vývojového prostředí Eclipse

7.2 Nástroj Apache Maven

Jedná se o sofistikovaný nástroj určený pro řízení sestavení vytvořených programů (tzv. buildů) a také celkovou správu závislostí projektu. Můžeme si to představit takovým způsobem, že Mavenu pouze řekneme podstatné informace o našem projektu, jako jsou název, způsob zabalení (soubor jar, soubor war), používané JRE (Java runtime environment), další přidávané knihovny (balíčky jar, které obsahuji využívané funkce), a celý zbytek procesu je již zařízen za uživatele automaticky. Jinak řečeno nemusíme se starat o distribuci doplňkových knihoven funkcí, čí správu jejich verzí, ale toto vše je zařízeno automaticky. Slovo Maven pochází z Jidišského jazyku a znamená „znalec“. Hlavním impulzem pro vznik tohoto nástroje byla snaha o znovu použitelnost vytvořených skriptů, která u dříve populárního nástroje Apache Ant, nebyla zcela dokonalá. K lepšímu pochopení poslouží následující komentovaný obrázek pod textem, který reprezentuje konfigurační soubor využívaný Mavenem. Více [16].

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
        <modelVersion>4.0.0</modelVersion>

        <groupId>cz.vut</groupId>
        <artifactId>vuttest</artifactId>
        <version>1.0-SNAPSHOT</version>
        <packaging>pom</packaging>

        <name>vuttest</name>
        <url>http://maven.apache.org</url>

        <properties>
                <project.build.sourceEncoding>UTF-
8</project.build.sourceEncoding>
        </properties>

        <modules>
                <module>vuttest-core</module>
                <module>vuttest-test</module>
        </modules>

        <dependencies>
                <dependency>
                        <groupId>junit</groupId>
                        <artifactId>junit</artifactId>
                        <version>3.8.1</version>
                </dependency>
                <dependency>
                        <groupId>org.testng</groupId>
                        <artifactId>testng</artifactId>
                        <version>6.8</version>
                </dependency>
                <dependency>
                        <groupId>com.google.inject</groupId>
                        <artifactId>guice</artifactId>
                        <version>3.0</version>
                </dependency>
                <dependency>
                        <groupId>com.google.inject.extensions</groupId>
                        <artifactId>guice-assistedinject</artifactId>
                        <version>3.0</version>
                </dependency>
        </dependencies>
</project>
```

Obrázek 21 – ukázka konfiguračního souboru Mavenu

Základní princip fungování Mavenu je založen na tzv. Project Object Modelu (krátce řečeno POM). Tento model popisuje softwarový projekt z pohledu zdrojového kódu, závislosti na externích knihovnách, procesu přeložení, kompilace a výsledného výstupního souboru. Avšak zde možnosti nekončí, je možné popsat i spuštění jednotkových testů, tvorbu reportů a mnoho dalšího. Na obrázku 21 si můžeme všimnout postupně různých částí. V první řadě se jedná o jednoznačný popis projektu, který následuje ihned po určení verze POM modelu. V další části uvádíme jméno projektu, jeho vlastnosti (properties) a dále (Protože se jedná o mateřský POM soubor)

jeho potomky, kteří obsahují obdobný popis projektu. V části závislostí uvádíme potřebné externí knihovny, které budou staženy z repozitáře Mavenu. Jmenovaný nástroj umožňuje také používání tzv. pluginů, které dále rozšiřují jeho možnosti. Například si jmenujme kompilační plugin, který pro nás zajistí přeložení zdrojového kódu. Na obrázku 22 a 23 se podíváme, jak vypadají oba POM soubory potomků a vysvětlíme si význam jednotlivých částí.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
	<modelVersion>4.0.0</modelVersion>

	<groupId>cz.vut.core</groupId>
	<artifactId>vuttest-core</artifactId>
	<packaging>jar</packaging>

	<name>vuttest-core</name>
	<url>http://maven.apache.org</url>

	<properties>
		<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
	</properties>

	<parent>
		<groupId>cz.vut</groupId>
		<artifactId>vuttest</artifactId>
		<version>1.0-SNAPSHOT</version>
	</parent>

	<dependencies>

		<dependency>
			<groupId>org.seleniumhq.selenium</groupId>
			<artifactId>selenium-java</artifactId>
			<version>2.37.0</version>
		</dependency>

	</dependencies>

	<build>
		<plugins>
			<plugin>
				<groupId>org.apache.maven.plugins</groupId>
				<artifactId>maven-compiler-plugin</artifactId>
				<version>3.1</version>
				<configuration>
					<source>1.7</source>
					<target>1.7</target>
				</configuration>
			</plugin>
		</plugins>
	</build>
</project>
```

Obrázek 22 – POM soubor k VUT Core projektu

V první řadě na obrázku výše, pomineme-li úvodní částí, které známe již z mateřského souboru, vidíme definici nadřazeného mateřského souboru. Následuje část závislostí (dependencí) projektu jádra frameworku. A konečně v poslední částí oddělené párovým tagem build, se nalézá část ovlivňující samotné přeložení projektu. Nalezneme zde například plugin spouštějící kompilátor a rovnou definující verzi Javy pod kterou se má projekt přeložit.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
	<modelVersion>4.0.0</modelVersion>

	<groupId>cz.vut.test</groupId>
	<artifactId>vuttest-test</artifactId>
	<packaging>jar</packaging>

	<name>vuttest-test</name>
	<url>http://maven.apache.org</url>

	<properties>
		<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
	</properties>

	<parent>
		<groupId>cz.vut</groupId>
		<artifactId>vuttest</artifactId>
		<version>1.0-SNAPSHOT</version>
	</parent>

	<dependencies>
		<dependency>
			<groupId>cz.vut.core</groupId>
			<artifactId>vuttest-core</artifactId>
			<version>1.0-SNAPSHOT</version>
		</dependency>
	</dependencies>

	<build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.1</version>
        <configuration>
          <source>1.7</source>
          <target>1.7</target>
        </configuration>
      </plugin>
    </plugins>
  </build>

</project>
```

Obrázek 23 – POM soubor k VUT Test projektu

Obrázek 23 reprezentuje obsah POM souboru posledního projektu. Kromě již známých části, zde stojí za povšimnutí fakt, že jednotlivé projekty na sobě mohou být

závislé. Z tohoto faktu vyplývá jedna úžasná věc, jsme schopni dodat přeloženou knihovnu jádra, kde bude ukryto celé naše „know how“ a tato knihovna bude pomocí Mavenu staženo do projektu testů. Dokonce jsme schopni vytvořit zcela odlišná jádra projektu s různou funkčností, u kterých zachováme pouze stejné názvy metod tříd. A následně díky tomuto počinu jsme schopni pouhou změnou závislosti v POM souborů testů, zcela změnit funkčnost a to bez nutnosti, či s minimální úpravou testů. Například testování webové a následně androidí aplikace provádějící stejnou činnost.

7.3 TestNG

TestNG není ničím jiným, než komplexním testovacím frameworkem vycházejícím z velice populárního a dříve hojně využívaného JUnitu. Cílem tohoto počinu je pokrýt celou oblast testování od unit testů, přes funkční testy až k testům integračním. A vytvořit silný nástroj s širokou funkcionalitou, které povede ke snadnému vytváření a vyhodnocování spouštěných testů. Nyní si pojďme vypsat hlavní přednosti TestNG a následně si k nim doplníme bližší informace. Více [17]

Hlavní funkcionalita:

- Podpora anotací
- Podpora parametrizace a řízení testů vstupními daty
- Podpora pro více různých instancí jedné třídy testu
- Flexibilní model spouštění (IDE, Maven, ANT)
- Paralelní spouštění testů
- Vytváření závislostí mezi testy

Anotace je označení třídy, či proměnné uvozené počátečním znakem „@“ následovaným názvem. Hlavním smyslem tohoto konceptu je schopnost jednoduchým a pro člověka čitelným způsobem označovat například třídu, či metodu testu (Část kódu, kterou TestNG bude spouštět a vyhodnocovat). V rámci podpory parametrizace jsme schopni řídit testy vstupními daty. Jsme tedy schopni testovat různé případy a to s očekávaným kladným i záporným výsledkem. Podporou více instancí jedné třídy se rozumí využití generických tříd (Tříd vytvářených z nějakého šablonového popisu pro konkrétní datové typy). Flexibilita modelu spuštění je zde podchycena existencí pluginů do Mavenu, ANTu, či existencí rozšíření do integrovaného vývojového prostředí Eclipse, umožňující přímé spouštění testů z vývojového studia. Při paralelním spouštění testů se TestNG postará o korektní inicializaci a správu vláken. Mimo to díky využití anotací můžeme vytvářet funkce, u kterých je zajištěno automatické spuštění ve stanovené fázi inicializace testu. Poslední neméně důležitou součástí je možnost definovat závislostí testů mezi sebou, jinak řečeno jsme schopni požádat framework o

spuštění daného testu jen při dosažení stanovené podmínky. Například testovat rezervační systém lze až po úspěšném přihlášení do něj. V tomto případě bude tedy testování funkčnosti rezervačního systému závislé na úspěšném proběhnutí testu přihlášení.

```
public class VUTLogin extends SeleniumTestCase {

    @Inject
    VUTLoginPage vutLoginPage;

    @Test
    public void testVUTLogin() {

      NavigatorService.navigateToVUT();

      vutLoginPage
      .initialize()
      .clickOnLoginLink()
      .fillLoginTextBox("xsiebe03")
      .fillPassTextBox("xxxxxx")
      .submit()
      .navigateToIntraportal();
    }

}
```

Obrázek 24 – ukázka třídy testu s využitím TestNG

Obrázek 24 zachycuje celou třídu testu frameworku TestNG. Při bližším prozkoumání zjistíme, že je zde definována třída „VUTLogin“, jež je rozšířena třídou „SeleniumTestCase“, avšak o té pojednáme více až později. Nyní si povšimneme anotace „@Test“, která uvozuje metodu (funkci uvnitř třídy) určenou ke spuštění frameworkem TestNG, jež vykonává operace námi vytvářeného frameworku. Obrovskou výhodou je právě schopnost takto snadného označení testovacích tříd, či metod. V podobném duchu maximální praktičnosti se dají využití i ostatní dříve jmenované výhody tohoto frameworku pro správu testů.

Teď se společně pojďme podívat na obsah rozšiřující třídy „SeleniumTestCase“, kterou nám blíže představí obrázek 25. Zde se zaměříme hlavně na dvě anotace, které vidíme nad metodami „beforeMethod“ a „afterMethod“. Jedná se o schopnost TestNG zaručit vykonání určitého kódu před a případně po spuštění nějaké metody testu. Jsme

tím pádem schopni zajistit například inicializaci vstupních proměnných, či jiných pro test důležitých podmínek a také následně v druhém kroku po proběhnutí testů uklidit. Ohledně dalších anotací, které zde můžeme vidět, bude pojednávat další kapitola věnovaná frameworku pro injekci dependencí.

```
@Guice(modules = GuicePageObjectsModule.class)
public class SeleniumTestCase {
    private final static Log log =
LogFactory.getLog(SeleniumTestCase.class);

    @BeforeMethod
    public void beforeMethod(ITestContext context) {

      DriverFactory.initDriver();

      log.info("------------------ VUTTest starting.-------------");

    }

    @AfterMethod
    public void afterMethod() {
      DriverFactory.getDriver().quit();
    }

}
```

Obrázek 25 – Obsah třídy „SeleniumTestCase“

7.4 Google Guice

Nástroj, jehož jméno se můžeme dozvědět v nadpise této kapitoly, slouží k tzv. „dependency injection“ neboli českým ekvivalentem injekci závislostí. Jedná se o techniku užívanou v objektově orientovaném programovacím modelu, která slouží k vkládání závislostí mezi jednotlivé komponenty programu. A to takovým způsobem, že jedna komponenta nemusí mít na druhou referenci v době sestavování programu.

Než se podíváme na konkrétní implementaci jmenovaného frameworku v rámci tohoto projektu, pojďme si výše uvedené ukázat na příkladu. Pro názornou ukázku budeme mít člověka, který se chce stát fotbalistou. Tuto skutečnost můžeme

reprezentovat třídami člověk a míč. Aby se tedy člověk mohl stát fotbalistou, budeme pro jednoduchost předpokládat, že stačí vlastnit míč. Bez využití injekce závislostí se dostáváme do následující situace, viz obrázek 26 s pseudo-kódem. Zde je na první pohled patrné, že třída člověk je pevně svázaná s nějakým konkrétním míčem a je zodpovědná za celou fázi vzniku objektu míč a jeho následné zničení při ukončení běhu programu. Jinak řečeno pokud vytvoříme novou třídu člověk, vždy bude mít jeden a ten samý míč pro názornost dodejme ještě značku xy. Více [18]

```
class MicXY
{
 Public void nafoukniMic() {
 ...
}
 Public void vyfoukniMic() {
 ...
}
}

Class Clovek
{
 Private MicXY mic;
...

 Public void nafoukniMic() {
      Mic.nafoukniMic();
}

...

}
```

Obrázek 26 – Závislost mezi třídami bez injekce závislostí

Nyní si představme, že chceme vytvořit člověka, který bude mít míč značky AB. V každém případě je tedy nutné vytvořit novou třídu míče značky AB, avšak jeho následnou změnou přímo v definici třídy člověk provedeme tuto změnu pro všechny instance. Jedním z možných řešení je převedení závislosti člověka a míče do jiné třídy. Takové třídě se říká továrna. Pro názornost ji představuje pseudo-kód na obrázku 27.

```
class Tovarna
{
 Public static Clovek vytvorClovekaMicAB() {
  return new Clovek(new MicAB());
}
...
}
```

Obrázek 27 – třída továrny oddělující závislost člověka a míče

Ještě než se pustíme do hlubšího prozkoumávání frameworku Google Guice, vysvětlíme si, co je to v programovacím jazyce Java interface třídy a jeho implementace. Rozhraní neboli interface třídy nám říká, jaké jsou dostupné metody třídy (funkce) navenek, tedy z pohledu uživatele dané třídy. Následně implementace rozhraní třídy veškeré tyto metody popisuje příkazy programovacího jazyka. Nejlepšího pochopení řečeného dosáhneme opět ilustrací na pseudo-kódu obrázku 28.

```
Public interface Clovek
{
 Public void vypisJmeno();
 Public void vypisPrijmeny();
...
}

Public class ClovekImpl implements Clovek
{
 Public void vypisJmeno() {
 System.out.println("Lukas");
}
Public void vypisPrijmeni() {
 System.out.println("Sieber");
}
}
```

Obrázek 28 – třída a rozhraní

Síla tohoto řešení tkví v tom, že můžeme navenek vystavovat díky interface pouze metody, které chceme právě využívat a ostatní skrýt. Mimo to může být implementující třída rozšířena mnoha interface a tím pádem máme jasně definováno, jaké funkce musíme uvnitř dané implementace vytvořit. Nyní se již společně podíváme,

jakým způsobem zabezpečuje injekci dependencí framework Google Guice a jaké výhody nám tímto poskytuje. Navíc se zmíníme o návrhové technice třídy zvané „Singleton“ neboli v překladu jedináček.

V první řadě, abychom mohli začít využívat naplno výhody několikrát zmiňovaného frameworku pro injekci závislostí, musíme vytvořit tzv. modul (třídu) definující závislosti mezi implementací a interface injektovaných tříd. Takový počin injektoru (Nástroji starajícímu se o vytvoření instancí daných tříd.) řekne, jaké třídy vytvářet může a jak se chovat k jejich vytvořeným instancím.

```
public class GuicePageObjectsModule extends AbstractModule {
    @Override
    public void configure() {
      bindListener(Matchers.any(), new LogTypeListener());
      bindVUT();
    }

    private void bindVUT() {
      bind(VUTLoginPage.class).to(VUTLoginPageImp.class)
      .in(Singleton.class);
      bind(VUTISPage.class).to(VUTISPageImp.class)
      .in(Singleton.class);
    }
}
```

Obrázek 29 – Třída modulu frameworku Google Guice

Tato ilustrace kódu třídy nutné pro inicializaci frameworku Google Guice za účelem schopnosti injektovat třídy uvedené v metodě „bindVUT()“, skrývá jednu důležitou věc. Připomeňme si, že page objekty jsou bez stavové a právě z toho důvodu je kromě propojení interface a implementace dodáno klíčové slovo „singleton“. Singleton je objekt, který v rámci celého injektoru existuje pouze jednou. Pokud tedy při první injekci takový objekt vznikne, každá další žádost o injekci tohoto objektu obdrží ten stejný vzniklý objekt. Nespornou výhodou tohoto přístupu je fakt, že třídy jsou po startu zavedeny do paměti a následně již nevznikají další. Šetříme tedy paměť a nedochází k častému „garbage collectingu“. Je však třeba si uvědomit, že zdroje počítače nejsou neomezené. A z toho důvodu pro programový popis stránek čítající více než tisíce tříd, je třeba volit přístup jiný s aktivním využitím „garbage collectingu“ u existujících tříd, na které není v současné chvíli běhu programu reference (nejsou nikde využívány). Ve chvíli nové potřeby je třída opět zavedena do paměti. Takový přístup dokáže značně šetřit paměťové nároky, avšak za cenu náročnosti na procesorový čas.

Díky kompatibilitě testovacího frameworku TestNG a frameworku pro injekci závislostí, můžeme elegantně s využitím anotací skrýt inicializaci injektoru. Předáním odkazu na třídu modulu injektovacího poskytovatele, využitím implementované anotace „@Guice“, lze dosáhnout automatického vytvoření injektoru viz obrázek 25. Vzhledem k faktu, že na obrázku 25 je zachycena rozšiřující třída pro všechny naše testovací třídy, vznikají z ní využitím dědičnosti, jsou definované třídy dostupné k injekci v jakémkoli našem testu. Obrázek 24 názorně vyobrazuje využití injekce, kdy je požadována injekce přihlašovací stránky VUT. Po takovém požadavku dojde injektorem k vytvoření instance třídy a zapsání do anotované proměnné, zde se jedná konkrétně o anotaci „@Inject“.

7.5 Hibernate

Nízko úrovňová práce s databází je pro programátora využívajícího techniky objektově orientovaného programování značně omezující. V ideálním případě chce takový člověk pracovat jen s objekty a vyhnout se psaní všemožných selectů, insertů, či updatů. Velmi elegantním řešením by byla možnost ukládat do databáze celé objekty a následně celé objekty z databáze načítat. A právě tuto možnost nabízí framework Hibernate. Jednoduše řečeno celou třídu s pomocí speciálních anotací označíme a následně jsme schopní s pomocí tohoto frameworku třídu uložit (persistovat) do databáze a opět zpětně načíst.

Aby něco takového, o čem hovoří předchozí odstavec, bylo vůbec možné, využívá představovaný framework techniku zvanou ORM neboli objektově relační mapování. Jedná se o implementaci JPA (Java persistence API), jinak řečeno schopnosti objektu uchovat si svůj stav mezi dvěma spuštěními JVM (Java virtual machine). Uvnitř Hibernatu jsou uloženy tzv. mapovací soubory, které říkají jakým způsobem převést datový typ do relační databáze a opačnou cestou. Případně můžeme mapovací soubor stanovit přímo pomocí anotování částí mapovaného objektu. K následnému získání objektů z databáze se využívá tzv. HQL (Hibernate query language), který je velice podobný jazyku SQL pro dotazování do relační databáze.

Obrázek 30 pro nás představuje ilustraci, jak může vypadat anotovaná třída. Úplně nahoře vidíme označení třídy anotací „@Entity“, která nám říká, že se bude jednat o persistovaný objekt a následně zde nalezneme ještě anotaci „@Id“, která označuje automaticky generované id a zároveň primární klíč záznamů vkládaných do relační databáze. Hibernate poskytuje velké množství další anotací, které umožňují například upravovat vztahy mezi objekty, jakou jsou databázové relační vztahy one-to-many, many-to-one, many-to-many apod. Pro úplný soupis, či podrobné informace se lze podívat přímo na stránky Hibernatu. Více [19]

```java
@Entity
public class Card {

    private int id;
    private String cardNumber;

    public String getCardNumber() {
      return cardNumber;
    }

    public void setCardNumber(String cardNumber) {
      this.cardNumber = cardNumber;
    }

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    public int getId() {
      return id;
    }

    public void setId(int id) {
      this.id = id;
    }

}
```

Obrázek 30 – Anotace vytvářející objektově relační mapování pro Hibernate

Pro správnou funkčnost Hibernatu musíme dodat konfigurační soubor, kterým nese informace o tom, kde hledat mapované třídy, přístupové hesla do databází a případně mnohá další nastavení jako například: connection pooling, povolení provádět v databázi mazání, nastavení logování, atd. Jsou dva možné způsoby, jak předat vstupní konfiguraci. Prvním a velice oblíbenou možností je vytvoření XML konfiguračního souboru a následně druhou možností je programová konfigurace využitím Java kódu. Vzhledem k faktu, že konfigurace s využitím XML souboru je široce využívaná, budeme se ji držet i my. A právě při vyplňování konfiguračního souboru pocítíme nespornou výhodu tohoto frameworku. Stačí nám pouze vyplnit přihlašovací jméno a heslo, typ databáze, mapované třídy a to je vše pro základní nastavení. Ale hlavně nás vůbec nezajímá, zda je to databáze Oracle, MySql, či jakákoliv jiná podporovaná, tohle

vše za nás již zařídí popisovaný framework. Na konfigurační soubor se podíváme na dalším obrázku.

```
<hibernate-configuration>
    <session-factory>
      <property name="connection.driver_class">
      oracle.jdbc.driver.OracleDriver
      </property>
      <property name="connection.url">
      jdbc:oracle:thin:@server.databaze.cz:1521/vut </property>
        <property name="connection.username">abc</property>
        <property name="connection.password">abc</property>
        <property name="hibernate.connection.useUnicode"> true
      </property>
       <property name="connection.characterEncoding"> UTF-8
      </property>
      <property name="hibernate.connection.charSet"> UTF-8
      </property>
      <property name="connection.defaultNChar">true</property>
        <!-- SQL dialect -->
      <property name="dialect">
      org.hibernate.dialect.Oracle10gDialect
      </property>
       <property name="default_schema">vut</property>
       <!-- Disable the second-level cache -->
       <property name="cache.provider_class">
      org.hibernate.cache.internal.NoCacheProvider
      </property>
       <!-- Echo all executed SQL to stdout -->
       <property name="show_sql">false</property>
        <!-- Only update the database schema on startup -->
        <property name="hbm2ddl.auto">update</property>
        <!-- Names the annotated entity class -->
        <mapping class="cz.vut.vuttest.domain.Card"/>
    </session-factory>
</hibernate-configuration>
```

Obrázek 31 – Konfigurační soubor pro Hibernate

Nyní si alespoň v bodech řekneme, co zde nastavujeme. Samozřejmě zcela kompletní informace nalezneme na internetových stránkách věnovaných projektu, a proto nebudeme čtenáře zatěžovat podrobným popisem všech uvedených nastavení, ale pouze těch podstatných.

V tomto konfiguračním souboru vytváříme předpis pro vznik tzv. „session-factory“, což není nic jiného než objekt komunikující s databází. V případě relačních databází si můžeme session představit jako samostatný blok příkazů, kdy na konci naší práce, která se průběžně ukládá, můžeme provést tzv. „commit“ a veškeré změny potvrdit, nebo provedeme tzn, „rollback“ a veškeré změny v rámci dané session vrátíme na původní hodnotu. V rámci konfigurace takového objektu musíme postupně poskytnout různé informace. Nejdříve říkáme, jaký driver v závislosti na použité databází požadujeme načíst, v našem ilustračním případě se jedná o databázi Oracle. Dále předáváme tzv. připojovací řetězec, jenž se skládá z názvu využitého klienta pro připojení, adresy serveru včetně portu a konečně samotného názvu instance běžící databáze. Pochopitelnou samozřejmostí je uvedení přihlašovacího jména i hesla pro úspěšné přihlášení. Nastavením používaného kódování znaků se zabývá několik následujících řádků. Také zvolit tzv. „SQL dialect“ jinak řečeno jakým způsobem bude prováděno dotazování pomocí SQL jazyku, který se může skrze různé verzi databáze mírně lišit. Ještě se zaměřme na poslední řádky, kde je možné zapnout výpis veškerých SQL příkazů, které Hibernate provádí a také způsob, jakým se chová k již vytvořeným tabulkám. Zde je zvoleno „update“, což zakazuje jakkoliv tabulky mazat a umožní pouze provádět mimo selektování, vložení, či update. Je důležité na toto nastavení nezapomenout, neboť standartní chování je „drop and recreate“ jinak řečeno smazání a nové vytvoření tabulek v databázi při provedení změny v mapovaném objektu. A konečně zcela na posledním řádku je uvedena cesta v projektové struktuře k mapované třídě.

Na dalším z mnohých obrázků reprezentujících kód se zaměříme na praktické využití v kódu zakončeného načtením objektu z relační databáze. První fázi bude popis, jakým způsobem vytvořit výše zmiňované „session-factory“. Tento popis provedeme do konfigurační třídy frameworku Google Guice a následně budeme moci injektovat objekt na potřebná místa v projektu. Druhou fází bude již samotné načtení objektu z databáze s využitím HQL (Hibernate query language).

Reprezentaci první fáze lze prohlédnout z obrázku 32. Princip funkce celé konstrukce je následující, pokud požádáme framework o provedení injekce, dojde k vyhledání názvu anotované třídy, což je „SessionFactory“ a ihned poté k vyhledání funkce „getSessionFactory()“. Je nám vrácen vytvořený objekt a zároveň je uložen pro pozdější využití, neboť požadujeme u tohoto objektu chování jedináčka. Zvolené řešení zajistí, že o správu veškerých „session“ se bude starat jedna jediná „session factory“. A my tím zvýšíme pravděpodobnost korektního ukončení všech vytvořených spojení k databázi.

```
@Provides
@Singleton
SessionFactory getSessionFactory() {
log.trace("Creating hibernate session factory.");
Configuration hibConfiguration = new Configuration().configure();
StandardServiceRegistryBuilder serviceRegistry = new
StandardServiceRegistryBuilder()
.applySettings(hibConfiguration.getProperties());

return
hibConfiguration.buildSessionFactory(serviceRegistry.build());
}
```

Obrázek 32 – Návod na vytvoření session-factory pro Google Guice

Přejděme nyní k druhé zmiňované fázi a ukažme si, jak vypadá praktické použití s vytvořením „session“ a vytažením objektu s databáze na obrázku níže. Injektujeme „session factory“ ze kterého získáme „session“. V rámci této transakce požadujeme získání karty s určitým id z databáze. Po ukončení operace získání provedeme ukončení započaté „session“. Z pohledu programátora je nutné následující kód opatřit ještě blokem „try a catch“ pro ošetření možných výjimek. Jinak řečeno v případě výjimky by nemuselo dojít k ukončení „session“ a zamčení tabulky v databázi.

```
Public Card getCard(int id) {
      @Inject
      SessionFactory sessionFactory;

      session = sessionFactory.getCurrentSession();
      Query query = session.createQuery(
      "from Card where id = :id");
      query.setParameter("id", id);

      Card card = query.list().get(0);

      session.getTransaction().commit();
      session.close();
      Return card;
}
```

Obrázek 33 – Získání objektu card z databáze pomoci frameworku Hibernate

7.6 AspectJ

S rostoucím rozmachem informačních technologií se vyvíjejí i programovací jazyky. Snahou tohoto rozvoje je usnadnit programátorovi převod tzv. oblastí zájmů neboli koncernu do kódu srozumitelného pro počítač. Nejdříve vznikají tzv. procedurální programovací jazyky, které se snaží řešit problém postupnou přesně danou posloupností kroků. Další zajímavou oblastí jsou jazyky založené na objektovém přístupu k programování, jinak řečeno program se skládá z různých oddělených objektů. Tyto objekty jsou z venkovního pohledu černými skřínkami a my pouze víme, co s nimi můžeme provádět, avšak do vnitřní logiky nevidíme. Obě tyto oblasti se snaží zvoleným způsobem přístupu minimalizovat vzájemné vazby koncernů. Konkrétněji můžeme říci, že se snaží vytvořit systém umožňující dokonalé oddělení koncernů.

Avšak představme si velmi důležitou oblast, kterou je logování v programu. A ihned zjistíme, že koncern logování se prolíná přes mnoho dalších oblastí. Opět uvedeme na příkladu, logujeme operace v „page objektech“, v doplňkových frameworcích, přímo ve třídách testů, či jednoduše kdekoliv to uznáme za vhodné. Pokud dojdeme tohoto uvědomění, je zřejmé, že musíme volat nějakou logující funkci na všech místech programu a není možné ji oddělit na jedno místo.

Právě zde přichází ke slovu aspektově orientované programování. Tato technika návrhu se snaží za pomocí třídy aspektu zapouzdřit koncerny, které procházejí skrze mnoho oblastí. Je toho dosaženo schopností pozměnit chování neaspektových částí kódu. Přesněji řečeno za pomocí přesně definovaných vstupních bodů tzv. „pointcut“ bodů jsme schopni vstoupit do vykonávané části neaspektového kódu a za pomocí tzv. rad „advice“ pozměnit jeho chování, případně vykonat jakoukoliv činnost navíc. K nejvíce využívaným implementacím tohoto řešení patří projekt AspectJ.

```
@Pointcut("execution(@org.testng.annotations.Test * *(..))  &&
@annotation(anot) && if()")
public static boolean pointcutAnnotTest(Test anot) {
      return true;
}
```

Obrázek 34 – Ukázka vstupního bodu tzv. pointcutu

Obrázkem 34 si názorně ukazujeme, jak takový vstupní bod může vypadat. Nejdříve se v našem vysvětlování zaměříme na anotaci nad názvem metody. Anotace nám říká: „Tento vstupní bod je platný pokud došlo ke spuštění jakékoliv metody, s jakýmkoliv návratovým typem a přebírající jakékoliv parametry, která má nad sebou anotaci @Test. Tuto anotaci načti do proměnné anot a celý vstupní bod proveď jen pokud, je návratovým typem metody níže hodnota true“.

Pokud je podmínka vstupního bodu splněna je vyvolána tzv. „advice“, což je kus kódu obalující vykonání odchycené metody, jež splňuje podmínky vstupního bodu. Následně jsme v rámci tohoto mechanismu schopni vyvolat příkazy před samotným vykonáním odchycené metody, nechat metodu proběhnout a následně také provést příkazy po vykonání metody. Právě toto chování vytváří takovou sílu tohoto nástroje, kdy můžeme dokonce upravit předané parametry, či provést změny v návratovém typu metody. Více nám napoví obrázek níže.

```
@Around("pointcutLoggedAnnotMethod(anot) && if(true)")
      public Object logLoggedAnnotMethod(final ProceedingJoinPoint
thisJoinPoint, Logged anot) throws Throwable {

// Gets hooked function name and class name.
final String className =
thisJoinPoint.getSignature().getDeclaringType().getName();
final String methodName = thisJoinPoint.getSignature().getName();
final String qualifiedMethodName = String.format("%s.%s", className,
methodName);

// Returned (value) object after function proceedings
Object returnedValue = null;

// Gets method signature to capture parametrs
MethodSignature methodSignature = (MethodSignature)
thisJoinPoint.getSignature();
String[] parametrNames = methodSignature.getParameterNames();
Logged loggedAnnotation = (Logged)
methodSignature.getMethod().getAnnotation(Logged.class);

// Proceeds hooked function
returnedValue = thisJoinPoint.proceed();

}
```

Obrázek 35 – Around advice v rámci AspectJ

Když se nyní blíže podíváme na vnitřní obsah advice metody nástroje AspectJ, můžeme si postupně všimnout získání jména odchycené metody pomocí vstupního bodu a následně získání parametrů, nebo návratového typu. Zavoláním metody „proceed()“ na posledním řádku předáme řízení odchycené funkci a necháme ji vykonat její kód.

V současné chvíli je již zcela jasné, jakou výhodu může použití frameworku AspectJ přinést pro oblast logování, která bývá často rozprostřená po celém programovém kódu. Když formulaci výhody hodně zobecníme, jedná se o snadnou možnost, jak selektivně odchytit metody a následně logovat jejich činnost včetně vstupních a výstupních parametrů. A to například na základě pevně stanovené jmenné konvence, rozdělení do separátního balíčku a dalších možností.

7.7 Relační databáze MySQL

Velice populární relační databáze, která byla původně vytvořena švédskou firmou MySQL AB, později vlastněna společností Sun Microsystems a tedy potažmo společností Oracle. Mezi největší přednosti této relační databáze patří dualita způsobu jejího licencování. Přesněji řečeno je nabízena, jak verze šířena pod GPL licencí, tak komerční verze databáze s plnou podporou. Jedná se o multiplatformní databázi vyvíjenou s pomocí jazyka C/C++.

Důvodem zvolení této relační databáze je v první řadě její dostupnost zdarma, široká komunita uživatelů, její zaměření na rychlost provádění dotazů. Právě poslední jmenovaný důvod směřující k volbě přinesl do MySQL absenci některých často využívaných nástrojů využívaných hlavně při vývoji webových aplikací. Jedná se hlavně o triggery, pohledy a rozšířené možnosti zálohování. Avšak pod tlakem požadavků uživatelů se začínají přidávat.

```
CREATE TABLE `reportiteration` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `createdBy` varchar(255) DEFAULT NULL,
  `creationDate` datetime DEFAULT NULL,
  `iterationName` varchar(255) DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `REPORTITERATION_DATE` (`creationDate`),
  KEY `REPORTITERATION_NAME` (`iterationName`),
  KEY `REPORTITERATION_CREATEDBY` (`createdBy`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
SELECT * FROM vuttest.report;
```

Obrázek 36 – vytvoření tabulky v MySQL databázi

8 TVORBA FRAMEWORKU – JÁDRO

Předchozí kapitola nás mohla zahltit informacemi o všech použitých nástrojích, které vstupují do tvorby vytvářeného frameworku pro automatizaci webových stránek. V rámci této kapitoly si uděláme jednoduché shrnutí a zaměříme se na tvorbu jádra, které bude obsahovat hlavní logiku celého řešení. Dosáhneme tak schopnosti dodávat vytvořené řešení jako přeloženou knihovnu, které bezpečně uchová naše „know how“ a také odstíníme uživatele od složitých programových konstrukcí.

Využívané nástroje v projektu jádra:

- Maven – pro správu celého projektu a jeho rozdělení
- Eclipse IDE – integrované vývojové prostředí pro jednodušší tvorbu kódu a integrovanou podporu pro používané nástroje
- Google Guice – pro schopnost využít injekci závislostí, či změnu chování stránek po změně napojení interface a implementace třídy.
- Hibernate – pro uložení objektů do databáze a jejich zpětné načtení rovnou jako instance daného objektu (třídy)
- AspectJ – pro zjednodušení komplikací s logováním

8.1 Stanovení jmenné konvence

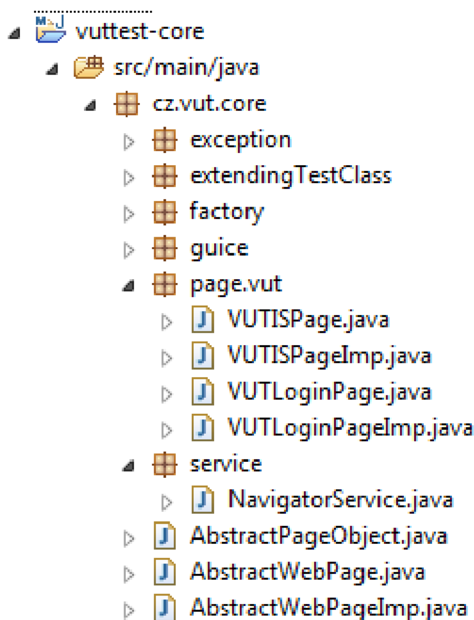
Abychom se vyhnuly budoucím komplikacím při implementaci logování s využitím nástroje AspectJ, je nutné vytvořit jmennou konvenci u požadovaných tříd, kde budeme logování provádět. Ideálním případem by bylo stanovení celkové jmenné konvence.

Stanovení pravidel:

- Pravidla pro PO:
  - Vytvářené PO vložíme do balíčku Page a následně podbalíčku určující přesnější oblast
  - Vytvářené PO, které jsou abstraktními třídami, začínáme slovem „Abstract“
  - Vytvářené PO vždy doplníme slovem „Page“
  - Implementaci třídy PO odlišíme od jeho interface příponou „Impl“
  - Všude, kde je to vhodné se používá komentář vysvětlující funkčnost dané metody a to formou javadoc komentáře

- Pravidla pro pomocné „service“ třídy:
  - Pokud je to možné, třída se vytváří jako statická včetně jejich metod
  - Název třídy je ukončen slovem „service“
  - Pokud existuje implementace a interface, řídí se stejnými pravidly jako u PO
  - Všude, kde je to vhodné se používá komentář vysvětlující funkčnost dané metody a to formou javadoc komentáře

Smysl některých pravidel se může zdá zpočátku svazující, avšak s narůstající velikostí projektu pochopíme jeho hlavní přínos. Díky oddělení PO do vlastního balíčku a jeho dalším členěním do podkategorií, jsme následně schopni selektivně logovat jen některé PO. Například budeme chtít v jedné kategorii PO provádět přesnější logování a tento koncept nám to umožní. Také přítomnost „javadoc“ komentářů se později ukáže jako přínosný krok, neboť jsme následně schopni jediným kliknutím vygenerovat HTML dokumentaci celého našeho projektu a mimo to, pro člověka píšící scénář testu, jsou okomentované funkce k nezaplacení. Rozdělení třídy na jejich interface a implementaci vnese přehledné oddělení kompletní implementace metod, od jejich definice a okomentování.



Obrázek 37 – letmý pohled na dělení projektu a jmennou konveci

Obrázek 37 přináší bližší pohled na vnitřní členění projektu jádra vytvářeného frameworku. Za povšimnutí zde hlavně stojí aplikace definované jmenné konvence v balíčcích service a page.vut. Také jsou zde dle nastavených pravidel pojmenovány třídy abstraktní, o kterých se více dozvíme v rámci další kapitoly.

8.2 Dědičnost

Bylo by čistou pošetilosti využívat objektově orientovaný jazyk a přitom nevyužít jednu z nejsilnějších výhod, kterou přináší a tou je právě v názvu kapitoly zmiňovaná dědičnost. Velmi často se stává, že nějaký objekt, či prvek se na stránce vyskytuje vícekrát anebo dokonce samotná webová stránka do sebe samotné načítá další webovou stránku. Právě tyto případy jsou jako dělané pro využití dědičnosti. Dobře navržená dědičnost dokáže v projektu usnadnit mnoho práce. Avšak jmenovaná výhoda může platit i opačně, při špatném navržení.

Obrázek 38 – výřez navigačního sloupce IS VUT

Výše zobrazený obrázek nám bude dobrou ilustrací pro demonstraci výhod dědičnosti. Představme si, že budeme chtít implementovat více stránek z tohoto navigačního sloupce. Budeme chtít popsat například odkaz Osobní informace, Studium, Aktuality a další. Každá z těchto stránek, ale přece obsahuje i tento navigační sloupec. Je tedy absurdní popisovat jeho funkčnost na každém takto vytvořeném PO popisujícím onu stránku znovu. A právě zde přichází ke slovu dědičnost, vytvoříme nadřazenou abstraktní třídu, která v sobě tuto funkcionalitu uchová a poskytne ji všem PO třídám, jež budou jejími potomky. Není třeba říkat, že tímto přístupem si ušetříme mnoho práce například při změně jednoho z odkazu, nám stačí provést změnu na jednom místě a je promítnuta do každého objektu, který tento navigační sloupec používá.

8.3 Pomocné servisní třídy

V rámci automatizace často prováděných operací si vytvoříme skupinu tzv. servisních tříd. Jedná se o skupiny statických tříd, které poskytují metody pro usnadnění činnosti tvorby testů. Může se jednat například o navigování na prvotní stránky, generování náhodných čísel, čtení a zápis do souboru Excelu, jednoduché operace nad databází a mnoho dalších často využívaných věcí. Na následujícím ukázkovém kódu si demonstrujeme, jak taková servisní třída vypadá.

```
public class NavigatorService {

    public static void navigateToVUT() {
      DriverFactory.getDriver().get("http://www.vutbr.cz");
    }

}
```

Obrázek 39 – statická pomocná třída pro navigaci na první stránku

Výhodou takto vytvořené servisní třídy je možnost volat její metody přímo skrze tečkovou konvenci. Jinak řečeno není nutné vytvářet instanci této třídy a teprve posléze volat její metody. S rostoucí komplexností prováděných testů se ukazuje centralizace často používaných funkcí, jako velice přínosná. Jakmile totiž použijeme servisní třídy v testech, například pro generování náhodných čísel, čísel účtu, nebo rodných čísel, získáme testy s rozdílnými daty. Takto napsané testy jsou snadno modifikovatelné z jednoho místa a jsou schopny odhalit větší množství komplikací, které by při pevně daných statických datech nebyly objeveny.

8.4 Shrnutí obsahu jádra

Tato kapitola nás provedla obsahem a hlavními částmi projektu jádra našeho frameworku. Řekli jsme si, že hlavním důvodem rozdělení celého projektu na část jádra a testů je v první řadě oddělení našeho „know how“, zjednodušení udržitelnosti projektu, odstíněný tvůrců testů od komplexního kódu. Díky využitému nástroji Apache Maven jsme schopni projekt jádra distribuovat skrze vložení závislosti do projektu testů. Jinými slovy řečeno vývojář testů bude mít uvnitř svého integrovaného vývojového studia importován pouze projekt testů a pouze do něj bude vidět. Metody PO implementované v jádře, pro něj budou dostupné přes tečkovou konvenci neboli tzv. „fluent interface“. Také jsme vysvětlili důležitost alespoň základní tvorby jmenné

konvence, které v konkrétním případě poslouží propojení mezi logovanými funkcemi a frameworkem AspectJ. Mimo to jsou veškeré deklarace a definice uvnitř PO odděleny do samostatných tříd a to přesně do interface a implementace dané třídy popisující funkčnost webové stránky. Demonstrací na navigačním panelu IS VUT byla prezentována užitečná vlastnost objektově orientovaného programování a byla ji dědičnost, která je při dobrém navržení schopna ušetřit mnoho práce. Myšlenka servisních třídy byla popsána poslední kapitolou před shrnutím a pojednává o konceptu centralizace často využívaných funkcí v testech, jako je například generování rodného čísla.

9 TVORBA FRAMEWORKU – TESTY

Kapitola je věnována přehledu vlastností projektu testů, kam se v závislosti na připojeném projektu jádra vytvářejí vlastní spustitelné testy frameworku TestNG. Abychom maximalizovali přehlednost vytvářených testových scénářů, je opět potřeba definovat několik základních pravidel. Další část kapitoly nás zavede k pokročilejším metodám využití testovacího frameworku a to především tvorby závislosti testů, paralelního spouštění, či k tvorbě tzv. „suite“.

9.1 Stanovení pravidel pro tvorbu testů

Soubor pravidel:

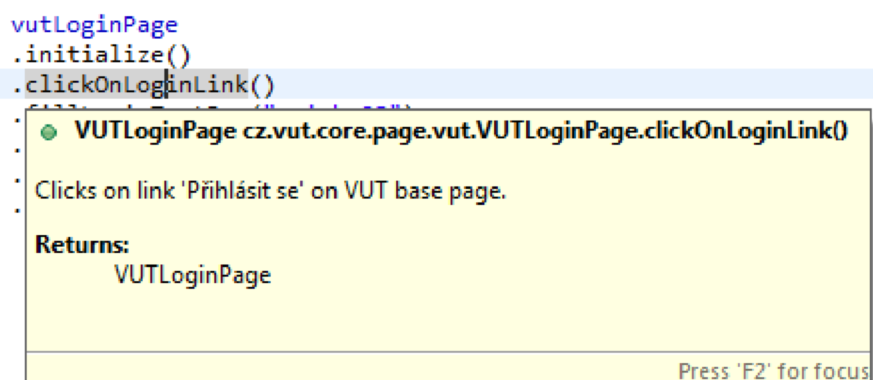
- Používáme smysluplný název testu
- Testy třídíme do balíčků podle oblasti testu
- Název testované metody uvnitř třídy testu uvozujeme slovem „test“ a pokračujeme názvem testovací třídy
- Každý krok testu v rámci „fluent interface“ zapisujeme na nový řádek

```
public class VUTLogin extends SeleniumTestCase {

    @Inject
    VUTLoginPage vutLoginPage;

    @Test
    public void testVUTLogin() throws IOException {

      NavigatorService.navigateToVUT();

      vutLoginPage
      .initialize()
      .clickOnLoginLink()
      .fillLoginTextBox("xsiebe03")
      .fillPassTextBox("abcde")
      .submit();
    }
```

Obrázek 40 – Samostatná třída testu s aplikovanými pravidly

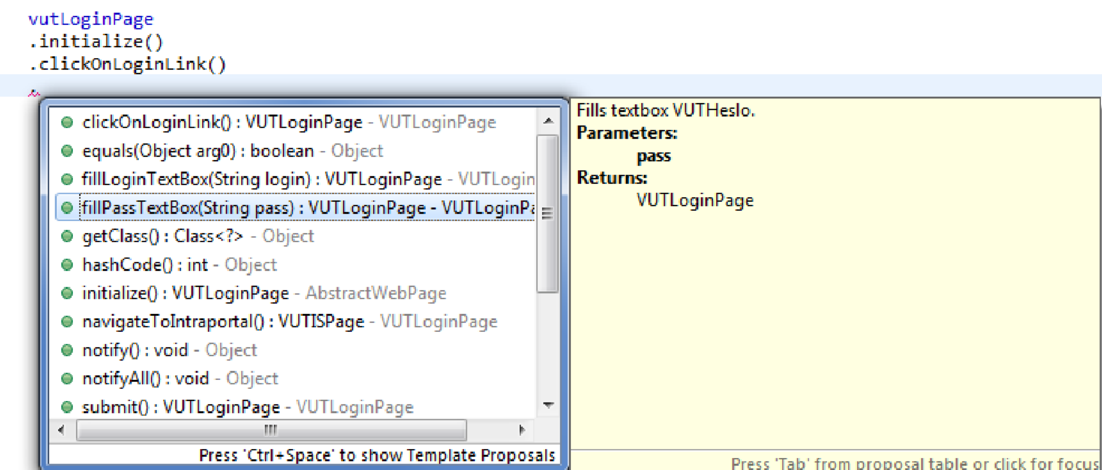
Předchozí kód třídy zachycený obrázkem 39 reprezentuje použitá pravidla. Nejdříve si všimněme názvu, který je „VUTLogin“. Pokud se podíváme na jednotlivé kroky testu, které jsou často sebevysvětlující se svým názvem, pochopíme, že se jedná o test přihlášení do IS VUT. Správně je také uveden název testovací anotované třídy, který je uvozen slovem „test“ a pokračuje názvem testovací třídy. Posledním důležitým pravidlem je odsazení každého kroku testu (metody PO) na nový řádek. Je to prováděno z důvodu dosažení větší čitelnosti a přehlednosti testu.

9.2 Javadoc komentáře a Fluent interface



Obrázek 41 – ukázka javadoc komentáře zobrazeného IDE studiem Eclipse

Pro úplnost informací nám obrázek 41 přináší ilustraci zobrazeného „javadoc“ komentáře, který je uveden v rámci interface PO. Pokud si tedy tvůrce testu není jistý, co dané funkce provádí, stačí nad ni najet myší a je zobrazena nápověda. Následující obrázek prezentuje sílu „fluent interface“ při tvorbě testu.

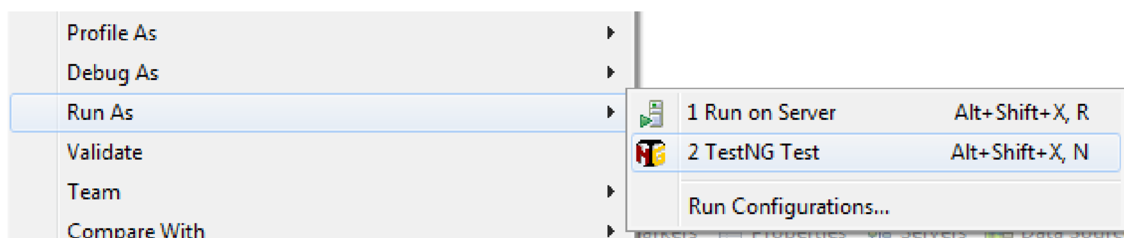


Obrázek 42 – ilustrace fluent interface

Až spojením komentáře a automatického nabízení dostupných metod „page objectu“ vývojovým prostředím vznikne opravdové symbióza umožňující tvůrci testu o něco bezstarostnější život. Díky tomu, že PO navrací sám sebe, pokud se zůstává na stejné webové stránce, anebo následující PO, pokud se přechází na další stránku, je vývojové studio schopné nabídnout soupis dostupných metod. Tento koncept částečně brání vytváření nefunkčních testů, neboť nabízí pouze dostupné metody daného PO.

9.3 Spuštění testu

Exekuci testů lze pohodlně provádět přímo z vývojového prostředí Eclipse. Jedinou nutnou podmínku je nainstalování doplňku pro framework TestNG. Následně klikneme pravým tlačítkem myši na třídu testu v projekt exploreru. Dále stačí zvolit z kontextové nabídky možnost „Run as => TestNG test“. Vývojové prostředí inicializuje framework a předá mu zvolenou třídu testu více obrázek 43.



Obrázek 43 – Spuštění testu skrze plugin pro IDE Eclipse

Postup popsaný výše odpovídá spuštění jedné samostatné třídy testu. Avšak framework TestNG umožňuje vytvářet tzv. suity testů, kde je možné soustředit více testovacích tříd zároveň a tím předat požadavek na jejich postupnou, či paralelní exekuci. V první řadě si ukážeme, jak taková suita vypadá. Jedná se o XML soubor čitelný pro testovací framework.

```
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd" >
<suite name="Suite" verbose="1">
      <test name="VUTLogin">
            <classes>
                  <class name="cz.vut.test.VUTLogin" />
            </classes>
      </test>
</suite>
```

Obrázek 44 – Ukázka suity pro TestNG

Dokonce pokud se rozhodneme pro spuštění většího množství testů a definujeme je do suity, jsou postupně jeden po druhém, v pořadí přidání od shora dolů, pokud není řečeno jinak, spouštěny. Ale až schopnost spouštět testy paralelně přidáním jediného parametru do XML souboru suity, ukáže skutečnou sílu a výhodu takového přístupu. Předpokládejme, že máme určité oblasti, které chceme otestovat. Pro jednoduchost si vytvořme imaginární oblasti přihlašování, operace v informačním systému VUT a další. A právě díky suitám jsme schopni testy přehledně oddělit a připravit oblasti s jasně definovaným plánem spouštění. Obrázek níže přiblíží popsané.

```
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd" >
<suite name="Suite" verbose="1" parallel="tests" thread-count="2">
      <test name="VUTLogin">
            <classes>
                  <class name="cz.vut.test.VUTLogin" />
            </classes>
      </test>
      <test name="VUTLogin">
            <classes>
                  <class name="cz.vut.test.VUTLogin" />
            </classes>
      </test>
</suite>
```

Obrázek 45 – Nastavení paralelního spuštění v rámci suity

Suitu spouštíme téměř identickým způsobem, jako je spuštění samostatné třídy testu. Jediným rozdílem je zvolení „Runs as => TestNG suite“ z kontextové nabídky. Následně dojde k načtení informací ze suity a k paralelnímu spuštění zvolených testů.

9.4 Vytvoření závislosti mezi testy

V některých případech, se můžeme setkat s nutností vytvořit mezi testy závislosti. Přesně řečeno, jeden test se nespustí bez úspěšného proběhnutí testu, na kterém je závislý. Konkrétně si můžeme představit operace v IS systému VUT, které není možné provádět bez předchozího přihlášení. Na dalším z obrázku si ukážeme, jak takovou závislost mezi testy definovat. Obě testovací metody vložíme do jedné třídy testu a jako parametr anotace „@Test“ předáme klíčové slovo „dependsOnMethods“ a předáme hodnotu rovnou názvu třídy s přihlášení do IS VUT. Samozřejmě je možné vytvořit závislost hned na několika testovacích metodách zároveň.

```
@Test
public void testVUTLogin() {

  NavigatorService.navigateToVUT();

  vutLoginPage
  .initialize()
  .clickOnLoginLink()
  .fillLoginTextBox("xsiebe03")
  .fillPassTextBox("abcd")
  .submit()
  .navigateToIntraportal();
}

@Test(dependsOnMethods = "testVUTLogin")
public void testShowSubjectSchedule() {
  …
}
```

Obrázek 46 – Ukázka vytvoření závislosti mezi testy

Tato vlastnost frameworku je výhodná pro tvorbu testů, které před vlastní spuštěním nejdříve zkontrolují vstupní podmínky a teprve posléze spustí provádění vlastního těla testu. Z procesního pohledu tedy nezačínáme vykonávat test bez splněných vstupních podmínek, což nám ušetří čas a přesně ukáže na místo výskytu chyby. Pojďme se společně ještě podívat na další velice užitečnou vlastnost frameworku TestNG. Jedná se o schopnost vytvářet skupiny testů s následným řízeným spouštěním v rámci suity.

```
@Test
public class Test1 {
  @Test(groups = { "windows.checkintest" })
  public void testWindowsOnly() {
  }
  @Test(groups = {"linux.checkintest"} )
  public void testLinuxOnly() {
  }
}
```

Obrázek 47 – rozdělení testovacích metod do tříd převzato z [17]

Nyní si představme hypotetickou situaci, kdy nasazujeme zcela nový informační systém a potřebujeme otestovat jen zlomek jeho funkčnosti. Tento zlomek odpovídá například úspěšnému přihlášení a odhlášení, neboť nemáme připojeny ostatní systémy zajišťující korektní funkčnost dalších komponent našeho systému. Avšak chceme otestovat alespoň tento základní test v rámci ověření správného nasazení webové aplikace. Máme k dispozici suitu s kompletními testy prováděnými po nasazení nové webové aplikace. A protože jsou naše testy rozděleny do skupin, stačí nám definovat v suitě, které skupiny si přejeme spustit. Další z obrázku přibližuje, jak můžeme taková suita vypadat.

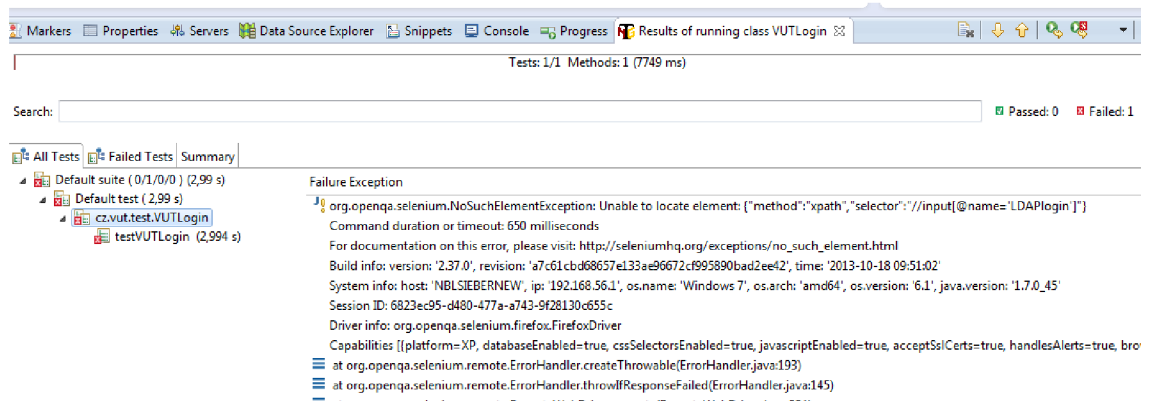
```
<test name="DeployTest">
  <groups>
    <run>
      <include name="login.*"/>
    </run>
  </groups>

  <classes>
    <class name="cz.vut.test.AfterDeployTest"/>
  </classes>
</test>
```

Obrázek 48 – Ukázka omezení spouštěných testovacích metod na základě definované skupiny testů

Frameworku takovou suitou dáváme jasný pokyn k tomu, aby v rámci definované třídy „AfterDeployTest" provedl spuštění všech testovacích metod, které patří do skupiny testující přihlášení do IS. Ostatní metody, které se nalézají v této třídě budou ignorovány.

9.5 Vyhodnocení výsledků spuštěného testu

Veškeré naše snažení o spouštění testů by bylo zbytečné, kdybychom neměli mechanismus jakým vyhodnotit stav proběhlého testu. Takový mechanismus je naštěstí vestavěn již formou zásuvného modulu do integrovaného vývojového prostředí Eclipse. Díky němu získáme přehled o názvech spuštěných testů, nastalých komplikacích, nebo jednoduše konečném výsledku testu. Dokonce jsou odchytávány výjimky a následně přehledně zobrazeny v rámci reportingového okna a zajímavou funkcionalitou je tvorba suity obsahující neúspěšné testy, které posléze můžeme jedním kliknutím spustit znovu.



Obrázek 49 – Ukázka vzniklého reportu po spuštění TestNG testu

Jak je patrné z přebytku červené barvy v levé části obrázku, kde je v postupné hierarchii zobrazena až samotná třída testu, nebo zcela vpravo hodnotou jedna u položky „Failed“, nebyl tento test úspěšný. Ve střední části okna si můžeme všimnout popisu „Failure Exception“, která podává bližší informace o přesných důvodech selhání testu. V našem konkrétním případě to byla neschopnost nalézt požadovaný element na internetové stránce. Velice užitečná tlačítka se nacházejí nad hodnotami položek „Passed“ a „Failed“ a jsou to tlačítka v obou případech obsahující znak „Play“ v zeleném kolečku. Tato dvě tlačítka jsou schopná znovu spustit celý test, nebo jen testy, které nebyly v minulém spuštění úspěšné.

10 TVORBA FRAMEWORKU – REPORTING

Reportingové schopnosti integrovaného vývojového prostředí Eclipse jsou výborným pomocníkem při online vyhodnocování výsledků. Avšak při nutnosti výsledky uchovávat, či poskytnout historický bližší informace o důvodu selhání testu, již nejsou použitelné. Právě z tohoto důvodu se nabízí možnost vytvořit vlastní reportingové rozhraní, které výsledky jednotlivých spuštění uchová a případně poskytne vývojáři automatizace relevantní údaje o chybě. Tyto relevantní informace poslouží k posouzení, jestli se jedná o chybu frameworku, nebo chybu testovaného systému. Je rozumné posbírat veškeré potřebné údaje automaticky a nespoléhat na lidský faktor, neboť člověk spouštějící testy, nemusí mít hluboké znalosti v oblasti programovacích jazyků a potřebné údaje nám nemusí posbírat všechny.

10.1 Tvorba tříd popisujících reporting

V první řadě bude nutné vytvořit soubor tříd popisujících navrhovaný systém reportingu. Tyto třídy následně poslouží pro Hibernate jako mapovací entity pro převod mezi databází a programovacím jazykem Java. Jiným slovy použijeme Hibernate jako vrstvu mezi databází a naším frameworkem a dokonale tak odstíníme navrhovaný framework od provázání s jednou konkrétní databází. Pokud se totiž v budoucnosti rozhodneme framework z nějakého důvodu provozovat nad jiným typem databáze a tento typ bude podporovaný frameworkem zajišťujícím ukládání do databáze, postačí nám jednoduchá změna nastavení v konfiguračním souboru. Právě tato výhoda nám dává neskutečnou volnost při volbě databáze a nesvazuje nás s jednou konkrétní databází, jak tomu často bývá u komerčních řešení.

Nyní se pojďme zamyslet nad hierarchií reportování, velmi pravděpodobně bude potřeba mít možnost zařadit vetší množství spuštění pod jednu seskupující entitu. Tuto entitu nazveme iterací. Ideálním řešením bude možnost nastavit iteraci při každém spuštění testu a to tak, abychom byli schopni zařazovat testy i mezi různými spuštěními. Dalším prvkem našeho reportování bude entita zvaná report. Touto entitou rozumějme spuštění samostatní metody / třídy testu. Obrazněji řečeno report bude ohraničen vstupem do metody anotované klíčovým slovem „@Test“ a jejím opuštěním. Posledním a nejmenším objektem našeho řešení bude jednotlivý krok reportu. Zde půjdeme o něco dále a vytvoříme vlastní anotaci „@Logged“ a veškeré metody, které takto označíme, vyvolají vytvoření právě reportovacího kroku.

Další z důležitých otázek, jejichž zodpovězení usnadní tvorbu reportovacího rozhraní, je vzájemný vztah mezi entitami. Entita iterace bude obsahovat množství entit reportu a ty zase budou obsahovat velké množství entit jednotlivých kroků. Takové rozložení nám později umožní rozličné možnosti filtrování již na úrovni databáze a

jazyka SQL. Mimo to, bude také možné rozložit exekuci testů do několika dnů a všechny posléze sdružit pod jedinou iterací a vyjádřit tak jejich náležitost například k určitě oblasti testování. A takové řešení zajistí vysokou míru přehlednosti i při rostoucím množství historických záznamů.

```
@Entity
public class Report {

    private int id;
    private String reportName;
    private Date creationDate;
    private double runTime;
    private State state;
    private String printscreen;
    private String stackTrace;
    private Set<ReportStep> steps;
    private byte[] binaryPicture;
    private String runByUser;
    private String runOnMachine;
    private String suite;
    private ReportIteration iteration;
    private String iterationName;
    private String actualNote;

  }
```

Obrázek 50 – Třída popisující entitu report

Obrázek výše nám zobrazuje část kódu třídy, které bude popisovat reportingový záznam ohraničený spuštěním metody anotované klíčovým slovem „@Test“ a koncem této metody. Ze zajímavých položek se zde pozastavíme nad proměnnou „steps“, která reprezentuje jednotlivé kroky neboli metody anotované námi vytvořenou anotací „@Logged“, jejichž definici si ukážeme později. Kvůli vzájemnému provázání je zde i třída sedící v reportingové hierarchii nejvýše a to iterace.

Mezi další zajímavé položky patří „reportName“, která ponese jméno reportu tvořené celou cestu k němu z pohledu našeho projektu. Datum vytvoření použitelné k lepšímu filtrování, celkové trvání testu využitelné k vyhledání performance problémů, stav celého běhu reprezentovaný úspěchem, nebo neúspěchem. Dokonce budeme ukládat snímek obrazovky při neúspěchu, nebo celkový „stack trace“ našeho

frameworku a to pro potřeby posouzení původu chyby. Samozřejmostí jsou položky reprezentující stroj, nebo uživatele, na kterém byl test spuštěn.

```
@Entity
public class ReportStep {

      private int id;
      private String stepName;
      private Date creationDate;
      private double runTime;
      private State state;
      private String params;
      private String paramValues;
      private Report report;
      private LoggedType type;
      private String addionalInfo;
      private String returnedValue;
      private Category category;
  }
```

Obrázek 51 – Třída popisující entitu reportovacího kroku

Dalším z vložených obrázků je třída popisující nejmenší reportingovou jednotku, kterou je samostatný krok v kódu označení anotací „@Logged“. Ze zajímavých proměnných zde jmenujme například položku „params“, která obsahuje soupis přebíraných parametrů dané logované metody a následně položka „paramValues“, které obsahuje jejich hodnoty. Také návratová hodnota je ukládána do položky „returnedValue“. Mimo řečené jsou kroky rozdělovány v rámci třídění do kategorií a typů, tento přístup je zvolen v rámci schopnosti omezení hloubky logování. Například se rozhodneme nelogovat servisní třídy apod.

```
@Retention(RetentionPolicy.RUNTIME)
public @interface Logged {
      public LoggedType type() default LoggedType.METHOD;

      public String info() default "";

      }
```

Obrázek 52 – Vytvořená anotace logged pro účely reportingu

Vytvořením anotace „@Logged“ umožníme v průběhu tvoření frameworku jednoznačně označit vznikající metody v rámci PO, nebo servisních tříd a následně je ještě zařadit do skupiny. Vidíme totiž, že definice anotace na obrázku 52, obsahuje proměnnou „type“, která může nabíhat více hodnot z výčtového typu „LoggedType“. Pokud tedy při tvorbě kódu frameworku zařadíme jednotlivé metody ještě i do skupin, budeme posléze schopni řídit hloubku logování a tím pádem vytvářenou zátěž na databázové operace. Mnohdy je také zbytečné logovat veškeré metody, neboť vzniká informační šum a je mnohem obtížnější „vydolovat“ potřebné informace.

```
@Entity
public class ReportIteration {

      private int id;
      private String iterationName;
      private Date creationDate;
      private Set<Report> reports;
      private String createdBy;
}
```

Obrázek 53 – Nejvíce nadřazená entita reportingu je iterace

Poslední z vytvořených tříd je třída reprezentující hierarchicky nejvýše umístěný objekt a to iteraci. Iterace nám umožní sdružovat skupiny testů pod společnou oblast. Například si vytvoříme iteraci přihlášení a budeme pod ni sdružovat různé reporty ohledně testů s přihlášením. Nyní se pojďme podívat, jak bude vypadat anotovaná metoda uvnitř PO, kterou budeme chtít logovat, a tedy bude reprezentovat vytvoření jednoho kroku.

```
@Logged
public VUTLoginPage fillLogin(String login) {
  loginField.clear();
  loginField.sendKeys(login);
  return this;
}
```

Obrázek 54 – ukázka definice metody uvnitř PO s aplikací anotace @Logged

Jak je patrné z obrázku 54, jediným klíčovým slovem v podobě anotace jsme schopni označit logované a nelogované metody. Obrázek 52 nám prozrazuje, že standardním zvoleným typem logovaného objektu je typ „Method“, který je zvolen automaticky. Pokud explicitně neuvedeme, do jakého typu logování označená metoda spadá, přesněji řečeno jestli je to servisní metoda, metoda v PO apod.

10.2 Konfigurace AspectJ frameworku

K uvedení celého navrženého řešení s pomocí frameworku AspectJ, je potřeba provést ještě pár doplňkových nastavení. V první řadě potřebujeme vytvořit třídu obsahující definici logovacího aspektu. Dalším nutným krokem je vytvoření konfiguračního xml souboru, které frameworku prozradí, kde má definovaný logovací aspekt hledat. A samozřejmě poslední částí je programová implementace vstupních bodů, aspektových rad a vlastního kódu reportingu.

10.2.1 Vytvoření vstupních bodů pro logování

Abychom mohli odchytávat anotaci „@Test“ a anotaci „@Logged“, je potřeba vytvořit vstupní body, která nám odchytí jejich spuštění. Lze k tomu s úspěchem využít nabízené možnosti řešení AspectJ, které zahrnuje využití zástupných znaků, regulárních výrazů a mnoho dalších užitečných funkcionalit.

```
/**
* If PointCut for annotation test
*
* @param anot annotation on function
* @return boolean
*/
@Pointcut("execution(@org.testng.annotations.Test * *(..))
&& @annotation(anot) && if()")
public static boolean pointcutLoggedAnnotTest(Test anot) {
        if(!anot.description().endsWith("NOLOG"))
            return true;
        else
            return false;
}
```

Obrázek 55 – ukázka vytvořeného vstupního bodu pro framework AspectJ

Na obrázku 55 je zachycena vstupní bod pro anotaci frameworku TestNG a to přesně „@Test“. Definice vstupního bodu nám říká: „Odchyť každou metodu, které je anotovaná anotací test, ať má jakýkoliv návratový typ, jakékoliv jméno a jakýkoliv počet vstupních argumentů. Tuto anotaci načti do proměnné anot a rozhodni, jestli se tento vstupní bod použije na základě definovaných podmínek“. Podmínka nutná pro rozhodnutí o použití tohoto vstupního bodu je poměrně jednoduchá. Tento vstupní bod se použije, pokud není v popisu daného testu uvedeno klíčové slovo „NOLOG“. Zavedením této podmínky si vytváříme možnost explicitně definovat testy, které si nepřejeme logovat, ať už z jakéhokoliv důvodu.

```
/**
* If PointCut for LoggedType.Method
*
* @param anot annotation on function
 * @return boolean
 */
@Pointcut("execution(@cz.airbank.airtest.core.AnnotationsClass
.Logged * *(..)) && @annotation(anot) && if()")
public static boolean pointcutLoggedAnnotMethod(Logged anot) {
return anot.type() == LoggedType.METHOD;
}
```

Obrázek 56 – vstupní bod pro odchycení vytvořené anotace @Logged

Z obrázku 56, který je velmi podobný předchozímu vstupnímu bodu, si osvětlíme návratovou hodnotu. V tomto konkrétním případě dojde k navrácení hodnoty pravda pouze v případě, že typ anotace „@Logged“ má hodnotu „METHOD“, což je implicitní hodnota, které je nastavena automaticky. A posléze na základě této návratové hodnoty je určen další osud tohoto vstupního bodu. Na dalším obrázku ukazujícím kód si osvětlíme funkci „Advice“ frameworku AspectJ a popíšeme, co se děje v případě validního vstupního bodu.

```
@Around("pointcutLoggedAnnotMethod(anot) && if(true)")
public Object logLoggedAnnotMethod(final ProceedingJoinPoint
thisJoinPoint, Logged anot) throws Throwable {
return createReportStep(thisJoinPoint, anot, new ReportStep());
}
```

Obrázek 57 – část „Advice“ funkčnosti frameworku AspectJ

Anotaci „@Around“ říkáme u této rady, že se má spustit se začátkem odchycené metody, převzít nad ní kontrolu, vykonat zamýšlené kroky, nechat samotnou metodu vykonat vlastní kód a provést dokončující kroky. Návratovou hodnotou metody, jež tuto radu reprezentuje, je pochopitelně obecný typ „Object“, neboť skutečný návratový typ může být jakýkoliv.

V rámci této metody provedeme vytvoření reportovacího kroku. V první řadě zjistíme název odchycené metody, necháme ji vykonat svoji činnost a změříme čas jejího vykonání. Následně jsme schopni ze zjištěných informací vytvořit kompletní objekt reportovacího kroku, který uložíme do již existujícího objektu reportu. Více prozradí další obrázek, který zachytí obsah funkce pro vytvoření reportovacího kroku. Na identickém principu funguje vytvoření objektu reportu při vstupu do metody anotované „@Test“, které logicky předchází volání vlastních metod implementovaných na PO a tedy vytváření kroků. Avšak z důvodu komplexnosti této funkce a pro jednodušší pochopení zde uvádíme právě vytvoření jednotlivého kroku.

```
private Object createReportStep(final ProceedingJoinPoint
thisJoinPoint, Logged anot, ReportStep step) throws Exception,
Throwable {
// Gets hooked function name and class name.
final String className =
thisJoinPoint.getSignature().getDeclaringType().getName();
final String methodName = thisJoinPoint.getSignature().getName();
final String qualifiedMethodName = String.format("%s.%s", className,
methodName);
// Returned (value) object after function proceedings
Object returnedValue = null;

thisJoinPoint.getSignature();
String[] parametrNames = methodSignature.getParameterNames();
Logged loggedAnnotation = (Logged)
methodSignature.getMethod().getAnnotation(Logged.class);
. . .
```

Obrázek 58 – částečná ukázka refaktorované funkce pro vytvoření reportovacího kroku

Část kódu na obrázku 58 nám umožňuje nahlédnout do logiky tvorby nového reportovacího kroku. Jedná se o záměrně oddělenou část do samostatného bloku pro udržení přehlednosti kódu. Na tomto krátkém útržku metody, můžeme vidět získání

popisných informací odchycení metody. Jinak řečeno zde získáváme její název, názvy jejich parametrů, celkovou cestu k dané metodě a to v rámci hierarchie našeho projektu.

10.2.2 Vytvoření konfiguračního souboru pro AspectJ

Abychom celé naše snažení uvedli úspěšně k životu, je nutné připojit k JVM nástroj, který pro náš bude třídy Aspektu interpretovat. Tento nástroj se nazývá weaver a v našem konkrétním případě pro nás vykoná tzv. LTW (Load Time Weaving). Tento způsob je oproti druhému existujícímu přístupu BCW (Byte Code Weaving) sice o něco náročnější na paměť, avšak zaručuje nám možnost kdykoliv „weaver“ odstavit. Ke správné funkci celého mechanismu je navíc nutné dodat do projektu konfigurační soubor, který informuje o třídách obsahujícími aspektové třídy.

```
-javaagent:\cesta\aspectjweaver-1.7.4.jar
-d64
-Xms1024m
-Xmx4g
-XX:MaxPermSize=256m
```

Obrázek 59 – parametry pro JVM nutné pro spuštění operace LTW frameworku AspectJ

Jak vidíme na obrázku 59, máme zde hned několik parametrů. Nejdůležitějším parametrem je samozřejmě „-javaagent:“ následovaný cestou k souboru příslušné verze AspectJ frameworku. Další z parametrů požaduje spuštění JVM v 64bitovém módu a to převážně kvůli schopnosti adresovat větší množství paměti RAM. Načež zbylé parametry upravují právě velikost alokované paměti po startu JVM. Více o AspectJ [20]

```
<aspectj>
<aspects>
<aspect name="cz.airbank.airtest.core.ReportAspect" />
<aspect name="cz.airbank.airtest.core.TimeMeasurementAspect" />
</aspects>
<weaver options="-
Xset:weaveJavaPackages=true,weaveJavaxPackages=true">
</weaver>
</aspectj>
```

Obrázek 60 – Konfigurační soubor pro AspectJ framework

10.3 Pohled do reportingové databáze

V rámci návrhu řešení s využitím frameworku Hibernate, který slouží k persistování celých Java objektů, dojde k vytvoření určité databázové struktury v rámci definice našich mapovacích tříd. Pojďme se nyní společně podívat na jejich strukturu a definovat užitečné SQL dotazy pro vytažení potřebných dat z reportingu.

```
CREATE TABLE `report` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `binaryPicture` longblob,
  `creationDate` datetime DEFAULT NULL,
  `iterationName` varchar(255) DEFAULT NULL,
  `printscreen` varchar(255) DEFAULT NULL,
  `reportName` varchar(255) DEFAULT NULL,
  `runByUser` varchar(255) DEFAULT NULL,
  `runOnMachine` varchar(255) DEFAULT NULL,
  `runTime` double NOT NULL,
  `stackTrace` longtext,
  `state` varchar(255) DEFAULT NULL,
  `suite` varchar(255) DEFAULT NULL,
  `iteration_fk` int(11) DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `REPORT_DATE` (`creationDate`),
  KEY `REPORT_NAME` (`reportName`),
  KEY `REPORT_SUITE` (`suite`),
  KEY `REPORT_ITERATION` (`iteration_fk`),
  KEY `REPORT_RUNBYUSER` (`runByUser`),
  KEY `REPORT_ITERATIONNAME` (`iterationName`),
  CONSTRAINT `FK_nnf1ln5ieebci2halbygwc078` FOREIGN KEY
(`iteration_fk`) REFERENCES `reportiteration` (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=11 DEFAULT CHARSET=utf8;
```

Obrázek 61 – SQL přikaz na vytvoření tabulky Report

Obrázek 61 reprezentuje SQL dotaz, který zapříčiní vytvoření tabulky uchovávající jednotlivé reporty. Mezi její nejdůležitější položky patří primární klíč „id“, díky kterému je možné tabulku reportu provázat s jeho podřízenou položkou a to jednotlivým krokem. Následně hodnota položky „iteration_fk“ slouží k provázání jednotlivých reportů do větší skupiny jediné iterace. Ještě je dobré si všimnout, že nad tabulkou jsou vytvořeny indexy pro jednodušší vyhledávání. Samozřejmostí je existence

omezení, která brání vymazání reportu, jež spadá do nějaké iterace. Další položkou databáze, na kterou se podíváme, bude reportingový krok.

```
CREATE TABLE `reportstep` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `addionalInfo` varchar(255) DEFAULT NULL,
  `creationDate` datetime DEFAULT NULL,
  `paramValues` varchar(255) DEFAULT NULL,
  `params` varchar(255) DEFAULT NULL,
  `returnedValue` varchar(255) DEFAULT NULL,
  `runTime` double NOT NULL,
  `state` varchar(255) DEFAULT NULL,
  `stepName` varchar(255) DEFAULT NULL,
  `type` varchar(255) DEFAULT NULL,
  `report_fk` int(11) DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `REPORTSTEP_DATE` (`creationDate`),
  KEY `REPORTSTEP_STATE` (`type`),
  KEY `REPORTSTEP_REPORT` (`report_fk`),
  CONSTRAINT `FK_2nblbkvkubmof4hk92ul1moat` FOREIGN KEY
(`report_fk`) REFERENCES `report` (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=32 DEFAULT CHARSET=utf8;
```

Obrázek 62 – SQL příkaz pro vytvoření tabulky reportingového kroku

Na obrázku výše vidíme příkaz, jehož zadáním bude vytvořena tabulka reprezentující jednotlivý reportovaný krok frameworku. Přesněji řečeno se jedná o metody tříd, které jsou anotované „@Logged“. Opět si můžeme všimnout, že je zde zavedeno omezení, které brání smazání kroku, který náleží k nějakému existujícímu reportu. Za povšimnutí také stojí nastavení automatické inkrementační strategie primárního klíče, které nám zajistí hlídání hodnoty ze strany databáze. Díky tomu není nutné hlídat hodnotu přímo na straně programu, kdy může docházet při spuštění ve více vláknech ke kolizím a nutnosti vlákna synchronizovat.

Poslední z obrázků nám přinese pohled na hierarchicky nejvýše postavenou entitu a to iteraci. Iterace vznikla převážně z důvodu možnosti sdružovat velké množství reportů do společné skupiny a vytváření skupin, které testují jedinou oblast. Avšak nejsou tuto oblast schopné otestovat pouze jedním testem, nebo je takové testování například rozloženo do několika dnů. Více o frameworku Hibernate provádějící mimo jiné i vytváření uváděných tabulek lze nalézt v [21].

```
CREATE TABLE `reportiteration` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `buildLog` longtext,
  `createdBy` varchar(255) DEFAULT NULL,
  `creationDate` datetime DEFAULT NULL,
  `iterationName` varchar(255) DEFAULT NULL,
  `locked` int(11) DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `REPORTITERATION_DATE` (`creationDate`),
  KEY `REPORTITERATION_NAME` (`iterationName`),
  KEY `REPORTITERATION_CREATEDBY` (`createdBy`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

Obrázek 63 – SQL příkaz k vytvoření tabulky iterace

10.4 Tvorba užitečných SQL příkazů

Ve snaze o historizaci dat spouštěných funkční testů vytvářeného frameworku, je dobré definovat základní sadu SQL příkazů, které následně můžeme využít pro tvorbu výstupných reportů. Tyto reporty budou reflektovat úspěšnost, postupný trend, či poskytovat doplňující informace k dříve objeveným chybám.

```
select * from report r
join reportstep s on s.report_fk = r.id
where r.id = 5
order by s.id
```

Obrázek 64 – SQL dotaz pro zobrazení reportu s jeho kroky

Výše uvedený SQL dotaz nám slouží k zobrazení zvoleného reportu dle jeho primárního klíče a následnému spárování všech dostupných kroků, které nesou jeho primární klíč. Jinak řečeno po sestupném seřazení, získáme chronologickou posloupnost logovaných kroků frameworkem.

creationDate	paramValues	params	returnedValue	runTime
2014-04-04 11:59:52	[illegible]	pass	cz.vut.core.page.vut.VUTLoginPageImp@6b321f49	759
2014-04-04 11:59:53	NULL	NULL	cz.vut.core.page.vut.VUTLoginPageImp@6b321f49	75
2014-04-04 11:59:50	NULL	NULL	cz.vut.core.page.vut.VUTLoginPageImp@6b321f49	73
2014-04-04 11:59:50	xsiebe03	login	cz.vut.core.page.vut.VUTLoginPageImp@6b321f49	1672
2014-04-04 11:59:53	NULL	NULL	cz.vut.core.page.vut.VUTISPageImp@5eb2e0fb	2303

Obrázek 65 – Ukázka části výstupu výše uvedeného SQL dorazu

11 VYUŽITÍ JINÝCH PROHLÍŽEČŮ A PARAMETRIZACE

11.1 Parametrizace

Často nastává situace, kdy potřebujeme testovat webovou aplikaci i na jiném systému, či prohlížeči. V této kapitole se společně zaměříme právě na možné využití dalších prohlížečů a to konkrétně Internetu Exploreru, který je stále hojně využívám v doprovodu Firefoxu a prohlížeče Chrome.

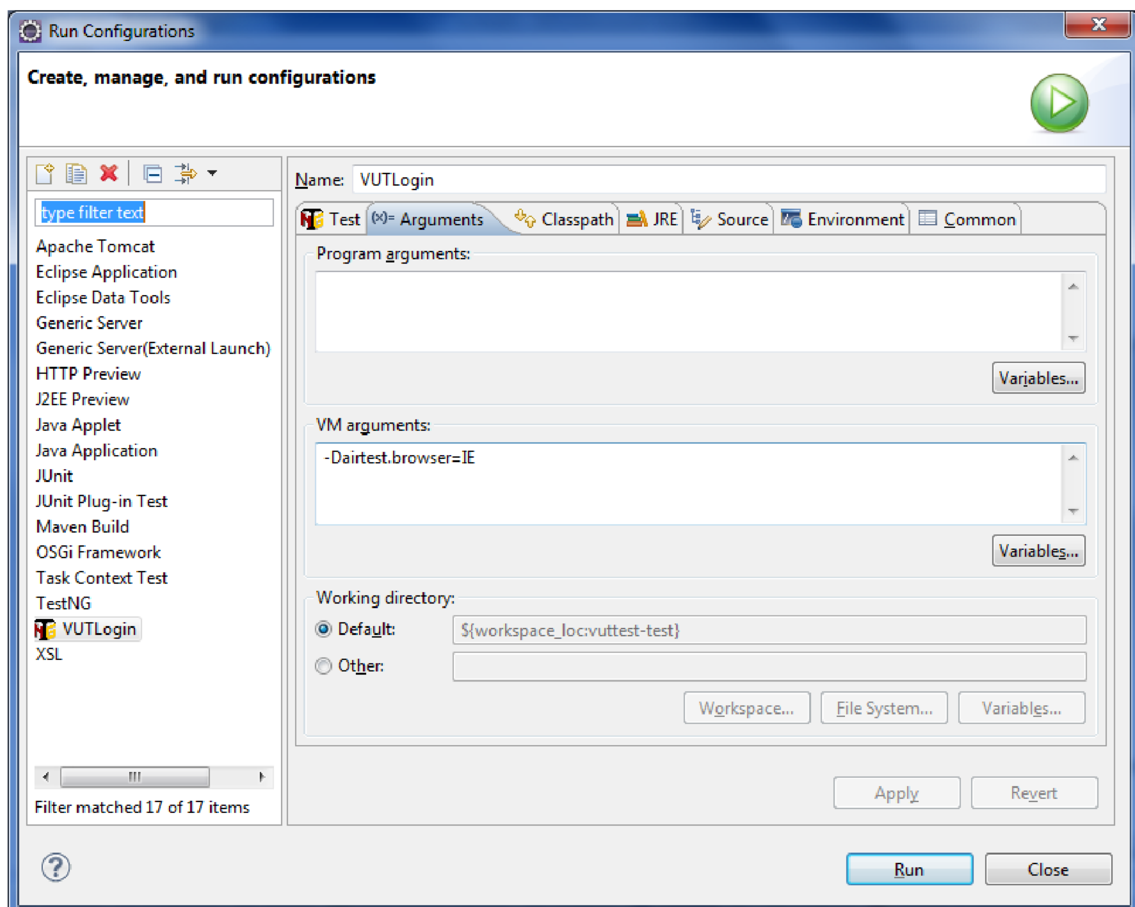
Abychom takovou myšlenku byly schopni realizovat, je potřeba navrhnout řešení, které umožní snadnou parametrizaci při spuštění frameworku. Přesněji řečeno vytvoříme třídu, které bude definovat sadu vstupních parametrů, které budou předány rovnou při spouštění JVM.

```
public final static String DRIVER_CONFIG = "vuttest.driver";
public final static String CLOSE_BROWSER_CONFIG =
"vuttest.closeBrowser";
public final static String BROWSER_CONFIG = "vuttest.browser";
public final static String REPORT_ITERATION = "vuttest.iteration";
public final static String REPORT_LOGGING = "vuttest.logging";
public final static String USERAGENT_STRING = "vuttest.userAgent";
```

Obrázek 66 – definice parametrů pro ovládání chování frameworku

Jak můžeme vidět na obrázku 66, definujeme několik různých parametrů. Pojďme si společně přiblížit, co který parametr dělá. V první řadě parametr „vuttest.driver“ udává, zdali se pro exekuci testovacího scénáře využije lokální, nebo tzv. vzdálený ovladač prohlížeče. Vzdálený ovladač prohlížeče není nic jiného, než běžící Selenium Grid na vzdálené testovací infrastruktuře. Tento přístup bude otestován později v této práci. Parametr „vuttest.closeBrowser“ nám říká, jakým způsobem se má ovladač prohlížeče (dále jen „driver“) zachovat po ukončení testu k oknu prohlížeče. Můžeme okno prohlížeče zavřít, anebo ho nechat zavřít jen v případě úspěšného testu, protože můžeme chtít chybový stav prozkoumat. Dalším z definovaných parametrů je „vuttest.browser“, který nám dává možnost zvolit mezi využitím Firefoxu, či Internetu Exploreru. Následující „vuttest.iteration“ umožňuje definovat iteraci a spouštěné testy do ní zařadit. Parametrem „vuttest.logging“ jsme schopni ovlivnit hloubku logování a případně donutit framework k okamžitému zapisovaní do databáze každého kroku, což sice umožní okamžitý přehled běhu, ale při větším množství neadekvátně vytíží

databázi. Posledním parametrem je „vuttest.userAgent“, v mnoho případech si u aplikací vystačíme s emulováním řetězce „user agent“, abychom vytvořili pro aplikaci dojem, že ji načítá například Google Chrome. Díky takovému řešení může test běžet v prohlížeči Firefox, který má ze strany Selenia nativní podporu a nejrozsáhlejší možnosti využití.



Obrázek 67 – ukázka předání parametru frameworku v rámci IDE Eclipse

Obrázek výše ukazuje možné předání parametru přímo do JVM a následně je tento parametr postoupen frameworku a jsou provedeny definované akce. Dokonce samotné integrované vývojové studio umožňuje vytvářet předpisy pro spouštění různých testů. Tzn., že si můžeme vytvořit pojmenovanou konfiguraci určenou pro spuštění nějakého testu a rovnou v ní definovat parametry.

11.2 Spuštění testu v prohlížeči Internet Explorer

Protože Internet Explorer není nativně podporován, vzhledem k faktu, že není možné jej ovládat zasíláním http příkazu ve formátu „JSON Wire Protocol“, tak je nutné využít prostředníka mezi Seleniem a instancí IE. Tento program je vytvořen

formou serveru, který potřebnou komunikaci obstarává. Nyní společně nahlédneme do třídy, které pro nás vytváří instanci „driveru“ a jakým způsobem ji říci, aby pro spouštěný test zvolila právě IE.

```
private static WebDriver getLocalDriver() {

WebDriver driver = null;

switch (ConfigConstants.getBrowserConfig()) {
case IE:
System.setProperty("webdriver.ie.driver",
ConfigConstants.IE_SERVER);
driver = new InternetExplorerDriver();
break;
case FIREFOX:
default:
driver = new FirefoxDriver();
break;
}

return driver;
}
```

Obrázek 68 – funkce pro získání Selenium driveru s přihlédnutím k možnosti volit mezi dvěma prohlížeči

Vidíme na obrázku 68, že volba prohlížeče je podmíněna získáním návratové hodnoty metody „getBrowserConfig“, která nám vrací enumerační typ, jež reprezentují možnosti přepínače IE a Firefox. Pokud není takový parametr vůbec předán, je návratová hodnota metody „getBrowserConfig“ opatřena standardním nastavením v podobě vrácení hodnoty Firefox. Na obrázku 69 se můžeme podívat, jak vypadá funkce určená pro navrácení hodnoty předávaného parametru frameworku, konkrétně se jedná o výše dvakrát jmenovanou. Pokud ji prozkoumáme blíže, můžeme si všimnout standardně vracené hodnoty enumeračního typu. Další neméně podstatnou věcí, která se nalézá ještě na obrázku výše je nastavení hodnoty systémové proměnné „webdriver.ie.driver“ na hodnotu reprezentující umístění spustitelného souboru serveru pro IE. Je zde však ještě jiná možnost a to umístění cesty odkazující v rámci souborového systému na spustitelný soubor serveru do proměnné „PATH“ operačního systému. Potom by nebylo nutné explicitně nastavovat systémovou proměnou „webdriver.ie.driver“, ale cesta by byla úspěšně nalezena automaticky. Avšak

s přihlédnutím k faktu, že tester využívající automatizační frameworku, nemusí mít práva systémového administrátora, byl zvolen tento způsob.

```
public static BrowserConfig getBrowserConfig() {
String browserConfig;
browserConfig = System.getProperty(BROWSER_CONFIG, "FIREFOX");

try {
BrowserConfig ret = BrowserConfig.valueOf(browserConfig);
log.trace("Browser option: " + ret.toString());
return ret;
} catch (IllegalArgumentException e) {
throw new RuntimeException("Bad configuration option for browser.
Possible values are: " + printConfigValues(BrowserConfig.class), e);
  }
}
```

Obrázek 69 – vzhled metody pro navrácení hodnoty předaného parametru

Ještě dodejme, že na analogickém principu funguje využití prohlížeče Google Chrome. V každém případě, pokud nemáme prohlížeče standardně nainstalované a tedy nejsou jejich spustitelné soubory umístěny v systémové proměnné „PATH“. Potom je nutné explicitně dodat cesty k jejich spustitelným souborům. Nejlepším řešením je však provést instalaci veškerých prohlížečů, které plánujeme využívat standardní cestou.

12 SELENIUM GRID PRAKTICKY

S rostoucím množstvím testů, souběžně roste čas potřebný k jejich spouštění. Samozřejmě se stále jedná o značně rychlejší způsob oproti ručnímu vykonávání testů, ale přece jen existuje lákavá možnost, jak testy spouštět paralelně a distribuovat na více vzdálených strojů. Celou tuto funkcionalitu máme již předpřipravenou v podobě nástroje „Selenium Grid".

Tento nástroj funguje na principu klient – server – klient. Pro názornou ukázku si za uvedené role dosadíme konkrétní stroje. První role klienta je obsazena naším testovacím frameworkem, který skrze „JSON Wire protocol" odesílá http příkazy serveru Selenium Gridu, který pracuje v módu tzv. „HUBu". Posledním článkem naší skládačky je klient prakticky vykonávající příkazy serveru a nese označení „NOD". Takovýchto vzdálených klientů může být k serveru připojeno hodně, každý z těchto vzdálených strojů umožňuje parametrizovat informace o sobě. Přesněji řečeno serveru poskytuje informace o běžícím operačním systému, množství a typ instancí prohlížečů, které je schopný spustit a současné vytížení.

V takových informacích je veliká síla, neboť nám umožňující požádat server o zvolení konkrétního z těchto vzdálených strojů, který vyhovuje například našim specifickým požadavků na verzi operačního systému, prohlížeče atd. Dokonce nám server poskytuje jednoduché webové rozhraní, ve kterém lze sledovat aktuální obsazenost volných instancí prohlížečů. Nejspíše není nutné zdůrazňovat možnost využití takového řešení pro zátěžové testování webových aplikací. Kdy za cenu mírně vyšší náročnosti na odbornost tvůrců frameworku, dosáhneme schopnosti snáze řešit komplikace související například s dnes velice oblíbeným AJAXEM (Asynchronous JavaScript and XML).

Obrázek 70 – ukázka webového rozhraní serveru s jediným připojeným nodem

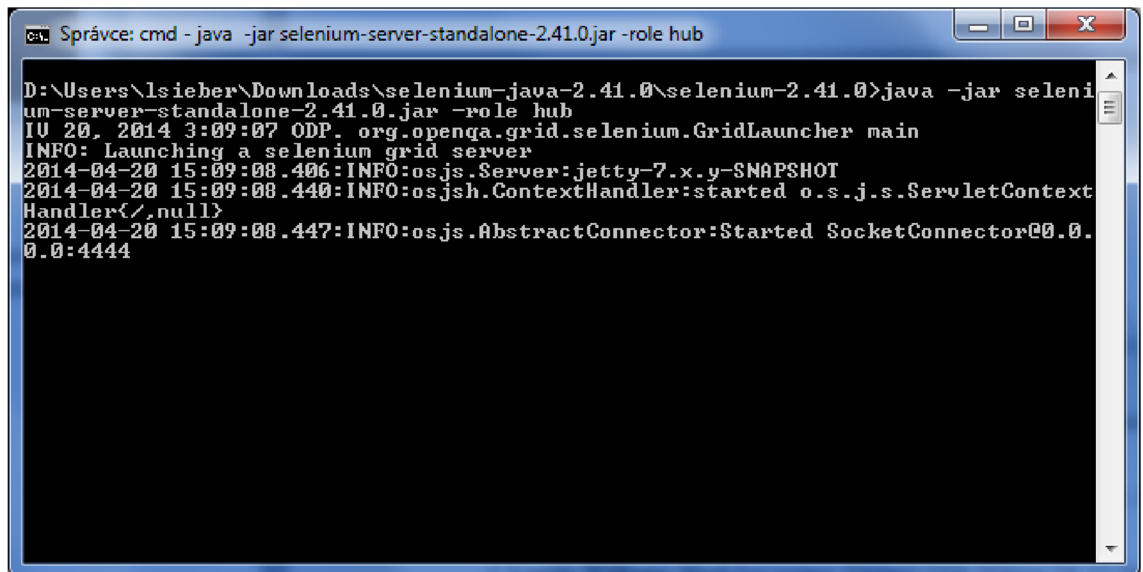
Jak je názorně vidět na obrázku 70, jsme schopni přehledně sledovat stav jednotlivých „Nod“ klientů připojených k Selenium Grid serveru. Pakliže je vykonáván test na některém z těchto klientů, je ikona prohlížeče zašedlá.

```
private static WebDriver getRemoteDriver() {
WebDriver driver;
DesiredCapabilities capability;

switch (ConfigConstants.getBrowserConfig()) {
case IE:
capability = DesiredCapabilities.internetExplorer();
break;
case FIREFOX:
default:
capability = DesiredCapabilities.firefox();
capability.setCapability("nativeEvents", false);

...

capability.setCapability(FirefoxDriver.PROFILE, profile);
break;
}

try {
driver = new RemoteWebDriver(new URL(ConfigConstants.GRID_HUB),
capability);
} catch (MalformedURLException e) {
throw new RuntimeException(e);
}

((RemoteWebDriver) driver).setFileDetector(new LocalFileDetector());
      return driver;
}
```

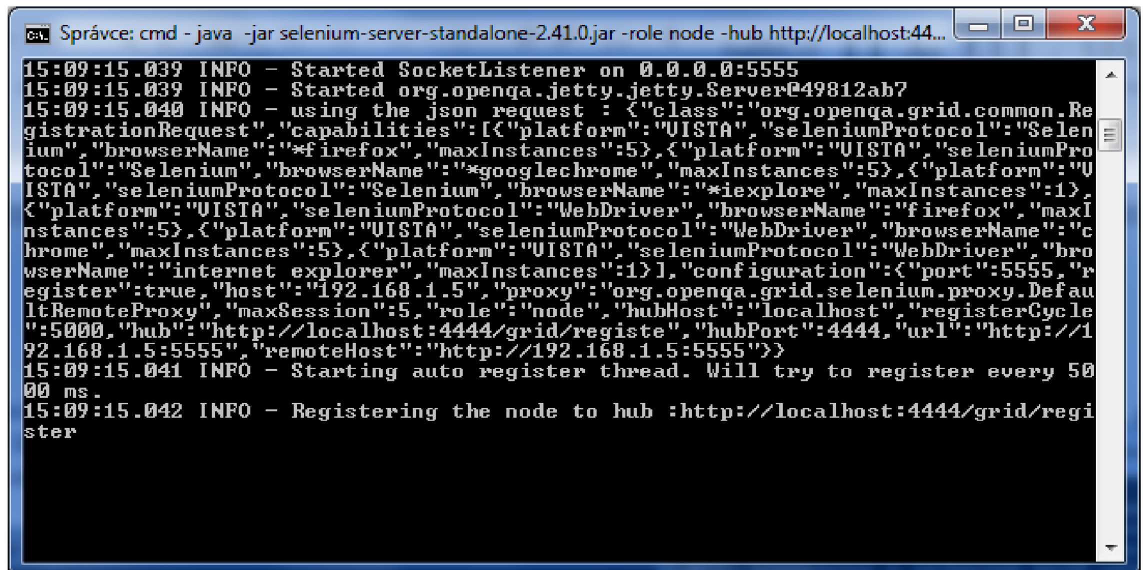
Obrázek 71 – částečná ukázka funkce zodpovědné za vytvoření tzv. „Remote driveru“ => driver komunikující se Selenium Grid serverem

Z praktického hlediska je pro nás nejzajímavější předposlední řádek, který říká vzdálenému driveru, aby používal lokální cesty vzdáleného serveru. Tento řádek dokáže

ušetřit mnoho starostí a hledání chyby u testů, které ve svém průběhu nahrávají nějaké soubory. Klasickým případem může být například nahrání fotky, nebo třeba souboru Excelu k nějakému bližšímu zpracování.



Obrázek 72 – okno reprezentující běžící Selenium Grid v režimu „Server“



Obrázek 73 – okno reprezentující běžící Selenium Grid v režimu „NODu“

Vzhledem k faktu, že předmětem práce není hloubkové vysvětlení principu fungování Selenium Grid, či jeho parametrizace. Tak pro bližší informace týkající se rozličných nastavení, či omezení, je doporučeno nahlédnout na [12].

13 NÁVRH NA ZLEPŠENÍ: JENKINS A SVN

S rostoucí komplexností projektu a množstvím vývojářů, kteří se jim zabývají, se nabízí možnost využití ozkoušených nástrojů pro týmovou spolupráci. V první řadě se jedná o vytvoření SVN serveru, který poskytne možnost verzování kódu a snadnou možnost návratu v případě způsobení nefunkčnosti provedenými změnami. Dalším velmi dobrým nápadem je nasazení serveru pro kontinuální integraci a tím je právě Jenkins. Smyslem tohoto nástroje bude při každé změně na SVN provést sestavení projektu a v případě chyby, či nemožnosti toto sestavení provést zaslání informačního mailu na vývojáře a upozornění na odeslání nefunkčního kódu. Možným druhotným využitím Jenkinse je vlastní spouštění testů a to díky integraci s nástrojem Maven.

13.1 Apache Subversion

Poměrně známou a hojně využívanou možnosti SVN je produkt „Apache Subversion“, který je šířen pod licenci „Apache Lincence 2.0“, která je kompatibilní s licencí GPL. Řečeno jinými slovy, jedná se o „Open source“ řešení a můžeme jej tedy využívat zcela zdarma. Pro bližší informace je možné zavítat na následující zdroj [22].

13.2 Jenkins

Abychom pochopili přínos serveru pro kontinuální integraci, pojďme si nejdříve povědět, co tento termín znamená. Jedná se o slovo, které označuje častou kompilaci projektu, které je obvykle doprovázena spouštěním testů. Jedná se tedy o přístup, který se snaží udržet vyvíjený a následně odeslaný kód, na co nejvyšší možné míře kvality. A právě jedním z nejznámějších a nejvíce využívaných serverů pro kontinuální integraci dostupných zdarma je Jenkins.

V našem případě můžeme tento produkt využít takových způsobem, že v první řadě bude stále po každém odeslání nového kódu frameworku provádět kompilaci a případně upozorňovat na nefunkční kód. Avšak hlavně jej můžeme využít na vzdálené spouštění testů formou vytvořených „Maven Jobů“. Úplně konkrétně si to můžeme představit například takto:

- Definujeme soubor testů, který chce spustit po nasazení nového IS VUT
- Vytvoříme Job s pomocí webového rozhraní Jenkinse, který bude možné přes jeho webové rozhraní spustit.
- Po nasazení nového IS VUT jednoduše spustíme tento test a ověříme výsledky
- Výsledky jsou historicky ukládány a jsou generovány grafy úspěšnosti

Už je tedy nejspíše zřejmě, že Jenkins ještě více usnadní spouštění testů a bude hlídat, že změny v našem frameworku zachovávají jeho funkčnost. A protože spouštět joby Jenkinse lze přes jeho webové rozhraní, zvládne tento úkol i úplný laik. Více [23]

All | All Disabled | **All Failed** | All Unstable | Infrastructure | Jenkins core | Libraries | Other Projects

S	W	Name ↓	Poslední úspěšný build	Poslední neúspěšný build	Délka posledního sestavení
		core_selenium-test	žádný	1 yr 5 mo - #18	12 min
		gerrit_master	11 mo - #4011	9 mo 26 days - #4252	10 min
		infra_backend-war-size-tracker	24 days - #533	10 hr - #557	1 min 24 sec
		infra_commit_history_generation	24 days - #326	19 hr - #350	3 min 57 sec
		infra_extension-indexer	1 mo 3 days - #98	22 hr - #103	2 hr 42 min
		infra_github_repository_list	24 days - #1141	9 hr 45 min - #1165	6 min 33 sec
		infra_plugin_changes_report	27 days - #304	6 days 6 hr - #309	13 min

Obrázek 74 – ukázka spustitelných jobů ve webovém rozhraní nástroje jenkins dostupné z https://ci.jenkins-ci.org/

14 PRAKTICKÉ OVĚŘENÍ FUNKČNOSTI

V rámci praktického ověření funkčnosti frameworku bylo vytvořeno 7 funkčních testů, které demonstrují postup vytváření jednotlivých testů a také možnosti frameworku. A to ať již z pohledu parametrizace, tak i z pohledu reportingových schopností vytvořeného řešení.

14.1 Popis vytvořených testů

V této podkapitole se zaměříme na popis testovacích scénářů a jejich vlastní exekuci s následným vyhodnocením výsledků v rámci dvou internetových prohlížečů a to přesně Internetu Exploreru a Firefoxu.

14.1.1 VUTNavigation

Test VUTNavigation sestává z přihlášení do systému VUT a následné navigace na několik zvolených stránek. Jedná se postupně o tyto stránky:

- Záložka Portál
- Záložka Student
- Student => Individuální rozvrh
- Student => Rozvrhy
- Student => Studium
- Student => Registrace termínu
- Student => Úvodní informace
- Student => Elektronický index
- Student => Osobní informace
- Portál => VUT zprávy
- Portál => Mikropoplatky

Po dokončení navigace na všechny vyjmenované stránky je test ukončen odhlášení z informačního systému VUT.

14.1.2 VUTLogin

Tento test provede vyplnění přihlašovacího jména a hesla s následným přihlášením a navigací na stránku intraportál v rámci IS VUT.

14.1.3 VUTCommonInformation

V rámci tohoto testu se provede přihlášení do IS a následná navigace na záložku student a stránku úvodní informace. V rámci této stránky je ověřen text u následujících položek oproti textu očekávanému:

- Fakulta
- Název studijního programu
- Typ studia
- Forma studia
- Obor
- Ročník
- Číslo studia

V případě, že některý z textů neodpovídání očekávanému, dojde k selhání celého testu.

14.1.4 VUTMessages

U tohoto testů se provede po přihlášení navigace na stránku VUT zpráv pod záložkou portál s následným zobrazením detailu vybrané zprávy. Posléze je provedena navigace z detailu s pomocí odkazu zpět a opětovně je zobrazen detail nějaké zprávy, avšak tentokrát s kliknutím na tlačítko odpovědět.

14.1.5 VUTMicroTransaction

Předmětem testu je navigace na stránku mikro poplatky v rámci záložky portál a ověření čísla karty včetně dostupného zůstatku.

14.1.6 VUTPersonalSchedule

U tohoto testu dojde k navigaci na stránku individuální rozvrh a posléze k změně zobrazeného rozvrhu na letní semestr, typu rozvrhu na standardní rozvrh, je zatrženo zobrazit volné dny a je zatrženo skrýt detail.

14.1.7 VUTSchedule

Tímto posledním testem je provedeno filtrování na stránce rozvrhů dle id osoby. Takto nalezený rozvrh je zobrazen a jsou s ním provedeny identické akce jako v případě testu VUTPersonalSchedule.

14.2 Porovnání trvání testů v rámci IE a FF

Jméno testu	IE 9 (s)	FF 24ESR (s)	compare (IE-FF) (s)
cz.vut.test.VUTCommonInformation	19,0442	13,3380	5,7062
cz.vut.test.VUTLogin	12,7218	10,5294	2,1924
cz.vut.test.VUTMessages	23,4182	16,9498	6,4684
cz.vut.test.VUTMicroTransaction	15,7104	11,7128	3,9976
cz.vut.test.VUTNavigation	38,0698	23,1718	14,898
cz.vut.test.VUTPersonalSchedule	28,0970	19,1176	8,9794
cz.vut.test.VUTSchedule	30,8370	21,6982	9,1388

Tabulka 1 – porovnání délky trvání testu IE vs FF

V rámci možného porovnání délky trvání testu (Rychlosti prohlížeče pro zobrazení daných stránek) byl proveden test sestávající z 5 x provedeného spuštění všech 7 testovacích scénářů a to pro každý prohlížeč zvlášť. Výsledná doba trvání testu je vyjádřena jako průměrná doba každého z testů.

Z pohledu rychlosti jasně vede prohlížeč Firefox. Je pochopitelné, že v rámci nativní podpory Firefoxu ze strany Selenium Webdriveru lze očekávat nižší hodnoty času. Avšak rozdíl hodnot obou prohlížečů blížící se hranici 10 sekund, již nelze přisuzovat pouze další mezivrstvě v podobě serveru ovládající prohlížeč Internet Explorer. Můžeme tedy říci, že lepší „User experience“ dosáhne uživatel využitím prohlížeče Firefox, a to v případě prohlížení tetovaných stránek VUT.

14.3 Vytvoření testovací suity

Abychom dosáhli ještě jednodušší možnosti spuštění všech 7 testovacích scénářů, byla vytvořena testovací suita. Tento XML soubor dodá testovacímu frameworku TestNG potřebné informace ke spuštění všech testovacích scénářů. Na obrázku níže se můžeme podívat, jak je taková suita definována.

```
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd" >

<suite name="Suite" verbose="1">
      <test name="VUTCommonInformation">
            <classes>
                  <class name="cz.vut.test.VUTCommonInformation" />
            </classes>
      </test>
...
```

Obrázek 75 – částečný výpis definice suity pro spuštění všech scénářů

14.4 Rozklad vybraných scénářů na jednotlivé kroky

Tato podkapitola se zaměří na rozpad testovacích scénářů na nejmenší jednotlivé reportingové kroky a to v rámci provedeného měření pro prohlížeče IE a FF. Řazení je provedeno sestupně dle trvání v sekundách.

14.4.1 VUTCommonInformation

Název kroku	Trvání (s)
VUTLoginPageImpl.submit	0.8898
VUTLoginPageImpl.fillLoginTextBox	1.1826
VUTLoginPageImpl.fillPassTextBox	1.3272
AbstractVUTISPageImpl.navigateToStudentPage	2.1094
AbstractVUTISStudentPageImpl.navigateToCommonInformationPage	2.1342
VUTLoginPageImpl.clickOnLoginLink	2.4666
VUTLoginPageImpl.navigateToIntraportal	2.7750
AbstractVUTISPageImpl.navigateToLogout	2.7790

Tabulka 2 – přehled kroků pro IE v rámci VUTCommonInformation scénáře

Název kroku	Trvání (s)
VUTLoginPageImpl.clickOnLoginLink	0.0720
VUTLoginPageImpl.submit	0.0756
VUTLoginPageImpl.fillPassTextBox	0.7606
AbstractVUTISPageImpl.navigateToLogout	1.0970
AbstractVUTISPageImpl.navigateToStudentPage	1.1596
AbstractVUTISStudentPageImpl.navigateToCommonInformationPage	1.7532
VUTLoginPageImpl.fillLoginTextBox	2.4444
VUTLoginPageImpl.navigateToIntraportal	2.9092

Tabulka 3 - přehled kroků pro FF v rámci VUTCommonInformation scénáře

14.4.2 VUTMessages

Název kroku	Trvání (s)
VUTLoginPageImpl.fillLoginTextBox	0.9004

VUTLoginPageImpl.fillPassTextBox	1.2118
VUTLoginPageImpl.submit	1.3168
VUTMessagesDetailPageImpl.clickOnReply	2.0506
VUTMessagesDetailPageImpl.clickOnBack	2.0868
AbstractVUTISPortalPageImpl.navigateToVUTMessagesPage	2.1266
VUTLoginPageImpl.navigateToIntraportal	2.2580
VUTLoginPageImpl.clickOnLoginLink	2.3058
AbstractVUTISPageImpl.navigateToLogout	2.5636
VUTMessagesPageImpl.shoMessageDetail	4.2722

Tabulka 4 - přehled kroků pro IE v rámci VUTMessages scénáře

Název kroku	Trvání (s)
VUTLoginPageImpl.clickOnLoginLink	0.0706
VUTLoginPageImpl.submit	0.0736
VUTLoginPageImpl.fillPassTextBox	0.7672
VUTMessagesDetailPageImpl.clickOnBack	1.0888
AbstractVUTISPortalPageImpl.navigateToVUTMessagesPage	1.1166
VUTMessagesDetailPageImpl.clickOnReply	1.1320
AbstractVUTISPageImpl.navigateToLogout	1.1402
VUTMessagesPageImpl.shoMessageDetail	2.2622
VUTLoginPageImpl.navigateToIntraportal	2.6440
VUTLoginPageImpl.fillLoginTextBox	2.9290

Tabulka 5 - přehled kroků pro FF v rámci VUTMessages scénáře

14.4.3 VUTNavigation

Název kroku	Trvání (s)
VUTLoginPageImpl.fillLoginTextBox	0.9312
VUTLoginPageImpl.submit	1.0986
VUTLoginPageImpl.fillPassTextBox	1.2108
AbstractVUTISStudentPageImpl.navigatesToVUTPersonalSchedulePage	2.0342
AbstractVUTISStudentPageImpl.navigateToCommonInformationPage	2.0614
AbstractVUTISPageImpl.navigateToPortalPage	2.0830
AbstractVUTISPageImpl.navigateToStudentPage	2.0866
AbstractVUTISPortalPageImpl.navigateToVUTMessagesPage	2.0890
AbstractVUTISStudentPageImpl.navigatesToVUTSchedulesPage	2.1776
AbstractVUTISStudentPageImpl.navigateToVUTIndexPage	2.2558
AbstractVUTISStudentPageImpl.navigatesToVUTTermRegistrationPage	2.2866
AbstractVUTISStudentPageImpl.navigateToVUTPersonalInformationPa ge	2.3148
VUTLoginPageImpl.clickOnLoginLink	2.3184

AbstractVUTISPageImpl.navigateToLogout	2.4932
VUTLoginPageImpl.navigateToIntraportal	2.5474
AbstractVUTISPortalPageImpl.navigateToVUTMicroTransactionsPage	2.8826
AbstractVUTISStudentPageImpl.navigatesToVUTStudyPage	2.8954

Tabulka 6 - přehled kroků pro IE v rámci VUTNavigation scénáře

Název kroku	**Trvání (s)**
VUTLoginPageImpl.clickOnLoginLink	0.0678
VUTLoginPageImpl.submit	0.0700
VUTLoginPageImpl.fillPassTextBox	0.7580
AbstractVUTISPageImpl.navigateToLogout	1.1218
AbstractVUTISStudentPageImpl.navigatesToVUTPersonalSchedulePage	1.1346
AbstractVUTISPageImpl.navigateToPortalPage	1.1396
AbstractVUTISStudentPageImpl.navigatesToVUTTermRegistrationPage	1.1452
AbstractVUTISStudentPageImpl.navigateToVUTIndexPage	1.1492
AbstractVUTISPageImpl.navigateToStudentPage	1.1572
AbstractVUTISStudentPageImpl.navigateToCommonInformationPage	1.1668
AbstractVUTISPortalPageImpl.navigateToVUTMessagesPage	1.1700
AbstractVUTISStudentPageImpl.navigateToVUTPersonalInformationPa ge	1.1792
AbstractVUTISStudentPageImpl.navigatesToVUTSchedulesPage	1.1886
AbstractVUTISPortalPageImpl.navigateToVUTMicroTransactionsPage	1.3828
AbstractVUTISStudentPageImpl.navigatesToVUTStudyPage	1.5372
VUTLoginPageImpl.fillLoginTextBox	2.3324
VUTLoginPageImpl.navigateToIntraportal	2.6334

Tabulka 7 - přehled kroků pro FF v rámci VUTNavigation scénáře

Tento přehled jasně ukazuje, že stejné kroky mohou trvat různou dobu v obou testovaných prohlížečích. Výsledky jsou získány agregací 5 spuštění každého testu a následným výpočtem průměru ze součtu časů u jednotlivých samostatných kroků. Rozdílnost v časech může být způsobena například různým systémem vyhodnocení konce načtení stránky u zkoušených prohlížečů, které vyústí v odeslání informace nástroji Selenium resp. našemu frameworku. A teprve posléze je ukončeno provádění metody (reportovacího kroku) a měření času.

ZÁVĚR

Cílem této práce bylo vytvoření nástroje (frameworku) v programovacím jazyce Java, který umožní svěřit tvorbu testovacích scénářů do rukou lidem, kteří nemají hluboké znalosti z oblasti programování. V rámci tohoto požadavku byla vytyčena cesta oddělení programového popisu stránek tzv. „Page objektů“ a dalšího pomocného kódu do projektu jádra. Tento projekt je zcela oddělen od projektů testů za účelem kompletního odstínění tvůrců testovacích scénářů od komplexních programových struktur a samozřejmě také uzamknutí dodávaného „know-how“. Oproti tomu projekt testů je pouze souborem scénářů a popřípadě testovacích suity frameworku TestNG. V rámci maximálního usnadnění tvorby testů je využíván tzv. „Fluent interface“, který díky automatickým návrhům dostupných metod dané třídy výrazně usnadňuje finální tvorbu scénáře. Každá z metod definovaných na PO využívá tzv. „Java-doc“ komentář, kdy nám integrované vývojové prostředí jmenovaný prvek zobrazí jako nápovědu.

Dalším důležitým krokem bylo vytvoření reportingového rozhraní, které je nedílnou součástí nutnou pro možné efektivní využití frameworku. A to především z pohledu schopnosti reportovat výsledky spouštění testů, ať již z krátkodobého, či dlouhodobého pohledu. Navíc jsou sbírány informace, které slouží k možnému posouzení, zda se jedná o chybu testované aplikace, nebo samotného frameworku. Vzhledem k ukládání výsledků do databáze s využitím framewoku Hibernate, lze využít řadu dostupných databází, které jsou podporovány. Mimo jmenované lze nasbíraná data využít například ke grafickému zpracování trendů výskytu chyb a zjišťovat tak, jakým směrem se projekt testované aplikace ubírá.

Poslední část práce v podání kapitoly 14. se zaměřila na praktické ověření funkčnosti frameworku a to otestováním na dvou hojně využívaných prohlížečích Internetu Exploreru (verze 9) a prohlížeče Firefox (verze 24 ESR). Bylo popsáno několik vybraných stránek informačního systému VUT a vytvořeno 7 samostatných testů, které byly následně zařazeny do jediné testovací suity pro jednodušší exekuci. Kompletní suita byla spuštěna 5x pro každý ze jmenovaných prohlížečů. Posléze byly z databáze vytaženy celkové časy trvání, jak testů, tak samostatných kroků a vypočteny jejich průměrné hodnoty.

Nespornou výhodou navrženého frameworku je jeho vytvoření z volně dostupných komponent, jež nevyžadují finanční náklady spojené s jejich zakoupením. Modularita daného řešení umožňuje za předpokladu dostupnosti kvalifikovaných pracovníků implementovat téměř jakoukoliv funkcionalitu, která se může ukázat jako problémová v případě využití komerčních řešení. Jako názorný příklad si uveďme dnes hojně využívaný AJAX, který může způsobit obtížné určení konce načítání webové aplikace. Samozřejmě existují velmi pokročilé komerční nástroje, avšak svoji cenou často překračující investice nutné do návrhu vlastního řešení s nástrojem Selenium.

BIBLIOGRAFIE

1. LAURIE, W. Intro to testing. In: [online]. © 2005 [cit. 2013-12-16]. Dostupné z: http://agile.csc.ncsu.edu/SEMaterials/IntrotoTesting.pdf
2. HUTCHESON, M. *Software Testing Fundamentals: Methods and Metrics*. Willey, 2003. ISBN 978-0471430209.
3. PATTON, R. *Software Testing (2nd Edition)*. SAMS, 2005. ISBN 978-0672327988.
4. LAURIE, W. Black box testing. In: [online]. © 2006 [cit. 2013-12-16]. Dostupné z: http://agile.csc.ncsu.edu/SEMaterials/BlackBox.pdf
5. LAURIE, W. White box testing. In: [online]. © 2006 [cit. 2013-12-16]. Dostupné z: http://agile.csc.ncsu.edu/SEMaterials/WhiteBox.pdf
6. KRUGER, J.*: When acceptance tests are better than unit tests* [online]. [cit. 2013-12-16]. Dostupné z: http://jonkruger.com/blog/2012/02/20/when-acceptance-tests-are-better-than-unit-tests/
7. MAZOCH, B. *Funkční testování webových aplikací*. Brno: Masarykova universita, Fakulta informatiky, 2012.
8. MCCONNELL, S. *Dokonalý kód*. Computer Press, 2006. ISBN 9788025108499.
9. HAAS, H. Service-Oriented Architecture. In: [online]. [cit. 2013-12-16]. Dostupné z: http://www.w3.org/2003/Talks/0521-hh-wsa/slide5-0.html
10. ECKEL, B. *Thinking in Java (4th Edition)*. MindView, Inc., 2006. ISBN 978-0131872486.

11. HEROUT, P. *Učebnice jazyka Java*. KOPP, 2010. ISBN 978-80-7232-398-2.

12. KOLEKTIV AUTORů.*: SeleniumHQ - Browser automation* [online]. [cit. 2013-12-17]. Dostupné z: http://docs.seleniumhq.org/projects/
13. KOLEKTIV AUTORů.*: WebDriver API* [online]. 12. 6. 2013 [cit. 2013-12-17]. Dostupné z: http://www.w3.org/TR/webdriver/
14. FOWLER, M. PageObject. In: [online]. 10. 9. 2013 [cit. 2013-12-17]. Dostupné z: http://martinfowler.com/bliki/PageObject.html
15. WIKIPEDIE. *Eclipse (vývojové prostředí): Eclipse (vývojové prostředí)* [online]. [cit. 2014-01-19]. Dostupné z: http://cs.wikipedia.org/wiki/Eclipse_(v%C3%BDvojov%C3%A9_prost%C5%99ed%C3%

AD)

16. WIKIPEDIE.: *Apache Maven* [online]. [cit. 2014-31-01]. Dostupné z: http://cs.wikipedia.org/wiki/Apache_maven
17. BEUST,. *TestNG: TestNG* [online]. 27. 4. 2004 [cit. 2014-2-23]. Dostupné z: http://testng.org/doc/index.html
18. KOLEKTIV AUTORů.: *Google Guice* [online]. [cit. 2014-3-5]. Dostupné z: https://code.google.com/p/google-guice/
19. KOLEKTIV AUTORů. *Hibernate. Everything data.: Hibernate* [online]. [cit. 2014-3-8]. Dostupné z: http://hibernate.org/
20. KOLEKTIV A. *AspectJ: Eclipse* [online]. [cit. 2014-4-18]. Dostupné z: http://eclipse.org/aspectj/
21. COMMUNITY, J. *Hibernate. Everything data.: Hibernate* [online]. [cit. 2014-4-18]. Dostupné z: http://hibernate.org/
22. KOLEKTIV A. *Subversion: Subversion* [online]. [cit. 2014-4-20]. Dostupné z: http://subversion.apache.org/
23. KOLEKTIV A. *Jenkins: Jenkins* [online]. [cit. 2014-4-20]. Dostupné z: http://jenkins-ci.org/

SEZNAM OBRÁZKŮ

Obrázek 1 – Reprezentace black box systému 2
Obrázek 2 – Reprezentace white box systému 3
Obrázek 3 – reprezentace gray box systému 3
Obrázek 4 – jednotlivé fáze vývoje software z pohledu testování 4
Obrázek 5 – pyramida postupné posloupnosti testů převzato z (6) 4
Obrázek 6 – pohled na testovací proces v širším záběru převzato z (7) 6
Obrázek 7 – ilustrace SOA architektury převzato z (9) 8
Obrázek 8 – logo programovacího jazyka Java převzato z (11) 10
Obrázek 9 – ukázka Java kódu převzato z (11) 11
Obrázek 10 – ukázka zdrojového kódu s užitím Selenium WebDriver 13
Obrázek 11 – přihlášení na www.vutbr.cz 14
Obrázek 12 – příkaz pro spuštění Selenium Gridu v módu serveru (hubu) 15
Obrázek 13 – spuštění Selenium Gridu v módu nodu na vzdáleném stroji (nodu) 15
Obrázek 14 – grafické znázornění možného využití Selenium Gridu převzato z (13) 16
Obrázek 15 – znázornění míry abstrakce při užití page objectu převzato z (15) 17
Obrázek 16 – výřez stránky www.vutbr.cz/login/ pro účely vytvoření page objectu 18
Obrázek 17 – implementace rozhraní page objectu pro VUT přihlašovací stránku 18
Obrázek 18 – implementace funkčnosti page objektu 19
Obrázek 19 – testovací scénář vytvořený s využitím page objectu 20
Obrázek 20 – ukázka hlavního okna vývojového prostředí Eclipse 22
Obrázek 21 – ukázka konfiguračního souboru Mavenu 23
Obrázek 22 – POM soubor k VUT Core projektu 24
Obrázek 23 – POM soubor k VUT Test projektu 25
Obrázek 24 – ukázka třídy testu s využitím TestNG 27
Obrázek 25 – Obsah třídy „SeleniumTestCase“ 28
Obrázek 26 – Závislost mezi třídami bez injekce závislostí 29
Obrázek 27 – třída továrny oddělující závislost člověka a míče 30
Obrázek 28 – třída a rozhraní 30
Obrázek 29 – Třída modulu frameworku Google Guice 31
Obrázek 30 – Anotace vytvářející objektově relační mapování pro Hibernate 33
Obrázek 31 – Konfigurační soubor pro Hibernate 34
Obrázek 32 – Návod na vytvoření session-factory pro Google Guice 36
Obrázek 33 – Získání objektu card z databáze pomoci frameworku Hibernate 36

Obrázek 34 – Ukázka vstupního bodu tzv. pointcutu ... 37
Obrázek 35 – Around advice v rámci AspectJ ... 38
Obrázek 36 – vytvoření tabulky v MySQL databázi ... 39
Obrázek 37 – letmý pohled na dělení projektu a jmennou konveci ... 41
Obrázek 38 – výřez navigačního sloupce IS VUT ... 42
Obrázek 39 – statická pomocná třída pro navigaci na první stránku ... 43
Obrázek 40 – Samostatná třída testu s aplikovanými pravidly ... 45
Obrázek 41 – ukázka javadoc komentáře zobrazeného IDE studiem Eclipse ... 46
Obrázek 42 – ilustrace fluent interface ... 46
Obrázek 43 – Spuštění testu skrze plugin pro IDE Eclipse ... 47
Obrázek 44 – Ukázka suity pro TestNG ... 47
Obrázek 45 – Nastavení paralelního spuštění v rámci suity ... 48
Obrázek 46 – Ukázka vytvoření závislosti mezi testy ... 49
Obrázek 47 – rozdělení testovacích metod do tříd převzato z (18) ... 49
Obrázek 48 – Ukázka omezení spouštěných testovacích metod na základě definované skupiny testů ... 50
Obrázek 49 – Ukázka vzniklého reportu po spuštění TestNG testu ... 51
Obrázek 50 – Třída popisující entitu report ... 53
Obrázek 51 – Třída popisující entitu reportovacího kroku ... 54
Obrázek 52 – Vytvořená anotace logged pro účely reportingu ... 54
Obrázek 53 – Nejvíce nadřazená entita reportingu je iterace ... 55
Obrázek 54 – ukázka definice metody uvnitř PO s aplikací anotace @Logged ... 55
Obrázek 55 – ukázka vytvořeného vstupního bodu pro framework AspectJ ... 56
Obrázek 56 – vstupní bod pro odchycení vytvořené anotace @Logged ... 57
Obrázek 57 – část „Advice“ funkčnosti frameworku AspectJ ... 57
Obrázek 58 – částečná ukázka refaktorované funkce pro vytvoření reportovacího kroku ... 58
Obrázek 59 – parametry pro JVM nutné pro spuštění operace LTW frameworku AspectJ ... 59
Obrázek 60 – Konfigurační soubor pro AspectJ framework ... 59
Obrázek 61 – SQL přikaz na vytvoření tabulky Report ... 60
Obrázek 62 – SQL příkaz pro vytvoření tabulky reportingového kroku ... 61
Obrázek 63 – SQL příkaz k vytvoření tabulky iterace ... 62
Obrázek 64 – SQL dotaz pro zobrazení reportu s jeho kroky ... 62
Obrázek 65 – Ukázka části výstupu výše uvedeného SQL dorazu ... 62
Obrázek 66 – definice parametrů pro ovládání chování frameworku ... 63
Obrázek 67 – ukázka předání parametru frameworku v rámci IDE Eclipse ... 64

Obrázek 68 – funkce pro získání Selenium driveru s přihlédnutím k možnosti volit mezi dvěma prohlížeči 65
Obrázek 69 – vzhled metody pro navrácení hodnoty předaného parametru 66
Obrázek 70 – ukázka webového rozhraní serveru s jediným připojeným nodem 67
Obrázek 71 – částečná ukázka funkce zodpovědné za vytvoření tzv. „Remote driveru“ => driver komunikující se Selenium Grid serverem 68
Obrázek 72 – okno reprezentující běžící Selenium Grid v režimu „Server“ 69
Obrázek 73 – okno reprezentující běžící Selenium Grid v režimu „NODu“ 69
Obrázek 74 – ukázka spustitelných jobů ve webovém rozhraní nástroje jenkins dostupné z https://ci.jenkins-ci.org/ 71
Obrázek 75 – částečný výpis definice suity pro spuštění všech scénářů 74

SEZNAM TABULEK

Tabulka 1 – porovnání délky trvání testu IE vs FF 74
Tabulka 2 – přehled kroků pro IE v rámci VUTCommonInformation scénáře 75
Tabulka 3 - přehled kroků pro FF v rámci VUTCommonInformation scénáře 75
Tabulka 4 - přehled kroků pro IE v rámci VUTMessages scénáře 76
Tabulka 5 - přehled kroků pro FF v rámci VUTMessages scénáře.. 76
Tabulka 6 - přehled kroků pro IE v rámci VUTNavigation scénáře 77
Tabulka 7 - přehled kroků pro FF v rámci VUTNavigation scénáře 77

DODATEK A

Soupis potřebného software pro korektní funkci frameworku

- Mozzila Firefox ve verzi 24.xx ESR (Verze s prodlouženou podporou)
- Databáze MySql od verze 5.6.xx
- JDK od verze 7uxx
- Maven od verze 3.x.x
- Internet Explorer od verze 9 včetně stažení serveru pro jeho ovládání ze stránek projektu Selenia
- Vývojové studio Eclipse od verze 4.3 (Kepler) včetně instalace doplňku pro TestNG ze stránek projektu

Potřebný software bude přibalen na dodávaném DVD společně s prací ve verzích dostupných k aktuálnímu datu 12.5.2014.