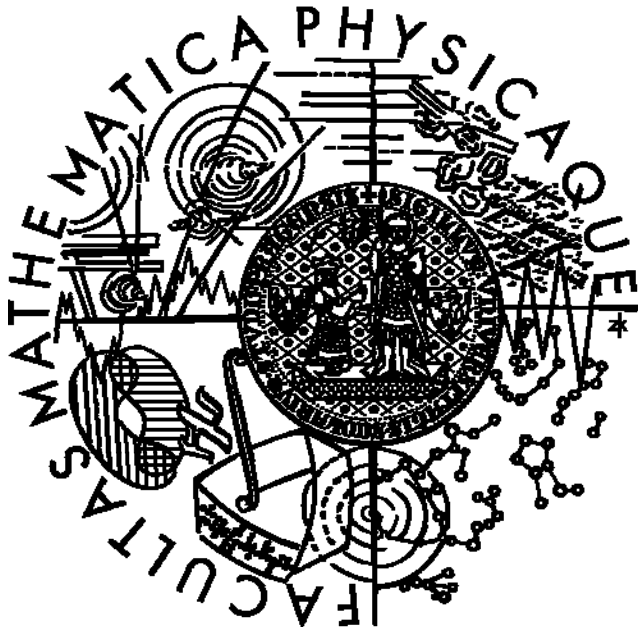
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Anna Borovcová

Testování webových aplikací

Katedra teoretické informatiky a matematické logiky
Vedoucí diplomové práce: Mgr. Vladan Majerech, Dr.
Studijní program: Informatika, obor softwarové systémy

Poděkování:
Ráda bych poděkovala Mgr. Vladanu Majerechovi za rady při vedení práce, Davidu Havlíčkovi za svolení k experimentálnímu testování jeho webových aplikací pro účely této práce a všem, kteří se mnou o testování diskutovali a odpovídali na mé dotazy.

Prohlašuji, že jsem svou diplomovou práci napsala samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne 5.8.2008 Anna Borovcová

…...……

Obsah

Obsah 3
Část I. – Úvod 6
1. Cíl práce 6
2. Historie 7
3. Základní kameny 8
4. Webové aplikace 11
5. Charakteristiky testování a vývoje webových aplikací 13
6. Metodiky pro web 16
6.1 Klasické metodiky a web 18
6.2 Agilní vývoj 20
6.3 Další metodiky 22
Část II. – Testování 25
7. O čem je testování 25
7.1 Definice 25
7.2 O čem je testování 26
8. Chyby 28
8.1 Motivace – některé známé chyby 32
9. Testovací tým 35
9.1 Vlastnosti testera 35
9.2 Pozice v testovacím týmu 37
9.3 Tester versus programátor 38
10. Kategorie testů 44
10.1 Statické a dynamické testování 44
10.2 Černá a bílá skříňka 44
10.3 Automatické a manuální testovaní 45
10.4 Stupně testování 45
10.5 Pokrytí testy 48
10.6 Dimenze kvality 50
11. Dokumentace 52
11.1 Nejdůležitější dokumenty podle praxe 54
11.2 Nejdůležitější dokumenty podle standardu 56
11.3 Testovací nápady 56
11.4 Reportování chyb 59
11.5 Metriky 62
12. Testovací cyklus 69
Část III. – Testy webových aplikací 70
13. Testy a techniky 70
13.1 Funkční testy: 70
13.2 Testy použitelnosti 73
13.3 Testy spolehlivosti 74
13.4 Výkonnostní testy 74
13.5 Testy podpory 75
13.6 Bezpečnostní testy 76
13.7 Další 76
14. Používání nástrojů 77
15. Automatizace funkčních testů 80
15.1 Canoo WebTest 81

15.2 JWebUnit ... 89
15.3 Rational Functional Tester ... 92
16. Automatizace výkonnostních testů ... 97
16.1 Apache JMeter ... 100
16.2 Grinder ... 105
16.3 LoadRunner ... 108
17. Bezpečnost webových aplikací ... 112
17.1 Běžné bezpečnostní problémy ... 114
17.2 Dělení bezpečnostních problémů ... 117
17.3 Proces testování bezpečnosti ... 118
17.4 Nástroje ... 120
Část IV. – Praktický příklad ... 121
18. Příklad testovací plánu ... 121
1) Testovaná aplikace ... 121
2) Cíl testování ... 122
3) Testovací přístup ... 122
4) Kritéria ne/připravenosti k předání ... 123
5) Zdroje ... 123
6) Role ... 124
19. Příklad testovacího scénáře ... 125
20. Příklad test result dokumentu ... 127
1) Souhrnné zjištění ... 127
2) Testy dokumentace ... 127
3) Odhad stavu komponent ... 128
4) Příprava testovacích scénářů ... 129
5) Zhodnocení použitého přístupu ... 129
Závěr ... 130
Seznam obrázků ... 131
Zdroje ... 132
Přílohy ... 139

Název práce: Testování webových aplikací
Autor: Anna Borovcová
Katedra (ústav): Katedra teoretické informatiky a matematické logiky
Vedoucí diplomové práce: Mgr. Vladan Majerech, Dr.
e-mail vedoucího: Vladan.Majerech@mff.cuni.cz
Abstrakt:
Předmětem diplomové práce je vysvětlení problematiky testování webových aplikací zejména potenciálním testerům. Práce proto srozumitelnou formou sděluje zkušenosti nasbírané z různých zdrojů a vychází z mojí tříleté praxe testování a roční praxe ve školení nových testerů převážně ve firmě Unicorn.
První část práce je věnována úvodu do webových aplikací, jsou zde zmíněny základní principy a historie internetu i proces vývoje a jeho metodiky. Druhá část se zabývá základními aspekty testování a jak je vidí různí zkušení testeři z České republiky i ze zahraničí. Třetí část vybírá a představuje některé techniky a nástroje vhodné pro testování webových aplikací. Čtvrtá část obsahuje praktickou ukázku některých testovacích dokumentů.
Klíčová slova: testování softwaru, webové aplikace, internet

Title: Web applications testing
Author: Anna Borovcová
Department: Department of Theoretical Computer Science and Mathematical Logic
Supervisor: Mgr. Vladan Majerech, Dr.
Supervisor's e-mail address: Vladan.Majerech@mff.cuni.cz
Abstract:
This diploma thesis matter is explanation of web application testing problems mainly to potential testers. That is why the thesis confides information gathered from different sources in intelligible way and it results from three years of testing and one year of teaching new testers mainly in Unicorn Corporation.
The first part is dedicated to introduction to web testing; basic principles and history of internet, development process and methodologies are mentioned there. The second part deals with core aspects of testing and how they are viewed by various experienced testers from Czech Republic or foreign countries. In the third part there are chosen and introduced some techniques and tools suitable for web application. The fourth part contains practical example of some testing documents.
Keywords: *software testing, web application, internet*

Část I. – Úvod

1. Cíl práce

Přestože v České republice působí hodně počítačových firem a rozsáhlou nabídkou knih s IT tématikou se můžou chlubit i malá knihkupectví, testováním softwaru se zabývá pouze jediná kniha vydaná v češtině. Testování softwaru od Rona Pattona [7] je přehledovou knihou a obsahuje hypotézy utvořené v minulosti. Bohužel většina další literatury, která byla napsána v předchozích letech v češtině si navykla přebírat pravidla a data z jiných zdrojů bez zamyšlení, zda platí i v současné době. Testování je složitou oblastí, s mnoha specifickými odvětvími, kterou se nevyplácí podceňovat ani zobecňovat.

S ohledem na aktuálnost diplomové práce jsem proto zvolila specifickou oblast testování webových aplikací, která v počátcích softwarového inženýrství neexistovala.

Tento problém nedostatečné literatury a s tím i nedostatečných znalostí testování ohrožuje řadu počítačových firem, které se mylně domnívají, že testování zvládají. Abych vyřešila tento křiklavý problém, stanovila jsem si tyto cíle mojí diplomové práce:

1) seznámit se se specifiky testování webových aplikací
2) vymezit úkoly testování v rámci vývoje
3) zamyslet se nad obecně známými poučkami ohledně testování v kontextu webových aplikací a různých přístupů dnešní doby
4) vysvětlit podle vlastních zkušeností z praxe, co vše testování obnáší
5) sepsat informace potřebné pro budoucí testery srozumitelně tak, aby mohly být předloženy jako školící materiál

2. Historie

Internet původně vznikl v šedesátých letech minulého století jako nekomerční vojenský projekt vedený agenturou ARPA (Advanced Research Project Agency). Tehdy byla tato síť, která měla umožňovat komunikaci na základě výměny paketů, označována jako ARPANET. Ovládání ARPANETU na tehdejších počítačích bylo složité a počítače byly velmi drahé, a tak měla tato síť od začátku sloužit spíše vojenským a výzkumným účelům. Postupně následovalo vytvoření emailu, tcp protokolu, domain name systému (DNS) a dalších utilit. Oblibu širší veřejnosti si získal internet ale až poté, co v roce 1989 vymysleli ve švýcarském CERNu World Wide Web, systém propojených hypertextových dokumentů.

Postupně více a více informací je zpřístupňováno na této síti a spolu s tím i stále větší procento lidí věnuje čas brouzdání na internetu namísto čtení papírových novin, časopisů nebo reklamních letáků. V současné době už internet najdeme všude tam, kde je i elektrický proud a v síti jsou tak propojeny miliardy počítačů po celém světě. Stále více aplikací se přenáší na web a už to nejsou čistě nekomerční informace, internet je už po několik let dobýván světem byznysu. Internet může sloužit nejen k propagaci firmy a prodeji zboží, ale firmy mohou na něm založit i celé své podnikání, tak je tomu například u Googlu, Aukra a dalších.

Internet je svobodná síť pro všechny, která nebyla navržena jako dostatečně bezpečná pro účely, ke kterým se dnes běžně používá. Rovněž znalosti a schopnosti uživatelů se liší, a to ve větší míře než jejich úmysly. Pokud uživatelé nebudou schopni v aplikaci najít potřebné informace nebo budou otráveni nevhodným uživatelským rozhraním, pravděpodobně už se nevrátí nebo dojde k nepříjemným vedlejším efektům. Vývoj webových aplikací proto přináší specifické požadavky a rizika, ale i výhody, kterých si testovací tým musí být vědom a patřičně z nich těžit.

3. Základní kameny

Chceme-li podrobně testovat webové aplikace, potřebujeme se něco dozvědět i o třech specifikacích, na kterých web stojí, a to jsou URI (URL), HTML a HTTP.

HTML celým názvem *HyperText Markup Language* je značkovací jazyk, který říká, co a jak má být na webové stránce zobrazeno. Existuje více jazyků, pomocí kterých můžeme obsah stránky ovlivňovat, ty však jsou pohodlným rozšířením, HTML zůstává základem, se kterým by se tester měl seznámit.

Příklad HTML kódu:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
<!-- Příklad html pro diplomovouu práci -->

<title> Příklad html </title>

</head>
<body>
<h1> Velký nadpis </h1>
<br />
Text text text text.....
</body>
</html>
```

URI (Uniform Resource Identifier) je obecný identifikátor zdroje reprezentovaného na webu, tedy jakýsi odkaz. Můžeme se setkat s různými druhy URI, ty jsou definovány svojí specifickou syntaxí a příslušným protokolem. URI tedy může ukazovat na webovou stránku, email, soubor, ale i na ISBN knížky. Podmnožinou URI je pak URL (Uniform Resource Locator), jenž navíc umožňuje přístup ke zdroji.

Příklad URI:

```
http://www.mff.cuni.cz/
ftp://somehost/resource.txt
urn:isbn:80-7226-636-5
```

HTTP (HyperText Transfer Protocol) slouží pro přenos hypertextových dokumentů, které jsou pro web tolik typické. HTTP je řádkově orientovaný bezstavový protokol a určuje syntaxi pro výměnu informací mezi jednotlivými uzly na internetu. Jedna strana, většinou uživatel, si vyžádá nebo zašle informace formou požadavku, na který druhá strana, server, zašle odpověď. Jak požadavek, tak odpověď mají společnou strukturu s řádky oddělenými od sebe netisknutelnými znaky CR a LF (\r\n). Na prvním řádku najdeme vždy typ akce říkající o jaký požadavek nebo odpověď se jedná. Následuje libovolný počet řádků hlavičky požadavku tvaru název: hodnota. Hlavička je ukončena prázdným řádkem, za kterým případně následují data. V následujícím příkladu je vidět, jak takový kompletní hypertextový dokument vypadá.

Příklad HTTP dokumentu požadavku:

```
GET / HTTP/1.1
Host: www.poeta.cz
Connection: close
User-Agent: Web-sniffer/1.0.27 (+http://web-sniffer.net/)
Accept-Encoding: gzip
Accept-Charset: ISO-8859-1,UTF-8;q=0.7,*;q=0.7
Cache-Control: no
Accept-Language: de,en;q=0.7,en-us;q=0.3
Referer: http://web-sniffer.net/
```

Příklad HTTP dokumentu odpovědi:

```
HTTP/1.1 200 OK :
Date: Fri, 11 Jul 2008 18:21:14 GMT
Server: Apache
Set-Cookie: PHPSESSID=3d7a5bc1e075883f309f5330d4ab03d7; path=/
Expires: Fri, 11 Jul 2008 18:04:34 GMT
Cache-Control: no-cache
Pragma: no-cache
Connection: close
Content-Type: text/html; charset=utf-8

<?xml version="1.0" ?><!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1
Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
```

```
<title>Poeta.cz - moderní literární server</title>
....
```

S informací, co tyto tři základní specifikace znamenají, a jak vytvořit jim vyhovující či naopak nevyhovující příklady, je jednoduché jejich instance získávat z webové aplikace a libovolně pozměnit. Z hlediska webové aplikace tedy není žádný klient bezpečný. Pokaždé, když data jsou odeslána z klienta na server, je třeba je zkontrolovat, i kdyby už předtím zkontrolována byla. Uživatel tato data může změnit přímo v kódu. Sverre H. Huseby tento přechod mezi serverem a klientem nazývá neviditelná bezpečnostní bariéra. Ideální je přistupovat k webové aplikaci velmi kriticky a při testování bezpečnosti si hlídat tuto bariéru pomocí zobrazování zdrojového HTML kódu, provádění změn a experimentů s URI a HTTP požadavky.

4. Webové aplikace

Jak je možno nalézt ve Wikipedii [47], webová aplikace se vyznačuje tím, že je poskytována uživatelům z webového serveru přes internet nebo případně jeho vnitropodnikovou obdobu intranet. Nejvýraznějším charakteristickým prvkem i největší výhodou je používání webového prohlížeče jako takzvaného tenkého klienta.

Takováto široká definice plně vyhovuje záměru této diplomové práce, v praxi se však setkáváme s různě složitými typy webových aplikací, což je třeba si uvědomovat a automaticky nepředpokládat, že co platí u jednoho typu, se bude stejně chovat i u ostatních webových aplikací.

V zásadě se podle složitosti dají webové aplikace rozdělit na:

Webové stránky – Webové stránky si může vytvořit každý, kdo jen trochu ovládá počítač a HTML. Jedná se hlavně o stránky osobní, firemní, zájmové a podobně, přitom se nemusí jednat o statické stránky malého rozsahu, ale naopak mohou využívat spoustu triků vytvořených za pomocí různých technologií a pracovat ve velmi malém rozsahu i s databází. Hlavním smyslem pro vytváření webových stránek je poskytování informací. Kvalita těchto stránek odráží kvalitu dostupných knih o tvorbě webových aplikací. Bohužel tyto knihy předkládají příklady bez vysvětlení jejich bezpečnostních nedostatků a bez popsání vhodných způsobů testování. Navíc tvorbu takovýchto stránek zvládne s přehledem jeden člověk, z tohoto důvodu nejsou dostatečně otestované nebo udržované.

Internetové aplikace – Internetové aplikace, už obsahují kromě webových stránek logiku srovnatelnou s běžnými počítačovými aplikacemi. Většinou se jedná o propracované aplikace, na kterých se podílel tým lidí. Patří zde redakční systémy, chaty, emaily a internetové obchody. Na rozdíl od předchozí skupiny internetové aplikace poskytují spíše služby. Kvalita se výrazně různí v závislosti na zkušenostech vývojového týmu. Pokud je taková aplikace psána pouze na základě zkušeností z knih určených přímo jen pro tvorbu webových stránek, je to receptem na katastrofu. Začínají být už velmi patrné chyby při nedostatečném zabezpečování a testování. U hodně důležitých systémů je proto nezbytný vhodně složený tým testerů s dostatečnými zkušenostmi. Podobně je to jistě i u vývojového týmu, jehož členové budou specializovaní.

Softwarové systémy s tenkým klientem – Složité komplexní informační systémy často vytvářené na míru konkrétnímu zadavateli získávají díky tenkému klientovi větší flexibilitu a stávají se rovněž speciální skupinou webových aplikací. Tyto systémy bývají trvale udržované, poskytují větší rozsah služeb a často řídí určitou část byznysu zadavatele. Takovéto kritické systémy se nedají vyvíjet bez zkušených lidí a jejich vývoj trvá řadu měsíců.

5. Charakteristiky testování a vývoje webových aplikací

Bohužel neexistuje jeden nejlepší testovací přístup, souhrn postupů a druhů testů, který by nám umožnil levně a přitom dostatečně otestovat každou aplikaci, dokonce ani pokud se omezíme na jeden konkrétní druh aplikací, nenajdeme ten jeden nejlepší způsob. Testy odrážejí potřeby různých osobností, jimž má aplikace sloužit, jako vlastnosti autorů, komponent a prostředí, do kterého je vyvíjena. Můžeme si ale všimnout, že čím více konkretizujeme aplikaci, tím více si můžeme alespoň mlhavě představovat její možné otestování. Protože pojem webových aplikací je stále ale příliš široký, ne vždy budou následující předpoklady platit. Mají spíše většinový význam.

Webové aplikace se od těch desktopových liší zejména v důsledcích používání prohlížeče, ten má mnoho výhod, ale taky nedostatků. Právě tyto specifické vlastnosti nám poskytují vodítko k tomu, na co bychom si měli dávat pozor a čeho bychom měli využívat při plánování testů.

Jeden z prvních rozdílů, který většinu lidí napadne, je **větší důraz na vzhled a použitelnost**. Lidé mají na stránkách větší volnost pohybu, než u firemních systémů nebo předem zakoupených aplikací. U běžných aplikací použitelnost ovlivňuje rychlost a kvalitu práce, u webových aplikací se častěji lidé rozhodují, zda se na stránky vůbec vrátí. Toto úzce souvisí i s tím, že lidé mají různé zvyky práce s internetem, mají zafixované jiné orientační body a používají rozličný hardware a software. Z toho vyplývá důraz na testy kompatibility, aplikace se zkoušejí s různými prohlížeči, při různém rozlišení, barevném spektru, se vším, s čím by aplikace měla spolupracovat. Testeři rovněž potřebují sledovat uživatele při práci s webovou aplikací, aniž by se jim snažili říkat, jak s ní mají pracovat, což je podstata testů použitelnosti. Toto jsou oblasti, které je nejlepší otestovat ručně, použití automatizovaných nástrojů zde není vhodné.

Příklad k použitelnosti:

Jedna vysoká škola zpřístupňuje přihlášky a výsledky přijímacího řízení za všechny katedry přímo na svých stránkách, případně přesměruje na konkrétní stránku podřízené fakulty s podrobnějšími informacemi. Nechávat se takto přesměrovávat bylo pro řadu studentů pohodlnější než se rozpomínat na internetovou adresu dané fakulty. Problém byl, že v menu stránek fakulty chyběl odkaz na stránku s aktualitami, která byla považována za domovskou, a na kterou bylo možno se dostat pouze zadáním

adresy dané fakulty nebo kliknutím na její logo na stránkách. Právě v těchto aktualitách byla zveřejněna informace, která zásadním způsobem měnila výsledky přijímacího řízení a která nebyla zpřístupněna skrze nadřízené stránky školy. Velké problémy byly odvráceny tím, že výsledky byly zájemcům dodány i v písemné podobě. Přesto došlo k ojedinělým případům dezinformace.

Při vývoji webové aplikace by mělo být testování použitelnosti zařazeno opakovaně od okamžiku, kdy je hotový grafický návrh a struktura stránek. Pokud při testech použitelnosti není k dispozici funkční verze, je potřeba zúčastněným přesně vysvětlit situaci. Co je cílem tohoto testování, a co naopak není, abychom je neznechutili tím, že „ta věc přece vůbec nefunguje“ nebo nevyvolali v nich pocit, že zkoušíme vlastně je. Během testování nebo krátce po jeho skončení, je vhodné zjistit, jaké pocity z aplikace měli. Uživatelé se většinou sami ozvou pouze, když jim něco dost vadí, případně když chtějí pomoct, a místo připomínek, vymýšlejí nové požadavky a možné vylepšení, což může testery zahltit úkoly, které nespadají do jejich práce. Řízení takovýchto testů proto nepatří mezi nejlehčí.

Se snahou uživatele zaujmout a zároveň jim poskytnout dobře použitelnou aplikaci souvisí **očekávání častých změn** do budoucnosti. Samozřejmě i klasické aplikace se mění, ale u webových aplikací tyto změny probíhají ve větší míře a v jiných směrech, než u ostatních aplikací. Tyto změny by pak měly probíhat hladce a rychle, bez překážek ze strany nepřehledného kódu, nevhodného návrhu a neaktuální dokumentace. Na druhou stranu je třeba si dát pozor na prozrazování informací v komentářích ve zdrojovém kódu HTML přístupném uživatelům. Testeři by neměli zapomínat projít nejen viditelné prvky, ale i ty méně zřejmé a hledat cokoli, co by způsobilo v budoucnosti problémy. To samozřejmě platí u všech projektů bez ohledu na vyvíjenou aplikaci.

Mezi další charakteristické vlastnosti patří, že **webová aplikace může být vystavena mnohem větší zátěži, než bylo předpokládáno**. Výkyvy v návštěvnosti bývají obrovské a dosahují klidně i desetinásobku oproti průměru. Zátěžové testy by tedy měly být variabilnější a pečlivěji zvažovány.

Webové aplikace mají ze své podstaty **ztíženou práci se soubory**, prohlížeč neposkytuje přístup k počítači, programátoři tedy musí hledat jiný způsob. Toto je pravděpodobně i jeden z důvodů, proč internet nikdy nebyl navržen jako bezpečný.

Mezi hlavní narušení jeho bezpečnosti patří

- promíchávání kódu a dat, jenž umožňuje jednoduché a často používané útoky typu cross site scripting a sql injection
- bezstavovost, kvůli které je třeba použít další prostředky pro zapamatování si, zda uživatel je přihlášen, a která způsobuje nepříjemná překvapení, když vývojáři zapomenou, že požadavky mohou chodit v libovolném pořadí
- …

Největším rozdílem vedle zranitelnosti webových aplikací ať už vůči útokům nebo nepřízni uživatelů, je okamžitá aktualizace, která je jejich obrovskou výhodou.
U desktopových aplikacích je po každém updatu nutno aktualizovat všechny kopie, aby se v nich opravy projevily. Proto jsou závažné chyby objevené až po vydání klasických aplikací tak drahé, i když je neobjeví zákazník, ale vývojový tým. V takovém případě je nutné vydat a rozeslat aktualizace, nebo je zpřístupnit na internetu a upozornit na ně uživatele. Opravy chyb už používaných webových aplikací naopak mohou být nahrány během okamžiku na webový server a uživatelé po přístupu do aplikace pracují s aktualizovanou verzí, aniž by nutně nějaké změny postřehli.

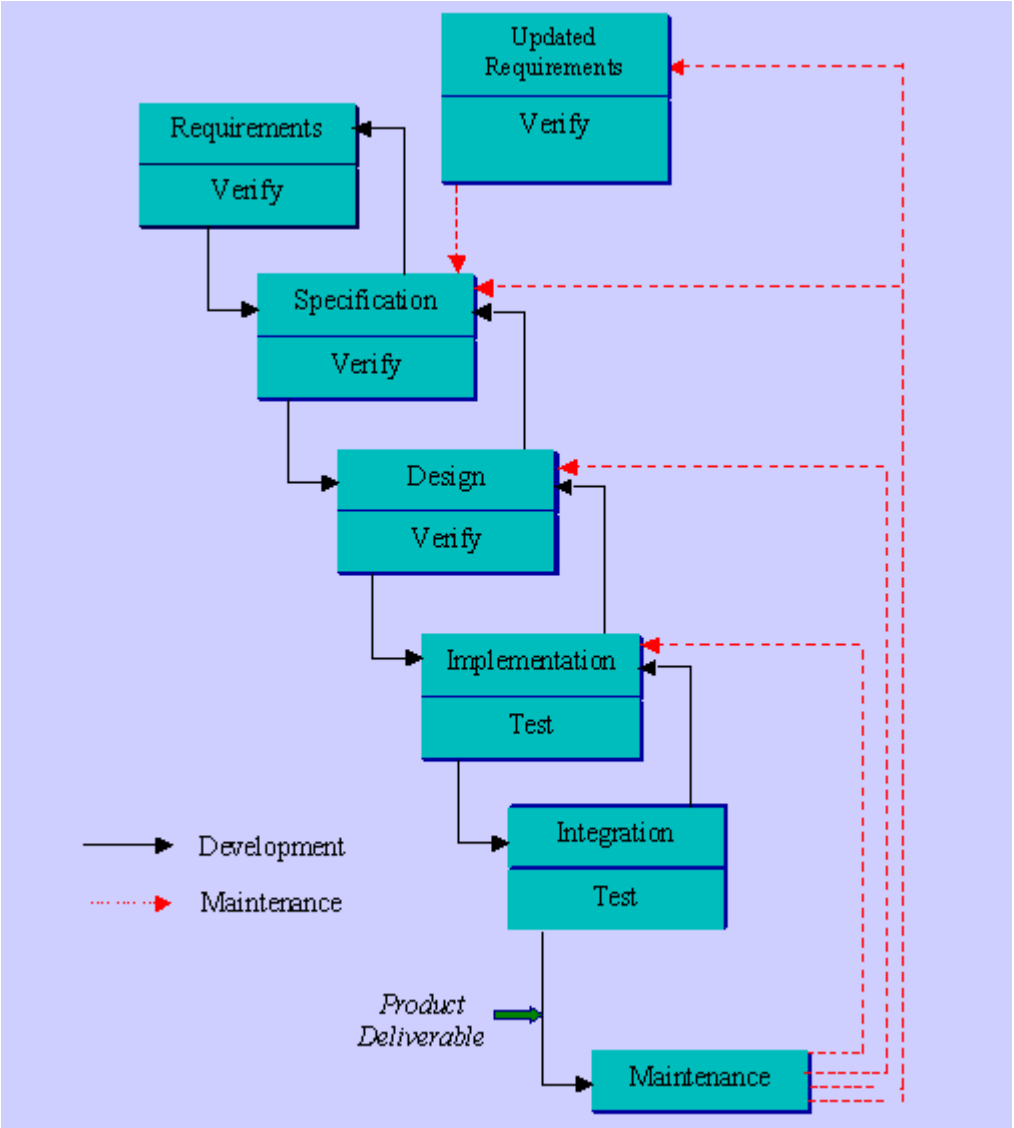
Václav Kadlec [4] považuje za jeden z rozdílů rychlost, se kterou musí být webová aplikace vyvíjena. Na str. 242 uvádí, "…u internetových projektů je však často rychlost dodání jedním z nejdůležitějších faktorů úspěchu či neúspěchu. U webových aplikací rozhodují často jednotlivé dny; prezentace spuštěná se zpožděním je automaticky znevýhodněna. … Někteří odborníci v souvislosti s vývojem webových aplikací zavedli pojem *web-time*. Upozorňují tak na extrémní pojetí času při vývoji internetových projektů."
S tímto rysem by se však dalo polemizovat. Výroba softwaru je silně konkurenční prostředí, protože buď dodává zákazníkovi konkurenční výhodu, nebo přímo o zákazníky bojuje. Kdykoli je vyvíjen podobný software dvěmi firmami a je pravděpodobné, že za dřívějším vydáním bude stát i větší zisk, ocitá se projekt pod tlakem a čas hraje jednu z hlavních rolí, bez ohledu na to, o jaký software se jedná. Webové aplikace jsou ale dostupnější pro širokou veřejnost, a proto je každý den, o který vyjde aplikace dřív, důležitější než normálně. Přesto však na rychlosti záleží jen u specifické části webových aplikací, která není až tak velká. Rozdíl tedy nevidím obecně v nutnosti vyvíjet webové aplikace rychle, ale v dopadu, jaký tento rychlejší vývoj, je-li opravdu potřeba, může mít. Pak opravdu pojem *web-time* dostává smysl, neboť tlak na vývojový tým je obrovský.

6. Metodiky pro web

Metodika je seznam doporučení, postupů a návrhů, která nám říká, jak by mělo něco probíhat, v tomto případě, jak by měl vypadat proces vývoje nějaké aplikace. Metodiky pochopitelně nebývají vždy vhodné pro všechny druhy softwaru, proto se různě specializují, nebo alespoň ohýbají a přizpůsobují charakteristickým rysům daného druhu aplikace.

Metodiky stály u vzniku softwarového inženýrství, které se snaží nalézt systematický a měřitelný způsob vývoje a údržby softwaru. Ten je často příliš složitý, aby se vyvíjel intuitivně. Přirovnáme-li vývoj softwaru k stavbě budovy, intuitivně se můžeme pokusit postavit boudu pro psa, ale ne katedrálu nebo chrám. Čím víc toho víme o architektuře, řízení lidí a shánění financí, tím do větší a úžasnější budovy se můžeme pustit a tím snazší je pro nás i díky novým technologiím, jako jsou motorové pily, vrtačky, bagry postavit budovy méně úžasné a náročné. Stejně tak, ještě nedávno obtížné úkoly při vývoji softwaru, jsou pro nás díky novým technologiím už hračkou. Ale zákazníci nebo i vývojáři sami před sebe kladou stále větší a komplexnější úkoly, proto metodiky hrají významnou roli. Ve své nezměněné podobě jsou platné mnohem delší dobu, než jsou dnes vývojové platformy nebo software.

Jelikož je softwarové inženýrství poměrně mladou oblastí, jejíž rozvoj se datuje od druhé poloviny 60. let, metodiky se ještě stále zlepšují a snaží nalézt vhodné postupy tak, aby vzniklý software byl co nejkvalitněji, nejrychleji a nejlevněji vyroben. S tím úzce souvisí i postavení testování v metodikách. První metodiky umísťovaly intuitivně testování až na konec procesu. Tento pohled má ale dvě nevýhody. První, obecně známá je, že se dosti pozdě našly zásadní chyby, jejichž oprava znamenala přepsání výrazné části kódu a tím prodloužení a prodražení projektu. Druhá je patrná, zamyslíme-li se nad tím, co se děje s chybovým kódem v průběhu projektu. Prvním uživatelem takového kódu je často jiný kód, který na něj navazuje. Takto se už dříve zjistí, že není možné ve vývoji pokračovat bez nalezení a opravení některých chyb, což ukazuje na nutnost kontrolovat kód okamžitě po jeho napsání. Testování bylo posouváno co možná nejdřív v procesu vývoje tak, aby každá část díla byla zkontrolována ihned po napsání. Příkladem je následující obrázek, kde vidíme upravenou verzi vodopádového modelu, jak ji popisuje Adrian Als a Charles Greenidge v [13].



Obr. 1: Vodopádový model vývoje podle [13]

Dalším krokem ve vývoji postavení testování v rámci metodik byl testy řízený vývoj odvozený od stejnojmenné metodiky. Při použití tohoto přístupu se nejprve napíšou testy na danou část aplikace a teprve potom se začne vyvíjet. Část je hotova v tom okamžiku, kdy projde předem napsanými testy. Části, na které se tento přístup aplikuje, bývají často velmi malé, například o velikosti tříd.

6.1 Klasické metodiky a web

První vlna metodik začala vznikat jako reakce na tak zvanou softwarovou krizi, která byla způsobena rostoucí složitostí softwarových projektů a jejich obrovskou neúspěšností. Edsger Dijkstra [19], autor Dijkstrova algoritmu a programátor, napsal:

"…the major cause [of the software crisis] is… that the machines have become several orders of magnitude more powerful! To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming had become an equally gigantic problem."

…

„To put it in another way: as the power of available machines grew by a factor of more than a thousand, society's ambition to apply these machines grew in proportion, and it was the poor programmer who found his job in this exploded field of tension between ends and means. The increased power of the hardware, together with the perhaps even more dramatic increase in its reliability, made solutions feasible that the programmer had not dared to dream about a few years before. And now, a few years later, he had to dream about them and, even worse, he had to transform such dreams into reality!"

Což v překladu znamená:

„Hlavní příčinou [softwarové krize] je … že stroje se staly o několik řádů výkonnější! Řečeno bez obalu: dokud neexistovaly stroje, programování nebylo vůbec problémem; když jsme měli pár slabých počítačů, programování se stalo mírným problémem, nyní když máme ohromné počítače, programování se stalo stejně ohromným problémem."

…

„Jinými slovy: jakmile výkon dostupných strojů vzrostl více než tisíckrát, očekávání společnosti ohledně těchto strojů vzrostlo stejně, a byl to chudák programátor, jehož práce se nacházela na tomto rozpínajícím se poli napětí mezi tím, jak to bylo míněno a jak to skutečně dopadlo. Zvýšený výkon hardwaru společně se snad ještě dramatičtějším zvýšením spolehlivosti, umožnil uskutečnit řešení, o jakých se programátor před pár lety neodvážil ani snít. A teď, o několik let později, o nich musí snít a což je horší, musí takové sny transformovat do reality!"

Ve snaze zajistit úspěšné zvládnutí složitosti nových softwarových projektů, byly navrhovány metodiky jako přísně organizované a disciplinované. Tyto metodiky jsou často označovány jako klasické. Řada těchto metodik byla podrobně propracována a s postupem času i vylepšována. Proto dosud platí, že pokud budeme vyvíjet aplikace, které se svou složitostí a charakterem podobají těm, pro které byly tyto metodiky určené, budou i tyto metodiky aktuální a používané.

Jedním s nejznámějších zástupců klasických metodik je Rational Unified Process (RUP), robustní komerční metodika vyvinuta firmou Rational. Je to metodika určená hlavně pro složité projekty, na kterých pracují desítky lidí nebo které jsou určitým způsobem kritické.

Zároveň je ale snadno přizpůsobitelná menším projektům, i když v omezené míře. Při snaze o aplikaci metodiky na velmi malé projekty by RUP ztratil své typické rysy a tím i svůj význam. Naopak při snaze o zachování systematického až byrokratického přístupu RUP u malého projektu, by byl vývoj méně efektivní a tým by mohl mít problémy přijmout v tomto případě zbytečně složité praktiky. Jak říká jedno české úsloví, bylo by to jako jít na vrabce s kanónem.

Důvodem, proč je možné v omezené míře RUP upravovat na menší projekty, včetně jeho využití při vývoji webových aplikací, je univerzálnost některých jeho přístupů. Šest takzvaných nejlepších praktik softwarového vývoje, jenž jsou zabudované do řady metodik a jenž jsou základem i RUPu, jsou naprosto v souladu s potřebami vývoje webových aplikací. Podle [3] tyto praktiky vznikly spontánně jako společný základ většiny řešení, které firmy implementovaly ve snaze předejít problémům spojených s nepředvídatelností a dynamičností vývoje. Jelikož vývoj webových aplikací bývá provázen těmi samými problémy, ale v ještě větší míře, je logické, že tyto praktiky léčí i problémy webových aplikací.

Těchto šest „nejlepších praktik“ je :

Iterativní vývoj

Komponentová architektura

Vizuální modelování

Řízení požadavků

Řízení změn

Neustálá kontrola kvality

Praktiky se navzájem podporují. Při pohledu na možnosti neustálé kontroly kvality, je vidět, že ostatní praktiky testování usnadňují a zpřehledňují. Iterativní vývoj a komponentová

architektura umožňují dřívější a častější testování. Vizuální modelování, řízení požadavků a změn umožňuje přesnější seznámení se s produktem, přípravu testů před tím, než je napsán kód, a lepší informovanost týmu, což výrazně zabraňuje přehlédnutí chyb nebo riziku reportování správné funkčnosti jako chyby.

Z metodiky RUP vznikla i její speciální verze určená pro vývoj webových aplikací. Hodí se nejen pro složité systémy s tenkým klientem, ale i pro běžné internetové aplikace. Podrobnosti je možné najít na [44].

6.2 Agilní vývoj

Druhá vlna metodik vznikla v 90. letech v reakci na používání klasických těžkopádných metodik, které nejsou dostatečně flexibilní. V oblasti vývoje softwaru chyběly pružné rychlé metodiky určené spíše malým týmům. To se rozhodla napravit skupina odborníků, kteří v půlce února 2001 sepsali agilní manifest, ve kterém dali najevo, jak si cení komunikace a flexibility při vývoji, viz příloha 1. Většina z nich v té době už měla publikovanou nebo aspoň promyšlenou nějakou metodiku, která se pak začala řadit do skupiny agilních metodik.

Agilní metodiky jsou svojí povahou přímo předurčené pro menší projekty, kde se požadavky často mění, a které je potřeba vyvíjet rychle a pružně.

Kent Beck, jež je autorem jedné z nejznámějších agilních metodik, Extrémního programování (také známého pod zkratkou XP), přirovnává řízení projektu za pomoci jeho metodiky k řízení auta [1]. Auto se řídí neustálými korekcemi jeho kurzu. Je potřeba cestu neustále hlídat a být připraven na změnu. I ve vývoji softwaru se vše stále mění, členové týmu, zadání, technologie, atd. Řidičem je v tomto případě zákazník. Ten detekuje změnu a zatočí volantem tu doprava tu doleva. Úkolem programátorů je pak poskytovat co nejdříve zpětnou vazbu, aby zákazník viděl, kde přesně se teď na silnici s autem nachází.

Tato flexibilita je společná všem agilním metodikám a může mít vliv i na cenu opravy nalezené chyby. Agilní metodiky vesměs doporučují vytvořit pouze nezbytné minimum dokumentace a dbají na čitelnost a srozumitelnost kódu. Díky tomu, je po opravě chyby v samotném kódu snadnější opravit související dokumentaci, je-li to potřeba. Některé praktiky dokonce mají pozitivní vliv na vyhledávání a izolaci chyb v kódu, což opět může snížit cenu opravy. Dá se předpokládat, že velký vliv má i postoj programátora. Pokud píše kód s tím, že očekává, že ho může v budoucnu někdo chtít změnit nebo použít i jinde, pravděpodobně bude snazší a tím pádem levnější udělat v tomto kódu změny včetně opravy chyb. Flexibilita a

minimalizace dokumentací mohou tedy snížit náklady na testování a opravy. Není ovšem dobré testování podceňovat.

Agilní metodiky bývají v tomto případě často ve vztahu k testování špatně pochopeny. To proto, že buď příliš testování nerozebírají nebo věnují pozornost jen některé jeho části. Vytváří tak občas dojem, že testování nepovažují za důležité, zatímco ve skutečnosti je už považováno za samozřejmou součást celého vývojového cyklu. Pokud není řečeno jinak, testování je prováděno stejným způsobem jako v klasických metodikách. Hlavní vylepšení, které agilní metodiky v této oblasti přinesly, je důraz na používání testovacích nástrojů. Ty zrychlují a zpřesňují práci testerům a přinášejí větší jistotu při vytváření kódu. Část automatických testů, jejichž psaní se podobá víc programování než testování, mohou navíc převzít programátoři. Toto značně usnadňuje a zrychluje práci testerů, kteří se pak mohou více věnovat činnostem, u kterých je stroj nenahradí, jako řízení testů, vyhodnocování, testování použitelnosti a komunikaci se zákazníkem, která je pro agilní metodiky velmi důležitá. To v konečném důsledku vede k tomu, že je možno provádět více druhů testů a provádět je levněji. Díky tomu dochází i ke zkvalitnění produktu.

Důraz na automatizované testování a na myšlenky testy řízeného vývoje můžeme najít mimo jiných u již zmíněného Extrémní programování. Za jeho prvotní myšlenku by se dalo označit „v maximální míře provádět vše, co se při vývoji osvědčuje“. Ani myšlenky této metodiky, která bývá považována za kontroverzní a revoluční, nejsou jedinečné. Stejně jako myšlenky všech metodik, je založena na něčem, co bychom nazvali „zdravým selským rozumem“ autorů.

Gerald M. Wienberg podle [28] napsal při osobní komunikaci o vývoji softwaru v období 50. až 60. let a o projektu Merkury[1], na kterém pracoval:

„We were doing incremental development as early as 1957, in Los Angeles, under the direction of Bernie Dimsdale [at IBM's Service Bureau Corporation]. He was a colleague of John von Neumann, so perhaps he learned it there, or assumed it as totally natural. I do remember Herb Jacobs (primarily, though we all participated) developing a large simulation for Motorola, where the technique used was, as far as I can tell, indistinguishable from XP. When much of the same team was reassembled in Washington, DC in 1958 to develop Project Mercury, we had our own machine and the new Share Operating System, whose symbolic modification and

[1] Projekt Merkury byl první pilotovaný kosmický projekt USA, jehož cílem bylo dopravit člověka na oběžnou dráhu a zpátky.

assembly allowed us to build the system incrementally, which we did, with great success. Project Mercury was the seed bed out of which grew the IBM Federal Systems Division. Thus, that division started with a history and tradition of incremental development. All of us, as far as I can remember, thought waterfalling of a huge project was rather stupid, or at least ignorant of the realities… I think what the waterfall description did for us was make us realize that we were doing something else, something unnamed except for 'software development'."

V překladu:

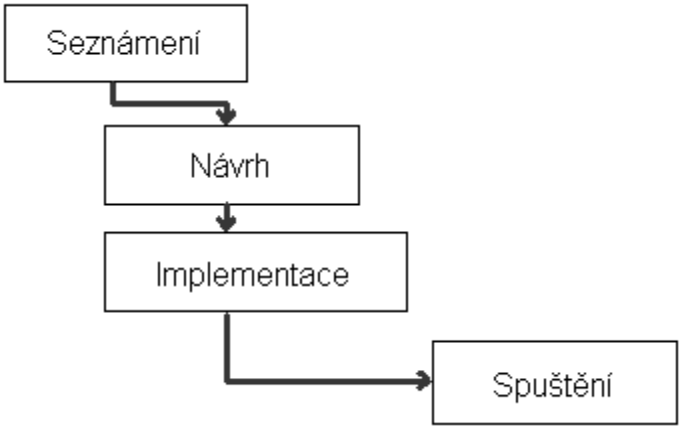
„My jsme inkrementálně vyvíjeli už v 1957 v Los Angeles pod vedením Berniho Dimsdala [v servisním výpočetním středisku IBM]. On byl kolegou Johna von Neumanna, tak se to možná naučil, nebo to považoval za naprosto přirozené. Pamatuju si Herba Jacobse (primárně, neboť všichni jsme se účastnili) vyvíjet velkou simulaci pro Motorolu, kde byla použita technika, alespoň co já mohu soudit, nerozeznatelná od XP. Když byl skoro stejný tým sestaven ve Washingtonu DC v 1958 k vývoji na projektu Merkury, měli jsme vlastní stroje a nový sdílený operační systém, jehož symbolická modifikace a sestava nám umožňovala vyvíjet systém inkrementálně, což jsme udělali s velkým úspěchem. Projekt Merkury bylo semeno, ze kterého vyrostlo oddělení federálních systémů IBM. Tudíž toto oddělení začalo s historií a tradicí inkrementálního vývoje. Všichni, pokud si vzpomínám, jsme si mysleli, že vodopádový vývoj velkého projektu je hloupost nebo alespoň ignorování reality… Myslím, že popis vodopádového vývoje nám umožnil si uvědomit, že děláme něco jiného, něco co nemělo jiný název než ‚vývoj softwaru'."

6.3 Další metodiky

Vedle těchto dvou vln metodik, existuje řada dalších, které vznikly mimo jmenovaná období, nebo neodpovídají rysům ani jedné z vln. Nejsem si vědoma žádného výzkumu, který by se snažil je všechny popsat a roztřídit, nehledě na to, že některé firmy mají své vlastní firemní metodiky pro vývoj softwaru. Často se vyskytují i metodiky s užší specializací zaměřující se třeba konkrétně na webové aplikace nebo pouze na jejich konkrétní třídu např. webové stánky (o rozdělení webových aplikací bylo psáno dříve). Tyto metodiky nebývají podrobně popracované a připomínají sadu doporučení doprovázených občas podrobnějším

návodem. Je přirozené, že metodika odráží složitost vývoje. Slouží k tomu, aby vývoj probíhal hladce a přehledně, neměla by projekt zbytečně brzdit.

Jednou z propracovanějších metodik je metodika WebWave [21], určená především pro nejkomplikovanější třídu webových aplikací - softwarové systémy s tenkým klientem. Na první pohled zaujme modelem životního cyklu vývoje, podle kterého začíná implementace zároveň s návrhem, viz obr. 2. Testování, které na obrázku chybí, je chápáno jako součást implementace, ale kontrola kvality jako celek začíná mnohem dříve a končí později. Jak je to u metodik pro webové aplikace běžné, klade WebWave důraz na tvorbu prototypu a komunikaci se zákazníkem.



Obr. 2: Model životního cyklu vývoje podle metodiky WebWave

Důležitým prvkem metodiky WebWave je porozumění cílům organizace, pro kterou se systém tvoří a tomu, jak k tomuto cíli má vytvářený systém přispět. Vývojový a testovací tým by měl chápat, jakou přidanou hodnotu jednotlivé části produktu budou přinášet a komu. Toto je situace, která je žádoucí pro vývoj jakéhokoli druhu aplikace.

Kromě již zmíněné tvorby prototypů a důrazu na komunikaci se zákazníkem se metodiky pro vývoj webových aplikací vyznačují oblibou workshopů[2] a brainstormingů[3] v úvodních fázích projektu. Společná je pro ně i snaha o rychlejší a kreativnější vývoj. U testování pak je kladen důraz na použitelnost, konzistenci a přesnost.

[2] Workshop = společné setkání za účelem vytvoření nějaké věci

[3] Brainstorming = druh řízené diskuze určený pro generování nápadů

V souvislosti s webovými aplikacemi se objevují i různé další sady doporučení, jejichž dodržení nezajistí plynulý vývoj, ale přináší určité výhody, například snadnější přenositelnost nebo přístupnost stránek zrakově postiženým osobám.

Pro zjednodušení vývoje webových aplikací existuje řada nástrojů, některé z nich jsou určené přímo pro testování.

Část II. – Testování

7. O čem je testování

7.1 Definice

To, že se testování týká vyhledávání chyb bývá docela jasné, často se však liší chápání jeho záběru, to, jakým způsobem jsou chyby při testování vyhledávány. Některé zdroje popisují testování pouze jako dynamickou kontrolu toho, že chování programu odpovídá specifikaci [12]. Tedy k tomu, abychom aplikaci testovali, je podle této definice nutné její spuštění a existence specifikace nebo alespoň sady podmínek a pravidel, oproti kterým bychom ji kontrolovali. Širší pohled na testování nabízí Hailpern a Santhanam v [22], kteří testování definují jako jakoukoli aktivitu, která odhalí, že chování programu porušuje specifikaci; tedy do své definice počítají i takzvané statické testování, jenž nevyžaduje k vyhledávání chyb spuštění daného kódu. Mezi aktivity statického testování patří zejména různé druhy odborného pročítání kódu. Nejobecněji bývá testování považováno za samostatnou disciplínu, jejímž úkolem je ověřování kvality. V rámci tohoto přístupu bývá testování definováno vágněji, což více odpovídá tomu, jak je v praxi na testování skutečně nahlíženo. Příkladem může být definice od Kanera [11]: Softwarové testování je empirický technický výzkum kvality testovaného produktu nebo služby, prováděný za účelem poskytnutí těchto informací stakeholdrům[4].

V jakémkoli pojetí ale vždy platí, že proces testování je podmnožinou procesu ověřování a plánování kvality. Proto mohou být úkoly testovacího týmu dosti široké a na modelech životního cyklu pozorujeme, že testovací disciplína se nejen protahuje do celého vývoje, ale často zastává místo zajišťování kvality.

Co si však představit pod kvalitou produktu? Ať už definujeme kvalitu jako přínos nějaké osobě [10] nebo stupeň splnění požadavků (podle [12] jde o definici z ISO 9001:2000, Quality Management Systems — Requirements), je důležité mít na paměti, že je na ni nahlíženo vždy ve vztahu k nějaké osobě nebo spíše osobám. Produkt má splnit požadavky konkrétních osob – stakeholdrů, a tyto požadavky nemusí být vůbec zapsány nebo vyřčeny.

[4] Stakeholder – člověk, který má zájem na dané věci.

Za vznikem softwaru stojí nějaká potřeba, tedy kvalitní software je takový, který tyto potřeby plně uspokojuje.

7.2 O čem je testování

Testování je založené na preciznosti vývojového týmu a jeho testerů, na porozumění produktu a zákazníkovi, pro kterého se produkt dělá. Tým se společně snaží vyvinout kvalitní produkt, který by zákazníkovi přinášel prospěch, protože jednou z přirozených motivací lidí je pocit z dobře vykonané práce. Takováto motivace ale nestačí, je potřeba dosažení kvality plánovat a řídit. Na začátku vývoje je třeba stanovit měřítka kvality, zvolit vhodný přístup a připravit implementaci tohoto přístupu, abychom měli jistotu, že nejen produkt, ale i proces jeho vývoje dosahuje požadované kvality. Dále se stanoví záběr testování, vybírají se testy, sbírají data a připravují nástroje, které tým k testování potřebuje. Navíc se kontroluje, zda všechny požadavky na produkt jsou ve formě, aby bylo možno jednoznačně zkontrolovat jejich splnění.

Přestože tato náročná disciplína v mnoha organizacích pohltí 30-50% rozpočtu, stále dost lidí věří, že software nebývá před dodáním dobře otestován [3]. Mnoho rozhodnutí ohledně testování závisí na zkušenostech a osobnostech lidí v týmu. A testování často připomíná boj s vícehlavou saní, protože testování naráží na mnoho omezujících překážek.

První z nich popsal Dijkstra v [19] když napsal:

„…program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence."

V překladu:

„…testování programu může být velmi efektivní způsob, jak ukázat přítomnost chyb, ale je beznadějně nevhodné k prokázání jejich nepřítomnosti."

Absenci chyb může ukázat jen formální důkaz, což ale kvůli své náročnosti není v praxi běžná metoda.

Mezi další problémy, se kterými je třeba počítat a vypořádat se s nimi, patří:

- Není možné otestovat kompletně všechny možné případy zadání.
- Je velmi složité odhalit, že bylo ve specifikaci na něco zapomenuto.

- Specifikace není vždy jednoznačná nebo dokonce chybí.
- Testování bývá podceňováno nebo špatně pochopeno.
- Výsledky testování mohou být ovlivněny špatnou komunikací v týmu nebo se zákazníkem.
- Testování musí být přizpůsobeno testovanému produktu, není možné stejným způsobem otestovat různý software.
- V průběhu testování se mohou stát některé testy neefektivními. Testy je nutné neustále přizpůsobovat měnícím se okolnostem.
- Dvě různé chyby se mohou navenek projevovat stejně, nebo jedna chyba se může navenek projevovat různě.
- Testování je citlivé na vstupní data, ta by měla být realistická. Vygenerování jednotvárných dat není ideální.

Samotné vyhledávání chyb k zajištění kvality produktu nestačí, přestože to není na první pohled patrné, v rámci testování je třeba udělat něco i pro to, aby chyba byla opravena. Nalezením chyby začíná další neméně důležitá práce testera a to je napsání přehledného reportu a další komunikace chyby. Pro připomenutí tohoto úkolu existuje následující věta, která je pro testery spíše mottem než definicí:

„Testing is process of searching defects in what more shortest period of time and of the fastest assurance of their correction.“ [11]

V překladu:

„Testování je proces hledání chyb za co nejkratší čas a s co nejrychlejším zajištěním jejich opravy.“

8. Chyby

V některých firmách se vedou vášnivé diskuze o tom, jak by měly být definovány pojmy úzce související s chybou a pečlivě rozlišují mezi takovými pojmy jako je odchylka, událost, anomálie, problém, selhání a mnohé další. Vysvětlení těchto pojmů závisí na firemní kultuře a není potřeba se jimi na akademické úrovni zabývat. Naopak každý tester musí vědět, co při testování hledá a pojem chyby je klíčový.

Definice chyby by se neměla soustředit pouze na specifikaci, protože pak by nastala situace, že projekty bez specifikace by nemohly být testovány, protože by se nikdy nemohla najít žádná chyba. Také by neměla definice zapomínat na očekávání klienta, která nejsou na první pohled patrná a jejichž ohrožení by klient ani nepoznal.

Následující dvě definice předešlým podmínkám vyhovují a jsou do jisté míry ekvivalentní.

Definice od Pattona [7]:

„O softwarovou chybu se jedná, je-li splněna jedna nebo více z následujících podmínek:

1) Software nedělá něco, co by podle specifikace dělat měl.
2) Software dělá něco, co by podle specifikace produktu dělat neměl.
3) Software dělá něco, o čem se specifikace nezmiňuje
4) Software nedělá něco, o čem se produktová specifikace nezmiňuje, ale měla by se zmiňovat.
5) Software je obtížně srozumitelný, těžko se s ním pracuje, je pomalý nebo – podle názoru testera softwaru – jej koncový uživatel nebude považovat za správný.“

Definice zformulovaná na základě [3]:

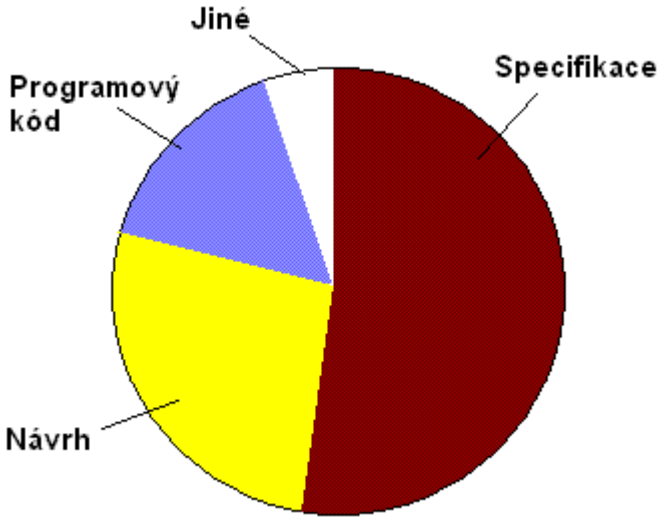
Chyba je cokoli ohledně programu, co podle některého ze stakeholdrů zbytečně snižuje hodnotu programu.

Osobně se přikláním ke kratší z definic, protože by platila zřejmě i v případě, kdyby se ukázalo, že v Pattonově definici ještě nějaký bod chybí. Navíc je pro testery lehčeji zapamatovatelná a tudíž praktičtější.

Další věc, co by měl tester o chybách znát je, kde se nejčastěji vyskytují a kde objevit ty nejdůležitější, jejichž oprava by v pozdějších fázích mohla být hodně drahá. Na taková místa je pak třeba se nejvíce zaměřit.

Řada materiálů o testování uvádí obrázek 3, který znázorňuje, že více jak polovinu všech chyb má na svědomí specifikace. Osobní zkušenosti moje a mých kolegů tomu ale neodpovídají. Je pravděpodobné, že kdysi tyto proporce původu chyb byly správné, ale s pokrokem v softwarovém inženýrství byly změněny.

Přestože ty nejhůře objevitelné nebo s největším dopadem jsou právě chyby mající původ ve specifikaci či v návrhu, nejčastější jsou chyby v programovém kódu nebo vzhledu aplikace, ty činí 50-70%. Navíc je zřejmé, že takováto charakteristika závisí na typu projektu. Vývoj řízený testy nebo schopný analytik či zkušení programátoři mohou procenta chyb podle původu značně ovlivnit.



Obr. 3: Příčiny chyb podle Pattona [10]

Přesto první věc, která si zaslouží značnou pozornost, je právě specifikace a návrh. Proč? V prvé řadě kvůli dopadu těchto chyb. Příliš pomalý software je jistě pro zákazníka horší, než překlep na jednom z formulářů. Dalším důvodem je, že na základě těchto dokumentů si testeři mohou zjistit něco o přáních zákazníka a seznámit se s vyvíjeným softwarem. Soustředění se na tyto části může vývoji ušetřit nepříjemná překvapení a nejvíce ovlivnit výnosnost projektu.

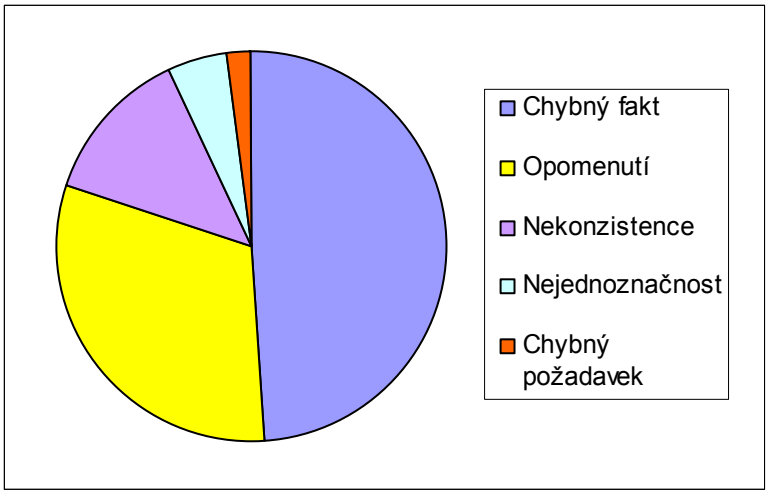
Pravidlo soustředění se na hledání nejzávažnějších chyb platí i obecně, a proto je třeba první testovat části, které jsou považovány z nějakého důvodu za kritické. Teprve, když ty nejdůležitější části aplikace jsou v pořádku, je vhodné se zaměřit na hledání ostatních chyb.

Kromě důležitých částí aplikace, je nutné podrobnější testování částí, u kterých existuje podezření, že nejsou úplně v pořádku. Okamžité sledování takovýchto podezření

často ušetří hodiny až dny práce navíc, než kdyby byly testy zaměřeny jinam. Mezi kód, který vyžaduje takovouto pozornost, patří kód napsaný ve spěchu, nezkušeným programátorem, nebo i konkrétním programátorem, který by zkušený být měl. Většinou nejlepší informace o tom, na který kód se zaměřit, lze získat právě od vývojového týmu.

Říká se, že hodina pátečního přesčasu programátora stojí celé pondělí strávené nad opravami.

Zajímavé, i když už ne tak užitečné, je i vědět s jakými druhy chyb se můžeme setkat, z čeho pramení. Nejčastějším neduhem je zřejmě jednoduše chybný fakt, který vyplynul z analýzy problému, ať už k tomu došlo při jakékoli činnosti. Zde například patří i řada bezpečnostních chyb. Špatně odhadneme problém. Další velmi častou chybou je opomenutí a občas ve velmi těsném závěsu, zvláště když se projekt už chýlí ke konci, jsou objevovány nekonzistence. Nejednoznačnosti a chybné požadavky naopak zabírají z celkového počtu chyb jen malou, ale i tak nepřehlédnutelnou část.



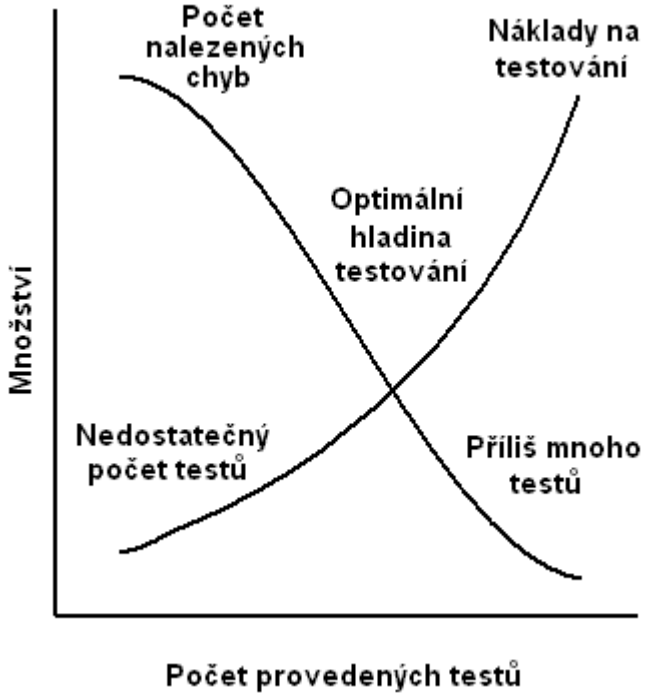
Obr. 4: Druhy chyb podle [20]

Další zajímavou charakteristikou chyb jsou náklady na jejich objevení a opravení, tedy náklady na testování. Otestovat všechny možné případy zadání, všechny situace není možné. Pro názornost stačí uvést známý příklad s kalkulačkou, kdyby šlo vyzkoušet všechny kombinace zadání: 1+0, 1+1, 1+2, 1+…?, 2+0, 2+1…?, a pak odečítání, násobení, dělení, desetinná čísla, záporná čísla, nečíselné znaky atd. atd.

Kdy se tedy vyplatí s testováním skončit? **Jelikož vývoj softwaru je ekonomická činnost, s hledáním dalších chyb by se mělo přestat v okamžiku, kdy náklady na nalezení**

a opravu chyby jsou v průměru stejné nebo větší než náklady na ponechání chyby v softwaru. Ve větě jsou dva zamlčené předpoklady, první je, že jsme schopni správného vyčíslení obou nákladů, druhý, že průměrné náklady na chybu nejsou počítány ze všech nalezených chyb, ale jen z chyb nalezených dalšími testy.

Na následujícím obrázku je znázorněn zjednodušený model vztahu nákladů a chyb, jak jej zachycuje Patton v [10].



Obr. 5: Vztah nákladů a testů podle [10]

Patton ve vztahu k obr. 4 tvrdí, že každý softwarový projekt má nějakou efektivní hladinu testování. To je pravda, ale poněkud zjednodušená. Situace se zde komplikuje neuniverzálností testování. Jak už bylo řečeno, každý produkt je jedinečný a testy jsou navrhovány na základě zkušeností a preferencí toho, kdo o jejich provedení rozhoduje. Dva lidé můžou pro jeden produkt navrhnout dvě různé sady testů s různými náklady.

Podobné testy vyhledávají stejný druh chyb. V průběhu testování se tak rychle stávají neefektivními, v takovém případě při nahrazení jiným zásadně odlišným testem počet nalezených chyb náhle rapidně stoupne. Původní pravidlo od Pattona by se tedy dalo zpřesnit tak, že pro každý softwarový projekt a pro každou množinu testů existuje efektivní hladina testování.

Poměrně nízkých nákladů při otestování co největšího počtu částí aplikace a při odhalení různých druhů chyb, jinými slovy při dobrém pokrytí testy, se dosahuje rozdělením testů podle jejich zaměření a vybráním protikladných a doplňujících technik, které jsou snadno proveditelné a lze je levně aplikovat.

S náročnějšími aplikacemi a systémy se ztížilo i vyhledávání chyb. Jeden tester stejně jako jeden programátor už svými znalostmi nepokrývá všechny oblasti vývoje. Z tohoto důvodu se někteří z nich specializují více na jednu nebo několik málo oblastí. Na druhou stranu ve snaze si tuto stále komplikovanější práci zjednodušit, začala vznikat řada nástrojů pro podporu nebo přímo pro testování. Některé nástroje pro automatické testy mohou i částečně nahradit práci testera u daného druhu testu, jindy pouze automatizují jeho práci, kterou pak dokáží kdykoli sami zopakovat. Jsou to však jen nástroje a nedokáží ohodnotit většinu aspektů kvality. Nástroje pouze zlevňují a zrychlují některé testy, bez zkušených testerů se ale projekt obejde hůře než bez nástrojů.

Díky pokroku se výrazně snížila cena opravy chyby. Jednak díky návyku komentovat kód a jednodušším programovacím jazykům, jednak díky lepším překladačům a vývojovým prostředím, ale zejména díky internetu, který umožnil levné a rychlé poskytnutí updatů. Toto byl pro kvalitu velký krok vpřed. Například software, který by jinak nebyl otestován vůbec, většinou proto, že se jedná o osobní nebo školní projekt jednoho nebo více programátorů, může dosáhnout zvýšení kvality tak, že je zpřístupněn zdarma ke stažení a uživatelé jsou požádáni o zaslání nalezených chyb. Na důležitějších projektech, kde na kvalitě více záleží, je internet dalším z nástrojů, kterých testeři mohou využít ke zefektivnění a zrychlení své práce. Vrátíme-li se k větě o tom, nakolik testovat, aby se to vyplatilo, vidíme, že nyní se vyplatí dodávat kvalitnější produkty, než tomu bylo v minulosti.

8.1 Motivace – některé známé chyby

3. prosince 1999 se při pokusu modulu Mars Polar Lander o přistání na Marsu došlo ke ztrátě komunikace. Modul se už neozval. Následující vyšetřování odhalilo, že nejpravděpodobnější událostí, ke které mohlo dojít, je předčasné vypnutí motoru a následné zřícení modulu na povrch Marsu.

Původně měl modul přistát pomocí motoru a padáku, po jehož rozvinutí a asi 70 až 100 sekund před přistáním by se vysunuly přistávací nohy. Jakmile by se nohy modulu dotkly povrchu, k čemuž sloužil kontakt nastavující určitý bit, přívod paliva do motoru by byl vypnut.

Pravděpodobnou příčinou bylo předčasné nastavení bitu, kdy kontakt vyhodnotil odpor a vibrace vzduchu při přistání jako povrch Marsu. Modul testovalo několik týmů, každý se zaměřením na jinou část modulu nebo jeho mise. Zapomněli však na testování přechodů, jak předchozí část mise ovlivní následující. A tak pokud byl kritický bit předčasně nastaven, jeden tým to ignoroval, jelikož se jich to netýkalo, a druhý začínal testy na čistém prostředí po restartu počítače a nastavení na defaultní hodnoty, takže s nenastaveným bitem [7], [45].

Z definice chyby jako čehokoli, co snižuje hodnotu systému v očích některého stakeholdra, vyplývá, že je velmi prospěšné zjistit, čím je daný systém ovlivněn nebo co vše ovlivňuje.

Další známá softwarová událost se stala 25. února 1991 v Saudské Arábii, v Dhahranu, kdy kvůli chybě v obranném raketovém systému Patriot, nedošlo k odhalení a pokusu o zneškodnění nepřátelské rakety, která následně usmrtila 28 vojáků.

Příčinou byla chyba v systémových hodinách, která způsobovala zkreslení času v závislosti na době běhu systému. Při incidentu byl systém v provozu přes 100 hodin, přičemž systémové hodiny byly vlivem chyby posunuty o třetinu sekundy. Při takto rychle pohybujícím se cíli, jako je raketa, dělala odchylka při určení pozice 600 metrů. Systém tedy raketu detekoval, ale když nic nenašel na dané pozici, byla označena jako falešný poplach a vymazána ze systému.

Chyba byla ohlášena už dva týdny před incidentem, jako prevence bylo doporučeno pravidelně rebootovat systém. Příslušní zmocněnci ale nevěděli, jak často je to pravidelně. Update byl dodán výrobcem den po incidentu [46].

Ponaučení z této situace je, za prvé, že i ta nejmenší nepřesnost za desetinnou čárkou, třeba při špatném zaokrouhlování, se může po nějaké době nakupit natolik, že její následky už nebudou tak zanedbatelné, jak se zprvu zdálo. Za druhé, uživatelé potřebují přesně vědět, co je chybou a jak chybný systém používat než dojde k opravě.

Jedny z nejzávažnějších chyb byly objeveny v softwaru zařízení Therac-25, radiologickém přístroji pro léčbu nádorů pomocí x-paprsků nebo elektronů. Následkem těchto chyb byli lidé, kteří měli tu smůlu, že při jejich léčení se projevily, vystaveni obrovským dávkám ozáření. Od června 1985 do ledna 1987 je známo šest případů, kdy byl pacient vystaven tenkému paprsku obrovské radiace, tři z nich na následky ozáření zemřeli [29].

Příčinou byla řada pochybení při vývoji, kdy produkt nepodstoupil podrobné prozkoumání. Části softwaru byly převzaty ze starší verze zařízení, přičemž byly zároveň odstraněny hardwarové pojistky proti chybám. Bezpečnost plně závisela na softwaru. Dalším pochybením byla ničím nepodložená přílišná důvěra v produkt. Po prvních incidentech

zaměstnanci firmy, která Therac-25 vyvinula, tvrdili zákazníkům, že je nemožné, aby příčinou potíží byl jejich produkt. A ani poté nebyli schopni sami problém nasimulovat, dokud uživatelé neodhalili sami, jak přesně k chybě dochází.

Vnitřní příčinou těchto smrtelných chyb v softwaru pak bylo přetečení proměnné a neplánovaná závislost na načasování procesů, což je jev zvaný „race condition“.

Ze zkušeností za několik posledních let vyplývá mnoho ponaučení, nejdůležitější ale je, že každý složitější software lze přivést do stavu, kdy se bude chovat neočekávaně. Na takovéto situace je třeba se připravit, snažit se snížit nejen jejich pravděpodobnost, ale i dopad.

9. Testovací tým

9.1 Vlastnosti testera

Existuje mnoho blogů a stránek o testování, kde se můžete seznámit s tím, jak testeři uvažují. Je to zvláštní zkušenost, protože neříkají jen, dělejte to, nebo ono, v článku sice vyjádří svůj názor, ale hlavně kladou čtenářům otázky a nechávají je, ať se sami nad tím zamyslí a určí, co si z toho odnesou. Mezi dobře udělané blogy plných takovýchto článků patří např. testertested.blogspot.com.

Velká část toho být testerem je o kritickém myšlení. Pro dobré testery není nic obecně platné. Zpochybňují vše o produktu i o samotném testování. Kladou otázky a skrz ně poznávají produkt, který testují. Kladení otázek je jeden z nejlepších způsobů, jak testeři dělají svou práci.

Následující seznam obsahuje vlastnosti, kterými se dobří testeři vyznačují. Není třeba mít všechny následující vlastnosti, vynikající testeři mívají třeba jen pár z nich, ale vhodně skloubené.

Pro testera je vhodné:

- Mít odlišné myšlení od průměru. Díky tomu vymýšlejí testy, které by ostatní nenapadly, používají software neobvyklým způsobem. Sada testů, kterými tak produkt má projít, rychle roste.
- Být zvědavý, snažit se objevit skryté. Testeři se nesnaží rozbít nebo pokazit produkt, ten už pokažený je, testeři se snaží odhalit nakolik je pokažený, a ukázat ostatním, co je jim skryto, konkrétně zviditelnit chyby.
- Být pečlivý a trpělivý. Pocit z dobře odvedené práce bývá největší odměnou, protože málokdo umí poznat, zda je testování dobře odvedené nebo ne.
- Umět dobře komunikovat s okolím. Vysvětlení nalezených chyb a zajištění jejich opravy je podstatná část práce, ke které má tester ve skutečnosti jen jediný nástroj – svoji schopnost komunikovat.
- Rád se učit novému. Stát se dobrým testerem je celoživotní úkol, není možné se dostat do bodu, kdy si člověk myslí, že o testování se už nemá co učit.
- Být průbojný. Často testeři přijdou na projekt a dozví se pouhý zlomek toho, co potřebují. Neví pořádně, kde najít dokumentaci, co se ohledně projektu rozhodlo, na jaké otázky se ptát a koho. Vyžádat si půl dne času někoho zkušenějšího, kdo

má hodně práce, vyžaduje odvahu. Ale bez toho není možné testovat dobře a efektivně.

Z osobní komunikace s Pradeepem Soundararajanem[5]:

„Good tester, as the world doesn't know, is someone with skills. When I say skills, it means - observation, hearing, questioning, critical thinking, analysis, general systems thinking, epistemology, self critique, techniques, approaches, knowledge, tools, brainstorming, scripting, technology, communication, interpersonal skills, management, theory..."

Přeloženo:

„Dobrý tester, jak svět neví, je někdo se schopnostmi. Když říkám schopnosti, myslím - postřeh, poslech, dotazování, kritické myšlení, analytičnost, obecné systémové myšlení, nauku o poznávání, sebekritiku, techniky, přístupy, znalosti, nástroje, brainstorming, skriptování, technologii, komunikaci, mezilidské vztahy, řízení, teorii…"

Dobrou lekcí o testování je příběh o diskuzi Bena Simo s Michaelem Boltnem[6] [38], který se odehrával zhruba takto:

MB: „Co čekáš, že dostaneš, když si objednáš pivo?"
BS: „Pivo".
MB: „Jasně. A očekáváš, že ho dostaneš co nejrychleji?"
BS: „Ano."
MB: „Očekáváš, že bude mít správnou teplotu?"
BS: „Ano."
MB: „Očekáváš, že bude mít správnou míru?"
BS: „Ano."
MB: „Očekáváš, že bude mít správnou barvu?"
BS: „Ano."
MB: „Očekáváš, že to bude stejné pivo, jako jsi si objednal?"
BS: „Ano."
MB: „Očekáváš, že bude mít správný poměr pěny?
BS: „Ano."
MB: „Očekáváš, že obsluha, která ti ho přinese, bude vhodně upravená?
BS: „Ano."

[5] Pradeep Soundararajan je výborný indický tester a autor blogu testertested.blogspot.com
[6] Michael Bolton je považován za skvělého testera a experta na testování. V současné době zejména poskytuje školení a konzultace ohledně testování, přičemž ovlivňuje řadu testerů na celém světě.

```
MB: „Očekáváš, že bude obsluha přívětivá?“
BS: „Ano.“
MB: „Očekáváš, že bude stůl čistý a ne politý nebo ulepený?“
BS: „Ano.“
MB: „Očekáváš, že v pivu nebude nic plavat?“
BS: „Ano.“
MB: „Očekáváš, že sklenice na pivo bude čistá?“
BS: „Ano.“
MB: „Očekáváš, že si k tomu pivu budeš moci sednout?“
BS: „Ano.“
…
```

Ten rozhovor může pokračovat ještě hodně dlouho a nezáleží na tom, koho se ptáte, nebo jaké jsou odpovědi. Jsou to otázky, které člověk klade, které z něj dělají testera.

9.2 Pozice v testovacím týmu

Na větších projektech, na které nestačí jeden tester, se testeři sdružují do testovacích týmů, v rámci nichž mají určené své role. Spektrum těchto rolí nebo pozic je dáno firemní kulturou a používanou metodikou. Nejčastěji se objevují hierarchické názvy pozic:

- Tester (junior / senior)
- Test designér
- Test analytik
- Test manager

Při diskuzi lidí z různých firem, popřípadě dokonce z různých týmů, je vhodné si vždy nejprve ujasnit, co se pod kterou rolí – pozicí – skrývá. Bez zacházení do detailů bývají pravomoci a zodpovědnosti k pozicím přidělovány následujícím způsobem. Tester slouží jako obecné označení člověka, který se zabývá testováním. V hierarchii rolí v testovacím týmu ale zastává nejnižší pozici. Zpravidla pak provádí nenáročnou práci, která je buď co nejpřesněji definovaná nebo není kritická. Zkušenější testeři, kterým je už přiznána jakási zodpovědnost, bývají test senioři nebo test designéři, kteří kromě provádění testů, je i sami navrhují. Ještě důležitější pozici zastává test analytik, který už velkou měrou zodpovídá ze testování a připravuje ho. Test manager, kterého občas nahrazuje projekt manager pak testování řídí.

Kromě rozdělení rolí na základě hierarchie se testeři odlišují i svou specializací. Vybírají si skupinu druhů testů, které jsou pro ně zajímavé, a na jejich provádění se pak

specializují. Mohou se zaměřit i na konkrétní druh aplikace, jazyk nebo nástroj. Takováto zaměření ale bývají méně častá, jelikož potřeba specializace vychází hlavně z nemožnosti ovládnout do hloubky všechny aspekty různých druhů testů.

9.3 Tester versus programátor

František Kunský se ve své diplomové práci Testování softwaru a testovací role [5] snaží přirovnávat testera a programátora. Přirovnávat k sobě dvě profese je velmi ošidné, to se rovněž můžeme ptát, zda je tester méně nebo více než astronom, právník nebo třeba politik. Samozřejmě zde jsou jisté schopnosti, které se vyžadují jak u programátorů, tak testerů a lidé pracující v těchto profesích spolu potřebují úzce spolupracovat. Jenže testeři toho mají hodně společného i s jinými profesemi. Kreativita a vynalézavost by měla být vlastní i umělcům a vědcům. Diplomatické jednání je vyžadováno i u právníků a politiků. Odvaha stát si za svým je ceněna napříč různými profesemi.

Zeptala jsem se proto několika expertů na testování z ciziny, jak je nahlíženo na pozici testera u nich a na jejich porovnání pozice testera a programátora zejména po stránce finanční a respektu, jelikož tyto měřítka jsou tradičně používána pro srovnávání profesí.

Tito odborníci se shodují, že testeři jsou méně respektováni a jejich práce obvykle zlehčována. Tento názor však podle nich zastávají lidé, kteří testování nerozumí a netuší, co obnáší. Práce dobrého testovacího týmu není tolik viditelná jako práce programátorů. Rovněž z průzkumu vyplývá, že představa o tom, kdo je to tester se liší mezi širší veřejností a odborníky.

Z emailu od Michaela Boltna:

„I think that testers are less respected than programmers in many places. It probably has more to do with the organization than with the country. It seems to be a world-wide thing.

Some people have suggested that being a good tester is harder than being a good programmer. The best programmers that I have ever known -- and the best testers -- are good critical thinkers. When people suggest that it's easier to be a good tester than a good programmer, I don't think that those people have thought about the issues very much.

When a tester is really good, few people tend to be aware of it. When a developer solves a big problem, many people get to see that and use the

program that she wrote. When a tester discovers big problems, those problems tend to get corrected before many people see them. So the value of a good tester is not always obvious. "

Přeloženo:

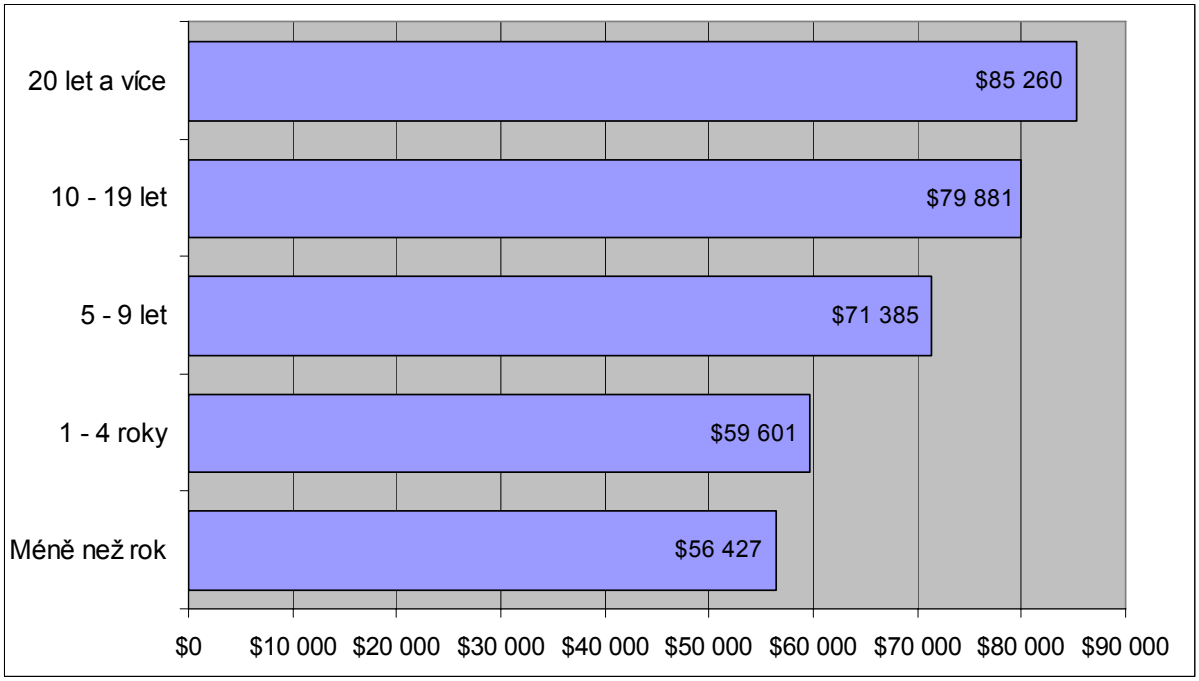
„Myslím, že testeři jsou méně respektováni než programátoři na mnoha místech. Pravděpodobně to má co do činění s organizací než se zemí. Zdá se, že je to celosvětová věc.

Někteří lidé naznačují, že být dobrým testerem je těžší než být dobrým programátorem. Nejlepší programátoři, co znám -- a nejlepší testeři -- jsou dobří v kritickém myšlení. Když lidé naznačují, že je lehčí být dobrým testerem než dobrým programátorem, nemyslím, že tito lidé o tom moc přemýšleli.

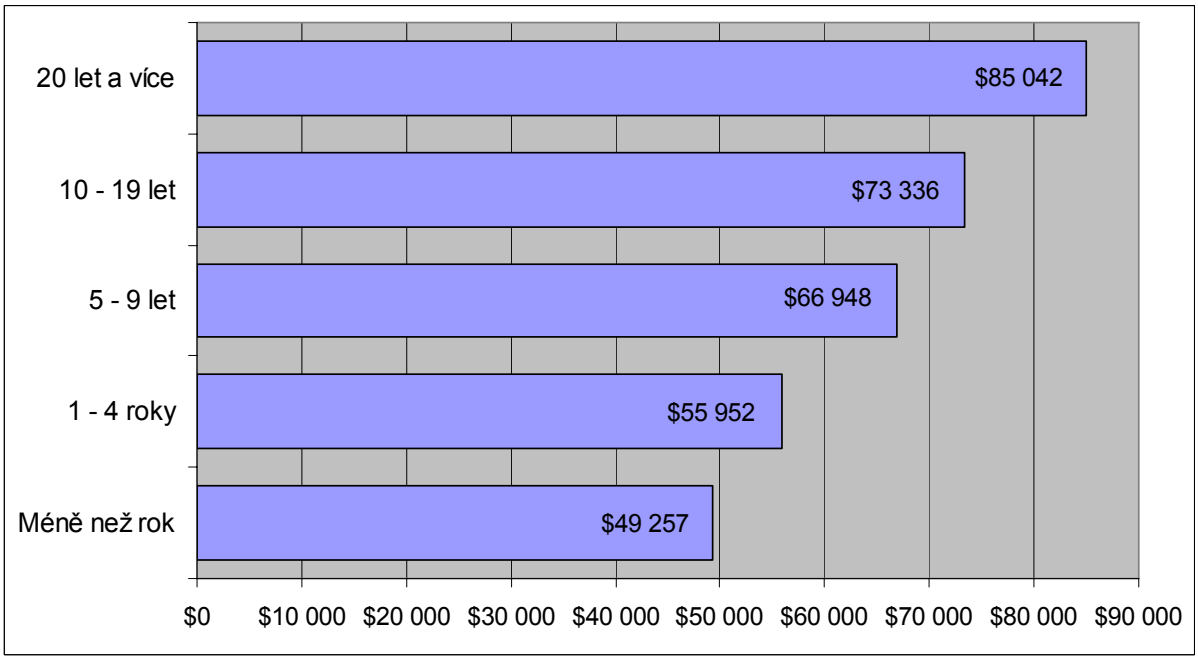
Když je tester velmi dobrý, málo lidí si je toho vědomo. Když vývojář vyřeší velký problém, hodně lidí to vidí a používají program, který napsal. Když tester objeví velké problémy, tyto problémy bývají opraveny dříve, než si jich hodně lidí všimne. Tedy hodnota dobrého testera nebývá vždy zřejmá."

Za testera bývá hlavně považován člověk, který je sebrán z ulice, dostane krok za krokem popsáno, co má dělat s produktem a tento scénář do puntíku plní. Oproti tomu experty na testování je za testera považován člověk s vlastnosti popsanými v kapitole Vlastnosti testera a s nadšením k testování. Takovýto člověk, scénáře popisující co s programem dělat, píše a sám je aktivně zkoumá.

Porovnáme-li čistě platy testerů a programátorů jako je tomu na obr.5 a obr.6, vidíme, že medián platů testerů je nižší.



Obr. 6: Mediány platů podle zkušenosti v letech na pozici softwarový inženýr/ vývojář / programátor v USA [33]



Obr. 7: Mediány platů podle zkušenosti v letech na pozici test / quality assurance inženýr v USA [34]

Na druhou stranu nesmíme zapomínat, že se jedná o mediány a plat závisí od celé řady schopností. Testeři ovládající dobře programování nebo mající jiné klíčové znalosti mohou dostávat větší plat než by dostal stejně zkušený programátor.

Například z bývalých hackerů se často stávají spíše testeři zaměřující se na bezpečnost než programátoři. "Jedna možnost je chovat se příkře. Jiná je uznat, že tito lidé jsou zažraní do bezpečnosti," řekl Kevin Kean, ředitel Microsoftu pro bezpečnostní odezvu, o hackerech. [23].

Z emailu od Cema Kanera[7]:

„Students from my lab who had a combination of programming and testing skill, but who were willing to work as testers have often received job offers for more money than comparable students for programming jobs."

Přeloženo:

„Studenti z mojí třídy, kteří mají kombinaci programovacích a testovacích dovedností, ale kteří jsou ochotni pracovat jako testeři, často dostávají pracovní nabídky s větším platem než porovnatelní studenti za programování."

Z emailu od Pradeepa Soundararajana:

"Bad testers are obviously paid less and good testers are paid more. For instance, I (I claim to be a good tester) get paid at least thousand dollars more than a good developer of my experience."

Přeloženo:

"Špatní testeři jsou očividně placeni méně a dobří testeři více. Například já (považuji se za dobrého testera) dostávám zaplaceno alespoň o 1000 dolarů více než dobrý vývojář se stejnou zkušeností."

V knize Software testing fundamentals Marnie L. Hutcheson vykládá o opačné zkušenosti, kdy testerem se stávali ti nejlepší programátoři a zároveň zmiňuje jeden z možných důvodů, proč testování není adekvátně uznávané [2]:

"One of my first mentors when I started testing software systems had been a tester in a boom-able industry for many years. He explained to me early on how a very good analyst could get promoted to programmer after about five years of reviewing code and writing design specifications; then after about five years in development, the very best programmers could hope

[7] Cem Kaner je profesorem softwarového inženýrství na floridském institutu technologie a ředitelem floridského technologického centra pro vzdělávání a výzkum v oblasti testování softwaru.

for a promotion into the system test group. The first two years in the system test group were spent learning how to test the system.

…

Few universities offer software testing classes. Even fewer require software testing classes as part of the software engineering curriculum. Unfortunately, this sends the message to business and the development community that software testing is not worthwhile."

Přeloženo:

"Jeden z mých prvních mentorů, když jsem začala testovat softwarové systémy, byl testerem v rozvíjejícím se průmyslu po mnoho let. Brzy mi vysvětlil, jak velmi dobrý analytik může být povýšen na programátora po pěti letech čtení kódů a psaní specifikací návrhu; pak po asi pěti letech ve vývoji ti nejlepší programátoři mohou doufat v povýšení do skupiny systémového testování. První dva roky ve skupině systémového testování stráví učením se jak systém testovat.

…

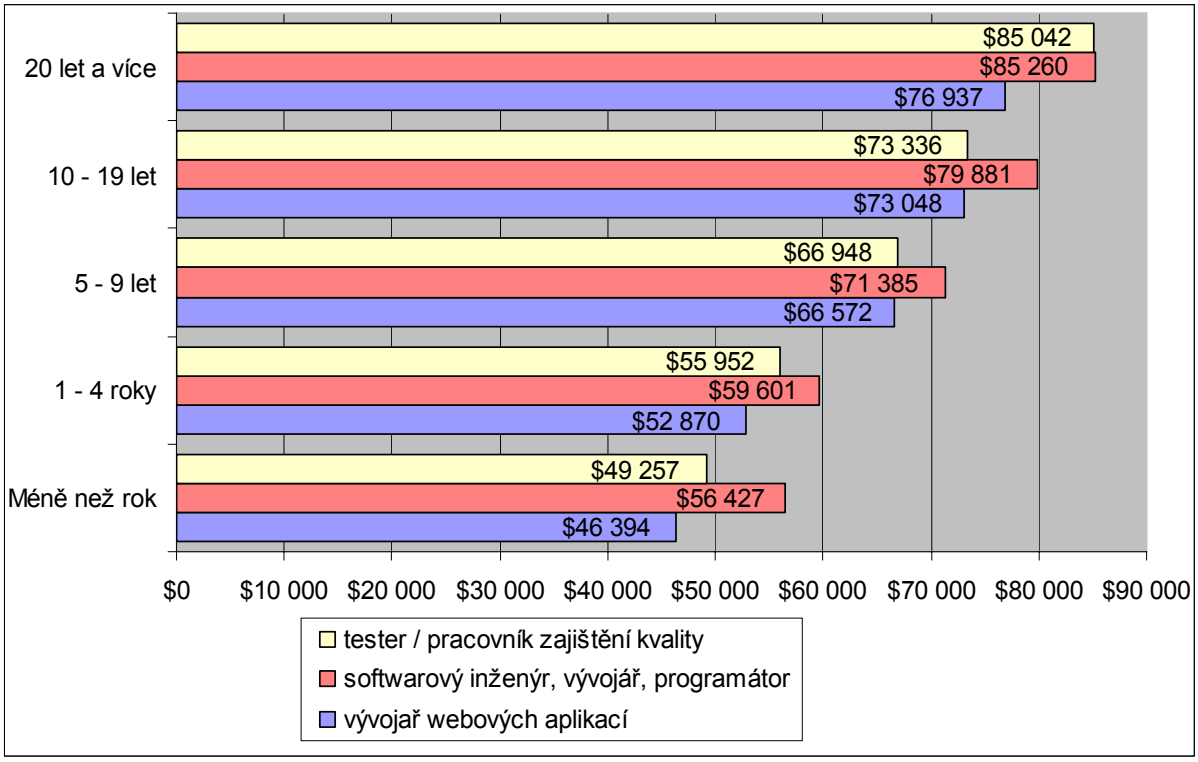
Málo univerzit nabízí semináře testování softwaru. Dokonce ještě méně jich vyžaduje testování softwaru jako součást vzdělání v softwarovém inženýrství. Naneštěstí tím zasílají vzkaz byznysu a vývojářské komunitě, že testování nestojí za to."

Podle tohoto zdroje [2] bylo odhadováno, že ke konci devadesátých let dvacátého století, tedy zhruba před deseti lety, pouze deset procent testerů a vývojářů má za sebou nějaké školení v technikách testování.

V české republice během těchto deseti let vznikla řada školení ohledně testování, kvalita a zejména rozsah těchto školení však stále není dostačující a představuje jen zlomek vědomostí, které by zkušenější tester měl mít.

Z tabulek a dalších průzkumů vyplývá, že plat zkušených testerů často přesahuje plat zkušených programátorů, důvodem k nižšímu mediánu zřejmě je i to, že opravdu schopných testerů je v poměru k celkovému počtu testerů méně než jak je tomu u programátorů.

Stejný jev můžeme ale pozorovat i u různých druhů vývojářů. Například řada programátorů v dnešní době začíná tvorbou webových stránek, medián platů programátorů webových aplikací tak vychází dokonce nižší než u testerů.



Obr. 8: Mediány platů podle zkušenosti v letech na pozice vývojáře, vývojáře webových aplikací a testera [22-24]

10. Kategorie testů

Jeden z prvních problémů při testování je rozhodnutí jak testovat, určení

- jaké druhy testů budou použity,
- kdy a do jaké míry budou aplikovány,
- jaké nástroje a data je třeba připravit.

Vzniklé množině testů se pak říká testovací mix. Spektrum možných testů je velmi široké, proto se třídí do kategorií podle různých hledisek. Obecně platí, že při plánování testů by žádná kategorie neměla být opomenuta a alespoň jeden její zástupce by se měl nacházet v testovacím mixu. O jednom z dělení testů už byla zmínka v předešlých kapitolách a to o statickém a dynamickém testování.

10.1 Statické a dynamické testování

Toto rozdělení vychází z toho, zda je k provedení testu potřeba software spustit. Statické testování nevyžaduje běh softwaru, proto je možné s ním začít ještě před vytvořením prvního prototypu. V současné době je poněkud opomíjeno, což je škoda, protože rychle objevuje chyby, které by se mohly stát kritickými. Navíc je ho možné aplikovat i na složitě otestovatelné části, ke kterým by testeři museli komplikovaně vytvářet dynamické testy a úroveň kvality testování těchto částí by byla nedostačující nebo by byla dosažena s většími náklady. Správně provedené statické testování, například procházení kódu, bývá velmi efektivní při hledání určitých druhů chyb. Dále budou podrobněji probrány některé statické techniky v samostatné kapitole.

Dynamické testování vyžaduje existenci spustitelné verze softwaru a probíhá hlavně na základě poskytování různých vstupů a posuzování výstupů testovaného programu.

10.2 Černá a bílá skříňka

Poněkud odlišné je rozdělení testů podle toho, na základě jakých znalostí o produktu k němu přistupujeme. Při testování černé skříňky se zaměřujeme na vstupy a výstupy programu bez znalosti, jak je naimplementován. Produkt je černou skříňkou, do které se nelze podívat, vidíme jen jak vypadá a jak se chová navenek. Smyslem je analyzovat chování softwaru vzhledem k očekávaným vlastnostem tak, jak ho vidí uživatel. Do této kategorie

spadají skoro všechny druhy testů uživatelských rozhraní, akceptační testy, testování podle scénářů, které krok za krokem provádějí uživatele tím, co má zadat a jaké jsou očekávané reakce systému.

Při testování bílé skříňky, v angličtině se jí říká taky skleněná (glass box) nebo průhledná (clear/transparent box), má tester přístup ke zdrojovému kódu a testuje produkt na základě něj. Vidí nejen co se děje na povrchu skříňky, ale i vnitřní reakce systému. Tím poněkud ztrácí pohled uživatele, ale může lépe odhadnout, kde hledat chyby a kde ne.

Mezi těmito kategoriemi vznikla ještě třetí, testování šedé skříňky, kdy víme něco málo o implementaci a vnitřních pochodech produktu, ale ne tolik, aby to bylo považováno za testování bílé skříňky. Například nemáme k dispozici celý zdrojový kód, ale html kód webových stránek nebo informace o designu aplikace.

10.3 Automatické a manuální testovaní

Podle toho, zda jsou testy prováděny člověkem nebo softwarem, se rozlišuje manuální a automatické testování. Pokud test vyžaduje lidské ohodnocení a úsudek nebo rozličné přístupy, které není třeba zaznamenat a pravidelně opakovat, je vhodnější manuální testování. Pro opakované spouštění velkého množství testů nebo testu s velkým množstvím generovaných dat, stejně jako pro zátěžové testování je dobré použít automatické testy. Důležitým hlediskem je taky to, zda vytvořené automatické testy budou levnější, než ručně provedené testy.

10.4 Stupně testování

Na základě toho na jaké úrovni se testování provádí, v jakém časovém horizontu od napsání kódu, se testování dělí na pět stupňů:

- Testování programátorem (Developer testing)
- Testování jednotek (Unit testing)
- Integrační testování (Integration testing)
- Systémové testování (System testing)
- Akceptační testování (Acceptance testing)

Jednotlivé stupně často provádí různí lidé a kód je na každém stupni testován z trochu jiného úhlu pohledu.

První se provádí **testování kódu programátorem** bezprostředně po jeho napsání, přičemž testování má ponejvíce podobu neformálního komponentového testování, které je založené na principu: Otestuj, co jsi napsal. Málokterý programátor by odevzdal kód aniž by ho alespoň jednou nespustil a nevyzkoušel.

Cíle tohoto testování jsou dva. Zaprvé programátor kontroluje, jestli kód dělá to, co on zamýšlel. Zadruhé hledá jestli kód funguje správně, protože v tomto okamžiku je oprava vzniklých chyb nejméně nákladná.

Na rozdíl od testování programátorem, je kód v následujících fázích testován z hlediska uživatele, zda dělá to, co uživatel požaduje nebo očekává.

Testování jednotek je proces podrobného testování co možná nejmenších částí kódu. U objektově orientovaného kódu se jedná o testování jednotlivých metod a tříd. Protože tyto testy se zapisují rovněž ve formě programového kódu, můžou je provádět programátoři pod dohledem testovacího týmu, který vyhodnocuje výsledky, nebo testeři se znalostmi programování. Pro usnadnění a rychlejší psaní těchto testů se používají nástroje na bázi frameworků. Tyto testy jsou automatické, snadno opakovatelné a dobře se zpracovávají. Vzniklé testy se nazývají test unitami. Pokud jsou napsány předem, jako tomu je u testy řízeného vývoje, můžou nahradit testování kódu programátorem.

Příklad kódu testu napsaného pomocí JUnit frameworku v rámci testování jednotek :

```
import junit.framework.*;

public class ChangeTest extends TestCase {

private Money old;
private String newCur1;
private String newCur2;
private float exRate;
Rates rateC = new Rates();

public ChangeTest(String name) {
      super(name);
}
```

```
// příprava prostředí na test
protected void setUp() throws Exception {
      old = new Money(25, "CZK");
      newCur1 = new String("CZK");
      newCur2 = new String("EU");
      exRate = rateC.getRateCZtoEU();
}

// úklid prostředí po testu
protected void tearDown() throws Exception {

}

public void testChangeSame() {
      Money expected = new Money(25, "CZK");
      assertEquals(expected, old.change(newCur1));
}

public void testChangeDiff() {
      Money expected = new Money(25*exRate, "EU");
      assertEquals(expected, old.change(newCur2));
}

}
```

Integrační testování se zaměřuje na zajištění správného chování nových podsystémů poté, co jsou zapojeny společně s ostatními částmi systému. Integrační testy mohou být jednoduché a automatické, ale i komplexní a manuální, záleží na situaci a systému. Testy připravuje hlavně testovací tým, a pokud se jedná o testy automatické, spouštět je pak může kdokoli.

Během **systémových testů** je aplikace testována jako celek, a proto tyto testy bývají prováděny v pozdějších fázích vývoje. Pokud se jednotlivé testovací týmy předtím zaměřovaly jen na určité funkčnosti, které spadaly pod jejich kompetenci, jako tomu bylo v případu Mars Polar Lander, při systémových testech by měly být testovány obsáhlé scénáře procházející napříč celým systémem a simulována běžná práce uživatele se systémem. Součástí této fáze je provádění řady různých druhů testů jako jsou testy výkonnosti, spolehlivosti, bezpečnosti atd. Tyto testy bývají vytvářeny testovacím týmem, který by měl obsahovat specialisty na zastoupené druhy testů.

Akceptační testy jsou součástí převzetí produktu zákazníkem a ověření, že software je připraven na nasazení do ostrého provozu. Často bývají prováděny podle scénářů, které jsou připraveny předem a dohodnuty oběma stranami. Software je při tom testován zákazníkem nebo jím pověřeným testovacím týmem. V rámci přípravy můžou být tyto testy simulovány i ze strany dodavatele, který zjišťuje, nakolik je systém připraven a hledá poslední chyby.

10.5 Pokrytí testy

Další možným dělením testů je podle toho, jakou část vyvíjeného softwaru pokrývají a jakým způsobem. Pokrytí je zároveň i metrikou, která nám říká, kolik z produktu jsme už otestovali a kolik nám zbývá. Pokrytí kódu se měří v procentech, kolik z celkového počtu sledovaného je pokryto testy. Nejprve je třeba se rozhodnout, jakým způsobem budeme pokrytí testy hlídat. Pokrytí bereme v úvahu vzhledem k

- požadavkům,
- funkcím,
- a zdrojovému kódu.

Zatím co první dva druhy pokrytí se chápou intuitivně, třetí typ - pokrytí kódu, v angličtině pod názvem code coverage, bývá popsán formálněji a rozdělen na další podtypy. U kritických aplikací, kde se klade důraz na bezpečnost a spolehlivost, je běžným požadavkem vysoké až úplné pokrytí kódu některého typu. Pokrytí kódu byla jedna z prvních technik systematického testování a bývá spojováno s automatickým testováním. Například právě jedna z výhod testovaní jednotek je, že umožňuje sledování pokrytí kódu. Existuje však řada způsobů, které se liší v pečlivosti s jakou požadují, aby byl kód otestován.

Pokrytí příkazů nebo taky řádků je tou nejjednodušší variantou. Pouze kontroluje, zda daný příkaz, řádek byl proveden v průběhu testování. Formálně množina testů T splňuje kritérium pokrytí příkazů pro daný kód K, jestliže pro každý příkaz p náležící kódu K existuje test t z množiny T, že při provedení t bude spuštěn příkaz p.

Pokrytí příkazů pouze kontroluje, zda existuje vůbec nějaký test na daný příkaz, ne pečlivost testu. Tento rozdíl je patrný u jakýchkoli podmínek, kdy pro pokrytí jejího příkazu stačí libovolné její vyhodnocení.

Příklad stoprocentního pokrytí příkazů bez zjištění chyby:

```
int* p = NULL;
if (x>0)
      p = &x;
*p = 1;
```

Při otestování tohoto kódu pro x=5 dosáhneme pokrytí všech příkazů. Ale není otestována varianta, kdy podmínka selže, takže ani nemusíme zjistit, že program selže při zapisování čísla na neplatnou adresu.

Pokrytí hran (větví, rozhodnutí) požaduje, aby v případě podmínek byly testovány jak pro kladné tak záporné vyhodnocení. Představme si graf, kdy příkazy tvoří uzly a možné přechody mezi nimi hrany. Pak za sebou provedené příkazy tvoří hranu a podmínka uzel, ze kterého vedou dvě hrany, jedna pro true a jedna pro false hodnotu. Pak množina testů T splňuje kritérium pokrytí hran pro daný kód K, jestliže pro každou hranu h výše popsaného grafu existuje test t z množiny T, že při provedení t projde výpočet hranou h.

Přestože toto pokrytí je o něco podrobnější, u složitějších podmínek může dojít opět k pokrytí bez detekování chyby.

Příklad stoprocentního pokrytí hran bez zjištění chyby:

```
if (a>0 && (b>0 || c>0))
a = a+b/c
```

Pokud mezi testy zahrneme případy, kdy podmínka selže, protože například a je -5, a případ kdy a, b i c jsou kladné, dosáhneme úplného pokrytí hran, ale neodhalíme v tomto případě možné dělení nulou z důvodu špatně zapsané podmínky.

Pokrytí podmínek pak vyžaduje nejen kladné a záporné vyhodnocení těchto podmínek, ale jedná-li se o složenou podmínku, také všech podčástí. Množina testů T splňuje kritérium pokrytí podmínek pro daný kód K, jestliže splňuje kritérium pokrytí hran a pro každou složenou podmínku a každou její část p, existují testy t, u z množiny T, že při provedení t se p vyhodnotí kladně a při provedení u záporně.

Pokud se vynechá část, že pokrytí podmínek musí splňovat pokrytí hran, tak k nepokrytí hran a pokrytí podmínek dochází v případě, že pro všechny možné vstupy je vždy celá podmínka vyhodnocena jako kladná nebo záporná.

Příklad stoprocentního pokrytí podmínek bez zjištění chyby:

```
if  x <> 0  then  y = y – x;
if  y > 1  then y = y/x;
```

Pokud množina testů bude obsahovat testy, kde x je různé od nuly, nebo je sice nula, ale y není větší než jedna, bude pokrytí podmínek úplné, ale chyba dělení nulou touto množinou testů nebude odhalena.

Pokrytí cest sleduje, zda jsou otestovány všechny možné průchody kódem. Množina testů T splňuje kritérium pokrytí cest pro daný kód K, jestliže splňuje kritérium pokrytí podmínek a pro každou cestu C v grafu kódu spojující vstupní a výstupní uzel grafu a obsahující nejvýše n cyklů existuje test t z množiny T, že při provedení t projde výpočet cestou C.

Pokrytí cest už představuje velmi pečlivé otestování kódu, ale jeho použití má velká omezení. Jednak počet cest programem roste až exponenciálně a jednak ne všechny cesty je možné provést z důvodu závislosti mezi daty a příkazy.

Existují i další typy pokrytí kódu, které nejsou tak časté a které se většinou specializují na určité části kódu, jako například pokrytí tabulek, cyklů, volání atd.

Více o pokrytí kódu je napsáno v [18], definice v této části vycházely z materiálu [8].

10.6 Dimenze kvality

Další dělení testů vychází z něčeho, čemu se říká dimenze kvality. Jsou to různé aspekty vyvíjeného produktu, ke kterým se vztahují požadavky a očekávání uživatelů. Jejich seznam je důležitý hlavně proto, aby nebylo opomenuto sesbírat požadavky a připravit testy pro žádnou z těchto dimenzí.

Dimenze kvality:

- Funkčnost (**F**unctionality) – zajišťuje správné chování funkcí systému
- Použitelnost (**U**sability) – zda je systém uživatelsky přívětivý, zda se s ním dobře pracuje
- Spolehlivost (**R**eliability) – zda se chová stejně za všech okolností, zvláště po přetížení, po chybě
- Podporovatelnost (**S**upportability) – zda se systém dobře instaluje, nemá problémy s cílovými hardwarovými a softwarovými konfiguracemi a další vlastnosti související s údržbou systému.
- Výkon (**P**erformance) – zda systém není pomalý a zvládne větší množství současně pracujících uživatelů
- Ostatní (+) – Zde patří třeba lokalizovatelnost - snadný převod do jiných jazyků, kompatibilita – možnost kombinace s jiným softwarem nebo hardwarem, bezpečnost a všechny další požadavky, které se nehodí do předchozích kategorií.

Tyto dimenze bývají někdy označovány zkratkou FURSP+, kterou tvoří začáteční písmena anglických názvů jednotlivých dimenzí.

11. Dokumentace

Součástí procesu testování je neodmyslitelně tvorba dokumentace, bez které by se projekt potýkal s řadou problémů, hlavně v komunikaci a se ztrátou znalostí při fluktuaci lidí v testovacím týmu. Dalo by se říct, že o testování se dá hovořit pouze v tom případě, když existuje alespoň nějaká forma dokumentace. V případě menších projektů může být dokumentace i hodně neformální, stačí zaznamenávání reportů chyb a jejich stavů, případně navíc plán toho, co se bude testovat a jakými druhy testů, vyvěšený na nástěnce.

V jiných případech bývá dokumentace naopak značně podrobná s přesně definovaným obsahem, připravovaná podle metodiky nebo dokonce standardů. Přímo tvorbou testovací dokumentace se zabývá standard IEEE-829, který je z tohoto důvodu pro účely dokumentace nejznámější. Často je možné se také setkat se standardem ISO 9000-3, která se zabývá kvalitou softwaru.

Rozhodnutí o formě dokumentace a použití standardů záleží na požadavcích vývojového a testovacího týmu a zákazníka. Jako návod mohou posloužit odpovědi následující otázky:

1) **Je dokumentace testování vaším produktem nebo nástrojem?** Pokud slouží testovací dokumentace pouze pro tým, záleží hlavně na jeho požadavcích, které na dokumentaci klade. Dokumentace se bude vyvíjet tak, aby poskytovala co největší užitek za co nejmenší cenu. Při psaní se bude hlavně hledět na účel, který má splnit. Oproti tomu, pokud zákazník, pro kterého je softwarový produkt vyvíjen, bude požadovat odevzdání testovací dokumentace a zaplatí si za ni, stane se tato dokumentace samostatným produktem. Pak je to zakázka, které je třeba věnovat patřičnou pozornost a dokumentace bude odpovídat jakémukoli standardu nebo metodice, jakou zákazník bude požadovat.
2) **Odvíjí se požadovaná kvalita od smyslu aplikace nebo je určována požadavky trhu?** Pokud smyslem aplikace je ovládání procesů, které mohou ohrozit lidský život nebo způsobit obrovské ztráty, může být testovací dokumentace vyžadována při auditech nebo u soudních pří a být důkazem, že společnost při vývoji nic nezanedbala. Pokud bude rozhodovat hlavně poměr kvalita/cena, kterého vyvíjená aplikace dosahuje, bude opět na prvním místě užitečnost. Dokumentace by měla poskytovat všechny aspekty, které testovací tým potřebuje, ale nebude se vytvářet nic navíc, bez

čeho by se dalo lehce obejít. Při uvažování o tom, co vše se vyplatí vytvářet, je dobré zvážit odpovědi na další otázky.

3) **Jak často se bude měnit specifikací požadované chování aplikace nebo vzhled? Jsou požadavky jasné? Očekávají se v budoucnu časté změny?**
   Podrobná dokumentace bude při častých změnách rychle stárnout a bude brzo stejně neplatná nebo její údržba zabere větší procento času testovacího týmu. Oproti tomu při stabilním návrhu aplikace je možné čerpat z výhod podrobnější dokumentace. Například by se mohlo vyplatit psaní a zdokumentování automatických testů, u kterých se dá v této situaci usoudit, že s minimální údržbou postupně ušetří týmu více času, než kolik stála jejich tvorba.
4) **Jak dlouho bude dokumentace používána?**
   Při psaní stejných dokumentů, z nichž jeden bude používán měsíc a druhý rok, a ani u jednoho se nepředpokládá, že by během této doby přestal být aktuální, je pravděpodobné, že podrobnější text se vyplatí u toho dlouhodobějšího dokumentu. Jednak z toho důvodu, že vynechané informace by se díky fluktuaci členů nedaly tak lehce zjistit jinde. A jednak proto, že zapsání podrobností ušetří čas, který by čtenáři strávili zjišťováním podrobností. Za rok si daný dokument bude potřebovat přečíst více lidí než kolik lidí si přečte druhý dokument za měsíc, ale zapsání podrobností v obou případech bude trvat stejně.
5) **Může už nějaká existující dokumentace alespoň částečně nahradit dokumentaci testů?** Někdy jsou vytvářeny scénáře, vedoucí uživatele k tomu, jak přesně má aplikaci testovat, a popisující, jaké je její očekávané chování. Tyto scénáře jsou jedním z dokumentů testování. Jindy se spíše spoléhá na úsudek a schopnosti testerů, na čemž je založené například průzkumné testování, kdy tester poznává a analyzuje aplikaci zároveň. V takovém případě se scénář, jak má tester postupovat, nepíše, a pro případné zjištění informací o aplikaci je možné využít specifikací, uživatelských manuálů a dalších už existujících ale aktuálních dokumentů.
6) **Komu všemu je dokumentace testování určena? Bude se využívat pro školení, výzkum či jiné činnosti nepatřící do testovacího procesu?** Je-li co nejlépe odhadnut přínos dokumentu, dokážeme snadno odhadnout, jestli se vyplatí ho vytvořit a kolik času na jeho napsání věnovat.

Pomoci při rozhodování mohou i měřítka zabývající se cenou dokumentace. Sleduje se zejména, kolik stojí tvorba dokumentace testování na jeden testovací případ nebo jeden případ

užití a kolik stojí údržba testování na jeden testovací případ nebo jeden případ užití. O use casu nebo-li případu užití se dá říci, že představuje jednu konkrétní funkčnost softwaru, používá se při use case analýze, která bývá součástí specifikace.

11.1 Nejdůležitější dokumenty podle praxe

Testovací plán je dokument řídící celý proces testování, proto téměř všechny zdroje považují testovací plán za nejdůležitější dokument. To ale je pravdou pouze pro samotný proces, řada projektů se bez něj obejde nebo dokonce testeři ani neví, zda takový plán existuje a kde ho najít.

Testovací plán obsahuje určení rozsahu testování, definuje jaké testy budou na co aplikovány, jaké zdroje je třeba zajistit včetně nástrojů, dat a lidí, určuje zodpovědnosti, stanovuje pravidla a identifikuje rizika.

Testovací plán bývá na větších projektech prvním vytvářeným dokumentem, připravuje se dřív, než se začne testovat, někdy bývá některou řídící pozicí v rámci testování vytvořen ještě dřív než je stanoven tým testerů.

Seznam testovacích nápadů nebo myšlenek (test idea list) je dokument, do kterého si testeři píší své návrhy na testy, které by stály za to vyzkoušet při testování. Tyto nápady jsou roztříděné na základě toho, který prvek aplikace zkoumají. Testovací myšlenkou je např. dělení nulou, zadání speciálních znaků, ovládání menu klávesnicí.

Testovací případ, známý i pod anglickým označením test case je seznam kroků testujících jeden konkrétní případ, který při používání testované položky může nastat. Součástí je i doprovodný výčet množiny vstupů, podmínek a očekávaných reakcí. Testovací případy vznikají na základě testovacích nápadů a případů užití získaných např. ze specifikace.

Příklad testovacího případu - pokus o změnu hesla na aktuální heslo:

```
Počáteční bod: Hlavní stránka aplikace
Předpoklady: Aktuální platné heslo uživatele je A456pk

Menu -> Osobní nastavení -> Změna hesla
Do políčka Aktuální heslo zadejte:  A456pk
```

```
Do políčka Nové heslo zadejte: A456pk
Do políčka Potvrzení hesla zadejte: A456pk
Potvrďte tlačítkem OK.

Očekávaná reakce: Aplikace zahlásí chybovou hlášku: „Dané heslo už
bylo v minulosti použito.“ Heslo není změněno, což se projeví mimo jiné
tak, že nedojde k žádným změnám v historii hesel.
```

Testovací scénář slučuje několik za sebou jdoucích testovacích případů do tématicky uspořádaného scénáře, např. testování změny hesla.

Testovací skript je přepis testovacího příběhu nebo scénáře do programovacího jazyka, scénáře jsou určené k automatickým testům aplikace.

Testovací data jsou sebraná skutečná nebo podle předem určených pravidel vygenerovaná data určená pro účely testování. Přestože nemívají většinou podobu dokumentu, v praxi bývají považována za součást dokumentace.

Hlášení chyby je oznámení popisující možnou chybu, postup vyvolání jejího projevu, její dopad, význam, stav opravy a další náležitosti chyby. Hlášení chyb je možno považovat za produkt testování, proto je množina těchto reportů ve skutečnosti nejdůležitějším dokumentem.

Test log slouží k zaznamenání proběhlých testů. Mívá tabulkovou podobu s informacemi kdo, kdy, jaké testy spustil a s jakým výsledkem.

Test result dokument obsahuje souhrnné informace o provedeném testování typicky za jeden testovací cyklus, začínající odevzdáním nové verze vyvíjeného produktu testovacímu týmu. Test result dokument poskytuje podrobné hodnocení kvality dané verze a zvoleného přístupu k testování.

Test evaluation dokument zpětně hodnotí proces testování, nejen za jednotlivé testovací cykly ale za dobu celého vývoje. Je to dokument na stejné úrovni jako testovací plán, ale dívá se na problém z opačného konce, poskytuje zpětnou vazbu, jak byl proces testování úspěšný a co je možné do příště zlepšit.

11.2 Nejdůležitější dokumenty podle standardu

Standard IEEE 829 definuje osm dokumentů pro proces testování softwaru, standard přitom ale nevyžaduje používání celé množiny, pouze popisuje jejich podobu.

- Testovací plán
- Test design specification: definuje zvolený testovací přístup, co se bude testovat a jak
- Test case specification: obsahuje vytvořené testovací případy
- Test procedure: popisuje spouštění testů .
- Test item transmittal report: specifikuje které části aplikace mají být otestovány. Vytváří se zvláště, pokud testuje několik týmů nebo je vývoj rozvětven do několika souběžných projektů
- Test log
- Test incident report: obecnější forma hlášení chyby. Reportuje se jakákoli událost vzniklá při testování
- Test summary report: Stejné jako test report, hlásí souhrnné výsledky testování a hodnotí proces testování.

11.3 Testovací nápady

Seznam testovacích nápadů může obsahovat úrovně pohledu, čím je představa o aplikaci konkrétnější, tím podrobnější můžou nápady být. V případě, že vytváříme seznam testovacích myšlenek pro celou kategorii webových aplikací, vzniklé nápady budou mít spíše podobu druhů testů, např. testování aplikace s různou rychlostí připojení k internetu, na různých prohlížečích a jejich nastavení, atd.

Testovací nápady vytvořené pro určitý typ vstupu, jako je pole očekávající pouze celočíselné hodnoty, jsou naopak velmi konkrétní a lehce znovupoužitelné na všechna taková pole obsažená i v jiných aplikacích.

Z tohoto důvodu nejčastěji bývají generovány myšlenky pro konkrétní aplikaci, se kterou je tester dobře seznámen.

Seznam testovacích nápadů se nejlépe vytváří za pomoci techniky brainstormingu poblíž počítače dostupného k internetu a za pomoci předem připravené struktury kategorií podle které se nápady budou generovat. Ta slouží k zacílení přemýšlení určitým směrem což podporuje vymýšlení nových nápadů.

Příklad krátkého seznamu nápadů vygenerovaných se zaměřením na aplikaci provozující internetový obchod [43]. Vytvoření seznamu testovacích nápadů zabere několik hodin nebo i dní, tento 15 minutový výstup proto nemůže být kompletní:

```
Strávený čas: 15 minut brainstormingu.
-------Start--------------------
Špatná použitelnost:
```

- Uživatel nemůže přidat položku do košíku přímo z výsledků vyhledávání.
- Uživatel neví kolik položek má v košíku a nezná celkovou cenu v každém časovém okamžiku.
- Uživatel musí projít příliš mnoho stránek než dokončí objednávku.
- Je obtížné používat systém: složité přidat, odstranit a aktualizovat.
- Není vidět konečná cena nebo její odhad.
- Vyhledávání je příliš složité nebo je těžké najít pole pro vyhledání na stránce.
- Nelze najít nápovědu.
- Formulář pro zpětnou reakci zákazníků je nedostupný.

```
Výpočetní chyby:
```

- Odstranění/přidání položky do košíku neaktualizuje celkovou cenu.
- Záporný počet položek sníží celkovou cenu.
- Slevy nejsou správně započítány.
- Poštovné a daně nejsou správně započítány.
- Funkce přepočítání selže.

```
Lokalizace:
```

- Pole registračního formuláře neakceptují rozšířené znaky cizích abeced.
- Pokud jsou tyto rozšířené znaky zadány při registraci, naruší se jimi databáze.
- Není možné aplikaci přeměnit na vícejazyčnou.
- Není možné přijmout objednávku do cizí země a integrovat poštovné do jiných zemí.

```
Selhání ISP/Web hosta:
```

- Uživatel se úspěšně zaregistruje nebo odešle objednávku, ale email s potvrzením mu nedojde.
- Nevratná ztráta dat na hostujícím centru.

- Zálohovací programy selžou na hostujícím serveru a způsobí ztrátu dat.

Síťové problémy

- Odkaz do databáze zboží selže.
- Odkaz do databáze uživatelských profilů selže.

Kompatibilita

- Stránky neodpovídají standardu HTML (W3C).
- Chybí kompatibilita s některými možnými platebními kartami nebo systémy.

Škálovatelnost[8]

- Přidání do vozíku, odeslání objednávky a vyhledávání trvá příliš dlouho během špičky.
- Žádostem vyprší doba platnosti během špičky

Bezpečnost

- Otestovat sílu šifrování.
- Otestovat zranitelnost vůči útoku přetečení bufferem.
- Otestovat zranitelnost vůči SQL injection.

Soukromí klienta

- Zkontrolovat, zda existuje politika na ochranu soukromí.
- Zkontrolovat vypršení platnosti cookies: zkontrolovat, zda někdo může přistupovat k obsahu košíku předchozího uživatele v případě sdílení počítače.
- Otestovat existenci automatického odhlášení v případě neaktivity.
- Neschopnost odmítnout účast v analýzách návštěvnosti a prodeje.

Selhání webového serveru

- Chybí uživatelsky přívětivá chybová stránka, v případě, že požadovaná stránka není nalezena.
- Server padá pod velkou zátěží.

Selhání softwaru třetích stran

- Selhání systému ověřujícího platnost platební karty

--------Konec----------

[8] Škálovatelnost je schopnost vypořádat se s růstem a změnami, bez nutnosti větších nebo vůbec nějakých zásahů do systému.

11.4 Reportování chyb

Reportování nalezených chyb je nedílnou součástí práce testera, která by se neměla odbývat. Srozumitelný dobře vystihnutý report může pomoci urychlit nalezení a odstranění chyby, zatímco špatný report bude přehazován i několikrát mezi testerem a vývojářem, který ho bude špatně interpretovat nebo vracet s žádostí o vysvětlení některých věcí. Nesrozumitelný report může stát až desítky hodin práce navíc, nebo chyba dokonce nebude opravena vůbec.

Osvědčeným způsobem jak zajistit kvalitní reporty je definovat jejich přesnou podobu, kterou je třeba dodržet a určit zkušeného testera, který bude za reporty a systém, kam se ukládají, zodpovědný. Reporty se mohou vytvářet a ukládat v textovém nebo tabulkovém editoru, ale častěji se používá speciálně k tomu vytvořený program. Těmto programům se říká bug report systémy nebo bugzilla.

Z povahy testování, jak byla vykreslena, vyplývají pro reporty dva důležité postřehy:

- Defekt report je u většiny testerů nejviditelnějším produktem jejich práce. Reporty mohou být jediná věc, co o testerovi bude spousta jeho nadřízených a kolegů znát. Jeho reputace tak bývá založena na tom, co do nich napíše.
- Nejlepším testerem není ten, který zahanbí nejvíce programátorů, ale ten, jemuž se podaří dostat co nejvíce chyb do opraveného stavu.

Dobře napsané hlášení chyby se snaží dodržet následující pravidla vždy, kdy je to možné:

- Je stručné,
- přesné a podrobné,
- napsané v neutrálním tónu (bez ironie, ponižování členů týmu),
- zaobírá se pouze jedinou chybou,
- ale postihuje ji v její nejobecnější podobě – zda má chyba i jiné projevy, zda se vyskytuje i jinde, např. zda je na jednom nebo všech formulářích,
- popisuje, jak je možné chybu reprodukovat
- a na koho chyba bude mít dopad,
- poskytuje co nejvíce informací pro hledání problému při ladění,
- předkládá důkazy o chybě.

Aby hlášení mohlo poskytnout všechny tyto informace, tak se po objevení chyby ještě před napsáním provádí takzvaný follow up testing (v češtině pod méně známým pojmem následné testování), které je zaměřené na zjištění skutečností výskytu a dopadu chyby. Toto následné testování trvá od 10 minut až do několika hodin v případě špatně reprodukovatelných chyb. Tyto špatně reprodukovatelné nebo se jim taky říká nereprodukovatelné chyby mizí a objevují se na první pohled zcela náhodně. Problém je v tom, že chyby se projevují pouze za jistých podmínek, pokud neznáme všechny podmínky, které je třeba nastavit, aby došlo k projevu chyby, vzniká dojem nereprodukovatelné chyby. V takovém to případě je nejlepším řešením si okolnosti chyby poznamenat bokem, promluvit si s programátory, zda neví, co by mohlo být další podmínkou k vyvolání chyby a při každém jejím opětovném výskytu, zkusit znovu nalézt postup, jak chybu nasimulovat.

Hlášení i zatím nereprodukovatelné chyby patří do bug report systému, protože je chybou a její oznámení i když není úplné, je užitečnou informací. Proces výroby softwaru ovlivňuje hodně lidí, a proto má i hodně lidí, kteří mají na něm zájem, jsou jeho stakeholdery. Pro každého z nich může být důležitý jiný aspekt kvality. Chyba patří do bugzilly, pokud ji tam některý ze stakeholderů chce, pokud v jeho očích snižuje hodnotu vyvíjeného softwaru.

Příklad hlášení chyby:

```
Shrnutí:          Rozhozený vzhled na mnoha místech aplikace
Závažnost:        C
Priorita:         1
Komponenta:       GUI
Prostředí:        Internet Explorer 6.0, SP2
Zadavatel:        Anna Borovcová
Datum nahlášení:  1.4.2007
Příloha:          C:\chyby\bug54.jpg

Popis:

      Přestože je Internet Explorer jednou z cílových platforem, vzhled
aplikace se po zobrazení v tomto prohlížeči liší od schváleného vzhledu,
ze kterého vychází uživatelská dokumentace.
      Špatné zobrazení se projevuje zejména posunutými tlačítky,
překrývajícími se prvky a rámečky, kterým občas chybí jedna nebo i více
bočních čar a špatné viditelnosti některých prvků, viz přiložený obrázek.
```

```
Špatné zobrazení bylo nalezeno v následujících částech GUI:
- v logu aplikace
- v záložce stahování
- v okně stahování položky a vytvoření nové záložky
- v záložce server

Další části se zobrazují správně.
```

Různé chyby snižují kvalitu softwaru různou mírou, proto při reportování chyb se uvádí závažnost, která vypovídá o tom, nakolik chyba brání používání aplikace a priorita, která určuje pořadí, ve kterém se chyby budou opravovat.

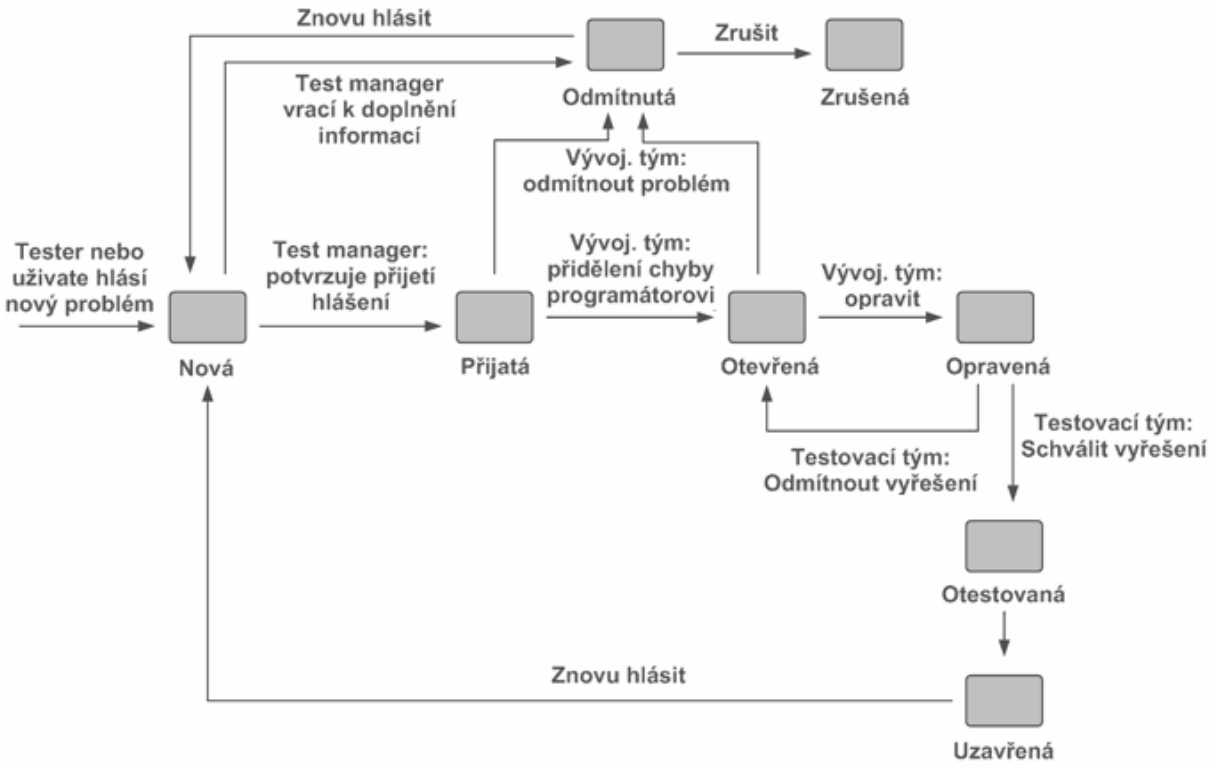
Běžné dělení chyb podle závažnosti (v angličtině severity):
A – V důsledku chyby není možné pokračovat v operaci
B – Došlo k závažné chybě, ale je možné chybu obejít a pokračovat
C – Důsledkem chyby jsou jen menší nepříjemnosti

Při prohlížení většího množství chyb se zobrazují hlášení ve formě seznamu, který obsahuje jen ty nejdůležitější údaje charakterizující chybu. Při prohlížení seznamu chyb vždy čtenáře nejvíce zajímá shrnutí, které v jedné větě popisuje podstatu chyby, dále pak závažnost a další údaje. Bug report systém neslouží jen programátorovi a testerovi, jen pár lidí bude číst hlášení chyby celé, většina si prolétne jen seznamy a vyhledá, co potřebuje nebo co ji zajímá. Proto je třeba věnovat shrnutí chyby stejnou pozornost, jakou věnují novináři titulku svého článku.

Po zapsání chyby do systému pro reportování, hlášení nezůstává neměnné, ale prochází vlastním životním cyklem. Kde se v tomto cyklu hlášení chyby nalézá je zaznamenáno v kolonce stav. Životní cykly chyb se liší nejen mezi projekty, ale i podle toho, zda chybu hlásí zákazník nebo tester, zda je projekt ve vývoji nebo se jedná o servis. Příklad možného životního cyklu hlášení je na obrázku 9.

Základním účelem hlášení je poskytnout vše potřebné k tomu, aby chyba byla správně a rychle opravena. Často jsou proto hlášení impulzem k diskuzi mezi členy vývojového týmu. Tato diskuze je často součástí hlášení a dokumentuje, co a proč se s chybou děje. Například pokud tester vrátí opravu programátorovi, tak mu vysvětlí, proč nepovažuje chybu za

opravenou. Pokud není možné diskuzi vést přímo jako součást chyby, je nejvhodnější přidat k chybě alespoň kolonku pro zdůvodnění poslední změny.



Obr. 9: Životní cyklus hlášení chyby

11.5 Metriky

Plánování a hodnocení procesu testování zahrnuje identifikaci měřítek a kritérií pro určení dosažené úrovně kvality. Při testování se pracuje s řadou neznámých, které je nemožné zjistit. Nejdůležitější hodnoty, kterými jsou celkový počet chyb v produktu a na kolik by přišlo neopravit nějakou chybu, je možné pouze hrubě a náročně odhadovat. Proto se hledání odpovědí na otázky, jak daleko je testování a jak je úspěšné, zaměřuje na kombinaci metrik, které jsou na projektu snadněji spočitatelné. Výběr měřítek záleží hlavně na tom, jaké odpovědi na otázky ohledně procesu testování jsou považovány za uspokojující.

Příklad, jak vypadají odpovědi na dotazy ohledně procesu testování na projektu bez použití metrik vyprávěný Hutchesnovou v [2]:

"The director asked the tester, "So you tested it? It's ready to go to production?
The tester responded, "Yes, I tested it. It's ready to go."
The director asked, "Well, what did you test?"
The tester responded, "I tested *it*."
In this conversation, I was the tester. It was 1987 and I had just completed my first test assignment on a commercial software system. I had spent six months working with and learning from some very good testers. They were very good at finding bugs, nailing them down, and getting development to fix them. But once you got beyond the bug statistics, the testers didn't seem to have much to go on except *it*. Happily, the director never asked what exactly *it* was."

Přeloženo:

„Nadřízený se ptá testera, "Takže jste to otestovali? Je to připravené na produkci?"
Tester odpoví, "Ano, otestovali jsme to. Je to připravené."
Nadřízený se ptá, "Co jste testovali?"
Tester odpoví, "Testoval jsem *to*."
V této konverzaci, jsem já byla testerem. Byl rok 1987 a já zrovna dokončila moje první přiřazení na testování komerčního softwarového systému. Strávila jsem šest měsíců prací a učením se od některých velmi dobrých testerů. Byli jsme velmi dobří v hledání chyb, následném testování a zajišťování oprav. Ale vyjma statistik o chybách, testeři neměli nic než *to*. Naštěstí, nadřízený se nikdy nezeptal, co přesně je *to*."

Bach a Bolton v [16] popisují, co se může v praxi skrývat za odpovědí, že to funguje.

"Jerry Weinberg has suggested that nailing "it works" may mean "We haven't tried very hard to make it fail, and we haven't been running it very long or under very diverse conditions, but so far we haven't seen any failures, though we haven't been looking too closely, either." In this pessimistic view, you have to be on guard for people who say *it* works without checking *even once* to see if *it* could work."

Přeloženo:

„Jerry Weinberg podotkl použití "funguje to" může znamenat, "Moc jsme se nesnažili to shodit a nespouštěli jsme testy moc dlouho nebo za různých podmínek, ale dosud jsme nenašli žádné selhání, ačkoli jsme se na to taky nedívali dostatečně podrobně." Z tohoto pesimistického pohledu, je třeba se mít na pozoru před lidmi, kteří říkají že *to* funguje, bez toho, aby *to* zkontrolovali *aspoň jednou*."

Z výše uvedených příkladů je vidět, jak vypadá testování bez použití metrik, a jsou v nich uvedené některé příznaky problémů, kterým je vhodné věnovat pozornost, protože povědomí o testování má pouze úzký okruh lidí v testovacím týmu. Mimo tento tým nemají nadřízení dostatečné informace o procesu testování, což může nejen vést k prodražování testování, ale hlavně k oddělení nositelů rozhodnutí od nositelů zodpovědnosti. Testovací tým nebývá zodpovědný za kvalitu ani rozhodnutí ohledně ní, většinou je poradním orgánem, který získává a předkládá informace potřebná pro rozhodnutí. Pokud není schopný poskytnout podklady a jen říká, jaké to rozhodnutí má být, manager zodpovědný za rozhodnutí většinou poslechne, tím pádem tým dělá rozhodnutí za něj.

Řešením není přenesení zodpovědnosti za tým, ten totiž nemá dostatečné pravomoci, navíc pokud není efektivní, tak sám nepřijme patřičná opatření a neprozradí nadřízeným, že by měli minimálně část týmu vyhodit a najmout někoho zkušenějšího.

V test reportu nebo v odpovědích na otázky nadřízených by se mělo vyskytovat několik základních nebo odvozených metrik, aby stav projektu byl co nejlépe odhadnut.

Metriky pro odhad kvality bývají založené na:

- chybách,
- testech
- nebo kódu.

Metriky založené na chybách

Samotný počet chyb není až tak důležitý a jako metrika moc význam nemá, ale v kombinaci s dělením chyb podle závažnosti, komponenty, ve které se chyba nalézá, stavu opravy, a zda chybu objevili testeři nebo zákazník vzniká sada užitečných měřítek, jejichž porovnávání a poměry už poskytují hodnotné informace.

Je-li řečeno, že na projektu je 130 otevřených chyb a zbývají dva měsíce z celkově ročního projektu do ukončení, tak bez dalších informací by jakákoli představa o kvalitě aplikace byla téměř určitě zavádějící. Více nám napoví srovnání s podobným projektem, předchozím test reportem ze stejného projektu nebo rozdělení podle závažnosti a komponenty.

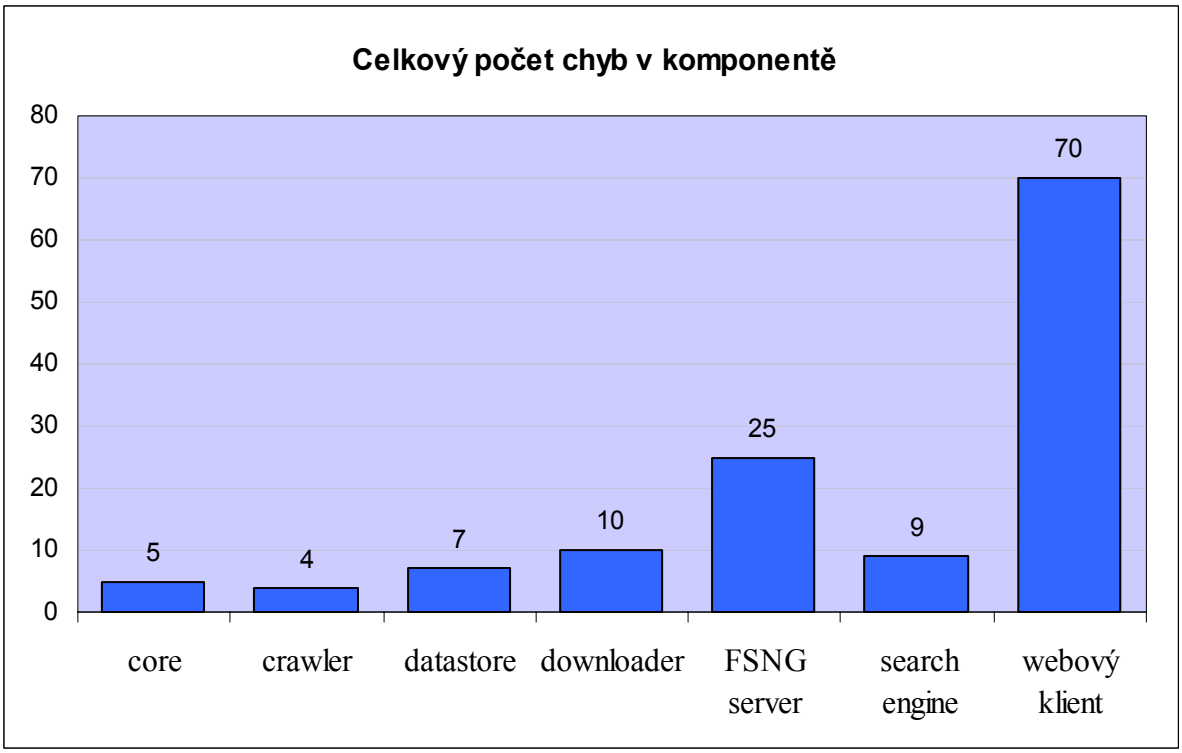
Z následujících statistik vidíme, že nejhůře na tom jsou komponenty webový klient, FSNG server, downloader a core. Chyby u webového klienta nejsou nijak závažné, ale pokud by ho viděl zákazník, pravděpodobně by byl kvalitou velmi znechucen. K tomu abychom odhadli nakolik velkým problémem jsou další tři komponenty, je zapotřebí vědět, jak jsou tyto komponenty velké a konkrétně co za chyby jsou u nich hlášeny.

Pokud by tedy manažer dostal jako odpověď na otázku, jak je na tom projekt s kvalitou, tabulku z níže uvedeného příkladu, snadno by si udělal představu o tom, jaké další otázky položit.

Příklad dělení chyb podle komponenty:

Komponenta	Závažnost A	Závažnost B	Závažnost C	Celkový počet chyb
core	4	1	0	5
crawler	0	4	0	4
datastore	1	6	0	7
downloader	4	5	1	10
FSNG server	5	17	3	25
search engine	3	4	2	9
webový klient	5	20	50	70

Tabulka 1: Dělení chyb podle komponenty a závažnosti



Obr. 10: Dělení chyb podle komponenty

Předchozí odpověď nadřízenému byla založena pouze na počtu chyb a jejich dělení. Kombinací jednoduchých cena, čas, počet chyb vznikají sofistikovanější metriky. Následující metriky uvádí Hatchesnová v [2]:

$$\text{Bug fix rate} = \frac{\text{počet opravených chyb}}{\text{počet nalezených chyb}} \text{ x } 100$$

- kolik procent chyb bylo opraveno. Namísto počtu opravených chyb se v praxi počítá s počtem uzavřených chyb, tedy s chybami, které byly úspěšně vyřešeny nebo byly zamítnuty.

$$\text{Efektivnost testu} = \frac{\text{chyby nalezené testem}}{\text{celkový počet nalezených chyb}} \text{ x } 100$$

- kolik procent z chyb v dané verzi bylo testem objeveno. Porovnává se efektivnost mezi testy nebo v čase a pokud test nepřináší dostatečné výsledky, vymění se za jiný.

$$\text{Test Effort Performance} = \frac{\text{opravené chyby nalezené během testu}}{\text{celkový počet nalezených chyb}} \text{ x } 100$$

- procentuelní přínos testu.

$$\text{Rychlost nalezení chyby} = \frac{\text{počet nalezených chyb}}{\text{počet hodin}}$$

- kolik chyb je průměrně nalezeno za hodinu. Počítá se stejně jako následující metriky v rámci jednotlivých verzí předaných k testování. Ze začátku projektu jsou obecně chyby nalézány rychleji než v závěrečné fázi.

$$\text{Cena nalezení chyby} = \frac{\text{náklady}}{\text{počet chyb}}$$

$$\text{Rychlost nalezení a zahlášení chyby} = \frac{\text{počet hlášení chyby}}{\text{počet hodin}}$$

$$\text{Cena hlášení chyby} = \frac{\text{náklady na hlášení chyby}}{\text{počet chyb}}$$

$$\text{Cena nalezení a hlášení chyby} = \text{cena nalezení chyby} + \text{cena hlášení chyby}$$

- sčítají se jen jednotky za stejnou verzi aplikace. Jakmile cena za nalezení a hlášení chyby stoupne nad cenu ponechání chyby v aplikaci, a nejedná se jen o přechodný stav, je lepší testování ukončit.

Metriky založené na testech

Systém projde testy za dvou různých situací, buď nejsou testy dostatečné nebo je připraven do provozu. Za předpokladu, že množina testů, kterými má aplikace projít, je známa, je možné sledovat, kolika procenty testů aplikace už dokáže projít. Protože testy stejně jako chyby se dají dělit do mnoha skupin, a ne u všech je použití této metriky vhodné, uvažují se v praxi pouze funkční testy aplikace. Jednotkou počtu testů bývá jeden test case nebo u automatizovaných testů jeho ekvivalent, případně test unita, pokud se jedná o testování jednotek.

Metriky založené na kódu

Metriky vzniklé na základě programovacího kódu tvořícího aplikaci už byly zmíněny u rozdělení testů, protože se odvíjejí od některého typu pokrytí kódu testy. Tyto metriky vypovídají o tom, kolik procent příkazů/hran/podmínek/cest bylo otestováno.

Na závěr kapitoly bych uvedla příklady možných odpovědí na otázku: „Kolik času zabere dokončení jednoho testovacího cyklu?“ Předpokládá se, že testovací případy pro verzi už jsou připravené. Tento příklad je upravenou verzí dotazu, který je původně součástí znalostního testu [40]. Smyslem je připomenout, že čím více informací je poskytnuto, tím kompletnější je odpověď. V tomto případě je nejlepší poskytnout přímo všechny odpovědi.

Pokud předpokládáte, že pouze provedeme testy, zabere nám to 5 dní.

Pokud předpokládáte, že při provádění testů nalezneme chyby a my jich najdeme a prošetříme a nahlásíme 20, zabere nám to 7 dní.

Pokud předpokládáte, že při provádění testů nalezneme 20 chyb, prošetříme je, nahlásíme, pomůžeme programátorům s jejich opravou, otestujeme opravu a budeme hledat 20 dalších chyb, které mohli vzniknout při opravách, bude nám to trvat 10 dnů.

12. Testovací cyklus

Proces testování v průběhu vývoje softwaru má podobu neustále se opakujícího cyklu, který začíná předložením nové verze testovacímu týmu. Cyklus má stejnou formu, ať má nová verze podobu dokumentace nebo kódu, liší se ale obsahem cyklu, kterým je příprava či provedení testů.

Po předložení je zkontrolována správnost verze, zda si požadavky nebo jiné části dokumentace navzájem neodporují, kontroluje se testabilita aplikace, schopnost být otestován. V případě kódu, se provádí tzv. smoke test nebo taky build verification test, jehož smyslem je potvrzení, že byl kód jednotlivých komponent správně sestaven dohromady. Pokud komponenty k sobě nepasují, v aplikaci se najednou objeví velké množství chyb. Testovací tým by mohl strávit dny jejich hledáním a reportováním, a pak opětovným přetestováním a zavíráním hlášení, v okamžiku, kdy by dostal správnou verzi. Pokud se tedy objeví velké množství chyb v už hotových a otestovaných částech aplikace, měla by být nová verze vrácena konfiguračnímu managerovi nebo jinému členu týmu, který sestavuje kód aplikace a ne podstoupena zbytečnému podrobnému testování.

V dalším kroku probíhá samotná příprava či provedení testů podle připraveného plánu a hlásí nalezené chyby do bug report systému. V posledním kroku se sepisují výsledky testovacího cyklu, hodnotí testovací přístup, případně navrhují zlepšení pro příští cyklus.

Podle použité metodiky a náročnosti vývoje může být testovací cyklus zasazen do mnohem formálnějšího procesu. V rámci cyklu pak navíc probíhají následující úkoly:

- Dohodnutí cíle testů – Úkoly testovacího týmu v daném cyklu mohou být různé, od nalezení co největšího množství chyb přes odhad celkové kvality až po nalezení scénářů, které jsou vždy úspěšné a mohou být předvedeny zákazníkovi.
- Určení předmětu testů a hodnotících kritérií – Na kterou oblast aplikace bude testování zaměřené, jaké budou hodnotící kritéria bývá většinou předmětem testovacího plánu, který se ale vztahuje na celý proces testování, v rámci cyklu probíhá už jen ujasnění na základě cílů stanovených pro daný cyklus.
- Příprava testovacího prostředí – Před každým spuštěním testů je nezbytné připravené stabilní prostředí, které není ovlivněno předchozími testy. V rámci nového cyklu toto prostředí může být změněno, aby vyhovovalo nové verzi předané k otestování.
- Analýza výsledků a reportování – – Formální verze cyklu končí sepsáním dříve zmíněného formálního test report dokumentu.

Část III. – Testy webových aplikací

13. Testy a techniky

Při testování webových aplikací jsou důležité stejné dimenze kvality jako u ostatních typů aplikací: Funkčnost, použitelnost, spolehlivost, výkon, podpora a ostatní. V případě webových aplikací je pak vhodné oddělit jako samostatnou dimenzi ještě bezpečnost.

V této kapitole bude vysvětlen výběr technik, které jsou vhodné pro zvážení při testování webových aplikací, a rozděleny podle dimenzí kvality, z nich žádná by neměla být opomenuta.

Techniky testovaní vznikaly na více místech současně, proto je běžné, že mívají i několik jmen nebo jich je několik známých pod stejným jménem. Rovněž některé techniky pokrývají více dimenzí nebo je možné se rozhodnout na jakou dimenzi budou aplikovány, proto by u nich měl být vysvětlen i zamýšlený způsob jejich použití.

Ještě před výběrem testů pro konkrétní webovou aplikaci, se provádí příprava, při které se sbírají a třídí informace související s účelem funkce webové aplikace. Pokud dosud není, tak se definuje cílová skupina uživatelů, na kterou se testeři mají soustředit.

Zkoumají se už existující aplikace podobného zaměření a styl práce jejich uživatelů. Rovněž se vybírají nástroje pro podporu testování. Vymýšlejí se konfigurační prostředí. Vytváří se testovací plán.

13.1 Funkční testy:

Testování podle scénářů – Na základě specifikací nebo způsobu, jakým s aplikací uživatel pracuje, se vytvoří testovací případy a scénáře, podle kterých je porovnáváno skutečné chování aplikace oproti očekávanému. Testování podle scénářů se zaměřuje na hlavní cíle a požadavky. Jeho výhoda spočívá v tom, že představuje řízené manuální testování a můžou ho provádět brigádníci bez zkušeností v testování. Na druhou stranu pokud toto testování provádí zkušení testeři, naleznou mnoho způsobů, jak scénář provést a experimentují se scénáři při zachování systematického přístupu.

Příklad různého provádění scénářů:

```
Scénář:
…
Krok 5: Vypněte a znovu zapněte aplikaci.
Očekávaný výsledek: Dříve zadané změny jsou po opětovném zapnutí aplikace
úspěšně provedeny.
…

Nezkušený tester: Vypne aplikaci křížkem a znovu ji spustí pomocí
ikony na ploše.
Zkušený tester:
1) Provede scénář stejně jako nezkušený tester.
2) Znovu provede scénář, ale v kroku 5, vypne aplikaci pomocí task
managera.
3) V kroku 5, zruší procesy aplikace.
4) Připraví si předem dvě instance aplikace a vypne a zapne pouze
jednu, druhá stále běží.
```

Testování založené na scénářích je velmi používaná technika, která má velké množství podtypů např.:

Basis path testing – Výběr testů je odvozen od různých průchodů aplikací, snaží se docílit pokrytí všech cest v aplikaci.

Happy day scénáře – Pokrytí pouze těch cest a použití pouze takových vstupních dat, která jsou od uživatele nejčastěji očekávána.

Specification testing – Scénáře jsou zaměřené výhradně na kontrolu souladu mezi specifikací a aplikací.

Orthogonal array testing – Jedná se o techniku vhodnou k minimalizaci množství testů, zejména je její použití vhodné v kombinaci s testováním podle scénářů nebo s konfiguračním testováním. Jejím základem je, že pokud je zapotřebí vyzkoušet různé kombinace prvků, je vhodné namísto všech možných kombinací, vyzkoušet jen všechny kombinace mezi dvěmi následujícími prvky. Při třech prvcích o třech možných hodnotách tak vznikne devět testovacích případů, viz. tabulka 2 [36].

Testovací případ	prvek A	prvek B	prvek C
1	1	1	1
2	1	2	2
3	1	3	3
4	2	1	2
5	2	2	3
6	2	3	1
7	3	1	3
8	3	2	1
9	3	3	2

Tabulka 2: Ze tří prvků o třech hodnotách vznikne devět testovacích případů

Testování hraničních hodnot – Testují se všechny vstupy v aplikaci na správné zacházení s hraničními hodnotami. Hraniční hodnoty mají nejen číselné vstupy, u kterých je hraniční hodnotou nejvyšší nebo nejnižší přijímaná a nepřijímaná hodnota, ale i vstupy přijímající textové hodnoty nebo hodnoty typu ano/ne.

Příklad:

Je úkolem otestovat pole přijímající název barvy, vždy se očekává jen jedna hodnota. Za hraniční hodnotu může být považována hodnota „červenámodrá“, protože testuje správné ošetření této hodnoty, která přesahuje hranici jedné barvy.

Regresní testy – Skládají se z velkého množství jednoduchých testů, které jsou prováděny po opravách chyb nebo funkčních vylepšení a které hledají nově zanesené chyby do už funkčních částí. Síla těchto testů je v jejich velkém množství a v automatizaci. Přestože definice těchto testů přímo nevylučuje, aby byly manuální, jejich smysl a charakter se při testování přímo testerem změní natolik, že už se jedná o zcela jiný typ testování, většinou testování podle scénářů nebo testování opravy chyby (fix testing).

Jednotkové testování (unit testing) – Testování co nejmenších částí aplikace, jakmile je k dispozici programový kód. Tento druh testování se moc nehodí na testování webových stránek, ale pro internetové aplikace a softwarové systémy s tenkým klientem je stejně přínosný jako u klasických aplikací.

Pokrytí hran nebo podmínek – Technika kontrolující, jaké množství kódu bylo při testech provedeno.

Akceptační testování – Kontroluje, zda a nakolik aplikace splnila předem domluvená kritéria mezi zákazníkem a dodavatelem.

Průzkumné testování (Exploratory testing) – Tester si libovolně zkouší aplikaci ve snaze nalézt co nejvíce chyb, přičemž se zároveň učí, co aplikace umí a jak funguje. Tento druh testování závisí plně na kreativitě testera. Přestože se jedná o testování primárně zaměřené na funkčnost a použitelnost aplikace.

13.2 Testy použitelnosti

Accessibility testing – Testuje možnosti a pohodlí používání aplikace pro lidi s hendikepem.

Error handling – Řada aplikací má problém se správným zvládnutím chyb, chybové hlášky jsou nesprávné, neposkytující žádné informace nebo naopak informace ohrožující bezpečnost aplikace, chyby vyvolávají nekorektní stavy. Toto testování spočívá v záměrném vyvolání nekorektních stavů, chybových hlášení nebo varování. Error handling testy bývají zaměřené na použitelnost nebo spolehlivost aplikace.

User testing – Pravděpodobně nejdůležitější z testů použitelnosti. Tester vybere vzorek z cílové skupiny uživatelů a ty sleduje při používání aplikace. V průběhu a po skončení testů může uživatelům klást otázky, ale jsou to uživatelé, kdo mu vysvětlují, jak aplikace působí a jestli jí rozumí, tester by v nich neměl vyvolat dojem, že jsou zkoušeni. I když tester sám spadá do cílové skupiny uživatelů, nikdy není vhodným představitelem této skupiny.

Testování dokumentace – Kontrola srozumitelnosti, správnosti a užitečnosti uživatelské dokumentace.

13.3 Testy spolehlivosti

Strukturní testování – Kontroluje zachování a dodržení designu a formy, u webových aplikací se jedná například o ověření, že všechny odkazy jsou správné a neexistuje obsah, ke kterému se nelze dostat.

Testování stability – Testování spolehlivosti za běžných podmínek používání aplikace. Zaměřuje se na schopnost aplikace pracovat dlouho a za využití jejího plného rozsahu bez vyvolání chyby.

Stress testing – Sleduje, jak se systém chová za neobvyklých okolností jako je extrémní zátěž, nedostatečná paměť, nedostupnost služby a další.

Memory leak testing – Hledá příznaky špatného zacházení s pamětí na serveru.

Monkey testing – Automatizované testování založené na náhodném klikání do aplikace. Na základě inteligence, jaká je stroji přidělena může být klikání absolutně náhodné, nebo stroj ví, do jakých dalších stavů může aplikace z aktuálního stavu přejít a jak. Testuje se spolehlivost a funkčnost aplikace.

13.4 Výkonnostní testy

Výkonnostní testování (Performance testing) – Rychlost odpovědí aplikace je měřena pod různou zátěží, testování probíhá na různých úrovních, časy se analyzují a hledají se místa, která omezují výkon aplikace. Výkon se posuzuje z hlediska reakčních časů, využití zdrojů (hardwaru) a počtu zpracovaných požadavků za určitou dobu. Slouží k určení rychlostních, škálovatelnostních a spolehlivostních vlastností. Rovněž se považuje za nadmnožinu ostatních výkonnostních technik.

Zátěžové testování (Load testing) – Aplikace se testuje pod velkou ale očekávatelnou zátěží, je zjišťováno, kolik toho vydrží, jestli selže nebo jen zpomalí. Kontroluje splnění požadavků ohledně výkonu aplikace.

Stress testing – Má zde stejné zaměření jako u testů spolehlivosti, protože se jedná defakto o podmnožinu spolehlivostního stress testu. Tato čistě výkonnostní varianta testu je ale nejznámější.

Kapacitní test – Slouží k určení kolik uživatelů popřípadě transakcí daný systém zvládne a zároveň stále dodrží výkonnostní limity [30].

Meier a kol. v [30] píše:

„In casual conversation, most people associate 'fast enough' with performance testing, 'accommodate the current/expected user base' with load testing, 'something going wrong' with stress testing, and 'planning for future growth' with capacity testing. Collectively, these risks form the basis for the four key types of performance tests for Web applications."

V překladu:

„V běžném hovoru si většina lidí spojuje 'dostatečně rychle' s výkonnostním testováním, 'přizpůsobení současnému/očekávanému počtu uživatelů' se zátěžovým testováním, 'něco se pokazí' se stress testováním a 'plánování pro budoucí růst' s kapacitním testováním. Společně tyto rizika tvoří podstatu čtyř klíčových typů výkonnostních testů pro webové aplikace."

Benchmark test – Výkonnostní test jehož úkolem je porovnání výkonu nové aplikace s už existující podobou aplikací.

Contention test – Je testována schopnost zvládnout více požadavků na stejný zdroj.

13.5 Testy podpory

Testování kompatibility – Kontroluje schopnost aplikace vyměňovat si informace s jinými systémy.

Konfigurační testování – Zjišťuje, jak aplikace pracuje na různém hardwaru a při různé konfiguraci.

Bandwidth testing – Testování webové aplikace s různými rychlostmi připojení k internetu.

13.6 Bezpečnostní testy

Audit kódu – Zdrojový kód je kontrolován osobou nebo nástrojem, zda odpovídá designu a standardům nebo za účelem vyhledávání chyb. Nemusí se jednat o testování bezpečnosti, je účinný i při funkčním nebo výkonnostním testování a při kontrole čitelnosti zdrojového kódu.

Penetration testing – Penetrační tester nebo tým se zvenku snaží útočit na aplikaci ve snaze odhalit slabé stránky bezpečnosti a jejich dopad.

13.7 Další

Beta testování – Zkušební zprovoznění aplikace za účelem testování. Uživatelé si aplikaci zkoušejí a hlásí chyby.

Lokalizační testování – Testování zaměřené na zvládání cizího textu a dat, kontrola překladů.

End-to-End testing – Spojuje více druhů testů dohromady, jedná se o systémový test, kdy se testuje celá aplikace v co nejreálnějším prostředí. Zahrnuje testování komunikace s databází, přes síť, s dalším hardwarem a softwarem, plné využití funkcí, pod reálnou zátěží.

Testování navigace a odkazů – Zda neexistují stránky, na které nevede žádný odkaz, stránky se zastaralými a nefunkčními odkazy a další chyby v navigaci, které bývají časté u rozsáhlých a často se měnících webových aplikací.

14. Používání nástrojů

Pro podporu testování existuje řada nástrojů, jejichž použití by mělo zpřehlednit a urychlit proces. Přehled nástrojů a praktické znalosti jejich ovládání jsou pro testera důležité, ale často bývají nástroje bohužel přeceňovány natolik, že jsou používány i v případě, kdy práce s nimi je spíše zdržením.

Automatizace testování má dost nevýhod, mezi které patří velké náklady na nasazení nástroje a vytvoření skriptů, omezení způsobená možnostmi nástroje, tendence okolí příliš spoléhat na množství testů a ne na jejich různorodost a kvalitu.

Michal Bolton podle [39] k tomu řekl:

"People write code to find bugs on another code. They write code that is buggy which claims to find bugs on another code."

"Writing test scripts is another development project. Most of them don't realize that they are running two development projects in parallel and hence their main development project suffers."

Přeloženo:

„Lidé píší kód, aby nalezli chyby v jiném kódu. Píší kód plný chyb a tvrdí, že nalézá chyby v jiném kódu."

„Psaní testovacích skriptů je další vývojový projekt. Většina z těch lidí si neuvědomuje že vedou dva paralelní vývojové projekty, a proto jejich hlavní vývojový projekt trpí."

V praxi tento problém může nastat hlavně při používání open source nástrojů, které jsou k dispozici volně ke stažení na internetu, ale které jsou často dosti osekané a proto si je testovací týmy běžně upravují. V menší míře ale stále přítomné jsou tyto problémy i při nasazení komerčního nástroje, kde je potřeba hledat způsob, jak se vypořádat s některými chybami a omezeními danými použitou technologií.

Pradeep Soundararajan v [39] vysvětluje, že důvodem pro používání různých technik a zkoušení různých věcí při testování je to, že každá z těchto technik nám přináší něco nového, že se vzájemně doplňují, ne proto, že by některé byly lepší něž jiné. Odmítá přitom porovnávání automatizovaného a manuálního testování a kritizuje vyzdvihování automatických testů.

Spoléhání pouze na jeden druh testování je krátkozraké, dosažení větší kvality úzce závisí na udržení diverzifikovaného a rozumného přístupu k testování. Automatizované testy by neměly stát samy o sobě, bez doplnění jiným druhem testů, který představuje hledání nových druhů chyb a nových scénářů testů, nebude testování přinášet dostatečné výsledky a křivka nově nalezených chyb strmě klesne těsně nad nulu.

Smyslem a přínosem automatizovaných testů je zadání monotónní práce nebo v reálném světě těžko nasimulovatelných případů stroji, zatímco člověk se může více soustředit na kreativní nebo smyslovou část testování. Zbaví se tak nepříznivých vlivů, jaké monotónní provádění scénářů může u někoho mít na chuť k experimentování a přemýšlení.

Do automatizovaného testování se obecně vyplatí investovat, pokud se očekává, že napsané skripty budou dlouhodobě využívány. Zároveň, protože nástroje nebývají příliš odlišné, tak pokud si tester měl možnost vyzkoušet nástroj pracující na určitém principu, přechod na jiný nástroj stejného typu by mu neměl činit žádný problém. Zaškolení testerů je tedy také dlouhodobou investicí.

Na internetu je k dispozici pro usnadnění testování řada open source nástrojů. Komerční nástroje například od IBM, Compuware nebo Mercury, což jsou firmy, jejichž produkty pro testování jsou nejpoužívanější, bývají k dispozici pro vyzkoušení jako časově omezené trial verze.

Nástroje pro testování se používají na:

- získání dat pro testování
- statistické testování – analyzování informací z designu nebo kódu
- dynamické testování
    - nástroje pro automatizované funkční testování
    - nástroje pro výkonnostní testování
    - nástroje pro jednotkové testování
    - simulátory/ drivery – nahrazují část systému, která s různých důvodů není dostupná pro účely testování
- test management – nástroje pro plánování a řízení testů
- řízení reportů (bug report systémy)

Jak správně upozorňuje Pavlíčková a kol. v [31], programátoři se mohou dopustit mnoha různých chyb, které se netýkají jen funkčnosti, ale mohou vést třeba k problémům při implementaci změn nebo omezit znovupoužitelnost kódu.

I takové chyby lze vyhledávat za pomocí nástrojů. Nevadí, že testovaná aplikace při nich není spuštěná, protože pracují na principu vyhledávání určitých konstrukcí při procházení kódu.

Za zmínku stojí ale i nástroje, které přímo pro testery určené nejsou, velmi užitečné jsou i klasické textové a tabulkové editory, task managery, nástroje na nahrávání obrazovky, případně webové kamery a řada dalších utilit.

V následujících kapitolách je vysvětlen princip automatizovaného funkčního a výkonnostního testování a uvedeno, jak takové nástroje mohou vypadat a jak se s nimi pracuje na příkladech vybraných open source i komerčních nástrojů. Hodnocení nástrojů nemá odpovědět na otázku, který je nejvhodnější pro určitý projekt, i když může být zmíněn můj názor v této oblasti. Smyslem je přiblížení různých přístupů k automatizaci testů a jak tyto nástroje spolu s přístupy mohou vyhovovat testerům začínajícím v dané oblasti testování.

15. Automatizace funkčních testů

Nástroje na automatizaci funkčního testování se používají u větších projektů pro regresní testy a testování souladu aplikace se specifikací. Skripty jsou tedy zaměřené na základní funkčnost a už nalezené chyby, případně je možné je použít pro vyhledávání chyb, pro které je tento přístup mnohem vhodnější než ruční testování, například pro kontrolu platnosti všech hypertextových odkazů na webovém sídle.

Skripty jsou tvořeny na základě přepisu testovacího scénáře do jazyka srozumitelného pro vybraný nástroj. Některé nástroje umožňují i nahrání scénáře, prováděného testerem. Záznam pak stačí poupravit odstraněním a pozměněním nevhodných příkazů a dodáním flexibilněji vkládaných dat, například načtených ze souboru nebo náhodně vygenerovaných. Protože testeři nemusí mít znalosti programovacích jazyků, autoři nástrojů pro testování často hledají způsob, jak testovací skripty přiblížit běžné angličtině. Existují dva přístupy, které k tomu používají. Buď je pro nástroj vytvořen hybridní jazyk mezi angličtinou a programovacím jazykem, případně se hledají čitelnější jazyky jako je například značkovací jazyk XML. Nebo druhou možností je vytvořit přepis programovacího jazyka do srozumitelnější formy, například sady anglických vět, přičemž je možné psát skripty jak v tomto programovacím jazyce, tak poskládat z předpřipravených vět.

Nástroje pro testování webového klienta poznávají, s jakým prvkem mají pracovat, pomocí html kódu testované stránky. V rámci něj je možné prvek identifikovat jeho hodnotou, např. text tlačítka nebo odkazu, pomocí cesty v jazyce XPath[9], nebo jednoznačným identifikátorem přiřazeným prvku. Protože tyto identifikátory nebývají běžně součástí stránek a ostatní možnosti určení prvku jsou často nejednoznačné a nedostačující, je vhodné na testování myslet už při programování. Nejvhodnější je zavést pravidla pro tvorbu html kódu a mít možnost v konfiguraci webové aplikace jednoznačné identifikátory zapnout nebo vypnout podle toho, zda se hodlá aplikace testovat automatizovaně nebo ne.

Po spuštění skriptu nástroj simuluje činnost testera v podporovaném prohlížeči, http odpovědi a získaný html kód kontroluje oproti očekávanému chování představovanému konkrétními kontrolami zapsanými do skriptu. Výhodné je, pokud nástroj obsahuje i možnost uložení získaných html stránek pro pozdější prohlédnutí a bližší určení zjištěných chyb.

[9] Dotazovací jazyk určený k identifikaci prvků dokumentu napsaném v jazyce XML.

15.1 Canoo WebTest

Canoo WebTest je open source nástroj určený k testování webových aplikací. Je napsaný v programovacím jazyce Java a k psaní testovacích skriptů se používá XML jazyk. Skripty WebTestu jsou v podstatě Ant skripty, což umožňuje snadnou integraci s tímto nástrojem. Informace týkající se WebTestu jsou kombinací zkušeností nabraných při zkoušení nástroje a z dokumentace k nástroji [17].

Instalace

Na stránkách http://webtest.canoo.com/webtest/manual/Downloads.html je možné si vybrat z vývojových verzí WebTestu. V této diplomové práci je popsaná stabilní verze WebTest 2.6. U každé verze jsou k dispozici ke stažení kromě buildů i zdrojové soubory, dokumentace a sada testů, kterými je možné si otestovat stažený nástroj.

K běhu aplikace se vyžaduje nainstalované vývojové prostředí Javy JDK verze 5 nebo vyšší, případně pro některé způsoby použití pak Ant verze nejméně 1.7.0. Místo instalace je po stažení potřeba rozbalit soubory do vybraného adresáře a nastavit cestu ke spustitelnému souboru webtest, který je v podadresáři bin do proměnných prostředí.

Struktura

Vytváření projektové struktury pro testy a spouštění testů probíhá z příkazového řádku. Vytvoření projektové struktury s už funkčním příkladem testu:

```
C:\> webtest -f path\to\webtest\webtest.xml wt.createProject
- webtest.xml by měl být přímo v adresáři, do kterého jste nástroj rozbalili
```

Spuštění testu se provádí příkazem webtest v adresáři právě vytvořeného projektu. V tomto adresáři se vyhledá soubor build.xml, podle jehož obsahu se test řídí. Uvnitř vytvořeného projektu je následující struktura adresářů a souborů pro testy:

Adresáře:

- definitions – Adresář jehož soubory jsou definice maker, které je pak možno využít v testovacích skriptech. Pomocí maker si uživatel může vytvořit vlastní xml značku i s definicí jejich atributů, která umožňuje zjednodušit často opakované podobné úseky.
- Dtd – Obsahuje soubor Project.dtd, který je automaticky generuje definice pro soubory adresáře includes.
- Includes – Obsahuje popisy entit, které stejně jako makra slouží k nahrazení opakovaných posloupností akcí, ale jsou jednodušší, protože nemají atributy.
- Results – Vytvoří se až po prvním spuštění, obsahuje záznam výsledků testů.
- Tests – Adresář pro vytvářené testovací skripty

Soubory:

- build.xml – Hlavní soubor, nastavuje proměnné celého testu, importuje soubory, specifikuje, kde najít soubor s testy, které mají být spuštěny.
- definitions.xml – Automaticky generovaný soubor z adresáře definitions.

Skripty je možné psát v jakémkoli textovém editoru a ukládají se s příponou .xml, výsledný report o průběhu testů má formát html.

Použití

Uživatelská dokumentace k WebTestu je k dispozici jak na webových stránkách nástroje tak ke stažení u jednotlivých verzí v souboru doc.zip. Tento soubor obsahuje navíc i dokumentaci ke zdrojovým souborům nástroje, tedy popis tříd a jejich metod, a další adresářové struktury, které činí orientaci v obsahu této struktury poněkud nepřehlednou. Uživatelskou dokumentaci v podobě, jak je k dispozici na internetu, najdeme pod cestou doc\info\manual.

Dokumentace WebTestu obsahuje zejména popis značek xml, které k tvorbě testů můžeme využít, ale už neobsahuje návod, jak si poradit s některými častými problémy, například jak se vypořádat s českými znaky na stránkách. Už při vyzkoušení jednoduchého vzorového příkladu testu z uživatelské dokumentace, kdy pouze dojde k vyvolání požadované

stránky a z kontrolování jejího názvu (obsahu html značky <title>), se může uživatel setkat s následujícím hlášením, kvůli kterého test nebude spuštěn:

```
    Error reading project file G:\WebTest\Poeta\tests\verifyTitle.xml:
Invalid byte 2 of 3-byte UTF-8 sequence.
```

Problémem je, že dokument s testem napsaný v xml obsahuje neočekávané znaky. Problém lze vyřešit jednoduše přidáním níže uvedeného řádku bezprostředně do souboru s českými znaky. V případě, že se čeština objevuje v definované entitě, je potřeba tento řádek uvést přímo v popisu entity, tedy v adresáři includes nebo definitions, nestačí je uvést na místě volání entity.

Pro akceptaci českých znaků je potřeba přidat na začátek dokumentu s problémovou sekvencí řádek:

```
<?xml version="1.0" encoding="windows-1250"?>
```

Příklad jednoduchého testu v nástroji WebTest:

```
Includes/goToPoeta.xml:
  <?xml version="1.0" encoding="windows-1250"?>
  <invoke url="http://www.poeta.cz"/>
  <verifyTitle text="Poeta.cz - moderní literární server"/>

tests/verifyTitle.xml
  <?xml version="1.0"?>
  <!DOCTYPE project SYSTEM "../dtd/Project.dtd">
  <project default="test">
    <target name="test">
      <webtest name="verify title">
  &goToPoeta;
</webtest>
    </target>
  </project>
```

Aby WebTest umožňoval verifikaci a navigaci po internetových stránkách musí obsahovat funkčnost na vyhledání internetové stránky, zkontrolování přítomnosti a textu jejích prvků, vyplnění formulářových polí a jejich potvrzení a pohyb mezi stránkami za pomocí klikání na odkazy. Tuto základní funkčnost zajišťuje sada definovaných xml značek,

mezi které patří invoke, clickButton, clickLink, verifyElementText, setInputField, setCheckbox.

Pro většinu testů by ale taková funkčnost nebyla dostatečná, například kdybychom chtěli ověřit, že počet děl, které se u jména autora objevují, souhlasí s počtem skutečně zobrazených děl po kliknutí na odkaz. Zde je třeba extrahovat správný text, z něj extrahovat a uložit číslo vyjadřující počet autorů, na další straně pak spočítat díla autora a to i přes několik stránek. Celý proces by bylo vhodné opakovat například pro 20 náhodně vybraných autorů.

Takové náročné testy jsou velmi časté a pokud nejsou umožněny, ztrácí automatizované testování na svých výhodách. O možnosti využití nástroje v praxi proto rozhoduje vyjadřovací síla jazyka určeného pro psaní testů, tedy, co vše je možné vyjádřit.

WebTest obsahuje několik užitečných konstrukcí, které sílu jeho jazyka významně zvyšují:

a) Regulární výrazy a dotazy v jazyce XPath, WebTest integruje do velké části svých značek, kde hrají roli atributů. Např. u verifikace, může být hledaný text specifikován regulárním výrazem, nebo odkaz či jiný prvek pomocí XPath dotazu. Přímo pro práci s regulárními výrazy a XPath dotazy jsou určené značky:

- storeRegEx – uloží regulární výraz do takzvané vlastnosti, která ukládá konstantní a proměnné hodnoty.
- storeXPath – uloží výsledek XPath dotazu do vlastnosti.
- verifyXPath – zkontroluje, že cesta určená dotazem existuje nebo že výsledek má určitou hodnotu.

b) Řízení za pomocí dat představuje jediná značka:

- dataDriven - umožňuje opakování akcí specifikovaných mezi počáteční a koncovou značkou pro všechny řádky tabulky. Cesta k této tabulce je povinným atributem.

```
<dataDriven tableContainer="./autori.xls">
  <echo message="Autor: ${FirstName}, pocet del: ${NumberOfPieces}"/>
</dataDriven>
```

Bohužel i tady se objevují problémy s kódováním. Přestože tabulka neobsahuje žádné české znaky, WebTest hlásí:

```
Error reading project file G:\WebTest\Poeta\tests\data.xml: Invalid
byte 1 of 1-byte UTF-8 sequence.
```

c) Nahrazení důležitých konstrukcí, které obsahují programovací jazyky:

- storeRandom – uloží náhodnou hodnotu ze zadané množiny do vlastnosti.
- not – u všech akcí uzavřených mezi počáteční a koncovou značkou not se očekává, že selžou. Pokud jedna z akcí bude úspěšná, znamená to, že takový test selže.

- repeat – akce uzavřené mezi počáteční a koncovou značkou se opakují tolikrát, kolikrát je zadáno v atributech.

- ifStep - zajišťuje podmíněné spuštění akcí.

d) Integrace skriptovacích jazyků může být považována za určitou pojistku, pokud ostatní konstrukce nebudou stačit:

- scriptStep – mezi počáteční a koncovou značku může být vepsán kód vybraného skriptovacího jazyka. Tento jazyk musí být podporován Bean Scripting Frameworkem (BSF), tedy to mohou být JavaScript, Python (Jython nebo JPython), Tcl (Jacl), NetRexx, XSLT Stylesheets, Java (BeanShell), JRuby, Groovy, ObjectScript a JudoScript, jak uvádí uživatelská dokumentace WebTestu dostupná z [17].

e) Dále WebTest poskytuje sadu značek pro specifické úkoly, jako je práce s emaily, pdf a excelovskými soubory.

Přehled výsledků testu

Výsledky testů jsou ve WebTestu zobrazovány ve formě stránek html, u každého testu je možné se podívat na detaily jeho průběhu, na zachycené stránky a vyhozené chyby.

Struktura výsledků je přehledná a obsahuje dostatek informací. Výsledky zůstávají do dalšího spuštění testů v adresáři results a poté jsou smazány. V případě, že si uživatel chce některé výsledky uschovat, musí si je překopírovat jinde.

webtest

WebTest Test Report Tests started at Sun Jul 13 19:57:59 CEST 2008

Result Summary

WebTests	#	%	Graph
✓	2	67	
✗	1	33	
Sum	3	100	

Steps	#	%	Graph
✓	5	83	
✗	1	17	
○	0	0	
Sum	6	100	

Server Roundtrip Timing Profile

Secs	#	%	Histogram
0 - 1	0	0	
1 - 3	8	62	
3 - 5	3	23	
5 - 10	2	15	
10 - 30	0	0	
> 30	0	0	
Sum	13	100	
Avg		6003 ms	

Test Scenario Overview (00:01:06)

#	Result	Name	# Steps	Timing profile			Failing step
				Duration	%	Graph	
1	✓	verify title	2 / 2	00:00:07	10		
2	✗	check validity of links	2 / 3	00:00:39	58		**verifyLinks** Check Links
3	✓	check number of pieces	19 / 21	00:00:21	32		

Obr. 11: Výsledek testu

× check validity of links

Test started at Sun Jul 13 19:58:06 CEST 2008, lasting 00:00:39 (38531 ms).
Source: G:\WebTest\Projekty\Poeta\tests\validityOfLinks.xml:9:
Base URL (used by **invoke** steps with a relative URL): http://localhost/

#	Result	Name	Parameter	
1	✓	**invoke** Resulting page	**url**	http://www.poeta.cz
2	✓	**verifyTitle**	**text**	Poeta.cz - moderní literární server
3	× Error Page	**verifyLinks** Check Links Resulting page	**-> valid links**	7
			depth	1
			excludes	.*poeta.cz/mc.*
			ignoreForeignJSErrors	true
			onsiteonly	true

Error

Message

JavaScript error loading page http://www.poeta.cz/forum/: ReferenceError: "pagename" is not defined. (http://www.poeta.cz/stat/phpmyvisites.js#116)

Location

G:\WebTest\Projekty\Poeta\tests\validityOfLinks.xml (line: 12)

Back to Test Report Overview

Obr. 12: Výsledek testu

Zhodnocení

WebTest zvolil xml pro psaní testů ve snaze zjednodušit práci testerům, kteří neovládají žádný programovací jazyk. Pro jednoduché procházení stránek, je i jejich řešení čitelné a jednoduché, zvláště, pokud tester zná html kód nebo se v něm alespoň orientuje při čtení.

V rámci zvýšení vyjadřovací síly jazyka pro psaní testů, se tato výhoda dle mého názoru odsunula do pozadí. Regulární výrazy, jazyk XPath, řada značek, jejichž význam je třeba si pamatovat a naučit se práci s nimi a také skriptovací jazyky, to vše se tester musí při používání WebTestu postupně naučit. Což může být složitější, než naučení několika příkazů poskytnutých frameworkem, například nad Javou a postupné rozšiřování znalostí o další Javovské příkazy. Navíc bude pro takového testera jednodušší přejít na nástroje pro výkonnostní nebo jednotkové testování, které jsou také postavené na principu frameworku nad Javou. Pokud by však testeři měli zkušenosti s psaním stránek, ale s žádným programovacím jazykem se dosud nesetkali, může být varianta XML-XPath-Skriptovací jazyk stále výhodnější.

Nástroji WebTest chybí podrobnější uživatelská dokumentace, která by nepopisovala pouze k čemu jsou jednotlivé značky a jaké mají atributy, ale podrobněji vysvětlovala používání nástroje a věnovala se řešení problémů. Částečně může tuto nevýhodu odstranit historie mailing listu, kde je možné procházet problémy a rady ostatních lidí požívajících WebTest. Tento mailing list je ale rozsáhlý a nepřehledný. Na druhou stranu ve srovnání s jinými open source nástroji je tato dokumentace jedna z těch lepších, obsahuje jak úvod do práce s nástrojem, tak katalog prvků, což je rozhodně dostačující.

Problémy s kódováním a další nedostatky, které nástroj může mít, nejsou příliš velkým problémem, pokud jich není neúnosně mnoho. O žádném větším produktu si tester nedovolí říct, že neobsahuje chyby, proto by s nimi měl počítat i u nástrojů. Výhodou open source nástrojů je to, že opravu nalezených chyb je možno ovlivnit.

WebTest je přes své chyby a přes řadu věcí, které je třeba k jeho používání se naučit, dobrý nástroj pro automatizaci funkčních testů a obstojná náhrada komerčních nástrojů.

15.2 JWebUnit

Výhodou frameworku postaveném nad Javou je to, že je třeba doprogramovat pouze základní metody jako je otevření prohlížeče, vyvolání stránky, procházení stránek za pomocí klikání na odkazy, tlačítka a v menu, vyplnění formulářů, získání html kódu a uložení stránky jako obrázku pro pozdější použití. Potřebuje-li tester něco navíc, použije programovací jazyk. Takovýto framework je pro pokročilého programátora poměrně jednoduché vytvořit. Nástroj, se kterým jsem pracovala, KWUnit, vytvořil softwarový architekt firmy Unicorn ve volném čase okolo Vánoc za zhruba 150 hodin. Tento nástroj byl pro testování webové aplikace dostačující, ale bohužel není dostupný veřejnosti. Proto jsem hledala jiný framework pro funkční testování, který by byl dostupný jako open source.

JWebUnit je právě takovým frameworkem pro automatizaci funkčních testů webových aplikací. JWebUnit dále popisované verze 1.5 je postavený na dvou dalších open source nástrojích: pracuje na základě frameworku JUnit pro jednotkové testování a k simulaci prohlížeče využívá HtmlUnit plugin stejně jako předchozí WebTest. Informace poskytnuté v této kapitole vychází z vlastních zkušeností s prací s nástrojem a z webových stránek k JWebUnitu [41].

Instalace

Na stránkách jwebunit.sourceforge.net [41], kde je nástroj i ke stažení, jsou popsány dva způsoby instalace podle toho, zda uživatel má další open source nástroj Maven nebo ne. Instalace bez Mavenu se skládá ze stažení a rozbalení souborů a zařazení .jar souborů do classpath[10] v používaném vývojovém prostředí.

Používání

Jediný návod k používání JWebUnit je rychlý úvod do nástroje, neobsahuje ale žádné instrukce, jak testy spouštět, jak pracovat s výsledky a další informace k používání. Jelikož je nástroj postaven na JUnit nástroji a ovládá se stejně, zřejmě je předpokládaná znalost JUnit. To je poněkud zavádějící, vezmeme-li v úvahu, že jednotkové testy s pomocí tohoto nástroje

[10] Proměnná určující, kde se nachází třídy a zdrojové soubory.

píší především programátoři a tester, zejména webových aplikací, nemusí mít s JUnitem žádné zkušenosti.

V rychlém úvodu se uživatel dozví spíše jak psát testy. Ke konci je také jediný odkaz na vygenerovaný popis tříd a metod.

Přestože problém s češtinou JWebUnit nemá, i zde jednoduchý test s vyvoláním stránky a zkontrolováním jejího titulu skončil výjimkou. Problémem bylo vyvolání stránky, které hlásí chybu JavaSkriptu použitého na stránce:

```
com.gargoylesoftware.htmlunit.ScriptException: TypeError: Cannot call method "toLowerCase" of null
```

Podobný problém hlásí i při vyvolání stránky seznam.cz nebo centrum.cz. Zajímavé je, že stejná výjimka nevzniká při testování těchto stránek pomocí nástroje WebTest 2.6, který používá pro simulaci prohlížeče stejný nástroj HtmlUnit i stejnou verzi. Právě tento nástroj by měl s JavaSkripty pracovat.

Pro pokračování v testování bez opravy je možné chyby JavaSkriptu do odvolání potlačit pomocí setScriptingEnabled(false).

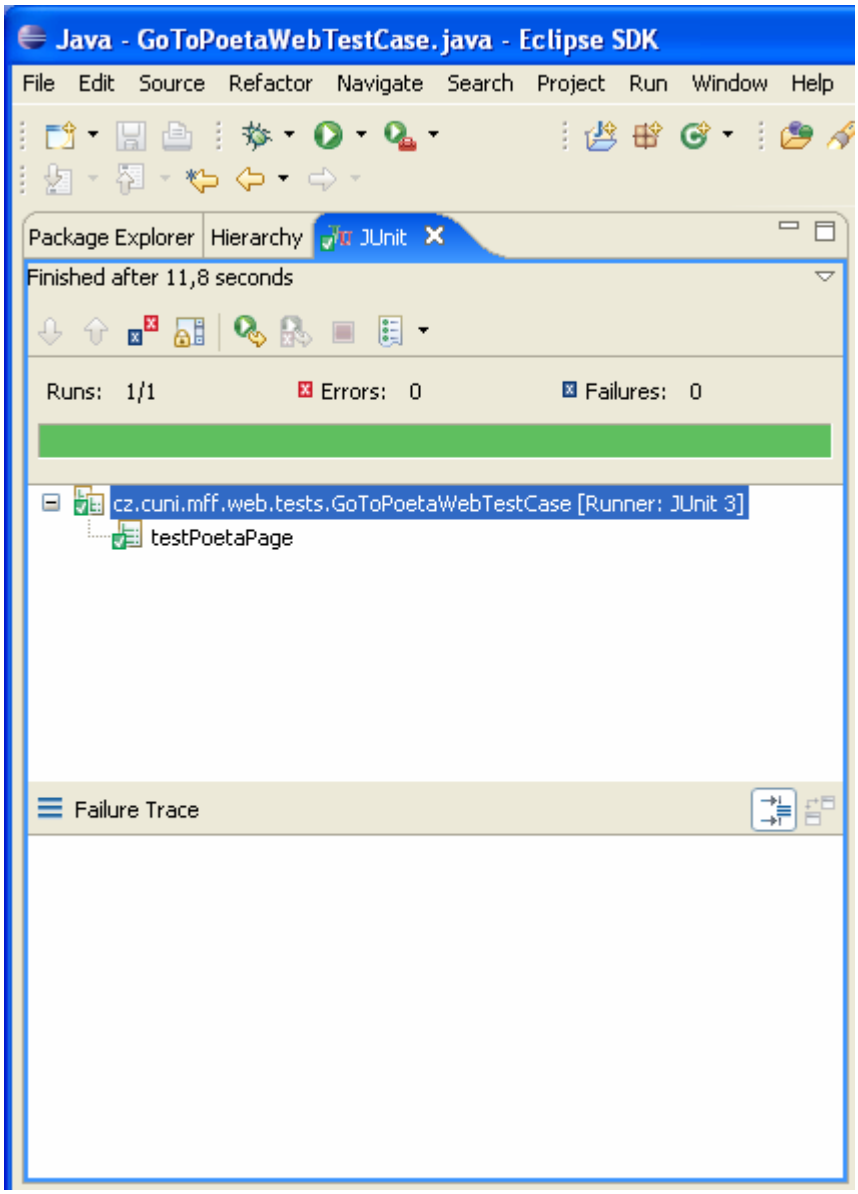
Příklad JWebUnit testu:

```
import net.sourceforge.jwebunit.junit.WebTestCase;
import net.sourceforge.jwebunit.util.TestingEngineRegistry;

public class GoToPoetaWebTestCase extends WebTestCase {

	public GoToPoetaWebTestCase() {
		super();
	}

    public void setUp() throws Exception {
      setTestingEngineKey(TestingEngineRegistry.TESTING_ENGINE_HTMLUNIT);
        getTestContext().setBaseUrl("http://www.poeta.cz");
    }

	public void testPoetaPage() throws Exception {
		setScriptingEnabled(false);
		beginAt("/");
```

```
            assertTitleEquals("Poeta.cz - moderní literární server");
            assertLinkPresentWithText("MůjCitát");
            assertElementPresentByXPath("//div[@class=\"prispevek\"]");
    }
}
```

Přehled výsledků testu

Výsledky testu se zobrazují stejně jako u JUnit. V případě úspěchu se zobrazí pouze zelený pruh a seznam proběhlých testů. V případě chyby nebo výjimky se navíc dozvíme, který příkaz selhal. K zachycení vzhledu testovaných stránek, například po odeslání formuláře je třeba použít kód Javy společně s příkazem frameworku saveAs.



Obr. 13: Výsledek JWebUnit testu

Zhodnocení

JWebUnit může být zajímavým nástrojem pro testery - programátory, nedostatečná dokumentace a slabá prezentace výsledků je však trochu odrazující. Navíc samotné internetové stránky nástroje jsou plné chyb, například se odkazují na neexistující fórum a mají problém s relativními odkazy.

A tak mám dojem, že JWebUnit je spíše polotovarem, který může být dobrým základem, pokud se někomu nebude chtít začínat vytvářet nový nástroj úplně od začátku. Ve stavu v jakém je teď by z něj ale testeři byli spíše nešťastní.

15.3 Rational Functional Tester

Rational Funtional Tester (dále FT) vlastněný firmou IBM je jeden z komerčních nástrojů určených pro automatické funkční a regresní testování. Většina těchto komerčních nástrojů má stejné nebo aspoň podobné rysy:

- Možnost testování aplikací založených na širším spektru technologií – FT je určená pro aplikace postavené na webových technologiích, Javě, SAPu, Siebelu a Microsoft Visual Studio .NETu.
- Funkčnost pro záznam a přehrání akcí testera.
- Možnost editace nebo psaní testů v programovacím jazyce. – FT nabízí Javu nebo Microsoft Visual Basic .NET
- Vytváření testů pomocí skládání z objektů, kontrolních bodů, klíčových slov nebo frází – FT v integraci s Rational Manual Testerem umožňuje používání klíčových slov, vytváření kontrolních bodů a změnu chování skriptu na základě měnění parametrů objektů.
- Řada možných rozšíření nástroje

Dále je popisován Rational Funtional Tester verze 7.0.1.2, jako skriptovací jazyk jsem zvolila Javu. Informace jsou kombinací zkušeností nabraných při zkoušení nástroje a informací z jeho nápovědy.

Instalace

Po zaregistrování na stránkách IBM [24] a stažení souboru pro instalaci, uživatel řídí

instalaci pomocí průvodce, který ho provede všemi nezbytnými kroky. Narozdíl od výše uvedených nástrojů může tento proces stažení a instalace trvat i hodiny.

Používání

Po zapnutí nástroje FT se v menu v nápovědě nachází odkazy na uživatelskou dokumentaci, pro úvod do nástroje je nejzajímavější galerie tutoriálů, pro více tématicky uspořádaných informací slouží odkaz Functional Tester Help. Naučit se vytvořit funkční test podle tutoriálu by mělo zabrat zhruba 45 minut. Díky generování akcí pomocí nahrávání činností a vytváření kontrolních bodů s průvodcem, není třeba se učit před vytvářením skriptu žádné příkazy. Na druhou stranu tato tvorba skriptu pomocí různých pomůcek a nastavení je oproti ručnímu psaní pomalá a zdlouhavá.

Příklad testovací metody v FT:

```
public void testMain(Object[] args)
{
	// otevření webové aplikace
	startApp("www.poeta.cz");

	// HTML Browser
	// Document: Poeta.cz - moderní literární server:
	// http://www.poeta.cz/
	// čekání na načtení stránky, konkrétně středního panelu
	html_stredniPanel().waitForExistence();

	// kontroly odkazu na nejnovější dílo
	link_prvniNove().performTest(prvniNove_kontrola2VP());
	link_prvniNove().performTest(prvniNove_kontrola1VP());

	// kliknutí na odkaz na nejnovější dílo
	link_prvniNove().click();

	// kliknutí na odkaz na nejlepší díla
	link_nejlepšídíla().click();

	// Document: Poeta.cz - moderní literární server:
	// http://www.poeta.cz/top-dilo.php
	// kontrola vzhledu obrázku
	image_nejlepšíDíla().performTest(NejlepšíDíla_kontObrVP());
	// kontrola tabulky s nejlepšími díly
	table_htmlTable_0().performTest(HtmlTable_0_standardVP());

	// zavření webové aplikace
	browser_htmlBrowser(document_poetaCzModerníLiterár2(),MAY_EXIT).close
();
}
```

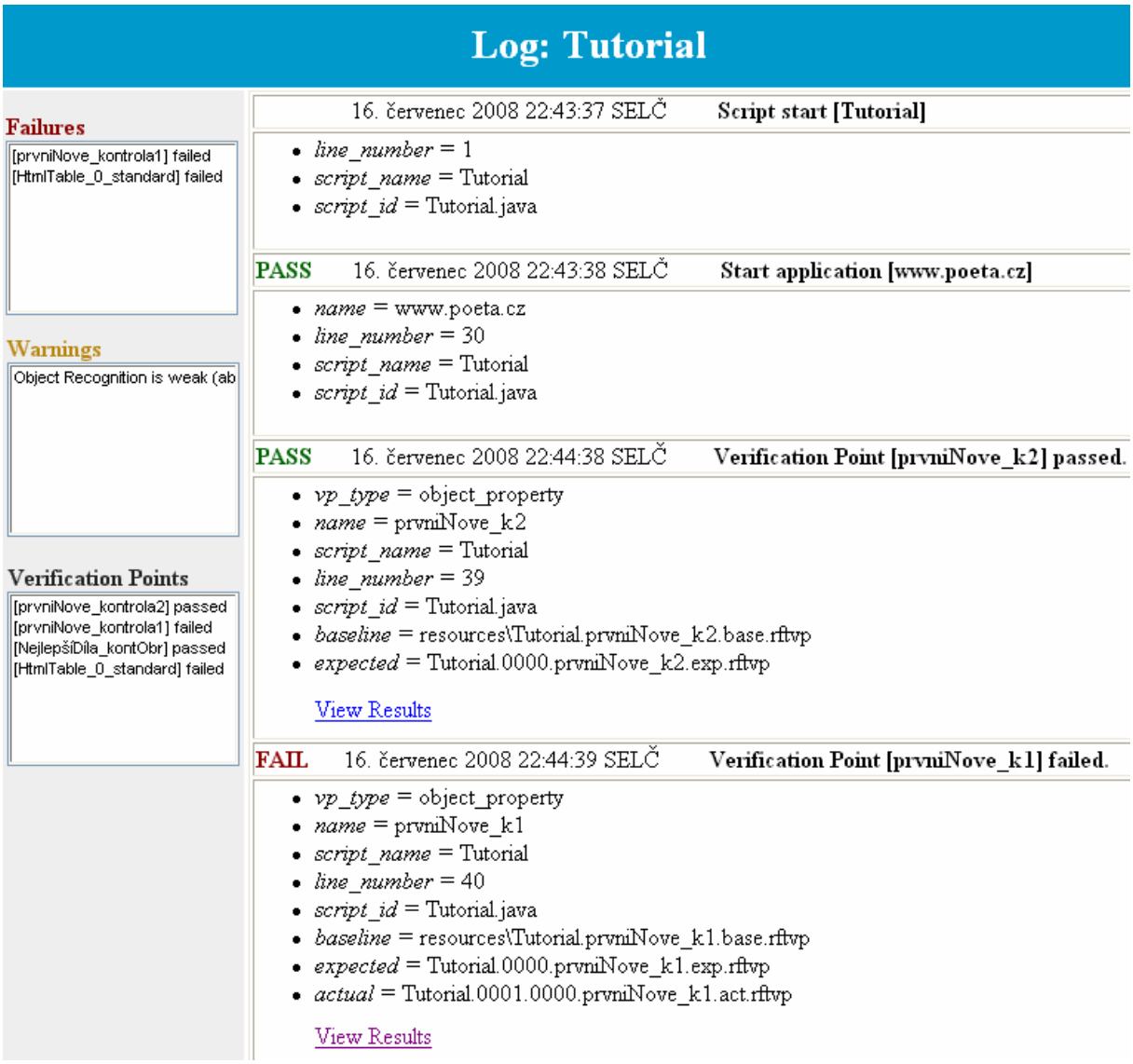
Jak je vidět na předcházejícím obrázku, testovací skript je úzce spjat s objekty, které se při nahrávání akcí rozpoznají a automaticky uloží do množiny objektů. Bez toho, aby tyto objekty byly FT takto zpracovány, není možné s nimi ve skriptu pracovat.

U objektu lze nastavit parametry, jak ho rozpoznávat, díky tomu nástroj pozná, že se jedná o daný objekt i po změnách. Například, přestože nejnovější dílo se změní a třeba se přesune odkaz jinam na stránce, může být stále identifikován podle cesty k němu v html dokumentu, která zůstala stejná.

Přehled výsledků testu

Po skončení testovacího skriptu se zobrazí výsledky testu ve formě tzv. logu. Log je možné prohlížet a ukládat v různých formátech, mezi nimi i ve formě html, která je na níže uvedeném obrázku.

Forma logu je účelová a jednoduchá, přesto trochu nudná a méně přehledná než bych od komerčního nástroje očekávala.



Obr. 14: Výsledek testu v FT

Zhodnocení

Functional Tester je silný nástroj, který lze použít při testování řady různých aplikací s různými požadavky a potřebami. Jeho všestrannost a komplexnost má ale své nevýhody, ovládání nástroje je trochu těžkopádné, nahrávání a práce s objekty zpomaluje psaní skriptů a i samotný nástroj má velmi pomalé reakce.

Například vyvolání vlastností objektu poklepáním na jeho název trvalo v 60% případů více jak 10 sekund (rozptyl od 1 po 46 sekund).

FT je typ nástroje, který je možné použít na mnoha projektech a ušetřit tak peníze na přeškolování testerů na nový nástroj při fluktuaci mezi projekty. Sada speciálních školení a

podpora je jistě také výhodou. Na druhou stranu u velké části webových projektů, kde je potřeba svižného a rychlého vývoje, by použití komplikovaného nástroje nebylo nejideálnější.

16. Automatizace výkonnostních testů

Meier a kol. [30] definuje výkonnostní testování jako typ testování sloužícího k určení rychlosti odpovědí, propustnosti[11], spolehlivosti a škálovatelnosti systému pod určitou zátěží. Přitom je výkonnostní testování běžně prováděno tak, aby dosáhlo následujících cílů:

- Odhadnout připravenost na nasazení do produkce
- Ohodnotit systém ve vztahu k výkonnostním kritériím
- Porovnat výkonnostní charakteristiky více systémů nebo systémových konfigurací
- Najít zdroj výkonnostních problémů
- Poskytnout podporu výkonnostnímu ladění systému
- Nalézt propustnost různých komponent.

Základem výkonnostního testování je opět důkladné zamýšlení se nad aplikací, tentokrát z pohledu architektonického a vývoje výkonnostních požadavků. Výkonnostní testování se vzájemně ovlivňuje s architekturou systému, proto by mělo začínat současně s jejím vytvářením nebo některé aktivity i dřív. Pozdní začátek výkonnostního testování a nevhodně stanovené výkonnostní požadavky vedou s velkou jistotou k neúspěšnému konci, který neodvrátí ani dobré nástroje nebo početný testovací tým.

Smyslem výkonnostního testování je odhalení a docílení odstranění výkonnostně úzkých míst, které se mohou projevit ve vyšší reakční době aplikace, omezení počtu souběžně pracujících uživatelů nebo větší náročnosti na systémové prostředky.

Proces výkonnostního testování je možné rozdělit na pět základních činností, které se v závislosti na událostech mohou opakovat:

1) Identifikace produkčního prostředí a výkonnostních cílů
2) Příprava testovacího prostředí
3) Navržení a napsání potřebných testů
4) Spouštění testů a analýza výsledků
5) Reportování a pomoc při výkonnostním ladění

Zejména během plánování výkonnostních testů je hlavním úkolem porozumět tomu, nakolik budou uživatelé aplikaci používat. Vhodné je sledovat uživatele, jaké stránky v aplikaci navštěvují a jak často. Z této studie, která by měla zahrnovat co nejširší počet

[11] Propustnost představuje počet zpracovaných požadavků klientů za nějakou pevně stanovenou dobu [6].

uživatelů a aspoň několik pracovních dnů, pak vznikají výkonnostní testovací scénáře. Například pokud 25% požadavků na aplikaci připadá na domovskou stránku, výkonnostní testy by tomu měly odpovídat.

Při stanovení výkonnostních požadavků je třeba počítat s postupným vzrůstem počtu uživatelů, ale i prudkými zvýšeními zátěže webové aplikace, třeba díky vydání nové hry, blížících se Vánoc, nebo spojení s jinou firmou. Podkladové informace k očekávané zátěži a jejímu růstu lze čerpat třeba z historických produkčních dat, marketingových studií nebo dokumentů týkajících se plánů firmy.

V souvislosti s výkonnostními požadavky a uživatelskou aktivitou se třídí akce a určují časové hranice na jejich provedení. Například se stanoví, že v 90 procentech případů by mělo vyhledání proběhnout do 1 sekundy. Při stanovení takovýchto hranic se přihlíží ke studiím sledujícím reakce uživatelů na rychlost odpovědí systému. Podle Microsoft ACE Teamu [6] jsou hranice vnímavosti uživatelů stanoveny následujícím způsobem:

do 0,1 sekundy – uživatel má pocit okamžité reakce
do 1 sekundy – není narušen myšlenkový pochod
10 sekund – limit pro udržení pozornosti

Velká pozornost při přípravě testování by měla být věnována vytvoření testovacího prostředí. Cílem je co nejpřesněji simulovat skutečný provoz včetně prostředí, na které bude aplikace nasazena. Důležitý není jenom stejný nebo přinejmenším ekvivalentní hardware a proměnlivá zátěž ale i co nejpodobnější testovací data, jejich objem a různorodost. Pokud vyhledáváme nad databází obsahující 20 uživatelů pojmenovaný uzivatel_01 až uzivatel_20 vyhledávání se může chovat jinak než při třech stovkách různorodých jmen s diakritikou. Čím přesnější je simulace produkčního prostředí, tím důvěryhodnější jsou výsledky testování.

Zatímco při funkčním testování je automatizace jednou z možností, jak si v některých případech ulehčit práci, pokud je rozhodnuto o provedení výkonnostního testování, není defakto jiná možnost než ho provést automatizovaně. Souběžné testování desítek, stovek, tisíců nebo desetitisíců skutečných uživatelů je příliš složité, drahé a špatně reprodukovatelné.

Je možné za určitých podmínek provádět výkonnostní testy za pomoci pár vlastních skriptů napsaných pokročilými programátory nebo zkušenými výkonnostními testery. Výhodnější ale je se naučit používat některý z dostupných open source případně i komerčních nástrojů.

Nástroje pro výkonnostní testování pracují tak, že vytvoří požadovaný počet vláken[12], popřípadě procesů[13], které vykonávají napsaný testovací scénář, někdy i několik scénářů, a zaznamenávají doby zpracování požadavků. Díky tomu mohou vytvořit desítky až tisíce virtuálních uživatelů. Testovací scénáře jsou velmi podobné těm, které se vytvářejí pro funkční testování, jen nejsou tolik zaměřeny na kontrolu funkčnosti a řídí se výsledky zkoumání uživatelské aktivity, tedy tím, které stránky a které funkce jsou nejčastěji zastoupeny a v jaké míře. Nástroje pro automatizované testovaní umožňují provádět tyto scénáře opakovaně a s postupným přidáváním uživatelů.

To má význam v tom, že testeři mohou spustit nejdřív jednoho virtuálního uživatele, aby se aplikace takzvaně zahřála, protože odpověď na první požadavek může být výrazně pomalejší, než u druhého nebo třetího. Postupně se pak přidávají další uživatelé. Opakováním scénářů se docílí toho, že prvně spuštění virtuální uživatelé neskončí dřív než se docílí nastaveného počtu uživatelů. Pokud není záměrem něco jiného než zjistit rychlost odpovědí aplikace při dané zátěži, tak by výsledky měly být sbírány až po dosažení plného počtu virtuálních uživatelů.

Po napsání skriptu lze s jeho pomocí zkoumat aplikaci na několika úrovních:

1) Spuštění a vyhodnocení pro jednoho uživatele a porovnání doby trvaní jednotlivých činností pomáhá identifikovat výkonnostní trendy.
2) Zkoumání souhrnných statistik přes více skriptů pro konkrétní případy užití poskytuje pohled na různé odezvy systému, jak se bude jevit uživatelům
3) Provádění detailních analýz za pomocí statistických metod ve snaze najít a pochopit vznik úzkých míst. Na základě těchto analýz se dělají rozhodnutí, odhadují se rizika a hodnotí jejich řešení.

Pokud výkonnostní testování odhalí nedostatky, programátoři provádějí výkonnostní ladění, které má za cíl odhalit konkrétní příčinu problému. Testovací tým by měl být schopen tento proces programátorům co nejvíce usnadnit podáním přesných informací a pomoci jim s laděním. Nezanedbatelnou přípravou na toto ladění by mělo být pravidelné provádění výkonnostních testů, díky čemuž je jasné, ve které verzi došlo k poklesu výkonnosti a seznam změn, z nichž některá zřejmě pokles způsobila.

[12] Vlákno je systémový objekt, který je charakterizován svým stavem. Dá se představit jako línie výpočtu, běhu. Vlákna existují uvnitř procesu.
[13] Proces je systémový objekt charakterizován svým obsahem, představuje tedy kód a data v paměti.

16.1 Apache JMeter

Apache JMeter je oblíbený open source nástroj pro výkonnostní testování zejména webových aplikací. Postupně je rozšiřován o další funkčnosti, které umožňují testování i jiných klient/server systémů. V současné době je nejnovější níže popisovaná verze JMeter 2.3.2, která podporuje servery založené na technologiích JMS, LDAP, POP3, JDBC a další. Není-li řečeno jinak, uvedený popis JMeteru vychází ze zkušeností nabraných při práci s nástrojem.

JMeter je napsaný v Javě a navržen tak, aby byl snadno rozšiřitelný. Vytváření testů neprobíhá psaním skriptů, ale skládáním požadavků, kontrol a přidáváním logických prvků. Stejným způsobem se vybírá forma zobrazení výsledků.

Instalace

Na stránkách jakarta.apache.org je k dispozici JMeter nebo jeho zdrojové soubory ve formě archívů s příponou .tgz a .zip. Na stránce pro stažení je i návod jak provést verifikaci stažené verze. Každá verze je podepsaná s cílem zajistit integritu souboru, že stažení proběhlo správně.

K běhu aplikace je zapotřebí mít nainstalovanou Java Virtual Machine (JMV) verze 1.4 nebo vyšší, další povinné instalace nejsou. Po stažení souboru s nástrojem jej stačí rozbalit do vybraného adresáře, cesta k tomuto adresáři by neměla obsahovat mezery. Pokud je v proměnných prostředí nastavená cesta k potřebné JVM, aby ji nástroj mohl nalézt, není potřeba dělat nic dalšího.

Spustitelný soubor k nástroji, jmeter.bat, je v adresáři bin spolu s dalšími klíčovými soubory. Dále adresářová struktura rozbaleného souboru obsahuje adresář lib s potřebnými knihovnami, adresář doc s popisem metod a tříd, tedy Javadoc dokumentaci a adresář printable_doc s uživatelským manuálem. Veškerá dokumentace je k dispozici i na výše zmíněných webových stránkách nástroje.

Použití

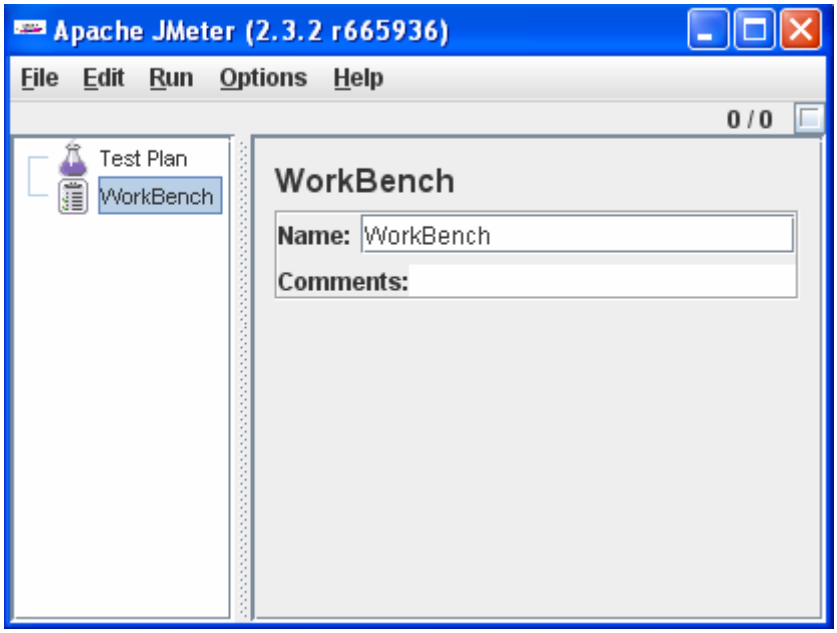
Přestože JMeter není komerční nástroj má dosti rozsáhlou dokumentaci, díky které lze nejen rychle začít vytvářet testy, ale i najít odpovědi na další otázky. Hlavním dokumentem je uživatelská příručka, která obsahuje základní informace o JMeteru, popis instalace, popis práce s nástrojem a přehled komponent, ze kterých se test připravuje. Protože s JMetrem se

pracuje méně obvyklým způsobem, čtení dokumentace bez otevřeného nástroje a okamžitého provádění napsaných instrukcí, není moc přínosné a může zprvu mást. Zajímavé jsou i tutoriály poskytující jednoduché návody typu jak provádět distribuované testování s JMetrem nebo nahrávat testy za pomocí proxy. Další dostupná dokumentace se pak hodí k zásahům do zdrojových souborů nebo k řešení problémů.

Po zapnutí JMeteru není bez uživatelské příručky příliš jasné, jak pokračovat. V menu kromě nápovědy není nic k vytváření testů a aplikace kromě dvou prázdných prvků nic neobsahuje, viz obr. 15. JMeter v tomto není intuitivní, na druhou stranu způsob práce s ním je jednoduchý a rychle si na něj lze navyknout.

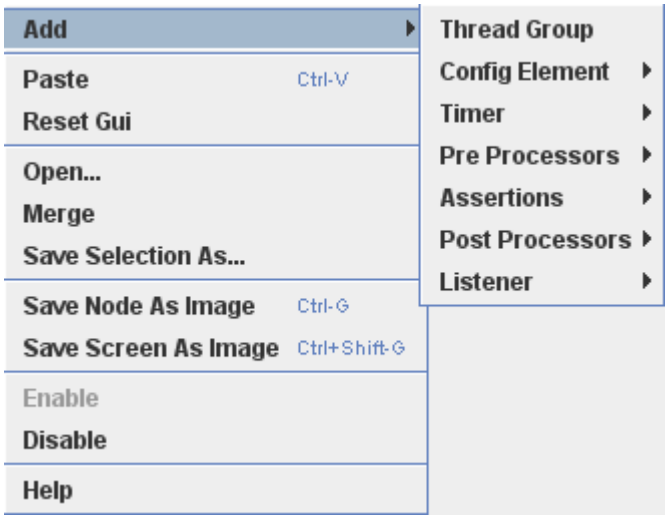
Dva prvky, které se zobrazí po zapnutí aplikace jsou:

- Testovací plán (Test Plan) – Pod tímto mírně matoucím označením se skrývá prvek zastřešující testovací scénář nebo scénáře, které zde lze vytvořit a spustit.
- Pracovní stůl (WorkBench) – Od předchozího prvku se příliš neliší, i zde se dají přidávat testy, rozdíl je v tom, že slouží pro odkládání momentálně nepotřebných prvků a pro prvky, které neslouží k testování, např proxy server.



Obr. 15: JMeter po zapnutí

Celý test je tvořen přidáváním a vyplňováním dalších prvků do stromečkové struktury a to kliknutím na nadřazený prvek pravým tlačítkem a zvolením nabídky přidat (Add), viz obr. 16.



Obr. 16: Ovládání přes kontextové menu

Jak bylo už výše řečeno samotné testy se tvoří přidáváním a kombinací prvků a jejich nastavením. Jelikož prvky jsou obecné a jejich nastavení spočívá v jednom nebo dvou parametrech, je v případě JMeteru tento přístup docela rychlý. A to i díky snadnému zkopírování libovolného prvku i se všemi jeho podřízenými prvky a úpravě parametrů.

Pro napsání testu se pod testovací plán přidá prvek Tread group představující vlákno nebo taky scénář. Tento prvek obsahuje parametry pro určení počtu uživatelů, jejich postupné přidávání a informace, kolikrát se má scénář opakovat.

Podle zdroje na wiki k JMeteru [26] se maximální počet uživatelů na jednu instanci JMeteru liší podle toho, jaké prvky jsou pod testovacím plánem. Ale není výjimkou, že uživatelé hlásili i 1000 vláken na jednu instanci. Většího počtu uživatelů pak lze dosáhnout použitím více počítačů, každého s vlastní instancí, z nichž jeden ostatní řídí. Jak na to, je popsáno ve výše zmíněném tutoriálu pro distribuované testování.

Aby výkonnostní testování co nejvíce odpovídalo realitě a měření byla co nejvíce přesná, je výhodou pokud nástroj neumožňuje pouze ledabylé naskládání požadavků za sebou, ale umožňuje simulaci funkčních testů. Důležité je zejména přidání nahodilého

chování uživatele, desítky současně pracujících uživatelů nebudou zadávat vždy stejná data, a doplnění kontrol, zajišťujících, že pokud požadavek skončí chybou, testeři se to dozví.

Ovládání zejména pořadí požadavků poskytují prvky zvané kontroléry (controllers). Ty řídí požadavky tak, jak by to dělaly řídící příkazy programovacího jazyka. Mezi zajímavé kontroléry patří například:

- Random controller – provede jen jeden náhodně vybraný podřízený požadavek.
- Random order controller – provede podřízené požadavky v náhodném pořadí.
- If controller – umožňuje regulovat, kdy provést podřízené požadavky a kdy ne.
- While controller – podřízené požadavky běží v cyklu, dokud nastavená podmínka se nevyhodnotí jako nepravda.
- ForEach controller – podřízené požadavky se opakují pro každou z množiny souvisejících hodnot.
- Record controller – jeden z kontrolérů, které neřídí pořadí požadavků, namísto toho slouží jako označení umístění, kam se uloží požadavky získané při nahrávání přes proxy server.

Kontroly zajišťují prvky zvané assertions, které se vztahují vždy ke konkrétnímu požadavku. Nejčastěji využívané zřejmě jsou následující dvě kontroly:

- Response assertion – umožňuje zadání regulárního výrazu nebo textu, který má odpověď na daný požadavek obsahovat.
- XPath assertion – kontroluje zda zadaný dotaz XPath je úspěšný, případně umí taky zkontrolovat zda je dokument správně utvořený a validní.

Kromě kontrol JMeter obsahuje další skupinu prvků pro práci s odpověďmi, kterou nazývá post-procesory (post-processors). Post-procesor je aplikován na všechny požadavky na stejné úrovni nebo na požadavek, ke kterému je přidán jako potomek ve stromové struktuře.

K individuálním požadavkům i k jejich množině se dobře hodí následující dva post-procesory pro práci s proměnnými:

- Regular Expression Extractor – ukládá do proměnných výsledky regulárního výrazu.
- XPath Extractor – ukládá do proměnné výsledky dotazu XPath.

Další post-procesory například umožňují uložení odpovědi, provedení daného kódu, nebo zastavení v případě chyby.

JMeter obsahuje i další skupiny prvků, které ovlivňují a rozšiřují možnosti nástroje a o kterých se lze dočíst v jeho uživatelské dokumentaci.

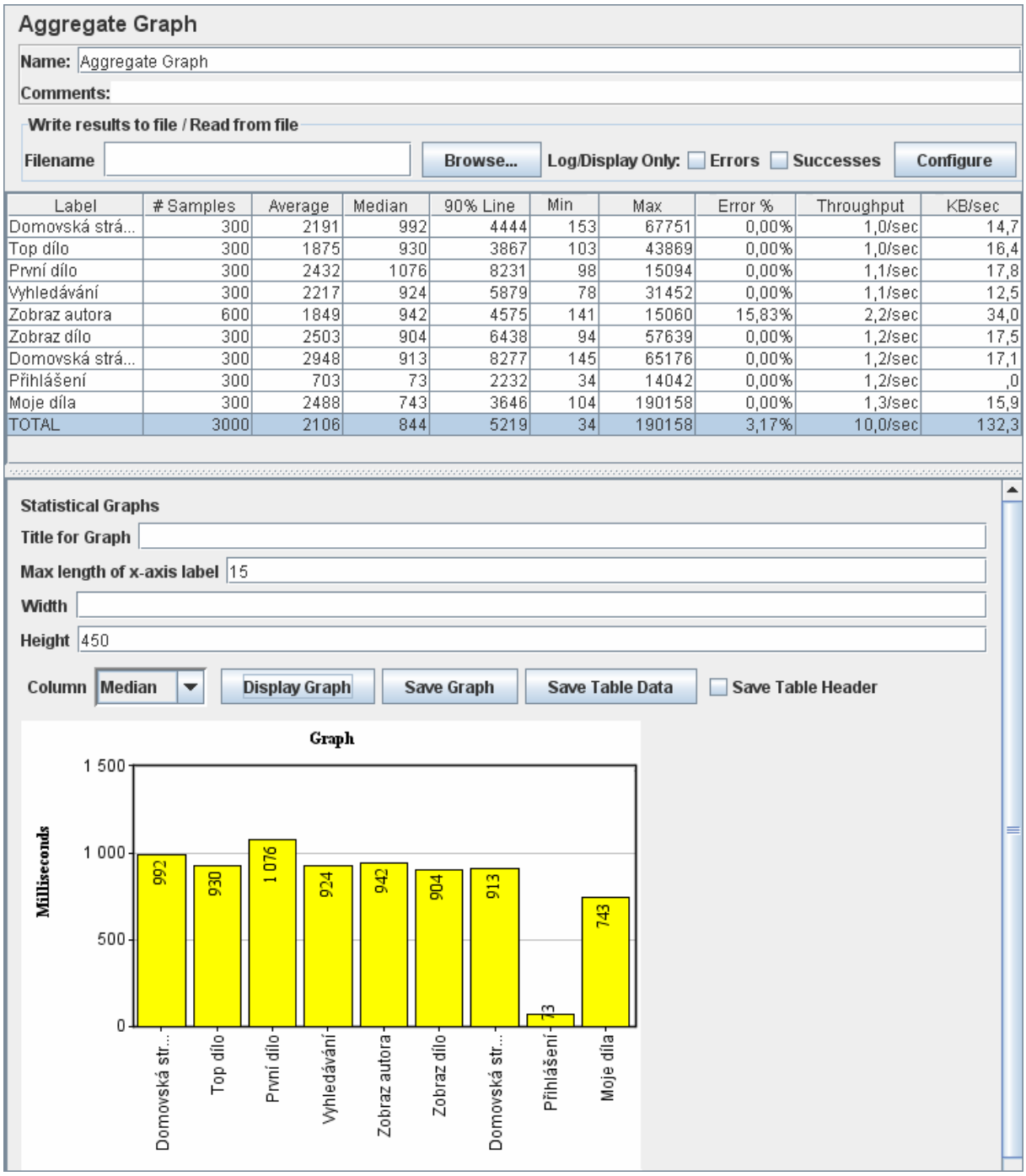
Přehled výsledků testu

JMeter umožňuje regulovat zobrazení výsledků testu stejným způsobem jako se test vytváří. Do stromové struktury testovacího plánu lze přidat prvky z množiny takzvaných listenerů, pojmenovaných podle toho že naslouchají akcím. Tyto listenery umožňují zobrazit, uložit a otevřít zachycené výsledky. Každý prvek z této skupiny poskytuje pohled na jiný typ výsledků a jiné zobrazení. Listenerů může být přidáno víc, což umožňuje podrobnější zkoumání výsledků.

Protože listenery ovlivňují množství paměti, kterou JMeter potřebuje, je lepší jejich počet a typ upravovat podle momentální potřeby. Pro psaní skriptu a testování s relativně nízkým počtem uživatelů je možné použít více podrobnějších listenerů. Při dlouhodobých testech s velkým počtem uživatelů je nejlepší použít souhrnné výsledky jako u listenerů: Aggregate Listener, Graph Listener nebo Spine Listener. V opačném případě může dojít k chybám nedostatku paměti (out of memory) [27].

Zhodnocení

S JMetrem se mi velmi dobře pracovalo. Umožňuje snadno a rychle vytvářet výkonnostní testy. Má dobrou dokumentaci, která poskytuje řadu návodů, které uživatele krok za krokem provázejí. Na druhou stranu jsem zaznamenala řadu negativních komentářů od lidí, kteří JMeter pro testování používali. Kritika se týkala zvláště chybovosti a nedostatečného monitorování systému. Jiné komentáře zase nástroj chválily. Zda se jednalo o problémy, které už byly vyřešeny nebo jsou stále přítomné, ale vadí jen na určitých projektech, vyžaduje podrobnější zkoumání. Případné pozdější objevení chyb nebo potřeby rozšířit stávající funkčnost lze zajistit pomocí úpravy zdrojových souborů, které jsou uživatelům k dispozici.



Obr. 17: Výsledky v Aggregate Graf Listener

16.2 Grinder

Grinder je framework postavený nad Javou pro výkonnostní testování dostupný pod open source licencí určený hlavně k testování webových aplikací, i když podle [15] zvládá

testování čehokoli, co má Javovské API[14]. Testovací skripty pro Grinder používají skriptovací jazyk Jython, jenž je javovskou implementací jazyka Python.

Grinder je navržen jako více vláknový a více procesový nástroj, díky čemuž umožňuje regulovat, jakým způsobem lze simulovat uživatele.

Počet uživatelů = počet strojů * počet procesů na jeden stroj * počet vláken na jeden proces

Nejnovější níže popisovaná verze je The Grinder 3.1 Popis nástroje vychází ze zkušeností nabraných při práci s nástrojem a z jeho popisu [15].

Instalace

Grinder je k dispozici volně ke stažení na stránkách sourceforge.net/projects/grinder ve formě souboru s příponou .zip.

Nástroj potřebuje J2SE verze 1.4 nebo vyšší. Po rozbalení obsahu souboru, je třeba uložit do proměnné prostředí CLASSPATH cestu k souboru grinder.jar, který se nachází ve vytvořené adresářové struktuře v podadresáři lib.

Poté je třeba vytvořit podle pokynů na webových stránkách nástroje [15] soubor grinder.properties. Jinou možností je použít a upravit vzor tohoto souboru, který se nachází v podadresáři examples.

V grinder.properties se specifikuje i název testovacího skriptu, který se má spustit. Pro první spuštění je vhodné vybrat některý z předpřipravených příkladů, které jsou rovněž v adresáři examples.

Nástroj se pak spouští zapnutím konzoly, která slouží k ovládání tzv. agent procesů, které na jednotlivých strojích spouští testy podle souboru grinder.properties.
Příkaz pro zapnutí konzoly:

```
java net.grinder.Console
```

a agent procesu:

```
java net.grinder.Grinder
```

[14] Application Programming Interface – zveřejněný seznam funkcí , které jsou dostupné k využití pro jiné programy.

Kromě příkladů dokumentace ve stažené adresářové struktuře není, ale lze ji stáhnout ze stránek nástroje ve formě pdf souboru.

Použití

Nástroj je zaměřený spíše na programátory než na testery, a proto je i dokumentace psána tak, že hodně kroků a vysvětlení se přeskakuje, protože je bráno jako implicitní. Zmatení z dokumentace ještě umocňuje to, že autor skáče z jednoho tématu na druhé, nebo je část kroků vyčleněna do jiného dokumentu.

Pro psaní testů by uživatel měl předem znát skriptovací jazyk Python/Jython a Javu. Struktura skriptů se dá ale pochopit z příkladů a pokud by byla neprogramátorům dobře vysvětlena zvládnou psaní jednoduchých testů také. Bohužel, to vyžaduje aby někdo Grinder zjednodušil a vypracoval dokumentaci zaměřenou i na tyto lidi.

Grinder obsahuje i pomůcku na nahrávání skriptů TCP proxy. Po jejím spuštění, se nastaví v prohlížeči připojení přes tento proxy server a začnou nahrávat požadavky. Nahráním testovacího scénáře obsahujícího osm akcí, tedy vyvoláním osmi webových stránek, vznikl skript, který měl třicet stran. Procházení takového automaticky nahraného skriptu a snaha o jeho úpravu, práci testerům bez programátorských zkušeností neulehčí.

Příklad velmi jednoduchého scénáře přepsaného do skriptu pro Grinder:

```
from net.grinder.script.Grinder import grinder
from net.grinder.script import Test
from net.grinder.plugin.http import HTTPRequest
from HTTPClient import NVPair

test1 = Test(1, "Poeta main")
request1 = test1.wrap(HTTPRequest())

test2 = Test(2, "POST hledat.php")
request2 = test2.wrap(HTTPRequest(url="http://www.poeta.cz/"))

class TestRunner:
   def __call__(self):
      result1 = request1.GET("http://www.poeta.cz/")
      result2 = request2.POST('/hledat.php',
         ( NVPair('Hledat', 'Fajko'), ),
         ( NVPair('Content-Type', 'application/x-www-form-urlencoded'), ))
```

```
        writeToFile(result2.text)

def writeToFile(text):
    filename = grinder.getFilenameFactory().createFilename(
        "page", "-%d.html" % grinder.runNumber)

    file = open(filename, "w")
    print >> file, text
    file.close()
```

Přehled výsledků testu

Přestože konzola, z které testy spouštíte, obsahuje záložky s výsledky a grafy, informace v nich nejsou až tak významné. Mnohem důležitější jsou soubory, do kterých procesy Grinderu logují výsledky:

- out-host-n.log – informace o běhu testovacího procesu číslo n na počítači s názvem host
- error-host-n.log – vytvoří se pouze pokud dojde k chybě, obsahuje její popis
- data-host-n.log – výsledky testu za konkrétní proces

Výsledky nejsou samy o sobě Grinderem zpracovány, pozdější analýza je ponechána na uživateli.

Zhodnocení

Grinder je silným nástrojem pro výkonnostní testy, ale není uživatelsky přívětivý k testerům. Pokud je brán pouze jako základ a ne jako hotový nástroj, je Grinder dobrým řešením. Zejména po zjednodušení psaní testů, dopsání podrobnější uživatelské příručky a nástroje na generování grafů a tabulek z dat, může být velkým přínosem pro projekt. Záleží, jak se programátoři vypořádají se zjednodušením skriptů pro testery.

16.3 LoadRunner

LoadRunner je komerční nástroj určený pro výkonnostní testování, který byl vyvinut firmou Mercury, která nyní patří firmě Hewlett-Packard. Kromě webových aplikací je

schopen otestovat aplikace založené na webových službách, Javě, .NETu, také Oracle, SAP, Siebele a další. Níže vyzkoušená a popsaná verze je LoadRunner 9.10.0.0.

Instalace

Po zaregistrování na stránkách Hewlett-Packard je k dispozici ke stažení soubor s instalací LoadRunneru a desetidenní licencí. Po stažení a rozbalení tohoto 2GB souboru, se ve vytvořené struktuře nachází soubor setup.exe, který nainstaluje LaodRunner i všechny potřebné součásti.

Použití

LoadRunner se skládá z několika provázaných aplikací, z nichž nejdůležitější jsou:
Virtual User Generator – Slouží pro nahrání a úpravu skriptů, skripty simulují činnost virtuálních uživatelů.
Controller – Spouští skripty, zobrazuje výsledky už v průběhu simulace uživatelů.
Analysis – Umožňuje podrobnou výkonnostní analýzu výsledků.

Aplikace je možné spustit buď samostatně nebo z LoadRunneru, který sám slouží pouze jako výchozí bod.

Každá větší část LoadRunneru obsahuje klasickou nápovědu s uspořádáním podle témat, pojmů a s vyhledáváním. Dále instalace obsahuje pdf soubory tutorial a podrobné příručky k výše zmíněným částem (například příručka k části Analysis má 668 stran), k instalaci, a další užitečné soubory s dokumentací.

Kvůli složitosti LoadRunneru a velkému množství dokumentace zvládnutí nástroje je pomalejší, ale ovládání není složité a zpočátku není třeba ani programátorských znalostí.

Testovací skripty se vytváří nahráním akcí skutečného uživatele a následnou úpravou. Skript je možné prohlížet v podobě psaného přehledného kódu nebo jako stromovou strukturu akcí.

```
web_add_cookie("pmv_ck_2=YTo2OntzOjg6ImlkY29va2llIjtzOjMyOiIwMDg5NDRiY2

web_add_cookie("pmv_ck_4=YTo2OntzOjg6ImlkY29va2llIjtzOjMyOiIyODYxYjRkNz

web_url("www.poeta.cz",
    "URL=http://www.poeta.cz/",
    "Resource=0",
    "RecContentType=text/html",
    "Referer=",
    "Snapshot=t1.inf",
    "Mode=HTML",
    EXTRARES,
    "Url=/css/poeta.css", ENDITEM,
    "Url=/css/img/back-main.gif", ENDITEM,
    "Url=/css/img/reklama-back.gif", ENDITEM,
    "Url=/img/banner_knihovnicka_468.swf", ENDITEM,
    "Url=/css/img/ramecek-back.gif", ENDITEM,
    "Url=/css/img/logo-server.gif", ENDITEM,
    "Url=/css/img/underheader-l.gif", ENDITEM,
    "Url=/css/img/underheader-r.gif", ENDITEM,
    "Url=/css/img/ramecek-h2-back.gif", ENDITEM,
    "Url=/css/img/ramecek-basne.gif", ENDITEM,
    "Url=/css/img/ramecek-povidky.gif", ENDITEM,
    "Url=/css/img/ramecek-sms.gif", ENDITEM,
    "Url=/css/img/ramecek-ostatni.gif", ENDITEM,
    "Url=/css/img/ramecek-cykly.gif", ENDITEM,
    "Url=/css/img/ramecek-komentare.gif", ENDITEM,
    "Url=/css/img/news.gif", ENDITEM,
    "Url=/css/img/ramecek-r-h2-back.gif", ENDITEM,
    "Url=/stat/phpmyvisites.php?url=//www.poeta.cz/&pagename=&id=1&res
    LAST);

lr_think_time(5);


web_link("NEJLEPÍ ÁŤ DÁŤLA",
    "Text=NEJLEPÍ ÁŤ DÁŤLA",
    "Snapshot=t2.inf",
    EXTRARES,
    "Url=/stat/phpmyvisites.php?url=//www.poeta.cz/top-dilo.php&pagenar
    LAST);


web_link("Top dilo",
    "Ordinal=23",
    "Snapshot=t3.inf",
    EXTRARES,
    "Url=/stat/phpmyvisites.php?url=//www.poeta.cz/read.php%3Fid%3D870
    LAST);


web_submit_form("hledat.php",
    "Snapshot=t6.inf",
    ITEMDATA,
    "Name=Hledat", "Value=mon", ENDITEM,
    EXTRARES,
    "Url=/stat/phpmyvisites.php?url=//www.poeta.cz/hledat.php&pagename=
    LAST);
```

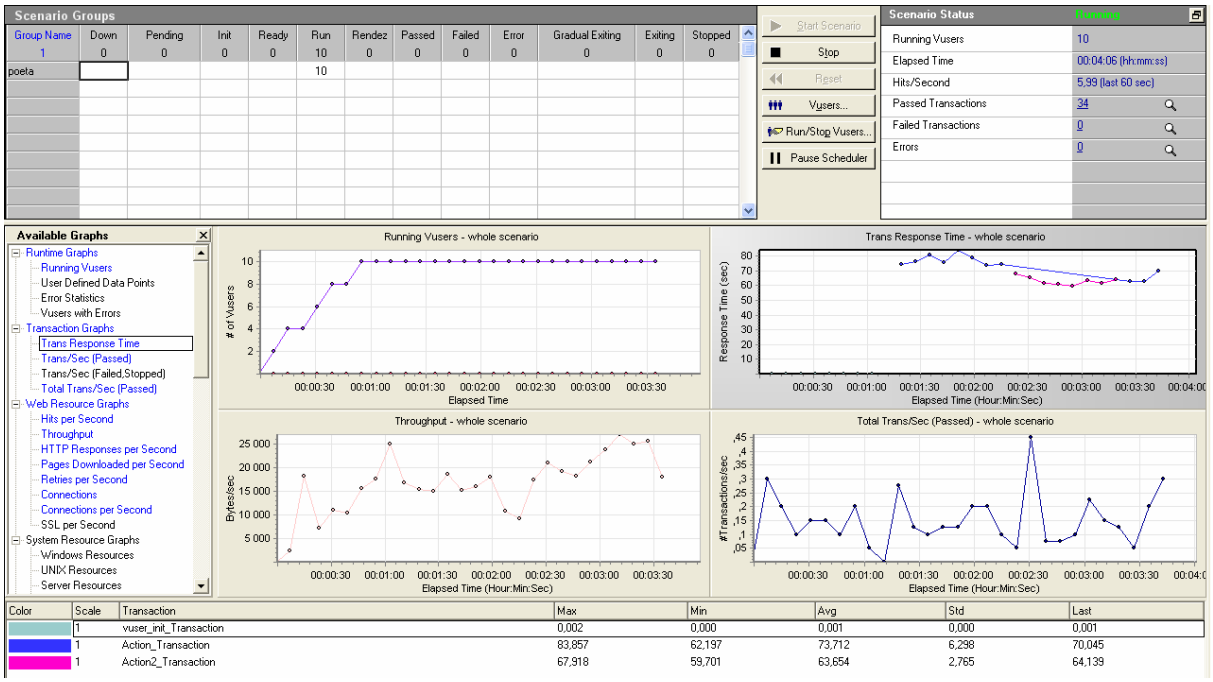
Obr. 18: Testovací skript LoadRunneru v podobě kódu

Po nahrání skriptu LoadRunner doporučuje jeho kontrolní přehrání pro jednoho virtuálního uživatele, vylepšení pomocí přidání proměnných, a kontrol obsahu a ujasnění toho, co má být měřeno.

Poté se připravuje skript pro spuštění výkonnostních testů, určuje se kolik virtuálních uživatelů a s jakými nastaveními má běžet, jak dlouho a další parametry běhu skriptu.

Přehled výsledků testu

Už v průběhu skriptu je možné si prohlížet výsledky zpracované do nejrůznějších grafů. Pro následnou analýzu slouží pak aplikace Analysis, kterou lze spustit příslušným tlačítkem po běhu výkonnostního skriptu. V případě že je otevřena zvlášť, je třeba otevřít uložené výsledky běhu (.lrr soubor) nebo už dříve analyzované výsledky (.lra soubor).



Obr. 19: Výsledky v Controlleru

Zhodnocení

LoadRunner je komplexní nástroj, který se trochu složitější ovládání snaží vynahradit uživatelskou přívětivostí.

17. Bezpečnost webových aplikací

Prošla jsem za poslední dobu mnoho knih o tvorbě webových stránek a programování webových aplikací. Sedla jsem si do knihovny nebo do knihkupectví a jednu knihu po druhé jsem brala do ruky a vyhledávala v nich zmínky o bezpečnosti. Jen jediná kniha z těch, co jsem během tohoto náhodného průzkumu viděla, věnovala kapitolu bezpečnosti, ostatní problémy bezpečnosti aplikací čtenářům nevysvětlovaly. Bohužel, tato jediná kniha, která se zabývala trochu bezpečností, spíše vyvolá mylný dojem, že bezpečnost si zajistíte firewallem a oddělením logiky. Bezpečností kódu se autor v knize nezabývá.

Bruce Schneier[15] o používání firewallu nedávno řekl [37]:

„The reason you're buying a firewall is because you're network's hardware and software isn't secure and that functionality should be embedded in your network."

Přeloženo:

„Důvodem, proč si kupujete firewall je, že váš síťový hardware a software není bezpečný a proto by tato funkcionalita měla být přidána do vaší sítě."

Firewall je tedy jakousi záplatou na problémy, jenomže ať je jakkoli dobře nakonfigurovaný, nezabezpečí bezpečnou komunikaci. Chrání pouze před určitým druhem útoků. V záznamu z konference pořádané společností Google[14] se můžeme setkat s pojmem Universal Firewall Bypass Protocol (Univerzální protokol pro obcházení firewallu), což je míněno jako přezdívka pro http. Z hlediska vývoje aplikace se bezpečnost dosahuje kvalitní a odbornou kontrolou kódu. Obecně je dosažení určité úrovně bezpečnosti otázkou procesu jejího řízení, ale to už přesahuje oblast testování.

Kromě nedostatečně vysvětlené problematiky bezpečnosti kódu, jsem často narazila i na ukázky kódu, které obsahovaly bezpečnostní díry. Pokud vycházíme z toho, že z těchto knih se učí začínající programátoři, znamená to, že každý takovýto nezkušený programátor přináší do tvorby webových aplikací velké riziko. Úkolem testovacího týmu je si toto riziko uvědomit a snažit se ho řešit. Například je možné věnovat mnohem více úsilí testování

[15] Bruce Schneier je odborník na bezpečnost a kryptografii, mimo jiné autor algoritmu Blow-fish.

bezpečnosti webové aplikace, kterou vyvíjí tým s velkým procentem nováčků, nebo navrhnout vytvoření školení o bezpečnosti, kterým si tito programátoři povinně projdou.

K pochopení bezpečnosti nestačí testerovi si přečíst jednu kapitolu, tato oblast vyžaduje nasazení člověka specializovaného na tento druh testování. Ten musí být zkušeným programátorem, mít znalost nástrojů pro testování bezpečnosti a neustále tříbit svoje schopnosti dostat se do cizích aplikací. Od testera bezpečnosti je požadováno držet krok s novými technologiemi. Zatímco ostatní testeři se snaží dívat na webovou aplikaci z hlediska potenciálních návštěvníků, u bezpečnostního testování se tester na ní dívá z pozice hackera.

Z tohoto důvodu nemají testeři dostatečné znalosti a zkušenosti, pokud se této oblasti usilovně nevěnují. Samozřejmě i za pomoci náhodných vědomostí některých známých bezpečnostních děr, může kdokoli nalézt pár chyb, ale v takovém případě se nejedná o komplexní testování, ale spíše o štěstí.

K testování bezpečnosti může být přijat jeden ze tří přístupů:

- Najmout nebo vyškolit vlastního testera pro testování bezpečnosti
- Najmout externí firmu na provedení těchto testů
- Spoléhat na štěstí a náhodu, protože výše uvedené možnosti se finančně nevyplatí.

U testování bezpečnosti existují při pravidla, která považuji za zásadní:

Důkladně otestovat je potřeba **vše, k čemu může útočník získat přístup nebo co může být útočníkem změněno**, přestože běžný uživatel by to nebyl schopen změnit. Taková místa může být obtížné najít a ještě těžší je naučit se jich využívat tak, jak by to udělal útočník.

Bezpečnost je třeba budovat ve více vrstvách, to znamená, že při jejím testování **se ptáme, co se stane, když se útočník dostane přes konkrétní vrstvu**. U tohoto pravidla vycházíme z toho, že testování neprokáže nepřítomnost chyb a také že příliš podrobné testování není ekonomické. Tady je třeba předpokládat, že v softwaru vždy zůstanou chyby. Z tohoto pravidla vyplývají mimo jiné dvě další, která platí pro programátory a systémové administrátory a to je princip udělování nejmenších práv a nikdy nevěřit v bezpečnost jiné části systému.

Při testování obecně a zejména při testování bezpečnosti webových aplikací je třeba **myslet kreativně**. Pokud se testování stává rutinou, je to obecně znakem toho, že provádíme jen zlomek testů, které jsou si navzájem velmi podobné.

Přestože se na první pohled zdá, že se klade na bezpečnost velký důraz, systémy jsou napadnutelné skrze řadu neopravených bezpečnostních děr, z nichž některé jsou zveřejněné. Podle studie IBM za rok 2007 [25] nechalo pět největších dodavatelů softwaru 20% zjištěných bezpečnostních děr neopravených, u ostatních dodavatelů to bylo 50%.

Pro využití těchto zranitelných míst řady systémů jsou vyvíjeny tzv. web exploit toolkity, nástroje, které už obsahují funkce pro útočení na častá zranitelná místa, takže útoky jsou pro majitele těchto toolkitů jednodušší. Nepotřebují vymýšlet, jak neošetřené bezpečnostní problémy využít, pouze jim stačí najít systém, který je obsahuje. Zdá se proto, že je čím dál lehčí být hackerem a napadat systémy.

Z tohoto důvodu se domnívám, že útoků je daleko méně, než by ve skutečnosti mohlo být. Doba kdy hackeři byli zromantizovanými postavami už končí. Útoky už se neprovádějí tolik pro zábavu jako kdysi, provádějí se více za účelem zisku, stejně jako normální krádeže a podvody v reálném světě. Málokdy by někdo vykradl dům pro zábavu, pro zábavu se vykrádají spíže a vandalizují stěny. Dá se proto předpokládat, že zatímco útoků ubude, budou nabírat na závažnosti. Bez zlepšování bezpečnosti, a ověření, zda současná opatření nedávají jen falešný pocit bezpečí, mohou být aplikace v budoucnu velmi snadným terčem. Proto je testování bezpečnosti tak významným a složitým úkolem.

17.1 Běžné bezpečnostní problémy

Některé níže popsané bezpečnostní chyby jsou všeobecně dobře programátorům známé a snaží se jim vyhýbat, to ale neznamená, že se neobjevují. Zpravidla na ně narazíme, pokud programátor podcenil útočníka nebo si neuvědomil všechny důsledky používání daných technologií.

Útoky typu vložení kódu nebo příkazu (SQL injection, shell injection, …) jsou založené na špatném odfiltrování metaznaků, na stejném principu funguje i cross site scripting, ale s jiným důsledkem. U toho jde u vložení škodlivého kódu html do stránky.

Útoky založené na přetečení bufferu se snaží pozměnit tok programu tak, že změní hodnoty podle kterých se program rozhoduje, co udělá dál. Tento útok využívá nevhodné práce s pamětí.

Pochopení a simulace těchto útoků vyžaduje programátorské znalosti, jako je práce s databázemi nebo jiným systémem, kterému je uživatelův vstup předáván, v případě útoku typu „injection“ nebo znalost principu programovacích jazyků v případě přetečení bufferu. Více na porozumění principu práce s internetem závisí objevení následujících častých chyb.

Pro změnu bezpečnostní chybou je využívání html a http jako ochranného prvku. Jelikož lze html a http kód snadno změnit a požadavky mohou chodit v libovolném pořadí, aplikace tak není skutečně chráněná.

Některé skriptovací jazyky umožňují nastavení proměnných z url, což není-li ošetřeno, dává útočníkovi možnost ovlivňovat běh webové aplikace.

Příklady testů zabezpečení spočívajících na znalosti fungování http, html a url:

```
1)    http://www.poeta.cz/basen/sedmikraska/../lucerny

2)    GET / http/1.0
Host: www.priklad.cz
Accept-Language: ../admin

3)    <input type=”hidden” name=”uziv” value=”h7p5”>
```

Všechny předchozí problémy se týkaly nedostatečného ošetření vstupu a poukazují na první pravidlo, důkladně otestovat je třeba vše, co může být uživatelem změněno.

Velmi častým problémem webových aplikací jsou trojské koně v podobě html. Trojští koně provádějí nežádoucí akce bez souhlasu uživatele, ale jeho jménem. V případě html se po zobrazení inkriminující stránky, provede nějaká akce, typické je odeslání formuláře. Tímto způsobem můžete nevědomky zahlasovat v anketě nebo poslat nadřízenému nepěknou zprávu skrze firemní webovou aplikaci. Protože trojský kůň provádí akce jménem uživatele, jenž si jeho kód v prohlížeči spustil, může provádět vše, k čemu má uživatel oprávnění. To znamená, že úspěch trojských koní často závisí na tom, zda je uživatel právě přihlášen do webové aplikace, na kterou kůň útočí.

Pokud byste si zobrazili webovou stránku následujícího příkladu a zároveň byli přihlášeni do webové aplikace www.poeta.cz, bylo by vašim jménem odesláno nové dílo. Přitom nezáleží na tom, zda si danou aplikaci zrovna prohlížíte nebo zda máte internetový prohlížeč zapnutý, stačí pokud vaše přihlášení je stále platné.

Příklad trojského koně:

```
      <body>
      <form name="formular" action="http://www.poeta.cz/newdilo.php"
method="post">
        <strong>Název</strong><br />
        <input type="text" name="nadpis" value="TEST" /><br /><br />
        <strong>Perex</strong> (povinné pole)<br />
        <textarea name="perex" rows="10" cols="45"> perex
test</textarea><br /><br />
        <strong>Žánr</strong><br />
        <select name="pbo_new">
          <option value="p">Povídky</option>
          <option value="b">Básně</option>
          <option value="o">Ostatní</option>
          <option value="s">Smsky</option>
        </select><br /><br />
        <strong>Text</strong><br />
        <textarea name="obsah" rows="30" cols="45"> TEEEEEEST trojskym
konem kvuli diplomce </textarea><br /><br />
        Neprovádět korekturu (dílo bude rychleji publikováno):
        <input type="checkbox" name="korektury" value="ne" ><br /><br />
        <input type="submit" value="Uložit dílo" />
                
        <input type="reset" value="Vrátit změny" />
      </form>
      <script>document.formular.submit()</script>
      </body>
```

Předchozí kód byl vytvořen velmi jednoduchým způsobem: ze stránky obsahující formulář pro odeslání nového díla se vykopíroval a uložil zdrojový kód, z něj se pak odstranilo vše nepotřebné, až zůstal jen samotný formulář, který se vyplnil vhodnými údaji. Nakonec se přidal řádek odesílající obsah formuláře.

V případě webové aplikace poeta.cz není ohrožení příliš velké, protože uživatel se přihlašuje do aplikace pouze, když chce přidat nové dílo nebo změnit nastavení, navíc jsou všechna díla před zveřejněním schvalována správcem aplikace.

U mnoha aplikací je však zranitelnost větší, zejména jsou-li používány často nebo umožňují-li trvalé přihlášení.

Předchozí příklad ukazoval pouze princip vytvoření html kódu trojského koně, jeho součástí bývá i určité pozlátko, na které jsou uživatelé lákáni a které zajistí větší šanci na nalezení přihlášeného uživatele, nebo přiměje uživatele se do cílové aplikace přihlásit:

Napadá mě situace, kdy bych vytvořila třeba užitečnou stránku k oblíbené hře. Tato stránka by poskytovala po zadání nějakého údaje ze hry hráčům něco, co samotná hra neumí. Například po zadání jména nějakého kmene, by zvýraznila členy daného kmene na herní mapce. Dá se předpokládat, že pokud by hráč třeba uvažoval o přestoupení do jiného kmene, otevřel by naši stránku a zadal pomocí ctrl-c a ctrl-v jméno kmene, aby zkontroloval polohu jeho členů. Často by tato stránka byla tedy navštěvována uživateli, kteří jsou do hry právě přihlášení. Pak bych jako součást této stránky mohla vytvořit rámec zúžený tak, aby jeho obsah nebyl vidět, a do rámce vepsala kód odesílající suroviny konkrétnímu uživateli.

Možnost provedení nějaké akce jménem někoho jiného je dosti nebezpečná, ale protože ochrana proti trojským koním vyžaduje programování navíc, není spousta webových aplikací proti nim chráněna.

Pokud tato ochrana není naplánována a tester se domnívá, že hrozba trojských koní je příliš velká a peníze investované do prevence se vyplatí, může podpořit své hlášení chyby praktickou ukázkou, při které sám vytvoří trojského koně a provede útok. Pochopitelně takový útok by měl být proveden na testovacím prostředí, které je k takovým věcem určené, aby nedošlo k žádné skutečné škodě.

17.2 Dělení bezpečnostních problémů

Pro lepší přehled nebezpečí spojených se sítí následuje několik kategorizací, které třídí bezpečnostní problémy.

Jedním z používaných dělení týkajících se síťové bezpečnosti je Jayramova a Morseova taxonomie. Ta dělí problémy na:

- Fyzické prolomení bezpečnosti, což představuje zcizení systému, jeho části nebo autentizačního prvku.
- Slabé místo softwaru, chyba nebo opomenutí v kódu, které může útočník využít.

- Maligní programy, nahrané na systém za účelem poškození informací uložených v systému.
- Ukradení přístupových práv, ať už prolomením hesla, jeho odchycením nebo obelstění osoby s přístupem pomocí sociálního inženýrství.
- Komunikační problémy spojené například s odchytáváním paketů.

Při třídění do tří tříd podle závažnosti obecně vzniká následující dělení:

- Vysoce nebezpečné jsou všechny problémy, které umožňují okamžitý neoprávněný přístup nebo umožňují pozměnění běhu aplikace či spuštění příkazu.
- Středně nebezpečné problémy jsou problémy které je těžší využít, ale s trochou snahy umožní provedení stejně nežádoucích akcí, jako je tomu u vysoce nebezpečných problémů.
- Málo nebezpečné problémy je nejtěžší využít, většinou slouží pouze k získání informací nebo přípravě na následující útok.

Dělení možných důsledků bezpečnostních děr podle materiálu IBM [25]:

- Získání přístupu
- Odepření služby (denial of service)
- Manipulace s daty
- Získání informací
- Obejití bezpečnostních opatření
- Získání práv
- Manipulace se soubory

17.3 Proces testování bezpečnosti

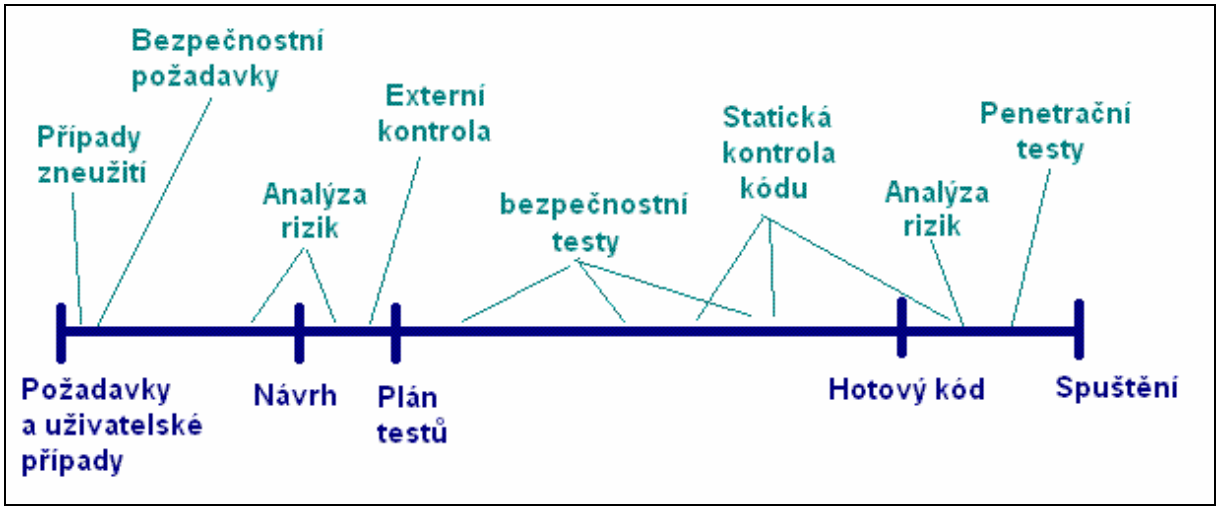
Testování a hlídání bezpečnosti je dlouhotrvající proces, který by měl být řízen na základě rizik, kterým je webová aplikace vystavena. Tento proces nekončí při odevzdání produktu, pouze se dále vyvíjí.

V první fázi zajištění bezpečnosti by pozornost měla být soustředěna na vytvoření kvalitního produktu, který když už selže, tak selže bezpečně a neohrozí bezpečnost dat, které chrání. Toto je dosaženo pomocí kombinace tří přístupů (první 2 převzaty z Potter a kol. [35]):

- Testování bezpečnostních mechanismů s cílem zajistit, že jejich funkcionalita je správně implementována
- White box testing celé aplikace s cílem hledání chyb v kódu nebo nedostatků v designu, které by mohly mít dopad na bezpečnost
- Provedení na riziku založeném bezpečnostním testování prováděném z pohledu útočníka

Aplikace těchto přístupů vyžaduje nějaký čas a provádění k tomu vyškolenými testery. Potter a kol. uvádí v [35] následující úkoly, jenž by měly být součástí procesu testování bezpečnosti a jejich návaznost na životní cyklus vývoje, která je v upravené podobě zachycena na obr. 11:

- Vytvoření možných případů zneužití bezpečnosti
- Vytvoření seznamu bezpečnostních požadavků
- Provedení rizikové analýzy architektury
- Zajištění a využívání nástrojů na statickou analýzu kódu
- Provedení bezpečnostních testů
- Provedení penetračního testování na konečném prostředí



Obr. 20: Dělení chyb podle komponenty (inspirace z [35])

V druhé fázi po spuštění webové aplikace v rámci správy by se mělo provádět skenování sítě, procházení logů, zda nezaznamenaly něco podezřelého, updatování softwaru, je-li vydána oprava nějaké chyby, a další činnosti, které úzce souvisí s bezpečností, ale nejsou prováděny testery. Ti provádějí jednorázové externí bezpečnostní testy zejména penetrační na požadavek managmentu. Výsledkem je pak zpráva o prověření celkového zajištění bezpečnosti včetně reakcí personálu na požadování citlivých informací neautorizovanými osobami.

17.4 Nástroje

Na internetu existuje mnoho stránek věnujících se přímo bezpečnosti, většinou jsou plné návodů na proniknutí do aplikace, názorů bezpečnostních expertů, a rad, jak testovat a zabezpečit aplikaci. Některé, zvláště ty rozsáhlejší weby, často přímo obsahují nebo aspoň zmiňují užitečné nástroje.

Za zmínku rozhodně stojí webgoat, záměrně špatně zabezpečená webová aplikace, která po nainstalování nabízí možnost si vyzkoušet otestování a proniknutí za pomocí nejrůznějších útoků ve zhruba třiceti lekcích. Tuto výukovou aplikaci napsanou v jazyce java najdete na stránkách www.owasp.org a v případě, že by některé lekce byly pro uživatele příliš těžké, je možné zde najít i odkaz na videa s řešením.

Web www.owasp.org [42] nabízí ke stažení i další zajímavý nástroj webscarab, který je opět napsaný v jazyce java a funguje jako mezičlánek mezi prohlížečem a webovým serverem zachytávající veškerou komunikaci a poskytující možnost modifikace dotazů a odpovědí.

Část IV. – Praktický příklad

18. Příklad testovací plánu

Autor: Anna Borovcová
Datum: 22.7.2008

1) Testovaná aplikace

Popis:

Předmětem testování je FileSearch New Generation (zkráceně FSNG), aplikace na vyhledávání a stahování souborů určené pro středně velké sítě FTP serverů. Jedná se o složitější softwarový systém, který využívá jako uživatelské rozhraní tenkého klienta.

Základní dvě konfigurace jsou:

a) Instalace produktu jako běžné webové aplikace umožňující uživatelům vyhledávání přes prohlížeč po zadání internetové adresy, kde aplikace běží.
b) Uživatel si instaluje aplikaci na svůj počítač a je mu umožněno i stahování. V tomto případě uživatel k aplikaci přistupuje přes prohlížeč, ale aplikace běží na localhostu.

Aplikace je napsána v Javě.

Související dokumentace:

Zdrojové soubory se nachází v svn na cestě projekty/FSNG/project. Další dokumentace se nachází v svn na cestě projekty/FSNG/dokumentace a obsahuje následující soubory:

Analýza požadavků FSNG
Konfigurace a přístupová práva FSNG
Návrh architektury FNSG
Popis designu FSNG
Problematika search enginu
Specifikace FSNG
Stručné uvedení do problematiky
Uživatelská dokumentace FSNG

2) Cíl testování

Úkoly testovacího týmu:

Testování proběhne jako součást přípravy produktu na předání jinému vývojovému týmu, který bude mít za úkol dokončit FSNG zejména po stránce uživatelské přívětivosti, doladit jej a splnit dříve stanovené akceptační kritéria.

Předchozí testování v průběhu vývoje nebylo řízeno a neexistují nebo nejsou dostupné jeho výsledky.

Cíle testování budou následující:

- Ohodnotit kvalitu dokumentace a připravenost produktu na předání
- Ohodnotit stav jednotlivých komponent
- Připravit testovací scénáře na základě uživatelské aktivity během experimentálního zpřístupnění aplikace.

Rozsah testování:

Otestovány budou všechny komponenty, není ale třeba testovat do hloubky, stačí základní funkčnost komponent.

3) Testovací přístup

Pro odhad celkového stavu a kvality komponent budou použity automatické a manuální funkční testy doplněné o test stability aplikace a několik stress testů. Pro automatizované funkční testy bude použito jednotkové testování s pomocí open source nástroje JUnit. Manuální funkční testování bude jak průzkumné, tak podle scénářů typu Happy day, které budou vytvořeny na základě uživatelské aktivity.

Test stability bude proveden tak, že aplikace bude spuštěna nepřetržitě po dvanáct hodin, během čehož budou maximálně využívány všechny její možnosti. Stress testy se budou soustředit na spolehlivost aplikace při omezení nebo odebrání zdrojů.

Kvalita dokumentace a připravenost produktu bude otestována pomocí upravené verze nástroje Code Quality Plugin a pluginu Metrics, a dále testeři projdou veškerou dokumentaci ve snaze ohodnotit její kvalitu.

.

4) Kritéria ne/připravenosti k předání

Aplikace nebude moci být předána, dokud bude splněna některá z následujících podmínek:

- Jakýkoli scénář typu happy day vždy selže.
- Existuje třída nebo metoda, jejíž funkčnost není okomentovaná.

5) Zdroje

Testování bude probíhat po 3 dny od 24.7. do 26.7., výsledky budou k dispozici vedoucímu projektu do devíti hodin 27.7. Po tu dobu bude projekt využívat následující zdroje:

Počítače:

3 počítače s minimální hardwarové konfiguraci:
Procesor: 1.5 GHz Intel(R) Pentium(R) 4 nebo ekvivalentní
Paměť: 1 GB RAM
Volné místo na disku: 4GB
Síťové připojení 1MBit/s

Nainstalovaný software:
Java JDK 1.6.x
Eclipse 3.1 (vyšší verze je nekompatibilní s požadovanými pluginy)
JUnit
Metrics 1.3.6
Code Quality Plugin – 0.1.5.b
Mozilla Firefox 2
Opera 9.4
Kancelářský balík obsahující tabulkový a textový procesor
Z toho dva počítače by měli mít libovolný vhodný operační systém Windows a prohlížeč Internet Explorer 6, na zbylém počítači bude vhodný unixový operační systém.

6) Role

Potřební lidé:

Senior tester 01 – zkušenost s programováním nebo procházením kódu

Senior tester 02 – zkušenost JUnit testy a testováním spolehlivosti

Junior tester

Zodpovědnost:

Funkční manuální testy – Junior tester

Jednotkové testy – Senior tester 02

Testy spolehlivosti – Senior tester 02

Testy dokumentace – Senior tester 01

Rozdělení prací:

Test	**Odhadovaný počet hodin**	**Role**
Průzkumné funkční	5 h	Junior tester
Příprava a provedení scénářů	11 h	Junior tester
Jednotkové	12 h	Senior tester 02
Příprava testu stability	3 h	Senior tester 02
Provedení testu stability	8 h	Senior tester 02
Řízení stress testu	1 h	Senior tester 02
Provedení stress testů	8 h	Junior tester
Testy dokumentace	24h	Senior tester 03

19. Příklad testovacího scénáře

Testovací scénář 1

Zkratka:TC01

Popis: Nejčastější scénář – vyhledání a stažení

Testovací případ 02:

Počáteční bod: Aplikace je vypnutá.

1) Spuštění souboru fsng.jar. Nejedná se o úplně první spuštění.

Jelikož nastavení a instalace probíhá jen při prvním spuštění, uživatel není na nic dotázán. V oznamovací oblasti spodního panelu na obrazovce, což je většinou vpravo dole se objeví ikonka s popisem FSNG tooltip. Automaticky se otevře okno defaultního prohlížeče na stránce s uživatelským rozhraním. Je zobrazena záložka vyhledávání.

Testovací případ 06:

Počáteční bod: Aplikace je zapnutá, je zobrazen obsah záložky Vyhledávání.

1) Do políčka Název zadejte slovo „Barbie“ a typ položky vyberte film. Klikněte na tlačítko vyhledat.

Zobrazí se výsledky vyhledávání se 4 filmy, všechny obsahují někde podřetězec Barbie nebo barbie.

2) U některého z výsledku vyhledávání klikněte na červené tlačítko s bílým znakem plus.

Pod danou položkou výsledku se zobrazí informace, kde se vyhledaná položka nachází a o podobných souborech nebo ve stejném adresáři.

Testovací případ 14:

Počáteční bod: Jsou zobrazeny podrobnosti alespoň o jednom z výsledků vyhledávání.

1) Stiskněte tlačítko pro stažení všech podobných souborů.

Zobrazí se dialogové okno pro vybrání umístění v počítači, kam se mají položky stáhnout.

2) Klikněte na jedno z dříve uložených umístění.

Předvyplní se cesta, kam stáhnout vyhledané soubory.

3) Stiskněte tlačítko Stáhnout.

Uživatel je zpátky na záložce Vyhledávání, ale v pravé části s přehledy pod Stavem stahování se objeví informace o tom, kolik z nastavených položek je staženo.

4) Přepněte se do záložky Stahování.

Objeví se seznam jednotlivých souborů a informace o průběhu stahování.

Po nějaké době se všechny položky úspěšně stáhnou.

20. Příklad test result dokumentu

Autor: Anna Borovcová

Datum: 26.7.2008

1) Souhrnné zjištění

Cíle testování byly následující:

a) Ohodnotit kvalitu dokumentace
b) Ohodnotit stav jednotlivých komponent
c) Připravit testovací scénáře na základě uživatelské aktivity během experimentálního zpřístupnění aplikace.

Tříčlenný tým strávil testováním přidělené tři dny, v rámci čehož se jim podařilo splnit všechny určené cíle. Žádná z podmínek pro zastavení předání FSNG aplikace novému týmu nebyla splněna.

Celkem bylo reportováno ? chyb, z nichž ? je ještě otevřených, ostatní již byly uzavřeny.

Reporty chyb jsou uložené v bugzille JIRA, popsané stavy chyb jsou z 26.7. 18:00.

Chyby mohou nabývat těchto priorit a stavů:

Priority:

A – nezbytné okamžitě opravit, musí být uzavřeno před dokončením testů

B – velký problém, ale neblokuje předání

C – drobnější problém

Stavy:

Nová, přidělená, opravená, vyřešená, odložená, zrušená.

Pokud je chyba v jednom ze stavů vyřešená, odložená nebo zrušená, jedná s o koncový stav, je tedy uzavřená.

2) Testy dokumentace

Uživatelská část	OK
Technická část	**X**
Zdrojové kódy	**X**

V uživatelské části dokumentace nebyl nalezen žádný problém.
Technická část dokumentace a komentáře zdrojového kódu byly poněkud nevyvážené.
Bylo nalezeno a nahlášeno 8 chyb:

ID chyby	Shrnutí	Priorita	Stav
BUG 01	Chybějící popis admin konzole	B	Odložená
BUG 03	Rozpor v dokumentaci	B	Vyřešená
BUG 06	Chybějící údaje	C	Vyřešená
BUG 07	Drobný rozpor v dokumentaci	C	Vyřešená
BUG 12	Nedostatečná tech. dokumentace klienta	B	Přidělená
BUG 15	Chybějící komentář	A	Vyřešená
BUG 16	Chybějící komentář	A	Vyřešená
BUG 19	Chybějící komentář	A	Vyřešená

3) Odhad stavu komponent

Uživatelské rozhraní:

Uživatelské rozhraní zjevně není doladěné a dokončené. Hotovo je zhruba ze 70%, tedy 30% testů selže. Chybí zejména práce s adresáři, doladění vzhledu ve více prohlížečích, správné umístění prvků na stránce.
Během funkčních testů bylo nalezeno a nahlášeno 28 chyb, z toho žádná a prioritou A, 11 s prioritou B a 17 C. Chyby nebyly zatím řešeny. Jejich seznam bude předán druhému vývojovému týmu.

Modul crawler:

Modul je hotový zhruba z 80%. Obsahuje závažné nedostatky, zejména je chybový a není optimalizovaný.
K modulu crawler bylo nalezeno a nahlášeno 8 chyb, jedna už vyřešená s prioritou A, 4 s prioritou B a 3 C. Jejich seznam bude předán druhému vývojovému týmu.

Modul dowloader:

Modul je hotový asi z 90%. Neobsahuje žádné větší nedostatky.
U modulu byla nalezena jedna chyba priority B a 2 C.

Modul datastore:

Modul je zřejmě hotový, všechny zvolené testovací případy skončili úspěšně. Šlo ale jen o hrubé testování s cílem odhadnout stav komponenty. Je pravděpodobné, že při podrobnějším testování, se můžou chyby objevit.

Modul vybavovač požadavků:

Modul je téměř hotový, úspěšně skončilo 90% testů, byla nalezena 1 chyba priority B.

Jádro:

Jádro je opět téměř hotové, byla nalezena 1 chyba priority B a 1 C, což představuje 15% neúspěšných a 85% úspěšných testů.

4) Příprava testovacích scénářů

Bylo připraveno 13 testovacích scénářů, které tvoří 72 případů.
Scénáře a případy je možné najít v svn pod projekty/FSNG/dokumentace/ test_scenario a projekty/FSNG/dokumentace/ test_case.

5) Zhodnocení použitého přístupu

Zvolený průběh testování se s ohledem na vybrané cíle osvědčil. Časový odhad byl realistický. Jako nedostatek se ale ukázalo nestanovení metrik pro měření připravenosti komponent. Odhad je tak postaven na názoru senior testera a vybrané množině testovacích případů.

Závěr

Tato práce si kladla mnoho cílů od revize současného stavu testovacího procesu po seznámení s automatizací testování pomocí nástrojů. Hlavním výsledkem je ale vznik materiálu, s jehož pomocí můžou být vyškoleni budoucí testeři.

Zatímco já jsem připravovala budoucí testery za pomoci řady komerčních školení trvajících dohromady několik týdnů, tento materiál poskytuje dostatečný úvod do testování během pár hodin. Spolu s doprovodnými školeními uzpůsobenými konkrétnímu účelu školení pak tester může být připraven během pár dní.

Vynechání přílišného zaměření na konkrétní metodiku nebo nástroj, což je častý neduh komerčních školení, a obsáhnutí podstaty testování umožňuje revidovat systém školení. Jako za adekvátní tedy navrhuji následující systém výuky:

Obecná část:
Testování webových aplikací (tato diplomová práce) – materiál k samostudiu
Úvod do xml a html – 10 hodin
Úvod do sql – 1 den
Základy programování – 1 den
Vyzkoušení naučeného v praxi – 1 den

Speciální dle dalšího zaměření testera a přidělení na projekt:
Zaškolení do konkrétního testovacího nástroje - 2 dny
Zaškolení do konkrétní testované aplikace – 1-2 dny

Seznam obrázků

Obr. 1: Vodopádový model vývoje podle [13] 17
Obr. 2: Model životního cyklu vývoje podle metodiky WebWave 23
Obr. 3: Příčiny chyb podle Pattona [10] 29
Obr. 4: Druhy chyb podle [20] 30
Obr. 5: Vztah nákladů a testů podle [10] 31
Obr. 6: Mediány platů podle zkušenosti v letech na pozici softwarový inženýr/ vývojář / programátor v USA [33] 40
Obr. 7: Mediány platů podle zkušenosti v letech na pozici test / quality assurance inženýr v USA [34] 40
Obr. 8: Mediány platů podle zkušenosti v letech na pozice vývojáře, vývojáře webových aplikací a testera [22-24] 43
Obr. 9: Životní cyklus hlášení chyby 62
Obr. 10: Dělení chyb podle komponenty 66
Obr. 11: Výsledek testu 86
Obr. 12: Výsledek testu 87
Obr. 13: Výsledek JWebUnit testu 91
Obr. 14: Výsledek testu v FT 95
Obr. 15: JMeter po zapnutí 101
Obr. 16: Ovládání přes kontextové menu 102
Obr. 17: Výsledky v Aggregate Graf Listener 105
Obr. 18: Testovací skript LoadRunneru v podobě kódu 110
Obr. 19: Výsledky v Controlleru 111
Obr. 20: Dělení chyb podle komponenty (inspirace z [35]) 119

Zdroje

Literatura

[1] Beck, Kent: *Extrémní programování*, Grada Publishing, Praha, 2002

[2] Hutcheson, Marnie L.: *Software Testing Fundamentals: Methods and Metrics*, John Wiley & Sons, 2003

[3] IBM, Rational software: *TST170 Principles of Software Testing for Testers*, v 2002.05.00, Instructor Guide

[4] Kadlec, Václav: *Agilní programování: Metodiky efektivního vývoje softwaru*, Computer Press, Brno, 2004

[5] Kunský, František: *Testování software a testovací role*, diplomová práce, VŠE, 2005

[6] Microsoft ACE Team: *Výkonnostní testování webových aplikací .NET* , Grada, 2004

[7] Patton, Ron: *Testování softwaru*, Computer Press, Brno, 2002

[8] Pavelka, Jan: *Zajištění jakosti projektů*, materiály k předmětu Softwarové inženýrství (sweng56.ppt), KSI MFF UK, 2005

[9] Pouzar, Lukáš: *Automatizované testování*,bakalářská práce, VŠE, 2007

[10] Weinberg, Gerald. M.: *Quality Software Management: Volume 1, Systems Thinking*, Dorset House Publishing Company, Incorporated, 1991

[11] Unicorn: *General test ideas,* kód artefaktu: USO.UES.TES/UES.TES.GTI, 2006

Zdroje z Internetu

[12] Abran, Alain; Moore, James W.: *SWEBOK, Guide to the Software Engineering Body of Knowledge*, IEEE Computer Society, 2004 Version
Dostupné z WWW:
www.inf.ed.ac.uk/teaching/courses/seoc/2006_2007/resources/SWEBOK_Guide_2004.pdf
Datum čerpání zdroje: 10.3.2008

[13] Als, Adrian; Greenidge, Charles: *The waterfall model,* 2005
Dostupné z WWW:
http://scitec.uwichill.edu.bb/cmp/online/cs22l/waterfall_model.htm
Datum čerpání zdroje: 7.2.2008

[14] Andrews, Mike: *How to break web software,* Google TechTalks, 2006
Dostupné z WWW:
http://video.google.com/videoplay?docid=5159636580663884360
Datum čerpání zdroje: 10. 11. 2007

[15] Aston, Philip; Fitzgerald, Calum: *The Grinder*
Dostupné z WWW:
http://grinder.sourceforge.net
Datum čerpání zdroje: 24.7.2008

[16] Bach, James; Bolton, Michael: *Rapid Software Testing,* v2.1.3, Satisfice, Inc., 2007
Dostupné z WWW:
http://www.satisfice.com/rst.pdf
Datum čerpání zdroje: 9. 4. 2008

[17] Canoo Engineering AG: *Canoo WebTest*
Dostupné z WWW:
http://webtest.canoo.com/
Datum čerpání zdroje: 2.7.2008

[18] Cornett, Steve: *Code Coverage Analysis*, Bullseye Testing Technology

Dostupné z WWW:
http://www.bullseye.com/coverage.html
Datum čerpání zdroje: 3.4.2008

[19] Dijkstra, Edsger W.: *The Humble Programmer*,
Dostupné z WWW:
http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD340.PDF
Datum čerpání zdroje: 25.2.2008

[20] Faigl, Jan: *Úvod do softwarového inženýrství,* (lecture11), ČVUT
Dostupné z WWW:
http://lynx1.felk.cvut.cz/mep/files/slides/lecture11.pdf
Datum čerpání zdroje: 1.4.2008

[21] gantthead.com: *Process/Project WWP - webWAVE Development Process*
Dostupné z WWW:
http://www.gantthead.com/process/processMain.cfm?ID=2-8435-2
Datum čerpání zdroje: 3.3.2008

[22] Hailpern, Brent T.; Santhanam, Padmanabhan: *Software debugging, testing and verification*. IBM Systems Journal, 2002, roč. 41, č. 1.
Dostupné z WWW:
http://www.research. ibm.com/journal/sj/411/hailpern.pdf
Datum čerpání zdroje: 9.3.2008

[23] Churý, Lukáš: *Z černého trhu na trh volný,* programujte.com, 2005
Dostupné z WWW:
http://programujte.com/index.php?akce=clanek&cl=2005102101-z-cerneho-trhu-na-trh-volny
Datum čerpání zdroje: 7.11.2007

[24] IBM: *Rational Functional Tester*
Dostupné z WWW:
http://www-306.ibm.com/software/awdtools/tester/functional/index.html

Datum čerpání zdroje: 8.7.2008

[25] IBM Internet Security Systems: *X-Force® 2007 Trend Statistics*, IBM Global Services, Leden 2008
Dostupné z WWW:
http://www-935.ibm.com/services/us/iss/pdf/etr_xforce-2007-annual-report.pdf
Datum čerpání zdroje: 27.5.2008

[26] Jakarta-jmeter Wiki editors: *HowManyThreads*
Dostupné z WWW:
http://wiki.apache.org/jakarta-jmeter/HowManyThreads
Datum čerpání zdroje: 20.7.2008

[27] Jakarta-jmeter Wiki editors: *JMeterFAQ*
Dostupné z WWW:
http://wiki.apache.org/jakarta-jmeter/JMeterFAQ
Datum čerpání zdroje: 20.7.2008

[28] Larman, Craig; Basili, Victor R.: *Iterative and Incremental Development: A Brief History*
Dostupné z WWW:
http://www2.umassd.edu/SWPI/xp/articles/r6047.pdf
Datum čerpání zdroje: 4.3.2008

[29] Leveson, Nancy: *Medical Devices: The Therac-25*, 1995
Dostupné z WWW:
http://sunnyday.mit.edu/papers/therac.pdf
Datum čerpání zdroje: 18.3.2008

[30] Meier, J.D.; Farre, Carlos a kol: P*erformance Testing Guidance for Web Applications*, patterns & practices, Microsoft Corporation, 2007
Dostupné z WWW:
http://msdn.microsoft.com/en-us/library/bb924375.aspx
Datum čerpání zdroje: 18.7.2008

[31] Pavlíčková, Jarmila; Pavlíček, Luboš: *Nástroje na zvyšování kvality kódu*
Dostupné z WWW:
http://honor.fi.muni.cz/tsw/2004/226.pdf
Datum čerpání zdroje: 8.6.2008

[32] PayScale: *Salary Survey Report for Job: Software Developer, Web Applications*
Dostupné z WWW:
http://www.payscale.com/research/US/Job=Software_Developer%2c_Web_Applications/Salary
Datum čerpání zdroje: 25. 3. 2008

[33] PayScale: *Salary Survey Report for Job: Software Engineer / Developer / Programmer*
Dostupné z WWW:
http://www.payscale.com/research/US/Job=Software_Engineer_%2f_Developer_%2f_Programmer/Salary
Datum čerpání zdroje: 25. 3. 2008

[34] PayScale: *Salary Survey Report for Job: Test / Quality Assurance (QA) Engineer (Computer Software)*
Dostupné z WWW:
http://www.payscale.com/research/US/Job=Test_%2f_Quality_Assurance_(QA)_Engineer_(Computer_Software)/Salary
Datum čerpání zdroje: 25. 3. 2008

[35] Potter, Bruce; McGraw, Gary: *Software Security Testing*, THE IEEE Computer Society, září/říjen 2004
Dostupné z WWW:
http://www.cigital.com/papers/download/bsi4-testing.pdf
Datum čerpání zdroje: 6.5.2008

[36] Samurin, Alex: *Orthogonal Array Testing*, leden 2006
Dostupné z WWW:
http://www.geocities.com/xtremetesting/OrthogonalArrayTesting.html

Datum čerpání zdroje: 5.4.2008

[37] Schooff, Peter: *Does the Security Industry Have a Future?*
Dostupné z WWW:
http://www.schneier.com/news-060.html
Datum čerpání zdroje: 20.5.2008

[38] Simo, Ben: *Testing Lessons from Beer*
Dostupné z WWW:
http://testertested.qualityfrog.com/TLFB.mp3
Datum čerpání zdroje: 23.3.2008

[39] Soundararajan, Pradeep: *Automation replaces humans - The truth about what kind of humans it replaces*
Dostupné z WWW:
http://testertested.blogspot.com/2008/03/automation-replaces-humans-truth-about.html
Datum čerpání zdroje: 1. 5. 2008

[40] Soundararajan, Pradeep: *The most challenging software testing quiz*
Dostupné z WWW:
http://testertested.qualityfrog.com/TMCSTQ.swf
Datum čerpání zdroje: 9. 12. 2007

[41] SourceForge: *JWebUnit*
Dostupné z WWW:
http://jwebunit.sourceforge.net
Datum čerpání zdroje: 7.7.2008

[42] The Open Web Application Security Project
Dostupné z WWW:
http://www.owasp.org
Datum čerpání zdroje: 7.6.2008

[43] Vijayaraghavan, Giri; Kaner, Cem: *Bug Taxonomies: Use Them to Generate Better Tests*, STAR EAST 2003
Dostupné z WWW:
http://testingeducation.org/a/bugtax.pdf
Datum čerpání zdroje: 20.5.2008

[44] Ward, Stan; Kroll, Per: *Building Web Solutions with the Rational Unified Process: Unifying the Creative Design Process and the Software Engineering Process*, Context Integration, Rational Software, 1999
Dostupné z WWW:
http://www.dcc.uchile.cl/~luguerre/cc61j/recursos/76.pdf
Datum čerpání zdroje: 20.3.2008

[45] Wikipedia contributors: *Mars Polar Lander*, Wikipedia, The Free Encyclopedia
Dostupné z WWW:
http://en.wikipedia.org/wiki/Mars_Polar_Lander
Datum čerpání zdroje: 22. 3. 2008

[46] Wikipedia contributors: *MIM-104 Patriot*, Wikipedia, The Free Encyclopedia
Dostupné z WWW:
http://en.wikipedia.org/wiki/MIM-104_Patriot#Failure_at_Dhahran
Datum čerpání zdroje: 22. 3. 2008

[47] Wikipedia contributors: *Web application*, Wikipedia, The Free Encyclopedia
Dostupné z WWW:
http://en.wikipedia.org/wiki/Web_application
Datum čerpání zdroje: 12.11.2007

Přílohy

Příloha 1 – Agilní manifest (http://agilemanifesto.org, datum čerpání zdroje: 23.2.2008)

Manifesto for Agile Software Development

We are uncovering better ways of developing
software by doing it and helping others do it.
Through this work we have come to value:

Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan

That is, while there is value in the items on
the right, we value the items on the left more.

Kent Beck
Mike Beedle
Arie van Bennekum
Alistair Cockburn
Ward Cunningham
Martin Fowler

James Grenning
Jim Highsmith
Andrew Hunt
Ron Jeffries
Jon Kern
Brian Marick

Robert C. Martin
Steve Mellor

Ken Schwaber
Jeff Sutherland
Dave Thomas