Softwarový systém pro testování webových stránek

A software system for testing of web pages

Bc. Pavlína Vodičková

Diplomová práce
2008

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

ABSTRAKT

Diplomová práce se zabývá testováním webových aplikací. Snaží se vysvětlit některé pojmy z oblasti testování a popsat způsoby testování, jež se v dnešní době nejčastěji používají. Nejprve popisuje testování obecně, pak se zaměřuje na specifika webových aplikací a následně uvádí několik příkladů testování bezpečnosti.

Diplomová práce se nezabývá nastavením platforem na kterých webové aplikace mohou běžet, to znamená například nastavením operačních systémů, databázových serverů, serverů IIS, Apache apod.

Cílem bylo podat určitý přehled o současné situaci testování webových aplikací a na závěr zpracovat jednoduchý program, která by pomohl zjednodušit otestování vstupů.

Klíčová slova:

testování, aplikace, systém, software, webové stránky, chyba, test, ruční testování, automatické testování, nízkoúrovňové testování, vysokoúrovňové testování, testování komponent, integrační testování, zátěžové testování, funkční požadavky, ne-funkční požadavky, testování použitelnosti, testování přístupnosti, akceptační testování, statické testy, dynamické testy, funkční testy, ne-funkční testy, white-box testy, black-box testy, progresní testy, regresní testy, testy splněním, testy selháním, verze produktu, alpha verze, beta verze, GA release, hotfix, service pack, featura, bug, milestone, enhancement, vodopádový model, spirálový model, validní kód, optimalizace, katalogové vyhledávání, fulltextové vyhledávání, pravidla použitelnosti, pravidla přístupnosti, Selenium, SQL Injection, krádež relace, cookie, skriptování stránek, vyhledávací portál, útok hrubou silou, slovníkový útok, útok typu sociálního inženýrství, šifrování hesel.

ABSTRACT

This diploma thesis deals with web sites testing. It tries to explain few definitions regarding the testing field and also tries to describe the most used ways of testing. At first, it describes testing in general, then it points some web application specifics and in the end, it shows few examples of security testing.

It does not deal with platform settings where a web application can run such as operating system settings, database server settings, settings of IIS or Apache servers and the like.

The target of this thesis was to bring an overview of the current way of web application testing and to create a simply program that would be able to simplify testing of an application inputs.

Keywords:

testing, application, system, software, web page, failure, test, manual testing, automated testing, low-level testing, high-level testing, unit testing, integration testing, stress testing, functional requirements, non-functional requirements, usability testing, accessibility testing, acceptation testing static tests, dynamic tests, functional tests, non-functional tests, white-box tests, black-box tests, progressive tests, regressive tests, compliance tests, failure tests, product version, alpha version, beta version, GA release, hotfix, Service Pack, featura, bug, milestone, enhancement, waterfall model, spiral model, valid code, optimalization, catalog search, fulltext search, usability rules, accessibility rules, Selenium, SQL Injection, Session Hijacking, cookie, Cross-site Scripting, search portal, brutte force attack, vocabulary attack, social engineering attack, password encryption.

Ráda bych poděkovala svým kolegům z GMC Software Technology, s.r.o., zejména kolegům Josefu Štefanovi a Tomáši Tintěrovi za poskytnutí námětů a připomínek, které mi velmi pomohly.

Prohlašuji, že jsem na diplomové práci pracovala samostatně a použitou literaturu jsem citovala. V případě publikace výsledků, je-li to uvolněno na základě licenční smlouvy, budu uvedena jako spoluautor.

Ve Zlíně …….

Podpis diplomanta

OBSAH

ÚVOD

Cílem práce je popsat současný stav a možnosti testování webových stránek. Je rozdělená na část praktickou a teoretickou. Teoretická část se dále dělí na tři části, praktická na dvě.

První teoretická část se snaží vysvětlit nezbytnost testování a popsat jeho základy. Zde jsou uvedeny principy testování, které platí již řadu let a vzhledem ke své důležitosti jsou uváděny téměř v každé publikaci, která se testováním zabývá. Dále jsou zde uvedeny některé charakteristiky testování, které by, podle mého názoru, mohly pomoci lépe pochopit podstatu testování. Následuje vysvětlení jednotlivých úrovní testování, typů testů a procesu testování, které se v dnešní době běžně používají. V kapitolách *Stavy verze produktu* a *Rozpis prací – milestones* jsou uvedeny a vysvětleny pojmy, se kterými se v oblasti testování velmi často setkáváme. V následující kapitole *Modely vývoje a testování* jsou zmíněny dva nejčastější modely vývoje ve vztahu k testování. V závěru první teoretické části je poukázáno na možnost dodržování dvou mezinárodně uznávaných standardů, a to norem ISO/IEC řady 9126 a norem ISO/IEC řady 14598.

Druhá teoretická část se zabývá odlišnostmi webových stránek od ostatních aplikací a možnostmi otestování těchto odlišností. Popisuje také dva způsoby testování, a to ruční testování a automatické testování. V závěru poukazuje na jazyk XPath, jehož znalost je při testování webových stránek často potřebná.

Třetí teoretická část je zaměřena na otestování webové aplikace z hlediska bezpečnosti. Jsou zde uvedeny nejčastější a nejznámější útoky na webové aplikace a možnosti jejich předcházení.

Nutno upozornit na skutečnost, že ani jedna část není popsána vyčerpávajícím způsobem. Tato práce se snaží objasnit základy a poukázat na oblasti, které s testováním souvisí a jež jsou v dnešní době rozpracovány do takových podrobností, které zdaleka přesahují rozměr této práce.

V první praktické části je popsán návrh na vylepšení testování. Tento návrh vyšel z potřeby zjednodušení otestování vstupů do webové aplikace. V druhé praktické části je pak tento návrh realizován.

I. TEORETICKÁ ČÁST

1 POPIS SOUČASNÉHO STAVU A VYPRACOVÁNÍ REŠERŠE K DANÉ TÉMATICE

1.1 Nezbytnost testování

V dnešní době je vyvíjeno velmi mnoho druhů softwaru, které mohou sloužit k různým účelům. Také ceny se liší. Některý software může být použit zcela zdarma, jiný může být použit zdarma pouze za určitých podmínek, za jiné je potřeba zaplatit poměrně vysoké částky. Všechny ale mají jedno společné – uspokojit potřeby uživatele co nejlépe. Chyby v softwaru jsou proto velice nežádoucí, obzvláště ty, které v konečném důsledku mohou způsobit velké ztráty peněz nebo způsobit neštěstí či dokonce újmy na životech. Také riziko poškození pověsti výrobce softwaru je vážným důvodem pro otestování produktu dříve, než se dostane k zákazníkovi.

Důvodů ke vzniku chyb může být opravdu mnoho. Lidé jsou omylní, což způsobuje zanesení chyb do kódu, případně do dokumentace. Také přechod na nové technologie může být příčinou vzniku nepříjemných chyb. Někdy se chyby vyskytnou při interakci systémů, jindy jsou způsobeny podmínkami prostředí. Negativní vliv zde má radiace, magnetismus, elektrická pole, znečištění, apod. Toto všechno pak způsobuje změny v hardwaru a následně nesprávný běh programu.

Cílem testování je tedy ve většině případů včasné odhalení chyb, jejich identifikování, lokalizace a odstranění.

1.1 Principy testování[1]

Existuje několik principů testování, které jsou zásadní a je potřeba je mít stále na paměti. Jedná se o to, že:

- *testování pouze zjišťuje přítomnost chyb, nikdy ale nemůže zaručit jejich neexistenci.* To znamená, že i když po provedeném testování není nalezena žádná chyba, tak se v

1 Zpracováno podle [1] a [2]

systému chyby vyskytovat mohou. Jedná se o tzv. „nenalezené chyby“. Jinak řečeno, testování může pouze pomoci snížit počet chyb v systému.

- *Kompletní otestování není možné.* Snad jen s výjimkou hodně jednoduchých aplikací. Většinu aplikací není možné kompletně otestovat z důvodu příliš velkého počtu vstupů nebo příliš velkého počtu výstupů anebo příliš velkého počtu možných cest, které vedou skrz aplikaci. Tento problém se řeší pečlivým vybráním testovacích případů a testuje se s rizikem, že některé chyby zůstanou neobjeveny.
- *Je žádoucí časné testování.* Čím dříve se na chybu přijde, tím menší jsou náklady na její opravu. Pokud se na chybu přijde pozdě, tak se také může stát, že její oprava už ani nebude možná, protože by byla příliš riskantní (s opravou jedné chyby lze vždy do kódu zanést chybu další)
- *Sdružování chyb.* Chyby se často objevují ve shlucích, což je dáno tím, že pokud je část kódu napsaná špatně, ovlivní to více částí aplikace. Například pokud v objektovém programování napíšeme špatně jednu třídu pak další třídy, které využívají její metody, budou s velkou pravděpodobností také dávat špatné výsledky.
- *Paradox pesticidů.* Paradox pesticidů se projeví tehdy, když napíšeme test a budeme ho pravidelně spouštět. Zpočátku bude test nalézat chyby, ty se však budou postupně opravovat, takže po čase bude test procházet bezchybně a nebude nic nového přinášet. V tomto okamžiku je dobré napsat jiný test, nebo stávající modifikovat.
- *Testování je závislé na kontextu.* Způsob testování záleží na zamýšleném využití aplikace. Například aplikace pro internetové bankovnictví se bude testovat jinak než aplikace pro přehrávání hudby.
- *Absence-of-errors fallacy.* Tím je myšleno, že i kdybychom našli všechny chyby a opravili je tak, že by aplikace běžela za všech podmínek bezvadně, neznamená to, že aplikace je použitelná a splňuje to, co od ní uživatel očekával.

1.2 Charakteristiky testování

1.2.1 Co testování obnáší

Testování obnáší vykonání mnoha činností. Začíná se obvykle *plánováním*. Nejprve je potřeba zjistit co všechno se bude testovat a jakým způsobem a vytvořit určitý plán. S tím

souvisí i *výběr podmínek* pro testování. Je často nutné provést ten samý test za odlišných podmínek. Pak navrhneme tzv. *testovací případy*. Testovací případ je test, který reprezentuje určitou skupinu podobných testů. Vytváříme je proto, abychom zkrátili dobu testování na únosnou míru a abychom nemuseli testovat kompletně vše, což obvykle ani není možné. Následně musíme *vytvořit test*, pokud to lze. Je často výhodné vytvořit skript, který provede samotné otestování. Pokud nelze vytvořit skript, testuje se ručně. Následně se *zjistí výsledky testu,* provede se *kontrola,* zda se shodují s očekávanými výsledky a pokud se liší, *zjistí se a lokalizují chyby*, které je následně třeba *nareportovat* programátorům. Nakonec se také *zkontroluje dokumentace*, vyhodnotí se testování, sepíše *závěr a ukončí testování*.

1.2.2 Cíle testování

Hlavním cílem zpravidla bývá včasné odhalení chyb, jejich identifikování, lokalizace a odstranění. Přičemž pro odhalení chyb se používají různé metody. Nejlépe je postupovat tak, že nejprve aplikaci testujeme za běžných podmínek. Pokud se žádné chyby neprojeví, necháme ji běžet za krajních podmínek. Například, jestliže testujeme vstupy, pak zadáme nejmenší a největší číslo, které je povolené. Pokud aplikace reaguje bezchybně, můžeme zkusit zadat hodnoty, které nejsou očekávané. Zadáme třeba číslo větší než je povolené číslo nebo desetinné číslo nebo rovnou text. Lze ale ovšem také spouštět aplikaci při malém místu na disku nebo s malou pamětí. To záleží na tom, z jakého hlediska aplikaci právě testujeme.

Ovšem někdy může být cílem jen ujištění, že systém pracuje tak, jak by měl a že splňuje všechny požadavky. V tomto případě pak není prioritou vyvolání chybových stavů – to už bylo pravděpodobně otestováno v některém z předešlých kroků. Tento cíl máme obvykle v případě, že byla do systému přidaná nová vlastnost a my se potřebujeme ujistit, že její přidání nežádoucím způsobem neovlivnilo některé další, již dříve otestované funkce či vlastnosti. K dalšímu otestování těchto starých featur obvykle tento cíl postačuje.

V určitých případech může být také cílem testování jen zjištění kvality systému bez úmyslu opravení chyb. Systém může být kvalitní třeba v tom smyslu, že je spolehlivý, přesný, přenositelný, použitelný nebo udržovatelný. Testy mohou pomoci zjistit míru kvality, ze které pak můžeme vycházet například při porovnávání dvou či více systémů.

1.2.3 Psychologie testování

Na proces testování může být pohlíženo různě. Testování samo o sobě přináší cenné informace. A to zejména informace o nalezených chybách. Chybu je pak možné okamžitě opravit a nedostane se tak ke koncovému uživateli, kde by mohla napáchat obrovské škody. Ovšem informace o chybách nejsou nikterak příjemné a také většina chyb se objeví v případech, kdy software vystavujeme krajním podmínkám, takže testování může být někdy chápáno jako destruktivní činnost s cílem zničit vše, co programátoři pracně vytvořili. V krajním případě si lze testování vyložit jako útok proti produktu a jeho tvůrcům. Proto velice záleží na přístupu testera a způsobu reportování chyb. Obecně lze říct, že platí zásada, že tester by měl spíš volit cestu spolupráce s programátorem, ne se snažit s ním bojovat. Při reportování chyby by měl věcně popsat problém, být neutrální a zdržet se jakékoliv kritiky produktu či dokonce programátora.

1.3 Úrovně testování

Testuje se v několika úrovních, obvykle postupujeme od testování jednodušších celků směrem ke složitějším. Jednotlivé úrovně testování se někdy pro lepší přehlednost dělí do dvou skupin, a to na *nízkoúrovňové testování* a *vysokoúrovňové testování*.

1.3.1 Nízkoúrovňové testování

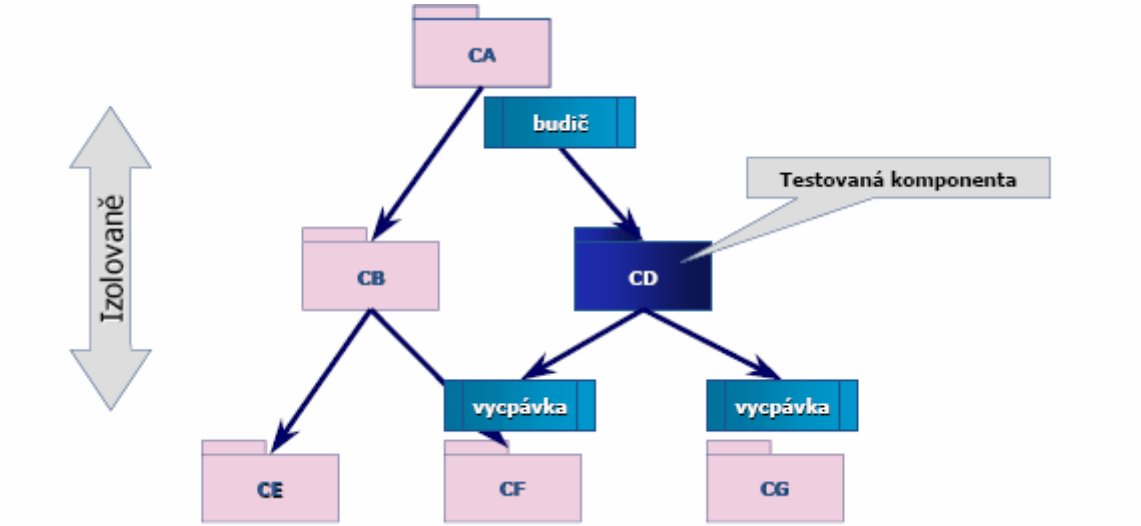
Testování komponent

Jako komponentu můžeme použít jednotlivý modul, program, objekt, třídu apod. Podmínka je ta, že musí být možné je testovat odděleně od ostatních částí systému. Při testování komponent tedy nepoužíváme ostatní komponenty systému, ale jako jejich náhradu použijeme různé ověřené simulátory a to drivery nebo stuby. V literatuře se lze také setkat s pojmy *budiče* či *ovladače* (tím jsou myšleny drivery) a s pojmy *vycpávky* nebo *makety* (což je označení pro stuby). Drivery používáme pro generování volání jiných komponent nebo modulů, stuby suplují funkcionalitu připojené komponenty nebo modulu.

Při testování komponent testujeme funkcionalitu i ne-funkční požadavky, používáme white-box testovací techniku.

Typické testování se provádí s pomocí vývojového prostředí, má přístup ke zdrojovému kódu testované komponenty a často ho provádí programátor, který komponentu vytvořil. Nalezené chyby se obvykle nezaznamenávají, rovnou se opraví.

Do skupiny testování komponent také patří programovací způsob, kdy se test provede ještě před vytvořením komponenty. Jinak řečeno, ke psaní vlastní komponenty se přikročí až když je test hotov, přičemž ta je hotova v okamžiku, kdy test už nenalézá žádné chyby. Tomuto způsobu se říká *programování řízené testy*.



Obrázek 1: Vztah mezi drivery, stuby a testovanou komponentou

Integrační testování

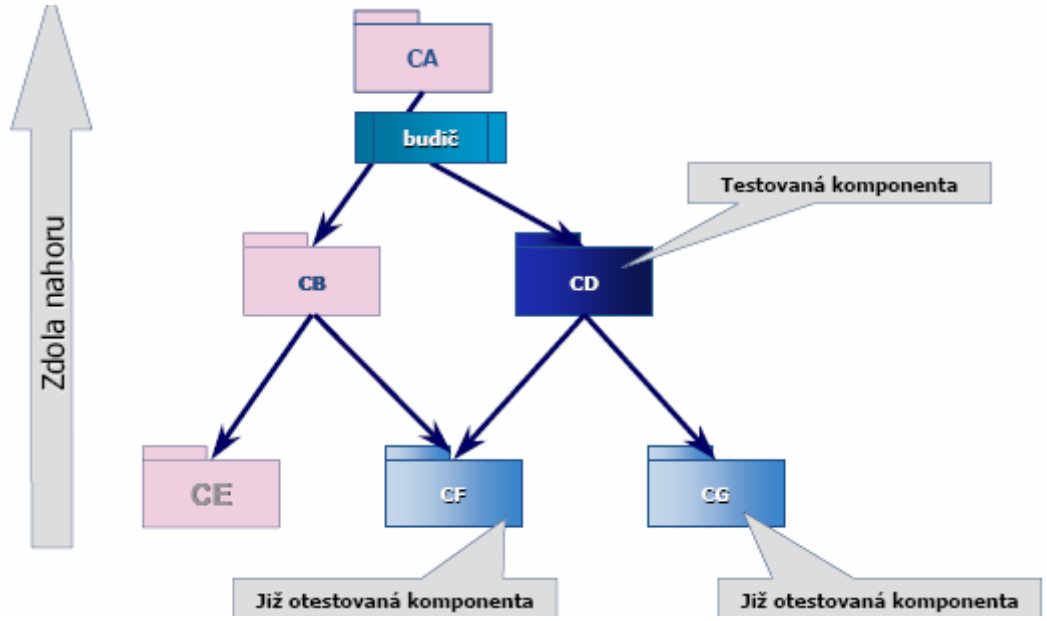
K integračnímu testování přikročíme v případě, že již máme otestované jednotlivé komponenty. V tomto kroku testujeme jednak interakci jednotlivých komponent mezi sebou, ale také interakci jednotlivých komponent s operačním systémem, souborovým systémem, hardware nebo interakci mezi rozhraními různých systémů. Nejprve začneme testovat integraci dvou modulů, pak postupně přidáváme další, až se nakonec dostaneme k testování systému jako celku, což už patří do skupiny vysokoúrovňového testování.

Při integračním testování můžeme postupovat dvěma způsoby. A to tak, že můžeme provádět testování *zdola-nahoru* nebo testování *shora-dolů*.

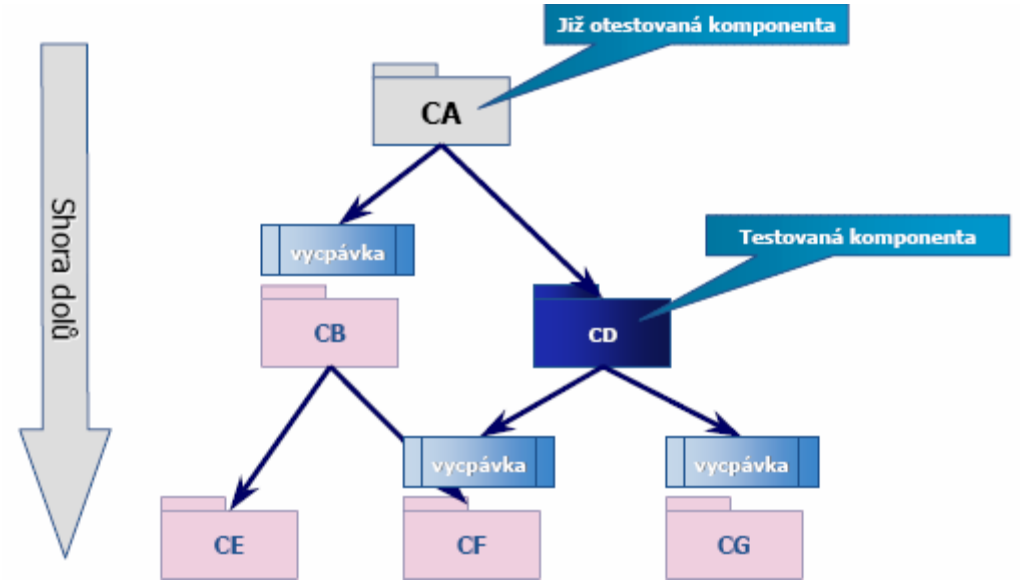
Při testování metodou *zdola-nahoru* testujeme interakci komponent bez využití stubů – namísto stubu použijeme skutečnou, již otestovanou komponentu a ověřujeme spolupráci s aktuální, připojenou komponentou.

Při testování metodou *shora-dolů* testujeme interakci komponent bez využití driverů – namísto driveru použijeme skutečnou komponentu, kterou jsme již prve otestovali a ověříme spolupráci s ní.

Obě metody jsou zobrazeny na následujících dvou obrázcích[2]:



Obrázek 2: Testování metodou zdola-nahoru



Obrázek 3: Testování metodou shora-dolů

2 Obrázky jsou převzaty z [3]

1.3.2 Vysokoúrovňové testování

Systémové testování

Celý systém testujeme jako konečný produkt, přičemž bychom měli věnovat velkou pozornost testovacímu prostředí. Mělo by být co nejvíce podobné prostředí, ve kterém bude systém běžet u zákazníka. Pro systémové testování je typické použití black-box testovací techniky a testuje se chování systému jako celku. Testujeme ne-funkční požadavky.

Do systémového testování zahrnujeme například:

- testování interakce s operačním systémem
- testování interakce se systémovými zdroji
- testování objemu
- zátěžové testování
- testování bezpečnosti
- testování spotřeby zdrojů
- testování konfigurace
- testování kompatibility
- testování instalace (čisté instalace i update)
- testování zálohování/obnovy
- testování obnovení systému po havárii

Testování použitelnosti

Použitelnost aplikace je obvykle to první, čeho si uživatel všimne. Do testování použitelnosti zahrnujeme zhodnocení celkového dojmu aplikace na uživatele – jak je aplikace graficky zpracovaná, jak je rychlá, zda je uživatel schopen se v ní rychle zorientovat, apod. Testování použitelnosti je velmi subjektivní záležitost. Pro webové stránky existují určitá pravidla, jejichž dodržování usnadní práci s webem většině uživatelů.

Testování přístupnosti

Pro handicapované uživatele existují tzv. pravidla přístupnosti, jež jsou světová nebo lokální. V České republice jsou pravidla přístupnosti zakotvená v novele Zákona č. 365/2000 Sb. o informačních systémech veřejné správy, provedenou zákonem č. 81/2006 Sb. Jsou závazná pro všechny weby veřejné správy od 1. března 2008.

Akceptační testování

Prioritou je otestování funkčních požadavků. Měli bychom mít k dispozici seznam požadavků zákazníka, ze kterého získáme informace o tom, co je chybou a co chybou není. Testujeme také jednoduchost použití, ovládání systému z hlediska uživatele, apod.

1.4 Typy testů

V průběhu celého procesu testování, používáme několik typů testů. Vždy se snažíme vybrat nejvhodnější typ testu a to podle úrovně testování, cíle testování a typu vlastnosti nebo funkce aplikace, který potřebujeme otestovat. Testy můžeme dělit z několika hledisek.

1.4.1 Statické a dynamické testy

Statické testy použijeme v případě, kdy testujeme tu část aplikace, která nevyžaduje běh aplikace. Například je můžeme použít pro kontrolu dokumentace nebo pro statickou analýzu programového kódu. Statické testy si můžeme představit jako určitou „revizi“.

Dynamické testy se používají častěji, jedná se o testování, které se provádí za běhu aplikace. Používáme je ve všech úrovních testování.

1.4.2 Funkční a ne-funkční testy

Funkční testy použijeme zejména v případech, kdy testujeme funkce aplikace. Používají se ve vysokoúrovňovém testování, nejčastěji při akceptačním testování.

Ne-funkční testy použijeme také ve vysokoúrovňovém testování, ale jsou učeny spíše pro systémové testování.

1.4.3 White-box a black-box testy

White-box testy jsou testy, při nichž máme přístup k programovému kódu. Výhodou tohoto testování je, že například vidíme, jak jsou různé vstupy zpracovávány a podle toho můžeme přizpůsobit testování. Používáme zejména v nízkoúrovňovém testování.

V případě black-box testů nemáme přístup ke zdrojovému kódu, což může být určitá nevýhoda v tom smyslu, že některé části testů pak mohou být duplicitní. Na druhé straně je však tester neovlivněný kódem a při testování se pak na aplikaci dívá spíše očima uživatele. Tyto testy se hodí spíše pro vysokoúrovňové testování, zejména pro testování použitelnosti.

1.4.4 Progresní testy a regresní testy

Progresní testy používáme při testování nových funkcí či vlastností aplikace. Používáme je v každé úrovni testování.

Regresní testy používáme při opětovném testování funkcí i vlastností aplikace. Používáme je proto, abychom se ujistili, že provedení změn, popřípadě implementace nových featur v aplikaci negativně neovlivnilo původní vlastnosti či funkce. Regresní testy používáme také v každé úrovni testování a je velmi vhodné je automatizovat, pokud to lze.

1.4.5 Testy splněním a testy selháním

U testů splněním ověřujeme, že získaný výstup se shoduje s výstupem očekávaným. Zadáváme pouze vstupy, které by aplikace měla zvládnout.

U testů selháním se ujišťujeme, že se výstup neshoduje s nežádoucí hodnotou. Zadáváme nestandardní vstupy s cílem aplikaci „shodit“.

Testy splněním a selháním se mohou prolínat, může se stát, že jeden test bude zároveň testem splněním a zároveň testem selháním.

1.5 Proces testování

Testování není nikterak jednoduchá záležitost, jak by se mohlo na první pohled zdát. Při testování vlastností a funkcí aplikace, které se navíc jednotlivě testují v různých podmínkách, je také třeba otestovat všechny další související části, jako například chybové zprávy, soubory nápovědy, reklamy, ukázky, příklady, informace o podpoře produktu,

apod. Aby bylo možné vše postupně otestovat, je potřeba provádět testování podle určitých pravidel.

1.5.1 Plánování a kontrola

Plánování je první činností při zahájení testování. Tester dostane seznam implementovaných featur a seznam opravených bugů. U každé položky je potřeba určit cíl testování. Jak bylo zmíněno výše, cíle se mohou lišit. Jestliže testujeme nově implementovanou featuru, pak by cílem mělo být nalezení co největšího počtu chyb. Zároveň ale zvážíme, které stávající vlastnosti či funkce produktu by mohly být nově implementovanou featurou ovlivněny. U těchto funkcí pak se pak obvykle provede jednodušší test, který pouze zkontroluje, zda stále pracují správně. Dále je také u každé položky potřeba zvážit, zda by bylo vhodné vytvořit automatický test pro regresní testování.

Kontrola, na rozdíl od plánování, probíhá po celou dobu testování. Je potřeba neustále kontrolovat zda testování probíhá podle plánu, či zda v některém směru odbočilo.

1.5.2 Analýza a design

V těchto dvou aktivitách se sestaví testy. Sestavování testů zahrnuje tyto činnosti:

- *Zjištění dostatečného množství informací, které se týkají jednotlivých testovaných problémů.* Může se jednat o informace o hraničních podmínkách, v nichž testovaná funkce musí umět pracovat. Nebo o potřebné konfiguraci softwaru a hardwaru. Nebo o typech souborů, se kterými musí umět pracovat. Také o chování funkce za určitých podmínek, o kritických hlášeních v případě že funkce selže, o údaje o minimální rychlosti zpracování určitého úkolu, apod. V ideálním případě získáme informace z produktové specifikace.
- *Navržení testů.* V případě, že máme všechny informace, pak navrhneme jednotlivé testovací případy. Následně u každého testovacího případu rozhodneme o použití určitého typu testu. Podle povahy testovaného problému určíme, zda bude statický nebo dynamický, white-box nebo black-box, apod. Pokud budeme vytvářet automatický test, rozhodneme se pro konkrétní testovací nástroj. Dále navrhneme postup kroků, které test bude provádět. Nakonec test s použitím testovacího nástroje napíšeme.

- *Nastavení testovacího prostředí.* V tomto kroku nainstalujeme potřebný software či hardware a patřičně nakonfigurujeme. Dále si připravíme potřebná testovací data. Může se jednat o soubory různých typů, velikostí, kódování apod.

1.5.3 Implementace a spouštění

Jednotlivé testy pak seřadíme podle priorit či potřebných nastavení. Pokud neprovádíme testování ručně, ale spouštíme pomocí testovacího nástroje, vytvoříme Test Suit a nastavíme logování. Testy spustíme.

1.5.4 Vyhodnocení ukončení testů

Po ukončení testů porovnáme výsledky získané s výsledky očekávanými. Jestliže zjistíme rozdíl nebo pokud během testování došlo k jiné chybě, zkontrolujeme, zda se tak stalo chybou v testu anebo chybou v aplikaci. K nalezení chyb použijeme log testovacího nástroje, a pokud testovaná aplikace také logovala, použijeme i log její. Jestliže nalezneme chybu v aplikaci, je nutné ji nareportovat programátorům. Při reportování bugu vždy uvádíme:

- použitý software a číslo jejich verzí,
- přesný postup, jak chybu zreprodukovat (tzn. jak znovu vyvolat chybový stav),
- popis chybového stavu, tedy popis nežádoucího stavu, který jsme získali,
- stav, který tester očekával, tedy stav aplikace, který měl nastat, ale nenastal,
- další informace, například výpis z logu aplikace, backup databáze, screenshot obrazovky se zachycením chybového stavu, popřípadě skript, který dovede aplikaci do chybového stavu, apod.

Uvedený seznam je základní ale ne konečný, záleží na tvůrci softwaru, co všechno bude při reportování chyb vyžadovat.

1.5.5 Uzavření testování

Probíhá například při těchto příležitostech:

- testování je ukončeno nebo zrušeno
- konec releasu systému

- byl dosažen milestone

Uzavření testování spočívá v tom, že se sepíše zpráva o průběhu a výsledcích testování. Z výsledku testování se pak vychází při rozhodování, jak s produktem dále naložit. Pokud například testujeme alpha verzi a během testování se v ní nenalezne příliš mnoho závažných chyb (limity si nastavují výrobci sami), pak z této alpha verze vznikne beta verze a odešle se k otestování vybraným uživatelům. Nalezené chyby pak budou opraveny v dalších verzích. Jestliže se ale počet chyb do limitu nevejde, pak se verze musí vrátit zpět k programátorům, kteří se zaměří na odstranění nalezených závad.

1.6 Stavy verze produktu

Postupně, jak je produkt vyvíjen a testován, prochází několika stavy. Názvy stavů jsou většinou známé, ale jejich význam či zacházení s nimi se může u jednotlivých výrobců trochu lišit. Zde uvádím nejobvyklejší význam názvů stavů.

1.6.1 Alpha

Produkt je ve stavu alpha v okamžiku, kdy je programátory dokončen a je předáván k otestování testerům. Tito testeři jsou obvykle zaměstnanci firmy. Produkt ve stavu alpha obsahuje mnoho chyb a je často nestabilní. Zákazníkům se obvykle neposkytuje.

1.6.2 Beta

Jakmile je produkt otestován firemními testery a nalezené chyby jsou opraveny, přejde do stavu beta. I ve stavu beta může produkt obsahovat mnoho chyb, ale obvykle už bývá použitelný. Produkt ve stavu beta se distribuuje vybraným uživatelům, kteří jej testují v prostředí, ve kterém bude produkt později používán.

1.6.3 GA release

GA release je výsledný, řádně otestovaný produkt, který může být prodáván a poskytován široké veřejnosti. Často mívá chyby, ale ty jsou uvedeny na *seznamu známých chyb*, se kterým se může zákazník před koupí seznámit. GA release může také bohužel obsahovat tzv. nenalezené chyby, to znamená chyby, na které se při testování nepřišlo. Jejich množství by ale mělo být minimální.

1.6.4 Hotfix

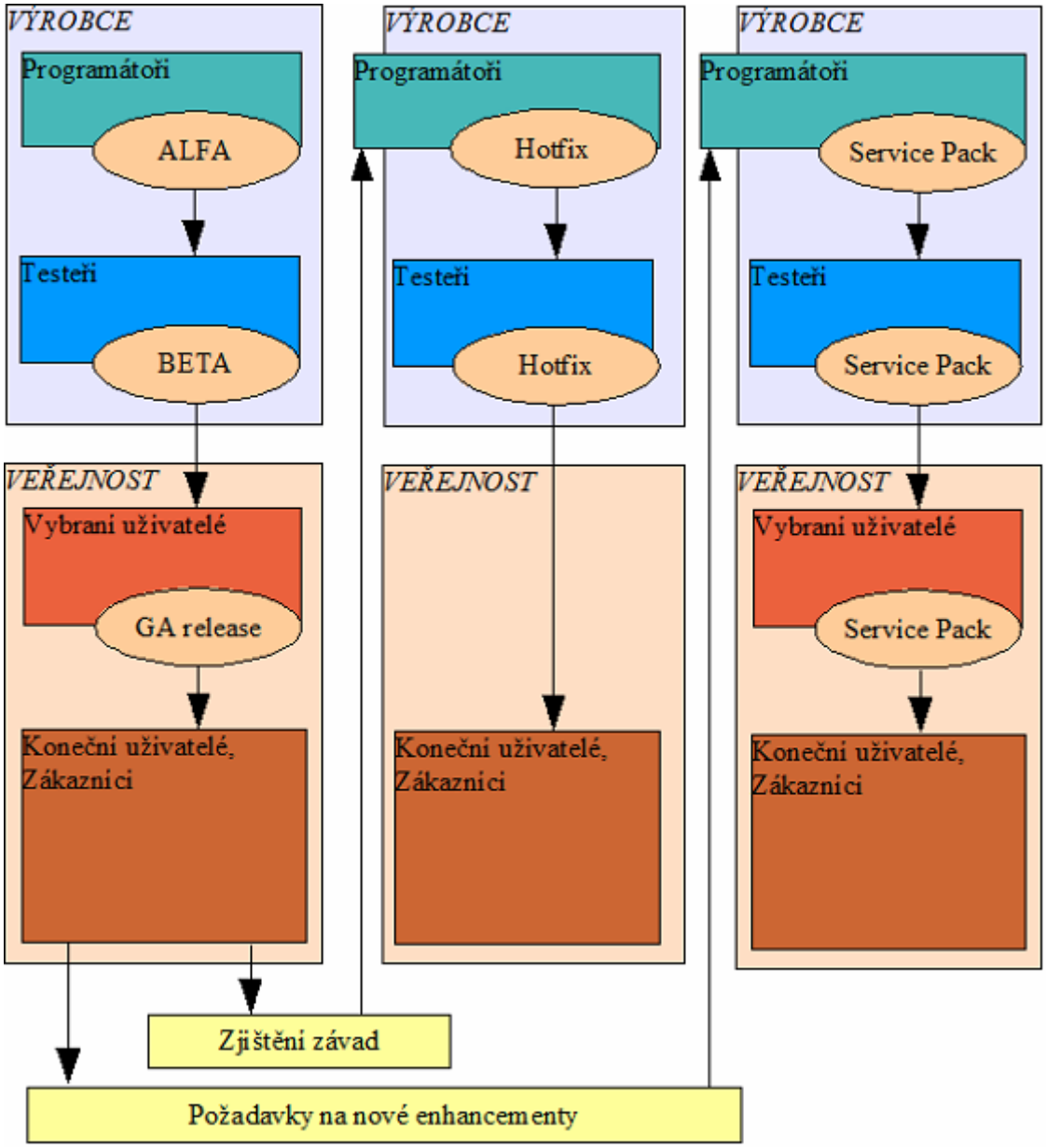
Hotfix je označením pro patch. Je to oprava chyby, obvykle zaměřená na konkrétní problém. Uživatel jej získá od výrobce a měl by si jej nainstalovat co nejdříve, zejména pokud se jedná o bezpečnostní problém. Hotfix neobsahuje nové *enhancementy*.

1.6.5 Service Pack

Service Pack obsahuje aktualizace, opravy a nové *enhancementy*. Často také obsahuje předchozí hotfixy. V tomto případě, jestliže si uživatel opomněl nainstalovat některý hotfix, pak bude tento nainstalován spolu s novým Service Packem.

Service Pack může mít charakter inkrementální nebo kumulativní. Inkrementální charakter znamená, že neobsahuje předchozí Service Packy. Kumulativní Service Pack je častější a předchozí Service Packy obsahuje. Tedy v případě, že si uživatel nainstaluje nejnovější Service Pack, nebude si muset již instalovat žádný z předchozích.

Na následujícím obrázku je znázorněna jedna z možností průběhu vytváření produktu. Výrobci sami rozhodují o tom, jak se bude postupovat, popř. jaké interní směrnice si zvolí. Testeři výrobce (alfa-testeři) zpravidla otestují vše, co opouští firmu, ale někteří výrobci například nedávají otestovat Service Packy uživatelům (beta-testerům).



Obrázek 4: Stavy verze produktu - příklad

1.7 Rozpis prací – milestones

Milestone je anglické slovo a v češtině znamená „milník“ či „patník“. Patníky jsou umístěny podél silnic a rozdělují dlouhý úsek na kratší úseky. Stejně je tomu tak i při vývoji softwaru. K tomu, aby byl software vyroben, obvykle potřebujeme dlouhý časový úsek, často celé roky. Tak dlouhý úsek je potřeba rozdělit na kratší. Pro oddělení těchto kratších úseků používáme pojem *milestone*. Každý milestone obvykle zahrnuje sekvenci aktivit, jejichž postupné plnění vede k dosažení cíle, který byl pro konkrétní milestone vytyčen.

Milestones mají výhody také další výhody:

- práce v rámci celého projektu lze lépe rozčlenit,
- podle milestone lze dobře poznat, v jakém stavu se projekt právě nachází,
- je možné průběžně kontrolovat odchylky od plánu a snažit se je redukovat co nejdříve,

Typický milestone zahrnuje několik kroků:

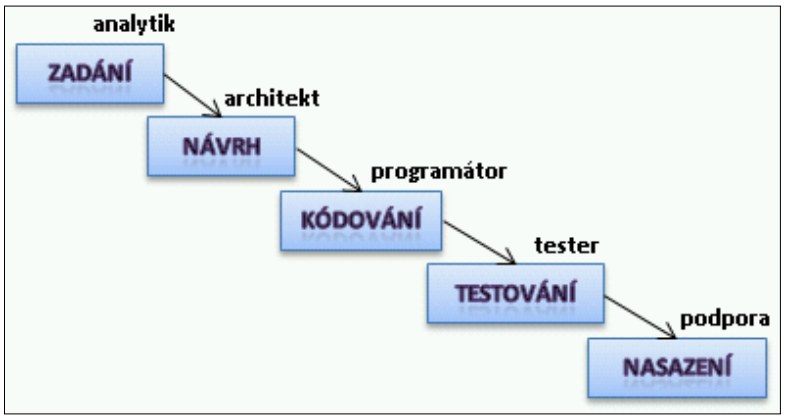
- nejprve se definují cíle milestone což jsou zpravidla nové featury, které budou přidány k vyvíjenému projektu. Také se mohou do milestone zahrnout známé bugy, které je třeba opravit,
- určí se, jak budou nové featury řešeny. To znamená výběr vhodných techologií a určení způsobu zpracování,
- následuje implementace featur, oprava bugů
- hotové featury a opravené bugy je potřeba řádně otestovat
- poté se provedou regresní testy, které ověří, zda implementace nových featur či oprava bugů neovlivnily jiné funkce či vlastnosti projektu
- provede se vyhodnocení jednotlivých kroků, sepíše závěr testování

1.8 Modely vývoje a testování

Nejpoužívanějšími modely jsou *Model vodopádu* a *Spirálový model. Spirálový model* je novější, řeší mnohé nedostatky vodopádového modelu a pro testery je výhodnější. Hodí se zejména pro větší projekty.

1.8.1 Vodopádový model

Vodopádový model vývoje softwaru se hodí spíše pro menší projekty, kde se neuvažuje o pozdějším zásahu do funkcionality projektu či změně jeho vlastností. Tento model dělí tvorbu aplikace do několika částí, přičemž každou z nich je nutno ukončit ještě předtím, než se postoupí do následující. Model je znázorněn na obrázku[3]



Obrázek 5: Příklad vodopádového modelu

Model je charakteristický tím, že jakmile se dostaneme do určité úrovně, není cesty zpět. Z toho plyne, že je nejprve potřeba důkladně překontrolovat případně otestovat všechny činnosti, které byly provedeny v aktuální fázi. Co se týče testování samotné aplikace, tak ta se provádí jednorázově téměř na konci celého projektu. To není právě nejšťastnější řešení vzhledem k zásadě, že testování by se mělo provádět tak brzy, jak je to možné.

1.8.2 Spirálový model

Spirálový model se také skládá z několika kroků, ale ty se neustále opakují, dokud není produkt hotov. Hlavní myšlenkou je postupné navazování nových částí na již řádně otestovaný a ověřený základ. Tester zde vstupuje do procesu velmi brzy a testuje aplikaci po částech, ne celou najednou. Tento model je vhodný pro větší projekty, protože je lze v průběhu tvorby ovlivňovat. To znamená, že jestliže se například změní podmínky na trhu, je možné aplikaci podle potřeb upravit.

---

3 Převzato z [18]

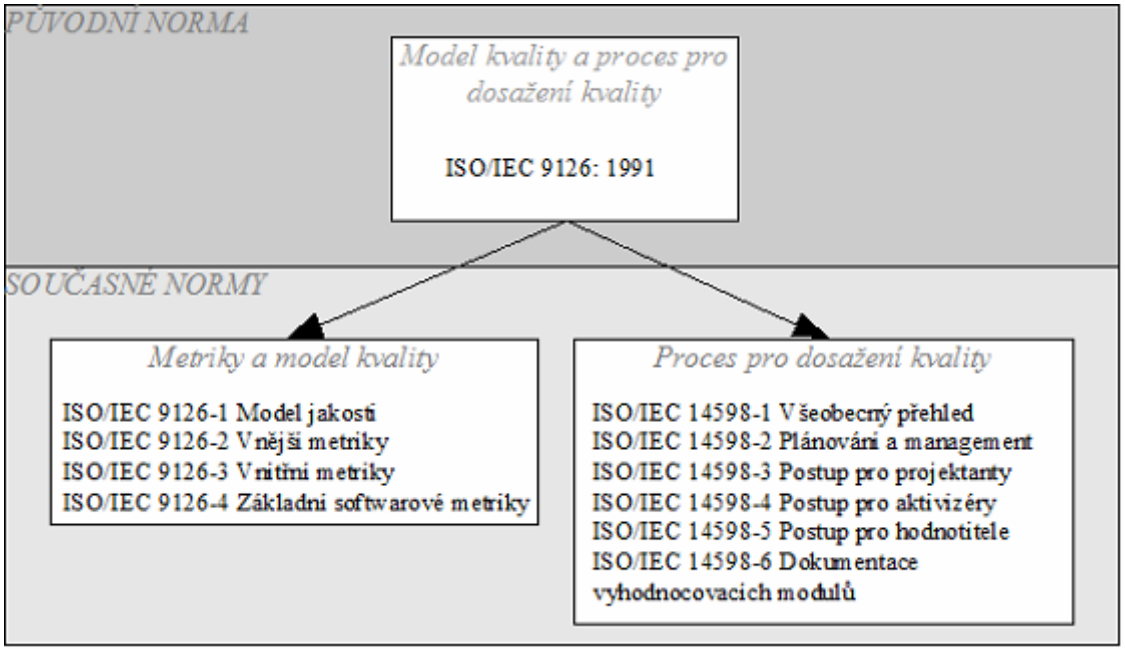
1.9 Normy ISO[4]

Existují mezinárodní normy, s jejichž pomocí lze definovat a ohodnotit míru kvality software. Tyto normy nejsou závazné pro většinu výrobců, ale jejich dodržováním lze zvýšit kvalitu produktu, spokojenost zákazníka a pověst výrobce.

1.9.1 ISO/IEC 9126 Informační technologie

ISO/IEC 9621 Informační technologie – Softwarové inženýrství je sada norem která vznikla z původní normy ISO/IEC 9126. Tato původní norma byla schválena v roce 1991 a obsahovala šest základních charakteristik, které se aplikovaly na všechny aspekty kvality softwaru. Subcharakteristiky a jejich metriky byly připojeny v dovětku, který nebyl součástí tohoto oficiálního standardu.

Norma z roku 1991 byla postupně nahrazena několika propracovanějšími normami. Proces nahrazení zobrazuje obrázek[5]:



Obrázek 6: Proces nahrazení ISO/IEC 9126

ISO/IEC 9126-1 Model jakosti

---

4 Zpracováno podle [10] a [11]

5 Zpracováno podle [11]

Tato norma obsahuje původní normu ISO/IEC 9126, s tím, že subcharakteristiky jsou již přesunuté do oficiální části normy (nejsou v dodatku). Definuje šest charakteristik: funkčnost, bezporuchovost, použitelnost, účinnost, udržovatelnost a přenositelnost.

ISO/IEC 9126-2 Vnější metriky

Produkt se spustí v simulovaném prostředí a pomocí vnějších metrik měříme jeho chování. Simulované prostředí upravujeme a porovnáváme požadované výsledky s očekávanými.

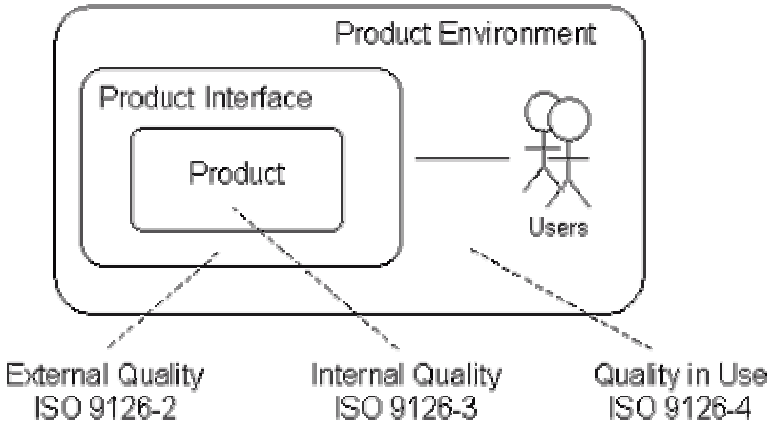
ISO/IEC 9126-3 Vnitřní metriky

Měří se statické vlastnosti produktu, například počet řádků zdrojového kódu na modul, počet volání procedur, počet modulů, apod.

ISO/IEC 9126-4 Základní softwarové metriky

Měří se kvalita software z hlediska uživatele. Například to, jak snadno uživatel dosáhne svého cíle. Měří se v prostředí, kde bude systém běžet, tj. prostředí by se mělo co nejvíce podobat prostředí, které bude mít nainstalován a nastaven uživatel. Měří se tyto vlastnosti: efektivita, produktivita, bezpečnost, spokojenost.

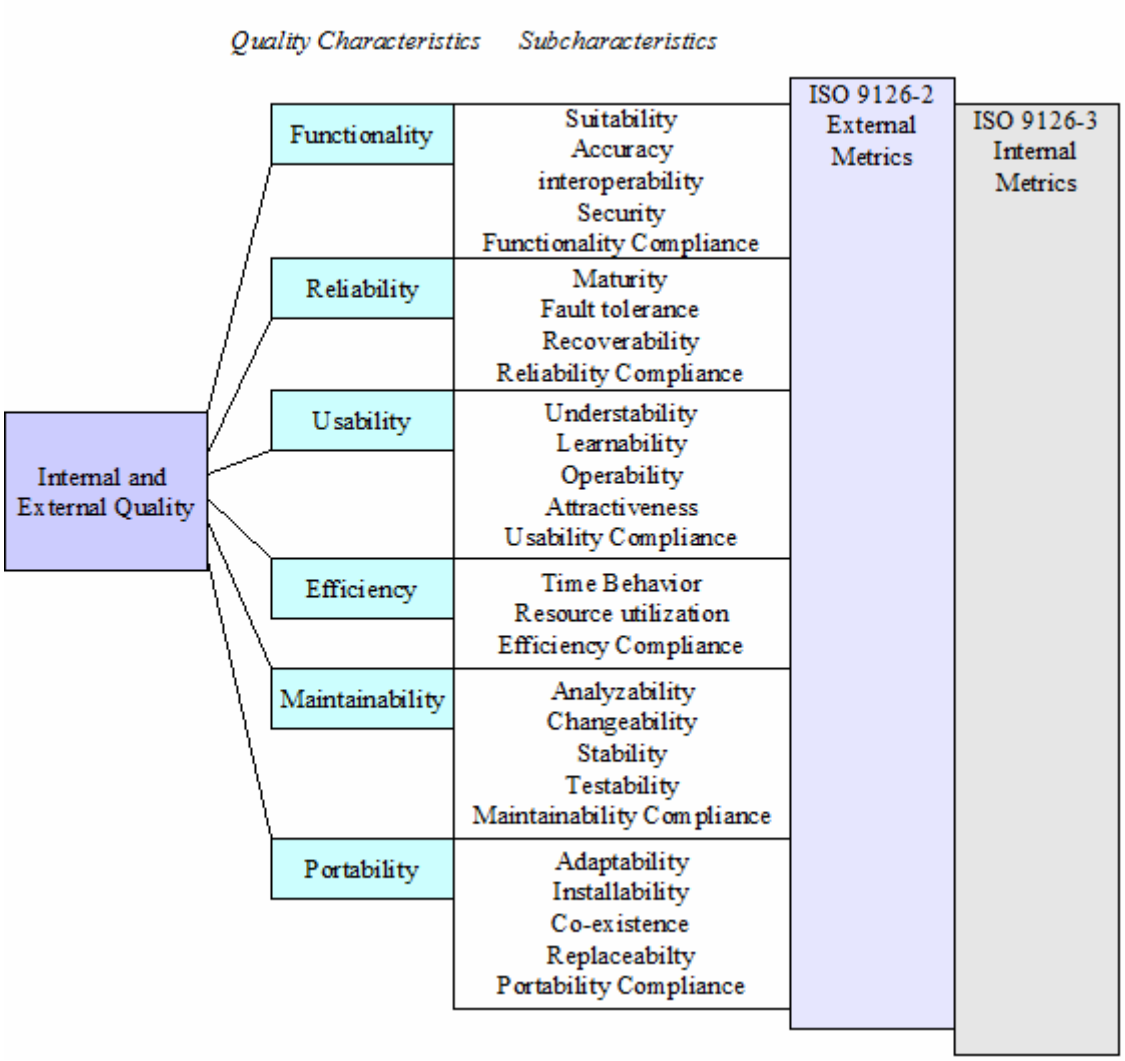
Obrázek 7 zobrazuje vztah mezi měřenými kvalitami softwaru, reprezentovanými samostatnými normami[6]:



Obrázek 7: Vztah mezi jednotlivými normami

6 Převzato z [11]

Následující obrázek zobrazuje všech šest charakteristik včetně jejich subcharakteristik a vztahu k metrikám.



Obrázek 8: Vztah mezi charakteristikami, subcharakteristikami a metrikami

1.9.2 ISO/IEC 14598 Informační technologie

Norma ISO/IEC 14598 obsahuje několik částí, jejichž společným názvem je *Informační technologie - Hodnocení softwarového produktu*. Vznikla jako doplněk k normám řady ISO/IEC 9126. Tyto normy definují určité vlastnosti, které by měl software splňovat, ale už neříkají jak je změřit. Norma ISO/IEC 14598 definuje proces hodnocení, který určuje postup v krocích, jak tyto informace o produktu získat. Postup v krocích je zde popsán pro: projektanty, aktivizéry a hodnotitele. Aktivizérem se myslí potencionální zákazník, který zvažuje případnou koupi produktu.

2 MOŽNOSTI TESTOVÁNÍ WEBOVÝCH STRÁNEK

2.1 Specifika testování webových stránek

Webové stránky mají některá specifika, k nimž je potřeba při testování přihlédnout:

- jsou umístěny na Internetu, což znamená, že k nim má přístup prakticky kdokoliv
- uživatelé, kteří se připojují, mohou mít různě velkou stahovací rychlost
- webové stránky jsou zobrazovány v různých prohlížečích, což znamená, že se nemusí zobrazovat stejně
- jsou prohledávány roboty, kteří se z nich snaží vyčíst různé informace
- ve stejný okamžik s nimi mohou pracovat stovky uživatelů

Proces testování by se měl těmto faktům přizpůsobit.

2.1.1 Validita

Webové stránky by měly být validní, což by mělo výrazně napomoci tomu, že budou v různých typech prohlížečů zobrazeny korektně. Validitu zpravidla kontrolujeme pomocí některého nástroje. Lze použít validátory W3C:

- (X)HTML validátor najdeme zde: http://validator.w3.org/
- CSS validátor najdeme zde: http://jigsaw.w3.org/css-validator/#validate_by_uri

V případě obou validátorů můžeme buďto vložit URL testované stránky, nebo můžeme uploadnout soubor, popřípadě vložit přímo kód k validaci.

Existuje i česká mutace validátor W3C, nalezneme ji zde: http://validator.w3.cz/

Pokud potřebujeme, můžeme na stránce http://www.w3.org/QA/Tools/ nalézt další validátory, například *XML Schema Validator* a jiné.

Ale i přestože je kód validní neodpadá testování aplikace v různých prohlížečích. Validita kódu pouze napomáhá korektnímu zobrazení, ale nezaručuje jej.

2.1.2 Funkčnost odkazů

Funkčnost odkazů rovněž obvykle testujeme pomocí některého nástroje. Na W3C můžeme použít *LinkChecker*: http://validator.w3.org/checklink

Opět stačí zadat URL testované stránky, a pokud potřebujeme, můžeme využít možnosti rekurzivního vyhledávání – stačí zadat hloubku rekurze.

2.1.3 Kontrola vzhledu webové stránky v různých prohlížečích

Otestovat můžeme ručně, rozhodně vyzkoušíme Internet Explorer a Mozillu Firefox. Dále je dobré otestovat ještě Operu, Konqueror či Safari. Pro jistotu ve více verzích.

Můžeme také využít nástroj *Browsershots* na adrese: http://browsershots.org/

Zadáme URL testované stránky a pak počkáme, až nástroj provede screenshoty ze všech prohlížečů a jejich verzí, které jsme před odesláním vybrali. Pak si můžeme screenshoty stáhnout a prohlédnout. Nebo si je také můžeme prohlédnout online.

2.1.4 Rychlost načítání stránek

Rychlost načítání stránek ovlivňuje zejména množství a velikost obrázků, které jsou na stránce umístěny. Při testování nám může pomoci nástroj *Web Page Analyzer*, který nalezneme zde: http://www.websiteoptimization.com/services/analyze/

2.1.5 Optimalizace pro vyhledávače

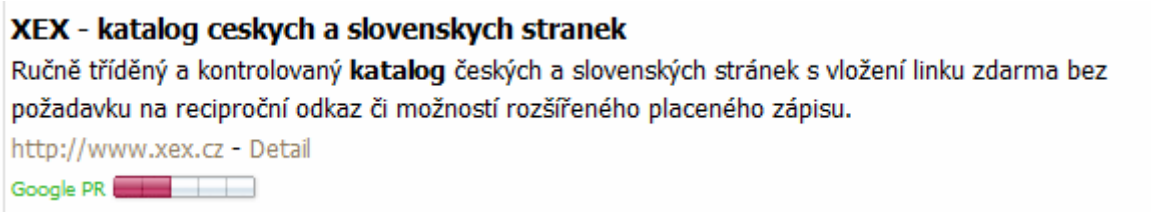
Optimalizaci webových stránek pro vyhledávače provádíme s cílem přimět vyhledávač, aby zacházel s naší webovou stránkou způsobem, jakým bychom si přáli. Jestliže se nejedná o interní aplikaci, která bude běžet na intranetu a ke které budou mít přístup pouze zaměstnanci firmy nebo o webové stránky, které jsou sice veřejné, ale informace na něm budou určeny pouze pro určitý okruh lidí, pak si budeme pravděpodobně přát, aby se naše stránky umisťovaly na co nejvyšších příčkách vyhledávačů.

Vyhledávače můžeme dělit podle způsobu vyhledávání na dva druhy:

Katalogové:

Katalogové vyhledávače pracují na principu katalogů, které jsou editovány lidmi. Příkladem může být vyhledávač na serveru Ukaž to: http://www.ukazto.eu/

Příklad výsledku hledání z katalogu:



Obrázek 9: Příklad výsledku hledání z katalogu

Zápis do katalogu provádí majitelé webových serverů zdarma či za poplatek.

Fulltextové:

Fultextové vyhledávače používají roboty, které automaticky procházejí webové stránky a při vyhledávání pak výsledky řadí podle určitých kritérií. Nejznámějším fulltextovým vyhledávačem je Google. Google používá pro ohodnocení stránek tzv. *PageRank*, což je číslo, které je vypočítáno podle určitého algoritmu. Toto číslo má rozmezí od 0 do 10, přičemž desítka je nejlepší. *PageRank* se nepohybuje po lineární křivce, takže například rozdíl mezi 4-5 je větší než rozdíl mezi 1-2. Algoritmus, kterým lze *PageRank* spočítat, si Google velmi pečlivě střeží a čas od času ho mění. Nicméně uvádí několik zásad, které bychom měli dodržovat, pokud chceme, aby naše stránky byly umístěny na vysokých pozicích:

- Umožnit robotům čtení potřebných informací z našich stránek. Otestovat můžeme tak, že stránku otevřeme v textovém editoru. Měli bychom bez problémů přečíst všechna data. Obrázky mohou chybět, protože je robot ke své práci nepotřebuje. Pokud můžeme stránky testovat online, je vhodnější pro tento účel využít některý nástroj, například *Spider Simulator*: http://tools.summitmedia.co.uk/spider/
- Vhodný text v titulku každé stránky, ideálně by měl obsahovat alespoň některá klíčová slova.
- Kvalitní obsah v textu stránky, opět pokud možno s použitím klíčových slov. Ovšem v tomto případě je nejdůležitější, aby se text líbil zejména návštěvníkům webu, kteří by si pak odkaz rádi přidali na své stránky.
- Popisy obrázků. Zde se má na mysli vyplnění atributu *alt* u každého obrázku. Opět použijeme klíčové slovo, pokud možno.
- Pro nadpisy bychom měli důsledně používat tagy <H1> až <H3>. Jiné označení, např. pro CSS skripty, je robotem ignorováno.

- URL adresa by měla mít takový tvar, aby se z ní uživatel mohl dovtípit, co asi na stránce najde. Ve všech případných odkazech na webu by měla mít tento text.

Existují další faktory, které mají také vliv na hodnotu *PageRanku*, ale nemůžeme je zpravidla ovlivnit. Jsou to například.

- Linky, které vedou na naše stránky z ostatních webů, by měly obsahovat vhodný text, jestliže obsahují klíčové slovo, tím lépe.
- Velkou váhu má též ohodnocení webové stránky, která obsahuje link na náš web.
- Webová stránka, která odkazuje na naší webovou stránku, by měla být podobného zaměření. To znamená, že jestliže máme webové stránky, jejichž tématem je například chov psů, pak nám není moc platné, když odkaz na naši stránku bude umístěn na stránce, která se zabývá prodejem akumulátorů.

Dalším důležitým faktorem pro výpočet *PageRanku* je též stáří domény. Nové domény budou mít při naprosto stejných ostatních podmínkách nižší ohodnocení.

Při testování je dobré také vědět, že roboti analyzují každou stránku zvlášť, proto nestačí otestování například pouze domovské stránky.

2.1.6 Pravidla použitelnosti

Při testování webových stránek se zaměřujeme na testování použitelnosti více než v jiných aplikacích. Existují obecná pravidla, která je dobré dodržovat, protože pak se použití webu stává jednodušší pro většinu uživatelů. Těmito pravidly jsou například:

- nepoužívání jakýchkoli rušivých vlivů, které nelze vypnout. Jedná se o různé blikající nebo rolující animace a texty nebo nepřetržitý zvuk, apod. Pokud je použijeme, ztížíme tak uživateli snahu se soustředit na důležité informace. Také bychom se měli snažit, aby doprovodné obrázky nevypadaly jako reklama.
- Text by neměl splývat s pozadím. Barvy by měly být dostatečně kontrastní, aby se uživateli dobře četlo.
- Nestandardní barvy odkazů mohou uživatele zmást. Je také dobré, když jsou odkazy podtržené, aby uživatel na první pohled viděl, že se za textem skrývají další informace.
- Zastaralé informace na web nepatří. Čas od času by se měla provést revize textů a odkazů a neplatné by měly být smazány nebo nahrazeny platnými.

- Dlouhý, jednotvárný text se uživatelům často nechce číst. Měl by být rozčleněn do kratších odstavců a seznamů. Je také vhodné používat nadpisy a podnadpisy a zvýraznit klíčová slova.
- Dlouhé rolující stránky je vhodné rozdělit do několika kratších úseků a rozložit na více stránek. Pomůžeme tak uživateli lépe se zorientovat v informacích.
- Dlouhé čekání na stažení stránky uživatele unaví. Snadno pak přechází na stránky konkurence.
- Složitá URL může uživateli způsobit problémy, jestliže ji zadává ručně. URL by měla snadno zapamatovatelná, jednoduchá, bez speciálních znaků.
- Důležitým aspektem je též navigace. Každá stránka by měla mít svůj titulek, nadpis a link na domovskou stránku. Také by měla zobrazovat místo, kde se na webu nalézá. Stačí, když se právě otevřená stránka zvýrazní v menu. Pokud je web rozsáhlejší, je dobré vytvořit mapu webu.
- Tam, kde předpokládáme, že uživatel bude chtít tisknout, odstraníme přebytečné prvky.

Těchto obecných pravidel existuje více, výše uvedené jsou nejzákladnější. Kromě těchto obecných pravidel také existují pravidla použitelnosti daná zákonem.

2.1.7 Pravidla přístupnosti

Pravidla přístupnosti vznikla z důvodu pomoci handicapovaným uživatelům. V České republice jsou závazná pro všechny weby veřejné správy. Ostatní weby je ale také mohou dodržovat, mají-li provozovatelé zájem.

Pravidla přístupnosti pro Českou republiku se dělí do čtyř kapitol:[7]

- Kapitola A: obsah webových stránek je dostupný a čitelný,
- Kapitola B: práci s webovou stránkou řídí uživatel,
- Kapitola C: informace jsou srozumitelné a přehledné,
- Kapitola D: Ovládání webu je jasné a pochopitelné.

---

7 Zpracováno dle [5]

Tato pravidla se dělí ještě do dvou skupin. A to na povinná pravidla a na podmíněně povinná pravidla. Obě skupiny pravidel se vyskytují ve všech čtyřech kapitolách.

Podrobnosti lze nalézt zde: http://www.pristupnost.cz/pravidla-pristupnosti/

2.2 Ruční testování

Ruční testování je oproti automatickému testování velice pomalé a provádí se prakticky jen v případech, pokud automatický test nemáme k dispozici. Jestliže testujeme novou funkci, musíme test nejprve vytvořit. Nejrychlejším způsobem vytvoření testu je použití nástroje, který je schopen nahrát postup ručního testování. Takže první otestování se provede ručně a další otestování je pak možné spustit automaticky. V některých případech se ale může stát, že test nelze vytvořit, pak nezbývá nic jiného, než provádět ruční testy s každou verzí.

2.3 Automatické testování

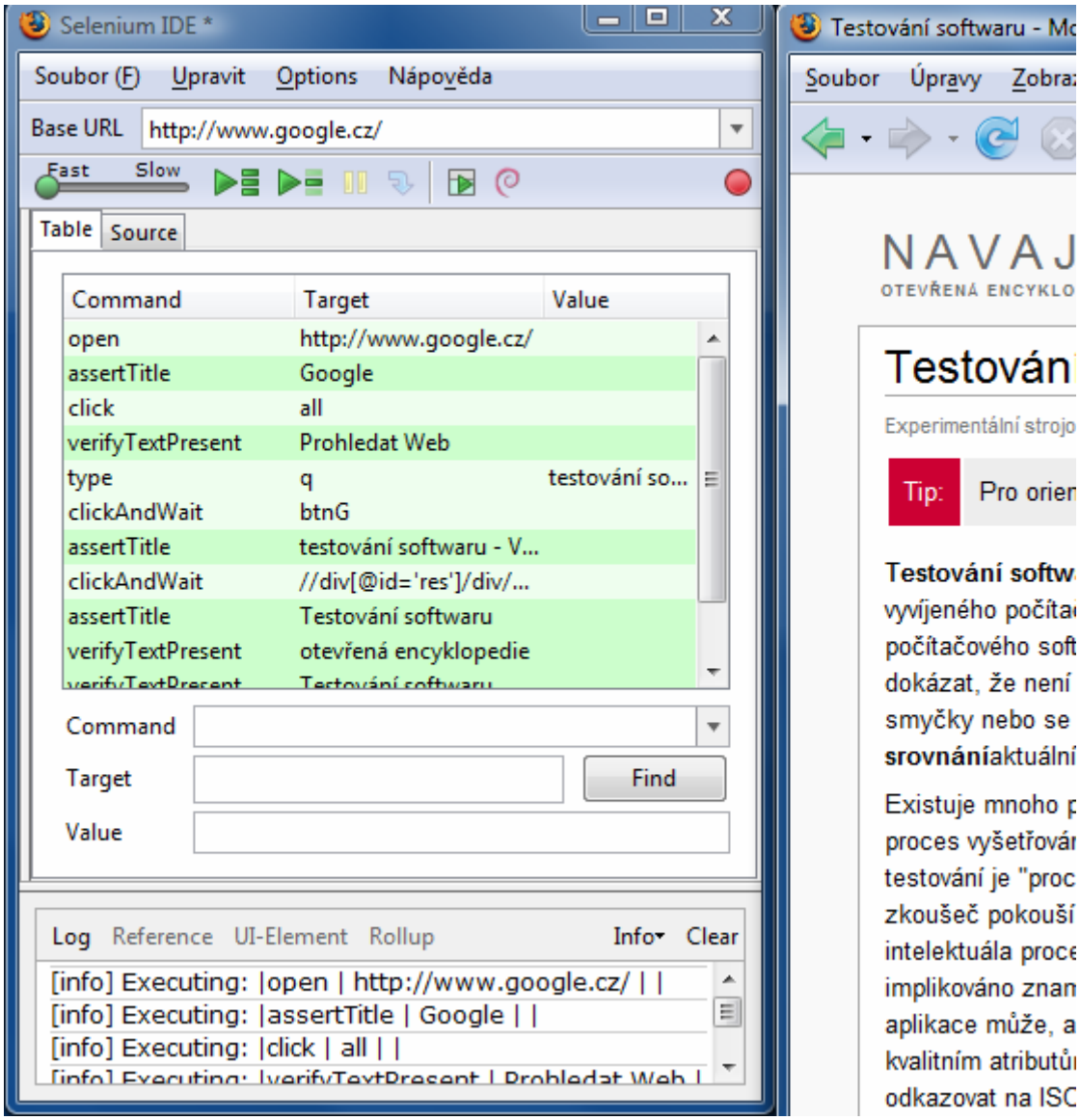
Automatické testování je velice oblíbené zejména z toho důvodu, že je časově a ekonomicky mnohem méně náročné než ruční testování. Využívá se velmi často, zejména v případech regresních testů. Automatický test si musíme vytvořit. Existují různé testovací nástroje, které nám mohou pomoci. Logiku testování si ovšem tester vždy musí vymyslet sám. Testy můžeme automatizovat pro různé typy testování.

2.3.1 Automatické testy pro akceptační testování

Pro akceptační testování webových stránek se často používá Selenium IDE. Tento nástroj se distribuuje jako doplněk k prohlížeči Firefox.

Selenium IDE funguje tak, že zaznamenává každou změnu ve webové stránce. Uživatel může například napsat text do inputu formuláře, zaškrtnout checkbox, kliknout na odkaz pro načtení nové stránky apod. Selenium zjistí ID komponenty, která byla pozměněna a změnu spolu s ID komponenty ihned zaznamená. Pokud komponenta nemá ID, pak se použije název komponenty. Celý záznam pak je možné uložit do souboru html, který lze opakovaně spouštět. Testy je snadné upravovat.

Na následujícím obrázku je znázorněn příklad použití Selenia IDE.



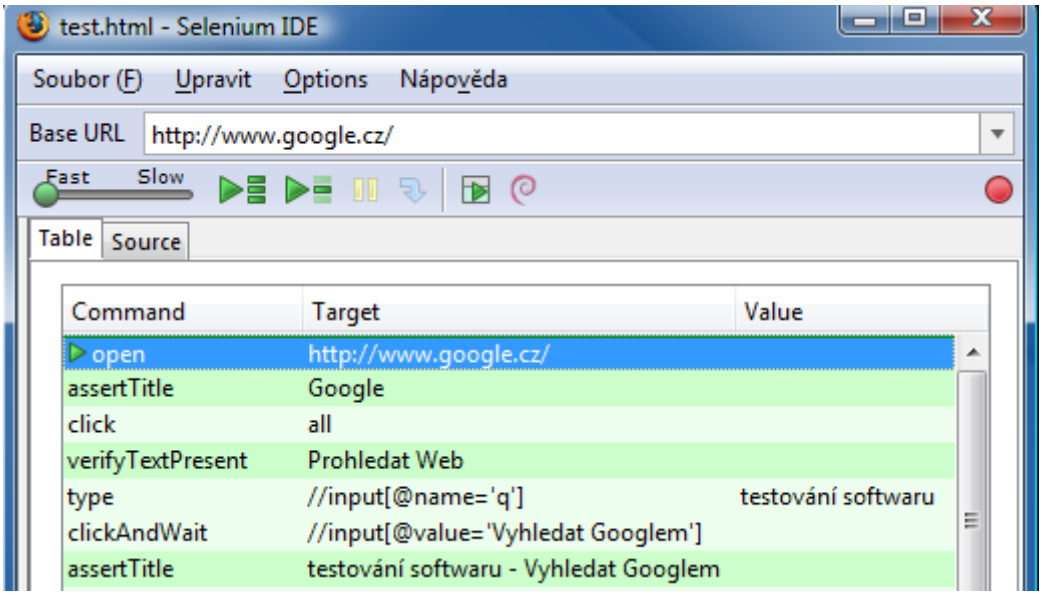
Obrázek 10: Použití Selenia IDE

Při opětovném spouštění pak Selenium přečte ID komponenty (ve sloupci Target) a provede u ní akci, kterou prve provedl uživatel. Pokud jsou webové stránky ještě ve vývoji, je zde riziko, že testy budeme muset upravovat často z toho důvodu, že se mohou měnit ID jednotlivých objektů. Jestliže se totiž změní ID, pak Selenium nebude schopné komponentu nalézt a test zhavaruje. Tomuto problému lze částečně přecházet používáním jazyka XPath, které nabízí více možností vyhledání objektu na stránce.

Na následujícím obrázku je záznam části testu, který Selenium IDE vytváří a za ním je obrázek příkladu použití XPath v Seleniu.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http:/
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="e
<head profile="http://selenium-ide.openqa.org/profiles/test-case
<meta http-equiv="Content-Type" content="text/html; charset=UTF-
<link rel="selenium.base" href="" />
<title>test</title>
</head>
<body>
<table cellpadding="1" cellspacing="1" border="1">
<thead>
<tr><td rowspan="1" colspan="3">test</td></tr>
</thead><tbody>
<tr>
        <td>open</td>
        <td>http://www.google.cz/</td>
        <td></td>
</tr>
<tr>
        <td>assertTitle</td>
        <td>Google</td>
        <td></td>
</tr>
<tr>
        <td>click</td>
        <td>all</td>
```

Obrázek 11: Ukázka testu uloženého v html



Obrázek 12: Příklad použití XPath v Seleniu

2.4 Jazyk XPath

Jazyk XPath používáme pro vyhledávání elementů ve stromové struktuře. Při testování webových stránek je nepostradatelný. Nejčastěji jej využíváme pro vyhledávání uzlů v HTML stránce, ale vyhledávat uzly může i v XML souborech, pro které byl původně vyvinut. Jednotlivé elementy vyhledáme buď tak, že půjdeme po cestě, na které leží anebo je najdeme podle určitého atributu anebo kombinací obojího. Zde je příklad:

Na následujícím obrázku je znázorněn příklad stromové struktury. Medvěda brtníka můžeme nalézt několika způsoby, rozdělím je do třech skupin.

Způsoby nalezení uzlu s použitím cesty:

- table/tr[2]/td[3]
- table/tr[2]/td[last()]
- //tr[last()]/td[3]

Způsob nalezení uzlu s použitím atributů:

- //td[text()='medved brtnik']

Způsob nalezení uzlu kombinací cesty a atributů:

- table[@class='savci']/tr[2]/td[text()='medved brtnik']
- //tr[@name='selmy']/td[text()='medved brtnik']

```
<table class="savci">
    <tr name="hlodavci">
      <td>veverka obecna</td>
      <td>bobr evropsky</td>
      <td>ondatra pizmova</td>
    </tr>
    <tr name="selmy">
      <td>kocka divoka</td>
      <td>rys ostrovid</td>
      <td>medved brtnik</td>
    </tr>
</table>
```

Obrázek 13: Příklad stromové struktury

Tyto příklady zdaleka nebyly vyčerpávající. Je možné si je vyzkoušet přímo v Seleniu. XPath pro zjednodušení pohybu po ose k hledanému uzlu definuje několik různých os.

Zde je několik příkladů těchto os včetně příkladů jejich použití a zobrazení ve stromové struktuře:

Osa	Popis osy	Příklad
Parent	Vyhledá rodiče uzlu.	**//PPP/parent::***
Child	Vyhledá potomka uzlu.	**//CCC/child::***
Self	Vyhledá sám sebe.	**//CCC/self::***
Preceding-sibling	Všichni předchozí sourozenci.	**//SSS/preceding-sibling::***
Preceding	Všechny předchozí uzly	**//OOO/preceding::***
Following-sibling	Všichni následující sourozenci	**//FFF/following-sibling::***
Following	Všechny následující uzly	**//PPP/following::***
Descendant	Vyhledá všechny potomky.	**//GGG/descendant::***
Ancestor	Vyhledá všechny předky.	

Tabulka 1: Osy XPath

```
<root>
    <BBB>
        <PPP>
            <UUU>
                <XXX>
                    <ZZZ/>
                </XXX>
            </UUU>
        </PPP>
        <SSS>
            <RRR/>
            <ZZZ/>
            <OOO/>
        </SSS>
        <CCC>
            <DDD>
                <GGG>
                    <HHH>
                        <XXX>
                            <YYY/>
                            <ZZZ/>
                            <RRR/>
                        </XXX>
                    </HHH>
                </GGG>
            </DDD>
            <NNN>
                <PPP>
                    <CCC>
                        <EEE>
                            <FFF/>
                            <QQQ>
                                <MMM/>
                            </QQQ>
                        </EEE>
                        <RRR/>
                    </CCC>
                </PPP>
                <SSS>
                    <KKK>
                        <RRR/>
                    </KKK>
                </SSS>
            </NNN>
            <OOO/>
            <QQQ/>
        </CCC>
    </BBB>
<root>
```

Obrázek 14: XPath osy - příklady

Tyto osy se v seleniových testech velmi často používají. V Seleniovém testu většinou potřebujeme konkrétní uzel. Některé osy ale ukazují na více uzlů. V tomto případě pak konkrétní uzel dále specifikujeme pomocí jeho názvu nebo některého atributu nebo funkce *position()*, případně funkce *last()*. Pokud tak neučiníme, selenium samo vybere vždy první uzel.

Velmi často se také používají operátory a zkrácené výrazy.

Přehled operátorů:

Operátor	**Typ výrazu**
/, //	Cesta k uzlu
|	Sjednocuje výsledky více výrazů
OR, AND	Logické výrazy
=, !=	Rovnost, nerovnost
<=, <, >=, >	Relační operátory
+, -, div, mod, *, -	Číselné výrazy

Tabulka 2: XPath operátory

3 TESTOVÁNÍ WEBOVÝCH STRÁNEK Z HLEDISKA BEZPEČNOSTI

Testování webových stránek z hlediska bezpečnosti se týká jen webových stránek samotných. Netýká se zranitelnosti operačního systému nebo webového serveru, na kterém webové stránky běží, trojských koní, jiných virů, spamu apod. To je jiný druh testování.

Testování bezpečnosti webových stránek je poměrně komplikovaná záležitost. Chyby mohou být různé a může jich být mnoho a není nikde řečeno ani napsáno co všechno se musí otestovat, aby stránky byly naprosto bezpečné. Navíc každá webová aplikace je unikátní a to i z hlediska výskytu chyb. Chyby, které nalezneme v jedné aplikaci, se nemusí objevit v jiných a naopak.

Existují obecné metody, které popisují nejznámější útoky, jako například SQL injection, skriptování stránek (Cross-site Scripting) nebo krádež relace. Kromě chyb, které umožní tyto typy útoků, ještě existuje velké množství dalších chyb. Programátor, který stránky vytváří, je ani všechny nemusí znát. Kromě toho se časem objevují další a další chyby přičemž jejich nárůst se může značně výšit například s přechodem na nové a z hlediska bezpečnosti dosud řádně neprozkoumané technologie.

3.1 SQL Injection

Jedná se o získávání dat z databáze pomocí uživatelského rozhraní. Uživateli je na webové stránce nabídnut formulář se vstupy pro zadání určitých dat. Útočník může vstup na formuláři zneužít tak, že do něj zadá dotaz SQL a odešle serveru. Pokud nejsou ošetřeny znaky, které databázový server považuje za speciální, pak útočník dostane data, na která se ptal.

Existují znaky, které lze považovat za rizikové. V každém případě by měla proběhnout kontrola znaků v zadaném řetězci ještě před tím, než aplikace začne řetězec zpracovávat. V některých případech je možné uživatele upozornit a přimět ho k tomu, aby znovu zadal řetězec, ale pouze složený z platných znaků. Ovšem ne vždy je možné nebo vhodné se snažit o odstranění těchto znaků. Například jedná-li se o heslo zadávané při vstupu do aplikace. Tam je naopak použití speciálních znaků žádoucí.

Také záleží na konkrétní databázi a jazyce, ve kterém je aplikace napsaná. Například jazyk PHP automaticky před apostrofy dosazuje zpětná lomítka. Nebo databáze MS Access je

výjimečná v tom, že nezná znaky používané v SQL pro označení začátku komentáře. Je zapotřebí všechny tyto specifikace nejprve nastudovat.

Sverre H. Husseby ve své knize *Zranitelný kód* uvádí varovný příklad v jazyce Java:

```
uzivJmeno = request.getParameter("uziv");
heslo = request.getParameter("hes");
query = 'SELECT * FROM Uziv "
      + "WHERE UzivJmeno='" + uzivJmeno + "' "
      + "AND Heslo ='" + heslo + "'";
```

Obrázek 15: Příklad SQL Injection

Zde se očekává, že uživatel do formuláře vyplní své uživatelské jméno a heslo. Pokud ovšem do inputu místo hesla zadá tento řetězec:

```
honza' --
```

Pak výsledný dotaz do databáze bude vypadat takto:

```
SELECT * FROM Uziv WHERE UzivJmeno='honza' --' AND Heslo=''
```

Pokud pracujeme s databází MS Access, dotaz upravíme takto:

```
SELECT * FROM Uziv WHERE UzivJmeno='honza' OR 'a'='b' AND Heslo=''
```

V tomto případě si je útočník vědom toho, že operátor AND má přednost před operátorem OR a tudíž se provede jako první. Výsledkem je, že jestliže uživatel honza v databázi skutečně existuje, pak útočník dostane všechna data z příslušného řádku tabulky.

Je potřeba podotknout, že podobně lze vkládat mnohem nebezpečnější příkazy, jako například smazání uživatele popřípadě vymazání všech dat z databáze. Stejně tak je možné vkládat data.

V konkrétních případech vždy záleží na tom, jaká práva útočník získá - zda může data jen prohlížet nebo i přidávat či mazat. Může pomoci určité nastavení databáze, ale na to se nelze spoléhat. Výrobce webové aplikace zpravidla neovlivní způsob, jakým si zákazník svou databázi nastaví. Taktéž po upgrade či zálohování se může nastavení databáze změnit. Proto by se mělo vše řešit přímo v aplikaci.

Dále by se aplikace měla postarat o to, aby se uživateli nikdy nevypisovaly podrobné chybové hlášky. Útočník je potřebuje k tomu, aby mohl zjistit informace o struktuře databáze, což k provedení SQL Injection potřebuje znát. Jestliže útočník nemá k dispozici žádná chybová hlášení, pak nemá ani žádné záchytné body což mu značně ztíží útok.

Sverre H. Husseby dále uvádí dvě možnosti, jak se útoku SQL Injection účinně bránit. Prvním je, že si programátor nastuduje dokumentaci k databázovému serveru, aby zjistil všechny problematické znaky. Pak tyto znaky ošetří tak, že je bude například zdvojovat. Jinými slovy, jestliže uživatel zadá problémový znak, pak jej aplikace před odesláním do databáze v řetězci zduplikuje. Je ale důležité žádný znak neopomenout. Je ale možné tyto znaky ošetřit i jinak, například jazyk PHP před každý takovýto znak přidává tři zpětná lomítka.

Dalším způsobem jak útoku SQL Injection zabránit je použití předpřipravených příkazů. Zde se jedná o to, že data jsou do databáze odesílána odděleně od příkazů. Ovšem ne každý databázový server tuto možnost podporuje. Příklad použití převzatý z knihy Sverre H. Hussebyho:

```
PreparedStatement ps = conn.prepareStatement(
                                "UPDATE zprávy SET nadpis=? WHERE id=?");
...
...
...
ps.setString(1, nadpis);
ps.setInt(2, id);
ResultSet rs = ps.executeQuery();
```

Obrázek 16: Příklad použití předpřipravených příkazů

3.2 Krádež relace

Relace vzniká po přihlášení uživatele k serveru. Je to množina proměnných, které uchovávají aktuální stav připojení klienta k serveru. Servery mohou obsluhovat mnoho připojení, proto spojují jednotlivé relace s konkrétními uživateli. Obvyklý postup je ten, že server vygeneruje cookie a zašle ji uživateli. Cookie obsahuje identifikační informace, které klient spolu s každým dotazem odesílá zpět serveru. Server podle přijaté cookie rozpozná klienta, pokračuje v relaci a odpovídá na dotazy. Po odhlášení klienta se cookie zruší nebo alespoň zneplatní.

Jestliže útočník zcizí cookie a odešle spolu s dotazem na server, pak server nic nepozná a na dotaz odpoví. Takto útočník získá identitu a všechna práva oběti.

Existuje několik možností obrany, ale žádná z nich není dostačující:

- Server může spojovat identifikátor uložený v cookie s některými dalšími informacemi o klientovi. Například použije IP adresu klienta. V tomto případě pak kontroluje, zda se nezměnila IP adresa, ze které byl zaslán identifikátor. Pokud ano, pak relaci okamžitě ukončí. Tato kontrola ale nebude fungovat v případě, kdy oběť i útočník používají stejný proxy server.
- Obdobně může server použít informace z hlavičky HTTP. Jestliže dorazí identifikátor a sledovaný údaj v HTTP hlavičce se bude lišit od předchozí hodnoty, pak server opět okamžitě ukončí relaci. Nevýhodou je fakt, že útočník může HTTP hlavičku snadno editovat.
- Dalším možným řešením je, že server bude s každou odpovědí zasílat klientovi nový identifikátor. V tomto případě existuje riziko, že útočník odešle dotaz na server dříve než klient. Server pak pošle odpověď a nový identifikátor útočníkovi a klientovi znemožní přístup k relaci.

Zcizení cookie útočníkem nemůžeme ve své webové aplikaci ovlivnit, to si musí každý uživatel ohlídat sám, ale minimálním opatřením by mělo být alespoň vygenerování nové cookie vždy po zalogování uživatele do aplikace.

Pro otestování změny identifikátoru po zalogování do aplikace a pro zjištění, jak naše aplikace reaguje na změny v HTTP hlavičce, můžeme použít například nástroj *Fiddler*. Tento nástroj je freeware a je ke stažení zde: http://www.fiddler2.com/fiddler2/

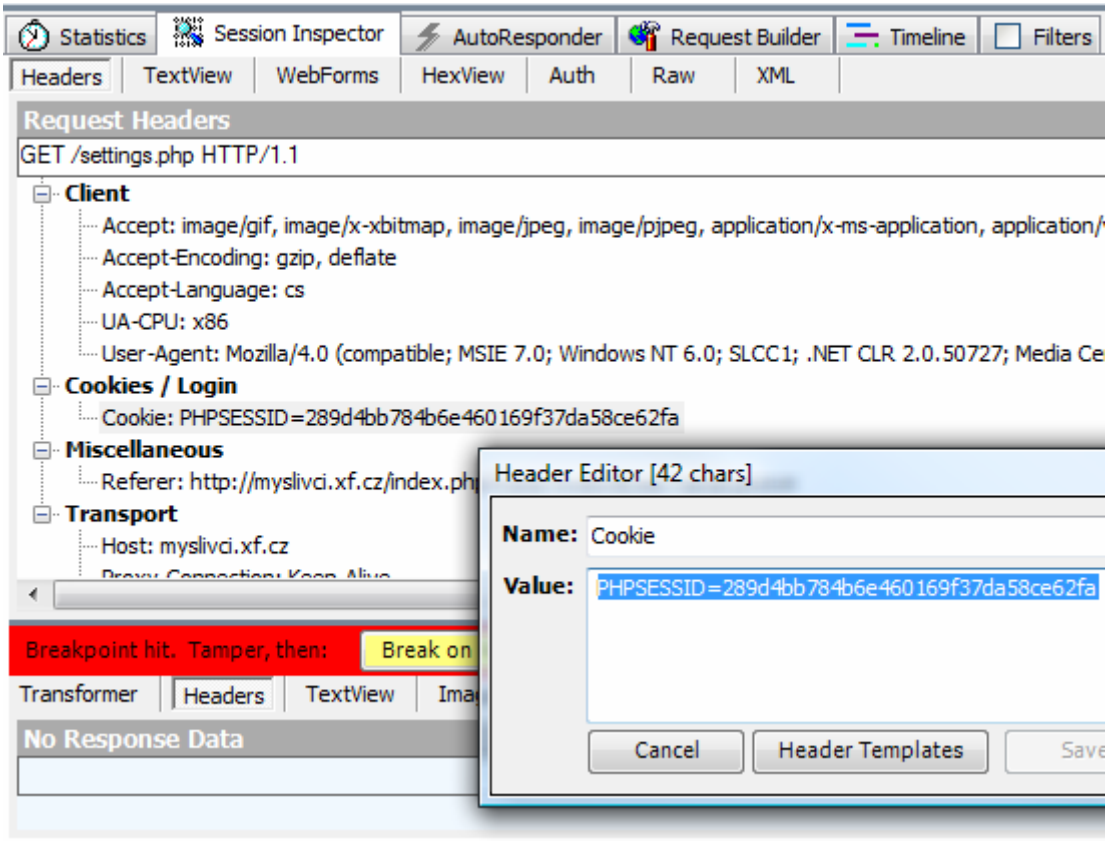
Umožňuje sledovat HTTP i HTTPS provoz mezi browserem a Internetem. Standardně pracuje s Internet Explorerem, ale lze jej použít i pro Firefox, Operu, a další.

Pokud bychom ho chtěli použít ve spojení s Firefoxem, je potřeba provést toto nastavení:

- na disku nalézt soubor *BrowserPAC.js*. Při instalaci se některé soubory Fiddleru kopírují do adresáře *Documents*. Soubor *BrowserPAC.js* se u mě na disku nalézá zde:

  C:\Users\Pavlina\Documents\Fiddler2\Scripts\BrowserPAC.js

- cestu k souboru zkopírovat, otevřít ve Firefoxu panel *Možnosti*, záložku *Rozšířené* a kliknout na tlačítko *Nastavení* u možnosti *Konfigurovat připojení aplikace Firefox k Internetu*. Otevře se panel *Nastavení připojení*. Zde zatrhneme možnost *Použít skript pro automatickou konfiguraci* a do příslušného pole vložíme zkopírovanou cestu k souboru. Vše uložit a zrestartovat Firefox.



Obrázek 17: Příklad práce s nástrojem Fiddler – zobrazení HTTP hlavičky a editace hodnoty cookie před odesláním na server

3.3 Skriptování stránek

Skriptování stránek obecně si neklade za cíl poškodit webovou aplikaci jako takovou. Hlavní myšlenkou je spíše zneužít aplikaci a poškodit její uživatele. Cílem skriptování stránek je právě velmi často krádež relace. Útočník nemá mnoho možností jak by zjistil hodnotu cookie, kterou server odesílá svým klientům. Jednou z těchto několika málo možností je právě vložení skriptu do stránky. Tento skript, má za úkol zjistit identifikátor oběti a zaslat ho útočníkovi.

Obranou proti tomuto typu útoku je zabránit možnosti vkládání skriptů do stránek. A to i HTML tagů, pokud možno. Často se používá technika, které se říká *kódování HTML*. Jde o to, že se metaznaky nahradí „neškodnými" znaky už při předávání řetězce aplikaci. Tyto znaky mají tu vlastnost, že se zobrazují v prohlížeči, kde mají podobu metaznaku, ale nemají jeho funkci. Nahrazení probíhá podle tabulky:

Nahrazovaný znak	Zástupný znak
&	&amp;
"	&quot;
<	&lt;
>	&gt;

Tabulka 3: Nahrazení metaznaků

Pokud bychom chtěli ponechat uživateli možnost formátování vstupu, pak máme několik možností řešení, z nichž nejznámější je vytvoření speciálního značkovacího jazyka. Tento přístup aplikují zejména Internetové encyklopedie typu *wiki*.

3.4 Co je potřeba dále otestovat

3.4.1 Dostupnost citlivých informací získaných z Internetu

Pomocí vyhledávacích portálů jako například www.google.com, www.yahoo.com, www.altavista.com, www.lycos.com, www.seznam.cz apod. se pokusit najít jakékoliv informace o webu nebo o zaměstnancích firmy, kterou web reprezentuje. V ideálním případě bychom měli najít pouze nejnutnější kontakty k tomu, aby společnost mohla fungovat. Neměli bychom nalézt jména, emailové adresy, poštovní adresy nebo telefonní

čísla zaměstnanců, kteří nemají za úkol komunikovat s potencionálními zákazníky, uchazeči o zaměstnání, apod. Je potřeba omezit počet veřejných kontaktů na minimum, neboť útočník by je mohl snadno zneužít, pokud by se rozhodl pro útok typu *sociálního inženýrství*.

Také existují různé konference a fóra, které se zabývají problematikou IT. Může se stát, že se někteří zaměstnanci firmy budou dotazovat na problém, s nímž si nevědí rady a který se týká příslušné webové aplikace nebo nastavení serveru na kterém aplikace běží. Útočník z těchto konferencí může získat cenné informace.

3.4.2 Dostupnost citlivých informací získaných z kódu

- Nejčastěji se jedná o technické informace v komentářích. Nemělo by být možné z nich vyčíst jak aplikace nebo její část funguje.
- Nemělo být možné vyčíst cesty k souborům či adresářům z kódu. Pro útočníka jsou soubory na webu vždy zdrojem cenných informací. Zejména jedná-li se o logy, ze kterých může vyčíst přihlašovací informace uživatelů (a přitom vůbec nevadí, že jsou třeba zaznamenány pouze neúspěšná přihlášení), nebo o soubory, které obsahují zdrojový kód, apod.
- V případě havárie systému by se nemělo stát, že se uživateli vypíše podrobné hlášení. Útočník, který nemá potřebné informace o systému, se je často snaží získat právě „shozením“ aplikace a následným čtením informací z errorových hlášení.
- Také je nutné dávat pozor na odesílání informací metodou GET. Tyto informace se posílají v adrese URL, která je snadno čitelná a pro realizaci útoku stačí, že ji jednoduše zkopírujeme. Citlivé informace jako například uživatelské jméno, heslo či hodnota cookie, apod. by se měly posílat zásadně metodou POST.

3.4.3 Uživatelské účty

Uživatelská hesla by měla být dostatečně dlouhá a pokud možno, měla by to být kombinace písmen, čísel a speciálních znaků. Jinak řečeno, aplikace by přinejmenším neměla umožňovat nastavení slabých hesel o délce kratší než 8 znaků.

V případě určitého počtu neúspěšných přihlášení by se měl účet zamknout. To je vhodné pro případ, kdy se útočník pokusí o útok hrubou silou, případně o slovníkový útok.

Dále je vhodné nevypisovat uživatelské jméno po neúspěšném přihlášení. Bude lépe, když si útočník nebude jist, zda uhodl alespoň uživatelské jméno.

3.4.4 Šifrování hesel

Hesla by se měla na serveru ukládat v zašifrované podobě. A to pro případ, že se útočníkovi podaří prolomit do databáze nebo získat soubor, ve kterém jsou hesla uložená. Pokud tato situace nastane, je lepší, když jsou hesla ještě zabezpečená nějakým dalším způsobem.

V ideálním případě aplikace neukládá hesla na server vůbec. Při registraci uživatele může aplikace předat heslo hašovací funkci, která z hesla vytvoří haš a ten pak uloží do databáze či souboru. Při dalším přihlášení uživatele se hašovací funkci opět předá zadané heslo, ze kterého se opět vytvoří haš a ten se pak porovná s uloženým hašem. Pokud hodnoty souhlasí, je zřejmé, že zadané heslo bylo správné. Útočník pak nemá žádnou šanci získat heslo a to ani v případě, že se do příslušné databáze prolomí. Může se sice pokusit o prolomení haše, ale to se mu nemusí podařit.

II. PRAKTICKÁ ČÁST

4 NÁVRH NA VYLEPŠENÍ TESTOVÁNÍ

Praktická část této práce se týká akceptačního testování. Akceptační testování patří do části vysokoúrovňového testování, kdy aplikaci testujeme jako celek a tedy nevyřazujeme z činnosti žádné její komponenty. Vytváření a používání regresních testů je zde velice výhodné a velmi často odhaluje chyby, které se do kódu zanáší při vytváření nových funkcí aplikace či při opravě chyb.

4.1 Používané nástroje

Při vytváření regresních testů pro webové aplikace využíváme různé nástroje, které by nám mohly usnadnit práci. Pro lepší přehlednost bych je rozdělila podle použitelnosti do dvou základních skupin. Do jedné skupiny bych zahrnula nástroje, které umožňují rychlé vytvoření testu, ale jejich použití se omezuje pouze na prohlížeč, ve kterém uživatel pracuje. Tester také zpravidla nepotřebuje téměř žádné znalosti programování. Použití těchto nástrojů je velmi jednoduché, intuitivní, vše je možné „naklikat“. Případně je vhodné některé výrazy přepsat do XPath, pokud používaný nástroj tento jazyk podporuje.

Vytvoření druhého typu testů je více pracné, ale testy jsou mnohem mocnější. Pomohou nám například v případech, kdy potřebujeme v průběhu testování pozměnit data v databázi, nebo zkontrolovat existenci či obsah určitého souboru na disku, automaticky pozměnit data v konfiguračním souboru, nebo ověřit zda aplikace odeslala email, když měla a případně zkontrolovat jeho obsah, nebo provést restart webového serveru, apod. Tyto typy testů nelze vytvořit automaticky, musíme je naprogramovat a jako nástroje používáme vývojová prostředí.

Ráda bych ještě dodala, že ať vytváříme jakýkoliv test, vždy bychom měli dbát na to, aby byl pokud možno co nejjednodušší a velmi dobře čitelný a pochopitelný. Důvodem je fakt, že v případě, kdy test selže, musí se tester nejprve přesvědčit, zda vada není v testu. Pokud ji tam nenalezne, pak teprve jde hledat chybu v testované aplikaci. Kód testu je tedy většinou kontrolován častěji než kód samotné aplikace a pokud je srozumitelný, pak chyba se najde mnohem rychleji. Další důvod je ten, že je potřeba čas od času, anebo podle potřeby, testy modifikovat. Úprava složitého testu by byla obtížná a časově náročná.

4.1.1 Nástroje zaznamenávající činnost uživatele

Jedná se o nástroje, které umí zachytit a uložit záznam uživatelských aktivit na webové stránce. S použitím tohoto typu nástrojů lze zpravidla rychle vytvořit automatický test bez velkých nákladů. Poté co dostaneme další verzi produktu, tento záznam na něm spustíme. V případě, že se při přehrávání objeví problém, začneme pátrat po příčině a nalézat chyby.

Na tomto principu fungují například nástroje *Selenium IDE* nebo *iMacros*. Oba dva lze získat zdarma a instalují se jako doplněk k Firefoxu. Selenium je používáno častěji. Oba uvedené nástroje pracují s reálným prohlížečem, ale existují také nástroje, které pracují s určitou náhradou reálného prohlížeče. Příkladem může být třeba *Canoo WebTest*.

Tyto nástroje nám mohou pomoci otestovat například logiku aplikace, kdy si zvolíme konkrétní cíl, ke kterému se snažíme „doklikat“. Nebo mohou pomoci otestovat prosté uživatelské „proklikávání“ kdy procházíme aplikací bez určitého cíle, ale přitom můžeme například sledovat a kontrolovat výskyt určitých objektů na webové stránce nebo výskyt určitého textu.

4.1.2 Vytváření testů s použitím programovacích jazyků

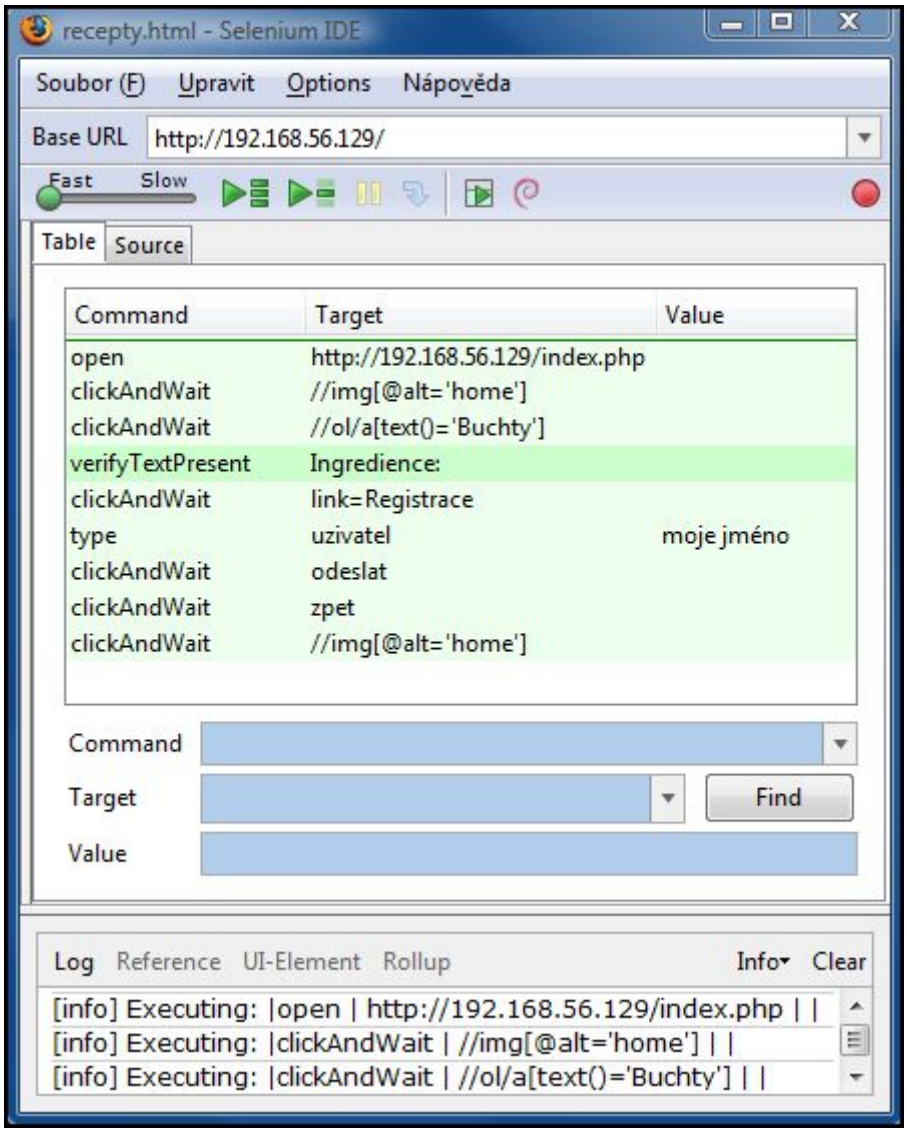
Tyto testy vytváříme, pokud chceme provést například download souboru a následně otestovat jeho existenci na disku, případně zkontrolovat jeho obsah. Problematické a zdlouhavé může být také třeba testování vyhledávání dat v aplikaci podle různých časových kritérií. V tomto případě, poté co dostaneme novou verzi webové aplikace a potřebujeme například otestovat funkcionalitu vyhledávání uploadnutých souborů podle data vytvoření, pak obvykle nemáme jinou možnost, než uploadnout soubory do aplikace a následně jít do databáze kde u nich pozměníme data vytvoření a nastavíme je podle potřeby, třeba i několik let zpětně. Pokud bychom chtěli celý proces zautomatizovat, pak nemáme jinou možnost, než si test naprogramovat.

Další výhodou těchto testů je možnost ošetření chyb. Nástroje typu pracující na principu zaznamenání činnosti uživatele se při jakékoliv chybě zastaví a nejedou dál. Při použití programovacího jazyka můžeme určit, jak se má test zachovat v případě výskytu určitého typu chyby.

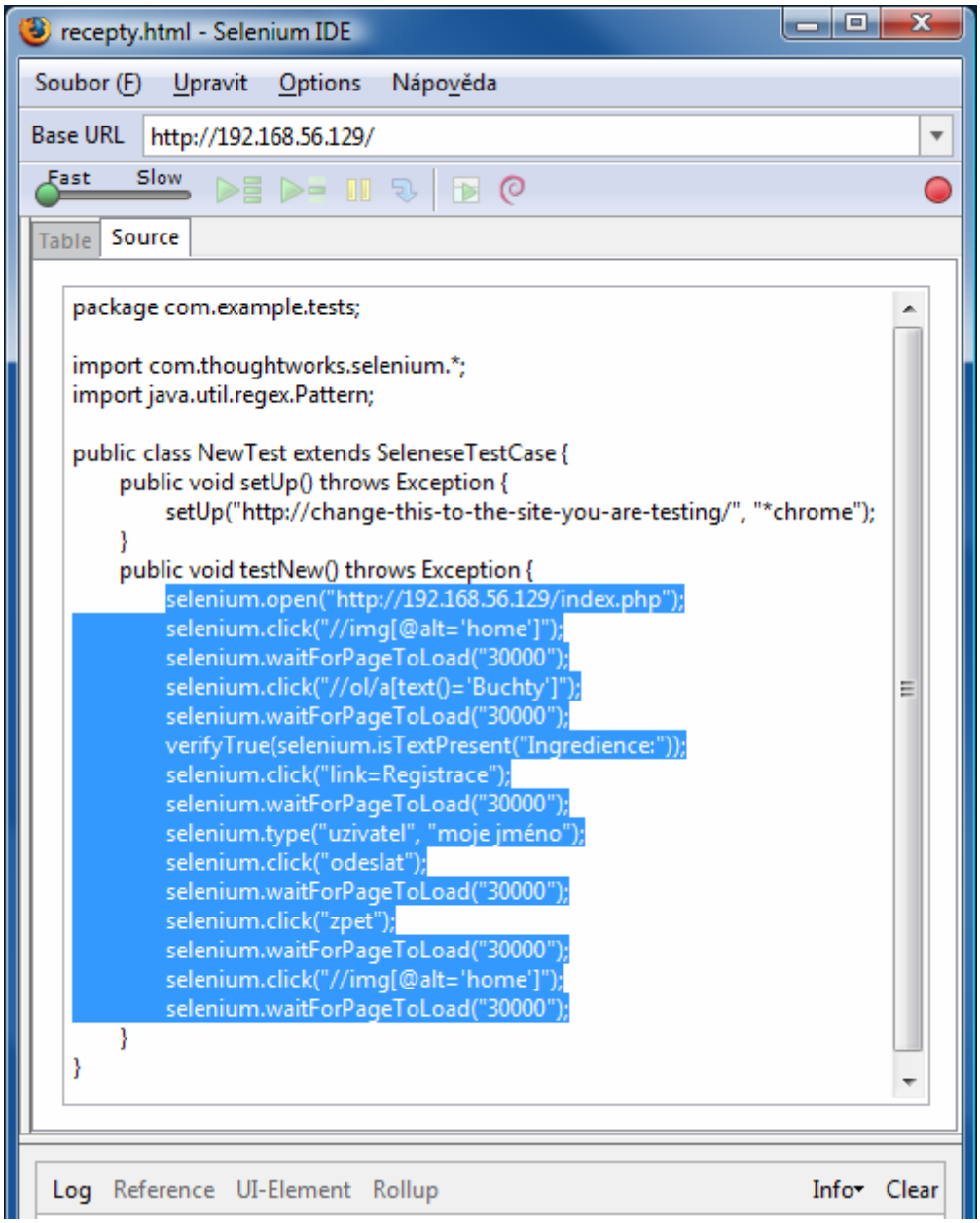
4.1.3 Použití více nástrojů při vytváření jednoho testu

Použití více nástrojů je možné v případě, že používáme *Selenium*. *Selenium* umožňuje vytváření tzv. x-Unitů, které lze psát v různých jazycích, například v Javě, C#, PHP, Perlu a dalších. Pro psaní těchto testů poskytuje *OpenQA* knihovny a tzv. *Selenium Server*, ve kterém se následně testy spouští. Část, kterou je možné jednoduše naklikat v browseru, vytvoříme v *Seleniu IDE*, které zároveň umožňuje konverzi naklikaného kódu do příslušného jazyka. Následně zkonvertovanou část zkopírujeme do svého x-Unitu.

Ukázka postupu vytvoření kombinovaného testu v Javě –Junit:



Obrázek 18: Vytvoření části testu, přepsání některých výrazů do XPath



Obrázek 19: Test zformátovaný do javového kódu

Nejprve vytvoříme část testu s použitím *Selenia IDE*, přepíšeme některé výrazy do Xpath a nakonec test spustíme, abychom měli jistotu, že funguje.

Poté test zformátujeme do javového kódu (Options -> Format -> Java) a potřebnou část zkopírujeme. Následně pak v některém vývojovém prostředí vytvoříme kostru testu, kam vybraný kód zkopírujeme a doprogramujeme zbytek. Nakonec ještě musíme vytvořit *Test Suite*, což je třída, která je určená pro spouštění testů. Testů obvykle máme větší množství, tak abychom nemuseli pouštět každý zvlášť, zaznamenáme je do *Test Suite*, které se o spuštění všech postará.

Na následujícím obrázku je ukázka kostry testovací třídy. Barevně označené části kódu je nutno pozměnit podle potřeby. Modře označený text musíme vždy pozměnit tak, aby obsahoval název třídy. Zelená část značí (zleva doprava):

- adresa, na které běží seleniový server,
- port, na kterém běží seleniový server,
- browser, ve kterém se má test spustit,
- webová stránka, kterou má browser při svém spuštění otevřít (stránka, na které test začíná)

Hnědý text nahradíme kódem vlastního testu.

```
import com.thoughtworks.selenium.*;

import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestSuite;
import org.openqa.selenium.server.SeleniumServer;

public class Basic extends TestCase {

   private SeleniumServer seleniumServer;
   private Selenium selenium = new DefaultSelenium(
         "localhost", 4444, "*chrome", "http://192.168.42.193/WebCenter/");

   public Basic(String name) {
      super(name);
   }

   @Override
   protected void setUp() throws Exception {
      seleniumServer = new SeleniumServer();
      seleniumServer.start();
      System.out.println("Basic is running...");
      selenium.start();
   }

   @Override
   protected void tearDown() throws Exception {
      selenium.stop();
      seleniumServer.stop();
   }

   public void testNew() throws Exception {
      // VLASTNÍ TEST
   }

   public static Test suite() {
      return new TestSuite(Basic.class);
   }
}
```

Obrázek 20: Kostra třídy

Červeně značený text můžeme buď ponechat v původním tvaru tak, jak je v ukázce uveden, anebo kompletně odebrat. Tímto kódem spouštíme seleniový server, ve kterém běží testy. Spuštění seleniového serveru před samotným testováním je nezbytné, máme ale dvě možnosti, jak server spustit. První možností je, že server spustíme na začátku každého testu a na jeho konci pak zastavíme. V případě, že máme testů více, pak bude s každým testem znovu nabíhat následně se zastavovat, což může způsobit nemalou časovou ztrátu. Tato možnost je zobrazena v ukázce. Druhou možností je, že si server jednoduše spustíme z příkazového řádku a necháme běžet tak dlouho, dokud budeme potřebovat.

Spuštění serveru z příkazového řádku provedeme tak, že se přepneme do adresáře, ve kterém máme uložený soubor *selenium-server.jar* a zadáme:

```
java –jar selenium-server.jar
```

V tomto případě spuštění seleniového serveru v testu zrušíme.

Následuje příklad testSuitu, což je třída která spouští testy:

```
import junit.framework.*;

public class AllTests extends TestCase {

  public static void main(String[] args) {
                junit.textui.TestRunner.run(suite());
  }

  public static Test suite() {
        TestSuite test = new TestSuite("Testy");

                test.addTest(Nazev_prvniho_testu.suite());
                test.addTest(Nazev_druheho_testu.suite());
                test.addTest(Nazev_tretiho_testu.suite());
                test.addTest(Nazev_pateho_testu.suite());

        return test;
  }
}
```

Obrázek 21: Kostra Test Suitu

Z příkladu je zřejmé, že se budou pouštět čtyři testy. Tento počet však může být mnohem vyšší. Modře označené části jsou názvy jednotlivých testů, což je nutné upravit podle potřeby.

4.1.4 Pod pojmem „JUnit“ se skrývají dva typy testů

Ráda upozornila na skutečnost, že se jeden název používá pro dva velmi odlišné typy testů. Programátor si obvykle pod pojmem „JUnit“ představí test nízkoúrovňového testování, který testuje jednotlivou komponentu – zpravidla třídu. Tento test vytvoří instanci testované třídy, postupně volá její metody, dává jim různé vstupy a zároveň kontroluje výstupy těchto metod.

JUnity popsané v předchozí kapitole sice používají stejné knihovny a také část jejich kódu se podobá kódu nízkoúrovňových testů, ale samotná podstata testování je velmi odlišná.

Nízkoúrovňové testování sleduje funkčnost jedné třídy a jejích metod často bez spolupráce ostatních tříd. Vysokoúrovňové JUnity naopak nevynechávají žádné části aplikace a snaží se otestovat určitou funkcionalitu hotové aplikace.

Na následujícím obrázku je uveden krátký příklad nízkoúrovňového JUnitu. Týká se otestování třídy, která má za úkol načítat data ze souboru csv. Test není úplný a zobrazuje pouze otestování jedné metody, ale pro představu by měl být dostačující:

```
public class ReadCSVTest {

    ReadCSV readCsv = new ReadCSV("C:\\usersNFR.csv");

    public ReadCSVTest() {    }

    public static void setUpClass() throws Exception {    }

    public static void tearDownClass() throws Exception {    }

    public void setUp() {    }

    public void tearDown() {    }

    // Test of getData method, of class ReadCSV.
    @Test
    public void getData() {
        System.out.println("getData");
        String zahlavi = "Surname";
        int radek = 5;
        ReadCSV instance = readCsv;
        String expResult = "Satan";
        String result = instance.getData(zahlavi, radek);
        assertEquals(expResult, result);

        assertEquals("Bronzový",instance.getData("Surname", 1));
        assertEquals("zelva.vroubena@krysatec.com",instance.getData("Email", 10));
        assertEquals("label=Mmc",instance.getData("HomeDirectory", 5));
        assertEquals("Calopus 854, Solný důl",instance.getData("Address", 5));
        assertEquals("mrales",instance.getData("Password", 6));
        assertEquals("587135476",instance.getData("Phone", 1));
        assertEquals("mravenec.loupezivy@krysatec.com",instance.getData("Email", 7));
    }
```

Obrázek 22: Příklad nízkoúrovňového JUnitu

4.2 Často řešené problémy

4.2.1 Otestování vstupů

Problémů, které se při testování vyskytují, je více. Jedním, s kterým se často setkáváme, je otestování vstupů aplikace. Vstupů může být mnoho a do webového formuláře je zadáváme stále stejným způsobem, což láká k zautomatizování celého procesu. Ale reakce aplikace mohou být různé, což nám naopak automatizaci dosti komplikuje.

Situaci lze řešit s použitím nástrojů, které umí zaznamenat a následně přehrát aktivitu uživatele. Tyto nástroje mají navíc velkou výhodu v tom, že při selhání okamžitě vidíme místo ve kterém se stala chyba: test se zastaví a příslušný řádek se zřetelně označí a browser zůstane ve stavu, do kterého jej přivedl poslední příkaz. Nevýhodou ale je, že tento test by byl neúměrně dlouhý, špatně přehledný a tudíž by se nesnadno modifikoval, což je právě při testování vstupů velmi nepříjemné.

Při použití JUnitů mi naopak nevyhovuje způsob zjišťování chyb. Hledání místa, ve kterém se chyba vyskytla a důvodu jejího výskytu není tak přehledné a vyžaduje více času.

4.2.2 Spravování testů

Tester při své práci testy vytváří, ale to není a ani by to neměla být práce, která by mu zabrala většinu času. Mnohem častěji se věnuje samotnému testování, úpravě testovacího prostředí a po každé nové verzi produktu stráví velmi mnoho času úpravou již hotových testů.

Při samotném testování je velmi užitečné, je-li ke každému testu připojena informace o tom, co se zhruba v aplikaci testuje a jaké kroky test provádí, aby svého cíle dosáhl. Přitom není vhodné a mnohdy ani možné psát tyto informace přímo do testu.

Tester také prakticky neustále pracuje s logy, kde nalézá důležité informace o chybách – pokud nevidí příčinu chyby přímo v prohlížeči, pokusí se jí najít v logu. Dále tam hledá například informace o tom, co aplikace prováděla předtím, než nastala chyba. Logy jsou pro testera nezbytné a měly by být přehledné a dobře čitelné. Tester by měl vždy mít možnost logovat výsledky svého vlastního testu, i když může nahlížet do logu testované aplikace (pokud aplikace nějaký má). Informace z těchto dvou logů se doplňují.

4.3 Návrh na vylepšení

Vylepšení se týká vstupů do webové aplikace. A to vstupů, které jsou umístěny na webových formulářích. Vstupem do aplikace také může být příkazový řádek browseru, ale to se zde řešit nebude.

Testování vstupů se bude řešit jen v rámci testovaného formuláře, tj. v každém testu testujeme vstupy umístěné na jednom formuláři, pokud je v aplikaci nebo na stránce víc formulářů, vytvoříme vlastní test pro každý z nich.

Test pro každý formulář můžeme rozdělit do několika částí:

- *kód, který nás k testovanému formuláři dovede* – testovaný formulář může být hned na první stránce webové aplikace, ale také nemusí. Někdy je ve struktuře webu umístěn tak, že se k němu lze dostat až po patřičných úkonech, jako například identifikace uživatele.
- *Kód, který provádí test vstupů* – tato část kódu se pravidelně opakuje a to tak dlouho, dokud jsme nezadali všechny hodnoty, které potřebujeme otestovat. Skládá se z několika akcí:
    - zadej hodnotu,
    - odešli,
    - zkontroluj výsledek,
    - vrať se na původní stranu.
- *Ukončení testu* – každý test by měl vrátit aplikaci do původního stavu, to znamená, že je například nutné se regulérně odhlásit a zobrazit úvodní stránku, smazat hodnoty z databáze, apod. Tato část je důležitá zejména v případě, kdy se má spustit další test, který počítá s výchozím nastavením.

Navíc by každý test měl poskytovat přehledný log a mělo by být možné připojit dodatečné informace o struktuře testu.

5 ZPRACOVÁNÍ PROGRAMU NA OTESTOVÁNÍ VSTUPU

Pro vytvoření aplikace jsem zvolila programovací jazyk Java, vývojové prostředí NetBeans a seleniové knihovny.

5.1 Logování testů

Každý testovací nástroj by měl poskytovat možnost logování, přičemž log by měl být přehledný a dobře srozumitelný.

Balík *Selenium* obsahuje třídu *DefaultSelenium*, která spouští browser, posílá mu příkazy a na konci testu browser zavře. Je to základní třída pro psaní JUnitových testů.

5.1.1 Třída DefaultSeleniumHTMLLog

Tuto třídu jsem rozšířila ze dvou důvodů. Jedním z nich je přidání příkazu zápisu do logu zároveň s provedením každé akce a druhým důvodem je snaha o zkrácení kódu. Třída *DefaultSelenium* má metodu *click()*, která vyvolá kliknutí na určitý objekt v prohlížeči. Tato metoda funguje výborně, pokud chceme zaškrtnout například checkbox nebo jiný objekt který nevyvolá načtení nové stránky. Pokud bychom chtěli kliknout na link, museli bychom za metodu *click()* přidat ještě metodu *waitForPageToLoad()*. Vzhledem k tomu, že při pohybu na webových stránkách velmi často klikáme na objekty, které mají za úkol otevřít jinou webovou stránku, pak sjednocení těchto dvou metod do jedné zkrátí výsledný kód zhruba o 1/3 až o 1/2 řádků z celkového počtu.

Na následujícím obrázku je část kódu třídy *DefaultSeleniumHTMLLog*. Je zde vidět, že rozšiřuje třídu *DefaultSelenium*.

Metoda *waitForPageToLoad()* se používá tam, kde je nutné, aby test počkal na načtení nové stránky. Defaultní hodnota je nastavená na 30 sekund. Pokud tester potřebuje, může ji nastavit na vyšší, případně nižší hodnotu pomocí metody *setLoadTime()*.

Pokud se blíže podíváme na kód třídy *DefaultSelenium*, zjistíme, že všechny metody které vyžadují parametry, přijímají pouze parametry typu String. Týká se to i metod, kde bychom parametr String neočekávali, například právě zmiňované metody *waitForPageToLoad()*.

Třída *DefaultSeleniumHTMLLog* z tohoto hlediska nic nemění.

```
public class DefaultSeleniumHTMLLog extends DefaultSelenium implements SeleniumHTMLLog {
    private HTMLLogger log;
    private String loadTime = "30000";

    public DefaultSeleniumHTMLLog(String rcUrl, int loadTime, String browser, String baseUrl) {
        super(rcUrl,loadTime,browser,baseUrl);
    }

    // zde nastavime dobu po kterou bude test cekat nez se nacte nova stranka
    public void setLoadTime(String loadTime) {
        this.loadTime = loadTime;
    }

    // zde nastavime log - trida HTMLLogger definuje do ktereho souboru
    // se budou zaznamy ukladat a tzv. level (uroven) logovani
    public void setHTMLLog(HTMLLogger log) {
        this.log = log;
    }

    public void clickAndWaitLog(String kam, String logText) {
        this.click(kam);
        this.waitForPageToLoad(loadTime);
        log.info("clickAndWait - " + kam + " - " + logText);
    }

    public void clickAndWaitLog (String kam) {
        this.click(kam);
        this.waitForPageToLoad(loadTime);
        log.info("clickAndWait - " + kam);
    }
```

Obrázek 23: Část kódu třídy DefaultSeleniumHTMLLog.java

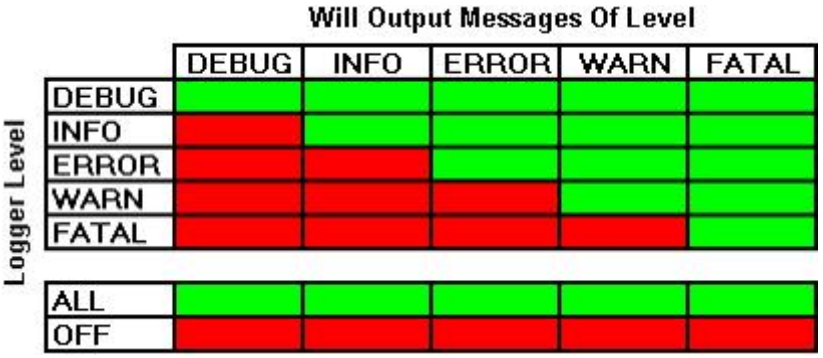
5.1.2 Třídy HTMLLogger. java a SeleniumHTML.java

Třída *HTMLLogger.java* se použije k nastavení logu – konstruktoru třídy je potřeba předat cestu k adresáři, ve kterém pak třída vytvoří soubor, do kterého bude ukládat logované informace. Tomuto konstruktoru je též nutno předat název logované třídy a level (úroveň logování). Při psaní této třídy jsem použila třídy z balíků apache *Log4J.FileAppender*, *Log4J.Level* a *Log4J.Logger*.

Konstruktor třídy *HTMLLogger.java* vytvoří instanci třídy *Log4J.Logger* a instanci třídy *Log4JFileAppender*, přičemž instanci třídy *Log4J.FileAppender* následně k nově vytvořenému loggeru přiřadí. Poté nastaví level (úroveň logování). Úrovně logování jsou definovány ve třídě *Log4J.Level*.

V této třídě je definováno sedm úrovní logování. Jsou seřazeny v určitém pořadí, a výběrem jedné určujeme i to, zda se budou či nebudou logovat ostatní úrovně.

Úrovně i logika jejich logování jsou zřejmé z následujícího obrázku:[8]



Obrázek 24: Úrovně logování

Z obrázku je vidět, že krajními úrovněmi jsou úrovně ALL a OFF. Pokud zvolíme úroveň ALL, bude se logovat vše. V případě úrovně OFF se nebude logovat nic. Jestliže například zvolíme úroveň ERROR, budou se logovat úrovně FATAL, WARN a ERROR.

Při testování jsem ale pocítila potřebu ještě minimálně jedné úrovně. A to z důvodu lepší přehlednosti logu. Ve formuláři na webové stránce bývá často několik vstupů. Pro případ, že určitá hodnota v testu „projde“ pak pro zápis do logu je vhodné použít level INFO. Jestliže nastane chyba, pak používám level ERROR. Test pro jeden formulář kontroluje několik vstupů a zápisy se do logu ukládají za sebou. Na každý vstup připadá několik řádků. Log tak může nabýt poměrně značné velikosti a vyhledávání pak zabere dost času. Proto jsem vytvořila třídu *LevelNadpis* která je rozšířením třídy *Level* a definuje úroveň NADPIS. Její instance mají nastavenou úroveň logování na hodnotu 1500, která se nachází mezi hodnotami definovanými pro úroveň DEBUG a INFO.

Dále jsem ještě pro zjednodušení čtení logu vytvořila třídu *LevelKonec*, která taktéž rozšiřuje třídu *apache.log4j.Level* a přidává další úroveň. Tuto úroveň používám pro značení konce testu. Jestliže otevřu log a vidím tam text této úrovně, pak bez dalšího zkoumání logu okamžitě vím, že test doběhl až do konce. Této úrovni jsem nastavila hodnotu 4500, která se nachází mezi hodnotami definovanými pro úroveň ERROR a FATAL.

---

8 Převzato z [24]

5.1.3 Třída HTMLLayoutCSS

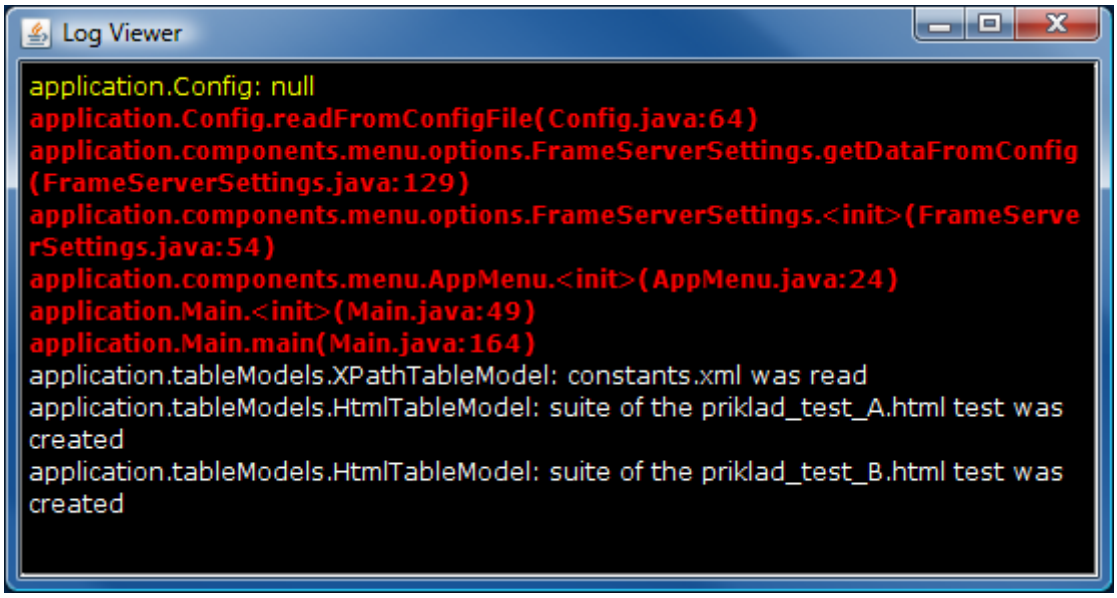
Třída *HTMLLayoutCSS* je též určená pro vylepšení logování a rozšiřuje třídu *Layout*.

Obecně řečeno, třídu *Layout* rozšiřují třídy, které mají za úkol formátovat logované informace. Existují již hotové třídy, například *PatternLayout*, *HTMLLayout*, *SimpleLayout* a *XMLLayout*, ze kterých si můžeme vybrat tu, která nám nejlépe vyhovuje.

Chtěla jsem, aby se log ukládal do html souboru, ale způsob formátování definovaný v *HTMLLayoutu* mi vyhovoval pouze částečně, neboť formátování výsledného html souboru je pevně dané a není možné ho měnit podle potřeby. Proto jsem se rozhodla vytvořit třídu *HTMLLayoutCSS*. Pro vytvoření této třídy jsem použila kód třídy *HTMLLayout* s tím, že jsem poupravila ty části, které se týkali pevně nastaveného formátování výsledného souboru a přidala jsem link na soubor css, ve kterém si můžu formátování html logu upravit kdykoliv podle svých potřeb.

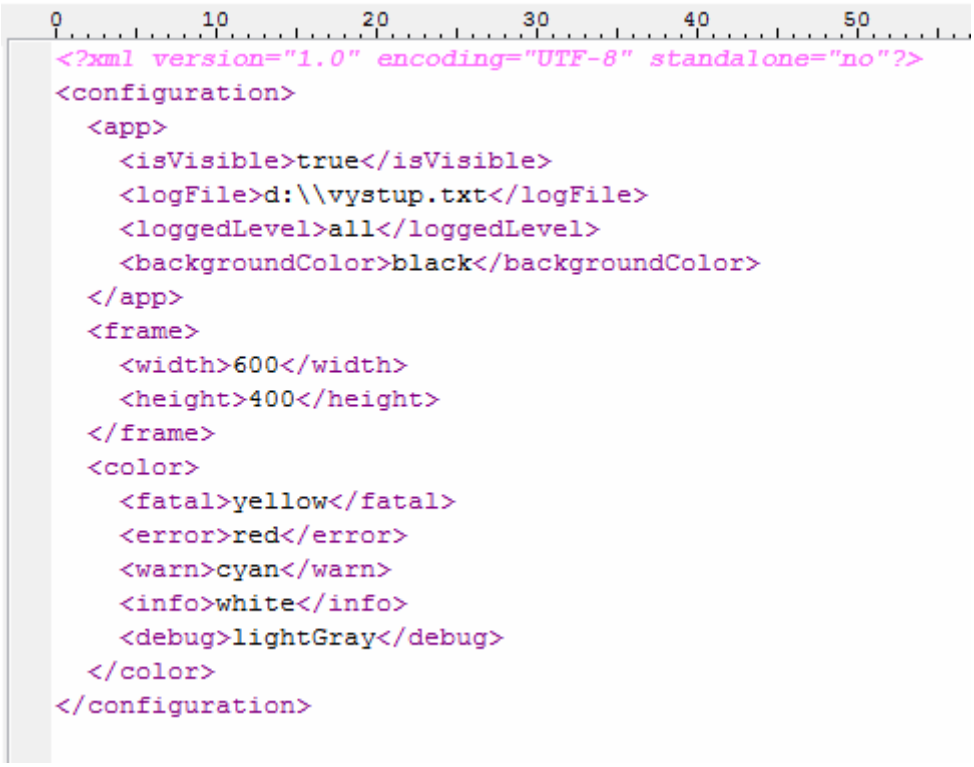
5.2 Logování testující aplikace

Stejně jako každý software, tak i software určený pro vytváření, spouštění či správu testů může obsahovat chyby. Proto jsem vytvořila nástroj *LogViewer*, který jsem připojila k aplikaci pro otestování vstupů. Pokud v aplikaci nastane chyba (chyba v aplikaci, která spouští testy, nikoliv chyba v testované aplikaci), provede se záznam do logu.



Obrázek 25: Screenshot nástroje LogViewer

LogViewer je jednoduchý nástroj a lze ho připojit k jakékoliv jiné javové aplikaci. Skládá se ze dvou tříd a jednoho konfiguračního souboru, kde lze nastavit několik parametrů. Obsah konfiguračního souboru je znázorněn na následujícím obrázku:



```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<configuration>
  <app>
    <isVisible>true</isVisible>
    <logFile>d:\\vystup.txt</logFile>
    <loggedLevel>all</loggedLevel>
    <backgroundColor>black</backgroundColor>
  </app>
  <frame>
    <width>600</width>
    <height>400</height>
  </frame>
  <color>
    <fatal>yellow</fatal>
    <error>red</error>
    <warn>cyan</warn>
    <info>white</info>
    <debug>lightGray</debug>
  </color>
</configuration>
```

Obrázek 26: Obsah konfiguračního souboru

Popis parametrů:

- *isVisible* – nástroj LogViewer použijeme v aplikaci a tímto parametrem určíme, zda se jeho okno při startu logované aplikace otevře. Defaultní hodnota je nastavena na *true*, protože pokud nástroj k nějaké aplikaci přidáme, pak většinou z důvodu, aby se okno otevřelo a zobrazovalo obsah logu. Nicméně je možné zobrazování vypnout.
- *logFile* – soubor, do kterého se budou logované zprávy ukládat. Je potřeba zadat celou cestu a název souboru.
- *loggedLevel* – úroveň logování. Nástroj LogViewer používá knihovny log4j, které definují pět úrovní logování. V tomto případě nelogujeme výstup testované aplikace, logování probíhá náhodně (ne podle očekávaného scénáře jak tomu bývá u logů z testů), takže úrovně, které jsem vytvářela pro testy, není možné zde použít a ty existující, které nabízí log4j jsou podle mého názoru dostačující.
- *backgroundColor* – defaultní barva pozadí je černá, ale je možné ji změnit.

- *frame.width* – určuje šířku okna loggeru.
- *frame.height* – určuje výšku okna loggeru.
- všechny hodnoty v sekci *color* – určují barvy záznamů jednotlivých úrovní. Uživatel si je může pozměnit podle potřeby.

Pouze dva parametry se týkají souboru, do kterého se logovací výstup ukládá. A to úroveň logování a název a cesta k souboru. Ostatní parametry se týkají nastavení okna, které se se startem logované aplikace zobrazí a za běhu aplikace vypisuje logovací hlášky. Výstup v textovém souboru se neformátuje.

5.2.1 Přidání LogVieweru do aplikace

Instanci LogVieweru je nutno vytvořit v každé třídě, která se má logovat. Stačí přidat do kódu dotyčné třídy tento příkaz:

```
LogViewer log = LogViewer.getInstance();
```

Třída LogViewer.java je navržená jako singleton, takže instance se ve skutečnosti vytvoří jen jedna, ostatní třídy už pak dostanou existující instanci.

Také je potřeba naimportovat knihovnu *LogViewer.jar*.

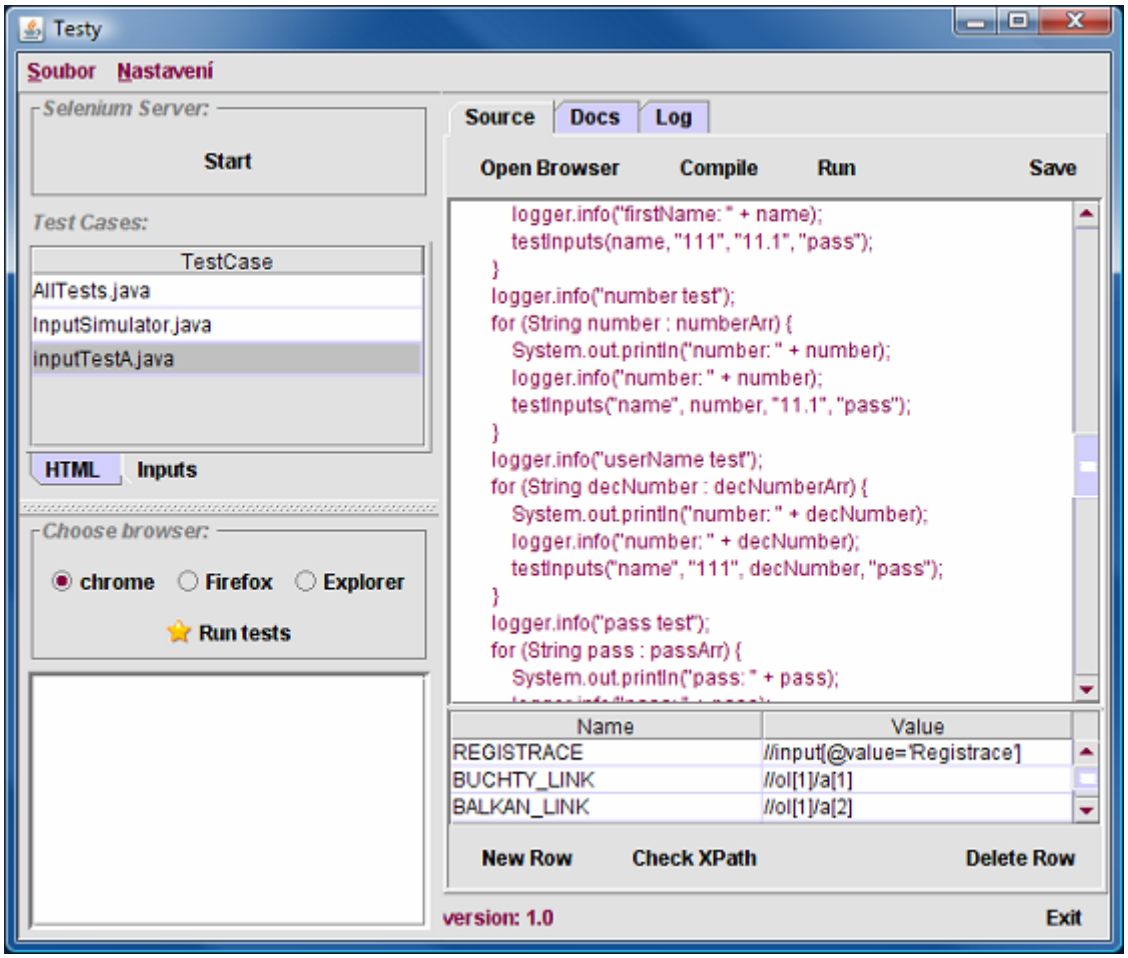
5.3 Popis testující aplikace

Aplikace pomáhá vytvářet testy a také je spravovat. Umožňuje vytvořit třídu na otestování vstupu tak, že vygeneruje část kódu, který si uživatel podle potřeby upraví. Poté je možné vytvořený kód zkompilovat a nakonec spustit. Pokud bychom chtěli do aplikace přidat hotový test, stačí ho nakopírovat do příslušného adresáře a aplikace si jej pak sama při spuštění načte.

Je potřeba, aby před instalací aplikace byl na počítači nainstalován JDK (Java Development Kit), což je balíček, který obsahuje soubor základních nástrojů pro vývoj v Javě. Aplikace využívá jeho překladač *javac* pro kompilaci a interpret *java* pro spuštění testů. Cestu k adresáři obsahující oba soubory (javac.exe i java.exe) je nutno zadat do systémové proměnné PATH.

V aplikaci je možné upravovat i spouštět testy uložené ve formátu html a java. Oba druhy testů jsou spouštěny pomocí příkazového řádku. Testy ve formátu java jsou pomocí

příkazového řádku také kompilovány. Aplikace umožňuje vytvoření pouze testů java, a to z toho důvodu, že pro vytvoření html testů je výhodnější použít nástroj *Selenium IDE*. Tyto testy lze pak do aplikace dodat.



Obrázek 27: Ukázka aplikace

Na levé straně obrázku je umístěno tlačítko *Start*, které spouští seleniový server. Tento server je nutné spustit vždy předtím, než spustíme javové testy či chceme otevřít browser. Parametry seleniového serveru se nastavují v souboru config.xml.

Pod tlačítkem *Start* je umístěna komponenta JTabbedPane se záložkami HTML a Inputs. Na záložce HTML jsou umístěny testy ve formátu html a na záložce Inputs vidíme test inputTestA.java, třídu InputSimulator.java, kterou je též možné upravovat a třídu AllTests.java, což je *Test Suite*.

Níže je uložen panel pro výběr prohlížeče, ve kterém budou testy spuštěny. Za spuštění prohlížečů odpovídá třída SeleniumServer. Je nutno dodat, že některé verze těchto prohlížečů nelze spustit.

Pod panelem pro výběr prohlížeče se nachází textové pole pro zobrazení logu seleniového serveru.

Pravou polovinu aplikace zabírá komponenta JTabbedPane obsahující záložky *Source*, *Docs* a *Log*. Na panelu Source se zobrazuje zdrojový kód testu, který můžeme upravovat. Na panelu Docs je textové pole, do kterého můžeme psát poznámky k testu, například popis testu, popis jednotlivých kroků testu, či popis dat s kterými test pracuje, apod. Na záložce Log pak hledáme log testu kde je zaznamenán výsledek posledního testování.

5.3.1 Záložka Source

Na záložce source se na horním panelu nacházejí tlačítka:

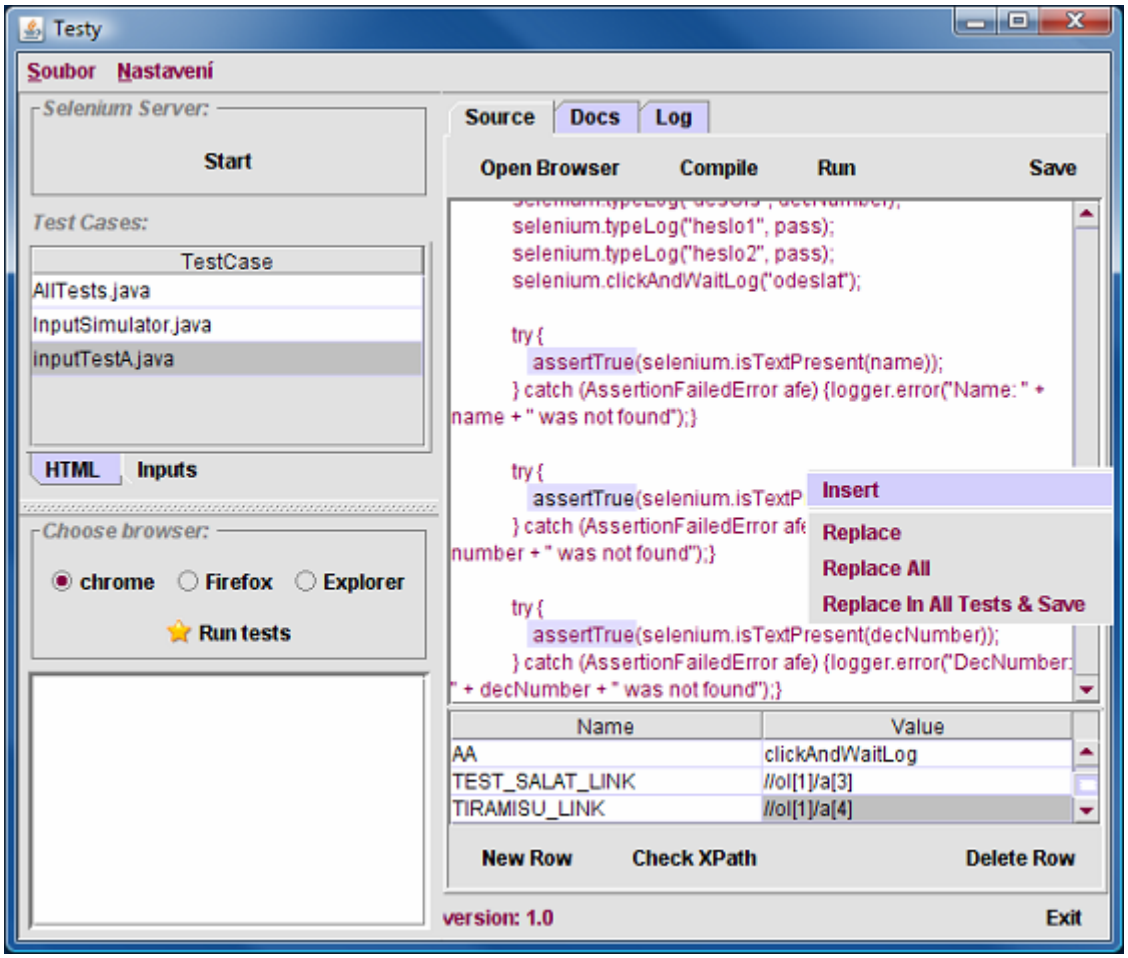
- *Open Browser* – jestliže je spuštěný selenium server, pak se otevře browser. Jinak se nestane nic. V otevřeném browseru pak můžeme kontrolovat existenci či správnost výrazů přepsaných do XPath. Pokud potřebujeme změnit URL, kterou browser po svém otevření načítá, přepíšeme ji v souboru config.xml.
- *Compile* – zkompiluje otevřený soubor. Musí se jednat o třídu napsanou v Javě, html testy se nekompilují.
- *Run* – tímto tlačítkem se spustí třída AllTests.java, což je testSuite, který postupně spustí všechny testy, které jsou v něm zadané
- *Save* – uloží změny, které v kódu provedeme

Pod těmito tlačítky se nachází editor zdrojového kódu. Zde můžeme testy upravovat. Aplikace obsahuje několik metod, které usnadňují úpravy textu. Tyto metody jsou vyvolány tlačítky, které nalezneme v menu, jež se objeví po kliknutí pravým tlačítkem do textové oblasti:

- *Insert* – vloží to textu řetězec znaků, který vybereme v tabulce pod editorem. Stačí označit příslušnou buňku v tabulce. Vkládat lze jak řetězce v kolonce „Name“, tak řetězce v kolonce „Value“.
- *Replace* – zamění označený řetězec v textu. V tabulce pod editorem označíme buňku která obsahuje požadovaný text, pak v editoru poznačíme text který chceme nahradit, následně použijeme tlačítko Replace

- *Replace All* – funguje jako předchozí s tím rozdílem, že nahradí všechny výskyty v dokumentu
- *Replace All & Save* – nahradí vybraný řetězec ve všech testech, nejen v otevřeném. Změny v otevřeném testu uloží.

Kromě výše uvedeného je zde ještě implementována funkcionalita označení všech výskytů vybraného řetězce v dokumentu.

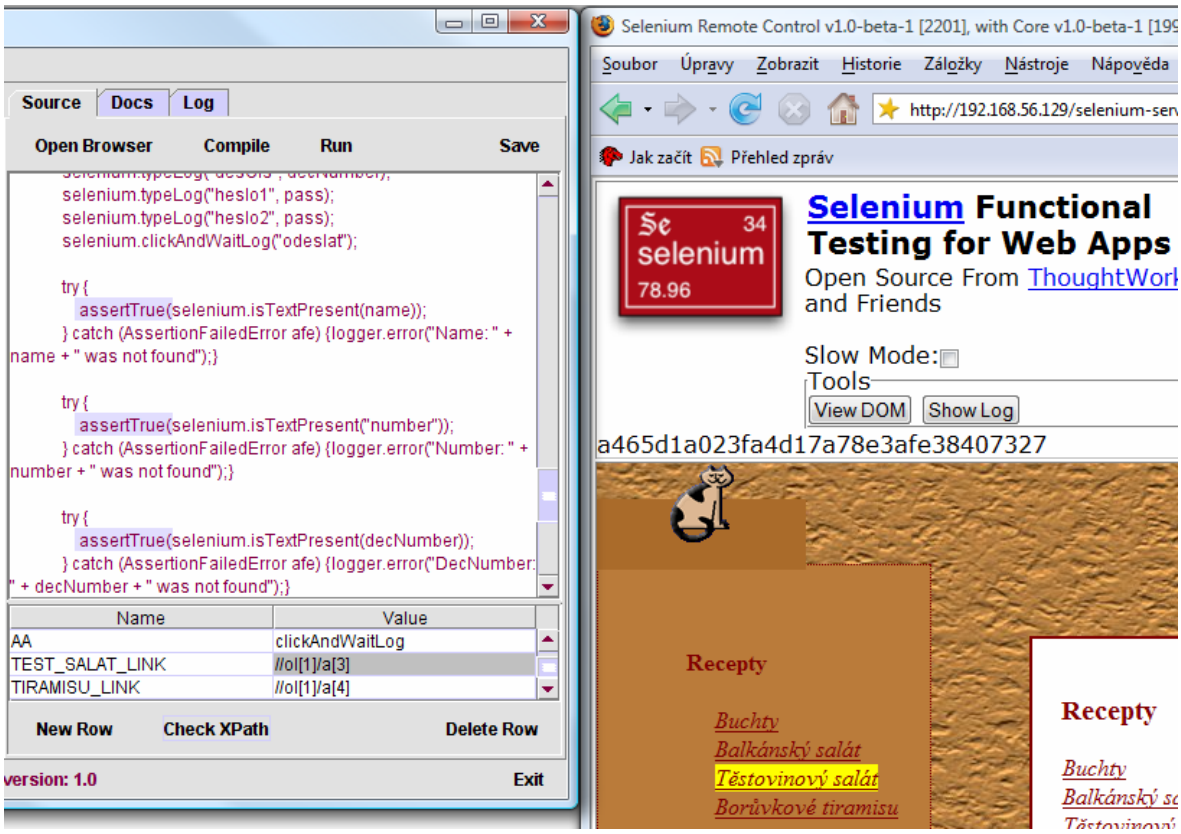


Obrázek 28: Ukázka editace

Ve spodní části panelu Source je tabulka se sloupci *Name* a *Value*. Tabulka byla implementována z toho důvodu, že při vytváření nebo editování testů týkajících se stejné aplikace, téměř neustále upravujeme stejné výrazy. Jedná se zejména o výrazy zaznamenané testovacím nástrojem, které ale následně přepisujeme do XPath. Tabulka slouží jako úschovna těchto výrazů, které můžeme později použít v jakémkoliv testu.

Pod tabulkou jsou dvě tlačítka, jejich funkce je zřejmá: tlačítko *New Row* přidává do tabulky nový řádek, který pak podle potřeby editujeme. Tlačítko *Delete Row* maže vybraný řádek.

Tlačítko *CheckXPath* umí zkotrolovat zda napsaný XPath výraz existuje. Nejprve musíme otevřít browser (předtím ale ještě spustit selenium server), pak označit XPath který chceme ověřit a následně tlačítko CheckXPath použijeme. Jestliže je XPath výraz správný, pak se příslušný objekt v browseru žlutě podsvítí. Na následujícím obrázku vidíme podsvícený odkaz s textem „Těstovinový salát“.



Obrázek 29: Ukázka podsvícení objektu nalezeného pomocí XPath

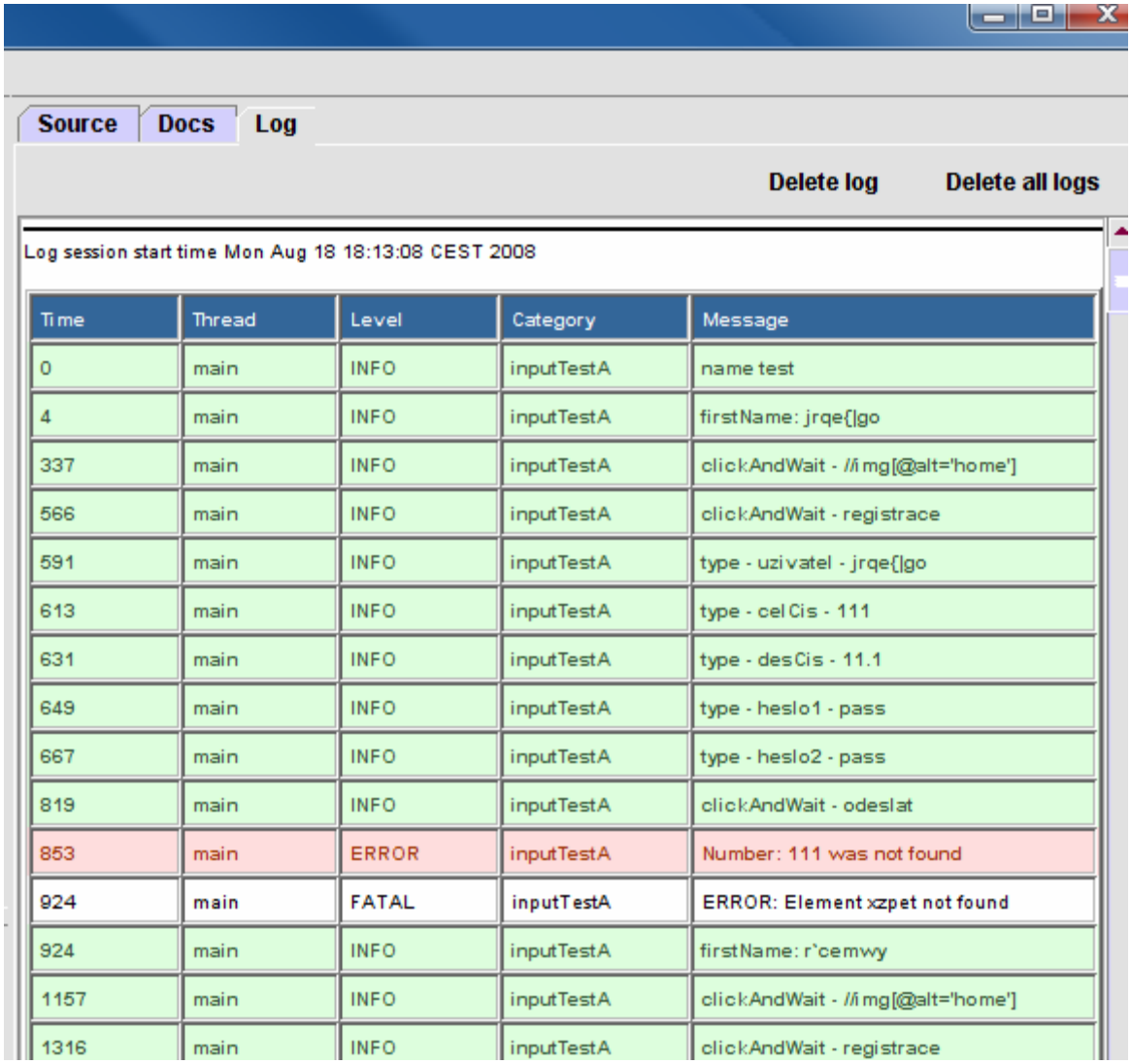
5.3.2 Dokumentace k jednotlivým testům

Je vhodné si ke každému testu vytvořit několik poznámek, které test charakterizují. Tyto poznámky by se neměly psát přímo do testu, proto aplikace pro každý test vytvoří nový dokument. Uživatel může tento dokument vyplnit, ale také nemusí. Vyplnění dokumentu není povinné a běh testu neovlivní.

Dokumenty k jednotlivým testům nalezneme na záložce *Docs*. Zde můžeme psát cokoliv, co uznáme za vhodné. Formátování není možné.

5.3.3 Logování testů

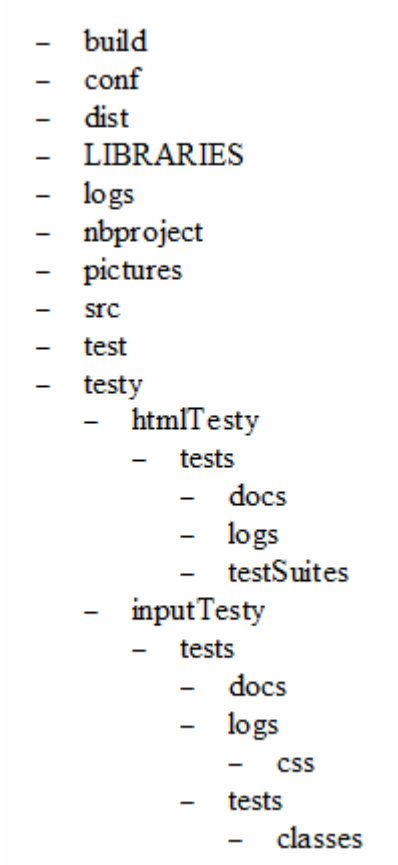
Logy jednotlivých testů nalezneme na záložce *Logs*. V logu bude vždy výsledek posledního spuštění testu. Pokud je test nový a ještě nebyl spuštěn, pak aplikace místo logu vypíše hlášení: „Log does not exist“. Aplikace vypisuje stejné hlášení i u tříd, které samy o sobě netestují a logy mít nemohou. V současné době se jedná se o třídy *AllTests.java* a *InputSimilator.java*, ale časem mohou přibýt i další.



Obrázek 30: Ukázka logu

5.4 Adresářová struktura projektu

Projekt obsahuje mnoho adresářů, ve kterých jsou uloženy důležité soubory. Na následujícím obrázku je znázorněna celá adresářová struktura, ale vzhledem k množství těchto adresářů popíšu jen ty, které při používání aplikace potřebujeme znát:



Ilustrace 1: Adresáře

5.4.1 Adresář conf

v tomto adresáři jsou umístěny konfigurační soubory a také několik dalších souborů, které je rovněž možné upravovat:

config.xml – konfigurační soubor který patří přímo aplikaci. Nastavují se zde parametry seleniového serveru,

constants.xml – soubor, který obsahuje data zobrazená v tabulce XPath na panelu source,

inputTemplate.java – soubor, který aplikace používá při vytváření javových testů, je to šablona kterou lze upravovat, popřípadě nahradit jinou šablonou s tím, že název a typ souboru musí zůstat nepozměněn.

logError.html – to je soubor, který načte panel Logs, pokud k určité třídě neexistuje její vlastní log. Obsah tohoto souboru lze změnit.

LogViwerConfig.xml – to je konfigurační soubor k aplikaci LogViewer. Parametry lze podle potřeby upravit.

Nutno dodat, že všechny výše uvedené soubory můžeme podle potřeby editovat, ale nesmíme je smazat. Pak by aplikace nebo některé její části nefungovaly.

5.4.2 Adresář LIBRARIES

Do tohoto adresáře je nutné nakopírovat všechny knihovny, které javové testy používají. Pokud kompilace neproběhne a zároveň se v logu nezobrazí žádný jiný výstup kromě hlášení, že se kompilace spustila, pak chyba bude s velkou pravděpodobností v tom, že zde bude některá knihovna chybět.

5.4.3 Adresář testy.htmlTesty

Do tohoto adresáře ukládáme testy ve formátu html. Aplikace si je při startu sama načte. Aplikace je psaná pro testy html vytvořené nástrojem Selenium IDE a rovněž používá seleniové knihovny k jejich spouštění. Jiné testy pravděpodobně nebudou fungovat i přesto, že by možná mohly být načteny. Ukládání jiných souborů tedy není doporučeno.

5.4.4 Adresář testy.htmlTesty.tests.docs

V tomto adresáři se ukládají soubory, které obsahují dokumentaci k jednotlivým html testům. Tyto soubory aplikace vytváří automaticky. Pokud se soubor smaže, nic se nestane, aplikace si vytvoří nový, ale případná původní data se ztratí.

5.4.5 Adresář testy.htmlTesty.tests.logs

Do tohoto adresáře aplikace ukládá logy. Log může vzniknout až poté, co byl test spuštěn. Jestliže test spustíme několikrát, pak se log vždy přepíše novou verzí a starý se neukládá. Logy také můžeme mazat ale s tím, že data, která jsou v nich uložená, ztratíme. Aplikaci

neexistence logu nevadí, jestliže ho nenajde, pak zobrazí soubor logError.html z adresáře conf.

5.4.6 Adresář testy.htmlTesty.tests.testSuites

Adresář je určen pro ukládání Test Suitů patřících k jednotlivým html testům. Tyto Test Suity si aplikace vytváří sama. Případné smazání by nemělo vadit, pokud se nebude provádět za běhu aplikace.

5.4.7 Adresář testy.inputTesty.tests.docs

Zde se ukládá dokumentace k jednotlivým javovým testům. Pokud nám nebudou chybět data v těchto souborech, můžeme je smazat. Aplikace si vytvoří nové.

5.4.8 Adresář testy.inputTesty.tests.logs

Do tohoto adresáře se ukládají výsledky testování javových testů. Navíc je zde podadresář css, který obsahuje styly pro všechny tyto javové logy. Soubor se styly se jmenuje styl.css a lze ho podle potřeby editovat.

5.4.9 Adresář testy.inputTesty.tests.tests

Adresář obsahuje všechny javové třídy. Pokud vytvoříme nový javový test, uloží se sem. Jestliže budeme chtít přidat už hotový test, pak ho musíme nakopírovat také do tohoto adresáře. V adresáři je také třída AllTests.java, což je Test Suit, do kterého je potřeba zaznamenat všechny testy které chceme spustit. Tato třída se nesmí smazat, protože pak by testy nebylo možné spustit. Totéž platí o třídě InputSimulator. Pokud ji smažeme, tak všechny testy, které ji používají, nebude možné ani zkompilovat. Jednotlivé testy smazat lze, pokud už nejsou potřeba.

5.4.10 Adresář testy.inputTesty.tests.tests.classes

Do tohoto adresáře se ukládají zkompilované testy. Jestliže je smažeme, pak musíme soubor zkompilovat znovu. Tyto třídy pak aplikace načítá, aby testy mohla spustit.

5.5 Některé další charakteristiky aplikace

5.5.1 Třída InputSimulator

Třída *InputSimulator* poskytuje testovací hodnoty. Všechny její metody vrací pole rizikových hodnot, které je potřeba otestovat. Pole rizikových hodnot může být již nadefinované, anebo je k jeho získání je použit generátor.

Třída *InputSimulator* nabízí tyto metody:

- *getDecimalNumber(float min, float max)* – tuto metodu použijeme, pokud potřebujeme otestovat vstup desetinných čísel. Do parametru *min* zadáme minimální číslo, které hodláme testovat. Do parametru *max* zadáme naopak nejvyšší číslo. Metoda vrátí pole, které bude obsahovat všechny rizikové hodnoty, které do tohoto rozmezí spadají.
- *getDecimalNuber(double min, double max)* – jedná se o stejnou metodu, jako byla předchozí. Je to metoda přetížená. V tomto případě zadáváme hodnoty typu double.
- *getNumber(long min, long max)* – tato metoda funguje stejně jako dvě předchozí s tím, že jako parametry dosazujeme celá čísla. Vrací pole rizikových celých čísel patřících do příslušné množiny.
- *getText(int minLength, int maxLength, int words)* – tato metoda generuje hodnoty. Hodnotou se zde rozumí řetězec znaků. Délky řetězců se mohou lišit. Parametr *minLength* je celé číslo které značí minimální délku řetězce, parametr *maxLenght* značí naopak maximální délku. Generátor bude vracet řetězce znaků o délkách v rozmení parametrů *minLength* a *maxLenght*. Do parametru *words* zadáme počet slov které má generátor vytvořit.
- *GetBoolean()* - tato metoda vrací pole se dvěma hodnotami: *true* a *false*.

Aplikace používá seleniové třídy nebo třídy od nich odvozené. Jak již bylo zmíněno, všechny metody těchto tříd vyžadují pouze parametry typu String. Z tohoto důvodu všechny metody třídy InputSimulator hodnoty před vrácením překonvertují.

Při výběru kritických hodnot jsem vycházela z knihy Rona Patonna, *Testování Softwaru* [2]. Autor uvádí, že problematické jsou tzv. hraniční hodnoty a mocniny dvou:

- za *hraniční hodnotu* je považována nejmenší (nebo naopak největší) hodnota, kterou je ještě aplikace schopna zpracovat.
- Rizika čísel, která jsou *mocninami dvou*, vyplývají ze skutečnosti, že počítače pracují s binárními číslicemi.

Třída *InputSimulator* tedy obsahuje pole těchto hodnot seřazené podle velikosti čísel. Uvažuje se i se zápornými čísly. Po získání minimální a maximální hodnoty se vyberou všechny hraniční hodnoty v příslušném rozmezí.

Pro lepší pochopení a přehled přikládám tabulku softwarových mocnin dvou ze jmenované knihy:

Výraz	Interval nebo hodnoty
Bit	0 nebo 1
Nibble	0 - 15
Bajt	0 – 255
Slovo	0 – 65 535 nebo 0 – 4 294 967 295
Kilo	1 024
Mega	1 048 576
Giga	1 073 741 824
Tera	1 099 511 627 776

Tabulka 4: Softwarové mocniny

5.5.2 Struktura šablony pro vytváření testů

Aplikace vytváří nové javové testy podle šablony. Tato šablona je uložena v souboru *inputTemplate.java* v adresáři *conf*. Šablonu je možné upravit. Nicméně i po použití šablony není test funkční. Šablona poskytuje pouze část textu, který je společný pro všechny testy, jejichž úkolem je zjistit funkčnost vstupů webového formuláře. Šablona vytvoří jakousi „kostru" třídy, kterou je možno zkompilovat i spustit, přičemž po spuštění se nic neprovede.

Šablona zajistí:

- import potřebných knihoven,
- vytvoření instance logu (text, který se bude zaznamenávat, si pak uživatel píše sám),
- nastavení selenia, které je následně zajišťuje otevření browseru,
- vytvoření instance třídy InputSimulator(),
- ukázkové zavolání jejích metod (v testu je pak potřeba modifikovat podle potřeb),
- vytvoří metodu *testInputs()*, do které se vkládá kód, který se v testu neustále opakuje,
- v metodě *testInputs()* uvede příklad zachycení možných výjimek,
- vytvoří statickou metodu *suite()*, kterou pak volá třída *AllTests()*.

Šablona uvažuje s dvěma výjimkami, které se při testování vyskytují:

AssertionFailedError

je výjimka, která se vyskytne v případě, že očekávaná hodnota se liší od zjištěné. Je vyvolána například metodami *assertTrue()* nebo *assertFalse()*.

Výjimka AssertionFailedError pochází z balíku junit.framework.

SeleniumException

je výjimka, která se vyskytne v případě, že objekt na webové stránce nebyl nalezen. Tato výjimka může být způsobena tím, že se objekt na stránce skutečně nenachází, nebo tím, že je špatně zadaný XPath výraz.

Výjimka SeleniumException pochází z balíku com.thoughtworks.selenium.

5.6 Vysvětlení pojmů

Během psaní této práce se vyskytlo několik situací, kdy jsem potřebovala použít určitý výraz, ale nenašla jsem prostor pro jeho dostatečné vysvětlení (případě jsem prostor našla, ale až o několik kapitol dále). Ráda vše napravila a shrnula v této kapitole.

Testovací případ	Test, který reprezentuje skupinu podobných testů. Pro vytvoření testovacího případu se rozhodneme tehdy, jestliže provedení všech možných testů by bylo neúměrně časově náročné, případně vůbec neuskutečnitelné.
Featura	Funkce nebo určitá vlastnost produktu. Jde o výraz, který se velmi často používá ve firmách zabývajících se vývojem softwaru. Zpravidla se tím myslí jednotka, kterou je potřeba samostatně implementovat a otestovat a zdokumentovat.
Funkční požadavky	Požadavky na funkci softwaru. Jsou to požadavky, kvůli kterým byla aplikace implementována. Seznam těchto požadavků dostaneme, pokud se pokusíme odpovědět na otázku „co by aplikace měla umět“.
Ne-fukční požadavky	Jsou to vlastnosti softwaru a požadavky které by měla aplikace splňovat. Seznam těchto požadavků dostaneme, pokud odpovíme na otázku „jaká by aplikace měla být“.
White-Box testovací technika	Testovací technika, při které máme přístup ke zdrojovému kódu. Ze zdrojového kódu můžeme zjistit, jaký výstup dostaneme, pokud vložíme určitý vstup.
Black-Box testovací technika	Testovací technika bez přístupu ke zdrojovému kódu. Zde vkládáme hodnoty podle určitých pravidel nebo intuitivně a čekáme, jaký výstup dostaneme.
Bug	Označení pro chybu. Je to výraz, který je velmi často používán ve firmách zabývajících se vývojem softwaru.

	Zpravidla označuje chybu v aplikaci či v některé její funkci.
Produktová specifikace	To je specifikace, která by měla obsahovat funkční i nefunkční požadavky. Tyto požadavky je vhodné získat přímo od zákazníka. Tester z této specifikace zjistí, co může považovat za chybu a co naopak chybou není.
Test Suit	Soubor, který je zodpovědný za spouštění testů. Každý test je zpravidla uložen v samostatném souboru. Do Test Suitu se pak zapíší odkazy na soubory obsahující testy a spustí se pouze Test Suite.
Milestone	Milestone je milník pro provedení plánovaných akcí. Je to zpravidla časový termín, do kdy má být dokončen vývoj, testování, dokumentace, atd. určitého, předem definovaného seznamu featur.
Enhancement	Návrh na rozšíření aplikace. Například přidání nové, dosud neexistující featury či vylepšení určitých vlastností aplikace
Sociální inženýrství	Je metoda, kterou používají útočníci, aby se dostali citlivým informacím, například k heslům oběti, apod. Používají se různé podvodné triky s cílem přesvědčit oběť, aby tyto informace poskytla dobrovolně. Oběť zpravidla netuší, že informace poskytuje útočníkovi.
Útok hrubou silou	Útok hrubou silou si lze představit jako pokus o rozluštění hesla. Útočník zkouší všechny možné kombinace čísel či znaků tak dlouho, dokud se prolomení nezdaří.
Slovníkový útok	Postup útoku probíhá stejně jako u útoku hrubou silou, slovníkový útok ale nepracuje s kombinacemi znaků, nýbrž používá kombinace předem vybraných slov, které jsou uloženy v tzv. slovníku.

ZÁVĚR

Mým původním přáním a záměrem bylo vytvořit aplikaci, které by se zadalo URL testované stránky, a ta by všechny inputy automaticky ověřila. Toto řešení by ale nebylo možné použít pro stránky, ke kterým webová aplikace umožní přístup pouze po zadání hesla, případně jiných údajů. Nicméně i pro ostatní stránky by takové řešení bylo dosti komplikované z toho důvodu, že aplikace může na různé vstupy reagovat různým způsobem. Tato komplikovanost mě odradila. Při testování ocením spíše jednodušší nástroje, u kterých je větší pravděpodobnost, že budou obsahovat minimum svých vlastních chyb.

Dospěla jsem tedy k závěru, že účast testera při vytváření testu bude stále nutná, nicméně jeho práci by bylo možné zjednodušit. Navíc jsem na trhu nenašla nástroj, který by umožňoval správu naprogramovaných testů zároveň se správnou jednodušších testů vygenerovaných některým z nástrojů, které fungují na principu záznamu aktivity uživatele. Přitom se tyto dva druhy testů v praxi velice často kombinují.

Pro rozhodnutí o zahrnutí obou problémů do jednoho projektu byla důležitá i skutečnost, že jsem stejně musela řešit otázku spravování vytvořených javových testů na ověřování inputu.

Přidání html testů do projektu znamenalo vytvoření několika tříd navíc, ale pravděpodobně důležitějším důsledkem je fakt, že projekt musel být napsán tak, aby byl z tohoto hlediska rozšiřitelný. Nemělo by být tedy velkým problémem k němu v budoucnu přidat i další typy testů.

CONCLUSION

In the beginning, my wish and my aim was to create an application, that would require only URL of a tested page to be able to complete its work. In another words, after getting URL, the application should be able to automatically test all inputs on the page. But such solution was not possible to use for pages, that is possible to enter only after the application gets password, eventually anothers data. Also, using this solution would be very complicated for another pages too, because an application can respond to various inputs by various ways. This complicacy dissuaded me. While testing, I prefer simplier tools where the probability of containing of minimum its own bugs is higher.

I persuaded that some tester's effort on creating a test still will be needed but it can be possible to simplify the work. Also, I did not find any tool that would allow to administrate programmed tests along with simplier tests that are generated by tools that record user's activity. Nevertheless, both kind of these tests are often combined.

Anyway, I had to solve administration of the tests that check inputs (and are written in java). This fact was important for my decision of including both problems into one project.

Addition html tests to this project involved creation of few more classes but probably the most important result was the fact that the project had to been written by the way that would allow to extend it. So, in the future, it should should be possible to add another kind of tests.

SEZNAM POUŽITÉ LITERATURY

[1] International Software Testing Qualifications Board, *Certified Tester – Foundation Level Syllabus,* verze 2007, (citace 2. 8. 2008). Přístup z Internetu: http://www.istqb.org/downloads/syllabi/SyllabusFoundation.pdf

[2] PATTON Ron. Testování softwaru. Přel. David Krásenský. 1.vyd. Praha: Computer Press, 2002, 313 s.

ISBN 80-7226-636-5

[3] LBMS, Techniky testování, (citace 2. 8. 2008)

Přístup z Internetu: http://www.lbms.cz/prezentace/TA-ukazka.pdf

[4] BECK Kent. Programování řízené testy. Přel. Slavoj Písek. 1.vyd. Praha: Grada Publishing, a.s., 2004, 204 s.

ISBN 80-247-0901-5

[5] Internet info, s.r.o., Pravidla přístupnosti, (citace 2. 8. 2008)

Přístup z Internetu: http://www.pristupnost.cz

[6] NIELSEN Jakob, Top ten mistakes of Web design, (citace 2. 8. 2008)

Přístup z Internetu: http://www.useit.com/alertbox/

[7] Info-Source.us, Information Technology and Search Engine Optimization, (citace 4. 8. 2008) , Přístup z Internetu:

http://www.info-source.us/system_assurance_software_quality_and_testing/

[8] ŠTRUPL Václav, Testování webových stránek, (citace 2. 8. 2008)

Přístup z Internetu: http://interval.cz/clanky/testovani-webovych-stranek/

[9] MATĚJKA Lukáš, Využití metod projektového řízení, (citace 6. 8. 2008)

Přístup z Internetu: http://is.muni.cz/th/49968/esf_b/bp_projriz.pdf

[10] PYROCHTA Tomáš, Navrhování testovacích prostředků a metod pro aplikace informačních systémů v prostředí klient/server. Edice PhD Thesis, sv. 146, Vyskoké učení technické v Brně, 2002,

ISBN 80-214-2231-9, ISSN 1213-4198

[11] Software Quality Management, (citace 6. 8. 2008)

Přístup z Internetu:

http://programminglarge.com/software%5Fquality%5Fmanagement/

[12] SKONNARD Aaron, GUDGIN Martin. XML pohotová refererrční příručka. Přel. Lucie Rút Bittnerová. 1.vyd. Praha: Grada Publishing, a.s., 2006, 344 s.

ISBN 80-247-0972-4

[13] BŘÍZA Petr, Základy jazyka XPath, (citace 8. 8. 2008)

Přístup z Internetu: http://interval.cz/clanky/zaklady-jazyka-XPath/

[14] NIC Miloslav, JIRAT Jiri, XPath Tutorial, (citace 8. 8. 2008)

Přístup z Internetu:

http://www.zvon.org/xxl/XPathTutorial/General_cze/examples.html

[15] HUSSEBY H. Sverre, Zranitelný kód. Přel. David Čepička. 1.vyd. Brno: Computer Press, a.s., 2006, 207 s.

ISBN 80-251-1180-6

[16] LOGN Johnny, Google Hacking. Přel. RNDr. Jan Pokorný. 1.vyd. Brno: ZONER software, s.r.o., 2005, 472 s.

ISBN 80-86815-31-5

[17] Nástroj Fiddler, (citace 10. 8. 2008)

Přístup z Internetu: http://www.fiddlertool.com/fiddler/

[18] KLOBASA Pavel, Obhajoba vodopádového vývoje, (citace 11. 8. 2008)

Přístup z Internetu: http://vyvojari.oxyonline.cz/vodopadovy-vyvoj-obhajoba

[19] Anotace obsahu normy ČSN ISO/IEC 14598-5, (citace 11. 8. 2008)

Přístup z Internetu: http://shop.normy.biz/detail-polozky.php?katcis=57557

[20] GRAPPONE Jennifer, GRADIVA Couzin, SEO – Optimalizace pro vyhledávače. Přel. Roman Skřivánek, Dana Balaštíková. 1.vyd. Brno: ZONER software, s.r.o., 2007, 328 s.

ISBN 978-80-86815-85-5

[21] WEIDA Petr, SEO – Search Engine Optimization, (citace 11. 8. 2008)

Přístup z Internetu: http://interval.cz/clanky/seo-search-engine-optimization/

[22] Search Engine Ranking Factors, (citace 11. 8. 2008)

Přístup z Internetu: http://www.seomoz.org/article/search-ranking-factors#f35

[23] Apache log4j, (citace 20. 8. 2008)

Přístup z Internetu: http://logging.apache.org/log4j/

[24] PALETA Petr, Co programáory ve škole neučí aneb softwarové inženýrství v reálné praxi. 1.vyd. Brno: Computer Press, a.s., 2003, 337 s.

ISBN 80-251-0073-1

[25] HAWLITZEK Florian, Java 2 – příručka programátora. Přel. Jiří Bráza. 1.vyd. Praha: Grada Publishing, spol. s r. o., 2002, 316 s.

ISBN 80-247-9060-2

[26] KISZKA Bogdan, 1001 tipů a triků pro programování v jazyce Java. 1.vyd. Brno: Computer Press, a.s., 2003, 519 s.

ISBN 80-7226-989-5

[27] DARWIN F. Ian, Java – kuchařka programátora. Přel. Jan Gregor. 1.vyd. Brno: Computer Press, a.s., 2006, 798 s.

ISBN 80-251-0944-5

[28] SCHILDT Herbert, JavaTM2 – příručka programátora. Přel. Miroslav Dressler. 1.vyd. Brno: SoftPress s.r.o., 2001, 552 s.

ISBN 80-86497-04-6

[29] SPELL Brett, Java - Programujeme profesionálně. Přel. Bogdan Kiszka. 1.vyd. Praha: Computer Press, a.s., 2002, 1022 s.

ISBN 80-7226-667-5

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

HTML	**H**yper**T**ext **M**arkup **L**anguage je značkovací jazyk. V jednom textu je obsažen vlastní obsah i jeho formátování. K formátování se používají předem definované značky, tzv. tagy. Je využíván zejména na Iternetu.
XML	E**X**tensible **M**arkup **L**anguage. Je to jazyk podobný HTML s tím rozdílem, že primární funkcí značek je oddělovat data aby dokument mohl být snadno zpracováván počítačovými programy.
XHTML	E**X**ensible **H**yper**T**ext **M**arkup **L**anguage je také značkovací jazyk určený zejména pro Internet. Lze ho chápat jako HTML rozšířený o jazyk XML.
CSS	**C**ascading **S**tyle **S**heets je jazyk určený pro formátování dokumentů napsaných v jazycích HTML, XHTML, a XML. V tomto případě je ale obsah důsledně oddělen od formátování.
XPath	**X**ML **Path** Language je jazyk, který je určen pro práci s XML dokumenty. Umožňuje vyhledávání jednotlivých elementů uvnitř dokumentu.
HTTP	**H**yper **T**ext **T**ranser **P**rotocol – internetový protokol pro přenášení dokumentů po síti. Původně byl určen pro přenášení HTML dokumentů.
HTTPS	Je to nadstavba protokolu http, která umožňuje šifrování dat.
URL	**U**niform **R**esource **L**ocator – určuje adresu souboru nebo služby na Internetu
GET	Způsob odeslání dat prohlížečem serveru. Data se posílají jako součást URL
POST	Také způsob odeslání dat prohlížečem serveru. V tomto případě se data posílají v těle dotazu

SEZNAM ILUSTRACÍ

SEZNAM TABULEK

SEZNAM PŘÍLOH