Univerzita Hradec Králové
Fakulta informatiky a managementu
Katedra informatiky a kvantitativních metod

Automatizované testování webových aplikací

DIPLOMOVÁ PRÁCE

Autor: Klára Smatanová
Studijní obor: Informační management

Vedoucí práce: Ing. Karel Mls, Ph.D.
Odborný konzultant: Ing. Marcel Veselka

Hradec Králové duben 2015

Prohlášení:

Prohlašuji, že jsem diplomovou práci zpracovala samostatně a s použitím uvedené literatury.

Podpis

V Hradci Králové dne Klára Smatanová

Poděkování:

Ráda bych poděkovala panu Ing. Marcelu Veselkovi za vedení, inspiraci a především za přínosné informace, které mi pomohly ke zpracování tohoto tématu. Dále bych ráda poděkovala vedoucímu mé diplomové práce, panu Ing. Karlu Mlsovi, Ph.D., za metodické vedení. Velké poděkování patří také kolegům, kteří se mnou o testování diskutovali a přátelům za nemalou podporu.

Anotace:

Tato diplomová práce se zabývá automatizovaným testováním webových aplikací. V teoretické části jsou popsány základní principy testování softwarových systémů, které obsahují úvodní informace o tom co je testování softwaru, popisuje jednotlivé fáze testování softwaru a upozorňuje na důležitost dokumentace testovacích případů. Následuje teoretická část o Automatizovaném testování softwarových systémů, kde jsou definovány přínosy a různé přístupy k automatizaci. Dále je zde také metodika automatizovaného testování, která vysvětluje 6 fází pro úspěšnou implementaci automatizovaných testů a nezapomíná ani na prioritizaci a údržbu automatizovaných testů. Následně se práce věnuje metrikám, které lze použít při procesu automatizace, popisuje důležitost odůvodnění použití automatizovaných testů pomocí metriky ROI a definuje Business case. Poté se diplomová práce věnuje možnostem zakomponování Frameworku do procesu automatizace webových aplikací a shrnuje výhody, kterých tím může být dosaženo. Teoretická část je zakončena popisem několika vybraných nástrojů pro automatizované testování. Praktická část se zabývá procesem částečné automatizace testování webové aplikace TestLink.

Annotation:

Title: Automated testing of web applications

This diploma thesis deals with web applications automated testing. Theory section introduces and describes the basic principles of software systems testing. The same section describes each phase of software testing and highlights the presence of documentation in testing. Following theoretical part of software systems automated testing defines benefits and different approaches to automation. Methodic of automated testing is provided as well, with explanation of the 6 phases for the successful implementation of automated tests. Continues with the prioritization and maintenance of automated tests. Further the thesis deals with metrics used in process automation, describes the importance of decision to use of automated tests using ROI metrics and defines the business case. Later the thesis describes the possibilities of Framework incorporation into process of web applications automation testing and summarizes the achievable benefits. The theoretical part ends with a description of several selected tools for automated testing.

Thereafter the practical part deals with the process of partial automation testing of web applications TestLink.

Obsah

1. Úvod

Softwarové systémy jsou nedílnou a velmi důležitou součástí dnešního světa. Vzhledem k tomu, že ovlivňují a zasahují velice důležité oblasti našich životů, je nutné, aby fungovaly tak, jak mají. Jednou z disciplín softwarového průmyslu, která se primárně věnuje kvalitě výsledných produktů, je testování softwaru. Přestože její důležitost a přínos pro softwarové systémy jsou nepopíratelné, jde někdy o opomíjenou a, především pro vývojáře, často nezajímavou oblast. Testování si ale postupně našlo své místo ve vývoji softwaru a při projektech se dnes tvoří specializované týmy věnující se testování výstupních produktů.

Podobně jako softwarové systémy automatizují a zrychlují některé činnosti běžného života, tak také tzv. „manuální testování“ je poslední dobou více automatizováno. V současnosti je sice v oboru dostupných vícero nástrojů podporujících automatizaci, ale samotné nástroje nemusí bez správně zvolených postupů dosáhnout očekávaných přínosů. Je třeba si uvědomit, že jde stále o relativně mladou disciplínu s omezeným množstvím zdrojů, ale s o to větším počtem otázek a názorů. Cílem diplomové práce je tedy analyzovat metody a přínosy automatizovaného testování a na zvoleném systému demonstrovat výběr konkrétní metody a nástrojů, následně navrhnout a vyvinout sadu automatizovaných testů/skriptů a ověřit tak realizovatelnost automatizovaného testování za pomoci volně dostupných nástrojů. Jako systém, pro který bude prakticky ověřena realizace automatizovaných testů, byla zvolena aplikace TestLink. Jde o webovou aplikaci, ke které koncoví uživatelé přistupují přes webový prohlížeč.

Druhá kapitola diplomové práce pojednává o základních principech testování softwarových systémů, zdůrazňuje přínosy testování jako takového a popisuje jednotlivé fáze testování. Neopomíná také důležitost dokumentace testovacích případů.

Třetí kapitola definuje automatizované testování, jeho přínosy vůči manuálnímu testování a vzájemnou koexistenci manuálních a automatizovaných testů. Je zde také uvedena metodika automatizovaného testování, která je klíčová

pro úspěšnou realizaci automatizace. Efektivita a úspěšnost automatizace je měřena různými metrikami, ty nejpřínosnější jsou popsány v podkapitole „Metriky“. Další součástí této kapitoly je popis přístupů k automatizaci testování, ze kterých je pak v další části vybrán konkrétní přístup pro praktickou tvorbu automatizovaných skriptů.

Při praktické tvorbě a následné údržbě automatizovaných testovacích skriptů může sehrát důležitou roli tzv. framework. Principy jeho tvorby a hlavní přínosy popisuje čtvrtá kapitola diplomové práce.

Jak již bylo zmíněno v úvodě kapitoly, samotná automatizace testů bude realizována pro aplikaci TestLink. Kapitola 5 uvádí stručný přehled funkcionalit této aplikace.

V kapitole „Nástroje pro automatické testování webových aplikací“ je uveden seznam a krátký popis nástrojů, které podporují tvorbu automatizovaných testovacích skriptů a vytvářejí základ technické platformy pro automatizaci.

V následujících kapitolách (7-9) diplomové práce je provedena analýza a návrh tvorby automatizovaných skriptů pro vybranou aplikaci TestLink. Následně je uveden popis samotného vývoje a spouštění vytvořených skriptů. Tato část práce začíná multikriteriálním výběrem konkrétních nástrojů, které budou ve vývoji skriptů využity. Po zvážení možných přínosů jsem se rozhodla vývoj skriptů podpořit implementací jednoduchého frameworku. Jeho koncept, architektura a jednotlivé komponenty jsou vysvětlené v podkapitole „Plánování, návrh a vývoj testování“. V podkapitole „Spuštění a řízení testů“ jsou pak uvedené ukázky zdrojových kódů vyvinutých testovacích skriptů a knihoven frameworku a následně jsou zde také shrnuty způsob a výsledky spuštění testů.

Kapitola „Závěr“ vyhodnocuje stanovené cíle diplomové práce, shrnuje zvolené principy a kroky automatizace a zároveň naznačuje možné směry dalšího využití práce.

2. Základní principy testování softwarových systémů

Softwarové systémy jsou v běžném životě dnes využívány k mnoha činnostem a zasahují do širokého spektra oblastí, jak v pracovním, tak v osobním životě. Jedná se o mnoho typů obchodních aplikací, přes telekomunikace, spotřebitelské produkty, aplikace pro finanční instituce až po sociální softwary aj. Obecně známé základní rozdělení rozlišuje softwary systémové a aplikační. Mezi systémové softwary řadíme operační systémy, firmware a obecně lze říci, že zajišťují běh počítače. Díky nim funguje aplikační software počítače, který umožňuje běžnému uživateli vykonávat různé činnosti na počítači a produkovat výstupy. Do aplikačního softwaru zahrnujeme různé programy např. kancelářské, bankovní, grafické, vývojové, zábavní aj. V současné době již není dělení na systémové a aplikační software tak jednoznačné jako v minulosti. Prosazují se různé stupně middleware, tedy software, který zprostředkovává určitou platformu jako základ pro nasazení aplikací. Příkladem takových programů jsou aplikační servery nebo různé „cloud“ služby poskytující prostředí pro tvorbu a provoz vlastního software na prostředcích třetích stran. Dalším aspektem je také to, zda se jedná o webovou aplikaci, ke které uživatel přistupuje přes webový prohlížeč, nebo se jedná o tlustého klienta, kdy si uživatel aplikaci musí nainstalovat přímo do počítače.

2.1. Proč je důležité testovat softwarové systémy

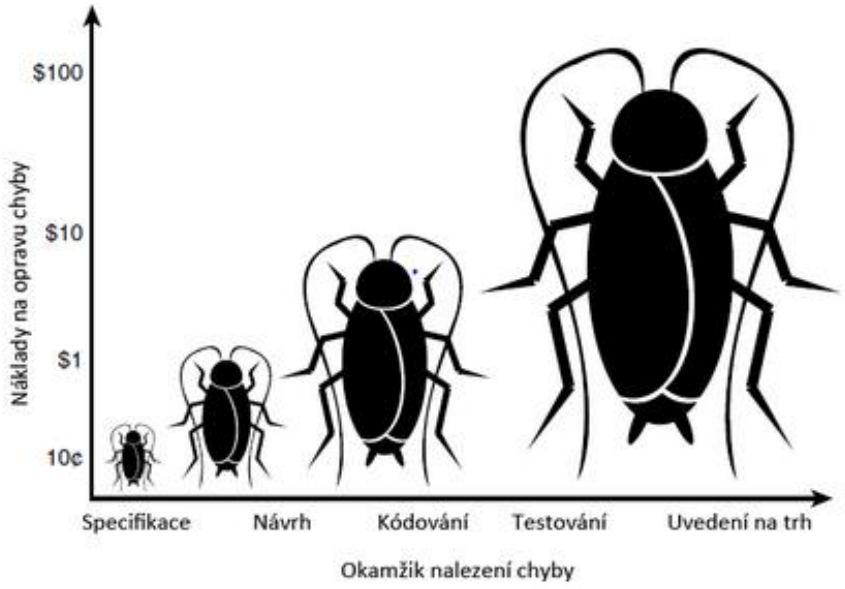
Vzhledem k velkému vlivu softwarových systémů na naše životy je důležité si uvědomit, že nesprávná funkčnost těchto systémů může vést i k velmi závažným problémům. Nemusí se jednat pouze o ztrátu času v případě nefunkční aplikace. Může se jednat i o ztrátu peněz například v případě aplikace pro správu cenných papírů nebo přímé ohrožení života, například v systémech pro řízení letového provozu, apod. Ale ztráta peněz může vznikat například i z důvodu ztráty klienta v důsledku chyb v internetové aplikaci. Tento trend lze sledovat zejména v poslední době. Změna je spojena s velkým rozmachem internetu, kdy je uživatelům nabízeno nepřeberné množství různých alternativ a v případě, že klient není spokojen s funkcí jedné aplikace, například elektronického obchodu, automaticky odchází jinam. Proto je důležité nasazovat do provozu takové systémy, které obsahují co možná

nejméně chyb. Jednou z možností, kterou toho lze docílit, je vhodné testování již v průběhu vývoje systému.

Petr Roudenský a Anna Havlíčková [23] ve své knize tuto problematiku shrnují tak, že mnoho systémů je jedinečných a často jsou vytvářeny na míru konkrétnímu zákazníkovi. Proto použitá technologie, postup a vývojový tým mohou být dohromady úspěšní na jednom projektu, ale nemusí být tak úplně vhodní pro projekt jiný. Nesmíme také opomenout, že všichni lidé chybují.

„Testování je tak nezbytnou součástí každého projektu – bez něho bychom si nemohli být jisti kvalitou vyvíjeného produktu.“

Při vývoji softwarových systémů všeobecně platí, že v čím pozdější fázi vývoje je chyba nalezena, tím jsou celkové náklady na její odstranění vyšší.



Obrázek 1 - Okamžik nalezení chyby. Převzato z: PATTON, Ron. Testování softwaru. Vyd. 1. Praha: Computer Press, 2002, 313 s. Programování. ISBN 80-722-6636-5.

2.2. Definice testování softwaru

Zejména starší definice popisují, že jediným cílem testování je nalézat chyby. Dnešní pohled už je ale o něco širší. Alain Abran a James W. Moore [13] ve své knize definují testování jako činnost prováděnou pro ohodnocení kvality produktu a pro

identifikování chyb a problémů. Tvrdí, že testování je vlastně dynamické[1] ověřování chování systémů pomocí konečné množiny vhodně vybraných testovacích případů.

Hailpern a Santhanam [14] do své definice naopak zahrnují i statické[2] testování. Testování je tedy v podstatě jakákoliv činnost, která odhalí, že se program nechová podle zadání. Mezi nejčastější aktivity statického testování patří statická analýza zdrojového kódu, různé formy revize business specifikace, funkční specifikace a technického designu.

Další pohled na testování popisuje Laurie Wiliamse [33], který testování chápe jako proces analýzy softwaru k detekování rozdílů mezi existujícími a požadovanými podmínkami a vyhodnocení funkcí systému. Testování by mělo probíhat v průběhu celého vývojového procesu.

Testování může znamenat mnoho různých aktivit, které mimo jiné závisejí na těchto dvou faktorech, kdo na vyvíjeném systému pracuje a kde v procesu se nacházíme. Programátoři, administrátoři, uživatelé i konzultanti smýšlejí o systému rozdílně a specializovaný tester se tak často může cítit ztracený v různých interpretacích systému. Pro efektivní testování je dobré se držet těchto 5 základních principů, které popisují Quadri a Farooq [22]:

1. Ověřování a validace
    - testování ověřuje a validuje, zda software splňuje podmínky potřebné pro používání.
2. Priorita pokrytí
    - není možné otestovat všechny funkčnosti systému na všechny kombinace vstupů, proto při definování pokrytí aplikace testy využíváme princip prioritizace.
3. Vyváženost
    - proces testování zohledňuje napsané požadavky na testování, technická omezení reálného světa a očekávání uživatelů. Proces

[1] Dynamické testování – Test vyžaduje spuštění testované aplikace.
[2] Statické testování – Test nevyžaduje běh softwaru.

testování a jeho výsledky musí být opakovatelné a nezávislé na osobě testera.

4. Vysledovatelnost
    - v procesu testování se využívá detailní dokumentace - co bylo testováno, jak bylo testováno, s jakým výsledkem (zaznamenává se nejen úspěch, ale také neúspěch).
5. Determinismus
    - hledání chyb v aplikaci je plánovaný proces. Musí se přesně vědět, co se testuje, proč se testuje, co se hledá, jaké jsou správné a nesprávné odezvy softwaru.

Důležité je také zmínit rozdělení testování z různých pohledů. Již zmíněné bylo statické a dynamické testování, dále rozlišujeme metody testování černé a bílé skříňky. Při testování černé skříňky nemá tester přístup k programovému kódu a naopak u testování bílé skříňky přístup ke zdrojovému kódu má.

Mimo jiné se ještě rozlišují manuální a automatizované testy. U manuálních testů tester přímo provádí celý test, hodnotí průběh a určuje výsledek testu. Přítomnost testera má svá specifika. Díky přítomnosti testera je již v rámci testu prováděna analýza výsledků testu. Jinak řečeno v rámci jednoho testu, který je prováděn manuálně testerem, může dojít k odhalení několika (někdy i mnoha) různých chyb. Negativní stránkou takového testování je značná časová náročnost takového testu. Oproti tomu automatizované testy probíhají s minimálním zásahem člověka. Vyhodnocení výsledků testu u automatizovaných testů je realizováno až po vyhodnocení testu jako neúspěšného. To má kladný dopad na cenu takového testu. Nevýhodou automatizovaného testu je potřeba investovat do implementace a chybně navržený automatizovaný test a zejména nedodržení některých pravidel pro dobrý návrh automatizovaných testů může značně prodloužit fázi analýzy a identifikace chyby v daném testu. Zároveň bývá u automatických testů obtížnější údržba těchto testů a jejich průběžné přizpůsobování změnám v aplikaci. Automatizovaným testům se nadále věnují následující kapitoly.

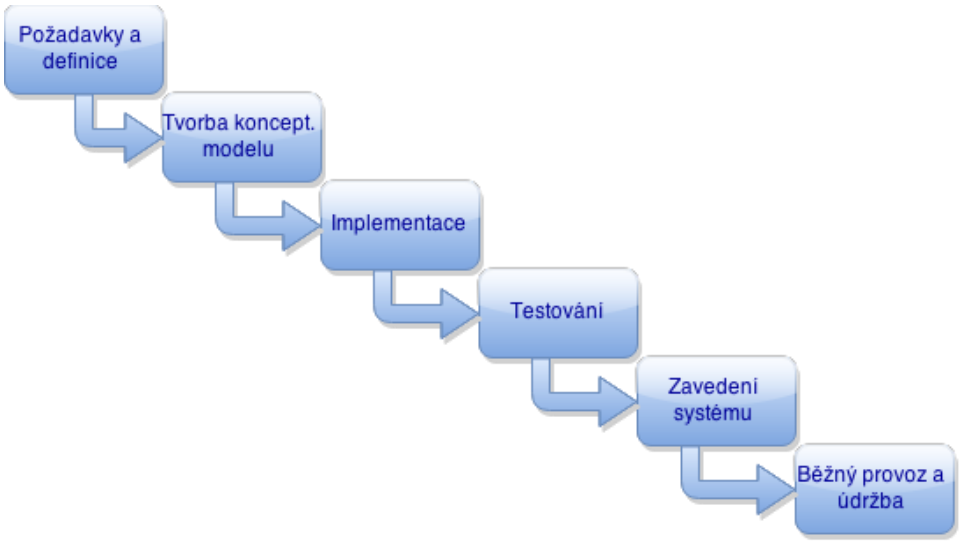
2.3. Životní cyklus vývoje softwaru

V textu "Automatic Test Software" od Y. K. Malayia [19] je doporučené se soustředit na vysokou kvalitu softwaru od počátku projektu, což podporují i zkušenosti velkých a stabilních softwarových firem. Obecně platí, že lze životní cyklus vývoje softwaru rozdělit do několika fází (1. - 5. dle [19], 6. dle [21])

1. Požadavky a definice
    - V této fázi organizace, která vyvíjí software, spolupracuje se zákazníkem, který udává, jak by měl být systém implementován. V ideálním případě by tyto požadavky měly zcela a jednoznačně definovat systém. V praxi je ale většinou potřeba udělat při vývoji softwaru opravné revize. Tyto revize nebo kontroly v této fázi vývoje provádí konstrukční tým, který odhaluje chybějící nebo konfliktní požadavky. Je velmi důležité, že významný počet chyb může být zjištěn již pomocí tohoto procesu. Případné změny v požadavcích v pozdějších fázích, by mohly způsobit zvýšenou chybovost systému.
2. Tvorba konceptuálního modelu
    - V této fázi je systém specifikován jako propojení jednotek tak, že každá jednotka je dobře definována a může být vyvíjena a testována samostatně. Návrh by měl být přezkoumán, aby se rozpoznaly případné nepřesnosti.
3. Implementace
    - V této fázi je napsán kód pro každou jednotku ve vyšším programovacím jazyku (Python, C, C++, PHP, Java…). Pro programátory systému jsou podkladem dokumenty, postupy a principy, které vznikly při výše uvedených fázích. Programátor by se těchto podkladů měl držet a měl by naplnit jejich podmínky.
4. Testování
    - Tato fáze je velmi důležitou součástí vývoje softwaru a zajišťuje vysokou spolehlivost vyvíjeného softwaru. Y. K. Malayia uvádí, že tato fáze může nabývat 30 - 60 % z celkové ceny vývoje. I testování se

obecně dělí na několik úrovní - Unit testování, integrační testování, systémové testování, akceptační testování, regresní testování (viz další kapitola).

5. Zavádění systému a jeho zkušební provoz
    - V této fázi dochází k instalaci softwaru a k jeho zavádění v organizaci. Probíhá zde také poskytnutí uživatelských příruček a školení uživatelů. Pokud by se tato fáze vývoje podcenila, mohla by u budoucích uživatelů vzniknout averze vůči novému systém.
6. Běžný provoz a údržba
    - Jde o konečnou fázi vývoje, kdy už by měl být systém zařazen do běžného provozu. V této fázi se musí neustále udržovat správný provoz a úprava parametrů aplikace tak, aby zajišťovaly nové požadavky uživatelů. Jde tedy o údržbu systému. Zároveň by měl být také zajištěn soulad s původním projektem a dokumentací, zabezpečení systému a ochrana dat před neoprávněným přístupem nebo minimalizace škod vzniklých výpadkem systému např. záložními systémy nebo archivací dat. Do této etapy se také řadí opětovné školení uživatelů.



Obrázek 2 - Vodopádový model životního cyklu vývoje softwaru. Vlastní tvorba inspirovaná: http://testovanisoftwaru.cz/manualni-testovani/modely-zivotniho-cyklu-softwaru/vodopadovy-model/

Životní cyklus vývoje softwaru vychází i ze způsobu vývoje. V poslední době se například čím dál více prosazují tzv. agilní vývojové techniky, které pro vývoj využívají iterativní[3] přístup. Implementace softwaru probíhá postupně. Není tedy jasně oddělena fáze testování. V některých případech se dokonce samostatné testování stává součástí tvorby implementačního modelu. Nebo se naopak fáze testů (v daném případě fáze implementace automatických testů) předřazuje fázi tvorbě implementačního modelu. Jedná se o Test Driven Development.[29]

2.4. Úrovně testování

Jak již bylo zmíněno, testování lze dělit na několik úrovní [19], které vidíme na následujícím obrázku. (2. - 5. dle [19], 1. - z praxe)

Obrázek 3 - Fáze testování v rámci vývoje softwaru. Vlastní tvorba.

1. Statická analýza zdrojového kódu
    - Standardní součástí většiny vývojových nástrojů dnešní doby je možnost provádět statickou analýzu zdrojového kódu. Zejména u staticky typovaných programovacích jazyků lze tímto testem odhalit riziková místa ve zdrojovém kódu, kde může potencionálně nastat problém v běhu aplikace. Testování na této úrovni zpravidla provádí analytik nebo tester.
2. Unit testování
    - Při tomto testování je každá jednotka (malá část kódu, "unit") testována samostatně. Díky tomu, že každá jednotka je relativně malá a testy se tak mohou provádět samostatně, může být tato jednotka otestována mnohem důkladněji než velký program. Unit testy by měl mít na starost vývojář.
3. Integrační testování

[3] Iterativní - Základním principem iterace je opakování určitého procesu

- Při integračním testování se často odhalují chyby v rozhraní a v interakcích mezi integrovanými komponentami nebo systémy. Dochází zde k postupnému sestavování jednotek a vznikají menší celky, které jsou testovány. Postupným přidáváním jednotek k menším celkům můžeme snadněji identifikovat jednotky, které způsobují poruchy systému. Tuto úroveň testování mívá na starost tester.

4. Systémové testování
    - Testování systému jako celku je vykonáváno během systémového testování. Tato činnost pokračuje, dokud nejsou splněna předem daná výstupní kritéria. Obecně platí, že vstupní kombinace parametrů nemusí představovat přesně to, s čím bychom se setkali v reálném provozu. Tuto úroveň testování mívá na starost tester.
5. Akceptační testování
    - Účelem tohoto typu testování je finální posouzení spolehlivosti systému a výkonu v provozním prostředí. To vyžaduje sběr informací o tom, jak sami uživatelé chtějí používat systém. Pokud je to možné, je velmi výhodné zapojit koncového uživatele.

2.4.1. Regresní testování

Dalším důležitým pojmem v oblasti kvality softwaru je regresní testování, které bývá součástí předchozích úrovní a zaměřuje se na testování stávajících funkcí a vlastností. Zjišťuje se, zda zavedení změn a nových vlastností nemělo na tyto stávající funkce vliv. Testování je tedy zaměřeno na nezměněné části kódu. Oblasti, které změněny byly, by již měly být otestovány v předchozím testování. Regresní testování tedy přichází na řadu po opravení známých chyb z předchozího testování.

Právě regresní testy jsou vhodné k automatizaci. Automatizují se hlavně ty regresní testy, které jsou využívány opakovaně. To nastává ve chvíli, kdy se fungující systém udržuje průběžným nasazováním změn a nových funkcionalit.

Regresní testy mohou pokrývat rozsáhlé funkcionality a i s využitím automatizovaných testů může jejich otestování trvat několik hodin, někdy dokonce i dní. Pro optimalizaci běhu se využívají tzv. „Smoke testy“. Často se jedná o podmnožinu regresních testů, která je spouštěna za účelem rychlého zjištění, zda byla aplikace správně nasazena a zda fungují alespoň základní funkce. Tyto testy jsou využívány v testovacích prostředích pro automatické testy, kde zajišťují spuštění plné sady regresních testů pouze v případě, že byla aplikace správně nasazena a základní funkcionality fungují korektně a pro testy po nasazení na produkci. Vzhledem k použití testů správného fungování produkční verze aplikace pak bývají tyto testy „neinvazivní“. Tedy takové, že nevytváří nová data a zároveň nedochází ke změně nebo smazání existujících. Případně je možné data modifikovat či vytvářet, ale tyto změny dat musí být následně “uklizeny”, aby nedošlo ke změně stavu dat v systému. [32]

2.5. Dokumentace testovacích případů

Vést si dokumentaci testovacích případů je velice důležitou součástí procesu testování a její vytváření by mělo být samozřejmostí.

„Je nutné si uvědomit, že tester má odpovědnost za chybovost (resp. nechybovost) aplikace. To je skutečnost. Přestože to není tester, kdo v aplikaci chyby dělá, on je tím, kdo garantuje, že aplikace žádné závažné chyby neobsahuje. Toho nelze dosáhnout žádnou formou volného klikání. Je to právě testovací dokumentace, která testerovi slouží jako opěrná soustava pro celé testování. “ [31]

Je tedy v zájmu testerů využívat cílené testování založené na zdokumentovaných testovacích případech.

Testovací případ, využívá se i anglický výraz „test case“, je soubor zdokumentovaných podmínek nebo proměnných, pomocí kterých tester určí, zda testovaný software splňuje požadavky a pracuje správně. Proces vývoje testovacích případů může také pomoci najít problémy v požadavcích nebo návrhu testované aplikace. [28]

Na webu Testování Softwaru určují použitelnost testovacích případů takto:

„Testovací případy se vytvářejí jak pro manuální, tak i automatizované testy. Pro účely manuálního testování jsou tvořeny seznamem prováděných kroků a očekávaných výsledků. Automatizované testovací případy se také někdy označují jako testovací skript. Tvoří je sada programových instrukcí a na rozdíl od manuálních by měly být schopny samy rozpoznat, zda uspěly či selhaly."[27]

Struktura testovacího případu by měla obsahovat minimálně tyto podstatné části, jejichž aplikace do konkrétního testovacího případu je znázorněna na obrázku níže:

- Identifikátor
- Účel
- Podmínky
- Specifikace kroků a vstupů (Testovací scénář)
- Očekávané výsledky

ID	#1
Název	Vytvoř projekt
Účel	Ověření zda proběhne
Typ testu	Verifikační
Čas	2 min
Podmínky	Uživatel je přihlášený a na hlavní stránce aplikace
Kroky	1. Uživatel klikne na Test project management
	2. Systém zobrazí obrazovku pro Test Project Management
	3. Uživatel klikne na tlačítko Create
	4. Systém zobrazí obrazovku pro vytvoření projektu
	5. Uživatel vyplní údaje
	6. Uživatel klikne na uložení projektu
Očekávaný výsledek	7. Systém zobrazí obrazovku pro Test Project Management a tam bude zobrazen nově vytvořený projekt
Provedení testu	OK
Poznámky	

Obrázek 4 - Testovací případ Vytvoř projekt pro aplikaci TestLink. Vlastní tvorba.

U menších projektů se často pro správu testovacích případů používá pouze tabulkový editor. Pokud ale vzniká velké množství testovacích případů, je nezbytné

k jejich uspořádání použít nějaký specializovaný systém pro správu testovacích případů, jedná se o Test case manager nebo Test management tool. Jedním z uživatelsky oblíbených nástrojů je TestLink, který v této práci bude použit jako testovaná aplikace.

Testovací případy lze seskupovat do testovacích sad. Pro testovací sady je používán anglický název „Test Suite".[27]

3. Automatizované testování softwarových systémů

Ve své knize si Dustin, Garrett a Gauf [12] pokládají základní otázku, jaký je rozdíl mezi manuálním a automatizovaným testováním. Odpovědí je, že automatizace:

- Je vývoj softwaru.
- Pomáhá rozšířit rozsah zaměření testování, jehož bychom manuálním testováním nikdy nedosáhli.
- Nenahrazuje potřebu manuálních, analytických dovedností testera, testovací strategii (know-how) a porozumění testovacím technikám. Tyto odborné znalosti testera z manuálního testování slouží jako vzor pro automatizaci.
- Nemůže být jednoznačně odděleno od manuálního testování. Manuální a automatizované testování je propojeno a vzájemně se doplňuje.

Pokud tedy existuje software, pro který jsou vytvořeny stabilní manuální testovací případy, které se opakovaně vykonávají, je nasnadě otázka, zda by nebylo možné tyto testy vykonávat automaticky. [4] Velice důležitým elementem je zde právě ale vhodné využití těchto testů. Vždy je nutné porovnat poměr cena/výkon a správně tak zhodnotit, zda náklady na výrobu automatizovaných testů nejsou zbytečně vysoké v závislosti na tom, co mají přinést (kolik ušetří času, finančních nákladů atd.). Je také nezbytné se zamýšlet nad tím, jak velkou část testovacího procesu je vhodné automatizovat a vybrat by se měly pouze takové testovací případy, které nejsou příliš komplikované, jinak se může velmi snadno stát, že se proces stane neefektivním. Této problematice se ještě budou věnovat další kapitoly.

Správné vytvoření a implementace automatizovaných testů může přinést výrazné úspory prostředků a zvýšení kvality výsledného produktu. K tomu si lze dopomoci i správným výběrem vhodného softwaru, pomocí kterého se bude automatizovat.

3.1. Přínosy automatizovaného testování

Jednou z největších aktuálních výzev vývoje softwaru je touha zákazníka po větším množství funkcionalit v co nejrychlejším čase a co nejlevněji, zatímco zároveň předpokládají kvalitu systému stejnou, ne-li vyšší než jejich očekávání. I přesto, že se neustále při tvorbě softwaru rozvíjejí postupy, zlepšují nástroje a zvyšuje se rychlost dodání softwaru, tak by se měl klást důraz na srovnatelné zlepšení postupů a nástrojů pro testování.

V článku [19], který publikovala katedra výpočetní techniky ze státní univerzity v Coloradu, je problematika automatizovaného testování softwaru shrnuta tak, že většina testování se dnes stále provádí ručně a proces je veden intuitivně. V této souvislosti ale zaostává vývoj softwaru za hardwarovým návrhem a za testy, u kterých je nyní používání nástrojů považováno za povinné. V blízké budoucnosti tak na trhu lze zaručeně očekávat, že vývojáři začnou spoléhat na automatizované testování a budou tak diktovat směr tohoto odvětví. V současnosti se již manuální testy pro hardware vytvářejí mnohem méně než dříve a to samé bude platit i pro software v několika málo letech.

Pokud má být automatizované testování, které reaguje na aktuální výzvy trhu, implementováno správně, mělo by být možné dopovědět na otázky:

- „Proč automatizujeme?“
- „Co nám má automatizace přinést?“

Odpovědí by pak měli být následující body, které ve svém díle pěkně shrnují Dustin, Garrett a Gauf [12]:

- Redukce nákladů a času na testování softwaru
- Zlepšování kvality softwaru
- Zvýšení přínosů manuálního testování systému prostřednictvím rozšíření testovacího pokrytí a nahrazením všedních, pracovně náročných úkolů

- Automatizace dosáhne i toho, čeho lze manuálním testováním těžce docílit, jako je například testování výskytu dvou a více aktivit ve stejném časovém intervalu, testování výkonu a další.

Na webu Software Testing Mentor [1] zase shrnují nejvýznamnější přínosy automatizovaného testování do těchto bodů:

1. Rychlost
    - Realizace testů je výrazně rychlejší než u lidských uživatelů.
2. Opakovatelnost
    - Testeři mohou ověřit, jak software reaguje po opakovaném provedení stejné operace.
3. Znovupoužitelnost
    - Testy mohou být znovu použity v různých verzích softwaru.
4. Spolehlivost
    - Test je proveden naprosto stejně při každém spuštění a je tak eliminován faktor lidské chyby.
5. Komplexnost
    - Tester může vytvořit několik testovacích případů, které budou pokrývat většinu funkcí softwaru.
6. Programovatelnost
    - Testeři mohou naprogramovat sofistikované testy, které mohou přinést skryté informace.

3.2. Přístupy k automatizaci testování

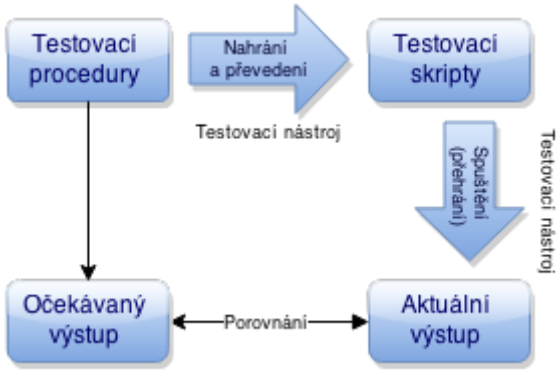
Jelikož jsou s tímto procesem propojeni testeři a různé nástroje i aplikace, existuje mnoho možností, jak lze přistupovat k automatizaci. Níže je uvedeno několik všeobecně známých přístupů:

3.2.1. Přístup zachytit/přehrát

Tento přístup je velice jednoduchý a spočívá pouze v nahrání posloupnosti kroků, které jsou obsaženy v jednotlivých testovacích procedurách. Tato posloupnost kroků se nahrává pomocí testovacího nástroje (např. Selenium IDE) pouhým "naklikáním". Testovací nástroj nahrané kroky převede do testovacího skriptu. Tyto skripty jsou pak manuálně spuštěny a přehrány testovacím nástrojem. Vstupy jsou zachyceny a výstupy mohou být zaznamenávány pro pozdější kontrolu, která může být manuální nebo automatizovaná.

Tento přístup lze použít pro testování na úrovni GUI[4] anebo API[5], kdy je počáteční nastavení a používání snadné. Tento typ skriptů je ale těžké udržet a rozvíjet, jelikož i velmi malé změny (např. změna v rozložení GUI) mohou mít vliv na testovací skripty.

I přes to, že uvedená metoda svádí k definici testů uživatelem, který „pouze nakliká test", ve skutečnosti je ve valné většině případů potřeba zachycený skript revidovat a odladit. V případě dlouhých testovacích scénářů nebo při implementaci rozsáhlejších testovacích sad, je pak i velmi obtížné promítnout do takového skriptu změny z aplikace tak, aby byl použitelný i pro další testování. Celý proces zachycení a revidování zachyceného skriptu je nutné absolvovat znovu při každé změně, což je časově a tedy i finančně velice náročná činnost.[18]



Obrázek 5 - Přístup zachytit/přehrát. Vlastní tvorba.

[4] GUI - Graphical user interface - Grafické uživatelské rozhraní

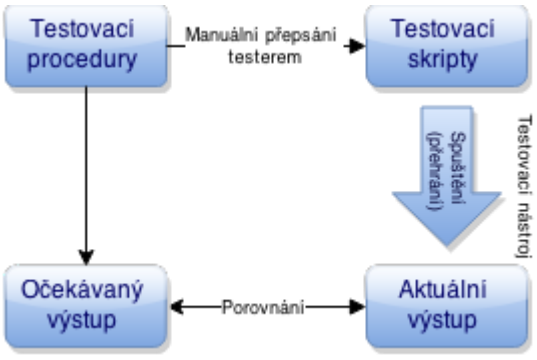
[5] API - Application Programming Interface – Rozhraní pro programování aplikací

3.2.2. Lineární skriptování

U tohoto přístupu se také začíná vytvořením testovacích procedur a každý test se spouští manuálně. Během spuštění testovací nástroj zaznamenává posloupnost akcí a v nějakých případech zachycuje viditelné výstupy z testu na obrazovku. Každá testovací procedura má svůj testovací skript, který lze dále upravovat pro lepší čitelnost, např. přidáváním komentářů, které vysvětlují co se na klíčových místech děje nebo přidáváním jiné kontroly, kterou poskytuje daný skriptovací jazyk používaného nástroje.

Tento přístup může být vhodný pro automatizaci testů na úrovni GUI, ale tato technika není dobrá ve chvíli, kdy potřebujeme automatizovat velké množství testů nebo jsou automatizované testy určené pro mnoho verzí softwaru. Znamenalo by to vynaložení vysokých nákladů na údržbu, jelikož si každá změna verze systému může vyžádat mnoho změn v testovacích skriptech.

Výhodou lineárního skriptování je především to, že vyžaduje jen málo přípravné práce před zahájením automatizace. Jakmile se tester naučí používat daný nástroj, je pro něj velice jednoduché nahrávat test, který vytvořil manuálně a přehrát jej. Programovací dovednosti zde nejsou nutné, ale obvykle jsou užitečné. [5]



Obrázek 6 - Přístup Lineární skriptování. Vlastní tvorba.

3.2.3. Strukturované skriptování

Hlavním rozdílem mezi strukturovanou a lineární skriptovací technikou je zavedení knihovny skriptu. Schéma lze vidět na obrázku č. 6 a rozdíl od lineárního

skriptování je znázorněn žlutou barvou. Knihovna skriptu obsahuje opakovaně použitélné skripty, které provádějí posloupnost akcí, jež jsou běžně vyžadovány v řadě testů.

Dobrým příkladem takových skriptů jsou testy rozhraní[6]. Nemusí se tedy všechny testy vytvářet od nuly manuálně a následně se automatizovat (jako u lineárního skriptování), ale použijí se i hotové testy z knihovny.

To přináší nespornou výhodu ve významném snížení změn při údržbě a snížení nákladů na automatizaci nových testů. Navíc pokud se testuje na základě API, je možné začít s automatizací ještě před tím, než je testovaný software kompletní a k dispozici, je potřeba pouze definice API. Důležitým předpokladem pro správné fungování je ale dobré řízení takových knihoven, skripty by tedy měly být zdokumentovány a pro testery by mělo být snadné najít potřebné skripty. [5]

Obrázek 7 - Přístup Strukturované skriptování. Vlastní tvorba.

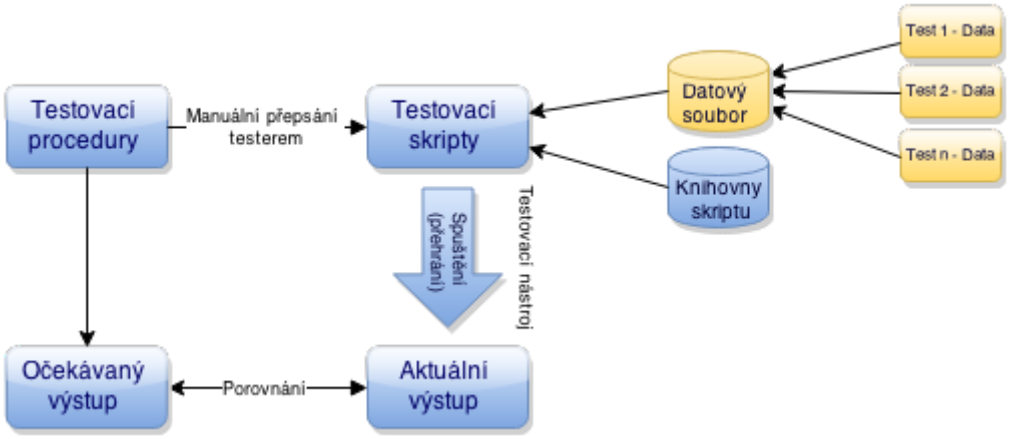
3.2.4. Datově řízené testování

Tato technika navazuje na strukturované skriptování. Největší rozdíl je v získávání vstupů. Ty se vyčleňují ze skriptů a dávají se do jednoho nebo více samostatných souborů, které jsou nazývány datové soubory. Testy implementované jako testovací skripty, které získávají data z datových souborů, se obyčejně nazývají řídící skripty. Je tedy běžné, že jeden testovací skript je použitý na více testů s různými datovými vstupy.

[6] Rozhraní - zařízení, program nebo formát zajišťující spojení mezi jinými zařízeními nebo programy

Díky této technice se dají náklady na přidávání nových automatizovaných testů výrazně snížit. Datově řízené testování nezvyšuje pokrytí testů, ale zajišťuje hlubší testování ve specifických oblastech, proto se tato technika využívá k automatizaci testů, které mají mnoho variací. Nutností je ale správa datových souborů a jejich srozumitelnost pro testera.[5]

Schéma této techniky je naznačeno na následujícím obrázku č. 7. Rozdíl od strukturovaného skriptování je znázorněno žlutou barvou.



Obrázek 8 - - Přístup Datově řízené testování. Vlastní tvorba.

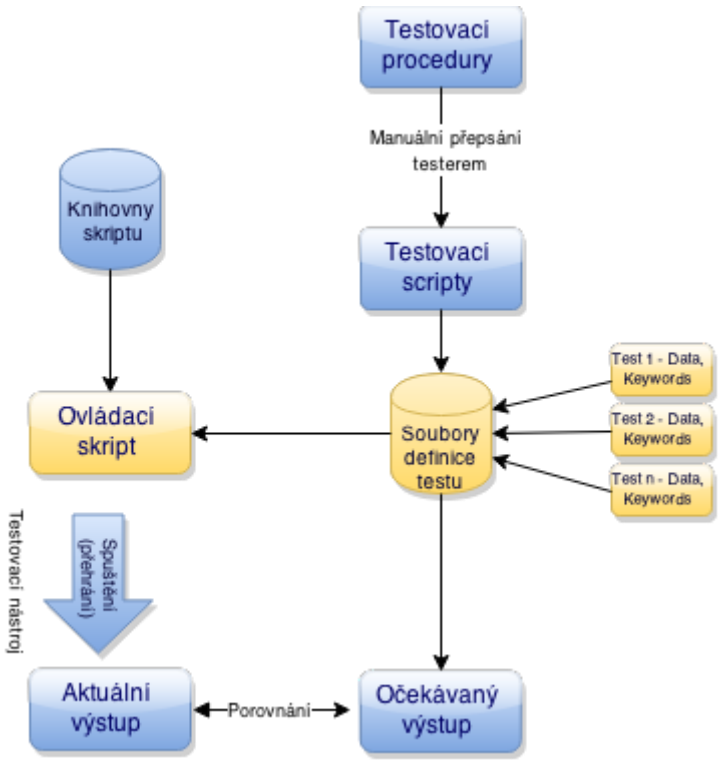
3.2.5. Testování řízené klíčovými slovy

Tato technika je postavena na technice datově řízeného testování. Existují ale dva hlavní rozdíly, které tyto přístupy odlišují. Prvním je přejmenování datových souborů na soubory definice testu a druhým je, že existuje pouze jeden ovládací skript. Soubor definice testu obsahuje datové soubory, jež obsahují testovací data, očekávané výsledky a klíčová slova, která slouží pro ovládání testované aplikace. Ovládací skript pak volá podpůrné skripty, které pro test interpretují klíčová slova. Klíčová slova by měla být vhodně zvolena tak, aby jim tester rozuměl, např. objednat, rezervovat, vytvořit účet, zkontrolovat stav objednávky atd. Dobrá klíčová slova se dají často využít u více testů, oproti tomu špatná klíčová slova použijeme pouze jednou, nebo jen párkrát.

Výhodou této techniky je, že jakmile byl popsán řídící skript i podpůrné skripty, tak se výrazně snižují náklady na přidávání nových automatizovaných testů. Hlavní výhodou jsou právě klíčová slova, ty dávají testerovi větší volnost ve vyhledávání jednotlivých testů.

Implementování klíčových slov je náročným úkolem pro inženýry automatizovaných testů a to zejména v případě, kdy je použit nástroj, který tuto techniku nepodporuje. Použití u malých systémů by mohlo znamenat příliš mnoho režie s implementací a náklady by pak mohly převýšit přínosy. [5]

Schéma této techniky je naznačeno na následujícím obrázku č. 8 a rozdíl od datově řízeného testování je znázorněn žlutou barvou.

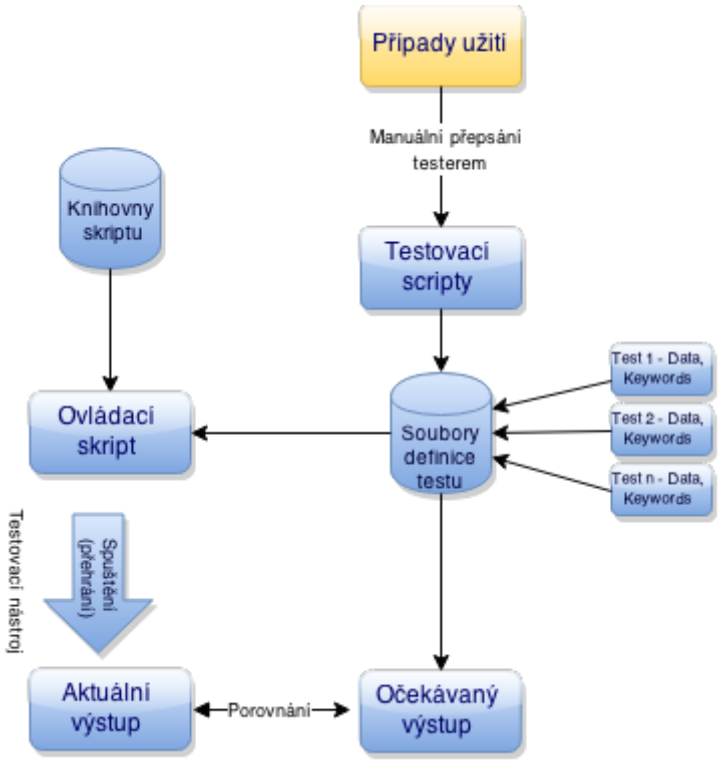


Obrázek 9 - Přístup Testování řízené klíčovými slovy. Vlastní tvorba.

3.2.6. Procesně řízený přístup

Procesně řízený přístup navazuje na testování řízené klíčovými slovy. Rozdílem je, že testovací procedury jsou zastoupeny případy užití [7] testovaného softwaru a z těch jsou tvořeny skripty, které jsou parametrizovány datovými soubory nebo soubory definice testu. Použití případů užití jako základ při vytváření testovacích skriptů umožňuje lépe definovat testovací procedury z hlediska pracovních postupů a workflow[8] aplikace. Speciální důraz musí být kladen na správné pojmenování jednotlivých procesů tak, aby byla zabezpečená jejich správná vzájemná propojenost. [5]

Schéma této techniky je naznačeno na následujícím obrázku č. 9 a rozdíl od testování řízeného klíčovými slovy je znázorněn žlutou barvou.



Obrázek 10 - Procesně řízený přístup. Vlastní tvorba.

[7] Případy užití = Use Case - seznam kroků, který obvykle definuje interakci mezi uživatelem a systémem
[8] Workflow – Průběh pracovní operace

3.2.7. Testování na bázi modelu

Testování na bázi modelu se zabývá automatickým generováním testovacích skriptů z modelů. Tento přístup umožňuje testerům soustředit se pomocí abstrakce na podstatu testování, umožňuje také generovat testy pro různé cílové systémy a cílové technologie. Další výhodou je, že modely, které jsou používány ke generování testů, mohou být při správné údržbě znovupoužitelné. Pokud se model změní, kód se automaticky přegeneruje. V případě, že by tedy nastaly změny v požadavcích, stačí jen, když je upraven model, ze kterého je kompletní sada testů generována automaticky.

Tato technika ale vyžaduje hluboké odborné znalosti, aby mohla být efektivně použita, jelikož může být velmi obtížná. Kromě toho i nástroje potřebné pro tento přístup jsou teprve v období zrání a vývoje. [5]

3.2.8. Programování řízené testy a chováním

Testy řízené programování (TDD) neodděluje testování od fáze vývoje. Základním principem využití tohoto přístupu k testům je velice krátký vývojový cyklus, kdy programátor nejprve nadefinuje automatizovaný test, který selže. Následně implementuje požadovanou funkci a automatickým testem ověří, že jím napsaná funkce pracuje podle předpokladu. V rámci vývoje se tak opakují následující kroky:

1. Vytvoření nových testů
    - Implementace každé nové funkce aplikace začíná vytvořením nového testu nebo sady nových testů. Případně změnu již existujících testů tak, aby zohledňovaly požadovanou změnu v aplikaci. Tento přístup má další přidanou hodnotu oproti testům definovaným až po vytvoření zdrojového kódu a to v tom, že přístup nutí programátora chápat funkční část aplikace před vlastním psaním zdrojového kódu.
2. Spuštění testů

- V tomto kroku je doporučováno spustit nově vzniklé testy a ověřit tak validitu těchto testů tím, že nové testy při spuštění selžou. Případně odhalit chyby v definici testů, které by mohly vést k nechtěnému vyhodnocení testu jako v pořádku.

3. Psaní kódu
    - Implementace vlastního zdrojového kódu aplikace. V tomto kroku není cílem vytvořit finální podobu kódu, ale vytvořit kód, který zajistí bezproblémový průchod testem.
4. Spuštění nových testů
    - V tomto kroku je validována správnost napsaného kódu z hlediska chování aplikace. Všechny nově definované testy, které v kroku 2 selhaly, musí být vyhodnoceny jako bez chyby, případně musí být opraven zdrojový kód.
5. Úprava nově vzniklého kódu
    - Takto implementovaný zdrojový kód je následně možné upravit. Je třeba odstranit zbytečné části kódu nebo optimalizovat některé jeho části apod.

Tento přístup je dobře použitelný v případě, že je zcela jasný záměr a zároveň způsob realizace. Naopak, obtížně použitelný je v případě, že programátor zkouší a vymýšlí nejlepší způsob řešení nějakého problému (realizace „proof of concept“). Většina zdrojů se shoduje, že tento způsob testování je vhodný pro definici testů na úrovni definovaného API. V případě testování například konkrétních tříd se ale názory výrazně liší.[29]

Programování řízené chováním (BDD) je rozšířením a zpřesněním metodiky pro programování řízené testy. Tato metodika se snaží zpřesnit jakým způsobem uchopit problematiku programování, které je řízené testy. Chováním řízené programování kombinuje myšlenky doménově řízeného návrhu aplikací s objektově orientovanou analýzou a návrhem. Aby bylo možné kombinovat pohled uživatele a technický pohled vývojáře, jsou pro programování, které je řízené chováním, využívány speciální nástroje, které umožňují definovat požadované chování aplikace pomocí speciálního jazyka, který je srozumitelný také uživatelům. Tzv.

ubiquitous language[9]. Tento jazyk vychází ze způsobu specifikace nových funkcí v agilních vývojových technikách, kde se používá formát zápisu tzv. „User stories[10]“. Tento zápis definice nové funkce je pak překládán do tříd a metod testů.

Výhodou programování řízeného chováním je specifikace fungování uživatelem namísto programátorem. Na druhou stranu je často potřeba využívat specializované nástroje, které dokáží realizovat překlad definice testu na zdrojový kód testu. [6]

3.3. Metodika automatizovaného testování

Tato strukturovaná metodika[12] je zaměřená na zajištění úspěšné implementace automatizovaných testů. Vychází z ověřených postupů a softwarových technických procesů. Skládá se z 6 fází, z nichž každá musí projít přes bránu kvality před přechodem k dalšímu kroku. Zavedením těchto bran zajišťujeme, že je celý proces automatizace kvalitnější a předem se tak bráníme pozdějšímu a nákladnému přepracování. Fáze této metodiky jsou:

1. Rozhodnutí o automatizaci testování
    - V této fázi probíhá vše, co se týká rozhodnutí o automatizaci. Stanoví se cíle, jaké od automatizace očekáváme, vyvíjí se strategie automatizovaného testování a celkově se zvolí přístup k automatizaci (viz. Kapitola 3.2.).
2. Výběr testovacího nástroje
3. Zavedení procesu automatizovaného testování
    - Tato fáze zahrnuje kroky, které zajišťují úspěšné zavedení automatizace. V této fázi je důležité stanovit požadavky, definovat testovací případy, pojmenovat očekávané výsledky, specifikovat rozhraní a popsat nastavení testovaného systému.
4. Plánování, návrh a vývoj testování

[9] Ubiquitous language - společný ale jednoznačný jazyk mezi vývojáři a uživateli

[10] User stories - říká co chcete dělat, pro koho a hlavně proč. User Story popisuje příběh. Lidský mozek vnímá obrázky a příběhy daleko snadněji než technický popis v bodech.

- V této fázi jsou již vyvíjeny konkrétní testovací skripty, analyzuje se existující AST Tool[11] a případně se vyvíjí testovací framework. Tématika frameworku je podrobně vysvětlena v kapitole 5.

5. Spuštění a řízení testů
   - V této fázi je již spouštěna celá sada vytvořených testovacích případů, řídí se zde jejich provoz a zaznamenávají se výsledky.
6. Přezkoumání vytvořené automatizace a zhodnocení
   - Tato fáze je určena ke zhodnocení automatizace, výsledků jednotlivých testovacích případů a shrnutí celého procesu. Důležitým krokem je také posouzení, zda byla vhodně zvolena strategie a přístup k automatizaci, případně zde lze navrhnout lepší způsob pro další testování. V případě dalšího využití testů nesmíme opomenout údržbu.

Obrázek 11 - Metodika automatizovaného testování. Vlastní tvorba inspirovaná: http://www.cs.nott.ac.uk/~cah/G53QAT/Report08/jrw06u%20-%20website/ATLM_clip_image002.gif.

Pro úspěšnou implementaci automatizovaných skriptů se musí brát v úvahu také následující faktory [5]:

1. Izolace

[11] Automated Software Testing Tool (nástroj pro automatizované testování softwaru) - veřejně dostupný nástroj, který slouží jako podpora pro vývoj testů. Obsahuje podpůrné komponenty (knihovny, návrhové vzory, hotové skripty atd.), které můžeme použít při naší automatizaci. Nemusíme se tak zabývat budováním infrastruktury, ale soustředíme se na vývoj řešení na vyšší úrovni. Samozřejmostí je podpora široké škály platforem a jazyků.

- Testovací prostředí musí být izolováno od vnějších rušivých vlivů tak, aby nebyly narušeny výsledky spouštěného testu. Toto prostředí musí být snadno ovladatelné a reprodukovatelné. Každý update tohoto prostředí musí být předem oznámen a schválen než bude použit.

2. Reprodukce
    - V ideálním případě by mělo být prostředí snadno reprodukovatelné, jelikož některé testy tlačí své prostředí až za možné hranice. Může se pak stát, že se testovací prostředí dostane do stavu, z něhož již není možná obnova.

3. Virtualizace spuštění testovacího prostředí
    - Virtualizace umožňuje spustit navržené řešení automatizace pro více instancí najednou, což dopomůže k rychlejšímu otestování testovaného systému. Virtualizace je škálovatelná, umožňuje snadnou obnovu chybových stavů a chrání jiné spuštěné testy.

    - V poslední době se čím dál více využívá virtualizace testovacího prostředí v cloudu, kdy jsou testy spouštěny na virtuálních serverech v internetu. Tento přístup má jednak pozitivní dopad z hlediska nákladů na údržbu HW a SW v rámci vlastní infrastruktury a jednak z hlediska času, kdy lze využívat velkou výpočetní kapacitu na krátkou dobu s minimálními náklady.

4. Samostatné spuštění testu
    - To nastává až ve chvíli, kdy je možné nechat testy běžet bez dozoru.

Další kapitoly jsou ještě zaměřeny na některé klíčové kroky, které v průběhu předchozích 6 fází vykonáváme.

3.3.1. Prioritizace a výběr rozsahu automatizace testů

Při výběru testů, které je vhodné automatizovat[12], je potřebné se držet základního pravidla, že ještě před výběrem konkrétních testů by se měla provést

analýza pomocí checklistu s danými kritérii. To umožňuje lépe určit testy, které by se měli automatizovat. Příklad jednoduchého checklistu je znázorněn na následujícím obrázku.

Kritéria pro automatizaci	**Ano**	**Ne**
Je test spuštěn víc jak jednou?		
Je test spuštěn na pravidelné bázi, např. pravidelně používaný v rámci jiných testů, jako součástí regresního testu nebo součástí testů spouštěných při tvoření aplikačního buildu?		
Pokrývá test kritické funkčnosti?		
Je ruční provedení testu komplikované nebo dokonce nemožné např. z důvodu paralelizmů v kódu, výkonnosti, přetečení paměti apod.?		
Existují v aplikaci časově kritické komponenty, které musí být automatizované?		
Pokrývá test většinu komplexnosti aplikace?		
Vyžaduje testování spuštění stejných testů s různými testovacími daty?		
Jsou výstupem testů očekávané konstanty, které se nemění spuštěním jednotlivých testů? Případně pokud se tyto výstupy mění, existuje procentuální tolerance pro akceptování testovacích výsledků?		
Je vyhodnocení testů časově náročné, např. analýzy stovek testovacích výsledků?		
Jsou testy spouštěné na stabilní aplikaci?		
Je potřebné testy verifikovat na různých softwarových nebo hardwarových konfiguracích?		

Obrázek 12 - Checklist pro rozhodnutí co automatizovat. Vlastní tvorba inspirovaná z [12].

Dále je také nutné se držet následujících pokynů:

- Nepokoušet se automatizovat všechno najednou
- Zvážit rozpočet, plán a odborné znalosti
- Analyzovat úsilí, které vynaložíme na automatizaci
- Analyzovat potenciální opětovné použití automatizovaných modulů
- Zaměřit automatizaci na opakující se úkoly

Dalším důležitým pojmem při výběru testů k automatizaci je tedy prioritizace12. Při implementaci softwaru upřednostňujeme nějaké funkce před jinými a to pro každé inkrementální13 vydání. Jak v případě iterativního vývoje, tak v případě jiných vývojových metodik. Výběr funkcí je také založený na potřebách zákazníka nebo se např. volí strategie "implementovat nejrizikovější funkce jako první". Plánování a vývoj automatizace testovacích procedur, kromě zohlednění svých specifických priorit, by mělo probíhat se zohledněním priorit, plánu a rizik spojených s implementací samotných softwarových funkcí. Oba přístupy totiž mohou diktovat pořadí, ve kterém budou dány k dispozici funkce pro testování. Je tedy důležité, aby plán vývoje softwaru, včetně plánu vývoje jednotlivých funkcí, byl deterministický a byl zpřístupněn testovacímu týmu tak, aby odpovídajícím způsobem pomohl jeho plánování.

Ve většině případů by měl životní cyklus vývoje softwaru přinést nejprve nejvíce potřebné funkce s nejvyšší prioritou a ty by měly být testovány nejdříve.

Seznam funkcí, které lze takto upřednostňovat, může být vybírán na základě různých kritérií. Tyto kritéria platí obecně, ne jen na automatizované testování. Kritéria, podle kterých můžeme upřednostňovat, jsou následující:

Od nejvyššího po nejnižší riziko - rizika by se měla zvážit při plánování projektu. Strategie automatizovaného testování pak může být založena na zvýšení priority automatizace pro nejrizikovější funkce software.

Od nejvyšší po nejnižší složitost - jako první se snažíme vyvíjet a testovat složitější funkce, čímž se minimalizuje pravděpodobnost překročení plánu.

- **Potřeby zákazníka** - ve většině projektů je obvykle nutné upřednostňování funkcí dle potřeb zákazníka
- **Rozpočtové omezení** - při prioritizaci funkcí, které jsou určené k automatizaci, je důležité zvážit rozpočet, který byl alokován pro

[12] Prioritizace = upřednostnění

[13] Inkrementální = přírůstková; přírůstkové vydání systému je běžně označované jako "release"

danou verzi. Nějaké funkce budou pro funkčnost software důležitější než jiné.

- **Časové omezení** - při prioritizaci je nutné zvážit časová omezení, která jsou pro danou verzi dána.

- **Personální omezení** - zvážit by se také měli zaměstnanci, kteří jsou k dispozici. V týmu totiž někdy mohou chybět klíčoví zaměstnanci, kteří jsou nezbytní pro výběr upřednostňovaných funkcí. To může být způsobeno rozpočtovými nebo jinými problémy. Při prioritizaci tedy není jen důležité zohlednit "co", ale také "kdo".

3.3.2. Údržba automatizovaných testů

Vývoj řešení automatizovaných testů není triviální záležitost. Testy musí být modulární[14], srozumitelné a spolehlivé. Jsou tedy jako každý jiný softwarový systém, který se musí neustále rozvíjet, a proto je údržba velice důležitým aspektem. Pomocí údržby se automatizované testy přizpůsobí novým verzím testovaného systému, zajišťují možnost použití jiného testovacího prostředí a samozřejmě zabezpečují plynulý a spolehlivý provoz.

Údržba se provádí na existujících a fungujících testech, vykonává se především kvůli potřebným úpravám testů nebo pro zastaralost architektury. Tento proces může být rozdělen na následující kategorie [5]:

- **Preventivní údržba** - změny jsou provedeny tak, aby se architektura automatizovaných testů přizpůsobila více typům testů, testům na více rozhraní, testování více verzí testovaného systému nebo k podpoře automatizace pro nový testovaný systém.
- **Opravná údržba** - změny jsou provedeny s cílem napravit selhání automatizovaných testů.

[14] Modulární - sestavitelný z typizovaných částí

- **Dokonalá údržba** - u tohoto typu je architektura automatizovaných testů optimalizována a nefunkční problémy jsou vyřešeny. Lze se tedy soustředit na výkon automatizovaných testů, jejich použitelnost, robustnost a spolehlivost.
- **Adaptivní údržba** - v dnešní době jsou na trhu neustále nové softwarové systémy (operační systémy, správce databází, webové prohlížeče, atd.) a může nastat situace, kdy je nutné, aby je automatizované testy podporovaly. Proto může být potřebné provést změny za tímto účelem.

3.4. Metriky

Součástí každé dobře zvolené metodiky vývoje software (a také metodiky automatizace testů) je i definice metrik.

Základní definice metriky je standardní měření. Je to systém souvisejících opatření, které usnadňují kvantifikaci některých charakteristik. Používají se k zobrazení minulé a současné výkonnosti nebo také k předpovídání budoucího výkonu. Většina softwarových metrik spadá do jedné ze tří kategorií a to pokrytí, průběh a kvalita. [20]

Metriky pro automatizované testování jsou používány k měření výkonnosti (minulé, přítomné a budoucí) implementovaných automatizovaných testů a souvisejícího úsilí. Metriky lze také rozlišit na ty, které souvisí s unit testy a na ty, které souvisí s integračními a systémovými testy. Metriky automatizovaného testování ale nenahrazují obecné testovací metriky, které měří míru pokrytí, průběhu a kvality, pouze je posilují a doplňují.

Pro dobrou metriku pro automatizované testování se musí jasně stanovit cíle, tedy to, čeho by se mělo dosáhnout. Samozřejmostí by mělo být neustálé sledování a průběžné měření. Na základě výsledků z těchto metrik, pak probíhá rozhodování o změnách v termínech projektu, v seznamu funkcí, v procesní strategii atd. Pro definování cílů si můžeme položit otázky týkající se aktuálního stavu automatizace, jako například[12]:

- Kolik času trvá spuštění všech testů?
- Jaké je pokrytí testů? Je požadováno, aby testy pokrývali jednotlivé části procesu, kódu nebo požadavky na software?
- Jak dlouho trvá analýza dat pro testování?
- Kolik času je potřeba k sestavení scénářů?
- Jak často se spouští vybrané testy?
- Kolik času a systémových prostředků je potřeba ke spuštění vybraných testů?

Dobrá metrika by tedy měla splňovat následující charakteristiky:

- Je objektivní
- Je měřitelná
- Je smysluplná
- Data pro ni jsou jednoduše shromážditelná
- Pomáhá identifikovat oblasti automatizace vhodné ke zlepšení
- Je jednoduchá

Generovat metriky je důležité pro kalkulaci hodnoty automatizace, speciálně v situaci, kdy je přístup automatizace v projektu použit poprvé. Tyto metriky jsou pak používány při výpočtu návratnosti investic, tzv. ROI[15] index. Této problematice se věnuje následující kapitola. Testovací tým by měl měřit čas strávený na vývoji automatizace a spouštění testovacích skriptů a porovnat jej s výsledky, které byly pomocí testů vyprodukovány. Například testovací tým porovnává počet hodin, které byly stráveny při vývoji a spouštění, s počtem nalezených chyb, které by pravděpodobně nebyly nalezeny během manuálního testování.

Dalším způsobem jak kvantifikovat nebo měřit výhody automatizace je prokázání, že nějaký specifický testovací případ, by se těžko pokrýval manuálním testováním, ale pomocí automatizace je jeho realizace prokazatelně jednodušší. Například některé chyby by nebyly během manuálního testování objeveny a lze je nalézt až díky automatizaci. Jde třeba o zátěžové testování, kdy nezanedbatelně velký počet virtuálních uživatelů (běžně v řádech tisíců) spouští specifické funkce

[15] ROI - Return On Investments - návratnost investic

systému ve stejném časovém okamžiku a ověřuje tak, zda je systém stabilní i při takovémto zatížení. Provedení takového testu manuálně by mělo netriviální dopad do nákladů projektu vzhledem k nutné akvizici dostatečného počtu testerů.[12]

Testovací úsilí lze dále minimalizovat použitím nástrojů, které samostatně nahrávají do testovacích skriptů vstupní data. Metrika, která toto řeší, měří čas potřebný k ručnímu nastavení v porovnání s časem nutným k nastavení automatizovaného nástroje, který bude data nahrávat. Například pokud má systém umožnit přidání až 10 000 účtů. Aby bylo možné takový požadavek otestovat, tak by se muselo buď ručně založit 10 000 účtů, nebo si vytvořit automatizovaný skript. Tento skript lze snadno podpořit tím, že informace k účtu budou načítány ze souboru. Datový soubor se pak jednoduše vytvoří pomocí generátoru dat. Snaha ověřit takovýto požadavek, zabere mnohem méně odpracovaného času při provedení automatizace, než při manuálním testování. Tento čas pak právě porovnává metrika zabývající se touto problematikou[26].

Při procesu automatizace lze použít i následující metriky, které jsou rozlišeny na metriky týkající se testovacích případů a pokroku a na metriky týkající se chyb a jejich odstraňování.

3.4.1. Index automatizace

Index automatizace je definován jako procentuální podíl testovacích případů, které mohou být automatizovány. To je reprezentováno následující rovnicí:

$$PA(\%) = \frac{ATC}{TC}$$

PA = Percent automatable = index automatizace
ATC = Number of test cases automatable = počet automatizovatelných případů
TC = Total number of test cases = Celkový počet testovacích případů

V případě dostatečných finančních prostředků lze většinu testovacích případů označit za automatizovatelné. Tato situace ale nenastává a za

neautomatizovatelné označujeme například testovací případy týkající se částí aplikace, které jsou stále ve fázi návrhu, nejsou příliš stabilní a samozřejmě také ty, které nemá smysl automatizovat. Smysl automatizovat mají především testovací případy, jejichž automatizace by přinesla vysokou návratnost investic.[12]

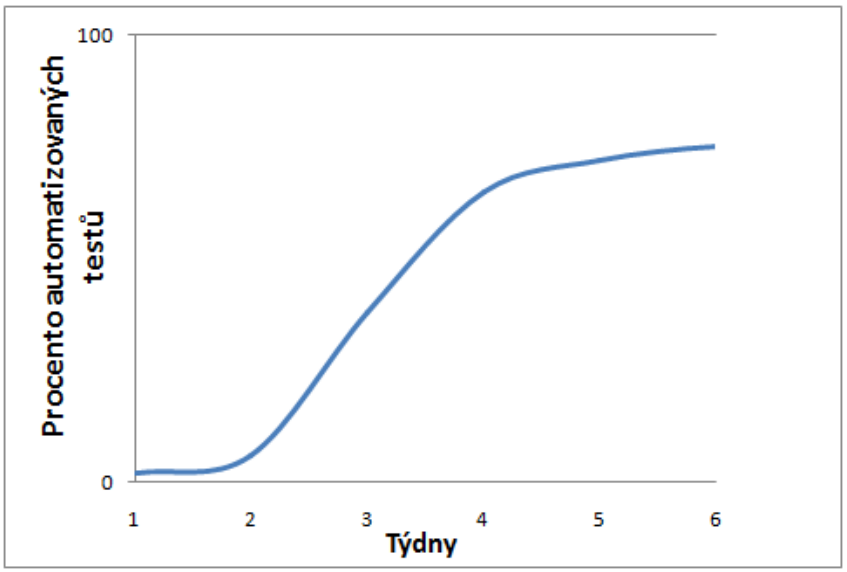
3.4.2. Průběh automatizace

Průběh automatizace udává, kolik procent automatizovaných testovacích případů existuje vůči počtu testovacích případů, které jsou automatizovatelné v daném časovém okamžiku. Tato metrika tedy vyjadřuje, jak dobře jsou plněny cíle automatizace. Cílem je automatizovat 100% automatizovatelných testovacích případů. Tuto metriku je důležité sledovat v průběhu vývoje automatizace.

$$AP\ (\%) = \frac{AA}{ATC}$$

AP = Automation progress - průběh automatizace
AA = Number of test cases automated - počet automatizovaných testovacích případů
ATC = Number of test cases automatable - počet automatizovatelných testovacích případů

Průběh automatizace je typická metrika sledovaná v čase, jejíž zobrazení lze vidět v následujícím grafu.[12]



Obrázek 13 - Procento automatizovaných testů. Vlastní tvorba inspirovaná z [12].

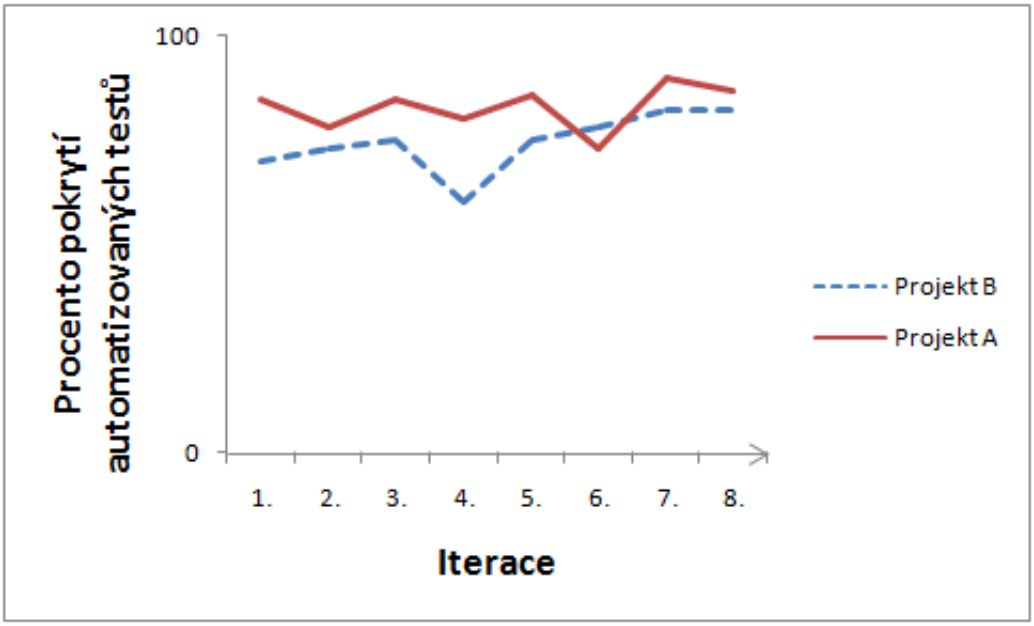
3.4.3. Procento pokrytí automatizovaných testů

Další metrika slouží ke stanovení procenta vyjadřujícího, jakého pokrytí bylo automatizací dosáhnuto. Jedná se o metriku, která indikuje úplnost testování. Tento ukazatel nepojednává o měření toho, kolik automatizace bylo provedeno, ale o tom, jak velká část funkcionalit byla pokryta. Procento pokrytí automatizovaných testů nespecifikuje efektivnost probíhajícího testování, ale určuje jeho rozměr.

$$PTC\ (\%) = \frac{AC}{C}$$

PTC = Percent of Automatable testing coverage - procento pokrytí automatizovaných testů
AC = Automation coverage - pokrytí automatizací
C = Total Coverage - celkové pokrytí (požadavky, jednotky/komponenty, pokrytí kódu atd.)

Následující obrázek ukazuje procento pokrytí pro projekt A ve srovnání s projektem B během různých iterací. U křivky pro projekt B lze například vidět, že po 3. iteraci byla dodána nová funkcionalita, která ještě nestihla být otestována při iteraci 4. a pokryta byla až v iteraci 5.



Obrázek 14 - Procento pokrytí automatizovaných testů. Vlastní tvorba inspirovaná z [12].

U měření pokrytí se tedy musí rozlišovat celkový počet vyvinutých testovacích postupů a celkový počet stanovených požadavků. Hloubka pokrytí se obvykle určuje na základě akceptačních kritérií. Pokud se testují kritické systémy, jako jsou například různé softwary sloužící k medicínským nebo bankovním účelům, musí být pokrytí opravdu vysoké oproti ne-kritickým systémům. Případná vyšší chybovost ne-kritických systémů nemá tak závažné dopady, jelikož mají jen pár set koncových uživatelů. [11]

3.4.4. Hustota chyb

Měření chyb by se mělo provádět bez ohledu na to, zda je, nebo není pro testování systému použit přístup automatizovaného testování. Je to ale jedna z metrik, která může být použita také k určení oblasti vhodné pro automatizaci. Předmětem automatizace se například může stát nějaká část, která vyžaduje mnoho přetestování kvůli vysoké hustotě chyb. Hustota chyb je míra celkového počtu známých chyb vůči velikosti entity systému, který je měřen. Pomáhá například v situacích, kdy je zjištěna vysoká hustota chyb v určité funkcionalitě. Po tomto zjištění je pak nutné provést analýzu příčin, kdy si odpovídáme na otázky typu[12]:

- Je funkcionalita příliš složitá, a proto bylo objeveno tolik chyb?
- Je problém v návrhu nebo v implementaci funkcionality?
- Nebyla pochopena složitost funkcionality?
- Je za chyby zodpovědný vývojář a je potřeba ho více proškolit?

$$DD = \frac{D}{SS}$$

DD = Defect density - hustota chyb
D = Number of known defects - počet známých chyb
SS = Size of software entity - velikost entity systém

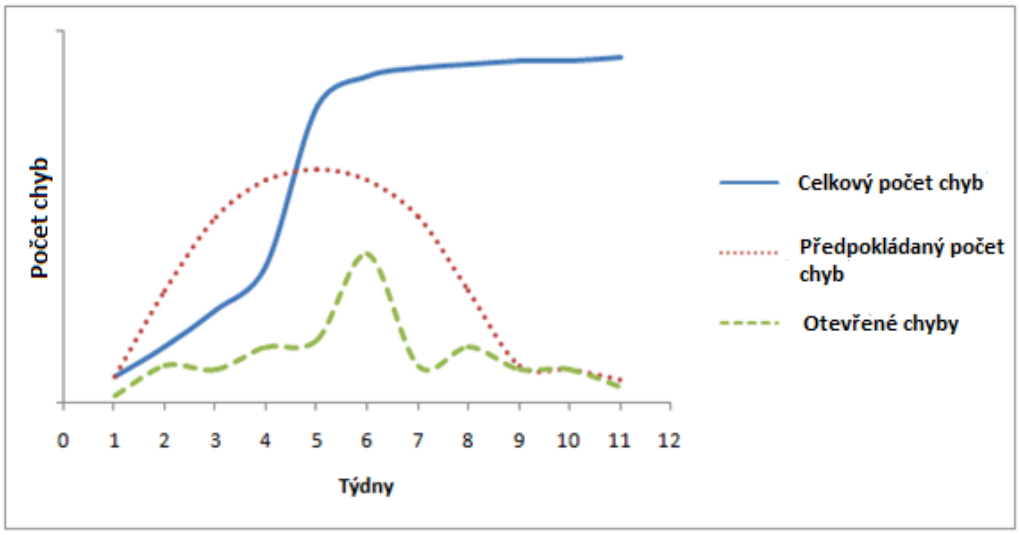
3.4.5. Analýza trendu chyb

Další důležitou metrikou je analýza trendu chyb, která může pomoci určit trend zjištěných chyb v průběhu času. Odpovídá na otázku, zda se počet chyb snižuje, zvyšuje nebo zůstává statický.

$$DTA = \frac{D}{TPE}$$

DTA = Defect trend analysis - analýza trendu chyb
D = Number of known defects - počet známých chyb
TPE = Number of test procedures executed over time - počet provedených testovacích procedur

Analýza trendu chyb tedy prezentuje zdraví projektu. Jedna z možností, jak tuto metriku zobrazit graficky v průběhu času, je znázorněna v následujícím obrázku[12].



Obrázek 15 - Počet chyb. Vlastní tvorba inspirovaná z [12].

3.4.6. Efektivnost odstraňování chyb

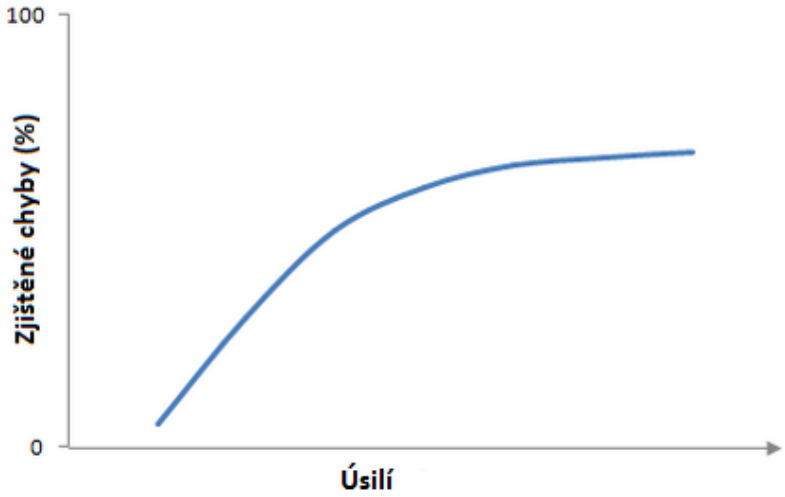
Jedna z nejoblíbenějších metrik je efektivnost odstraňování chyb. Tato metrika sice není specifická pro automatizace, ale může být při automatizaci velmi užitečná. Nepřímo také měří kvalitu produktu a efektivitu úsilí při odstraňování chyb. Hodnota této metriky je vyjádřená procentuálně, a čím vyšší toto procento je,

tím efektivnější odstraňování chyb je. Jinými slovy udává, kolik procent chyb bylo identifikováno a vyřešeno během vývoje a testovací fáze, tedy před dodáním zákazníkovi [12].

$$DRE = \frac{DT}{DT + DA}$$

DRE = Defect Removal Efficiency - Efektivnost odstraňování chyb
DT = # of defects found during testing - počet chyb nalezených během testování
DA = # of defects acceptance defects found after delivery - počet chyb nalezených po dodání

Nejvyšší dosažená hodnota DRE může být 1, což se rovná 100%, to ale v praxi není pravděpodobné. Tato metrika by měla být měřena během různých fází vývoje.



Obrázek 16 - Zjištěné chyby. Vlastní tvorba inspirovaná z [12].

3.5. Buisiness case a ROI

Business case je v překladu obchodní případ, jenž na webu BusinessDictionary [8] definují jako typ rozhodovacího nástroje, který pomáhá určit, jaký vliv na ziskovost budou mít určitá rozhodnutí. Business case by měl také ukázat, jak určitá rozhodnutí změní peněžní toky v průběhu času a jak se změní náklady a příjmy. Příkladem může být analýza finančních výsledků pro firmu při výběru jiného dodavatele.

Na webu Project Management Docs [9] popisují dobrý business case jako takový, který zachycuje a dokumentuje důvody pro zahájení nového projektu. Dále pomáhá určit, zda se vyplatí investice do nového projektu provedením analýzy nákladů a přínosů navrhovaného řešení. Zabývá se i různými alternativami možných řešení.

Existuje řada definic obchodního případu a všechny se shodují, že se jedná o dokument, který analyzuje návratnost investic do nového projektu či do nového řešení. V automatizaci testů tuto návratnost investic pomáhá vyjádřit metrika ROI.

3.5.1. Return on investment (ROI)

Profesionální automatizace nevyžaduje jen rozvoj automatizovaných skriptů, ale také odůvodňuje jejich použití. Kromě toho zhodnocuje snahu o automatizaci a v reálném čase umožňuje rozhodování o rozsahu pokrytí aplikace automatizovanými testy. Toto rozhodování je podpořené právě metrikou ROI. Výpočet návratnosti investic může být požadován před, během, ale i po provedení automatizace. Může tak být i kontrolována efektivnost automatizace, tedy zda se podařilo uspořit náklady, zvýšit výkonnost a kvalitu aplikace.

Při měření návratnosti investic z automatizace je potřebné zhodnotit celkové náklady a přínosy z automatizování. Pokud jde o náklady, tak jsou brány v úvahu například celková ohodnocení úsilí týkající se vývoje, implementace, infrastruktury, náklady na různé licence, podporu a školení, dodatečné testy automatizace a především náklady na údržbu.

Přínosy, tedy ušetřené finance, které automatizace přináší, mohou být například uspořený čas a využití menšího počtu lidí, kteří jsou do testování zapojení. Přínosy lze zhodnotit a zkalkulovat porovnáním množství času potřebného pro manuální testování a času potřebného pro použití již vytvořených automatizovaných testů. Mezi nefinanční přínosy automatizace patří vyšší kvalita testovaného softwaru a dobrá pověst dodavatele softwaru u zákazníka.

Po zhodnocení a porovnání těchto nákladů a přínosů je zjevné, zda investice do automatizace má, nebo by měla mít, smysl. Obecně může být podle velikosti projektu složitost ROI kalkulace od velmi podrobné a rozsáhlé až po poměrně jednoduchou a přímočarou. Základní výpočet celé kalkulace vychází z následujícího předpisu[26][12]:

$$ROI = \frac{čistý\ zisk - počáteční\ investice}{počáteční\ investice} * 100[\%]$$

4. Framework

Důležitou oblastí v procesu automatizace je zakomponování Frameworku do procesu. Vytvoření Frameworku může totiž kladně ovlivňovat nákladovou stránku projektu.

Náklady se všeobecně dělí na fixní a variabilní. Fixní náklady jsou v kontextu automatizovaných testů prvotní náklady na vytvoření základní sady testů. Z praxe víme, že ty se dají relativně přesně na začátku projektu odhadnout a vypočítat. Variabilní náklady se projevují při dlouhodobém udržování vytvořených testů a jsou proto rizikovější. Tyto náklady rostou v čase a s množstvím testů a je tedy nutné se zaměřit na jejich udržení na definované úrovni.

Na automatizované testy se lze dívat jako na klasický vývoj softwaru, jde vlastně o naprogramovanou testovací aplikaci sloužící k reálnému ověření samotné business aplikace. Jednou z možností jak efektivně testy realizovat je využít některé techniky z programátorského světa, například tzv. „frameworkizace".

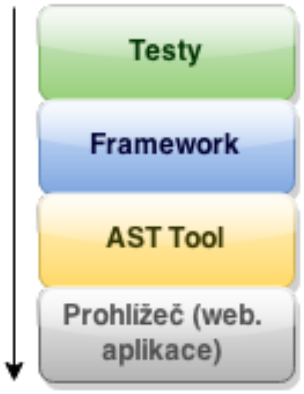
Framework lze definovat mnoha způsoby a jedním z nich je tato definice[18]:

„Framework (aplikační rámec) je softwarová struktura, která slouží jako podpora při programování a vývoji a organizaci jiných softwarových projektů. Může obsahovat podpůrné programy, knihovny API, podporu pro návrhové vzory nebo doporučené postupy při vývoji. Cílem frameworku je převzetí typických problémů dané oblasti, čímž se usnadní vývoj tak, aby se návrháři a vývojáři mohli soustředit pouze na své zadání."

Framework, tedy podporu pro programátory, má význam vytvářet pro systémy, u kterých lze předpokládat jejich následné časté využívání. Vzhledem k tomu, že v dnešní době je vývoj uživatelského softwaru směřován pro webové prohlížeče, tak i vytvoření frameworku a tím zjednodušení automatizovaného testování bude směrované právě na tuto platformu aplikací. Zaměření se na testování webových aplikací podporuje také moderní trend předělávání tradičních

desktopových aplikací do internetových prohlížečů, viz docs.google.com, draw.io a další.

Architekturu automatizace testů webových aplikací s využitím frameworku znázorňuje následující obrázek:



Obrázek 17 - Architektura automatizace testů. Vlastní tvorba inspirovaná z [25].

Základní myšlenkou je v maximální možné míře oddělit testy od testované aplikace. Jak je již naznačeno na obrázku, mezi samotnou aplikací a testy se nachází vrstva AST Tool, která zabezpečuje technologickou vrstvu testů a tzv. framework, který zabezpečuje právě oddělení některých technických parametrů (např. ID políčka v HTML kódu, ID stránky, případně samotný page flow[16]) aplikace od samotných testů. Tento princip umožňuje dodržet jeden ze základních principů automatizace testů a to udržet samotné testy co nejjednodušší.

Jednoduchost testů je nutná proto, aby testy byly lépe čitelné a srozumitelné pro testera. U složitého testu hrozí riziko, že v situaci kdy test neprojde, tester nerozezná chybu testu od chyby aplikace a může mít tendenci tento složitý test označit za nefunkční a nebrat jeho výsledky v úvahu.

Z hlediska programování bychom měli dodržet princip, že by hlavní programátorská složitost neměla přesáhnout hranice Frameworku. Testy by měly zůstat i z pohledu čitelnosti jejich kódu co nejjednodušší.

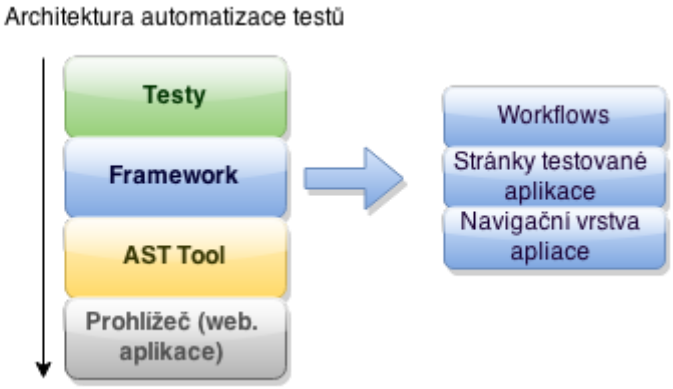
[16] Page flow - souslednost obrazovek ve webové aplikaci

Rozdělení architektury testů do zmíněných vrstev má další nespornou výhodu, která spočívá v efektivitě a rychlosti úprav. Například při změně některých technických parametrů aplikace pak stačí tuto změnu implementovat na jediném místě (ve vrstvě Framework) a není potřeba upravovat desítky, případně stovky, testů.

Jednou z typických služeb Frameworku pro automatizaci testů je také umožnění zakomponování testů do Continuous integration[17] nástrojů. Touto službou je zabezpečeno pravidelné využívání vytvořených testů a průběžné čerpání benefitů z automatizace.

I z hlediska technické implementace testů jsou implementovány dvě části. V jednom jsou naprogramované samotné testy a ve druhém Framework. Testovací projekt neinteraguje přímo s AST Tool, ale pouze přes Framework.

Samotný Framework se většinou dále vrství. Jednu z možností jak logicky rozvrstvit Framework znázorňuje následující obrázek.



Obrázek 18 - Architektura automatizace testů - vrstvy Frameworku. Vlastní tvorba inspirovaná z [25].

Spodní vrstva se stará o navigační prvky aplikace, vrstva nad ní o celou stránku aplikace a u rozsáhlejších aplikací je výhodné stránky seskupovat do tzv. workflows. Tyto vrstvy ještě dále zjednodušují tvorbu samotných testů. Funkcionality aplikace se dají efektivně dělit do stránek (případně až workflows) a na jednotlivých stránkách se testy dají stavět na úrovni business funkcionalit (např.

[17] Continuous integration - průběžná integrace

„*P*řihlášení uživatele", „Změna hesla", „Potvrzení objednávky" atd.). Není vhodné na jednotlivých stránkách otestovat všechny komponenty a varianty, doporučujeme se zaměřit na klíčovou business funkcionalitu dané stránky (např. na stránce „Registruj uživatele" je potřebné otestovat samotnou registraci a ne vedlejší funkčnosti typu překliknutí na číslo verze aplikace). [25]

Při tvorbě architektury automatizovaných testů včetně architektury samotného frameworku je třeba vhodně přizpůsobit komplexitu Frameworku ke složitosti aplikace, četnosti změn různých technických parametrů aplikace a zároveň také k životnosti samotné aplikace. Je tedy důležité dbát na to, aby nebyl vytvořen složitější Framework pro testování, než je složitost testované aplikace.

Podobně jako při běžných vývojových frameworkách, i při frameworku pro automatizaci testů bývá standardní součástí logování a reporting. Logování se běžně dělí na více úrovní – v nejúspornějším režimu v úrovni „error" (z hlediska potřeby diskového) se logují jenom chyby, další úroveň bývá tzv. „warning" (logují se již i varování) a nejvyšší pak „info" (logují se podrobné informace o běhu). Logování je užitečné z důvodu dohledání případných chyb v naprogramovaných testech, resp. slouží jako dokumentační nástroj běhu testů. Reporting, oproti logování, přináší uživatelsky příjemnější/čitelnější formu výstupů o běhu testů a různé pohledy na shromážděné výsledky testů. V závislosti od použitého nástroje a struktury ukládaných informací se dají připravit jednoduché reporty typu „test prošel/neprošel" až po složité analýzy stavu testované aplikace zpestřené různými tabulkami a grafy.

5. TestLink

Teoretické principy, které jsou popsány v předchozí části práce, budou aplikovány při testování webové aplikace TestLink. Před popsáním samotného procesu automatizace je nutné uvést krátký popis nástroje.

TestLink je nástroj, který slouží pro správu a organizaci v oblasti testování. Jde o open-source[18] software, který je zdarma a veřejně dostupný. Uživatelé k němu přistupují přes webový prohlížeč a nemusí si samotný TestLink instalovat do svého PC. Vhodné webové prohlížeče jsou např. Internet Explorer, Mozilla, Firefox a Chrome.

V praxi existuje mnoho projektů, kdy vytváříme až stovky testovacích scénářů a tento nástroj umožňuje implementovat strategii testování a podpořit řízení testovacího procesu v průběhu příprav a realizace testování. Uživatelům je možné definovat určité role, podle kterých se určují funkcionality, ke kterým budou mít přístup. TestLink obsahuje komponenty, které umožňují:

- vytváření plánu procesu testování,
- psaní testovacích scénářů
- organizaci testovacích scénářů do hierarchické struktury
- zaznamenávání softwarových požadavků
- zapisování průběhu vykonávání testovacích scénářů
- zápis výsledků proběhlého testování a reporting těchto výsledků.

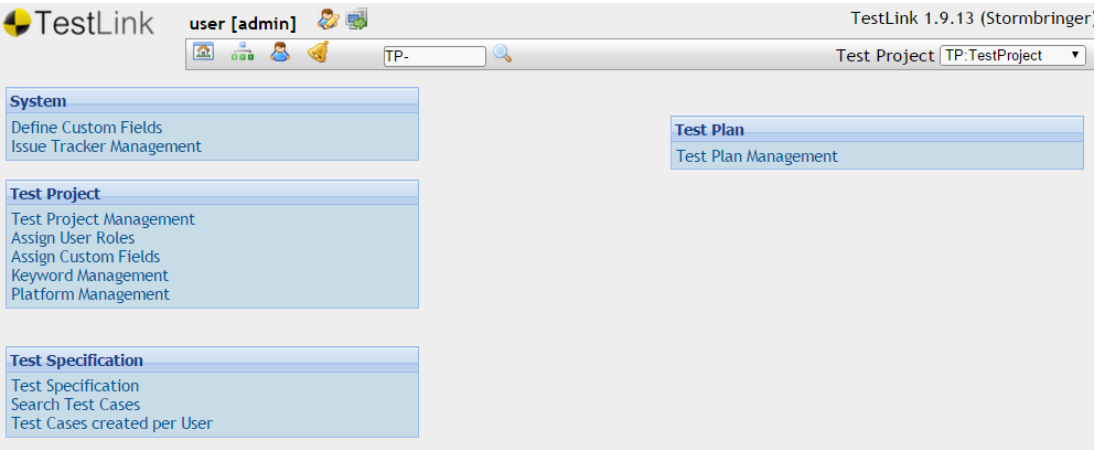
Chyby zjištěné v průběhu testování je nutné předat k opravě. K tomu slouží programy pro správu chyb. Program TestLink umožňuje spolupráci s některými takovými programy. Konkrétně to jsou:

- Bugzilla
- JIRA
- TrackPlus
- A další…

[18] open-source software - počítačový software s otevřeným zdrojovým kódem

Mezi další funkcionality systému patří možnost určité testy prioritizovat a přiřazovat ke konkrétním testerům. Dále je zde možnost reporty odeslat emailem a exportovat do různých formátů jako např. HTML, MS Word nebo MS Excel.[30]

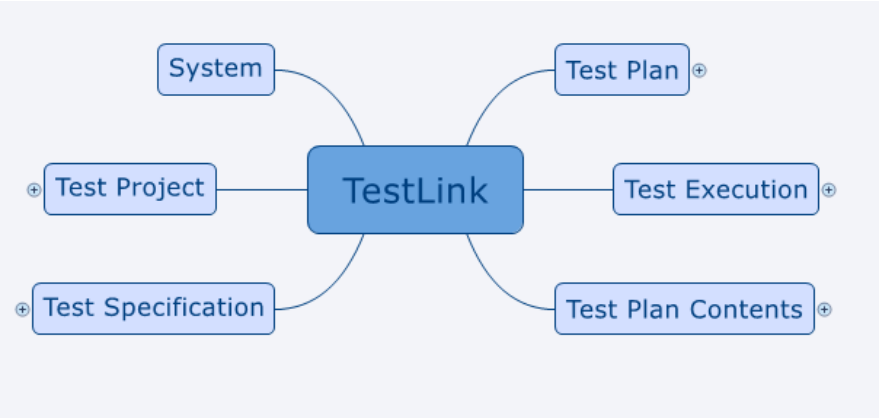
Na následujícím obrázku je znázorněna úvodní obrazovka aplikace. Jde o verzi 1. 9. 13, která byla použita pro účely této práce.



Obrázek 19 - Úvodní obrazovka aplikace TestLink. Vlastní tvorba.

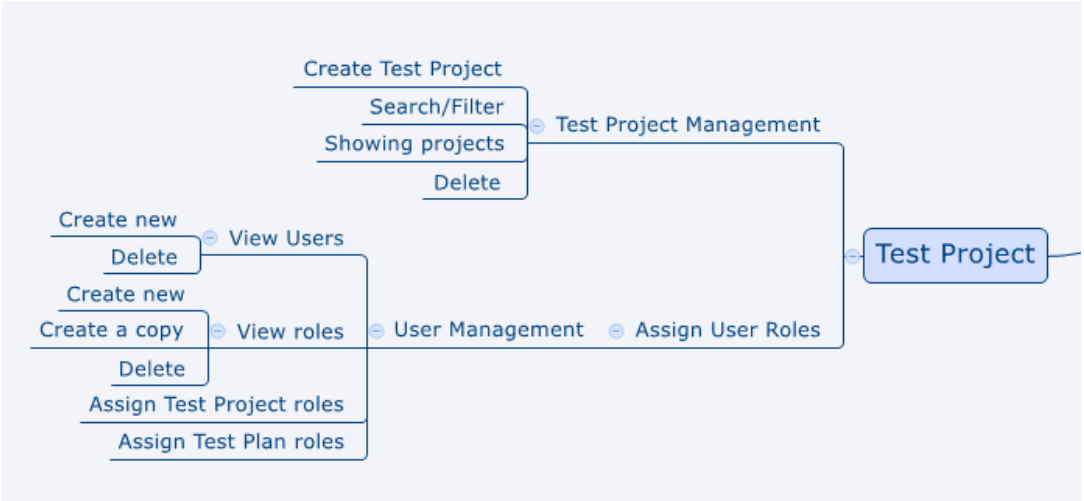
5.1. Mind mapa TestLinku

Pomocí jednoduché myšlenkové mapy jsou představeny funkcionality, vztahy a souvislosti Test Linku. Cílem této podkapitoly není pokrýt a zobrazit celou funkčnost aplikace, cílem je pouze představit základní možnosti, které nástroj nabízí, seznámit se tak s TestLinkem a zorientovat se v něm. V roli administrátora jsou tedy k dispozici na úvodní obrazovce sekce System, Test Project, Test Specification a Test Plan. Po vytvoření testovacího plánu se pak zobrazí ještě sekce Test Execution a Test Plan contents.



Obrázek 20 - Struktura úvodní obrazovky aplikace TestLink. Vlastní tvorba.

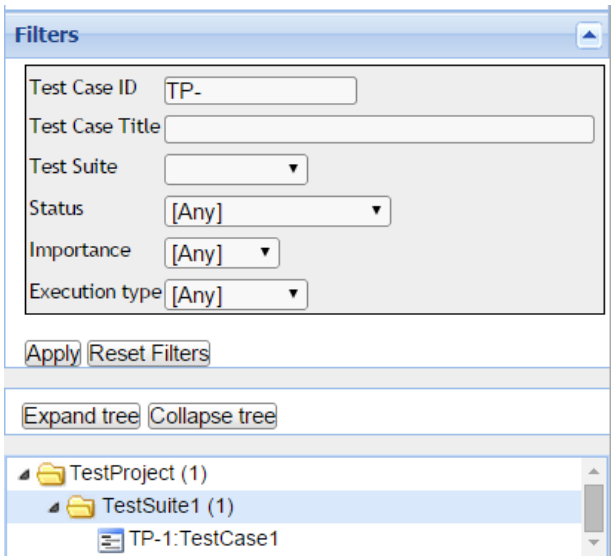
Sekce Test Project umožňuje na záložce Test project management vytvoření nového projektu, zobrazení vytvořených projektů, dále nabízí jejich vyhledávání pomocí filtru a samozřejmě také mazání již vytvořených projektů. Záložka Assign User Roles obsahuje vše, co se týká uživatelů nástroje a rolí, které zde těmto uživatelům lze přidělit. Je možné zde také editovat přidělená práva k jednotlivým rolím a přidělovat konkrétní uživatele ke konkrétním projektům, které jsou v nástroji vytvořeny. Na následujícím obrázku je znázorněna struktura sekce Test Project.



Obrázek 21 - Struktura sekce Test Project aplikace TestLink. Vlastní tvorba.

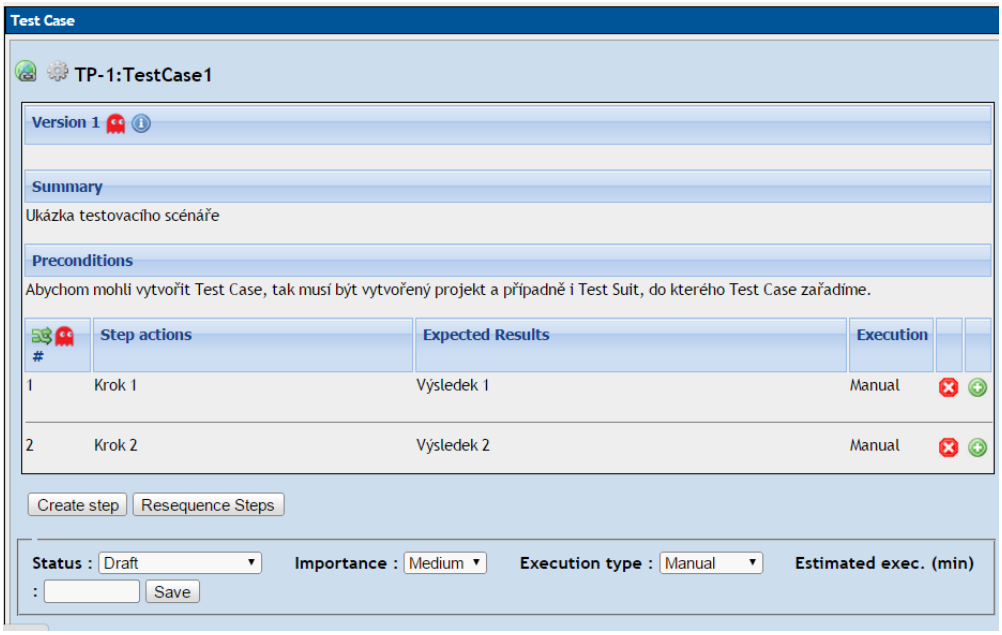
Další sekce je Test Specification. Ta je určená k zobrazení hierarchické struktury projektů a je zde také filtr k vyhledávání jednotlivých projektů, testovacích případů atd. Tato sekce zajišťuje všechny základní operace, které se

týkají správy testovacích případů a jejich seskupování do Test Suite a samozřejmě také vytváření hierarchie projektu.



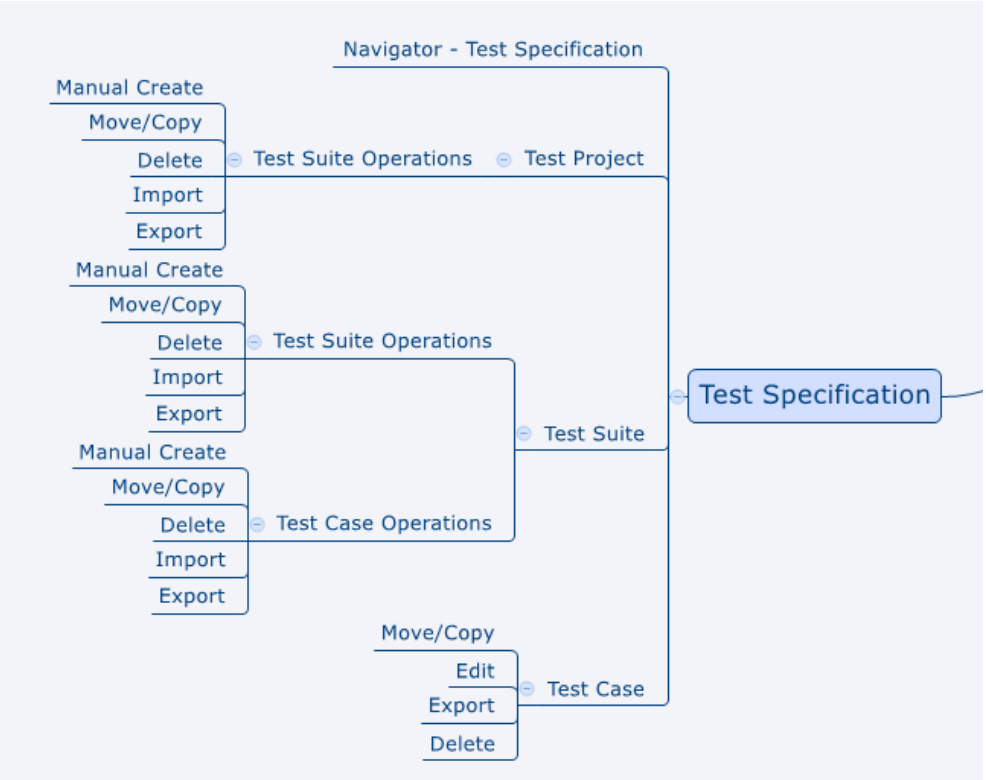
Obrázek 22 - Obrazovka Test Specification - Filtry. Vlastní tvorba.

V daných testovacích případech jsou již zaznamenávány konkrétní testovací scénáře a potřebné požadavky k vykonání daného scénáře.



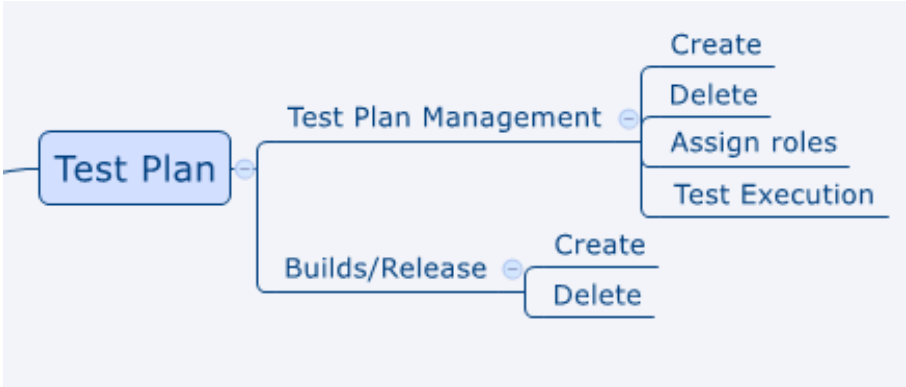
Obrázek 23 - Obrazovka Test Specification - Test Case. Vlastní tvorba.

Na následujícím obrázku je znázorněna základní strukturu sekce Test Specification.



Obrázek 24 - Struktura sekce Test Specification. Vlastní tvorba.

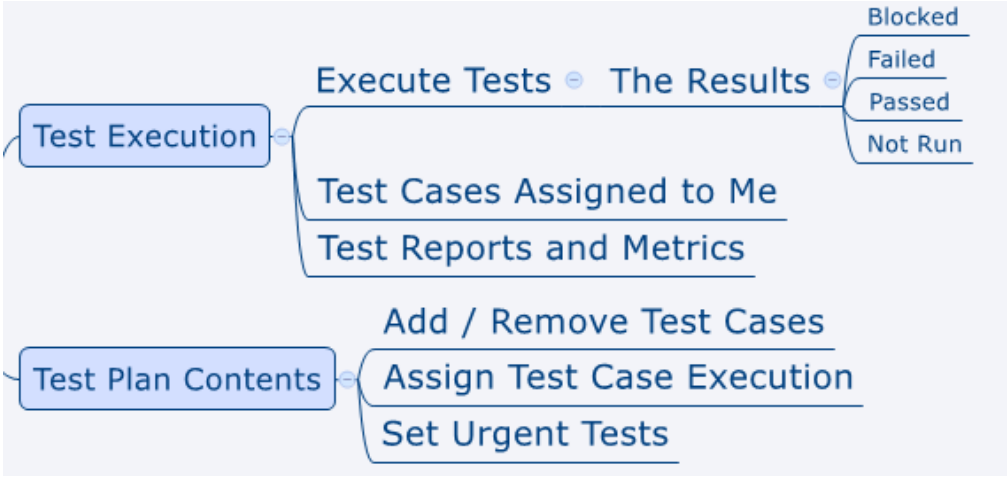
V sekci Test Plan jsou spravovány testovací plány. Je možné zde vytvářet, editovat a mazat testovací plány. Dále je zde k dispozici odkaz na obrazovku, kde se dají uživatelům přidělovat role pro daný plán a odkaz, který směruje na obrazovku, kde jsou zaznamenávány výsledky z vykonaných scénářů. Po vytvoření plánu je získána možnost vytvořit nové Buildy/Release, což umožňuje do aplikace zaznamenat, ke kterému releasu se dané scénáře v testovacím plánu vztahují, do kdy má být release odevzdaný a do kdy musí být aplikace dotestovaná. Tyto data jsou pak zobrazována na potřebných místech v aplikaci. Díky těmto nastavením je možné sledovat, jestli proces testování časově běží podle očekávání.



Obrázek 25 - Struktura sekce Test Plan. Vlastní tvorba.

Poslední dvě sekce jsou Test Execution a Test Plan Contents. Obě sekce se zobrazí až ve chvíli, kdy je vytvořený testovací plán. Přes sekci Test Execution se získává přístup k obrazovce, kde se k jednotlivým scénářům zaznamenávají výsledky z provedení konkrétních kroků a samozřejmě také výsledek celého testu. Lze si zde zobrazit testovací případy, které jsou danému uživateli přiděleny. Dále je odsud dostupná obrazovka týkající se správy reportů a metrik.

V sekci Test Plan contents jsou přidávány testovací případy ke konkrétním plánům, nastavována důležitost testů a další funkcionality, které zpřehledňují a lépe organizují testovací plán.



Obrázek 26 - Struktura sekcí Test Execution a Test Plan Contents. Vlastní tvorba.

6. Nástroje pro automatické testování webových aplikací

Při automatizovaném testování je nutné využívat nástroje, které automatizované testování podporují. Těchto nástrojů existuje celá řada a v následujících podkapitolách jsou krátce představeny ty, nad kterými bude uvažováno pro vývoj automatizovaných testů pro aplikaci TestLink.

6.1. AutoIt

Jedná se o freeware[19] a je určen pro platformu MS Windows. Původně se jednalo primárně o nástroj pro vytváření automatizovaných skriptů pro MS Windows. Nicméně postupem času se nástroj vyvinul do plnohodnotného testovacího nástroje a umožňuje testování uživatelského rozhraní.

Aktuální verze nástroje podporuje tvorbu testovacích skriptů v programovacím jazyku Basic. Výhodou tedy je, že zdrojový kód testu lze kompilovat do samostpustitelného kódu, který lze spustit i na počítačích, které nemají instalováno prostředí pro tvorbu testů. Vývoj je realizován v IDE SciTE.

Za nevýhodu může být považováno právě přímé propojení s programovacím jazykem. Zdrojový kód pro test tak pozbývá jednoduchou čitelnost a vytvoření testů vyžaduje zkušeného programátora.[3]

6.2. HttpUnit

Tento nástroj je přímo určený pro testování webových aplikací. Nástroj přímo simuluje práci internetového prohlížeče včetně funkce odesílání formulářů, práce JavaScriptu, nebo http autentizace. Umožňuje tak psát testy přímo v programovacím jazyku JAVA. Výhodou je i snadná integrace s JUnit, čehož lze dobře

[19] Freeware - volně dostupné softwary

využít v případě automatického spouštění testů například v rámci Continuous integration(CI)[20] apod.

Nevýhodou nástroje je skutečnost, že jeho vývoj byl zastaven v roce 2008. Nástroj tedy nepodporuje vlastnosti nového standardu HTML5. Tato skutečnost není ve vztahu k záměru testovat aplikaci TestLink nijak zásadní, protože aplikace je optimalizována i pro starší prohlížeče a nové vlastnosti nevyužívá, nicméně z dlouhodobého hlediska by se mohlo jednat o zásadní problém a tato skutečnost by mohla vést k nutnosti převodu všech testů na jiný nástroj. [17]

6.3. JMeter

Jedná se o nástroj z rodiny aplikací vyvíjených pod hlavičkou firmy APACHE. Nástroj je určen primárně pro testování výkonnosti. Lze testovat jak dynamický, tak statický obsah. Velkou výhodou tohoto nástroje je možnost simulace souběžné práce několika uživatelů najednou. Naopak nevýhodou nástroje je naproti tomu neschopnost spouštět JavaScript na cílových webových stránkách.[2]

6.4. Selenium IDE

Jedná se o nástroj typu „zachyť a přehraj“ (viz. Kapitola 3.2.1.). Tester může jednoduchým způsobem nahrát test do počítače, následně doplnit požadované validace, případně modifikovat test a pak již pouze přehrát test znovu. Zároveň může test exportovat do některého z programovacích jazyků a převést tak test přímo do nástroje Selenium Webdriver. Výhodou je jednoduchost používání tohoto nástroje. Nevýhodou je poněkud obtížná orientace v definici testů. To zásadním způsobem ztěžuje udržovatelnost takových testů. Jakákoliv úprava testované aplikace tak znamená poměrně zásadní pracnost na úpravách již existujících testů. Další nevýhodou je skutečnost, že nástroj existuje pouze pro internetový prohlížeč Firefox. Případný převod do podoby umožňující spuštění na jiných prohlížečích

[20] Continuous integration - průběžná integrace

znamená práci navíc. Poněkud obtížnější by mohla být i integrace do Continuous integration.[10]

6.5. Selenium Webdriver

Jde o Selenium 2.0, které se skládá ze Selenia IDE 1.0 a Selenia Serveru. Je to nástroj určený přímo pro testování GUI webových aplikací. Základem nástroje je tzv. Webdriver. Jedná se o jakéhosi prostředníka, pomocí kterého lze ovládat reálný internetový prohlížeč v počítači. Zdrojový kód testů je možné definovat hned v několika programovacích jazycích a existuje i převodník z testů v Selenium IDE (viz předchozí kapitola) přímo do některého z programovacích jazyků.

Za výhody tohoto nástroje lze považovat definici testu přímo v programovacím jazyku, což dává testerům značnou volnost v dalším rozvoji definovaných testů. Například k přechodu k datově řízeného testování apod. Zároveň nástroj umožňuje spouštět testy na vzdálených počítačích. To je výhodné zejména pro zrychlení testů pro několik internetových prohlížečů, protože testy mohou běžet paralelně na několika různých počítačích.

Nevýhodou tohoto řešení je jednak znepřehlednění definice testu, neboť je definován jako zdrojový kód aplikace, a také ne zcela jednotné rozhraní na jednotlivé internetové prohlížeče. V některých případech je tak třeba testy modifikovat pro konkrétní prohlížeč nebo verzi prohlížeče.[10]

6.6. Sikuli

Jedná se o specializovaný nástroj pro testování GUI bez ohledu na platformu. Je určen pro testování obecně jakýchkoliv aplikací. Může se tedy jednat například o tlusté klienty apod.

Je založen na algoritmu rozpoznávání tvarů na obrazovce. Obrazovka je chápána jako jeden velký obrázek. Tester pak definuje aktivity pomocí částí obrázků a aktivit, která se má provést. Například pro test vyhledávání pomocí GOOGLE tester

sejme z obrazovky obrázek tlačítka s ikonou lupy a v aktivitě definuje „klikni 50 px vlevo od této ikony“. Nástroj na obrazovce vyhledá odpovídající tvar (tlačítko) a klikne do pole pro vyhledávání.

Výhodou tohoto nástroje je možnost testovat „cokoliv co je na obrazovce“. Tento nástroj tedy nemá problém s testováním částí aplikací v různých technologiích.

Nevýhodou je snímání obrazovky. Jedná se o poměrně náročný proces i pro zběhlého programátora. Další nevýhodou je právě rozpoznávání částí obrazovky podle grafické shody. Jednak se jedná o časově náročnou operaci, což prodlužuje běh testu a jednak je metoda zvláště u internetových prohlížečů poněkud náchylná k chybám. Stačí změna velikost textu v internetovém prohlížeči a test nemusí doběhnout.[16]

6.7. Canoo WebTest

Canoo WebTest je nástroj pro definici a spouštění automatických testů s možností deklarace testovacího scénáře ve formě XML nebo v programovacím jazyku Groovy. Je založen na emulaci[21] internetového prohlížeče přímo v rámci nástroje.

Výhodou je přehlednost, jednoduchost a rychlost. Další výhodou je, že test může běžet bez uživatelského rozhraní.

Nevýhodou je neúplná podpora JavaScriptu. Další nevýhodou je pak silná závislost na správném formátu HTML.[15]

[21] Emulace - napodobení činnosti jednoho zařízení pomocí jiného zařízení

7. Test analýza

Principy automatizovaného testování budou v této diplomové práci aplikovány na nástroji TestLink. Hlavním cílem je navrhnout a vytvořit sadu automatizovaných skriptů pomocí vhodných metod, přístupů a běžně dostupných nástrojů. Pro úspěšné splnění tohoto cíle bude postupováno podle jednotlivých fází metodiky automatizovaného testování (viz. Kapitola 3.3). Nejprve je vhodné stručně shrnout plán postupu při vývoji automatizace a až následně bude realizace jednotlivých fází metodiky podrobněji popsána v dalších podkapitolách.

Proces automatizace bude tedy zahájen zvážením přínosů a využitelnosti vytvořených automatizovaných skriptů. Po definování přínosů, které jsou od automatizace očekávány, by měla být zvolena strategie a přístup pro vývoj automatizovaných skriptů. Velice důležité je také předem promyslet a určit jaká kombinace nástrojů pro vývoj bude zvolena. Vhodná kombinace vybraných nástrojů může velmi pozitivně ovlivnit celý proces vývoje, měli by se tedy definovat kritéria, která jsou od vybraného nástroje očekávána a vyžadována, aby v průběhu vývoje automatizovaných skriptů nedošlo ke zjištění, že používané nástroje nepodporují nějakou nutnou funkčnost pro další vývoj (např. podpora více druhů prohlížečů).

Následně je nutné přistoupit k analýze funkcionalit aplikace TestLink a pomocí principů prioritizace se rozhodnout, které funkcionality bude nejefektivnější automatizovat. K těmto vybraným funkcionalitám je pak vhodné následně vytvořit testovací případy pro manuální testování, podle kterých se budou vytvářet automatizované skripty. V této fázi je pak také důležité definovat, jak se bude při vytváření automatizovaných skriptů k TestLinku přistupovat. Po vyhodnocení těchto bodů již nic nebrání naplánování a navrhnutí architektury automatizovaných skriptů a následnému vývoji podle této architektury. Vytvořené skripty pak budou spouštěny a zaznamenají se jejich výsledky. Celý proces automatizace je poté nutné zhodnotit a promyslet případná vylepšení.

8. Návrh a realizace

8.1. Rozhodnutí o automatizaci testování

TestLink je aktuálně uživatelsky velmi oblíbený nástroj, u kterého lze předpokládat, že bude na trhu ještě dlouhou dobu a proto proběhne řada releasů, ve kterých by se daly automatizované skripty efektivně využít. Vytvoření automatizovaných skriptů by mohlo pomoci s udržováním, případně zvyšováním kvality aplikace a zároveň by tak mělo v dlouhodobém měřítku ušetřit část nákladů vynaložených na testování.

Při návrhu a vývoji bude tedy počítáno s dlouhodobým využíváním těchto skriptů a zároveň také s případným budoucím rozšiřováním. Z tohoto důvodu bude zvoleno procesně řízené skriptování, které umožňuje navrhnout skripty podle případů užití, tedy funkcionalit. Tyto skripty bude také možné parametrizovat pomocí datových souborů a bude tedy možné vytvářet mnoho instancí vytvořených skriptů s různými daty. Po zhodnocení složitosti aplikace TestLink, rozsahu automatizace a vzhledem k předpokladu dlouhodobého používání skriptů je zavedení Frameworku považováno za efektivní a právě procesně řízený přístup je pro implementaci Frameworku vhodný. Možnost zavedení Frameworku, nám umožní oddělit metody opakovaně používané v řadě skriptů a mimo jiné zjednoduší i údržbu.

Konkrétní funkcionality, které jsou vhodné pro automatizaci, budou vybrány na základě strategie “implementovat nejrizikovější funkce jako první”. Vybranými funkcionalitami by bylo vhodné pokrýt základní možnosti, které využívá převážná většina uživatelů.

8.2. Výběr nástrojů

K vytvoření automatizovaných skriptů pro aplikaci TestLink je nutné vybrat několik nástrojů. Musí být zvolen nástroj, který použijeme pro vytvoření automatizovaných testů, dále musí být vybrán programovací jazyk, ve kterém budou

automatizované testy vyvíjeny a samozřejmě také vývojové prostředí, které bude podporovat vybraný programovací jazyk. Důležité je také určit ve kterém prohlížeči budou testy spouštěny, protože nástroje podporující automatizaci nemusí podporovat vybraný prohlížeč.

8.2.1. Výběr nástroje pro automatizované testování

Bylo by vhodné, aby byl nástroj obecně známý a byl volně dostupný. Tyto kritéria splňují nástroje z kapitoly 5, bude tedy vybráno z těchto popsaných nástrojů. Pro objektivní výběr byla definována následující hodnotící kritéria související s tvorbou skriptů, podle kterých bude rozhodnuto, který z těchto nástrojů je nejvhodnější.

Hodnotící kritéria:

1. **Možnost testovat přímo GUI webové aplikace**
   - Cílem testování je testovat funkčnost aplikace z hlediska uživatele. Je tedy nutné vybrat nástroj, který umožňuje testovat přímo uživatelské rozhraní s tím, že není kladen tak velký důraz na výkonnost prováděných operací, ale na uživatelskou přívětivost a uživatelské rozhraní jako celek.
2. **Možnost testovat na více internetových prohlížečích**
   - Aplikace TestLink je univerzální aplikace, jejímž cílem je správné fungování pokud možno na všech moderních prohlížečích. Je tedy poměrně důležité umožnit provádění testů na několika internetových prohlížečích, případně na několika verzích internetových prohlížečů.
3. **Silná podpora JavaScript**
   - Vzhledem k velkému rozmachu JavaScriptu lze očekávat, že dříve nebo později bude čím dál tím více částí aplikace pracovat s JavaScriptem. Je tedy vhodné, aby testovací nástroj uměl dobře pracovat s JavaScriptem.
4. **Podpora dalších webových technologií pro GUI**

- Je vhodné vybrat nástroj, který bude umět spolupracovat nejen s HTML a JavaScriptem, ale i s jinými technologiemi. Například FLASH apod. Do budoucna nelze vyloučit rozvoj aplikace za použití takových technologií. V současnosti se ale v aplikaci nepoužívají.

5. **Podpora a rozvoj nástroje**
   - Z plánu na dlouhodobé využívání nástroje je rozumné zvolit takový nástroj, který je rozvíjen a jeho tvůrce poskytuje podporu pro nové technologie a standardy.
6. **Dostupnost informací**
   - Je vhodné vybrat nástroj, který je dobře a detailně dokumentován a kolem kterého se vytvořila široká a aktivní komunita. U takového nástroje lze předpokládat, že případné problémy již byly řešeny a lze na ně najít odpověď.

Výběr nástroje:

Kombinace hodnotících kritérií a jednotlivých nástrojů je znázorněna v následujícím obrázku. Pro každou kombinaci nástroj/hodnotící kritérium byla zvolena hodnota ANO nebo NE. Volba je založena na dostupných informacích a nezohledňuje více variant.

Nástroj	AutoIT	Http Unit	Jmeter	Selenium WebDriver	Selenium IDE	Sikuli	Canoo WebTest
Možnost testovat přímo GUI webové aplikace	Ano	Ano	Ne	Ano	Ano	Ano	Ano
Možnost testovat na více internetových prohlížečích	Ano	Ne	Ne	Ano	Ne	Ano	Ne
Podpora JavaScript	Ano	Ano	Ne	Ano	Ano	Ne	Ne
Podpora dalších webových technologií pro GUI	Ne	Ne	Ne	Ne	Ne	Ano	Ano
Podpora a rozvoj nástroje	Ano	Ne	Ano	Ano	Ano	Ano	Ano
Dostupnost informací	Ne	Ne	Ano	Ano	Ano	Ne	Ne
Počet kladných odpovědí	4	2	2	5	4	4	3

Obrázek 27 - Jednotlivé kombinace hodnotících kritérií a nástrojů. Vlastní tvorba.

Z obrázku je patrné, že nejvhodnějším nástrojem je testovací nástroj Selenium. Nástroj je určen k testování uživatelského rozhraní. Je navržen tak, aby byl snadno integrovatelný do různých vývojových prostředí a aby testovací skripty v tomto nástroji vytvořené bylo možné spustit na různých internetových prohlížečích. Vzhledem k faktu, že skripty běží přímo ve zvoleném internetovém prohlížeči, je podpora JavaScript plně dostačující. Nástroj je neustále vyvíjen a kolem nástroje v současnosti funguje velká a aktivní komunita. Vzhledem k počtu uživatelů tohoto nástroje a počtu zdrojů na internetu by neměl být problém s nedostatkem informací. Zároveň dokumentace je velice detailní a přehledná. Jediným nedostatkem nástroje je absence podpory technologií jako je Flash apod. Tyto technologie však nejsou využívány ve stávající aplikaci TestLink.

8.2.2. Výběr ostatních nástrojů

Zdrojový kód testů je ve vybraném testovacím nástroji možný definovat hned v několika programovacích jazycích. Vzhledem ke zkušenostem autorky s vyvíjením automatizovaných testů byl zvolen programovací jazyk Python 3.2 a vývojové prostředí PyCharm 4.0.6. Vyvinuté testy budou následně spouštěny v prohlížeči Firefox.

8.3. Zavedení procesu automatizovaného testování

Při automatizaci vybrané množiny funkcionalit je vhodné vytvořit několik testovacích sad, které budou sloužit v různých fázích testování a užívání aplikace. Podle toho k čemu budou jednotlivé sady sloužit, budou vybrány funkcionality, které budou testovány. Při definováním těchto testovacích sad, je nutné určit, co mají otestovat a kdy a v jakých časových intervalech je vhodné je využívat. Pro účely této práce byly vybrány tři základní sady testů a to monitorovací testy, smoke testy a regresní testy.

První sada bude určená k monitorování dostupnosti TestLinku. Bude tak možné zjistit, zda je možné se do TestLinku přihlásit a následně se z něj odhlásit.

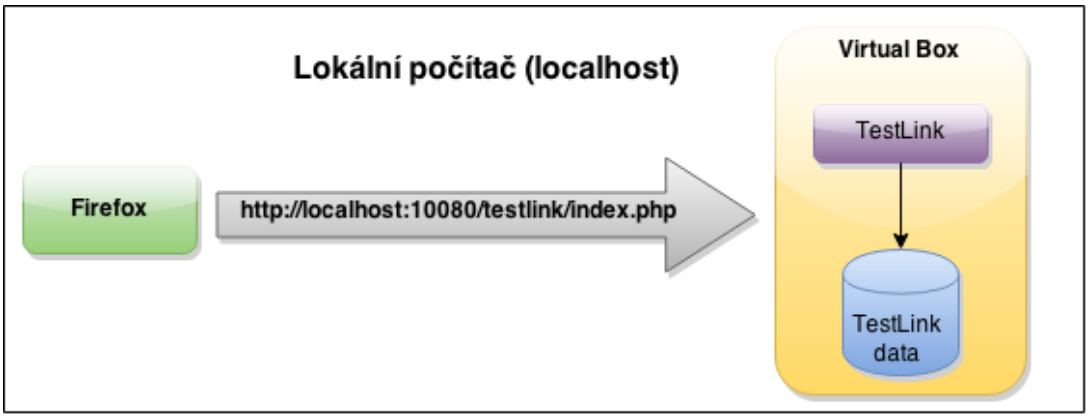
Jedná se tedy o neinvazivní typ testu, kdy neověřujeme možnost vkládání ani modifikaci dat. Testy pro ověření dostupnosti aplikace je vhodné spouštět v krátkých intervalech, např. každé 3 hodiny. Je tak zajištěna včasná reakce a následná oprava v případě nedostupnosti aplikace.

Další sadou budou smoke testy. Lze pomocí nich ověřit nejen dostupnost, ale i ověření toho, zda byla aplikace správně nasazena nebo také možnost vkládání a modifikace dat. Vložená data by ale po sobě měly testy rovnou "uklidit", nebylo by totiž vhodné, aby se aplikace plnila zkušebními daty, které nikdo nemaže. Spuštěním několika základních funkcionalit tak otestujeme, že je možné s aplikací aktivně pracovat. Takové testy je vhodné pouštět každé ráno, před zahájením projektových manuálních testů nebo před spuštěním sady regresních testů. Možnost aktivní práce s aplikací TestLink bude ověřena spuštěním testovacích skriptů pro funkcionality Přihlášení, Vytvoření projektu, Vytvoření Test Suitu, Vytvoření Test Casu, Smazání projektu a Odhlášení.

Třetí sadou pak bude sada regresních testů. Tato sada má pokrývat množinu testovacích případů, které pokrývají hlavní funkce aplikace. Po zanalyzování aplikace TestLink byly jako hlavní vybrány tyto funkcionality: Přihlášení, Vytvoření projektu, Změna projektu, Vytvoření uživatele, Vytvoření testovací sady, Vytvoření testovacího případu, Vytvoření kroků v testovacím případu, Vytvoření testovacího plánu, Vytvoření buildu, Přidání testovacího případu k buildu, Provedení testu, Vytvoření požadavku, Smazání testovacího plánu, Smazání projektu a Odhlášení. Vybrány byly proto, že pokrývají celý základní proces, který aplikace umožňuje a pomocí této sady testů lze ověřit, že nebyl základní proces narušen.

Nyní je již vybrána množina funkcionalit, která bude otestována, je ale ještě důležité definovat, jak se bude při vytváření automatizovaných testů k TestLinku přistupovat. K implementaci automatizovaných testů je možné zvolit jeden z několika možných přístupů k aplikaci TestLink. Jednou z možností je využití nainstalovaného nástroje TestLink na webových stránkách výrobce, kde se dá do aplikace přihlásit pomocí demo účtu a využívat tak jednotlivé funkčnosti. Druhou variantou, kterou výrobce umožňuje, je stáhnutí předinstalované verze TestLinku

na linuxové distribuci a její spuštění pomocí VirtualBoxu. Díky této variantě lze aplikaci spouštět v lokální síti anebo přímo na lokálním počítači. Pro účely této diplomové práce byla zvolena druhá možnost a TestLink byl instalován přímo na lokální počítač tvůrce diplomové práce. To umožnilo maximálně zefektivizovat tvorbu testů, jelikož není nutné stálé připojení k internetu. Při využití této konfigurace navíc zůstávají všechna vytvořená data na lokálním počítači.



Obrázek 28 - Graficky znázorněný použitý přístup k TestLinku. Vlastní tvorba.

8.4. Plánování, návrh a vývoj testování

Nyní je ještě nutné navrhnout architekturu automatizovaných testů, podle které budou automatizované testy vyvíjeny. Vzhledem k tomu, že je pro programování univerzálním jazykem angličtina, budou i jednotlivé komponenty této architektury pojmenovány anglicky.

Jelikož bylo zavedení Frameworku zhodnoceno za efektivní, lze při návrhu architektury počítat s možností oddělení naprogramovaného kódu a konfigurací pro jednotlivé testovací sady. Tester, který bude mít za úkol testovací sady vytvářet, editovat a nastavovat parametry, tedy nebude muset mít znalosti týkající se programování. Zároveň bude také výrazně zjednodušena práce testera, který se bude starat o údržbu vytvořených automatizovaných testů. Vymyšlená struktura automatizovaných testů by měla odpovídat obrázku níže.

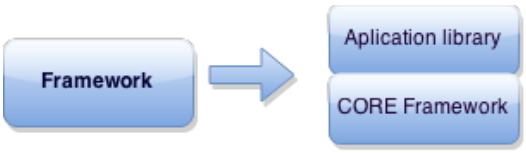
Obrázek 29 - Navržená struktura automatizovaných testů. Vlastní tvorba.

Nástroje jsou již určené a nyní je ještě potřeba detailněji navrhnout vrstvy Test Suites a Framework.

Do vrstvy Test Suites budou vkládány jednotlivé testovací sady, které bude potřeba v daném okamžiku otestovat. Může jich být libovolný počet. V jednotlivých testovacích sadách bude popsána konfigurace a parametry pro spuštění těchto sad, budou zde definovány funkcionality, které má daná sada otestovat, a budou zde parametry týkající se těchto funkcionalit. Testovací sady budou ideálně psány jednoduchou syntaxí a pro testera starajícího se o testovací sady, tak budou lépe čitelné. Díky tomu bude velmi jednoduché testovací sady měnit a upravovat a bude také snadné a rychlé vytváření sad nových.

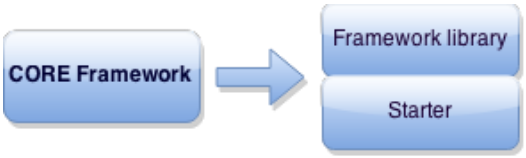
Obrázek 30 - Navržená struktura Test Suites. Vlastní tvorba.

Vrstva Framework zajišťuje to, že tester, starající se o testovací sady, tedy vrstvu Test suites, se opravdu stará pouze o konfiguraci testů v jednoduché syntaxi a nemusí postup kroků implementovat, nemusí tak zasahovat do kódu ani jej vytvářet. Může ale pracovat pouze s funkcionalitami a parametry, které jsou naprogramovány ve vytvořeném Frameworku. Samotný Framework bude ještě rozdělený na dvě části a to CORE Framework a Application library.



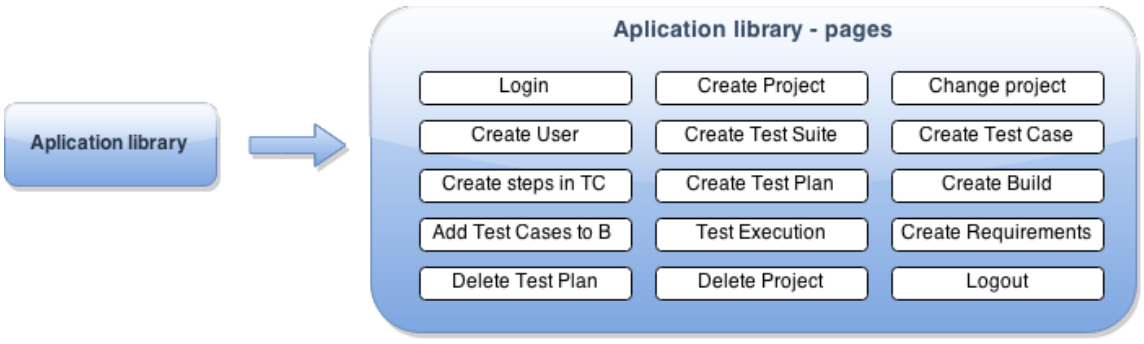
Obrázek 31 - Navržená struktura Frameworku. Vlastní tvorba.

Část CORE Framework zajišťuje správné spuštění testovacích sad a funguje jako samostatný element, který půjde bez zásahu do kódu použít i při testování jiných aplikací než TestLink. Tato část bude obsahovat spouštěcí soubor Starter a Framework library, jejichž funkce bude ještě vysvětlena.



Obrázek 32 - Navržená struktura CORE Frameworku. Vlastní tvorba.

Část Application library je nadstavbou CORE Frameworku a bude obsahovat kód, který bude popisovat kroky na jednotlivých obrazovkách potřebných k vykonání funkcionalit, které jsme si již definovali. V případě potřeby se do Application library mohou přidávat popisy dalších obrazovek a kroků, které na nich lze vykonat a rozšiřovat tak množinu funkcionalit, kterou bude možné vytvořenými automatizovanými testy testovat. Vzhledem k tomu, že pro každou obrazovku bude vytvořen pouze jeden modul v Application library, bude také jednodušší údržba testů. Pokud dojde ke změně nějaké obrazovky, tak tato změna ovlivní pouze jeden modul v Application library a díky tomu tak bude oprava rychlejší.



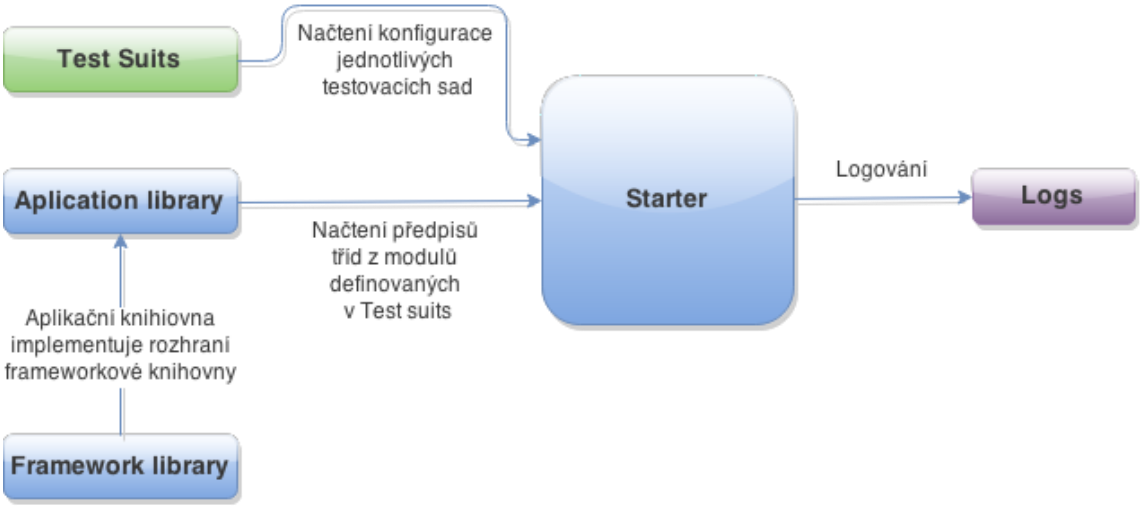
Obrázek 33 - Navržená struktura Application library. Vlastní tvorba.

Výsledkem plánování a návrhu je tedy detailně rozvrstvená struktura, která je jako celek zobrazena na následujícím obrázku. Před zahájením programování

automatizovaných testů podle této struktury, je ještě nutné definovat fungování a propojení jednotlivých komponent.

Obrázek 34 - Finální navržená architektura automatizovaných testů. Vlastní tvorba.

Automatizované testy budou inicializovány pomocí Starteru, který spustí testy podle konfigurace jednotlivých testovacích sad v TestSuites. Starter si na základě těchto konfigurací načte informaci o tom, jaké moduly z části Application library využije a načte si požadované předpisy tříd těchto modulů. K vytvoření jednotlivých instancí těchto tříd bude nutná konstrukční funkce, kterou musí obsahovat každá třída modulů Application library. Z principu dědění tříd bude tato konstrukční funkce definována pouze ve Framework library. Framework library bude tedy obsahovat Abstraktní třídu, ve které budou funkce opakující se ve více modulech a konstruktor společný pro všechny třídy testů z Application library. Po vytvoření jednotlivých instancí z Application library bude podle konfigurace testovacích sad známé, jaký postup testovacích kroků se má vykonat v jakém prohlížeči. Průběh spuštěných testů bude zaznamenáván pomocí logování. Budou vytvářeny logy pro každé spuštění jednotlivé testovací sady. To, které informace mají tyto logy obsahovat, bude definováno ve Starteru.

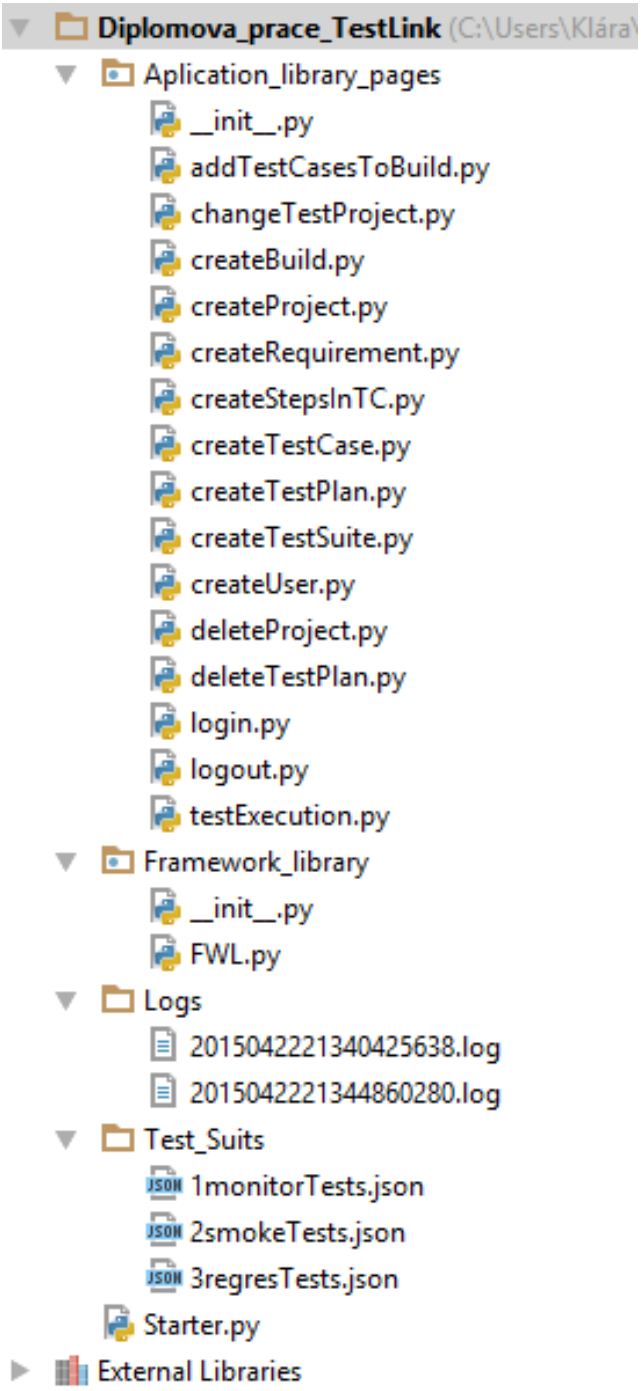


Obrázek 35 - Schéma architektury Frameworku. Vlastní tvorba.

8.5. Spuštění a řízení testů

Podle navržené architektury byl vytvořen projekt „*Diplomova_prace_TestLink*", kterého obsahem jsou všechny komponenty předem navržené architektury. Tento projekt je uložen v příloze diplomové práce na DVD. Zdrojový kód je podrobně dokumentován formou komentářů, a proto nebude detailně popisován v textu diplomové práce. Na následujících obrázcích jsou uvedeny ukázky jednotlivých komponent s jejich jednoduchým popisem.

Celkový pohled na adresářovou strukturu projektu a jeho jednotlivé částí je zobrazen na obrázku č. 36.



Obrázek 36 - Stromová struktura vytvořených automatizovaných testů v nástroji PyCharm. Vlastní tvorba.

Všechny soubory zdrojového kódu jsou psané v jazyku Python a mají proto souborovou příponu .py. Tyto soubory jsou podle svého významu v rámci projektu na 3 místech:

- Application_library_pages - zdrojové kódy aplikační knihovny

- Framewrok_library - abstraktní třída testu, ze které vychází aplikační knihovna
- Starter.py - inicializační skript pro spuštění automatizovaných testů

V obrázku č. 37 je znázorněna ukázka zdrojového kódu skriptu CreateProject.

```
__author__ = 'Klára'

from Framework_library.FWL import AbstractTestCase


class CreateProject(AbstractTestCase):
    def create_project(self):
        driver = self.webdriver
        params = self.params

        driver.get(self.base_url + params['first_path'])    #načtení úvodní stránky
        driver.switch_to_frame('mainframe')
        driver.find_element_by_link_text("Test Project Management").click( )    #kliknutí na odkaz Test Project Management
        driver.find_element_by_id("create").click()     #vytvoření nového projektu
        driver.find_element_by_name("tprojectName").clear()
        driver.find_element_by_name("tprojectName").send_keys(params['name_project'])   #pojmenování projektu
        driver.find_element_by_name("tcasePrefix").clear()
        driver.find_element_by_name("tcasePrefix").send_keys(params['name_prefix_for_test_case'])   #pojmenování prefixu
        driver.find_element_by_xpath("//td[@id='cke_contents_notes']/iframe").send_keys(params['description_test_project'])
        driver.find_element_by_name("optReq").click()   #zaškrtnutí checkboxu pro zobrazení requirements v projektu
        driver.find_element_by_name("doActionButton").click()   #uložení zakládaného projektu

        # Ověření, že se projekt opravdu vytvořil:
        expected_text = params['name_project']
        real_text = driver.find_element_by_link_text(params['name_project']).text

        if expected_text == real_text:
            print('createProject ok')
        else:
            print('createProject failed')
```

Obrázek 37 - Zdrojový kód Create project z Application library. Vlastní tvorba.

V adresářové struktuře projektu jsou dále konfigurační soubory jednotlivých Test Suites, podle kterých jsou spouštěné jednotlivé testy a postupné vykonání jejich kroků. Tyto konfigurační soubory jsou v běžně používaném formátu JSON[22], kterého přednost spočívá také v jednoduché čitelnosti pro uživatele. Tyto soubory mají příponu .json a jsou uloženy v adresáři Test_Suites. Obrázek č. 38 zobrazuje konfiguraci testovací sady Monitor tests.

[22] Formát .json - JavaScript Object Notation neboli JavaScriptový zápis objektů - formát pro výměnu dat

Obrázek 38 - Konfigurace testovací sady Monitor tests. Vlastní tvorba.

Pomocí této syntaxe lze vložit a nastavit libovolnou posloupnost testovacích skriptů z Application library. Podle příkladu na obrázku je zjevné, že formát JSON napomáhá jednoduché orientaci a samopopisovatelnosti nastavené konfigurace, čímž přispíva k rychlejší tvorbě testovacích skriptů.

Test Suites jsou podle navržené architektury spouštěné pomocí inicializačního skriptu - Starter. Ten lze spouštět přímo v nástroji PyCharm, nebo také z příkazového řádku. V případě využití nástroj PyCharm může být přínosem také debugger, který se s úspěchem osvědčil při vývoji Application library. Tato forma spouštění je zejména ve

spojení s laděním vhodná pro programátora jednotlivých testů. Naopak spouštění přímo z konzole neboli příkazové řádky, bude typická pro samotné vykonávání testů.

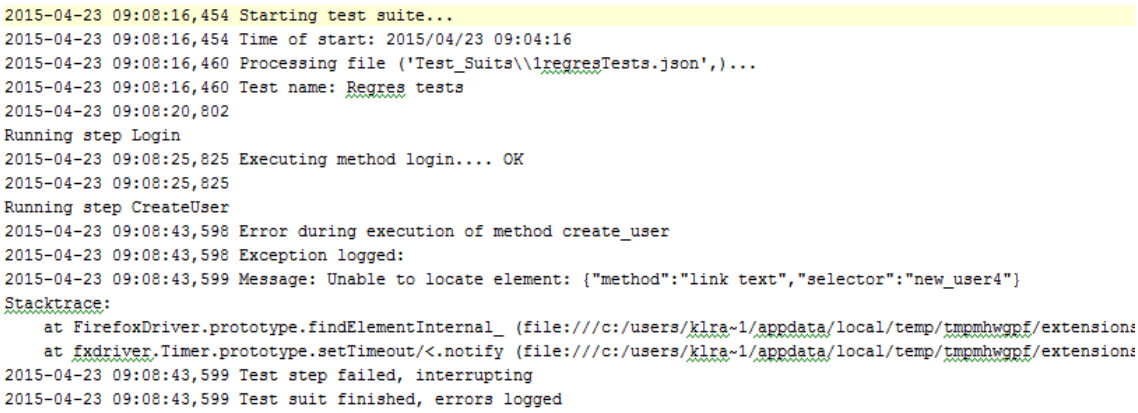
Spouštění testů je v této verzi frameworku manuální, je zde tedy případný prostor na vylepšení. V budoucnu by se testy mohly spouštět také automaticky na základě konceptu Continuous Integrations, který je popsán v následující kapitole.

Poslední částí v adresářové struktuře projektu jsou pak log soubory z jednotlivých spuštění testů. Logy jsou ukládány do textového souboru s příponou .txt a jsou vytvářeny při každém spuštění testů. Na následujícím obrázku č. 39 je zobrazen Log úspěšně spuštěné sady Smoke tests.

```
2015-04-23 08:51:12,122 Starting test suite...
2015-04-23 08:51:12,122 Time of start: 2015/04/23 08:04:12
2015-04-23 08:51:12,124 Processing file ('Test_Suits\\2smokeTests.json',)...
2015-04-23 08:51:12,124 Test name: Smoke tests
2015-04-23 08:51:17,385
Running step Login
2015-04-23 08:51:22,966 Executing method login.... OK
2015-04-23 08:51:22,967
Running step CreateProject
2015-04-23 08:51:30,816 Executing method create_project.... OK
2015-04-23 08:51:30,816
Running step ChangeTestProject
2015-04-23 08:51:32,611 Executing method change_test_project.... OK
2015-04-23 08:51:32,611
Running step CreateTestSuite
2015-04-23 08:51:39,881 Executing method create_test_suite.... OK
2015-04-23 08:51:39,881
Running step CreateTestCase
2015-04-23 08:51:48,848 Executing method create_test_case.... OK
2015-04-23 08:51:48,848
Running step DeleteProject
2015-04-23 08:51:54,599 Executing method delete_project.... OK
2015-04-23 08:51:54,599
Running step Logout
2015-04-23 08:51:56,769 Executing method logout.... OK
2015-04-23 08:51:56,769 Step Logout finished succesfully
2015-04-23 08:51:56,769 Test suit finished, no errors found
```

Obrázek 39 - Log úspěšně spuštěné sady Smoke tests. Vlastní tvorba.

Na obrázku č. 40 je znázorněn Log po spuštění sady Resgres Tests, které skončilo s chybou. Toto spuštění neproběhlo správně, jelikož se skript pokusil uložit nového uživatele, který již existuje.



Obrázek 40 - Log po spuštění sady Resgres Tests, které skončilo s chybou. Vlastní tvorba.

Způsob implementace, architektura a framework byly zvolené adekvátně ke specifickým potřebám testování TestLinku. Nicméně stále je zde prostor pro další rozšiřování. Ať už cestou zapojení Continuous Integration, nebo vytvořením silné platformy pro automatizované testy, která bude programátorům testů poskytovat širší podporu při psaní testů.

9. Trendy a vizionářství

V současné době není automatizace testů používána v takovém rozsahu, v jakém by mohla být. V některých společnostech je i dnes význam psaní automatizovaných testů opomíjen a testování je delegováno až na osobu testera. Tester pak nachází i elementární chyby ve zdrojovém kódu. Proces reportování takových chyb pak pouze prodlužuje dobu vývoje a prodražuje celý vývoj. Stejná situace je například i u testů uživatelského rozhraní. Důvodem, proč není vyvíjen dostatečný tlak na automatizaci těchto testů, často bývá obava z nárůstu nákladů na údržbu testů.

Situace kolem automatizace testů se ale postupně zlepšuje. Důkazem může být vznik konceptu „Continuous Integration“ a s tím souvisejících nástrojů (např. Continuous Integration server Jenkins). Základní myšlenkou je automatické spouštění automatických testů tak, aby byl minimalizován čas, který musí vývojový tým strávit testováním. Proces vývoje pak může vypadat například takto:

1. Programátor implementuje novou funkci a zdrojový kód (včetně unit testů) uloží do verzovacího nástroje.
2. Continuous Integration server je notifikován o nové verzi zdrojového kódu a provede:
    - Pokus o kompilaci zdrojového kódu (zdrojový kód může být nekompilovatelný z důvodu změny od jiného programátora).
    - Spuštění všech Unit testů (systém validuje, že nebyly, třeba nedopatřením, změněny předpoklady, na kterých je aplikace postavena).
    - Vyhodnocení pokrytí zdrojového kódu unit testy.
    - Statickou analýzu zdrojového kódu (systém se snaží podle předdefinovaných pravidel odhalit potencionální chyby ve zdrojovém kódu).
3. Pokud vše proběhne správně, nasadí Continuous Integration server aplikaci do testovacího prostředí a provede:

- Spuštění integračních testů pro správnou funkčnost z hlediska ukládání a čtení dat do a z databáze.
- Spustí testy pro uživatelské rozhraní.

4. Pokud vše proběhne správně, nasadí Continuous Integration server aplikaci do předprodukčního prostředí a provede spuštění integračních testů pro ověření správné funkčnosti z hlediska komunikace s okolními systémy.
5. Systém je připraven pro regresní testování testerem.

Výhody takového systému jsou:

- Testy jsou spuštěny po každé změně zdrojového kódu a případná chyba je tedy odhalena v řádu minut.
- Spuštění testů spotřebovává čas serveru a ne členů vývojového týmu.
- Je vyvíjen tlak na vývojový tým na zlepšování práce. Zvyšování procenta pokrytí unit testy, snižování počtu chyb ve způsobu psaní zdrojového kódu na základě statické analýzy kódu.
- Aplikace je rychle otestována v izolovaném prostředí a integrace s okolními systémy je testována až následně.
- Tester se zapojuje do vývoje až v okamžiku, kdy již jednou definované testy ověřili správnost aplikace a tím šetří čas.
- Tester má k dispozici vždy poslední funkční verzi aplikace.

Takto nastavený projekt účinně eliminuje nevýrobní činnost ve vývojovém týmu a významně zefektivňuje činnost testerů.

Ještě účinnější je koncept „Continuous delivery“. Jde vlastně o rozšíření konceptu Continuous Integration s tím, že proces je doveden až do konce vývojového cyklu. V rámci takového projektu je využívána metodika pro „programování řízené testy. Tester nejprve nadefinuje test, který samozřejmě není validní. Následně programátor implementuje danou funkci a odešle zdrojový kód do verzovacího nástroje. V tuto chvíli jsou již všechny testy, včetně testů nové funkce, testerem nadefinovány. V případě, že všechny testy aplikace jsou systémem vyhodnoceny jako validní, není důvod, proč takovou změnu nenasadit rovnou do

prostředí pro akceptační testy. Koncept Continuous delivery ale vyžaduje velkou míru pokrytí testy a to na všech úrovních testů. [7][24]

10. Závěr

Cílem diplomové práce bylo v teoretické části analyzovat metody a přínosy automatizovaného testování.

V úvodu teoretické části jsem se věnovala základním principům testování softwarových systémů a postupně definovala automatizaci testů, důvody jeho zavedení a očekávané přínosy. Byla popsána metodika automatizace a jednotlivé přístupy k tvorbě skriptů. Inspirací z osvědčených postupů vývoje software je tvorba frameworku, která, jak se ukazuje, může být přínosem i při automatizaci testů.

Praktická část práce si kladla za cíl vybrat konkrétní metody a nástroje pro automatizaci testů a na aplikaci TestLink navrhnout a vyvinout sadu automatizovaných skriptů. Následně pak ověřit vhodnost a realizovatelnost automatického testování webových aplikací za použití běžně dostupných nástrojů.

Pro automatizaci byl vybrán procesně řízený přístup, který je vhodný i pro rozsáhlé systémy s častými release a změnami. Z dostupných nástrojů na trhu byl použitý open-source nástroj Selenium Webdriver, který umožňuje definici testu přímo v programovacím jazyku, což dává testerům značnou volnost v dalším rozvoji definovaných testů. Jako programovací jazyk pro tvorbu skriptů byl kvůli rychlosti vývoje a autorčiným znalostem jazyka vybrán Python. V rámci této části byla také nainstalována aplikace TestLink. Před vývojem samotných testovacích sad (Test Suite) jsem také investovala úsilí do vytvoření frameworku, který pak při tvorbě finálních skriptů zrychlil a zpřehlednil jejich tvorbu. Výsledný log spuštěných testů a také zdrojové kódy jednotlivých skriptů a frameworku jsou přiloženy jako příloha diplomové práce.

Zvolený přístup a nástroje v kombinaci s frameworkem umožnily rychle a efektivně vyvinout několik testovacích sad a tvoří také dobrý základ pro dlouhodobou údržbu skriptů. Vznikla základní verze frameworku v jazyce Python, který se dá dále rozvíjet, čímž byl potvrzen předpoklad vhodnosti zvolených

postupů. Další rozvoj zkoumané oblasti automatizace testů autorka shrnula v kapitole „Trendy a vizionářství“. Tímto považuji zadání diplomové práce za splněné.

Zvolený přístup nemusí být přímo aplikovatelný pro všechny webové aplikace. Konkrétní přístup je vždy závislý od rozsahu aplikace, četnosti a charakteru změn v aplikaci, zkušeností vývojového a testovacího týmu, ochoty investic do automatizace atd. Na základě zpracovaného tématu je ale možné tvrdit, že nad automatizací testů se vždy vyplatí alespoň zamyslet a vykonat úvodní analýzu realizovatelnosti a přínosů. Tato práce má snahu k tomu poskytnout solidní teoretický základ i praktické doporučení k následné realizaci.

Kromě pomoci týmům doposud „nedotknutých“ automatizací testů a poskytnutí uceleného obrazu od teorie až po samotný vývoj a spouštění testů, mi tato práce pomohla do větší hloubky prozkoumat zajímavou oblast softwarového průmyslu a získané zkušenosti pro mne budou velmi užitečné v dalším odborném růstu.

11. Seznam obrázků

12. Použitá literatura

[1] Advantages of Automation. Software Testing Mentor. [online]. [cit. 2015-02-23]. Dostupné z: http://www.softwaretestingmentor.com/automation/advantages-of-automation/

[2]Apache JMeter™. The Apache Software Foundation. [online]. 19.4.2015 [cit. 2015-04-19]. Dostupné z: http://jmeter.apache.org/

[3] AutoIt. Wikipedia. [online]. 19.4.2015 [cit. 2015-04-19]. Dostupné z: http://en.wikipedia.org/wiki/AutoIt

[4] Automatizované testování. *Testování softwaru.* [online]. [cit. 2015-02-16]. Dostupné z: *http://testovanisoftwaru.cz/automatizovane-testovani/*

[5] BAKKER, Bryan, Graham BATH, Armin BORN, Mark FEWSTER, Jani HAUKINEN, Judy MCKAY, Andrew POLLNER, Raluca POPESCU a Ina SCHIEFERDECKER. Expert Level Syllabus Test Automation – Engineering: International Software Testing Qualifications Board. [online]. 2014, s. 82 [cit. 2015-04-18]. Dostupné z: http://www.istqb.org/downloads/finish/18/146.html

[6] Behavior Driven Development. Welcome to behave!. [online]. [cit. 2015-04-18]. Dostupné z: https://pythonhosted.org/behave/philosophy.html

[7] BERG, Alan. Jenkins continuous integration cookbook: over 80 recipes to maintain, secure, communicate, test, build, and improve the software development process with Jenkins. Birmingham: Packt Pub., 2012, iv, 328 p. ISBN 978-1-849517-40-9

[8] Business case. Business Dictionary. [online]. [cit. 2015-03-14]. Dostupné z: http://www.businessdictionary.com/definition/business-case.html

[9] Business case. Project Management Docs. [online]. [cit. 2015-03-14]. Dostupné z: http://www.projectmanagementdocs.com/project-initiation-templates/business-case.html

[10] Downloads. SeleniumHQ. [online]. [cit. 2015-04-19]. Dostupné z: http://www.seleniumhq.org/download/

[11] DUSTIN, Elfriede, Jeff RASHKA a John PAUL. *Automated software testing: introduction, management, and performance*. Reading, Mass.: Addison-Wesley, 1999, xxi, 575 p. ISBN 0201432870.

[12] DUSTIN, Elfriede, Thom GARRETT a Bernie GAUF. *Implementing automated software testing: how to save time and lower costs while raising quality*. Upper Saddle River, NJ: Addison-Wesley, c2009, xxv, 340 p. ISBN 9780321580511.

[13] *Guide to the software engineering body of knowledge: 2004 version*. Los Alamitos, CA: IEEE Computer Society Press, 2005, p. cm. ISBN 0769523307.

[14] HAILPERN, B., P. SANTHANAM a Cyrille ARTHO. Software debugging, testing, and verification. *IBM Systems Journal* [online]. 2002, vol. 41, issue 1, s. 99-113 [cit. 2015-02-16]. DOI: 10.1007/978-3-642-01702-5_13.

[15] Home. Canoo WebTest. [online].[cit. 2015-04-19]. Dostupné z: http://webtest.canoo.com/webtest/manual/WebTestHome.html

[16] Home. Sikuli Script. [online]. [cit. 2015-04-19]. Dostupné z: http://www.sikuli.org/

[17] HttpUnit. Sourceforge. [online]. [cit. 2015-04-19]. Dostupné z: http://httpunit.sourceforge.net/

[18] J.Prabhu, G. Gunasekaran " A Model for GUI Automated Testing Framework in Software System ", International Journal of Computer Applications, Vol. 64 No. 15 (2013). Dostupné z: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.278.3988&rep=rep1&type=pdf

[19] K. MALAIYA, "Automatic Test Software", Computer Science Department, Colorado State University. Dostupné z: http://www.cs.colostate.edu/~malaiya/tools2.pdf

[20] Metric. The Free Dictionary. [online]. [cit. 2015-04-18]. Dostupné z: http://thefreedictionary.com/metric

[21] P. JURKA, *Analýza a návrh změn informačního systému firmy,* Brno 2008. 65 s. Diplomová práce. Vysoké učení technické v Brně Fakulta podnikatelská Ústav

managementu. Dostupné z:
http://www.vutbr.cz/www_base/zav_prace_soubor_verejne.php?file_id=3894

[22] QUADRI, S.M.K a Sheikh Umar FAROOQ. Software Testing – Goals, Principles, and Limitations. *International Journal of Computer Applications*[online]. 2010, vol. 6, issue 9, s. 189-233 [cit. 2015-02-16]. DOI: 10.1007/0-387-21658-8_7. Dostupné z: http://www.ijcaonline.org/volume6/number9/pxc3871448.pdf
[23] ROUDENSKÝ, Petr a Anna HAVLÍČKOVÁ. *Řízení kvality softwaru: průvodce testováním*. 1. vyd. Brno: Computer Press, 2013, 208 s. ISBN 978-80-251-3816-8.

[24] RUBINGER, Andrew Lee a Aslak KNUTSEN. Continuous enterprise development in Java. 1st ed. xiii, 239 pages. ISBN 1449328296.

[25] SONMEZ, John. How To Create An Test Automation Framework Architecture With Selenium. Øredev Conference, 2014. Dostupné z:
https://vimeo.com/111847786

[26] Test Automation ROI Calculator. Sourceforge. [online]. [cit. 2015-04-18]. Dostupné z: http://ta-roi.sourceforge.net/

[27] Test Case – Testovací případ. *Testování softwaru.* [online]. [cit. 2015-02-28]. Dostupné z: *http://testovanisoftwaru.cz/dokumentace-v-testovani/test-case/*

[28] TEST CASE Fundamentals. *Software Testing Fundamentals.* [online]. [cit. 2015-02-28]. Dostupné z: *http://softwaretestingfundamentals.com/test-case/*

[29] Test Driven Development in PHP. *DZone.* [online].[cit. 2015-04-17]. Dostupné z: *http://css.dzone.com/news/test-driven-development-php*

[30] TestLink. Wikipedie. [online]. [cit. 2015-03-28]. Dostupné z:
http://cs.wikipedia.org/wiki/TestLink

[31] Testovací dokumentace – plán, scénář, případ. *SW Testování.* [online]. [cit. 2015-02-28]. Dostupné
z: *http://www.swtestovani.cz/index.php?option=com_content&view=article&id=15:testovaci-dokumentace-plan-scena-pipad&catid=3:zaklady&Itemid=11*

[32] Testování Softwaru. *Smoke testy.* [online]. [cit. 2015-04-17]. Dostupné z: *http://testovanisoftwaru.cz/tag/smoke-testy/*

[33] WILLIAMS, Laurie. Testing Overview and Black-Box Testing Techniques [online]. 2006 [cit. 2015-02-16]. Dostupné z: http://agile.csc.ncsu.edu/SEMaterials/BlackBox.pdf

13. Slovník pojmů

Pojem	Definice
API	Rozhraní pro programování aplikací
AST Tool	Nástroj pro automatizované testování softwaru
Behavior driven development	Programování řízené chováním
Capture and playback	Zachyť a přehraj
Continuous integration	Průběžná integrace
Data driven testing	Datově řízené programování
Defekt	Chyba v testované aplikaci
Emulace	Napodobení činnosti jednoho zařízení pomocí jiného zařízení
Freeware	Volně dostupné softwary
GUI	Grafické uživatelské rozhraní
Keyword driven testing	Testování řízené klíčovými slovy
Linear scripting	Lineární skriptování
Model based testing	Testování na bázi modelu
Open-source software	Počítačový software s otevřeným zdrojovým kódem
Page flow	Souslednost obrazovek ve webové aplikaci
Process driven testing	Procesně řízené testování
Release	Verze vydání systému
ROI	Metrika pro návratnost investic
Structure scripting	Strukturované skriptování
Test Case	Testovací případ
Test driven development	Programování řízené testy
Test Suite	Testovací sada
Ubiquitous language	Společný, ale jednoznačný jazyk používaný mezi vývojáři a uživateli při popisech funkčností
Use Case	Případ užití
Workflows	Definovaná sekvence kroků v pracovní operaci

14. Přílohy

Příloha 1:

K diplomové práci je přiloženo datové médium DVD s následujícím obsahem:

- TestLink.xmind – Mind mapa aplikace TestLink
- Diplomova_prace_TestLink.rar – Vše vytvořené týkající se automatizace testů pro aplikaci TestLink (Application_library_pages, Framework_library, Logs, Test_Suites, Starter)
- Config.txt – Návod na instalaci nástrojů potřebných ke spuštění automatizovaných testů pro aplikaci TestLink
- Diplomova_prace_Klara_Smatanova.pdf – Elektronická verze diplomové práce

15. Zadání práce

Univerzita Hradec Králové
Fakulta informatiky a managementu
Akademický rok: 2014/2015

Studijní program: Systémové inženýrství a informatika
Forma: Prezenční
Obor/komb.: Informační management (im5-p)

Podklad pro zadání DIPLOMOVÉ práce studenta

PŘEDKLÁDÁ:	ADRESA	OSOBNÍ ČÍSLO
Smatanová Klára	Brněnská 2681, Česká Lípa	I1001193

TÉMA ČESKY:

Automatizované testování webových aplikací

NÁZEV ANGLICKY:

Automated testing of web applications

VEDOUCÍ PRÁCE:

Ing. Karel Mls, Ph.D. - KIT

ZÁSADY PRO VYPRACOVÁNÍ:

Cíl práce:
Analýza metod a přínosů automatizovaného testování a praktické ověření na softwarovém nástroji TestLink.

Osnova:
1. Úvod
2. Základní principy testování softwarových systémů
3. Automatizované testování softwarových systémů
4. Framework
5. TestLink
6. Test analýza
7. Návrh a realizace
8. Závěr

SEZNAM DOPORUČENÉ LITERATURY:

bude upřesněno

Podpis studenta: Smatanová ..... **Datum:** 20.4.2015

Podpis vedoucího práce: ..... **Datum:** 20.4.2015

(c) IS/STAG , Portál - Podklad kvalifikační práce , I1001193 , 20.04.2015 14:46