Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačního inženýrství

Diplomová práce

Automatizované testování softwaru

Simon Sloup

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Katedra informačního inženýrství

Provozně ekonomická fakulta

ZADÁNÍ DIPLOMOVÉ PRÁCE

Simon Sloup

Informatika

Název práce

Automatizované testování softwaru

Název anglicky

Automated software testing

Cíle práce

Diplomová práce je tematicky zaměřena na problematiku tzv. testování softwaru, obecněji zajišťování jakosti softwaru, někdy také nazývané QA (Quality Assurance). Hlavním cílem je popis celého procesu této problematiky při analýze vybraných informačních zdrojů z různých možných úhlů použití.
V praktické části bude následně řešeno testování softwaru na konkrétním příkladu a to pomocí automatizace i manuálně. Oba případy budou porovnány. Hlavní cílem této části je tedy zjištění, v jakém případě se vyplatí automatizovat testování a zda může nahradit testování manuální.

Metodika

Metodika řešené problematiky práce je založena především na analýze a studiu vybraných odborných informačních zdrojů a literatury, zabývajících se testováním a vývojem softwaru. Metodika práce spočívá v sumarizaci, analýze, syntéze informací z odborné literatury. Praktická část je založena na popisu daného podniku a analýze vybraného produktu, na který bude následně implementován manuální i automatizovaný testovací případ. Na základě analýzy informačních zdrojů a řešení praktické části budou formulovány závěry diplomové práce.

Doporučený rozsah práce

60 – 80 stran

Klíčová slova

testování softwaru, automatizace, QA, jakost softwaru, Selenium

---

Doporučené zdroje informací

BURNS, David. Selenium 2 Testing Tools: Beginner's Guide. 2. vydání. Birmingham: Packt Publishing, 2012. ISBN 1849518300

CRAIG, Rick D. a Stefan P. Jaskiel. Systematic Software Testing. 1. vydání. Londýn: Artech House, 2002. ISBN 9781580535083

KADLEC, Václav. Agilní programování: metodiky efektivního vývoje softwaru. 1. vydání. Brno: Computer Press, 2004. ISBN 80-251-0342-0

KANER, Cem a spol. Testing Computer Software. 2. vydání. New York: John Wiley and Sons, 1999. ISBN 0471358460

PATTON, Ron. Testování softwaru. 1. vydání. Brno: Computer Press, 2002. ISBN 80-7226-636-5

SCHAEFER, Hans a spol. Software Testing Foundations. 4. vydání. Santa Barbara: Rocky Nook, 2014. ISBN 9781937538422

Předběžný termín obhajoby

2015/06 (červen)

Vedoucí práce

Ing. Jiří Brožek, Ph.D.

Elektronicky schváleno dne 29. 1. 2014

Ing. Martin Pelikán, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 11. 3. 2015

Ing. Martin Pelikán, Ph.D.

Děkan

V Praze dne 14. 03. 2015

Čestné prohlášení

Prohlašuji, že svou diplomovou práci "Automatizované testování softwaru" jsem vypracoval samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu literatury na konci práce. Jako autor uvedené diplomové práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne _____

Poděkování

Rád bych touto cestou poděkoval Ing. Jiřímu Brožkovi Ph.D. za jeho ochotu, na kterou jsem se mohl vždy spolehnout a velmi cenné rady, jimiž mi pomohl k vypracování této diplomové práce. Dále bych chtěl poděkovat Ing. Davidu Kristovi za odborné rady, kterými mi pomohl objasnit některé procesní nesrovnalosti a hlavně také společnosti Socialbakers a.s. za přátelskou atmosféru, ve které jsem pracoval během psaní této práce.

Automatizované testování softwaru

Automated software testing

Souhrn

Tato práce se zabývá testováním softwaru jako celku s tím, že důraz je zde kladen i na ne tolik rozšířenou možnost testování – automatizace testování softwaru. Jedná se o disciplínu, která je rozhodně nedílnou součástí vývoje každého softwaru a to již desítky let zpět do minulosti. Nicméně i přes svou historickou hloubku je stále mnohými brána jako druhořadá činnost. To dělá tuto disciplínu o to těžší. Člověk, který v tomto oboru pracuje, musí vše dopodrobna znát a musí umět argumentovat, proč je tato práce v rámci vývoje softwaru zásadní a dosáhnout tak tedy toho, aby ostatní oddělení práci testerů respektovala a brala jako pomocnou ruku, ne jako soupeře. Tato práce si dává za úkol definovat základní charakteristiky testování, tedy popsat testovací proces, rozdělení testů, popis jednotlivých rolí, a ukázat možnosti automatizace.

Klíčová slova: testování softwaru, automatizace, QA, jakost softwaru, Selenium

Summary

This thesis deals with software testing discipline as a whole with focus on software testing automation which is not a widespread part of software testing. Software testing as a discipline is an important part of software development and its history is very long. Even with this historical depth many people think it is still a minor part of software development. That makes this discipline all the more difficult. People working in software testing need to know every detail of this discipline and they need to be able to argue why testing is so important, because it is necessary for other departments to respect testers and see them as a helping hand, not as an enemy. This thesis aims to define fundamental characteristics of testing which requires describing the testing process, the distribution of tests, and the description of individual roles, and to show the possibility of automation.

Keywords: software testing, automation, QA, software quality, Selenium

Obsah

1 Úvod

Pro vznik informačních systémů včetně jejich jádra, kterým je software, je dnes typické to, že jejich vývoj se stává čím dál dynamičtější. Informační systémy dnes neslouží pouze jako podpora firemních procesů, ale často také jako ochrana proti potenciálním hrozbám, případně mohou zvýšit konkurenční výhodu či vylepšovat vztah se zákazníky a tak být oboustranně prospěšnými. Je tedy zcela pochopitelné, že požadavky na vyvíjené informační systémy prudce rostou. V minulosti se dělo to, že se testery stávali samotní zákazníci, kteří byli první, kdo produkt vyzkoušel. Po několika obrovských problémech se testování stalo nedílnou součástí vývoje softwaru. Obecně si lze uvést speciální případy, kde by se v případě selhání mohlo jednat o velký problém. Například zdravotnický software nebo software v elektrárnách, kde by případné problémy mohly znamenat ztrátu lidských životů. Dalším zvláštním případem jsou informační systémy pro banky a jiné finanční společnosti, kde mají klienti uložené své peníze a tak je potřeba vynaložit nemalé úsilí v ochraně zákazníků před stále rostoucí hrozbou počítačové kriminality. Nejen z těchto důvodů je podstatnou součástí při vývoji informačních systémů měření a zlepšování kvality softwaru a jeho testování.

Díky testování lze zjistit, zda software obsahuje všechny uživatelské a technické požadavky. Nedostatečné testování může způsobit vážné problémy. Příčinou pro to může být špatně definovaný či mylně pochopený testovací proces. Případně to může být opomenutí některého typu testu nebo uvažování o některém druhu testu jako o podřadném.

Testování softwaru je disciplína, která je nedílnou součástí vývoje softwaru a přesto se často upozaďuje. Tento problém není způsoben jen neznalostí veřejnosti o tom, jak se software vyvíjí. Jde hlavně o problém vnitřní, kdy je tato podstatná část všech projektů, kde se tvoří software, opomíjena vývojáři i ostatními členy týmu. To vše je vážným důvodem pro co největší důraz na znalosti o této problematice. O testování softwaru je potřeba vzdělávat všechny účastníky softwarového vývoje. Musejí si uvědomit pomocnou sílu, kterou jim může tato disciplína poskytnout. Dále je potřeba vzdělávat

i samotné testery, aby uměli vyargumentovat svojí pozici v rámci týmu a získali si tak potřebný respekt a přijmutí od ostatních členů.

Téma jsem si vybral z toho důvodu, že se již nějakou dobu prakticky touto problematikou zabývám. Práce by měla být přidanou hodnotou pro každého, kdo chce do této disciplíny nahlédnout ať už jako úplný začátečník, případně jako zkušenější manuální tester, který má zájem si ukotvit pojmy z tohoto podoboru a chce nahlédnout "pod pokličku" automatizovaným testům.

2 Cíl práce a metodika

2.1 Cíl práce

Diplomová práce je tematicky zaměřena na problematiku tzv. testování softwaru, obecněji zajišťování jakosti softwaru, někdy také nazývané QA (Quality Assurance). Hlavním cílem je popis celého procesu této problematiky při analýze vybraných informačních zdrojů z různých možných úhlů použití. V praktické části bude následně řešeno testování softwaru na konkrétním příkladu a to pomocí automatizace i manuálně. Oba případy budou porovnány. Hlavní cílem této části je tedy zjištění, v jakém případě se vyplatí automatizovat testování a zda může nahradit testování manuální.

2.2 Metodika

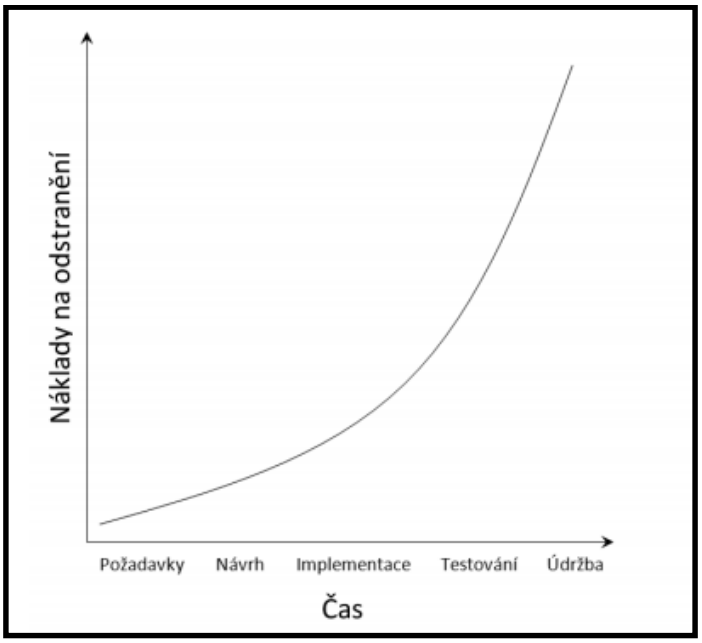
Metodika řešené problematiky práce je založena především na analýze a studiu vybraných odborných informačních zdrojů a literatury, zabývajících se testováním a vývojem softwaru. Metodika práce spočívá v sumarizaci, analýze, syntéze informací z odborné literatury. Praktická část je založena na popisu daného podniku a analýze vybraného produktu, na který bude následně implementován manuální i automatizovaný testovací případ. Na základě analýzy informačních zdrojů a řešení praktické části budou formulovány závěry diplomové práce.

3 Teoretická východiska

3.1 Proč testovat software?

Softwarové systémy jsou stále se rozšiřující součástí našich životů. Jedná se o obchodní aplikace, např. v bankovnictví, výukové či různé školní systémy a třeba i spotřebitelské produkty, jako jsou například auta. Někteří lidé mají jistě zkušenosti s tím, kdy nějaký softwarový systém nepracoval správně. A to může v důsledku znamenat spoustu problémů, od ztráty peněz až po ztrátu reputace společnosti. V krajních případech může chyba způsobit zranění nebo smrt (Müller, 2007).

Obrázek č. 1: Vztah mezi fází detekování chyby a cenou za odstranění



Zdroj: Řízení kvality softwaru (Roudenský, 2013)

Obrázek číslo 1 ukazuje přehledně nekompromisní důvod, proč je potřeba testovat software. Díky testování, které započne co nejdříve, může být nalezen defekt o mnoho dříve, než by se stalo bez této aktivity. Pokud se nalezne chyba ve fázi specifikace požadavků, tak náklady na její odstranění jsou téměř nulové. Nalezení té samé chyby po nasazení může však způsobit velké ztráty.

3.2 Příčiny defektů

Je přirozené, že se člověk může dopustit omylu a ten následně vyvolá defekt (chybu, bug) v kódu, softwaru, systému, dokumentu. Pokud se takový kód, který obsahuje chybu, vykoná, systém potom nemusí udělat, co by měl (respektive udělá to, co by neměl) a způsobí selhání. Důležité je podotknout, že ne každý defekt musí selhání způsobit. Příčiny defektů jsou různé - lidé pracují pod časovým tlakem, se složitým kódem, ve složité infrastruktuře, s technologiemi, které se mění, nebo kvůli působení různých systémů navzájem. A hlavně z podstaty toho, že lidé jsou omylní. Nesmí se zapomenout také na přírodní podmínky, které mohou defekty způsobovat - radiace, magnetismus, elektrické pole a znečištění mohou vyvolat chyby ve firmwaru nebo ovlivnit chování softwaru změnou hardwarových podmínek (Müller, 2007).

3.3 Zajištění kvality softwaru

Koncept a význam slova „kvalita“ je velmi těžce definovatelný. Různí lidé definují kvalitu různě. Někteří ji mohou definovat měřitelnými termíny, které lze realizovat. Pokud se zeptáte, co odlišuje něčí produkt nebo službu, každý odpoví jinak. Když se zeptáte bankéře, řekne, že služby, pracovník ve zdravotnictví řekne „kvalitní péče“, zaměstnanec hotelu řekne „uspokojení zákazníka“ a výrobce odpoví „kvalitní produkt“ (Ross ,2009).

Důsledné testování softwaru může snížit riziko problémů, které mohou nastat. Tím přispívají ke kvalitě systému, protože se na defekty přijde dřív, než se software uvolní do provozu. Pomocí testování je tedy možné měřit kvalitu softwaru podle zjištěných defektů, pro funkcionální i nefunkcionální požadavky na software. Testování tedy může dodat důvěru v daný software, protože pokud dobře navržený test projde s žádným nebo jenom minimálním počtem defektů, snižuje to celkovou úroveň rizika v systému. Pokud test nalezne defekty a ty se opraví, kvalita softwaru se zvýší. Pokud je správně pochopena podstata příčin defektů zjištěných v předchozích projektech, mohou být zlepšeny procesy, které by měly zabránit vytváření stejných defektů v následujících projektech v budoucnu, což je aspekt zabezpečení kvality (Schaefer, 2014).

Jak bylo zmíněno, testování softwaru přispívá k jeho kvalitě. Což ovšem neznamená pouze to, aby se dostatečně rychle našly chyby, které jsou v softwaru obsaženy. Testování také přispívá k celkové kvalitě softwaru. Podle ISO/IEC 9126-1 standardu charakterizují kvalitu softwaru tyto faktory:

- Funkčnost
- Spolehlivost
- Použitelnost
- Účinnost
- Udržovatelnost
- Přenositelnost (Schefer, 2014).

3.4 Krátký vhled do historie

V roce 1947 existovaly počítače jako obrovské stroje zabírající celou místnost, které pracovaly s mechanickými relé a žhavily stovky vakuových elektronek. V té době byl nejmodernějším počítačem Mark II, obrovské monstrum sestavené na Harvardské univerzitě. První kroky tohoto nového přístroje doprovázeli technici, když najednou zařízení přestalo pracovat. Technici se vrhli do zjišťování problému, který zapříčinil havárii, a nakonec mezi sadami reléových kontaktů hluboko v počítači našli zaklíněnou můru. Do počítače nejspíše zabloudila z důvodu, že jí tam nalákalo světlo, nakonec ovšem našla smrt pod vysokým napětím mezi kontakty relé. A tak se zrodil počítačový "bug" (v překladu z angličtiny hmyz) (Patton, 2002).

3.5 Známé případy chyb v softwaru

Je jednoduché brát software za samozřejmost a tak si neuvědomit, nakolik se dostal do našich každodenních činností, i když byl vývoj vcelku rychlý. Od "počítačového pravěku", tedy zhruba roku 1947, kdy Mark II potřeboval celou řadu

programátorů, aby ho udrželi v chodu, lidé si vůbec neuměli představit mít svůj počítač doma. Dnes není problém dostat v krabici křupek CD a počítač dětí na hry obsahuje více softwaru než kosmická raketa. Dost často se tak stávalo, že těmi, kdo software testoval, byli až koncoví zákazníci (Patton, 2002).

Disney, Lví král, 1994 - 1995

Roku 1994 společnost Disney vydala svojí první hru Lví král (distribuce na CD-ROM) pro děti. Programy pro děti v té době nebyly již ničím revolučním, nabízela je řada firem, nicméně pro firmu Disney to byl první počin a jako takový byl na trhu náležitě propagován. Hra byla okamžitě rozprodána. Výsledkem byl ovšem pro firmu obrovský debakl. Den po Vánocích začaly u firmy Disney zvonit telefony zákaznické podpory a zvonit nepřestaly. Technici na lince telefonní podpory byli naprosto zahlceni telefonáty rozlícených rodičů plačících dětí, kteří software nedokázali rozchodit. Stránky novin a televizní zpravodajství plnily četné příspěvky. Příčinou této pohromy pro firmu Disney bylo to, že opomenula hru důkladně otestovat na různých typech osobních počítačů, které se daly na trhu pořídit. Výsledný software tak pracoval pouze na některých typech systémů - pravděpodobně pouze na takových, na kterých byla hra vytvořena. Ne ovšem na těch, které měla doma široká veřejnost (Patton, 2002).

Chyba v dělení s pohyblivou řádovou čárkou u procesoru Intel Pentium, 1994

Napište do kalkulačky ve svém počítači tento výpočet:
(4195835 / 3145727) * 3145727 - 4195835

V případě, že dostanete jako výsledek nulu, počítač je v pořádku. Pokud byste ovšem dostali cokoliv jiného, jedná se o chybu ve vašem procesoru, a tedy pravděpodobně máte starý procesor Pentium s chybou v dělení čísel s pohyblivou řádovou čárkou. Jedná se o softwarovou chybu, kterou se podařilo vypálit do integrovaného obvodu a v procesu výroby ji tedy mnohokrát zopakovat. Roku 1994 Dr. Thomas R. Nicely z Lynchburg College (Virginia) přišel na to, že při dělení vznikají problémy s neočekávanými výsledky. Přišel na

to při jednom ze svých experimentů, který byl řešen právě na tomto procesoru Pentium. Svoje zjištění záhy zveřejnil na internetu a vyvolal tím bouři, protože nebyl sám, kdo tuto chybu dokázal vyvolat. Nicméně se stále jednalo o malé množství lidí, pouze těch, kteří prováděli extrémně náročné matematické výpočty, tedy nějaké vědeckotechnické operace. Většina lidí si při běžné práci ničeho nevšimla. Inženýrům z Intelu se podařilo na chybu přijít, nicméně jí Intel nepovažoval za závažnou a tak jí původně ani nechtěl opravovat. Ve výsledku se ale strhl veřejný křik, kdy lidé na internetových diskuzích žádali opravu problému a novinové články líčily firmu Intel jako nezodpovědnou a nedůvěryhodnou. Nakonec se Intel veřejně omluvil a uvolnil přes 400 milionů dolarů jako náklady na náhradu vadných čipů (Patton, 2002).

Obranný raketový systém Patriot, 1991

Americký prezident Ronald Reagan plánoval program Strategické obranné iniciativy (Strategic Defense Initiative) neboli systém "hvězdných válek". Obranný raketový program Patriot byl jeho očesanou verzí. Poprvé byl uveden do provozu v Perském zálivu a měl sloužit jako obrana proti irácký raketám Scud. O tomto systému bylo psáno v mnoha novinových článcích, které obranný systém oslavovaly. Nicméně se mu nepodařilo ubránit proti několika raketám, včetně jedné rakety, která následně usmrtila 28 amerických vojáků. Pomocí analýzy se později přišlo na to, že příčinou problému byla chyba softwaru. Jednalo se o malou chybu v časování systémových hodin, která po zhruba 14 hodinách akumulace přerostla natolik, že naváděcí systém již nebyl přesný. Při dhahránském útoku, jež zabil oněch 28 vojáků, systém pracoval nepřetržitě více než 100 hodin (Patton, 2002).

3.6 Metody vývoje softwaru

Jedna ze základních vlastností dobrého testera je, aby porozuměl celkovému procesu vývoje softwaru, který je uplatňován na daném projektu. Při programování ve škole či doma, kdy jde o koníček, se používají zcela jiné metody, než ty, které jsou použity

pro vývoj softwaru ve velkých společnostech. Do takového procesu mohou být zapojeny desítky, stovky a dokonce i tisíce členů vývojového týmu. Každý z nich má jiné cíle a jiné role a pracuje se podle plánu, který je stanoven (Patton, 2002).

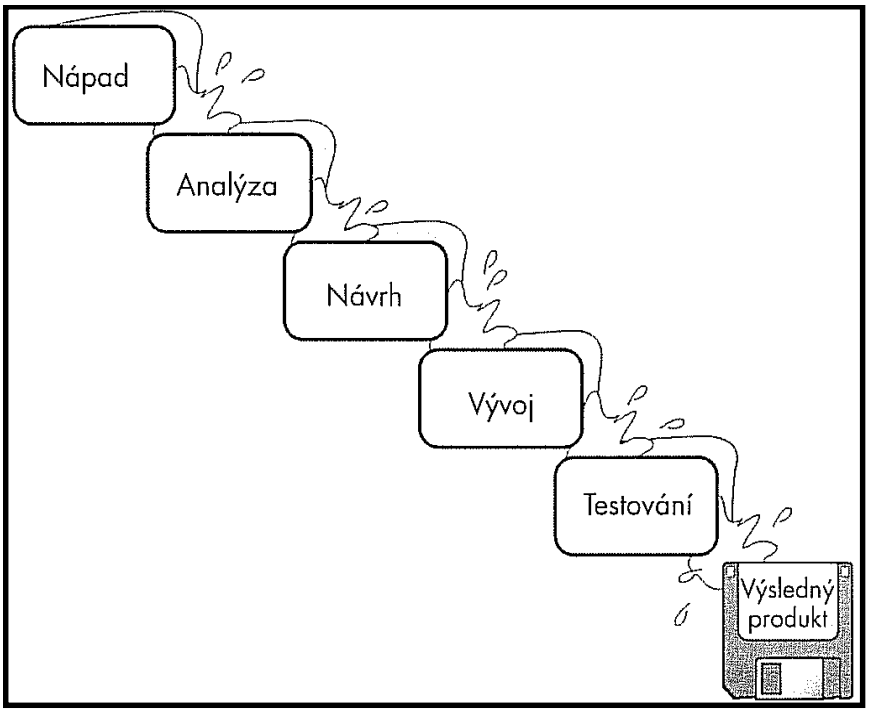
To, kdo do procesu tvorby softwaru je zapojen, je velmi závislé na konkrétním projektu a firmě. Nicméně i tak jsou role lidí velmi podobné a většinou se liší pouze označením pracovní pozice. Uveďme si seznam nejdůležitějších členů společně s jejich úkoly:

- ***Manažeři (vedoucí projektu)*** – lidé, kteří od začátku do konce řídí projekt, jsou často zodpovědní za plánování postupu prací, za rozhodování a za psaní specifikace produktu
- ***Architekti (systémoví inženýři)*** – techničtí experte produktového týmu, navrhují celkovou architekturu systému (návrhu softwaru), úzce spolupracují s programátory, bývají to velice zkušení a kvalifikovaní pracovníci
- ***Programátoři (vývojáři, kodéři)*** – navrhují software, odstraňují z něj chyby, úzce spolupracují s architekty a manažery projektu a také s testery kvůli opravě chyb
- ***Testeři (Quality Assurance, QA)*** – vyhledávají a oznamují nalezené problémy v produktu, úzce spolupracují se všemi členy týmu
- ***Techničtí tvůrci*** – tvorba papírové a elektronické dokumentace, která je dodávána spolu s produktem
- ***Správce konfigurace*** – zajišťuje proces slučování veškerého softwaru napsaného programátory a dokumentace do jednoho balíku (Patton, 2002)

3.6.1 Model vodopádu

Nejtradičnější model, který bývá jako první přednášen při výuce programování. Je používán snad odjakživa, pokud je použit u správných projektů, tak dokáže fungovat i přes jeho stáří. Dá se o něm říci, že je jednoduchý, elegantní a dává smysl. Jednotlivé kroky tohoto modelu jsou popsány níže uvedeným obrázkem (Patton, 2002).

Obrázek č. 2: Vývoj softwaru v modelu vodopádu



Zdroj: Ron Patton

Z obrázku je patrné, že projekt tvorby softwaru, jenž je vyvíjen v rámci modelu vodopádu, postupuje „dolů“ s posloupností kroků začínajících nějakým nápadem, nějakou prvotní myšlenkou. Na konci každého kroku je vývojovým týmem zhodnoceno, zda je vhodné pustit se do dalšího kroku. Projekt končí samozřejmě výsledným produktem (Patton, 2002).

Model vodopádu obsahuje tři důležité věci:

- Na specifikaci výsledné podoby produktu je kladen velký důraz. Z obrázku je vidět, že fáze vývoje se nachází pouze v jednom jediném bloku
- Jednotlivé kroky této metody se nepřekrývají, jsou diskrétní.
- Není zde cesty zpět. Je potřeba dokončit všechny potřebné věci v daném kroku a poté přejít dále (Patton, 2002).

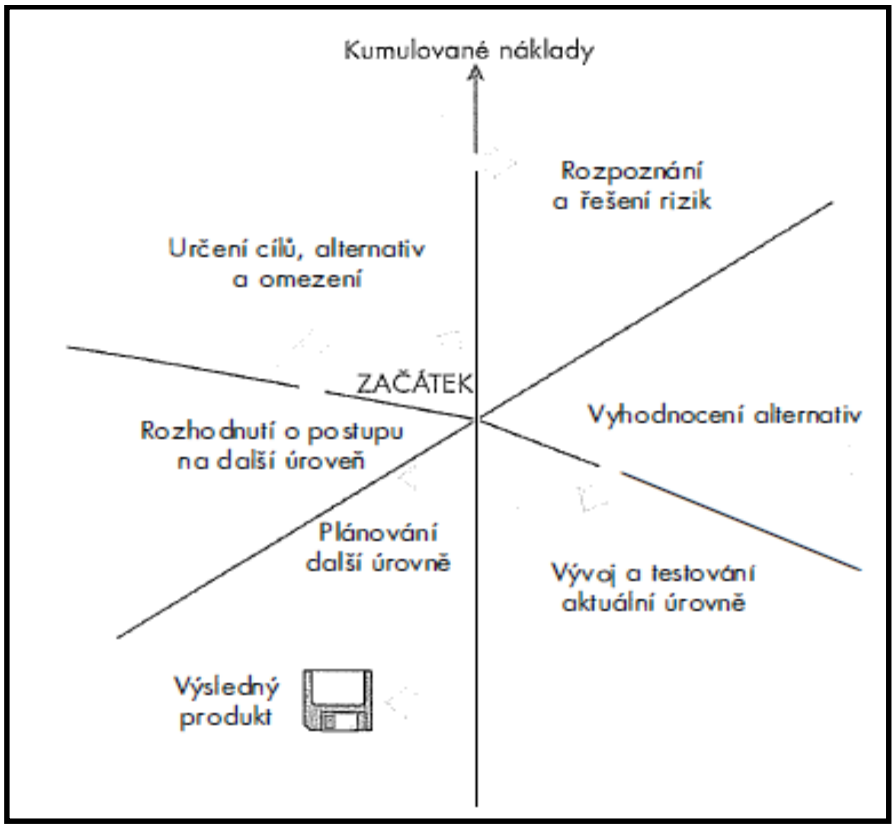
Takovýto přístup se může zdát omezující, nicméně pokud se pracuje na projektu s dobře srozumitelnou definicí produktu a pokud je vývojový tým disciplinovaný, tak může tato metoda poskytovat opravdu dobré výsledky. Před napsáním prvního řádku kódu je potřeba popsat všechny detaily. Nevýhodou této metody je to, že v dnešní době, kdy velká část projektů vzniká na internetu, tak promyšlená definice produktu se může zcela změnit v průběhu vývoje (Patton, 2002).

Co se týká testování, tak má tato metoda jednu velkou výhodou. A tou je opět specifikace, která definuje naprosto vše. A tedy v okamžiku, kdy se software předá testovací skupině, bylo již o každém detailu rozhodnuto. Z těchto předpokladů může testovací skupina vytvořit jasný plán a rozvrh testů. Testeři přesně vědí, co testují a jaké jsou vlastnosti systému. Na druhou stranu přichází s touto výhodou také nevýhoda a to ta, že pokud byla při vývoji zanesena nějaká velká chyba, tak se na ní přijde až v situaci, kdy se má hotový produkt umístit na trh (Patton, 2002).

3.6.2 Spirálový model

Spirálový model byl zaveden roku 1986 (autorem je Barry Boehm). Základní myšlenka je taková, že na začátku nelze definovat naprosto vše podrobně. Začne se tedy s definicí těch nejdůležitějších funkcí systému. Funkce se vyzkouší, jsou vyžádány připomínky zákazníků a následně se přejde na další úroveň vývoje. Proces se takto opakuje, dokud se nedojde k výslednému produktu (Patton, 2002).

Obrázek č. 3: Spirálový model vývoje softwaru



Zdroj: Patton

Obrázek lze popsat tímto seznamem:

1. Určení cílů, alternativ a omezení
2. Rozpoznání a řešení rizik
3. Vyhodnocení alternativ
4. Vývoj a testování aktuální úrovně
5. Plánování další úrovně
6. Rozhodnutí o postupu na další úroveň (Patton, 2002)

Tento model je směsicí vícero modelů. Součástí je částečně i vodopádový model. Týká se toho, že jsou zde také analýzy, návrh, vývoj a testování. Výhodou je zde včasné odhalení chyb, čímž se redukují samozřejmě náklady spojené s opravou. Z pohledu testera je tento model atraktivní z toho důvodu, že je zde pozice testera zapojena do procesu hned od začátku. Tester je zapojen do předběžných fází vývoje a vidí, kam projekt směřuje

a odkud přišel. Nikdo testera nebude tlačit, aby testoval zrovna teď, v posledních minutách. Testuje se průběžně a neustále, tudíž se v posledním průchodu akorát ověřuje, že je všechno skutečně v pořádku (Patton, 2002).

3.6.3 Metodika Rational Unified Process (RUP)

Tato metoda je do hloubky i do šířky velmi propracovanou komerční metodikou vývoje softwaru. Je založena na objektově orientované iterativní koncepci. Samotná společnost Rational chápe svůj produkt tak, že se jedná o jakousi online instruktáž, jež poskytuje metodické pokyny, instrukce a šablony pro všechny fáze vývoje. Jsou zde podrobně popsány veškeré otázky související s vývojem softwaru, tedy „kdo“ (pracovníci), „co“ (meziprodukty), „kdy“ (pracovní procesy) a „jak“ (činnosti) (Kadlec, 2004).

Metodika RUP je vhodná spíše pro rozsáhlé projekty a větší vývojové týmy, které potřebují stanovený a důsledně zdokumentovaný vývojový proces. Nicméně lze RUP přizpůsobit i konkrétním požadavkům projektu a konkrétním týmům vývojářů. Je tedy použitelný i pro menší týmy nebo internetové projekty (Kadlec, 2004)

3.6.4 Agilní metody

Jedná se o inovativní metody pro výrobu softwaru. Tyto metody sice na jednu stranu vychází z ortodoxního softwarového inženýrství, na druhou stranu ovšem tyto dogmata boří. Jako hlavní cíl si kladou vytvořit software mnohem rychleji, svižněji a celkově efektivněji. Tyto metody se nechtějí zakopávat v jednotlivých vývojových fázích, chtějí postupovat co nejrychleji k cíli, což znamená, že chtějí co nejdříve dodat fungující aplikaci a vyhovět požadavkům dnešní dynamické doby (Kadlec, 2004).

Tým, o kterém je smýšleno jako o agilním, se drží hodnot agilního manifestu, používá nějakou agilní metodiku (klidně vlastního původu) a používá agilní techniky (např. prototypování či párové programování) (Knesl, 2014).

Manifest pro agilní vývoj softwaru je dostupný na webu agilemanifesto.org. Kent Beck (2001) spolu s dalšími tvůrci uvádí: „Objevujeme lepší cesty vývoje softwaru tím, že to tak děláme a tím, že to pomáháme dělat ostatním. Skrze práci jsme přišli k těmto hodnotám:

- Individualita a interakce nad procesy a nástroje
- Pracující software je více než obsáhlé dokumentace
- Spolupráce se zákazníky je více než smluvní vyjednávání
- Reakce na změny oproti následování plánu“

Agilní metody předpokládají měnící se požadavky na funkcionalitu, na rozdíl od těch klasických, kde se množina prvků považuje za neměnnou. Mezi základní principy agilních metodik patří:

- Inkrementální vývoj s krátkými iteracemi
- Osobní komunikace
- Nepřetržitý kontakt se zadavatelem/uživatelem
- Opakované automatizované testování (Kadlec, 2004)

Existuje celá řada agilních metod (např. Lean Development, Crystal, Kanban, testy řízený vývoj SW, atd.). Dvě nejznámější si lze uvést:

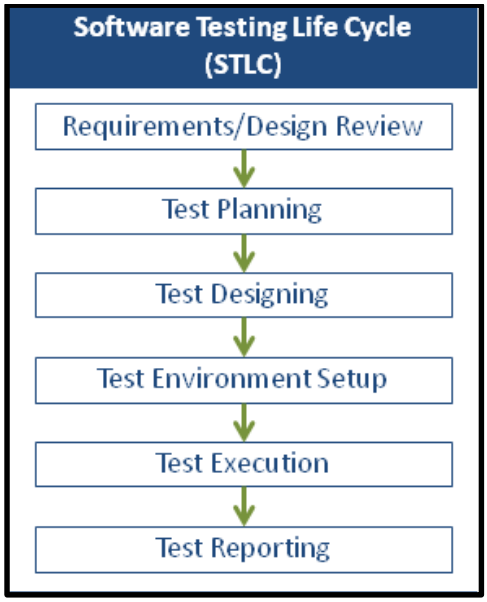
- ***Metodika extrémní programování*** – velice známá agilní metodika určená pro menší týmy. Je progresivní a má odlehčenou filozofii, která boří konzervativní představy o týmové spolupráci při vývoji, počítá se zde totiž se změnami požadavků a nehledí tolik dopředu. Problém se prozkoumá a ihned je zahájena implementace (návrh probíhá současně s implementací

a testováním), s čímž je spojen refaktoring a odvaha i komplet přepracovat nebo zahodit již hotové zdrojové kódy.

- ***Metodika Scrum*** – název z ragbyové terminologie jako paralela pro vývoj SW. Tato metoda probíhá v iteracích, tzv. sprintech (mohou být např. dvoutýdenní, měsíční, atd.), nicméně je dopředu známo, jak bude dále vývoj probíhat. Koordinace a rekapitulace probíhá v tzv. Scrum meetings. Tato metoda vychází z objektově orientovaného přístupu. Scrum týmy bývají tříčlenné až šestičlenné. Každodenní reflexe a pružná reakce jsou hlavní přednosti Scrumu (Kadlec, 2004).

3.7 Základní testovací proces

Obrázek č. 4: Životní cyklus testování softwaru



Zdroj: softwaretestingfundamentals.com

Širší části okolí vývoje softwaru se může zdát, že jedinou částí při testování je samotné vykonávání testů. Je to ta nejvíce viditelná část. Nicméně aby testování bylo efektivní a účinné, měly by samozřejmě testovací plány obsahovat také čas na plánování testů, navržení

testovacích případů, přípravu pro vykonávání testů a vyhodnocení stavu (Müller, 2007).

Testovací proces je seskládán z několika aktivit, které jsou sice logicky postupné, nicméně se mohou v průběhu procesu překrývat, nebo mohou být vykonávány paralelně. Jsou to:

- Plánování a řízení
- Analýza a návrh
- Implementace a vykonávání
- Vyhodnocení výstupních kritérií a reportování
- Činnosti související s ukončením testování (Müller, 2007)

3.7.1 Plánování a řízení testování

Při plánování testování jde především o to, aby se definovaly cíle testování a specifikace testovacích činností s cílem dosažení cíle a poslání. Jde vlastně o takovou verifikaci poslání testování (Müller, 2007).

Oproti tomu řízení testování je aktivita, která probíhá průběžně a kontroluje nynější postup vůči plánu a reportování stavu a odchylek od plánu. Realizují se činnosti, které jsou nezbytné k tomu, aby se naplnil stanovený cíl a poslání projektu. Testování musí být sledováno v průběhu projektu, aby mohlo být řízené. Plánování testování bere v úvahu také zpětnou vazbu z monitorovacích a řídících aktivit (Müller, 2007).

3.7.2 Analýza a návrh testování

V analýze a návrhu testování probíhají takové činnosti, které transformují všeobecné cíle testování do konkrétních testovacích podmínek a testovacích scénářů.

Analýza a návrh testování zahrnují následující hlavní činnosti:

- Revidování základu testování (požadavky, architektura, návrh, rozhraní)
- Vyhodnocení, zda bude základ a objekt testování testovatelný
- Identifikace stanovování priorit testovacích podmínek, což je založeno na tom, aby se analyzovaly testovací položky, specifikace, chování a struktura
- Navrhování a stanovování priorit testovacích případů
- Identifikování toho, jaká testovací data budou potřeba pro podporu testovacích podmínek a testovacích případů
- Návrh, jak nastavit testovací prostředí a identifikace požadované infrastruktury a nástrojů (Müller, 2007)

3.7.3 Implementace a vykonávání testování

Při implementaci a vykonávání testování je nastaveno testovací prostředí, kombinací testovacích případů v určitém pořadí a zahrnutím všech dalších informací potřebných pro vykonávání testů jsou specifikovány testovací procedury nebo skripty a testy jsou spuštěny (Müller, 2007).

Hlavní činnosti pro implementaci a vykonávání testů:

- Tvorba, implementace a prioritizování testovacích případů
- Tvorba a prioritizace testovacích procedur, tvorba testovacích dat, psaní automatizovaných testovacích skriptů
- Tvorba sestav testů z testovacích procedur pro účinné vykonávání testů
- Verifikace nastavení testovacího prostředí
- Vykonání testovacích procedur podle pořadí a to manuálně nebo s pomocí nástrojů pro vykonávání testů
- Výsledky vykonaných testů a jejich zaznamenávání a zaznamenávání označení a verzí testovaného softwaru, testovacích nástrojů a testwaru
- Skutečné výsledky versus očekávané výsledky

- Analýza incidentů (tedy neshod) za účelem stanovení příčiny (defekt v kódu, chyba v testovacích datech, ve způsobu vykonání testu)
- Opakování testovacích aktivit, tedy výsledek přijetí opatření pro nějakou nesrovnalost. Například retestování případu (konfirmační testování), vykonání opraveného testu, regresní testování, atd. (Müller, 2007)

3.7.4 Vyhodnocení výstupních kritérií a reportování

Vyhodnocení výstupních kritérií by mělo být prováděno pro každou úroveň testování. Jedná se o aktivitu, kde je vykonání testů hodnoceno vůči definovaným cílům (Müller, 2007).

Hlavní činnosti vyhodnocování výstupních kritérií:

- Kontrola protokolů o testech vůči výstupním kritériím specifikovaným během plánování testování
- Rozhodnutí o tom, zda jsou potřeba další testy, případně zda by měla být změněna specifikovaná výstupní kritéria
- Souhrnná zpráva o testování pro klíčové osoby (Müller, 2007)

3.7.5 Aktivity uzavření testu

Tyto aktivity shromažďují data z ukončených testovacích aktivit. Účel je konsolidace zkušeností, testwaru, fakt a čísel. Může to být po uvolnění softwaru, skončení testovacího projektu, případně po dosažení milníku, nebo když byla ukončena oprava provozní verze.

Hlavní činnosti aktivit uzavření testu:

- Kontrola plánovaných dodávek, kontrola uzavření incidentů, nahlášení otevřených incidentů, kontrola dokumentace ohledně akceptace systému

- Finalizace a archivování testwaru, testovacího prostředí a testovací infrastruktury v případě toho, že by se později opětovně použil
- Odevzdání testwaru organizaci, která zajišťuje provoz
- Analýza, ponaučení pro další releasy a projekty a zlepšení vyspělosti testování (Craig, 2002)

3.8 Úrovně testování

3.8.1 Testování komponent (unit testy)

Jedná se o hledání defektů uvnitř softwarových komponent a verifikaci fungování softwarových komponent. Mohou to být např. moduly, programy, objekty, třídy, atd., které jsou testovatelné samostatně. Testy těchto jednotek se zapisují ve formě programového kódu, proto jsou většinou obsluhovány vývojáři (Hlava, 2011).

Testování komponent může být (je to v závislosti na kontextu životního cyklu vývoje a systému) vykonáno v izolaci od zbytku systému. Mohou zde být použity nástavce, ovladače, simulátory. Může zde být zahrnuto testování funkcionality a specifických nefunkcionálních charakteristik. Může se jednat o chování zdrojů (přetečení paměti) anebo testování robustnosti, jakože i strukturální testování. Testovací případy jsou odvozeny z pracovních produktů jako specifické komponenty, návrh softwaru anebo datový model. Testování komponent se nejčastěji realizuje tak, že je zde přístup ke kódu, který je testován, a s podporou vývojového prostředí, jako je třeba framework jednotkového testování anebo nástroj pro ladění. V praxi je to většinou tak, že je přítomen programátor, který kód psal. Incidenty se nezapisují a defekty jsou opravovány hned, jak jsou nalezeny (Müller, 2007).

V testování komponent existuje např. jeden z přístupů, kdy je ještě před kódováním příprava a automatizace testovacích případů. Tento přístup je nazýván "testování na prvním místě" anebo "vývoj řízený testováním". Jedná se o vysoce iterativní přístup, který je založený na cyklech, ve kterých se vyvíjejí testovací případy, následně se vytvářejí

a integrují malé části kódu a probíhají testy komponent, do té doby, než jsou úspěšné (Müller, 2007).

3.8.2 Integrační testování

Integrační testování je testování interakce mezi různými částmi systému jako jsou operační systém, souborový systém, hardware anebo rozhraní mezi systémy. Jedná se tedy o testování rozhraní mezi komponentami. Nemusí zde existovat pouze jedna úroveň integračního testování a navíc může být vykonáno na objektech různé velikosti. Například:

1) Interakci mezi komponentami softwaru prověřuje integrační testování komponent a je vykonáno po testování komponent
2) Interakci mezi různými systémy prověřuje systémové integrační testování a je vykonáváno po systémovém testování. Organizace, která systém vyvíjí, může kontrolovat pouze jednu stranu rozhraní, takže změny by mohly narušit stabilitu systému. Byznys procesy implementované jako workflow mohou zahrnovat série systémů, proto mohou být problémy na úrovni více platforem závažné (Müller, 2007).

Platí, že čím větší záběr integrace je, tím složitější je izolovat selhání na specifickou komponentu nebo systém, což vede ke zvýšení rizika. Integrační strategie mohou být založeny na architektuře systému (shora-dolů, zdola-nahoru), funkcionálních úlohách, sekvencích zpracování transakcí nebo na jiných aspektech systému nebo komponenty. Riziko pozdních objevení defektů se snižuje tím, že je integrace realizována jako inkrementální namísto "velkým třeskem". Co se týká integračního testování, může zde být zahrnuto testování specifických nefunkcionálních charakteristik (např. výkon) (Müller, 2007).

Co se týká testování, testeři se zaměřují výhradně na samotnou integraci a to v každém stádiu integrace. Tedy například pokud je modul A integrován s modulem B, tak se zajímá o komunikaci mezi moduly a ne o funkcionalitu jednotlivých modulů. Ideální

případ by byl takový, kdyby testeři rozuměli architektuře a ovlivňovali integrační plánování. Pokud jsou integrační testy plánovány předtím, než jsou vytvořeny komponenty nebo systémy, mohou být vytvořeny v takovém pořadí, aby bylo testování co nejúčinnější (Müller, 2007).

3.8.3 Systémové testování

Systémová testování se zabývá tím, zda se chová systém/produkt tak, jak byl definován rozsahem projektu. Testovací prostředí by zde mělo v co největší míře korespondovat s finálním cílovým nebo provozním prostředím a to za účelem co nejmenšího rizika, aby chyby specifické pro prostředí byly nalezeny. Systémové testování může zahrnovat testy, které jsou založeny na rizicích, specifických požadavcích, případech užití, byznys procesech nebo jiných popisech toho, jak se má daný systém chovat. Systémové testování je často realizováno nezávislým testovacím týmem. (Müller, 2007).

Systémovým testováním by měly být prozkoumány funkcionální i nefunkcionální požadavky systému. Tyto požadavky existují jak v textové podobě, tak také ve formě modelů. Je potřeba se i vypořádat s nekompletními nebo nedokumentovanými požadavky. Systémové testování funkcionálních požadavků začíná použitím vhodných technik založených na specifikaci (černá skříňka) vzhledem k aspektu systému. Jako příklad si lze uvést vytvoření rozhodovací tabulky pro kombinace následků popsaných v byznys pravidlech. Techniky, které jsou založeny na struktuře (bílá skříňka) mohou být posléze použity pro zhodnocení důkladnosti testování s ohledem na strukturální element. Např. navigace web stránky (Müller, 2007).

3.8.4 Akceptační testování

Akceptační testování je zodpovědností pro zákazníky a uživatele systému. Mohou zde být zapojeny i další klíčové osoby. V podstatě se zde jedná o nastolení důvěry v software, jeho části a nefunkcionální charakteristiky systému. Je potřeba ukázat, že je

systém založen na uživatelských požadavcích a je potřeba demonstrovat, že jim bylo vyhověno. Není zde hlavním účelem nalézt defekty, ale ohodnotit systém z toho pohledu, zda je připraven k nasazení a používání. Nemusí však být poslední úrovní testování. Jako příklad si lze uvést rozsáhlé integrační testování, které může být vykonáno až po tom akceptačním (Craig, 2002).

Akceptační testování nemusí být pouze jednou úrovní testování. Například:

- Krabicový produkt - akceptační testování může probíhat při instalaci a integraci
- Použitelnost komponenty - akceptační testování může probíhat v rámci testování komponenty
- Funkcionální zlepšení - akceptační testování může probíhat před systémovým testováním (Craig, 2002)

Typické formy akceptační testování

- Uživatelské akceptační testování - verifikace připravenosti na použití systému byznys uživateli

- Provozní akceptační testování - akceptování systému systémovými administrátory:
    - testování zálohy/obnovy
    - regeneraci po havárii
    - správy uživatelů
    - úlohy údržby
    - periodická kontrola bezpečnostních nedostatků

- Smluvní a regulační akceptační testování - toto testování je vykonáváno vůči smluvním akceptačním kritériím pro provoz systému vyvinutého na zakázku. Akceptační kritéria je potřeba definovat v době, kdy se odsouhlasuje kontrakt. Regulační testování je prováděno vůči nařízením, které musí být dodrženy. Může se jednat o vládní, právní nebo bezpečnostní předpisy.

- Alfa testování a beta testování (nebo testování “v terénu”) - vývojáři, kteří dělají trhový nebo krabicový software, mohou chtít zpětnou vazbu od potenciálních nebo stávajících zákazníků na jejich trhu před tím, než bude uveden do komerčního prodeje. Alfa testování je prováděno na půdě organizace, beta testování je prováděno osobami v jejich vlastním prostředí. Obě testování jsou prováděny potenciálními zákazníky, ne vývojáři produktu (Müller, 2007).

3.9 Typy testů

Testovací aktivity jsou zaměřeny na verifikaci softwaru na základě specifického důvodu nebo cíle testování. Každý typ testu se zabývá určitým účelem testování. Může se jednat o testování určitě funkce, kterou má software umět, nefunkcionální kvalitativní charakteristika jako například spolehlivost nebo použitelnost, struktura nebo architektura softwaru nebo systému. Mohou to být potvrzení o tom, že defekty již byly opraveny a hledání neúmyslných změn (Müller, 2007).

3.9.1 Funkcionální testování (Černá skříňka)

Funkcionální testování neboli testování funkcionality. Systém, subsystém nebo komponenta, vykonává určité funkce, které mohou být popsány v pracovních produktech (specifikace požadavků, případy užití nebo funkcionální specifikace nebo mohou být nezdokumentovány). Funkce vlastně představují to, co systém dělá. Funkcionální testy mohou být vykonány na všech úrovních testování a jsou založeny na funkcích a vlastnostech (jejich popis je buď v dokumentech, případně se hodnotí tím, jak jsou chápány testery) a jejich kooperaci se specifickými systémy (Müller, 2007).

Funkcionální testování zkoumá externí chování softwaru, jedná se tedy o testování “černé skříňky”. Techniky, které jsou založené na specifikaci, mohou být použity s tím cílem, aby se daly odvodit testovací podmínky a testovací případy z funkcionality softwaru nebo systému (Müller, 2007).

Jeden speciální typ funkcionálního testování zkoumá bezpečnost, testuje funkce (např. firewall) vztahující se k detekci hrozeb, jako například viry zlomyslných outsiderů. Další typ funkcionálního testování - testování spolupůsobení (interoperability), kde se hodnotí, jak je produkt schopen spolupůsobit s jednou nebo více komponentami nebo systémy (Müller, 2007).

3.9.2 Nefunkcionální testování

Testování nefunkcionálních softwarových charakteristik je takové testování, které řeší testování výkonu, zátěžové testování, stres testování, testování použitelnosti, testování udržovatelnosti, testování spolehlivosti a testování přenositelnosti. Není omezeno však pouze na ně. Je testováním toho, jak systém pracuje (Müller, 2007).

Toto testování popisuje testy, které jsou vyžadovány pro měření charakteristik systému. Ty testy, které mohou být kvantifikovány vůči stupnicím měření. Může to být doba odezvy systému jako test výkonu. Jedná se o testování, které může být prováděno na všech úrovních testování (Müller, 2007).

3.9.3 Strukturální testování (Bílá skříňka)

Testování struktury/architektury softwaru je testováním, které také může být vykonáno na všech úrovních testování. Nejlepší použití strukturálních charakteristik je tam, kde se používají techniky založené na specifikaci s účelem měřit důkladnost testování pomocí stanovení pokrytí typu struktury. Pokrytí je míra, do které je struktura prověřena testovací sadou, vyjádřena jako procento položek, které daná sada pokrývá. Pokud se zjistí, že pokrytí není 100%, tak položky, které nejsou pokryty, mohou být otestovány dalšími testy, které budou navrženy a zvýší tak pokrytí. (Müller, 2007).

Při strukturálním testování mohou být použity nástroje pro měření pokrytí elementů kódu, například příkazů nebo rozhodování. To může být použito na všech úrovních testování, ale

obzvláště při testování komponent a integračním testování. Strukturální testování může být založeno na architektuře systému, například hierarchie volání částí systému. Tyto přístupy mohou být rovněž použity na systémové, systémové integrační nebo akceptační úrovni testování (například byznys modely) (Müller, 2007).

3.9.4 Retestování a regresní testování

Retestování nebo také konfirmační testování je zde za účelem toho, že pokud má systém nějaký defekt, který je nalezen a následně opraven, musí být potvrzeno, že se tak opravdu stalo a chyba byla odstraněna. Nutno dodat, že ladění, tedy oprava defektů, je aktivita vývoje a ne testování (Müller, 2007).

Dále do testování souvisejícího se změnami patří regresní testování, což je opakované testování již testovaného programu po úpravě. Jedná se testování, které má za cíl najít všechny defekty, jež byly do softwaru zaneseny jako důsledek změny. Tyto chyby se mohou nacházet v systému, který je testován nebo v jiné související či nesouvisející softwarové komponentě. Regresní testování je prováděno v případě, kdy se mění software nebo jeho prostředí. Rozsah se odvozuje od rizika nenalezení defektů v softwaru, který předtím fungoval. Důležitým faktem je to, že tyto testy by měly být opakovatelné, aby se mohly používat v konfirmačním testování a pomáhat při regresním testování. Regresní testování se může použít ve všech úrovních testování a je využito na funkcionální, nefunkcionální i strukturální testování. Sady regresních testů se vyvíjejí pomalu a jsou spouštěny mnohokrát. Proto je regresní testování velmi silným kandidátem na automatizaci (Müller, 2007).

3.10 Statické techniky testování softwaru

Rozdíl mezi statickým a dynamickým testováním je v zásadě takový, že při dynamickém testování musí být spuštěn kód softwaru. Dynamické testování je tedy v podstatě takové to klasické testování, kdy se spustí software a detekují se chyby

a zkoumá kvalita kódu. S použitím známosti testovacích výstupů porovnáváme vstupy a pozorujeme, zda si odpovídají. Tato kapitola se však zabývá testováním statickým, které se zabývá testováním pracovních meziproduktů, tedy např. specifikacemi požadavků, testovacích plánů, uživatelských manuálů, atd. Ty nejsou samozřejmě spouštěny, ale i tak mohou být testovány pomocí daných procesů. Statické techniky poskytují silný nástroj pro zlepšení kvality softwaru a zvýšení produktivity softwarového vývoje (Agnihotri, 2014).

3.10.1 Statická revize a její výhody

Statická revize, jak bylo řečeno, dokáže hned od počátku zlepšit vývoj softwaru v daném projektu. Hned na začátku dokáže identifikovat problémy, které jsou obsaženy v projektu a opravit je v raných fázích vývoje. Všechny softwarová firmy by měly mít revize ve všech hlavních aspektech jejich práce - tvorba požadavků, design, implementace, testování a údržba. Výhody a síla statické revize je v:

1. Typy defektů, jež mohou být nalezeny při statické revizi, jsou: odchylky od standardů, chybějící požadavky, designové defekty, neudržitelný kód a nekonzistence ve specifikaci rozhraní.

2. Pokud statické testování proběhne v rané fázi projektu, tak také v této brzké fázi mohou být nalezeny nedostatky a chybějící věci v požadavcích uživatele. Pokud se tak nestane, pravděpodobně budou nalezeny až velmi pozdě, tedy v akceptačním testování.

3. Díky detekování defektů v brzké době může být cena opravy relativně nízká a kvalita softwaru může být dosažena levně.

4. Zpětná vazba a návrhy v dokumentu pro statické testování, které se snaží vylepšit proces vývoje, mohou pomoci vyhnout se stejným chybám v budoucnu (Agnihotri, 2014).

3.10.2 Revizní role a jejich zodpovědnost

V revizním procesu je definováno mnoho rolí a jejich zodpovědností. Nicméně je zde 5 rolí, které by měly být obecné pro každý revizní proces:

- ***Moderátor*** je vedoucí revizního procesu. Má za úkol označit typ revize, přístup a složení revizního týmu. Dále moderátor svolává schůze, rozesílá dokumenty před schůzemi, řídí schůze a ukládá případná data.

- ***Autor*** je člověk, který napsal daný dokument. Hlavní úkol autora je naučit se možné věci, v rámci revizí, k vylepšení svého díla. Dále tam je pro to, aby objasňoval místa v dokumentu, kde by mohlo přijít k nedorozumění.

- ***Zapisovatel*** je v revizním procesu pro to, aby zapisoval nalezené defekty a návrhy pro zlepšení procesu.

- ***Revidující*** je potřeba pro to, aby zkoumal defekty a vylepšení, které se vztahují k business specifikaci, standardům a znalostem v jeho oblasti.

- ***Manažer*** je v revizi zapojen kvůli rozhodnutí o započnutí revize. Alokuje čas v projektech a rozhoduje, zda byly cíle revizního procesu splněny (Agnihotri, 2014).

3.10.3 Fáze formální revize

Formální revize obsahuje 6 základních kroků:

1. Plánování. Revizní proces začíná, když autor pošle žádost moderátorovi. Ten potom musí sestavit rozvrh pro revizi. Plán projektu potom musí zahrnovat čas pro revizi a čas pro přepracování. Následuje vstupní kontrola pro dokumenty. Rozhoduje se,

který dokument bude přezkoumáván. Musí se zvolit kompozice revizního týmu, role a strategie.

2. Vykopnutí. Cílem tohoto setkání je seznámit všechny s daným dokumentem. Jsou diskutovány vstupní a výstupní kritéria. V rámci tohoto setkání dostanou revidující pojednání o tom, co jsou cíle revize. Přiřazení rolí a případné změny v procesu jsou také součástí této schůze.

3. Příprava. V této fázi se účastníci revize individuálně podílejí na práci na dokumentu s použitím dokumentů, procedur a pravidel vztahujících se k dokumentu, který se reviduje. Účastníci identifikují defekt. Sepíší dotazy a komentáře k případným nedorozuměním.

4. Revizní schůze. Toto setkání se skládá ze tří částí. Protokolování, kde jsou zmíněny defekty, které byly nalezeny v rámci přípravy. Tato fáze slouží k tomu, aby se sepsaly všechny problémy. Pokud potřebuje být nějaký problém prodiskutován, je zaprotokolován a přesune se do další fáze. Další fází je tedy diskuze, kde se řeší všechny problémy, které byly zaprotokolovány. Výstupem je dokument pro budoucí reference. Poslední fází je fáze rozhodnutí, kde je nejdůležitější metrikou počet defektů na stránku. Pokud počet defektů na stránku přesáhne určitý stanovený počet, tak je potřeba dokument znovu revidovat.

5. Přepracování. V této fázi autor přepracovává dokument podle nalezených defektů.

6. Zpětná vazba. Poslední krok. Po přepracování se zjistí, zda všechny vylepšení byly zapracovány. Probíhá zde sbírání metrik a kontrolují se výstupní kritéria (Agnihotri, 2014).

3.10.4 Typy revize

- Walkthrough je revizní setkání vedené autorem. Jedná se o sezení s otevřeným koncem. Mezi hlavní účely tohoto setkání patří nalezení defektů, ale také učení a získání porozumění (Hower, 2015).

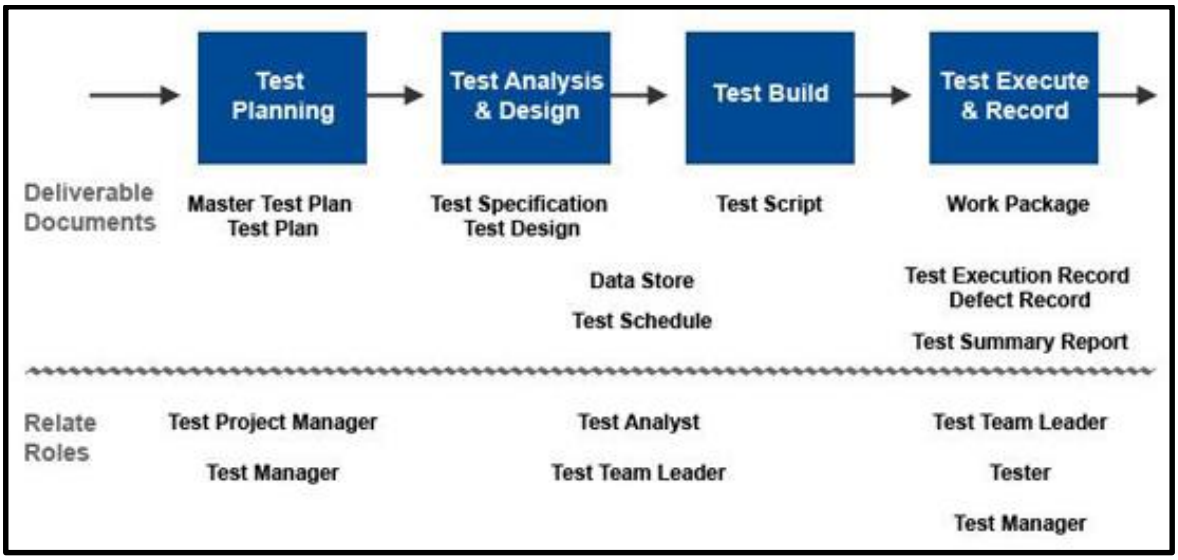
- Technická revize; je dokumentovaná. Jedná se o definovaný proces odhalení defektů. Jsou přítomni kolegové a techničtí experti. Může být bez účasti managementu s tím, že revidující je kolega na stejné úrovni. Příprava je před setkáním. Mezi hlavní účely patří diskuze, učinění rozhodnutí, vyhodnocování alternativ, nacházení defektů, řešení technických problémů (Hower, 2015).

- Inspekce je nejvíce formálním typem revize založeným na vstupních a výstupních kritériích. Je obvykle vedena specializovaným moderátorem, který nesmí být autor. Dokument je zkontrolován revidujícími před setkáním (Agnihotri, 2014).

3.11 Složení testovacího týmu

Složení testovacího týmu není pevně stanoveno a každá společnost má svojí danou hierarchii. Proto je potřeba říct, že následující výčet obsahuje opravdu pouze popis rolí, nikoliv však popis konkrétních osob, které jsou v daném týmu obsaženy. Jeden člověk může pojmout více rolí a naopak, tedy že jednu roli může vykonávat více osob. Jako příklad si lze uvést to, kdy manažer testovacího týmu vykonává roli analytika testů a zbytek týmu vykonává designera testů a zároveň vykonavatele testů.

Obrázek č. 5: Role v rámci testovacího cyklu



Zdroj: bayamp.com

Obrázek popisuje možné role a jejich vazbu na základní proces životního cyklu testování softwaru. Je zde vidět, že test manažer provádí plánování testování, zatímco test analytici a vedoucí testovacích týmů provádí jak specifikaci testovacích případů a jejich navrhování, tak zároveň psaní testovacích skriptů. Tester potom slouží pouze jako ten, co testy prochází a zapisuje nalezené chyby. Zvláštností v tomto případě je, že provádění testů zde dělá i test manažer a vedoucí testovacích týmů.

3.11.1 Test manažer / Test leader

Test manažeři jsou lidé, kterým nestačí znát pouze disciplínu testování softwaru jako takovou. Musí také umět řídit a implementovat testovací proces v dané organizaci. To vše vyžaduje potřebu umět vést lidi, komunikovat s nimi a schopnost umět měřit návratnost investic testovacím týmem (Johnson, 2007).

Funkce test manažera je zodpovědná za kvalitu a úsilí, jenž naplňuje testovací tým a samozřejmě za dohled nad všemi zaměstnanci zařazenými v testovacím týmu. Test manažer

zjišťuje, zda tester potřebuje trénink. A také musí komunikovat s ostatními odděleními a zajišťovat stabilní vztahy mezi nimi a testovacím týmem (Kaner, 1999).

Test manažer má zodpovědnost za:

- Definování a implementaci rolí v rámci organizace
- Definování rozsahu testování v rámci kontextu každé dodávky
- Nasazování a řízení testovacího frameworku
- Řízení a růst hodnot v testování, např.:
    - Členové týmu
    - Testovací nástroje
    - Testovací procesy
- Udržování vyškoleného testovacího týmu (Johnson, 2007)

Test manažer či Test lead musí umět rozumět tomu, jak testování zapadá do organizační struktury daného podniku. Jinak řečeno, musí umět definovat roli testovacího týmu v rámci organizace. To může být zaznamenáno a definováno interním testovacím nařízením. Například: „Preventivně zabraňovat, detekovat, zaznamenávat a řídit defekty v rámci další verze.“ Tím vzniká hlavní práce test manažera / test leadera, tedy vykomunikovat a implementovat efektivní manažerské a testovací techniky, které podpoří toto jednoduché nařízení (Johnson, 2007).

Test leader / manažer musí zařídit vhodný testovací framework nebo testovací architekturu, která je v souladu s organizačními testovacími potřebami. Nicméně požadavky testovacího frameworku pro jakoukoliv organizaci jsou těžké definovat, nejprve musí test manažer získat několik odpovědí. Tyto odpovědi dají základ krátkodobým i dlouhodobým cílům testovacího frameworku (Johnson, 2007).

Řízení nebo vedení testovacího týmu je jednou z nejnáročnějších pozic v rámci informačních technologií. Testovací tým má většinou nedostatečný počet zaměstnanců a má mezery v získání dostatečných nástrojů a financí od nejvyššího managementu. Termíny se neposouvají i přesto, že testovací fáze je čím dál více pod tlakem kvůli zpožděním v rámci

produktu. Motivace je jednou z klíčových osobnostních vlastností, které pomohou tyto problémy překonat. Pár základních rad, které mohou pomoci tyto problémy usměrnit:

- Udržovat dostatečnou komunikaci s vývojem a projektovým managementem
- Pokud je časová osa nějak ovlivněna, je potřeba modifikovat testovací plán v rámci rozsahu testování
- Neustále „prodávat“ důležitost testovacího týmu
- Jasně definovat role v rámci testovacího týmu a mít jasně definovaný možný kariérní růst pro členy svého týmu (Johnson, 2007)

Častou chybou, jež se někteří manažeři dopouštějí, je hodnocení testerů podle počtu nalezených chyb. Jako příklad si lze uvést testery, kteří se snaží nalézt příčinu chyby, případně se jí snaží reprodukovat v dalších částech aplikace. Takovým testerům se říká seniorní testeři, ne „méně produktivní“ testeři (Kaner, 1999).

3.11.2 Test analytik / Test designer

Test analytik, role někdy nazývaná test designer, je rolí, která pomáhá test manažerovi při specifikaci dílčích záležitostí (Králová, 2013).

Obrázek č. 6: Role test analytik - zodpovědnost

Zdroj: http://metodikamezitest.asp2.cz/

Na obrázku můžeme vidět, za co je test analytik zodpovědný a co sám provádí. Hlavní náplní test analytika tedy je:

- Analýza podkladů pro testování
- Identifikace testovacích podmínek
- Návrh testovacích případů
- Organizace a prioritizace testovacích případů
- Příprava testovacích dat
- Vytváření testovacích skriptů (Králová, 2013).

Po odsouhlasení testovacího plánu bere test analytik zodpovědnost za veškerou přípravu testování. Musí analyzovat a připravit podklady pro testování. V případě, že v nich nalezne nějaké nesrovnalosti, musí je autorům oznámit a předat zpět k opravě či upřesnění. Následně začne s přípravou testovacích podmínek na základě testovacího plánu a podkladů pro testování. Posléze začne navrhovat testovací případy, vytvářet testovací skripty (ty mohou být v některých případech automatizovány), připravovat testovací data a organizovat a prioritizovat testovací případy (Králová, 2013)

3.11.3 Test automator

Co se týká automatizace testovacích skriptů, může test analytik, pokud nemá dostatečné technické dovednosti, požádat o přípravu vývojáře. Případně může povolat speciálně vyškoleného technického testera (Králová, 2013).

Test automation experti se znalostmi testovacích základů, zkušenostmi s programováním a samozřejmě hlavně s výbornými znalostmi testovacích nástrojů a znalostmi skriptovacích jazyků. Využívají testovací nástroje a skriptovací jazyky podle potřeby daného projektu. Tato pozice se může často spojovat s pozicí testera či test analytika, což je typické pro testování v agilním vývoji softwaru (Schaefer, 2014).

3.11.4 Tester

Testeři jsou experti v procházení testů a reportování problémů. Testeři by měli mít základy v informačních technologiích, základy v testování, musí umět používat testovací nástroje a hlavně rozumět produktu, který testují (Schaefer, 2014).

Typické úlohy testera mohou zahrnovat:

- Revidování a podílení se na tvorbě testovacích plánů
- Analýzu, revidování a posouzení požadavků uživatelů, specifikací a modelů testovatelnosti
- Tvorbu specifikací testů
- Nastavování testovacího prostředí (častá koordinace s administrací systému a síťovým managementem)
- Přípravu a získávání testovacích dat
- Implementaci testů na všech úrovních testování, vykonávání a zaznamenávání testů, vyhodnocení výsledků testů a dokumentování odchylek od očekávaných výsledků
- Používání nástrojů pro administraci a management testování a nástrojů pro monitorování testování podle potřeby
- Automatizování testů (může být podpořeno vývojářem nebo expertem pro automatizaci testů) (Müller, 2007)

3.12 Automatizované testování uživatelského rozhraní

Testování uživatelského rozhraní (UI – User Interface) je často bráno jako manuální činnost, protože je s ním spojena spousta problémů. Vzhledem ke snadné opakovatelnosti tohoto testování se vývojáři a testeři často uchylují k nástrojům, které sami vytváří kód pro automatické testy. Nicméně takto vytvořený kód je velmi těžké číst, jelikož je automaticky

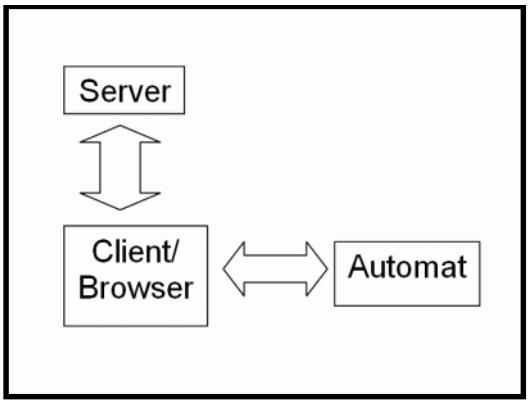
generován daným nástrojem a tím pádem je ho velmi těžké udržovat. Z těchto důvodů jsou potom takové testy často mazány. Nutno také podotknout, že myšlenka toho, že manuální testy skončí s příchodem automatizace, je mylná. Automatické testy nikdy plně nenahradí ty manuální a ani to není jejich úkolem (McWherter, 2010).

Dobří vývojáři či testeři jsou schopni zvážit význam automatického testování a jeho výhod. Tvorba automatických uživatelských testů není složitá, je ovšem složité vytvořit takové testy, které opravdu přinesou hodnotu. Některé firmy začnou s automatizací z toho důvodu, aby vytvořily pouze „smoke“ testy, tedy testy, které projdou základní funkčnost systému. Po úspěšných smoke testech jsou potom připraveni na manuální testování. Jiné společnosti zas automatizaci využívají na kontrolu každého prvku ve své aplikaci a pokrytí každého aspektu a to včetně barvy pozadí. Ani jeden z těchto přístupů není špatný, je pouze potřeba se správně rozhodnout, co chceme automatizací získat (McWherter, 2010).

Automatizované testování aplikací, které jsou typu „klient – server“, můžeme rozdělit do dvojího typu – simulace uživatele a simulace stroje.

Oba typy ukazují případ, kdy je potřeba na nějakém stroji spustit robota, jenž bude testy provádět. Rozdíl je v tom, co robot bude simulovat. Buď uživatele dané aplikace, případně stroj, který uživatel používá (Čermák, 2011).

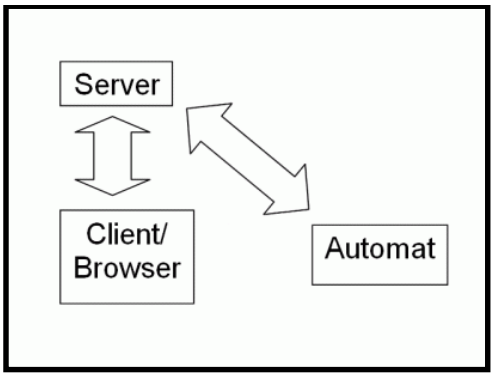
Obrázek č. 7 Simulace uživatele



Zdroj: http://www.cleverandsmart.cz/

V případě simulace uživatelee se stroj bude chovat právě jako uživatel. Bude tedy provádět dané operace takovým způsobem, jako by je prováděl sám uživatel systému. To, čemu musí automatizace v tomto případě porozumět, je obsah zobrazovaných údajů. Uživatel ihned vidí, jaký je výsledek. Automat však musí správně identifikovat prvek či oblast zobrazování výsledku. To je těžké zejména v případě tlustých Windows klientů. Hlavním problémem je ovšem krátká životnost takovýchto testů. Pokud se změní rozvržení stránky, je dost pravděpodobné, že si to vyžádá i přepsání kódu testu (Čermák, 2011).

Obrázek č. 8: Simulace stroje



Zdroj: http://www.cleverandsmart.cz/

V případě simulace stroje se klient obchází. Testuje se čistě funkcionalita serveru, tedy volání serverových funkcí. Requesty (volání na server) jsou simulovány jako http Request, Remote Function Call, WebService Call nebo jiná technika. Užitá technika je závislá na typu klienta a typu prováděného testu. Výhoda tady je to, že testy jsou odolné vůči změnám v grafickém uživatelském rozhraní. Nicméně, zcela logicky, je zde nevýhodou to, že se netestuje strana klienta. Pokud se tedy např. testují validace vstupů právě na straně klienta či jiné důležité činnosti, tak zůstávají neotestovány (Čermák, 2011).

3.12.1 Důležitost automatického UI testování

Je jisté, že UI testování je důležitou součástí vývoje softwaru. Unit testy jsou jednoduše orientovány na nějakou izolovanou metodu s jasně definovanými vstupy a výstupy. A tak je

při nich jednodušší rozumět interakcím a průchodem testů. S UI testy ztrácíme tuto schopnost. To tedy znamená, že při jejich psaní musíme toto kombinovat s myšlenkou, jak UI testy strukturovat. Je totiž samozřejmě složitější rozumět vnitřním částem systému, pokud se jedná o černou skříňku (McWherter, 2010).

Největší výhodou automatických testů je to, že dokážou identifikovat rozbití systému. Pomocí unit testů se snažíme mít co největší pokrytí kódu, zatímco UI automatické testy přináší největší hodnotu tak, že zachytí rychle varovné známky nějakého nebezpečí. Pokud tuto úlohu testy splní, zjistíte, že se vyplatí do nich (a jejich údržby) investovat čas a peníze (McWherter, 2010).

Díky této dávce jistoty (unit testy + UI automatické testy + integrační testy) mají testeři ten komfort, že se mohou věnovat testování toho, na čem právě pracují vývojáři. Pokud vývojáři a testeři pracují zároveň na právě vyvíjených částech systému, přináší tím velkou produktivitu. Testeři nejsou tolik vázáni na regresní testování, protože jim „kryje záda“ alespoň částečně právě automatizace. Tento argument je relevantní speciálně pro agilní vývoj softwaru (např. Scrum). Základem Scrumu je soustředěnost celého týmu na právě vyvíjenou věc a pokud se testeři musí většinou času věnovat regresnímu testování, nemohou se plně soustředit na části systému vyvíjené právě teď a nemohou tedy produkt tlačit dopředu (McWherter, 2010).

Poté, co je uživatelské rozhraní pokryto automatickými testy, další vhodnou fází pro pokročení je zautomatizovat testy napříč všemi platformami a webovými prohlížeči. Tím je získána plná výhoda automatických testů a plně se využije investice, která byla do automatizovaného testování vložena (McWherter, 2010).

3.12.2 Problémy s UI automatizací

Testování uživatelského rozhraní s sebou přináší spoustu problémů. V kruzích testerů se jedná o velice ožehavé téma. Spousta lidí si myslí, že to je ztráta času a jejich argumentem je to, že s testováním uživatelského rozhraní přichází také spousta potřebného času na

údržbu, protože tyto testy jsou velice křehké a je tedy zapotřebí, aby je daný vývojář neustále udržoval a upravoval. Tyto testy jsou typem testování černé skříňky, tedy že tester zná pouze vstupy a k nim dané výstupy a neřeší to, co se děje uvnitř. Nelze pomocí nich ověřit to, zda je daný kód efektivně napsaný. Problémy s UI automatizací mohou být následující:

- ***Křehkost kvůli úpravám.*** Někteří vývojáři, když zjistí, že se dají automatizovat testy UI, tak okamžitě začnou nahrávat pomocí nahrávacích nástrojů typu Selenium, VisualStudio, atd., které sledují jejich prokliky v aplikaci. Potom se ale může stát, že přijdou do práce, spustí testy a všude bude červeno, kvůli nalezeným chybám. A protože tyto testy byly bezmyšlenkovitě nahrávány pomocí „rekordérů“, tak nemusejí být vůbec opětovně upravitelné do rozumně spustitelné podoby. Takové případy se stávají, pokud vývojáři poupraví aplikaci a jednotlivé kroky již nesedí. Vyhnout se tomu dá použitím nástrojů, kde se zapisuje kód s tím, že testy budou mít strukturu a nebudou psány bezmyšlenkovitě.

- ***Problémy s časováním.*** Tento problém může nastat v případě, kdy vývojáři k testování nepoužívají množinu dat, která je použita na produkci a slouží pouze k testovacímu účelu. Systém se díky těmto datům nezachová tak, jak se očekávalo a tak vznikne timeout. Na produkci se to nestane, nicméně samotné testy jsou v tomto případě předurčeny k tomu, aby vypsaly chybu.

- ***Těžko testovatelné elementy.*** UI testování je obtížné. Nicméně testování některých konkrétních elementů je ještě těžší. Například testování některého labelu, který se postupně objeví a potom zase zmizí s nějakým časovým intervalem je těžší, než testování obyčejného labelu. Proto je potřeba, aby testeři / vývojáři, jež se věnují UI testování, měli schopnost myslet kreativně.

- ***Těsné spojení vůči ostatním částem systému.*** Mnoho let se různé odborné zdroje snaží naučit testery / vývojáře psát testy, které nejsou vázány na ostatní části systému. Nicméně se stále děje to, že business logika se promítá do logiky testování UI. Takové testování je velmi obtížné a pro lidi, kteří testy píší, je mnohem příjemnější psát jednotlivé testy odděleně.

- ***Nevyžádané změny.*** Webové aplikace mohou obsahovat nějaké události, které se stanou, aniž by je vyvolal sám uživatel. Takovým příkladem může být to, když se aplikace sama po nějaké době (uživatelově neaktivitě) odhlásí, což se dá těžce simulovat.

- ***Různé cesty.*** Jednou z vlastností webových aplikací je to, že se do každé části aplikace dá přistoupit různým způsobem. Uživatel může přistupovat z menu, změnou URL adresy, nějakým tlačítkem a nebo třeba externím odkazem. Je potřeba si definovat, kolika různými přístupy se do aplikace dá přistupovat a kolika způsoby se v ní dá pohybovat. Jednotlivé přístupy do částí aplikace jsou acyklické a může se tedy jedna to velké množství. Nastává otázka, zda je potřeba otestovat všechny možnosti či ne, případně jak velkou množinu.

- ***Nekonečné sekvence vstupů.*** Aplikace umožňuje uživatelům vkládat data, do formuláře, v pořadí jakém sami chtějí. Zde nastává otázka, zda testovat to, co se stane, pokud uživatel vyplní pole 2 dříve než pole 1, atd. To může generovat obrovské množství testů, jejichž počet exponenciálně roste.

- ***Náhodná selhání.*** Pří testování uživatelského rozhraní se objevují často selhání, která je třeba velmi těžké reprodukovat. Z těchto selhání se potom stávají chyby, které jsou odkládány nebo se testy pouštějí s tím, že tam jsou a ví se o nich. Někdy dokonce vývojáři takový test smažou, čím snižují hodnotu celkového testu (McWherter, 2010).

3.13 Selenium

Selenium je balík open source nástrojů pro automatizované testování webového uživatelského rozhraní na různých platformách. Selenium má schopnost řídit Internet Explorer, Firefox, Operu, Safari, atd. Selenium využívá JavaScript and Iframes na vkládání automatizačního stroje do prohlížeče. Projekt Selenium začal Jason Huggins v roce 2004.

Balík Selenium se stal velmi rychle populárním pro testování UI a to z důvodů široké podpory webových prohlížečů a podpory pro tvorbu testů v různých jazycích (Java, Python, C#, Ruby). Balík se skládá ze tří částí: Selenium IDE, Selenium Remote Control a Selenium Grid. V době, kdy projekt Selenium vznikl, existovalo pouze pár testovacích nástrojů. Například Quick Test Pro a Test Director, které byly vytvořeny společností Mercury, jež byla odkoupena Hewlett Packard v roce 2006. Tyto nástroje byly drahé, těžko naučitelné a skripty bylo těžké udržovat (McWherter, 2010).

3.13.1 Selenium IDE

Selenium IDE je doplněk do Firefoxu, který může být využit pro vytváření a spouštění testů ve webovém prohlížeč. Selenium IDE má podporu pro nahrávání nahrávání činnosti v prohlížeči, nastavování bodů pro pozastavení a přehrávání testů (McWherter, 2010).

Obrázek č. 9: Selenium IDE

Zdroj: Testing ASP.NET (McWherter, 2010)

Obrázek ukazuje jednoduchý příklad testu, který má za úkol navigovat se na google a napsat asp.net.

3.13.2 Selenium RC

Selenium Remote Control (RC) se skládá ze serverové části a klientských knihoven, což umožňuje tvorbu testů v jazyku podle libosti a testy mohou být integrovány do již existujícího testovacího frameworku (McWherter, 2010).

Selenium RC obsahuje Selenium server, který se chová jako proxy server (tedy jako prostředník mezi prohlížečem a testovanou webovou aplikací) pro webové prohlížeče a může spustit seleniové testy. Selenium nástroje obsažené v Seleniu RC se automaticky spustí a zavřou prohlížeč. Následující kód vytvoří spojení k Selenium serveru a nanavigují tak na danou adresu v testu (McWherter, 2010).

Selenium Server

je jedna ze dvou částí Selenium RC. Je napsaná v jazyku Java. Server akceptuje příkazy odeslané prohlížeči přes HTTP a prohlížeče tedy otevře a následuje tyto příkazy (McWherter, 2010).

Klientské knihovny

Jádro Selenia využívá jazyk zvaný Selenese k ovládání prohlížeče. Klientské knihovny Selenia RC jsou obaleny do tohoto jazyka a příkazy napsané v použitém jazyce budou přeloženy do Selenese k transportování přes HTTP. Tyto knihovny tedy umožňují to, aby se koncový uživatel API nemusel učit specifickou Selenese syntaxi, nicméně je důležité se naučit, jak tyto příkazy pracují. Příkazy jsou tří typů:

- ***Actions*** – příkazy pro manipulaci stavů aplikace, např. kliknutí na odkaz

- ***Accessors*** – ukládání výsledků o stavu aplikace

- ***Assertions*** – tyto příkazy kontrolují stav aplikace a porovnávají ho s tím, co se očekává (McWherter, 2010)

Lokátory elementů

Při vytváření automatických testů velkou část času zabere označení daného prvku v HTML pro potvrzení toho, zda test prošel nebo ne. Selenium RC má koncept toho, jak takové elementy označit a říct Seleniu, na který prvek uživatel ukazuje. Naučit se správně daný prvek specifikovat je velice důležité pro budoucí údržbu testů. Selenium podporuje tyto typy označování jednotlivých elementů:

- ***Identifikátor*** – vybere element s daným ID, pokud takové ID není nalezeno, vybere první element, jehož atribut Jméno je ID

- ***ID*** – vybere element, který je specifikován daným ID

- ***Jméno*** – vybere element s daným atributem Name (jméno)

- ***DOM*** – vybere prvek specifikovaný nějakým řetězcem

- ***Xpath*** – jedná se o jazyk pro vyhledávání částí v XML dokumentu

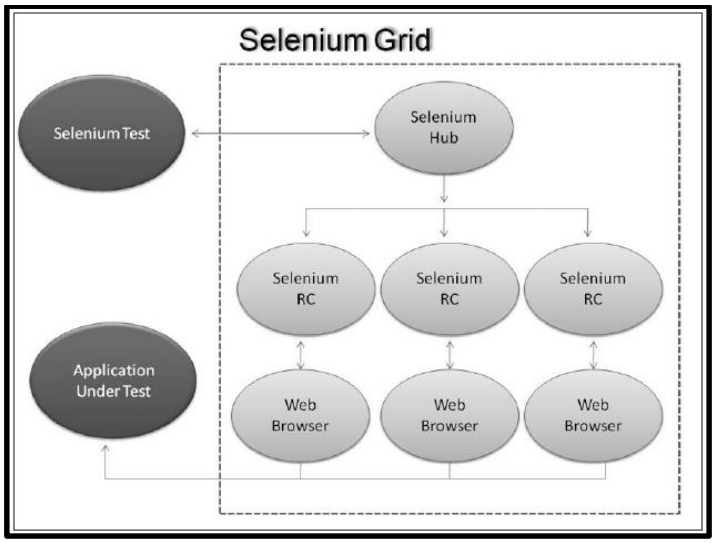
- ***Odkaz*** – vybere odkaz s daným jménem

- ***CSS*** – vybere element používající daný CSS prvek (McWherter, 2010)

3.13.3 Selenium Grid

Selenium Grid je nástroj, který umožňuje Selenium testy vykonávat paralelně. To umožní ušetřit spoustu času, respektive zrychlit dosažení výsledku, který by mohl trvat při velkých testovacích celcích několik hodin až dní. Testy tak mohou probíhat na více webových prohlížečích (McWherter, 2010).

Obrázek č. 10: Architektura Selenium Grid



Zdroj: Testing ASP.NET (McWherter, 2010)

Na obrázku výše je vidět architektura Selenium Grid, která je oproti typické architektuře Selenia odlišná v tom, že je zde navíc centrální server, tzv. Selenium Hub. Ten umožňuje udržovat testovací relace a směrování požadavků mezi testem a danou instancí Selenium RC (McWherter, 2010).

4 Vlastní práce

4.1 Popis společnosti Socialbakers a.s.

Obrázek č. 11: Logo společnosti Socialbakers a.s.

Zdroj: Socialbakers.com

IČ: 29098271
Obchodní firma: Socialbakers a.s.
Sídlo: Pod Všemi svatými 427/17, Severní Předměstí, 301 00 Plzeň
Právní forma: Akciová společnost
Představenstvo: Jan Řežáb (předseda představenstva), Lukáš Maixner a Martin Homolka

Socialbakers je společnost, která se zabývá managementem a analýzou dat na sociálních médiích a pomocí svých nástrojů, které vyvíjí, pomáhá ostatním firmám tyto aspekty rozvíjet a udržovat. Společnost byla založena Janem Řežábem, Martinem Homolkou, Jiřím Vovesem a Lukášem Maixnerem v roce 2008, tehdy ještě jako společnost Candytech. V roce 2009 se společnost přejmenovala na Facebakers a V roce 2010 se transformovala do společnosti s názvem Socialbakers.

Dnes má Socialbakers více než 300 zaměstnanců složených z více než 30 národností a to v 11 pobočkách po celém světě. Společnost vznikla v Plzni, dnes má ovšem hlavní sídlo v Praze a pobočky v New Yorku, Londýně, Singapuru, Istanbulu, Dubaji, Mnichově, Sao Paolu, Mexico City.

V roce 2014 společnost Socialbakers získala obrovské finanční prostředky na rozvoj od investičních společností Index Ventures a Earlybird. Jednalo se o 26 milionů dolarů (cca 517 milionů korun).

Portfolio Socialbakers obsahuje více než 2500 klientů ze 100 zemí světa. Na společnost spoléhají například firmy Samsung, Danone, Lufthansa, Disney, KLM, Henkel, NHL či Lamborghini. S milionem návštěv měsíčně je Socialbakers.com dnes největším světovým portálem, který nabízí marketingovým expertům a agenturám široké spektrum dat a analytických informací ze sociálních sítí.

Co se týká portfolia produktů, tak mezi nejdůležitější, tedy ty, co spadají do Socialbakers Marketing Suite, patří:

- Analytics – více popsán v následující kapitole.

- Builder – velmi komplexní produkt, který slouží k manažování profilů a publikování příspěvků v reálném čase. Lze přes něj sledovat příspěvky zvolených stránek z různých sociálních sítí. Také je přes tento nástroj možné ihned odpovídat na dotazy odběratelů. Hlavní náplní je ovšem publikování příspěvků pro různé profily na různých sociálních sítích a to v jednom čase či s nějakou prodlevou. To vše lze dělat v jednom pohledu, který je sestaven z několika kanálů.

- Ads – nástroj, který slouží pro efektivní, intuitivní a multi-platformní řešení, co se týká vytváření, manažování a optimalizaci reklamy na sociálních sítích.

- Listening – tento produkt slouží, jak název napovídá, k odposlouchávání toho, co se kde zmíní na Twitteru nebo Facebooku. Samotný produkt obsahuje spoustu pomocných funkčností a grafických výstupů, nicméně samotné gró je právě ve zjištění toho, jak moc se na sociálních sítích „mluví“ o daném tématu. Pro firmy to může sloužit např. jako nástroj pro zjištění síly PR.

4.2 Socialbakers Analytics

Jedním z největších a nejdůležitějších produktů společnosti Socialbakers je analytický nástroj Analytics, který je k nalezení na adrese *analytics.socialbakers.com.* Aby nedošlo k případnému nedorozumění, tak v době psaní této diplomové práce je na zmíněné adrese k nalezení starší verze tohoto produktu, kterou se dále práce nezabývá. Sekce, pro kterou bude uvedena tvorba testu, pochází již z nové verze produktu, která je k nalezení (v době psaní této práce) na adrese *new.analytics.socialbakers.com*, jež je zpřístupněna pouze určitému procentu zákazníků. Nicméně se v brzké době počítá s nasazením nové verze pro všechny. Zákazníci si tak budou moci vybrat, jakou verzi produktu budou chtít používat.

Aplikace je primárně rozdělena po jednotlivých platformách (+ souhrnná sekce):

- ***Dashboard*** – stránka, na kterou je uživatel po přihlášení automaticky přesměrován. Jedná se souhrn několika metrik za všechny stránky sociálních sítí, které daný uživatel sleduje.

- ***Facebook*** – několik sekcí obsahujících spoustu různých metrik, od jednoduchých po složitější, pro platformu Facebook. Uživatel zde může sledovat grafy jako růst/pokles fanoušků, jejich aktivitu, jejich interakci, jejich nejoblíbenější příspěvky, atd.

- ***Twitter*** – viz Facebook (s tím rozdílem, že je vše pro Twitter profily)

- ***Youtube*** – viz Facebook (s tím rozdílem, že je vše pro Youtube kanály)

- ***VKontakte*** – viz Facebook (s tím rozdílem, že je vše pro VK stránky)

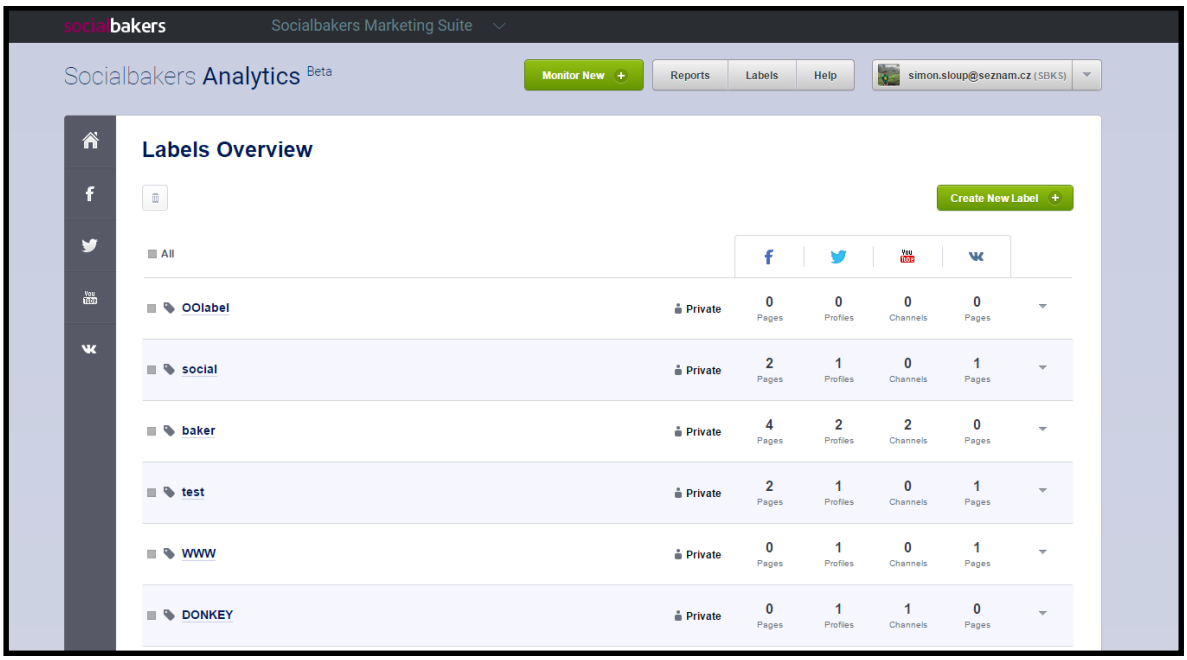
Dalšími podstatnými součástmi produktu Analytics je Compare a Multicompare, díky nimž lze lehce porovnávat metriky jednotlivých stránek, kdy se do grafu promítnou výstupy obou z nich, v případě multicomparu až deseti stránek dané sociální sítě.

Mimo rámec rozdělení aplikace po jednotlivých platformách jsou sekce „Reports“ a „Labels“. Díky sekci „Labels“ lze označovat skupiny stránek sociálních sítí pod určité štítky (např. značky automobilů). Tato sekce bude dále detailněji rozebrána, jelikož na ní bude provedena ukázka tvorby testů. Sekce „Reports“ ukládá a eviduje PDF, PNG a XLSX reporty, které uživatel stáhl, případně ty, které jsou automaticky zasílány na mail v zadaných časových intervalech. Dále aplikace obsahuje samozřejmě správu uživatelů, týmů a spoustu jiných součástí.

4.3 Sekce „Labels“ jako ukázka pro tvorbu testu

Tato sekce umožňuje „onálepkovat“ stránky sociálních sítí a tím si uživatel může seskupit stránky např. s podobným tématem, všechny svoje stránky či jakkoliv ho napadne. K manažování těchto štítků slouží sekce „Labels Overview“, která byla vybrána jako ukázková sekce pro tvorbu testu.

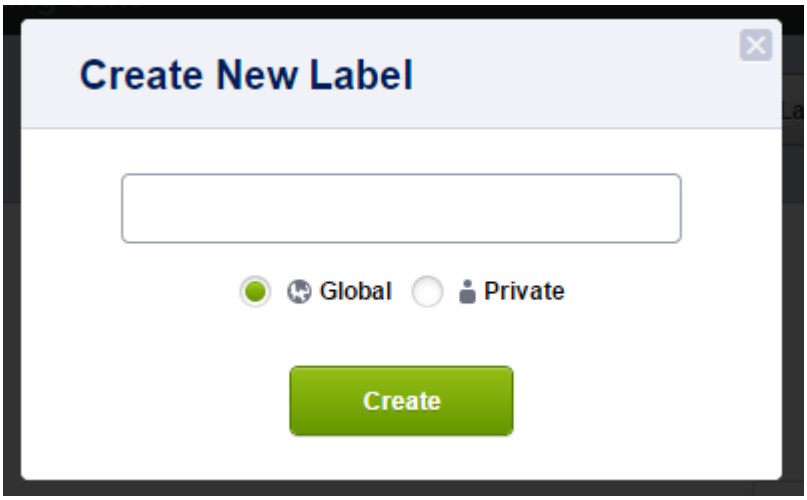
Obrázek č. 12: Výřez z aplikace Analytics - „Labels Overview“



Zdroj: vlastní

Obrázek výše uvedený ukazuje otevřenou sekci Labels, tedy přesněji pojmenovanou „Labels Overview“. V řádcích vidíme názvy jednotlivých labelů (např. baker, test, atd.). Každý štítek (label) může být buď soukromý (private) nebo veřejný (global). To označuje, zda ho může používat pouze daný uživatel či všichni uživatelé v daném účtu. Dále je v daném řádku zobrazena informace o tom, kolik stránek z jednotlivých sociálních sítí label obsahuje. Na konci řádku je šipka, která rozbaluje malé menu umožňující label mazat, nějak pozměnit či ho převést na globání (veřejný) label v případě, že je privátní. V hlavičce této sekce je tlačítko pro vytváření štítku a tlačítko pro mazání, které se aktivuje v případě, že je zaškrtnut jeden nebo více checkboxů u daných labelů.

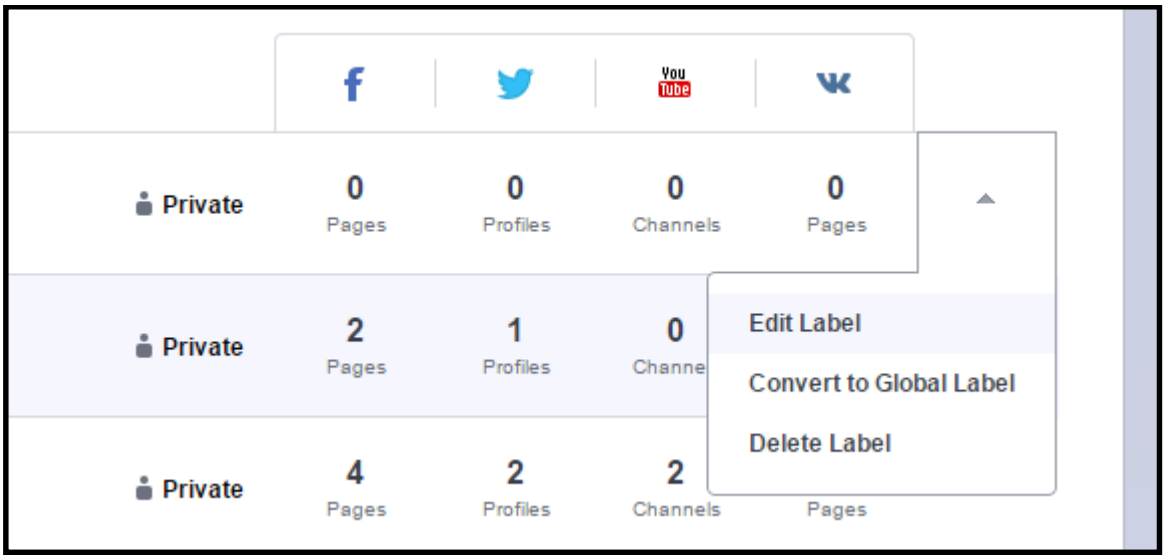
Obrázek č. 13: Výřez z aplikace Analytics – modál tvorby štítku



Zdroj: vlastní

Na obrázku je vidět modální okno, které se otevře při kliknutí na tlačítko „Create New Label“. Umožňuje vybírat mezi globálním a privátním typem labelu. Je potřeba otestovat všechny možnosti. Nezapomínat např. na křížek, který modál zavře, atd.

Obrázek č. 14: Výřez z aplikace Analytics - dropdown



Zdroj: vlastní

Výřez výše uvedený prezentuje dropdown, který se rozbalí při kliku na šipku u příslušného labelu. Může obsahovat buď 2, nebo 3 možnosti. V případě, že se jedná o privátní label, obsahuje právě na obrázku uvedené možnosti. Tedy editovat label (zobrazí se modální okno pro změnu jména), převést label na globání a nebo label smazat. Pokud by se jednalo o globální label, logicky chybí možnost konvertovat štítek z privátního na globální.

4.4 Popis metody vývoje

Metoda vývoje softwaru probíhá v rámci agilní metody Scrum. Jedná se o čtyři vývojové týmy pracující na projektu Analytics. Počet programátorů se pohybuje okolo pěti. Každý tým vede Scrum Master a ke každému týmu je přiřazen jeden QA Engineer (tester), který sedí společně s týmem v jedné místnosti, aby byla co nejvíce naplněna myšlenka Scrumu.

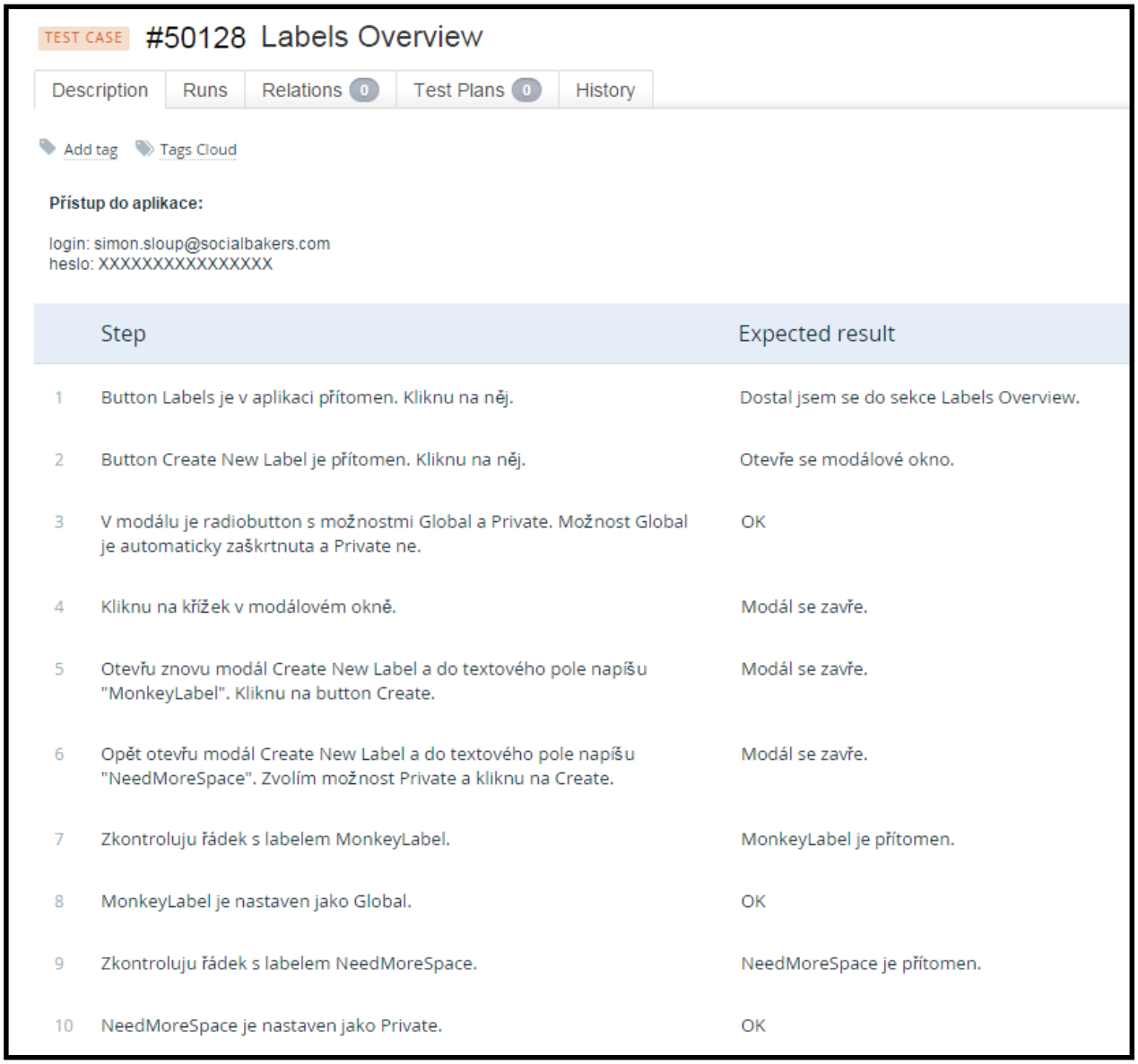
Sprinty trvají dva týdny. Sprint je doba, za kterou by měl každý tým dodat ten počet user stories a opravených bugů, který si na začátku toho sprintu zadal. User stories jsou zadání částí systému či jiných požadavků, které je potřeba naprogramovat, otestovat a nasadit do příslušné verze produktu (release). Proces, kterým prochází user story je takový, že je nejdříve v backlogu nějakého týmu. Následně je přiřazen do příslušeného sprintu (záleží na prioritě). Jakmile je ve sprintu, nějaký programátor si ho přiřadí a začne na něm pracovat. Jakmile je hotovo, je předán do code review, kde jiný programátor kontroluje kód po svém kolegovi. Pokud je to v pořádku, je předán do QA, kde si ho vezme tester a otestuje. Pokud je v pořádku, pošle ho do akceptace. Pokud není, vrátí ho programátorovi. V akceptaci si ho vezme product owner (člověk, který dohlíží na produkt a je za zodpovědný) a schválí či vrátí k přepracování. Pokud je schválen, je nasazen na příslušnou release větev, kde je znovu otestován.

Co se týká testovacího týmu, je zhruba jedenáctičlenný. Zastřešuje ho QA Manager. Tři lidé se starají o psaní automatizačního frameworku. Zbytek je přiřazen ke svým týmům, kde provádí jak manuální testování v rámci sprintu, tak také psaní automatizovaných testů. Místem, kde se střetávají spolu testeři, je testování regresních testů před nasazením nové verze produktu, případně po nasazení, kdy se pracuje na after deploy testu, který testuje, že je v pořádku již nasazený produkt.

4.5 Vlastní tvorba manuálního testu

Manuální testy, tedy jednotlivé Test Cases (testovací případy), které jsou následně shlukovány do Test Plan (testovací plán) a ty jsou následně spouštěny jako Test Plan Run, se spolu s celým Scrum procesem manažují ve webové aplikaci TargetProcess (dále TP). Uvedený praktický příklad napsaného testovacího případu mohl být uveden čistě textově, nicméně pro lepší přehled a ukázku jsou obrázky vyfoceny právě z nástroje TP.

Obrázek č. 15: Testovací případ: část 1



TEST CASE #50128 Labels Overview

Description | Runs | Relations 0 | Test Plans 0 | History

Add tag | Tags Cloud

Přístup do aplikace:

login: simon.sloup@socialbakers.com
heslo: XXXXXXXXXXXXXXX

	Step	Expected result
1	Button Labels je v aplikaci přítomen. Kliknu na něj.	Dostal jsem se do sekce Labels Overview.
2	Button Create New Label je přítomen. Kliknu na něj.	Otevře se modálové okno.
3	V modálu je radiobutton s možnostmi Global a Private. Možnost Global je automaticky zaškrtnuta a Private ne.	OK
4	Kliknu na křížek v modálovém okně.	Modál se zavře.
5	Otevřu znovu modál Create New Label a do textového pole napíšu "MonkeyLabel". Kliknu na button Create.	Modál se zavře.
6	Opět otevřu modál Create New Label a do textového pole napíšu "NeedMoreSpace". Zvolím možnost Private a kliknu na Create.	Modál se zavře.
7	Zkontroluju řádek s labelem MonkeyLabel.	MonkeyLabel je přítomen.
8	MonkeyLabel je nastaven jako Global.	OK
9	Zkontroluju řádek s labelem NeedMoreSpace.	NeedMoreSpace je přítomen.
10	NeedMoreSpace je nastaven jako Private.	OK

Zdroj: vlastní tvorba

Pro správné srovnávání automatického a manuálního testu, byl ten manuální napsán přesně tak podrobným způsobem, jakým stylem je tvořen ten automatický. Tedy každý automatizovaný příkaz, který je vykonávám v selenium testu, je také přepsán do manuálního testu. Ovšem samozřejmě s tím, aby kroky manuálního testu dávaly smysl.

Obrázek č. 16: Testovací případ: část 2

11	Zaškrtnu checkboxy u svých labelů MonkeyLabel a NeedMoreSpace.	Checkboxy zaškrtnuty.
12	Kliknu na nyní zaktivovaný button Delete.	Otevře se modál.
13	V modálu kliknu na button Remove.	Modál se zavře.
14	V sekci zkontroluju ikonky pro sociální sítě (Facebook, Twitter, Youtube, VK)	Ikonky přítomny.
15	Vytvořím nový label s názvem "SpaceFunk" a nastavím ho jako private.	Label se vytvořil.
16	V řádku labelu SpaceFunk kliknu na šipku na konci řádku.	Rozbalí se dropdown.
17	V dropdownu zvolím možnost Edit label.	Otevře se modál.
18	V textovém inputu vymažu název SpaceFunk a zadám SpaceMonkey. Kliknu na button Save.	Modál se zavře a label je přejmenován.
19	U labelu SpaceMonkey kliknu na šipku pro dropdown a zvolím "Convert to Global Label".	U labelu se změnilo Private na Global.
20	Opět kliknu na šipku u labelu SpaceMonkey a zvolím možnost delete.	Label se vymazal.

Zdroj: vlastní tvorba

V druhé části manuálního testu je vidět zbytek příkazů, které jsou konzistentní s automatickým testem. Výsledná podoba sekce Labels Overview vypadá tak, že žádný testem vytvořený label v sekci nezůstal.

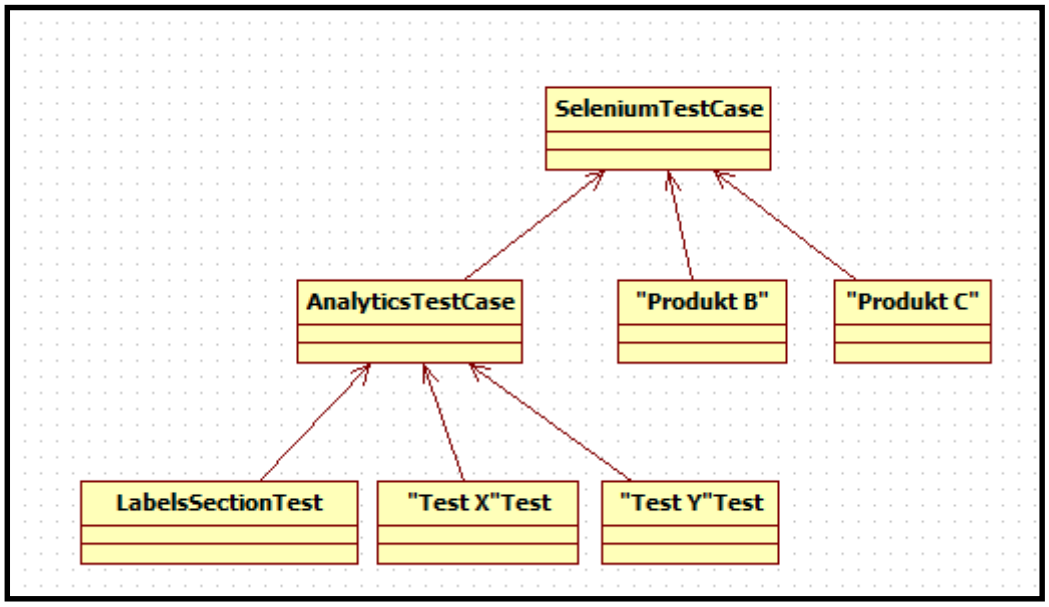
Samotný TC v žádném kroku neobsahuje vstup a přihlášení do aplikace, jelikož se jedná o testovací případ, který navazuje na nějaký předchozí. V automatickém testu bude zmíněn pro lepší ukázku i vstup a přihlášení do aplikace.

4.6 Vlastní tvorba automatického testu

Tvorba automatického testovacího případu (Test Case) pro popsanou aplikaci je provedena pomocí nástroje Selenium. Používaný jazyk je PHP, konkrétně jsou testy psány v jazyku PHPUnit. Jedná se o testovací framework pro programovací jazyk PHP. PHPUnit je implementací architektury xUnit.

Konkrétní použitý příklad je automatickým synonymem pro manuální test popsaný v předešlé kapitole. Aby šlo dané alternativy porovnat, je potřeba mít přesně stejné kroky v obou případech. Proto zde bude přesně oněch 20 kroků manuálního testu popsáno pomocí automatického seleniového testu.

Obrázek č. 17: Architektura automatických testů



Zdroj: vlastní tvorba

Aby se dalo porozumět některým příkazům, i když v tomto případ není použit moc velký rozsah, je potřeba nastínit alespoň mírně architekturu testů. Nejnižší třídou jsou jednotlivé testy, v našem případě LabelsSectionTest, který bude následně popsán. Nad ním třída AnalyticsTestCase, jenž obsahuje všechny předdefinované funkce potřebné pro rychlejší tvorbu psaní jednotlivých testů pro produkt Analytics. AnalyticsTestCase má svojí nadtřídu SeleniumTestCase, které obsahuje ještě obecnější funkce, které mohou být použity pro všechny produkty. Může se jednat například o přihlášení do aplikací, které je stejné pro všechny. Nad popsanou architekturou už je samotný PHPUnit framework.

Pro identifikaci html elementů je použit jazyk XPath. Jedná se o celkem jednoduchý jazyk, u kterého se ovšem může vyskytnout problém při identifikaci jednotlivých elementů, jejichž „přístupová cesta“ může být shodná a poté (obzvláště pro začátečníky) nastává problém s jasně danou identifikací. Cesty jsou potom složité a jejich údržba může stát hodně času. V tomto případě se opravdu vyplatí dohoda s vývojáři o pravidlech označování elementů.

Následující příklad je tedy automatizovanou alternativou k manuálnímu testu v předchozí kapitole. Jednotlivé části kódu jsou označeny rámečky a popsány texty pod nimi. Automatizovaný test pro sekci Labels:

```
<?php

include 'bootstrap.php';

class LabelsSectionTest extends AnalyticsTestCase {

public function testLabelsSection() {
```

Vzhledem k použitému jazyku začíná každá funkce tagem <?php. Bootstrap.php nám ve zkratce připraví celé prostředí, tedy umožní používat základní knihovny WebDriveru. Následuje pojmenování třídy (testu), které je podtřídou AnalyticsTestCase obsahující základní nadefinované funkce. Poté již začíná tvorba funkce, která zde bude pouze jedna, protože naše ukázka obsahuje pouze krátký test, který prověřuje funkci jedné menší části aplikace. Ve většině případů ovšem jedna třída (test v našem případě) obsahuje několik funkcí.

```
$this->driver->setText("\n\nTest Labels Overview");

$this->driver->load($this->getHomeURL());
```

Metoda „setText“ vypisuje ve všech výstupech námi zadaný text. Slouží k rozeznání toho, co jsme testovali, abychom mohli identifikovat následnou chybu. Metoda „load“ volá domácí URL adresu, v našem případě se samozřejmě jedná o adresu testované aplikace.

```
$this->driver->get_element("//a[@id='fb-login']", "Sign in with
Facebook")->click();

$this->logInWithRedirect("simon.sloup@socialbakers.com",
"XXXXXXX");
```

Metoda „get_element“ slouží k tomu, aby byl identifikován daný element a následně se nějak využil. V tomto případě se jedná o tlačítko pro přihlášení do aplikace, které se objeví po načtení zadané URL adresy. První část závorky je XPath cesta k tlačítku. Druhá část závorky je pouze informační popis opět pro lépe čitelný výpis. Po úspěšném nalezení elementu následuje akce „click“, tedy kliknutí na daný button.

Nutno připomenout dodatek z předchozí kapitoly, že tyto kroky nemají svou alternativu v manuálním testu. Zde by prakticky také nebyly zahrnuty, jelikož tento automatický test navazuje na nějaký předchozí, kde by již přihlášení bylo provedeno a zde by se kvůli úspoře času již neopakovalo. Pro lepší ukázku automatizace však bylo zahrnuto do této práce.

```
$this->driver->get_element("//a[contains(@href, '#/labels') and
contains(@class, 'an-btn')]","Labels Button")->click();
```

Posledním krokem pro vstup do sekce Labels je kliknutí na button Labels na hlavní stránce aplikace. Tím jsme pokryli první krok manuálního testovacího případu (dále jen TC).

```
$this->driver->get_element("//div[contains(@class, 'an-addpage-btn
an-btn-primary')]","Create New Label Button")->click();

$this->driver->assert_element_present("//div[contains(@class,
'key-global')]//div[contains(@class, 'an-checkbox an-
checked')]","radiobutton global checked");

$this->driver->assert_element_present("//div[contains(@class,
'key-private')]//div[contains(@class, 'an-checkbox an-
notchecked')]");

$this->driver->get_element("//i[contains(@class, 'icon-
close')]","icon close")->click();
```

Tato část kód kontroluje modáln9 okno, které slouží pro tvorbu labelu. Okno je uvedeno v kapitole, kde je popsána sekce Labels v aplikaci. První příkaz klikne na daný button, čímž se otevře okno. Dále je zkontrolováno, že je přítomen radiobutton „global“ a je deafultně zaškrtnut. Třetí příkaz kontroluje nezaškrtnutý radiobutton „private“. A poslední příkaz zavře dialog křížkem.

```
$this->driver->get_element("//div[contains(@class, 'an-addpage-btn
an-btn-primary')]","Create New Label Button")->click();

$this->driver->get_element("//input[contains(@class, 'an-textbox-
input') and contains(@name, 'labelTitle')]","Create New Label
text")->send_keys("MonkeyLabel");

$this->driver->get_element("//div[contains(@class,'an-modal-
wrapper')]//div[contains(@class,'an-btn-primary')]","Create Button
in modal")->click();
```

Těmito třemi příkazy se vytvoří label s názvem „MonkeyLabel“. První příkaz opět otevře modáln9 okno, další do textového pole vloží text MonkeyLabel a třetí příkaz klikne na button „Create“.

```
$this->driver->get_element("//div[contains(@class, 'an-addpage-btn
an-btn-primary')]","Create New Label Button")->click();

$this->driver->get_element("//input[contains(@class, 'an-textbox-
input') and contains(@name, 'labelTitle')]","Create New Label
text")->send_keys("NeedMoreSpace");

$this->driver->get_element("//div[contains(@class, 'an-radiogroup-
item key-private')]//div[contains(@class, 'an-checkbox an-
notchecked')]","Private radiobutton")->click();

$this->driver->get_element("//div[contains(@class,'an-modal-
wrapper')]//div[contains(@class,'an-btn-primary')]","Create Button
in modal")->click();
```

Stejné jako v předchozím případě. Je vytvořen další label s názvem „NeedMoreSpace“. Je zde pouze jeden příkaz navíc, který zaškrtne radiobutton „private“, aby byl tento label soukromý.

```
$this->driver-
>assert_element_present("//div[descendant::em[contains(text(),
'MonkeyLabel')] and contains(@class,
'row')]//span[contains(@class, 'an-label-global')]", "global label
control");

$this->driver-
>assert_element_present("//div[descendant::em[contains(text(),
'NeedMoreSpace')] and contains(@class,
'row')]//span[contains(@class, 'an-label-private')]", "private
label control");
```

Tyto dva příkazy slouží ke kontrole již vytvořených dvou labelů. Konkrétně se kontroluje to, zda mají správné příznaky globální či privátní. Zde je již použita komplikovanější XPath konstrukce. Je to z důvodu toho, že vytvořené labely jsou jako řádky, které jsou proměnlivé

a tak je těžší je přesně identifikovat. Zde je to pomocí identifikátoru osy potomek (descendant::), který vezme všechny potomky elementu em, které obsahují text MonkeyLabel (případně NeedMoreSpace).

```
$this->driver->get_element("//div[descendant::em[contains(text(),
'MonkeyLabel')] and contains(@class, 'row')]//div[contains(@class,
'an-checkbox-input')]", "MonkeyLabel label selected")->click();

$this->driver->get_element("//div[descendant::em[contains(text(),
'NeedMoreSpace')] and contains(@class,
'row')]//div[contains(@class, 'an-checkbox-input')]",
"NeedMoreSpace label selected")->click();
```

Další dva příkazy slouží k zaškrtnutí checkboxů u námi vytvořených labelů. Konstrukce XPath je zde v podstatě stejná s tím rozdílem, že hledáme checkbox, který pomocí metody get_element zaškrtneme (resp. na něj klikneme).

```
$this->driver->get_element("//div[contains(@class, 'an-btn-
delete')]","labels deleted")->click();

$this->driver->get_element("//div[contains(@class, 'an-btn-red-
remove')]","delete button")->click();
```

Jelikož jsou checkboxy u labelů zaškrtnuty, zaktivní se button “Delete” a můžeme labely smazat. Což pomocí těchto dvou příkazů uděláme. První klikne na tlačítko “Delete”, což otevře příslušné modálové okno a druhý klikne na button “Delete” v modálním okně. Zde by se vybízelo kontrolovat určité prvky v modálovém okně, které je otevřeno po kliku na “Delete”, ale pro co největší konzistenci s manuálním testem je to vynecháno.
Tímto je zkontrolována první část testu, kdy byly vytvořeny dva labely, jeden privání a druhý globální a pomocí tlačítka “Delete” v horní liště byly smazány.

```
$this->driver->assert_element_present("//span[contains(@class,
'an-icon an-facebook')]", "facebook icon");

$this->driver->assert_element_present("//span[contains(@class,
'an-icon an-twitter')]", "twitter icon");

$this->driver->assert_element_present("//span[contains(@class,
'an-icon an-youtube')]", "youtube icon");

$this->driver->assert_element_present("//span[contains(@class,
'an-icon an-vk')]", "vk icon");
```

Tyto čtyři příkazy kontrolují ikonky pro jednotlivé sociální sítě v sekci Labels, které ukazují počet stránek daných platforem, kterou daný label obsahuje.

```
$this->driver->get_element("//div[contains(@class, 'an-addpage-btn
an-btn-primary')]","Create New Label Button")->click();

$this->driver->get_element("//input[contains(@class, 'an-textbox-
input') and contains(@name, 'labelTitle')]","Create New Label
text")->send_keys("SpaceFunk");

$this->driver->get_element("//div[contains(@class, 'an-radiogroup-
item key-private')]//div[contains(@class, 'an-checkbox an-
notchecked')]","Private radiobutton")->click();

$this->driver->get_element("//div[contains(@class,'an-modal-
wrapper')]//div[contains(@class,'an-btn-primary')]","Create Button
in modal")->click();
```

Tímto se dostáváme do kontroly dropdownu, který je zobrazen po rozkliknutí šipky v nějakém řádu u nějakého labelu. Opět je tedy vytvořen label, tentokrát s názvem "SpaceFunk". Label je privátní, aby šla zkontrolovat změna do globálního typu (zpětně to nelze).

```
$this->driver->get_element("//div[descendant::em[contains(text(),
'SpaceFunk')] and contains(@class, 'row')]//div[contains(@class,
'an-btn-dropdown-trigger')]", "dropdown")->click();

$this->driver->get_element("//div[contains(text(), 'Edit
Label')]", "edit label")->click();

$this->driver->get_element("//input[contains(@class, 'an-textbox-
input') and contains(@name, 'labelTitle')]","Edit label modal")-
>clear();

$this->driver->get_element("//input[contains(@class, 'an-textbox-
input') and contains(@name, 'labelTitle')]","Edit label modal")-
>send_keys("SpaceMonkey");

$this->driver->get_element("//div[contains(@class, 'an-btn-primary
an-btn') and not (contains(@class, 'an-addpage-btn'))]")->click();
```

Nyní se dostáváme k první možnosti, kterou je možno vybrat z dropdownu, který je otevřen u příslušného labelu. První příkaz, podobný některým z předchozích, opět identifikuje řádek s nově vytvořeným labelem “SpaceFunk”, pomocí identifikátoru potomek. Na řádku chceme identifikovat šipku, která po rozkliknutí otevře dropdown se třemi možnostmi: editoval label, konvertovat na globální label, anebo smazat label. V tomto případě vybíráme první možnost, tedy editovat label. Po kliknutí na tento výběr v dropdownu se otevře opět dialog (modální okno), kde je možnost změnit název labelu. Třetí příkaz identifikuje textové pole a následně pomocí metody “get_elemenet->clear()” vymaže původní název. Další příkaz je téměř stejný s tím rozdílem, že do prázdného pole vložíme nový název “SpaceMonkey”. Posledním příkazem se potvrdí změna kliknutím na tlačítko “Save”.

```
$this->driver->get_element("//div[descendant::em[contains(text(),
'SpaceMonkey')] and contains(@class, 'row')]//div[contains(@class,
'an-btn-dropdown-trigger')]", "dropdown")->click();

$this->driver->get_element("//div[contains(text(), 'Convert to
Global Label')]", "label convert")->click();

$this->driver-
>assert_element_present("//div[descendant::em[contains(text(),
'SpaceMonkey')] and contains(@class,
'row')]//span[contains(@class, 'an-label-global')]", "global
control");
```

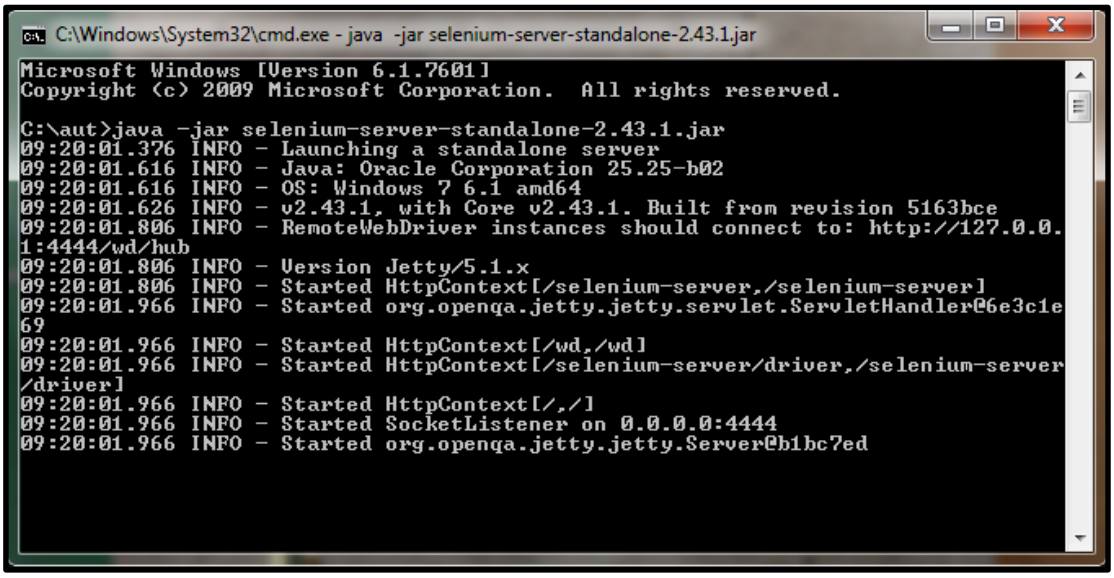
První z těch tří příkazů opět klikne na šipku na konci řádku labelu. Tentokrát ovšem musíme řádek přesně identifikovat pomocí nového názvu, čímž zkontrolujeme I to, že se název opravdu změnil. Dalším příkazem vybereme druhou možnost z dropdownu, což je změna z privátního labelu na globální. Třetí příkaz kontroluje to, zda se změna opravdu provedla.

```
$this->driver->get_element("//div[descendant::em[contains(text(),
'SpaceMonkey')] and contains(@class, 'row')]//div[contains(@class,
'an-btn-dropdown-trigger')]", "dropdown")->click();

$this->driver->get_element("//div[contains(text(), 'Delete
Label')]", "Delete Label")->click();

$this->driver->get_element("//div[contains(@class, 'an-btn-red-
remove')]","delete button")->click();}}
```

Poslední tři příkazy slouží ke smazání labelu. Tedy kontrola třetí možnost v dropdownu. Opět tedy rozbalení dropdownu, kliknutí na možnost „Delete Label“. Otevře se dialog, který je potvrzen. Label je smazán. Tím jsme docílili velice důležitého stavu, tedy že aplikace byla dána do původního stavu a žádný, tímto testem vytvořený, label nezůstal. Díky tomu může být test bez potíží opětovně spuštěn.

4.6.1 Spuštění automatického testu

Obrázek č. 18: Spuštění selenium serveru



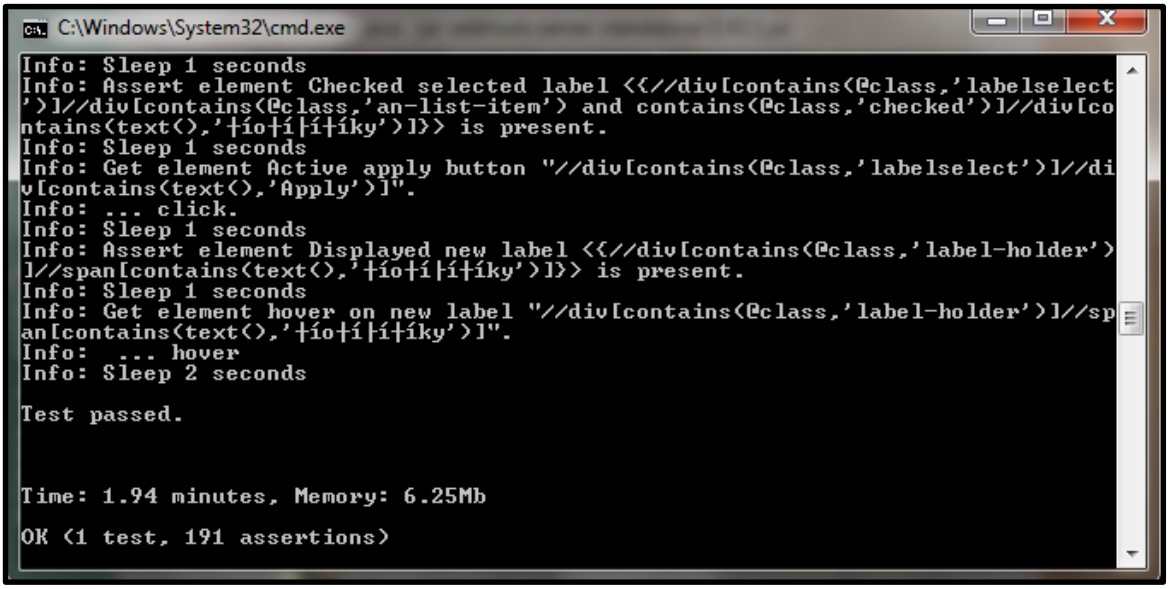
Zdroj: vlastní

Společnou věcí, která bude potřebná pro psaní testů, je Selenium server. Jednoduše se dá stáhnout jeho poslední verze na internetu. V našem případě je uložen na disku C ve složce aut, tudíž si stačí otevřít příkazovou řádku v dané složce a tam vložit příkaz:

java –jar selenium-server-standalone-2.43.1.jar

Pokud chceme test vyzkoušet, že proběhl v pořádku a že je tedy napsán tak, aby v budoucnu odhalil chyby, které tam teď momentálně nejsou, musíme si spustit selenium server lokálně. Na obrázku s příkazovou řádkou a spouštěním selenium serveru v ní je vidět, co je odpovědí při správném spuštění. V našem případě není nutně potřeba spouštět server v rámci našeho počítače, protože samotné regresní automatické testy probíhají jinde a jsou spuštěny přes aplikaci Jenkins. Uloženy jsou v repozitáři Git.

Obrázek č. 19: Doběhlý test



Zdroj: vlastní

Na obrázku výše je vidět, že test proběhl v pořádku. Trval 1.94 minuty. Jednalo se o spuštění pouze jednoho testu. Ve výstupu v příkazové řádce také vidíme jednotlivé asertace a jiné metody použité v testu, které proběhly. Pokud by v testu byla nějaká chyba a nedoběhl by tedy v pořádku (byl by rozpor mezi napsaným testem a aplikací), tak by v příkazové řádce byly vypsány jednotlivé nesrovnalosti a šlo by z nich vyčíst, kde se problém stal.

4.6.2 Prostředí pro psaní

Test byl psán v prostředí NetBeans. Jedná se o zdarma distribuovanou platformu, vyvíjenou společností Oracle Corporations. Primárně slouží k programování v jazyku Java, je v něm i naprogramováno, nicméně lze využít samozřejmě jakkoliv. Volba prostředí pro psaní testů nebyla nijak zásadní, většina funkcí stejně není používána. Nicméně jako silnou výhodu lze brát to, že v levé části tohoto prostředí je umístěna stromová struktura, díky které lze snadno procházet jednotlivé testy.

5 Závěr

Teoretická část této práce měla za úkol definovat jednotlivé náležitosti související s testováním softwaru. Byly tedy popsány jednotlivé typy testů, úrovně testování a samozřejmě popsán testovací proces. Všechny tyto aspekty se mohou v různých literaturách mírně lišit, je tedy dobré čerpat z podobných zdrojů, aby se nerozcházely. Proto zde autor vycházel především z uznávané certifikace ISTQB a jejích učebních osnov, případně z jiných zdrojů, které se také této certifikace drží.. Jedná se o materiály, které tvoří zkušení odborníci v softwarovém testování. Tato práce může tedy sloužit jako dobrá příprava na případné zkoušky v rámci ISTQB. Dále byly popsány základní vývojové metody softwaru s důrazem na to, jak v nich probíhá testování a čím se liší. Součástí byl také popis rolí v testovacím týmu. Na závěr teoretické části byl popsán nástroj Selenium sloužící pro psaní testovacích automatických skriptů, který byl následně uplatněn v části praktické, kde byl, pomocí tohoto nástroje, automatický test vytvořen.

V praktické části byla potom představena společnost, jejímž produktem se práce zaobírala. Daný softwarový produkt byl následně také představen a část, na kterou byl aplikován manuální i automatický test, byla analyzována. Analyzování a zamyšlení se nad částí softwarového produktu, která bude testována, je velmi důležitou součástí, která pomáhá tomu, aby test dával logicky smysl, jednotlivé kroky na sebe navazovaly a aby si test „po sobě uklidil“. S použitím teoretické části, tedy literární rešerše, poté proběhlo psaní jednotlivých testů. Hlavním cílem praktické části bylo nastínit tvorbu automatický testů a následně oba druhy porovnat a ukázat, zda lze manuální testování nahradit automatickým a za jakých okolností se automatizovat vyplatí.

Na první pohled je zřejmé, že psaní automatických testů je mnohonásobně náročnější než tvorba manuálních testovacích případů. Je to způsobeno jednak časovou náročností, ale i náročností technickou. Na testera automatizéra jsou kladeny větší nároky týkající se toho, že musí znát techniky psaní (v našem případě selenium). Navíc musí mít, pokud nepíše pouze automatické testy, také všechny vlastnosti manuálního testera (cit pro detail, atd.).

Tvorba manuálních testů je vcelku jednoduchá. Krok po kroku se popíše, jak se má daná část aplikace otestovat. Zde potom samozřejmě záleží na zkušenostech jednotlivých testerů a jejich znalostech dané aplikace. Pokud se jedná o jednu aplikaci, která je vyvíjena delší dobu a tester pracuje kontinuálně pouze na ní, jeho zkušenosti s aplikací jsou obrovské a testy nemusí být popsány vůbec podrobně. Pokud jsou ovšem jeho zkušenosti malé a aplikaci nezná, musí být test popsán dopodrobna (viz případ v kapitole 4.5).

Automatický test je v každém případě potřebné psát podrobně, aby vykonal sled kroků, který je určen. Tento test neudělá nic navíc, oproti manuálnímu testování, které je kontrolováno lidským okem a tester si může všimnout i věcí mimo testovací krok. Automatické testy navíc mohou mít problém s identifikací designových chyb. Po interním dotazování testerů začínajících s psaním automatizace bylo zjištěno, že napsání prvního automatického testu může trvat až desetkrát déle, než napsání testu manuálního.
I vzhledem k tomu, že zde bude existovat křivka učení, která bude postupně snižovat čas strávený psaním automatického testu, manuální testy zaberou mnohem méně času. Další nákladnou položkou je údržba automatizačního frameworku, o kterou se musí starat speciální pracovníci se znalostmi programování.

Oba přístupy mají společné to, že je potřeba je udržovat. Při změně zadání je vždy nutné aktualizovat jak testy automatické, tak testy manuální. U automatických to však vyžaduje údržbu vyšší. V případně webové aplikace, kdy se při přidání nové funkcionality změní některá část kódu, se může změnit XPath k některému elementu a test začne selhávat a je potřeba ho upravit.

Co tedy z hodnocení obou typů přístupů testování vychází je to, že tvorba manuálních testů je jednodušší, rychlejší a ve výsledku i levnější. Automatické testy jsou dražší a náročnější. Je potřeba více kvalifikovaný personál a větší údržba nejen samotných testů, ale i frameworku. Nicméně síla automatických testů je v tom, že se dají jednoduše a téměř zdarma spouštět. K provádění manuálních testů je potřeba personál, který znamená náklady navíc. Automatické testy mohou sloužit jako pravidelné smoke testy, které probíhají například každou hodinu a hodnotí, zda je aplikace životaschopná a splňuje alespoň základní požadavky. Případně mohou sloužit k regresním testům, které se

pravidelně opakují a kontrolují aplikaci před vypuštěním nové verze do produkce. Po dlouhém psaní automatického testu tak není spuštěn jen jednou a poté vyhozen.

Rozdíl v ceně obou přístupů je v tom, že manuální testování má po celou dobu zhruba konstantní náklady. U automatického testování jsou počáteční náklady na automatizaci obrovské, ale mezní náklady minimální. To znamená, že průměrné náklady v čase klesají a vyplatí se tedy v případě, že se testy mnohokrát opakují.

Z analýzy této práce vyplývá, že automatické testy mohou testy manuální nahradit v případech smoke testů a regresních testů, tedy testů, které se neustále opakují a kontrolují především funkcionalitu a životaschopnost aplikace. Závěrem lze říci, že se tedy nedají plně nahrazovat a spíše se tyto přístupy doplňují.

Literatura

AGNIHOTRI, Poorvai, abodeqa.com [ONLINE], [cit. 2015-01-07], 2014. *Reviews, Walkthrough and sinpection in software Testing.* Dostupné z WWW: http://www.abodeqa.com/2014/03/22/reviewswalkthrough-and-inspection-in-software-testing/

BECK, Kent a spol., agilemanifesto.org [ONLINE], [cit. 2015-02-17], 2001. *Manifesto for Agile Software Development*. Dostupné z WWW: http://agilemanifesto.org/

BURNS, David. *Selenium 2 Testing Tools: Beginner's Guide*. 2. vydání. Birmingham: Packt Publishing, 2012. ISBN 1849518300

COPELAND, Lee. *A Practitioner's Guide to Software Test Design.* 1. vydání. Londýn: Artech House, 2003. ISBN 1-58053-791-X

CRAIG, Rick D. a Stefan P. Jaskiel. *Systematic Software Testing*. 1. vydání. Londýn: Artech House, 2002. ISBN 9781580535083

ČERMÁK, Miroslav, cleverandsmart.cz [ONLINE], [cit. 2015-02-03], 2011. *Automatizované testy aplikací typu klient-server*. Dostupné z WWW: http://www.cleverandsmart.cz/automatizovane-testy-aplikaci-typu-klient-server/

HLAVA, Tomáš, testovanisoftwaru.cz [ONLINE], [cit. 2014-11-26], 2011. *Fáze a úroveň provádění testů*. Dostupné z WWW: http://testovanisoftwaru.cz/tag/systemove-testovani/

HOWER, Rick, softwareQATest.com [ONLINE], [cit. 2015-02-01], 2015. *What is walkthrough*. Dostupné z WWW: http://www.softwareqatest.com/qatfaq1.html

JOHNSON, David W., searchsoftwarequality.techtarget.com [ONLINE], [cit. 2015-01-28], 2007. *The role of a software manager*. Dostupné z WWW: http://searchsoftwarequality.techtarget.com/tip/The-role-of-a-software-test-manager

KADLEC, Václav. *Agilní programování: metodiky efektivního vývoje softwaru*. 1. vydání. Brno: Computer Press, 2004. ISBN 80-251-0342-0

KANER, Cem a spol. *Testing Computer Software*. 2. vydání. New York: John Wiley and Sons, 1999. ISBN 0471358460

KNESL, Jiří, knesl.com [ONLINE], [cit. 2015-01-02], 2014. *Agilní vývoj není (jenom) Scrum a Kanban*. Dostupné z WWW: http://www.knesl.com/articles/view/agilni-vyvoj-neni-jen-scrum-nebo-kanban

KRÁLOVÁ, Iveta, metodikamezitest.asp2.cz [ONLINE], [cit. 2015-02-10], 2013. *Role: Test analytik.* Dostupné z WWW: http://metodikamezitest.asp2.cz/

MCWHERTER, Jeff a Ben Hall. *Testing ASP.NET Web Applications.*1. vydání. Indianapolis: Wiley Publishing, 2010. ISBN 978-0-470-49664-0

MÜLLER, Thomas a spol., castb.org [ONLINE], [cit. 2014-11-20], 2011. *Syllabus ISTQB Certifikovaný tester - ucební osnovy pro základní stupeň*. Dostupné z WWW: http://castb.org/wp-content/uploads/2013/11/ISTQB-CTFL-Syllabus-v2011-CZ-Beta1.pdf

MYERS, Glenford J. *The Art of Software Testing.* 3. vydání. New York: John Wiley and Sons, 2011. ISBN 978-1-118-03196-4

PAGE, Alan a spol. *Jak testuje software Microsoft*. 1. vydání. Brno: Computer Press, 2014. ISBN 9788025128695

PATTON, Ron. *Testování softwaru*. 1. vydání. Brno: Computer Press, 2002. ISBN 80-7226-636-5

ROUDENSKÝ, Petr a Anna Havlíčková. *Řízení kvality softwaru*. 1. vydání. Brno: Computer Press, 2013. ISBN 9788025138168

ROSS, Joel E., totalqualitymanagement.wordpress.com [ONLINE], [cit. 2014-11-20], 2009. *Definition of Quality*. Dostupné z WWW: https://totalqualitymanagement.wordpress.com/2009/08/27/definition-of-quality/

SCHAEFER, Hans a spol. *Software Testing Foundations*. 4. vydání. Santa Barbara: Rocky Nook, 2014. ISBN 9781937538422

Seznam obrázků