

VYSOKÁ ŠKOLA EKONOMICKÁ V PRAZE

Fakulta informatiky a statistiky

Katedra informačních technologií

Studijní program: Aplikovaná informatika

Obor: Informační systémy a technologie

**Testování webových aplikací
s využitím nástroje Selenium WebDriver**

DIPLOMOVÁ PRÁCE

Student: Bc. Lucie Třísková

Vedoucí: doc. Ing. Alena Buchalceková, Ph.D.

Oponent: Ing. et Ing. Michal Doležel

2015

Prohlášení

Prohlašuji, že jsem diplomovou práci zpracovala samostatně a že jsem uvedla všechny použité prameny a literaturu, ze kterých jsem čerpala.

V Praze dne 27. 4. 2015

.....

Bc. Lucie Třísková

Poděkování

Na tomto místě bych chtěla poděkovat paní doc. Ing. Aleně Buchalcekové, Ph.D., za cenné připomínky, rady a ochotu při vedení mé diplomové práce.

Abstrakt

Diplomová práce se zabývá problematikou automatizace testování webových aplikací. První část popisuje automatizované testování a porovnává jej s testováním manuálním, kategorizuje dostupné testovací nástroje a představuje testovací nástroje nazvané Selenium. V další části je pak detailně popsána metodika automatizace testování webových aplikací, která poskytuje systematický návod, jak při automatizaci testování postupovat, jakým chybám se vyvarovat, a také úkoly a odpovědnosti jednotlivých rolí, které se procesu účastní. Součástí přílohy diplomové práce je pak příručka k nástroji Selenium WebDriver, který je jedním z nejpoužívanějších nástrojů pro funkcionální testování webových aplikací. Příručka provádí čtenáře od základů práce s nástrojem až po pokročilé možnosti využití a poskytuje všechny potřebné informace pro vytváření automatizovaných testů. Tato práce tedy představuje komplexní materiál, na jehož základě lze realizovat proces automatizace testování s využitím konkrétního testovacího nástroje.

Klíčová slova

Automatizace testování webových aplikací, metodika pro automatizaci testování, nástroje pro automatizaci testování, Selenium 2, Selenium WebDriver, tester, testovací nástroje, testování softwaru.

Abstract

This diploma thesis deals with automated testing of web applications. First of all, it introduces automated testing and compares it with manual testing, then classifies currently available testing tools and describes Selenium testing tools. In further chapter a methodology for automation of testing of web applications is described, providing a systematic manual, how to automate testing, which mistakes to avoid, and also tasks and responsibilities of involved roles. Appendix of this thesis consists of a user manual for Selenium WebDriver, one of the most popular tools for automated functional testing of web application. This manual guides reader from the basics of using Selenium WebDriver tool to its advanced features and provides complete information set needed for writing automated tests. This thesis represents a complex material that can be used for carrying out a process of test automation using a particular testing tool.

Keywords

Automated testing of web applications, methodology for test automation, Selenium 2, Selenium WebDriver, software testing, test automation tools, tester, testing tools.

Obsah

1	Úvod	1
1.1	Zaměření práce a důvod výběru tématu	2
1.2	Cíl práce	3
1.3	Cílová skupina	3
1.4	Předpoklady a omezení	4
1.5	Struktura práce	5
1.6	Očekávaný vlastní přínos	5
1.7	Vymezení základních pojmů	6
2	Rešerše zdrojů na téma automatizace testování softwaru	14
2.1	Odborné knihy	14
2.2	Akademické práce	17
2.3	Internetové zdroje	19
3	Automatizované testování softwaru	21
3.1	Charakteristika automatizovaného testování	21
3.2	Nástroje pro automatizované testování webových aplikací	24
3.3	Nástroje rodiny Selenium	30
3.3.1	Historie	30
3.3.2	Selenium Remote Control	33
3.3.3	Selenium IDE	34
3.3.4	Selenium Grid	36
3.3.5	Selenium WebDriver	36
3.4	Nástroje pro zkoumání struktury webové stránky	37
3.4.1	Firebug	38
3.4.2	Firefinder	42
3.4.3	Internet Explorer Developer Tools	43
3.4.4	Chrome Developer Tools	44
4	Metodika pro automatizaci testování webových aplikací	46
4.1	Proces automatizace testování	53
4.1.1	Rozhodování o automatizaci testování	54
4.1.2	Výběr testovacího nástroje	58
4.1.3	Příprava automatizace testování	61
4.1.4	Plánování, návrh a vytváření testů	65
4.1.5	Řízení a provádění testů	74
4.1.6	Kontrola a vyhodnocování procesu testování	77
4.2	Vztah automatizace testování a životního cyklu softwaru	79
4.3	Role	80
4.3.1	Projektový manažer	81
4.3.2	Test manažer	82

4.3.3	Tester	83
4.3.4	Tester pro automatizaci	83
4.3.5	Vývojář	84
4.3.6	Zákazník	85
4.4	Zavedení metodiky	86
4.5	Shrnutí	87
5	Závěr	88
	Terminologický slovník	89
	Seznam použité literatury	91
	Seznam obrázků a tabulek	101
	Seznam obrázků	101
	Seznam tabulek	102
	Rejstřík	103
	Příloha A: Uživatelská příručka k nástroji Selenium WebDriver	104
A.1	Jak začít	104
A.1.1	Spouštění testů v prohlížeči Internet Explorer	109
A.2	WebDriver API	111
A.3	Vyhledání elementů na webové stránce	113
A.3.1	Vyhledávání elementů pomocí atributu id	114
A.3.2	Vyhledávání elementů pomocí atributu name	115
A.3.3	Vyhledávání elementů podle atributu class	115
A.3.4	Vyhledávání elementů podle značky elementu	116
A.3.5	Vyhledávání elementů pomocí textu odkazu	116
A.3.6	Vyhledávání elementů pomocí CSS	117
A.3.7	Vyhledávání elementů pomocí XPath	121
A.4	Práce s elementy	126
A.4.1	Kliknutí na element	126
A.4.2	Dvojklik	126
A.4.3	Vyplňování formulářových polí	127
A.4.4	Získávání textové hodnoty elementu	128
A.4.5	Získávání hodnoty atributu elementu	128
A.4.6	Kontrola stavu elementu	129
A.4.7	Práce s rozevíracími a výběrovými seznamy	129
A.4.8	Práce se zaškrťovacími políčky a přepínači	131
A.4.9	Provádění úkonu táhni-a-puť	131
A.5	Řízení průběhu testu	132
A.5.1	Implicit wait	133
A.5.2	Explicit wait	134
A.5.3	Fluent wait	138

A.6	Pokročilé možnosti.....	139
	5.1.2 Ovládání aplikace pomocí klávesnice.....	140
	A.6.1 Práce s okny prohlížeče	141
	A.6.2 Provádění JavaScript kódu	145
A.7	Záznam průběhu testů	146
	A.7.1 Snímek obrazovky	146
	A.7.2 Videozáznam.....	147
A.8	Architektura automatizovaných testů	150
A.9	Automatizované spouštění testů	153

1 Úvod

Stále více podniků a institucí se dnes již neobejde bez využívání informačních technologií a softwarových aplikací, ať už pro samotné fungování svého byznysu, či alespoň pro účely managementu a administrativy. S touto měrou závislosti zároveň roste potřeba správného fungování těchto aplikací, neboť jejich funkčnost, anebo naopak nefunkčnost, může významně ovlivnit konkurenceschopnost firmy. Totéž však platí i v soukromém životě – jen málokdo už si dnes dokáže představit život bez internetu a webových aplikací, které využívá pro komunikaci, nákupy, vzdělávání a zábavu. Softwarové aplikace tedy ovlivňují život současného člověka v mnoha směrech, od roviny osobní až po pracovní. Z toho důvodu je žádoucí, aby výrobci softwaru produkovali kvalitní aplikace, které naplňují potřeby svých uživatelů.

Na kvalitu softwarové aplikace má přitom vliv celá řada faktorů – kvalifikovanost vývojářů a dalších osob, které se na vývoji softwaru podílí, zavedené standardy kvality, dále řízení, monitorování a vyhodnocování procesu vývoje softwaru a nakonec samozřejmě také testování softwaru, které je ústředním tématem této diplomové práce. [1 s. 271] Testování přitom hraje mezi výše zmíněnými faktory velice významnou roli, neboť i velice zkušený vývojář může udělat chybu, standardy nezvládnou pokrýt zcela všechny aspekty kvality a ani nejlépe nastavený proces vývoje softwaru nedokáže bez prováděného testování aplikace zajistit, aby výsledný produkt neobsahoval žádné závažné chyby. Proto je potřeba do procesu vývoje softwarových aplikací zapojit i testování, které v kombinaci s výše zmíněnými faktory (a za předpokladu, že je prováděno správně) dokáže zajistit, aby vytvořená aplikace odpovídala požadavkům zákazníka a uspokojovala jeho potřeby.

I v dnešní době je testování softwaru často vnímáno jako činnost prováděná až v závěru projektu, tedy těsně před představením produktu zákazníkovi, což je však odborníky považováno za špatný přístup. Náklady na opravu chyb totiž rostou exponenciálně v čase; chyba nalezená v úvodní fázi projektu je odstraněna s výrazně nižšími náklady než tatáž chyba odhalená v závěru projektu, kdy už je software v podstatě dokončen a je v něm potřeba provádět rozsáhlé změny. [1 s. 17] Z toho důvodu je žádoucí, aby bylo testování zapojeno do procesu vývoje softwaru v co nejranější fázi projektu, aby mohly být chyby odhaleny co nejdříve. Tento přístup je zohledněn v rámci metodiky pro automatizaci testování, která je součástí této diplomové práce.

Testování aplikací může probíhat buď manuálně, nebo automatizovaně. Oba dva způsoby mají své výhody i nevýhody – zatímco manuální testování, které provádí testeři, může odhalit množství chyb různého charakteru (zaměřuje se tedy na kvalitativní hledisko), automatizované testování umožňuje aplikaci podrobit definované sadě testů opakovaně, nad větším množstvím dat a v kratším čase (čímž představuje kvantitativní hledisko). Oba typy se tedy zaměřují na odlišné aspekty testování a přináší jiné výhody. Z toho důvodu je příhodné kombinovat manuální testování s testy automatizovanými, které využívají nástrojů pro automatizované

testování. Jedním z těchto nástrojů je Selenium WebDriver, který je v současné době jedním z nejpopulárnějších nástrojů pro testování uživatelského rozhraní webových aplikací (v roce 2013 jej Gartner označil jako nástroj s velice rychle rostoucí popularitou, který se již brzy stane standardem v oblasti funkcionálního testování [2]). Podrobný návod k použití tohoto nástroje je součástí přílohy této diplomové práce.

Jak již bylo zdůrazněno výše, testování je důležitou součástí procesu vývoje softwaru. Vývoj softwaru přitom může probíhat podle některé z metodik budování informačních systémů, které poskytují systematický návod, jak při vývoji aplikací postupovat, a tím napomáhají tento proces zefektivnit a především dovést k úspěšnému konci. Obdobný přístup by se dal aplikovat také na automatizaci testování, neboť ani vytváření automatizovaných testů není triviálním úkolem a i tato oblast by si zasloužila svou metodiku. Ta však doposud nebyla v českém jazyce k dispozici, a proto jsem se rozhodla vytvořit vlastní metodiku pro automatizované testování webových aplikací a v této práci ji poskytnout široké veřejnosti.

1.1 Zaměření práce a důvod výběru tématu

Diplomová práce je zaměřena na automatizované funkcionální testování webových aplikací s využitím nástroje Selenium WebDriver a programovacího jazyka Java. Toto téma jsem zvolila, neboť již několik let pracuji při studiu na pozici testera a součástí mé práce je mimo jiné i vytváření a správa automatizovaných testů. S nástrojem Selenium WebDriver jsem se poprvé setkala v září 2013, kdy mi byl přidělen úkol vybrat vhodný nástroj a zavést automatizované testování webových aplikací vyvíjených v rámci daného projektu. Po provedení průzkumu trhu jsem zvolila Selenium WebDriver, neboť jde o nástroj v komunitě testerů velice populární, lze k němu na internetu nalézt celou řadu návodů a rad a navíc je možné jej využívat zcela zdarma, neboť jde o open-source software.

Selenium WebDriver je tvořen souborem knihoven, které jsou k dispozici pro celkem sedm programovacích či skriptovacích jazyků, mimo jiné Java, C#, PHP aj. Pro účely vytvoření příručky, která je výstupem této práce, jsem zvolila jazyk Java. A to nejen proto, že jde zřejmě o nejrozšířenější programovací jazyk využívaný pro vytváření automatizovaných testů v tomto nástroji (soudě podle množství návodů a diskusních vláken na internetu), ale také z toho důvodu, že je příručka určena především pro studenty Vysoké školy ekonomické v Praze zapojené do aktivit Kompetenčního centra SQA, kteří mají díky povinným kurzům programování v Javě k tomuto jazyku nejblíže.

Znalost samotného nástroje však není dostačující podmínkou k tomu, aby bylo automatizované testování provedeno správně. Proto je diplomová práce doplněna i o návrh metodiky pro automatizaci testování webových aplikací, která poskytuje systematický návod, jak při tomto úkolu postupovat a jak jej propojit s procesem vývoje softwaru.

1.2 Cíl práce

Hlavním cílem diplomové práce je přiblížit čtenáři problematiku automatizace testování softwaru a poskytnout systematický návod pro úspěšné zavedení automatizovaných testů webových aplikací s využitím vybraného testovacího nástroje.

Dílčí cíle jsou následující:

1. vymezení základních pojmů z oblasti testování;
2. kategorizace nástrojů pro automatizované testování webových aplikací;
3. vytvoření metodiky pro automatizaci testování webových aplikací;
4. vytvoření uživatelské příručky k testovacímu nástroji Selenium WebDriver.

Výše uvedených cílů je dosaženo prostřednictvím rešerše českých i zahraničních (anglicky psaných) knih a internetových zdrojů věnovaných tématu automatizace testování. Informace získané z různých zdrojů jsou dále doplněny o vlastní zkušenosti a jejich syntézou tak vzniká komplexní metodický materiál poskytující informace, jak postupovat při zavádění automatizace testů webových aplikací a následně i samotném vytváření a spouštění automatizovaných testů napsaných v jazyku Java s využitím nástroje Selenium WebDriver.

1.3 Cílová skupina

Hlavní výstupy diplomové práce, kterými jsou metodika pro automatizaci testování webových aplikací a příručka pro nástroj Selenium WebDriver, byly vytvořeny primárně pro účely zvyšování kvalifikace testerů v rámci Kompetenčního centra SQA na Vysoké škole ekonomické v Praze. Jistě však budou přínosem pro každého čtenáře, který se zajímá o problematiku automatizovaného testování webových aplikací a uvažuje o zavedení automatizace testů s využitím nástroje Selenium WebDriver.

1.4 Předpoklady a omezení

Pro pochopení a úspěšné využití informací uvedených v této diplomové práci je potřeba, aby čtenář znal alespoň základy programování v jazyku Java, orientoval se ve zdrojovém kódu webových stránek a byl seznámen s CSS¹ a XPath² selektory. Dále je potřeba mít nainstalovaný prohlížeč Firefox v nejnovější nebo předposlední verzi (popř. v poslední nebo předposlední ESR verzi), který je použit pro spouštění automatizovaných testů vytvořených v nástroji Selenium WebDriver. [5]

Diplomová práce je zaměřena především na představení průběhu procesu automatizace testování a využití konkrétního testovacího nástroje; jejím cílem není vysvětlovat problematiku testování softwaru obecně. Pro efektivní využití poznatků z této diplomové práce v praxi je tedy vhodné se nejprve seznámit s problematikou testování softwaru, neboť znalost konkrétního testovacího nástroje je sice nutnou, ale nikoli postačující podmínkou pro to, aby se člověk stal dobrým testerem a dokázal naplánovat a úspěšně provést automatizaci testování. Jako vhodný úvod do problematiky bych doporučila knihy *Testování softwaru* od Rona Pattona [1] a *Řízení kvality softwaru* od Petra Roudenského a Anny Havlíčkové [6].

Dále bych také chtěla dodat, že přístupů k automatizaci testování existuje celá řada a metodika, která je součástí této práce, je pouze jedním z nich, přičemž se snaží poskytnout ucelený obraz procesu automatizace testování a vysvětlit jeho vztah k procesu vývoje softwaru a nezaměřuje se tolik na konkrétní praktiky a metody, jako tomu bylo v doposud vydaných knihách o testování.

Tato diplomová práce se také nezabývá testováním webových aplikací na mobilních zařízeních, neboť by začlenění této tematiky přesahovalo původně plánovaný rozsah práce. Nicméně je vhodné zmínit, že i tuto oblast Selenium WebDriver pokrývá.

¹ CSS (Cascading Style Sheets, česky kaskádové styly) – jazyk, pomocí kterého lze definovat, jak mají být jednotlivé elementy na webové stránce graficky zobrazeny. Pro identifikaci elementů, na které mají být aplikovány vybrané grafické vlastnosti, se využívají tzv. CSS selektory, které pracují s HTML kódem webové stránky. [3]

² XPath – jazyk, pomocí kterého lze vyjádřit cestu k XML elementům nebo atributům, zjišťovat jejich počet, provádět matematické operace nad získanými hodnotami atd. XPath selektory lze ale využívat i pro identifikaci HTML elementů na webové stránce. [4]

1.5 Struktura práce

V první části této diplomové práce je představena problematika automatizovaného testování, jeho výhody a nevýhody a srovnání s testováním automatizovaným. Následuje kategorizace testovacích nástrojů podle různých hledisek, včetně uvedení konkrétních komerčních i nekomerčních produktů. Poté jsou představeny nástroje spadající do projektu Selenium, mezi něž patří i Selenium WebDriver, kterému je věnována uživatelská příručka v příloze této diplomové práce. Na závěr této kapitoly jsou pak ještě uvedeny nástroje pro zkoumání struktury webové stránky, které jsou velice užitečnými pomocníky při vytváření automatizovaných testů webových aplikací.

Další kapitola je věnována metodice pro automatizaci testování webových aplikací. Na úvod jsou uvedeny nejznámější metodiky vývoje softwaru, na nichž je předvedena role testování v procesu vývoje aplikací. Následuje představení samotné metodiky pro automatizaci testování webových aplikací, popis jednotlivých fází a jejich vazba na životní cyklus softwaru, dále popisuje zúčastněné role a jejich úkoly a odpovědnosti a nakonec zachycuje postup při zavádění metodiky do praxe.

Součástí přílohy této diplomové práce je pak uživatelská příručka k nástroji Selenium WebDriver, která poskytuje komplexní návod k použití daného nástroje od vytvoření prvního automatizovaného testovacího skriptu až po pokročilé funkce a doporučení k návrhu architektury testů.

1.6 Očekávaný vlastní přínos

Hlavním přínosem této diplomové práce je vytvoření metodiky pro automatizaci testování webových aplikací, která doposud nebyla v českém jazyce k dispozici. Dalším přínosem je pak doplnění informací z doposud publikovaných knižních a elektronických zdrojů věnovaných tematice automatizovaného testování s využitím nástroje Selenium WebDriver o poznatky z vlastní testovací praxe.

1.7 Vymezení základních pojmů

Pro začátek je potřeba specifikovat alespoň základní pojmy z oblasti testování, se kterými se bude čtenář v rámci této diplomové práce často setkávat, neboť jejich vymezení je pro správné pochopení problematiky klíčové:

Testování softwaru

I přes dlouhou historii testování softwaru není definice tohoto pojmu zcela ustálena. Existuje několik formulací podle úhlu pohledu a míry detailu, do jakého je problematika rozebrána. Nalézt lze tedy jak obecné, tak úzce zaměřené a jak stručné, tak velice podrobné definice od různých autorů a institucí.

ISTQB definuje testování jako „*proces sestávající se ze všech aktivit životního cyklu, jak statických, tak dynamických, zaměřených na plánování, přípravu a hodnocení softwarových produktů a s nimi souvisejících pracovních produktů s cílem rozhodnout, zda splňují specifikované požadavky, předvést, zda jsou vhodné pro svůj účel a objevit chyby.*“³ [7]

Cem Kaner uvádí, že testování softwaru je „*empirické technické zkoumání prováděné s cílem poskytnout zainteresovaným osobám informaci o kvalitě testovaného produktu nebo služby.*“⁴ [8]

Podle Institutu pro elektrotechnické a elektronické inženýrství (Institute of Electrical and Electronics Engineers, IEEE) se testování softwaru „*skládá z dynamického porovnávání chování programu na konečné sadě testovacích případů, vhodně vybraných z obvykle nekonečného množství možností použití, s očekávaným chováním.*“⁵ [9 s. 5.1]

Roudenský a Havlíčková pak definují testování v užším slova smyslu jako „*proces řízeného spouštění softwarového produktu s cílem zjistit, zda splňuje specifikované či implicitní potřeby uživatelů.*“ [6 s. 45]

³ Původní znění: „*The process consisting of all lifecycle activities, both static and dynamic, concerned with planning, preparation and evaluation of software products and related work products to determine that they satisfy specified requirements, to demonstrate that they are fit for purpose and to detect defects.*“ [7]

⁴ Původní znění: „*Software testing is an empirical technical investigation conducted to provide stakeholders with information about the quality of the product or service under test.*“ [8]

⁵ Původní znění: „*Software Testing consists of the dynamic verification of the behaviour of a program on a finite set of test cases, suitably selected from the usually infinite executions domain, against the expected behaviour.*“ [9 s. 5.1]

Je tedy zřejmé, že pojem testování softwaru lze definovat různými způsoby, nicméně i přesto lze specifikovat jeho základní, klíčové charakteristiky:

- existují jistá **očekávání**, jak by se software měl chovat (a to nejen na základě explicitně specifikovaných požadavků, ale často i nevyslovených představ),
- software je zkoumán a **zjišťuje se jeho skutečné chování**,
- skutečné chování je porovnáváno s očekávaným chováním a na základě tohoto srovnání se **posuzuje kvalita softwaru**, o níž jsou **informovány zainteresované osoby**.

Typy testů

Cílem této diplomové práce není dopodrobna vysvětlovat problematiku testování softwaru, přesto považuji za vhodné přiblížit alespoň několik základních typů testů, které jsou v dalším textu práce zmiňovány.

Statické vs. dynamické testování:

- **Statické testování** – způsob testování, při kterém není kód programu spuštěn – jde pouze o prohlížení a kontrolu artefaktů vznikajících v procesu vývoje softwaru – nejčastěji zdrojového kódu aplikace nebo specifikace požadavků. Jeho výhodou je možnost začít provádět testování již v dřívějších fázích vývoje softwaru, kdy ještě není dokončena první testovatelná verze aplikace. Statické testování programového kódu, nazývané také revize kódu, provádí sami vývojáři. Testeři se při statickém testování zabývají především zkoumáním specifikací požadavků od zákazníka, hledají v nich nesrovnalosti a upozorňují na chybějící podstatné informace, které by mohly být později příčinou chyb ve vyvíjené aplikaci. [1 s. 49]
- **Dynamické testování** – nejznámější druh testování, který se při vyslovení pojmu testování většině lidí vybaví. Jde o testování již spuštěné aplikace, při kterém tester s aplikací pracuje podobně jako koncový uživatel a přitom zkoumá její chování a výstupy, které porovnává se specifikovanými požadavky a případné nesrovnalosti reportuje vývojářům. [1 s. 49]

Strukturální testování, funkcionální testování a nefunkcionální testování:

- **Strukturální testování, testování bílé (či průhledné) skříňky** – tester má přístup ke zdrojovému kódu aplikace a tuto znalost při testování využívá. Nevýhodou je, že testování tak může být neobjektivní, neboť je „ušito na míru“ činnosti programového kódu a nebere příliš v úvahu potřeby a požadavky zákazníka. [1 s. 49]
- **Funkcionální testování, funkční testování, testování černé skříňky** – tester nemá možnost vidět zdrojový kód aplikace a neví, jakým způsobem zpracovává data. Vidí pouze to, jak se software chová navenek a jaké výstupní hodnoty produkuje – při testování černé skříňky tedy tester zadává do aplikace vstupní hodnoty a pozoruje, zda

dostane odpovídající výsledek. Základním zdrojem informací je specifikace požadavků, na základě které se vytvářejí testovací případy a podle nich je pak testování prováděno. [1 s. 48] Specifickým typem funkcionálního testování je pak **testování bezpečnosti**, které zkoumá, zda je software schopen odolat hrozbám zvnějšku, např. počítačovým virům nebo jiným útokům hackerů. [10 s. 28]

- Na rozhraní výše uvedených kategorií je pak **testování šedé skříňky**, které je typické pro testování webových aplikací, kdy tester sice nemá možnost nahlédnout do programového kódu aplikace, ale může procházet zdrojový kód webové stránky (obvykle v HTML, CSS a JavaScriptu či jiném skriptovacím jazyce vyhodnocovaném na straně klienta). Je tedy na pomezí mezi testováním bílé skříňky a černé skříňky, stále však jde o testování funkcionální. [1 s. 173]
- Vedle funkcionálního testování lze pak také rozlišovat tzv. **nefunkcionální testování**, které je zaměřeno na vlastnosti aplikace, které přímo nesouvisí s jejími funkcemi, ale zároveň jsou podstatné pro její správné fungování. Patří sem výkonnostní testování, testování použitelnosti, testování udržitelnosti, spolehlivosti, přenositelnosti a další. Zjednodušeně řečeno, nefunkcionální testování se zaměřuje na to, JAK systém pracuje, zatímco testování funkcionální zkoumá, CO systém dělá. [11]

1. **Výkonnostní testování** – jeho cílem je zjistit, zda aplikace vykazuje požadované výkonnostní charakteristiky (doba odezvy, úspěšnost načtení stránek apod.) při plánovaném standardním zatížení budoucími uživateli. Výkonnostní testy se dále dělí na: [12 s. 35–38]

- a) **Zátěžové testy (Load Testing)** – jsou nejčastějším typem výkonnostních testů a jejich hlavním cílem je simulovat a ověřit výkonnostní předpoklady testovaných aplikací při vyšším počtu současně pracujících uživatelů – zda bude aplikace použitelná i v období špiček (pokud jde např. o e-shop, pak lze v období před Vánoci očekávat vyšší počet uživatelů, než je běžné po zbytek roku).
- b) **Objemové testy** – další z nejčastějších typů testů výkonnosti. Také simuluje víceuživatelskou práci s testovanou aplikací, přičemž se ale zaměřuje na vkládání, zpracování a získávání velkého objemu dat.
- c) **Stress testy** – „jsou testy simulující zátěž, která dosahuje hraničních podmínek, za kterých aplikace přestává fungovat. Jejich cílem je analyzovat, jakým způsobem se v takových situacích bude testovaná aplikace chovat a tím poskytnout dostatečné informace k naplánování řešení krizových situací (výpadek disku zapojeného v RAIDu, výpadku jednoho radaru systému protivzdušné obrany apod.).“ [12 s. 38]

2. **Testování použitelnosti** – jde o testování, které provádí reprezentant koncového uživatele, který nemá zkušenosti s vývojem softwaru ani s testováním, tedy nikdo z projektového týmu. Obvykle jde o dobrovolníka vybraného na základě charakteristik cílové skupiny uživatelů, který dostane k dispozici testovanou aplikaci a několik úkolů, které v ní má provést – v případě e-shopu to může být například nakoupit zboží podle zadaných charakteristik, zaregistrovat se, najít telefonní kontakt na zákaznickou podporu apod. Jeho počínání je sledováno a zaznamenáváno (obvykle kamerou) a tyto výsledky se následně vyhodnocují, identifikují se problémová místa a na jejich základě se poté formulují závěry a doporučení pro vylepšení aplikace. [13]
3. **Testování spolehlivosti** – jeho cílem je poskytnout informaci o spolehlivosti aplikace – tedy, jaká je pravděpodobnost, že bude aplikace fungovat zamýšleným způsobem, bez chyb a pádů. Tento typ testování není možné provádět manuálně, ale je nutné využít některý z nástrojů pro automatizaci testování, neboť výsledky testování spolehlivosti musí vycházet z vysokého počtu opakování testů aplikace při stejných podmínkách. [12 s. 41]
4. **Testování udržovatelnosti (Maintainability Testing)** – zkoumá, jak snadno se dá systém udržovat. Prakticky každou aplikaci je i po dokončení vývoje a předání zákazníkovi potřeba čas od času upravit tak, aby i v měnících se podmínkách dokázala naplňovat potřeby uživatelů. Testování udržovatelnosti pak zjišťuje, jak náročné je systém před plánovanou změnou analyzovat, poté změnu provést a nakonec otestovat jeho funkčnost. [14]
5. **Testování přenositelnosti** – zjišťuje, jak snadno lze aplikaci přenést z jedné platformy na jinou, typicky mezi operačními systémy (např. z Windows XP na Windows 8 či dokonce na některou z Linuxových distribucí). Sleduje, s jakou námahou lze vytvořit verzi aplikace pro jinou platformu, než pro jakou byla původně vyvinuta. [15]

Testování související se změnami v testované aplikaci:

- **Dýmové testování (Smoke Testing)** – velmi rychlé ověření, zda je software dostatečně stabilní a všechny jeho klíčové součásti fungují správně. Jeho cílem je zjistit, zda je testovaná aplikace provozuschopná a zda má smysl ji v daném stavu dále podrobovat důkladnějším testování. Tento druh testů je obvykle automatizován a prováděn automaticky po každém novém sestavení (angl. build) testované aplikace. [6 s. 70]
- **Testování konfirmační (retestování)** – přichází na řadu po opravě nalezeného defektu a jeho cílem je potvrdit, že byl daný defekt úspěšně odstraněn. Zaměřuje se tedy pouze na tu část aplikace, která byla změněna. [10 s. 29]
- **Testování regresní** – opakované testování již otestované aplikace s cílem najít všechny defekty, které mohly být nově zaneseny nebo objeveny jako následek provedených

změn (ať už oprav dříve nalezených chyb, nebo změn funkcionality aplikace či dokonce změn prostředí, ve kterém aplikace funguje). Testuje aplikaci jako celek, nikoli jen změněné části. [10 s. 29]

Manuální vs. automatizované testování

- **Manuální testování** – prováděno testerem-člověkem, který provádí testy manuálně obvykle podle předem připravených testovacích případů, vyhodnocuje správnost výsledků a chování aplikace a případné nesrovnalosti reportuje vývojářům. Manuální testování může mít podobu statických testů (revize specifikace požadavků) nebo dynamických (testování spuštěné aplikace).
- **Automatizované testování** – sada předem připravených testovacích skriptů, které jsou spuštěny nad testovanou aplikací a prověřují její vlastnosti a chování. Výsledkem je pak report zachycující, které testy proběhly bez chyby a ve kterých se naopak vyskytl rozpor mezi očekávaným a skutečným stavem. Obvykle jde o testování dynamické, tedy nad spuštěnou aplikací. Podrobnější srovnání manuálního a automatizovaného testování je uvedeno v kapitole 3 Automatizované testování softwaru.

A nakonec je vhodné zmínit i **ad hoc testování a exploratorní testování**:

- **Ad hoc testování** (náhodné, volné, angl. Random Testing, Monkey Testing) je testovací aktivita prováděná nesystematicky, bez plánování a náhodně s cílem objevit co nejvíce defektů. Ačkoli jde o testování nesystematické a zdánlivě neefektivní, v případě provádění zkušeným testerem není jeho úspěšnost zanedbatelná, proto může být vhodným doplňkem k formálnějším testovacím technikám. [6 s. 118]
- **Exploratorní testování** (angl. Exploratory Testing) sice může na první pohled vypadat stejně jako ad hoc testování, je ale systematictější a důkladnější. Při exploratorním testování tester zkoumá aplikaci, učí se s ní pracovat, snaží se pochopit logiku jejího fungování a na základě toho navrhuje a současně s tím i provádí testovací případy. Jde o velice kreativní přístup k testování a vedle nalezených chyb může poskytnout i užitečné informace o použitelnosti a uživatelské přívětivosti aplikace. [6 s. 118–119]

V oblasti testování webových aplikací je pak důležité také **testování kompatibility s různými prohlížeči** (testování napříč prohlížeči, angl. cross-browser testing). V současné době na trhu existuje spousta internetových prohlížečů a jejich verzí, prostřednictvím kterých uživatelé k webovým aplikacím přistupují. Přitom se může stát, že se webová aplikace v různých prohlížečích chová či vypadá odlišně. Není však možné provést všechny testovací případy ve všech prohlížečích a všech jejich verzích, proto je potřeba identifikovat ty nejběžnější z nich a určit, v jakém rozsahu budou testy kompatibility prováděny. Tyto testy jsou obvykle automatizovány. (Vlastní definice)

Kategorií testování softwaru však existuje nesčetné množství a výše uvedený přehled popisuje jen nejběžnější z nich.

Tato diplomová práce se zaměřuje na automatizované dynamické funkcionální testování webových aplikací (testování šedé skříňky) – automatizované testy vytvořené s využitím nástroje Selenium WebDriver pracují se spuštěnou instancí aplikace a zaměřují se na funkcionalitu aplikace, bez potřeby zkoumat pozadí jejího fungování, pouze se znalostí struktury webové stránky. Nicméně metodika pro automatizaci testování webových aplikací, která je součástí této diplomové práce, byla vytvořena tak, aby se dala stejně dobře aplikovat na automatizaci kteréhokoli jiného typu testování softwaru.

Softwarová chyba

Chyba v softwaru (angl. Bug) označuje stav, kdy je splněna alespoň jedna z následujících podmínek: [1 s. 14]

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

Specifikace požadavků na software (angl. Software Requirements Specification)

Nebo také funkční specifikace, produktová specifikace. Jde o poměrně obsáhlý dokument s detailním popisem požadavků na software na základě informací získaných od zákazníka. Definuje, jak se má software chovat a jaké by měl poskytovat výsledky, ovšem bez vazby na konkrétní technologii či detailní architekturu aplikace. Dokument by měl být psán tak, aby byl srozumitelný jak projektovému týmu, tak zákazníkovi. [16]

Součástí specifikace požadavků by měly být následující informace: [16]

1. **Úvod** – účel (proč je vlastně daná aplikace potřeba) a rozsah softwarové aplikace, reference na další dokumenty, definice pojmů a používané zkratky a struktura a obsah dokumentu.
2. **Všeobecný popis** – kontext aplikace (vazby na ostatní systémy, okolní prostředí atd.), základní přehled funkcí, profil koncového uživatele, přehled omezení, předpoklady a závislosti.

3. **Vlastní specifikace požadavků** – seznam požadavků na funkcionalitu, výkon, vlastnosti a uživatelské rozhraní aplikace. **Požadavky by měly být:**
 - **jednoznačné** – jak zákazník, tak vývojář i tester by jej měli chápat stejně,
 - **úplné** – formulace požadavku obsahuje všechny podstatné informace (nezapomínat na různé podmínky a alternativní průběh práce s aplikací – tzv. „co když...“),
 - **verifikovatelné, měřitelné** – aby bylo možné ověřit jejich splnění,
 - **konzistentní** – požadavky si navzájem neodporují,
 - **modifikovatelné** – často se stane, že je požadavek potřeba upravit,
 - **trasovatelné** – mělo by být jasné, kdo požadavek vznesl a kdo jej upravoval.
4. **Akceptační kritéria** – specifikují, jak bude ověřováno splnění uvedených požadavků, ideálně ve formě konkrétních testovacích případů.
5. **Přílohy** – např. vstupní a výstupní data, různé diagramy atd.

Specifikace požadavků je jedním z klíčových dokumentů v procesu vývoje a testování softwaru, neboť na základě ní je software vytvářen a prověřován.

Testovací případ (angl. Test Case)

Definice testovacího případu dle knihy *Jak testuje software Microsoft* od autorů Alan Page, Ken Johnson a Bj Rollinson z roku 2009 je následující: „*Testovací případ popisuje konkrétní akce, prováděné s určitou softwarovou komponentou, a jejich očekávané výsledky.*“ [17 s. 209]

Stejný zdroj také uvádí: „*Testovací případy mohou být představovány seznamem prováděných kroků a očekávaných výsledků (v případě manuálního testovacího případu) nebo sadou programových instrukcí (v případě automatizovaného testovacího případu).*“ [17 s. 209]

Jde tedy o formální dokument obsahující sadu instrukcí (kroků), podle kterých by testování dané funkcionality (ať již manuální, nebo automatizované) mělo probíhat, doplněných o očekávané výsledky každé akce.

Testovací případ by měl obsahovat následující informace: [17 s. 211]

1. **Účel** – jakou funkcionalitu či vlastnost aplikace daný test prověřuje.
2. **Podmínky** (angl. Preconditions) – za jakých podmínek může být test proveden – např. stav aplikace (např. uživatel je přihlášen a má zobrazenou úvodní stránku aplikace), dostupnost dat, rychlost internetového připojení atd.
3. **Jednotlivé kroky testu** – detailní popis všech kroků, které mají být pro otestování dané funkcionality provedeny.

4. **Vstupní data** – při práci s aplikací je obvykle potřeba zadávat nějaká vstupní data, i ta by měla být součástí testovacího případu (popř. by mělo být specifikováno, kde a jakým způsobem lze data získat).
5. **Očekávané výsledky** – jak by měla aplikace reagovat na provedení daného kroku. Tato informace je důležitá pro ověření správné funkcionality aplikace.

Manuální testovací případy jsou obvykle dokumentovány v systému pro správu testovacích případů, kde k nim mají přístup všichni členové testovacího týmu. Důvodem pro jejich pečlivou dokumentaci je nejen to, aby testeři věděli, jak při testování postupovat, ale především prokazatelnost (jaké kroky byly provedeny, kdy, kým a s jakým výsledkem) a opakovatelnost testů. Výsledky dokumentovaných testů se také dají mnohem lépe sledovat a vyhodnocovat.

2 Rešerše zdrojů na téma automatizace testování softwaru

Tématu automatizovaného testování softwaru bylo doposud věnováno velké množství knih, článků, akademických prací i webových stránek a internetových příspěvků. Nástrojem Selenium WebDriver se však zabývá jen část z nich, přičemž naprostá většina těchto textů je psána v anglickém jazyce.

V této kapitole jsou uvedeny nejvýznamnější doposud zveřejněné anglicky nebo česky psané publikace a internetové příspěvky věnované tématu automatizace testování webových aplikací a dále pak zdroje zaměřené na nástroj Selenium WebDriver.

2.1 Odborné knihy

Ačkoli na trhu existuje velké množství knih, které se věnují testování softwaru obecně, na tematiku automatizace testování softwaru jich je zaměřeno o poznání méně a většina z nich je psána v anglickém jazyce. Pro účely vytvoření této diplomové práce jsem prostudovala následující z nich (seřazeny dle roku vydání od nejnovější):

1. **Jak testuje software Microsoft**, autoři Allan Page, Ken Johnson a Bj Rollinson, rok vydání 2009. [17] Jak již název napovídá, kniha popisuje, jakým způsobem je testování softwaru prováděno ve společnosti Microsoft. Popisuje proces vývoje softwaru v dané firmě, charakterizuje pozici testera a dalších rolí v testovacím týmu, představuje různé techniky návrhu a provádění manuálních i automatizovaných testů, uvádí nástroje a systémy pro testování a nakonec se věnuje i vizím o budoucnosti testování. Nicméně automatizovanému testování se věnuje spíše okrajově.
2. **Automated Software Testing: Introduction, Management, and Performance**, autoři Elfriede Dustin, Jeff Rashka a John Paul, rok vydání 1999. [18] Tato kniha se zabývá především procesem automatizace testování softwaru, který zasazuje do uceleného rámce nazvaného ATLM (Automated test Life-Cycle Methodology). Jde o metodiku detailně popisující jednotlivé fáze procesu automatizace testování a jejich propojení s procesem životního cyklu softwaru, dále uvádí celou řadu rad a doporučení k náboru členů testovacího týmu, technik návrhu a vytváření testů, popisuje architekturu testů a to vše je doplněno mnoha případovými studii a příklady z praxe. V příloze se pak nachází podrobný návod, jak testovat požadavky, dále kategorizace nástrojů a systémů pro podporu automatizace testování, popis kariérního postupu od testera k test manažerovi a požadavky na jejich znalosti a zkušenosti, vzorový testovací plán a nakonec soubor několika nejlepších praktik (angl. Best Practices) pro automatizaci testování softwaru. Přestože jde knihu staršího data, obsahuje mnoho důležitých

informací a poznatků, které lze zcela bez problémů využít i v současné době. Tato kniha představuje velice dobrý zdroj informací pro realizaci automatizace testování.

3. **Software Test Automation: Effective use of test execution tools**, autoři Dorothy Graham a Mark Fewster, rok vydání 1999. [19] V první části knihy jsou popisovány rozdíly mezi automatizovaným a manuálním testováním, jejich výhody a nevýhody a co od nich lze očekávat, dále prezentuje různé techniky automatizace testování, které, ačkoli je kniha již staršího data, lze stejně dobře aplikovat i v dnešní době. Druhá část knihy pak poskytuje 17 případových studií, které popisují různé problémy, se kterými se při automatizaci testování přispěvatelé knihy potýkali, a jejich možná řešení. Tyto příběhy sice vzhledem ke stáří knihy ne vždy odpovídají situacím, které musí dnešní test manažeři řešit, ale i přesto si lze z některých příkladů vzít ponaučení.

O nástroji Selenium WebDriver bylo doposud vydáno několik málo knih (seřazeny sestupně dle mého hodnocení):

1. **Selenium WebDriver Practical Guide: Interactively automate web applications using Selenium WebDriver**, autor Satya Avasarala, rok vydání 2014 [20]. Tato kniha je zaměřena zejména na praktické problémy, se kterými se musí tester při psaní automatizovaných testů v nástroji Selenium WebDriver vypořádat. Kniha začíná stručnou historií nástrojů rodiny Selenium, popisuje rozdíly mezi nástrojem Selenium 1 (Selenium RC) a Selenium 2 (Selenium WebDriver) a následně předvádí nastavení projektu ve vývojovém prostředí Eclipse. Poté představuje způsoby nalézání a následně i práce s elementy webové stránky. Ve druhé kapitole popisuje pokročilejší způsoby interakce, jako různé způsoby klikání, úkon táhni-a-puť (angl. drag-and-drop), ovládání aplikace klávesnicí apod. Následně uvádí možnosti ovládání prohlížeče, pořizování snímků obrazovky v průběhu testu, práci s okny prohlížeče, čekání na různé události a práci se soubory cookies. Na rozdíl od ostatních knih také popisuje odlišnosti mezi jednotlivými internetovými prohlížeči (zejména specifika prohlížeče Internet Explorer), které jiné knihy opominají, což může začátečníka uvést do problémů. Zajímavá je také kapitola o rozhraní WebDriverEventListener, které umožňuje sledování událostí, které se v průběhu testu stanou, a o němž se jiné knihy nezmiňují. Dále se lze v této knize na rozdíl od jiných dočíst také o možnostech práce se soubory a složkami. Další kapitola popisuje, jak lze automatizované testy spouštět na vzdáleném zařízení pomocí třídy RemoteWebDriver a migraci testů napsaných v nástroji Selenium 1 do verze druhé, nazývané Selenium WebDriver, s využitím WebDriver API WebDriverBackedSelenium. Dále se dostává k nástroji Selenium Grid, který umožňuje spouštění testů paralelně na více platformách či různých internetových prohlížečích. Předposlední kapitola se věnuje návrhovému vzoru PageObject, díky kterému lze vytvořit sofistikovanou architekturu testů, kterou lze efektivněji spravovat. Poslední kapitola se pak zabývá úvodem do testování mobilních aplikací s využitím nástroje Selenium WebDriver. Dle mého názoru tato kniha poskytuje všechny podstatné informace pro začátek práce s nástrojem Selenium

WebDriver, proto bych ji zcela jistě doporučila každému, kdo uvažuje o automatizaci testování s využitím tohoto nástroje.

2. **Selenium Testing Tools Cookbook: Over 90 recipes to build, maintain, and improve test automation with Selenium WebDriver**, autor Unmesh Gundecha, rok vydání 2012 [21]. Tato kniha poskytuje komplexní návod, jak pracovat s nástrojem Selenium WebDriver, od úvodního seznámení až po zvládnutí pokročilých úkolů. Kniha popisuje, jak psát automatizované testy s využitím tohoto nástroje, představuje rozhraní WebDriver a možnosti jeho využití, zabývá se i řízením průběhu testů pomocí podmínek a čekání a ukazuje, jak pracovat s více otevřenými okny současně, pop-up okny či chybovými hláškami a upozorněními, které se na webové stránce mohou v průběhu testování objevit. Mimoto kniha poskytuje i návod, jak rozšířit nástroj Selenium o technologie podporující testování řízené daty⁶, konkrétně JUnit/Apache POI a JDBC technologie. Dále se věnuje návrhovému vzoru Page Object, testování webových aplikací na mobilních zařízeních a poskytuje návod, jak rozšířit rozhraní WebDriver a získat tak ještě mocnější testovací nástroj. Další část je věnována otázce měření výkonu testované aplikace na straně klienta pomocí nástrojů dynaTrace a HttpWatch. V neposlední řadě se pak kniha zabývá testováním HTML5 aplikací, zaznamenáváním průběhu testů v podobě videozáznamu nebo také vývojem řízeným požadavky na chování⁷ s využitím nástrojů Cucumber-JVM a JBehave pro jazyk Java, SpecFlow.NET pro .NET a Capybara pro Ruby. Nakonec kniha odkazuje ještě na další dvě kapitoly dostupné online. První z nich je zaměřena na integraci nástroje Selenium WebDriver s dalšími užitečnými nástroji, jako například Maven, Ant, Jenkins aj. Druhá kapitola pak popisuje, jak vytvořit distribuované testovací prostředí⁸ s využitím nástroje Selenium Grid. Tato kniha tedy poskytuje poměrně rozsáhlý přehled o možnostech využití a rozšíření nástroje Selenium WebDriver a dle mého názoru jde o druhou nejlepší publikaci ze všech zde zmíněných, hned vedle výše uvedené knihy *Selenium WebDriver Practical Guide: Interactively automate web applications using Selenium WebDriver*.

⁶ Testování řízené daty (Data-Driven Testing, DDT) – velice rozšířený přístup k testování využívaný v případech, kdy je potřeba opakovaně spouštět stejný test s různými vstupy nebo podmínkami a ověřovat, zda aplikace v závislosti na těchto změnách poskytuje odpovídající výstupy. Test se tedy vždy skládá ze stejných kroků, ale liší se vstupní data. [21 s. 105]

⁷ Vývoj řízený požadavky na chování (Behaviour-Driven Development, BDD) – metodika agilního vývoje, která odstraňuje některé problémy techniky vývoje řízeného testy (Test-Driven Development, TDD) a doplňuje ji o osvědčené přístupy z řady dalších metodik či technik, čímž vzniká rámec, kterým se mohou softwarové projekty řídit. [22 s. 10–11]

⁸ Distribuované testování (Distributed Testing) – označuje přístup k testování, kdy je test rozdělen na několik částí, přičemž každá je prováděna na jiném počítači. Nepředstavuje však pouze paralelní běh testů na různých počítačích, ale i situaci, kdy spolu jednotlivé části testů interagují. [23]

3. **Selenium 2 Testing Tools Beginner's Guide**, autor David Burns, rok vydání 2012 [24]. Tato kniha obsahuje postup, jak psát automatizované testy s využitím nástroje Selenium WebDriver, popisuje jeho historii a architekturu, využitelné návrhové vzory, uvádí příklady testování webových stránek na mobilních zařízeních a zmiňuje i práci s HTML 5. Mimoto se zabývá také dalšími dvěma produkty z rodiny Selenium nástrojů – doplňkem do internetového prohlížeče Firefox nazývaným Selenium IDE a nástrojem Selenium Grid, používaným ke spouštění paralelních automatizovaných testů. Nakonec uvádí i způsob, jak převést stávající automatizované testy vytvořené v nástroji Selenium 1 do novějšího nástroje Selenium WebDriver.
4. **Instant Selenium Testing Tools Starter**, autor Unmesh Gundecha, rok vydání 2013 [25]. Útlá příručka poskytující rychlý úvod do tematiky automatizovaného testování v nástroji Selenium WebDriver. Představuje čtenáři Selenium WebDriver i Selenium IDE a ukazuje práci s oběma nástroji. V části věnované nástroji Selenium WebDriver se také zabývá spouštěním automatizovaných testů v různých internetových prohlížečích a využitím návrhového vzoru Page Object. Poslední kapitola poskytuje užitečné odkazy na další rady a návody dostupné na internetu.
5. **Test Automation using Selenium WebDriver with Java: Step by Step Guide**, autor Navneesh Garg, vydáno v prosinci 2014. [26] V době psaní této diplomové práce šlo o jednu z nejnovějších knih na téma Selenium WebDriver, proto jsem ji do své práce již nestihla zařadit.
6. V rámci služby CreateSpace Independent Publishing Platform, která umožňuje vydávat knihy bez účasti nakladatele, jsou pak k dispozici i další dvě knihy - **Selenium WebDriver in Java: Learn with Examples** [27] a **Selenium WebDriver in C#.Net: Learn With Examples** [28], obě byly vydány v únoru 2014 a jejich autorem je Sagar Shivaji Salunke. Dle zveřejněných recenzí čtenářů jde však o knihy nízké kvality s mnoha chybami, a tak jsem se rozhodla je ze svých zdrojů vynechat.

2.2 Akademické práce

V době psaní této diplomové práce jsem nenalezla žádnou bakalářskou nebo diplomovou práci, která by byla zaměřena na automatizované testování webových aplikací s využitím nástroje Selenium WebDriver. Podařilo se mi však nalézt několik akademických prací, které se této tematice věnovaly alespoň okrajově:

1. Diplomová práce **Automatické testování webových aplikací**, autor Martin Sokol, Masarykova Univerzita, Brno, rok 2013. [29] Práce popisuje výhody a nevýhody automatizovaného testování uživatelského rozhraní webových aplikací a navrhuje způsob, jak ve vybrané firmě oXy Online, s.r.o., zavést komplexní automatizované testování. V práci je uveden popis všech 4 nástrojů rodiny Selenium a v části věnované

návrhovému vzoru Page Objects uvádí příklady jeho použití na kódu automatizovaného testu vytvořeného s využitím nástroje Selenium WebDriver.

2. Diplomová práce **Automation of regression testing of web applications**, autor Dávid Chmurčiak, Masarykova Univerzita, Brno, rok 2013. [30] Práce uvádí popis všech 4 nástrojů rodiny Selenium a navrhuje komplexní automatizované regresní testování webové aplikace Xythos Enterprise Document Management System (EDMS) s využitím nástrojů Selenium WebDriver a Selenium Grid, TestNG, Perforce, Ant a Hudson.
3. Diplomová práce **Automatizované testování webových aplikací**, autor Michal Pietrik, Masarykova Univerzita, Brno, rok 2012. [31] Tato diplomová práce se zabývá testováním webových aplikací na platformě ASP .NET, konkrétně návrhem automatizovaného testování webové aplikace Kentico CMS. V části věnované nástrojům pro automatizované testování popisuje mimo jiné i všechny 4 nástroje rodiny Selenium.
4. Diplomová práce **Testování webových aplikací za pomoci detekce změn jejího vzhledu**, autor Pavel Sklenář, Vysoká škola ekonomická v Praze, 2012. [32] Cílem této práce bylo vytvoření pomocné testovací knihovny, která dokáže porovnat dvě libovolné webové stránky v různých prohlížečích a zachytit jejich rozdíly. Vytvořená knihovna představuje určitou nadstavbu nad nástrojem Selenium WebDriver, která jej doplňuje o užitečné funkce z nástroje ImageMagick. Stejně jako ve výše zmíněných akademických pracích jsou i zde popsány všechny 4 Selenium nástroje.
5. Bakalářská práce **Automated Testing of the Component-based Web Application User Interfaces**, autor Juraj Húska, Masarykova Univerzita, Brno, rok 2012. [33] Práce je zaměřena na vytvoření vlastního aplikačního programového rozhraní, které by mohlo být použito pro testování uživatelského rozhraní webových aplikací. Nástroj Selenium WebDriver je zde zmíněn zejména v souvislosti s návrhovým vzorem Page Objects. Výklad tohoto návrhového vzoru je doplněn příklady automatizovaných testů vytvořených s využitím nástroje Selenium WebDriver.
6. Bakalářská práce **Funkční testování webových aplikací**, autor Břetislav Mazoch, Masarykova Univerzita, Brno, rok 2012. [34] Tato práce se zabývá představením nástrojů pro automatizované testování a popisuje, jakým způsobem lze vytvářet a spouštět funkční testy webových aplikací s využitím nástrojů Selenium, TestNG a Ant. Práce poskytuje stručný popis všech 4 nástrojů rodiny Selenium (Selenium 1, Selenium IDE, Selenium WebDriver i Selenium Grid) a poukazuje na možnost spojení Selenium WebDriver a TestNG, frameworku⁹ pro automatické provádění testů.

⁹ Framework – platforma, která usnadňuje vývoj softwarových aplikací pro určitý operační systém. Typicky obsahuje předdefinované třídy a funkce, které se v aplikacích často opakují. [35]

2.3 Internetové zdroje

Na českém internetu se doposud nevyskytuje žádný portál specializovaný na automatizaci testování, nalézt lze pouze několik webů či blogů věnovaných tematicce testování softwaru obecně, přičemž automatizací se zabývají pouze okrajově v několika příspěvcích.

Co se týče webů psaných v anglickém jazyce, ani zde není situace o mnoho lepší. Naprostá většina výsledků vyhledávání na téma automatizace testování směřuje na webové stránky výrobců nástrojů pro automatizaci testování, ale ucelené informace obecnějšího charakteru, nevázané na konkrétní nástroj, lze nalézt o poznání obtížněji. I přesto však může pátrání na internetu přinést celou řadu zajímavých informací, byť roztržitých na různých webových portálech. Zajímavé články a tutoriály se nacházejí například na webech:

1. **Testing Excellence** – <http://www.testingexcellence.com/category/automation-testing/> [36]
2. **Guru99** – <http://www.guru99.com/automation-testing.html> [37]
3. **Tutorials Point** - http://www.tutorialspoint.com/software_testing_dictionary/automated_software_testing.htm [38]

Základním online zdrojem informací o nástrojích Selenium jsou samozřejmě **oficiální webové stránky projektu Selenium HQ** (<http://docs.seleniumhq.org/>). [39] Zde se nachází popis všech 4 nástrojů rodiny Selenium (Selenium RC, Selenium IDE, Selenium WebDriver a Selenium Grid), základní návody k těmto nástrojům, instalační balíčky, odkazy na další zdroje a informace o projektu a přispívajících členech.

Dalším velice důležitým zdrojem je **oficiální dokumentace k rozhraní WebDriver pro jazyk Java** (<http://selenium.googlecode.com/git/docs/api/java/index.html>) [40], kde lze nalézt kompletní seznam balíčků, tříd, rozhraní, výčtových typů, výjimek, metod atd. Analogicky je k dispozici takový seznam i pro další jazyky podporované nástrojem Selenium WebDriver na adrese <https://code.google.com/p/selenium/> [41].

K nástroji Selenium WebDriver lze na internetu nalézt několik diskusních fór a nesčetné množství příspěvků, např. na velice populárním diskusním portálu **Stack Overflow** (<http://stackoverflow.com/questions/tagged/selenium-webdriver>) pod štítky *selenium*, *webdriver* nebo *selenium-webdriver* [42] nebo na **Selenium User's Group** fóru (<https://groups.google.com/forum/#!forum/selenium-users>) [43].

Celá řada příspěvků na téma Selenium WebDriver se nachází i na sociální síti LinkedIn v zájmových skupinách, např.:

1. **Selenium Test Automation User Group** - <https://www.linkedin.com/groups/Selenium-Test-Automation-User-Group-961927> [44],
2. **Selenium 2.0 and WebDriver** – <https://www.linkedin.com/groups/Selenium-20-WebDriver-3985798> [45],
3. **Selenium WebDriver** – <https://www.linkedin.com/groups/Selenium-WebDriver-4067187> [46].

Mimoto existuje i několik webů zaměřených na automatizované testování s využitím nástroje Selenium WebDriver s mnoha návody a doporučeními:

1. **Selenium Simplified** (<http://seleniumsimplified.com/get-started/>) [47] – poskytuje celou řadu návodů od úvodního seznámení až po pokročilé funkce doplněné video tutoriály a názornými příklady. Nabízí také možnost zakoupení online výukového kurzu pro Selenium WebDriver.
2. **Selenium Easy** (<http://seleniueasy.com/selenium-tutorials>) [48] – obsahuje mnoho stručných a výstižných návodů a řešení častých problémů při psaní automatizovaných testů s využitím nástroje Selenium WebDriver. Na tomto webu lze nalézt také informace o dalších nástrojích pro automatizaci testování a sestavení, jako například TestNG, Ant, Maven aj.
3. **Guru 99** (<http://www.guru99.com/selenium-tutorial.html>) [49] – web s mnoha výukovými kurzy zaměřenými na testování, SAP, programování webových aplikací a spoustu dalších oblastí. Poskytuje rychlý úvod do problematiky automatizovaného testování s využitím nástroje Selenium WebDriver doprovázený mnoha obrázky a názornými ukázkami.

Popularita nástroje Selenium WebDriver již dosáhla takové úrovně, že už i portály specializované na online výuku začaly nabízet online kurzy zaměřené na práci s tímto nástrojem. Takový kurz lze absolvovat např. na **Udemy** (<http://www.udemy.com/selenium-for-entrepreneurs/>) [50].

Pro ty, kteří mají zájem se podílet na dalším vývoji nástrojů Selenium, je pak určeno **GitHub úložiště pro vývojáře frameworku Selenium** (<https://github.com/seleniumhq/selenium>) [51], kde lze nalézt projektovou Wiki, zdrojové kódy, hlášené problémy atp.

3 Automatizované testování softwaru

Každou vytvořenou softwarovou aplikaci je potřeba před představením zákazníkovi důkladně otestovat, aby se minimalizovalo množství kritických chyb v aplikaci a zákazník neměl důvod k nespokojenosti. Toto testování je obvykle nutné provést v několika iteracích, neboť každou opravenou chybu je potřeba znovu přetestovat a prověřit, zda byl problém skutečně odstraněn (provést tzv. retestování) a opravou nebyly do aplikace zaneseny chyby jiné (tedy regresní testování). [1 s. 186]

Provádění testovacích scénářů se opakuje tak dlouho, dokud se již v aplikaci neprojeví žádné závažné chyby nebo dokud není testování ukončeno z jiného důvodu (např. do určitého data nebo na základě rozhodnutí test manažera). Tester tedy musí projít aplikaci několikrát podle stejného testovacího případu, a právě zde se naskýtá příležitost k automatizaci. Automatizované testy mohou opakovaně prověřovat aplikaci podle připravených testovacích skriptů, zatímco tester má více času na manuální testování, při kterém může objevit další chyby, na které automatizovaný test nebyl připraven nebo to jednoduše není v jeho silách (ano, i automatizované testy mají své limity), a další aktivity, jako vytváření testovacích případů, revizi specifikace požadavků atd.

Tato kapitola se věnuje automatizovanému testování webových aplikací, které je ústředním tématem této diplomové práce. Nejprve charakterizuje automatizované testování jako takové, srovnává jej s manuálním testováním a uvádí jejich výhody a nevýhody. Následně se zabývá nástroji pro automatizované testování, které kategorizuje do skupin podle jejich zaměření. Detailněji se pak věnuje jednotlivým nástrojům rodiny Selenium, mezi které patří Selenium Remote Control, Selenium IDE, Selenium Grid a nakonec také Selenium WebDriver. Nakonec pak uvádí také nástroje pro zkoumání struktury webové stránky, které uživateli usnadňují práci při vytváření automatizovaných testů.

3.1 Charakteristika automatizovaného testování

Pro usnadnění průběhu testování a zkrácení doby jeho trvání bylo doposud vytvořeno obrovské množství testovacích nástrojů. Tyto nástroje umožňují testovací případy automatizovat nebo alespoň ulehčit jejich manuální provádění.

Jak je zřejmé, rozdíl v automatizovaném a manuálním testování spočívá v subjektu, který testování provádí. Zatímco manuální (ruční) testování provádí tester-člověk, automatizované testování je realizováno testovacím nástrojem, tedy strojem, který spouští předpřipravený zdrojový kód testů vytvořených testerem nebo vývojářem. [6 s. 175]

Nejdůležitějšími vlastnostmi testovacích nástrojů a automatizovaných testů jsou: [1 s. 186]

1. **Rychlost** – nástroje pro automatizované testování dokáží testovat několikanásobně rychleji než člověk, což zejména v případě velkého množství testovacích případů představuje významnou úsporu času. Díky tomu je možné provádět opakované regresní testování i v časovém presu při blížícím se termínu akceptačních testů.
2. **Efektivita** – díky tomu, že se automatizované testy věnují opakovaným regresním testům, má tester čas se věnovat jiným aktivitám, které jsou v danou chvíli potřeba, např. promýšlení nových testovacích případů nebo detailnímu manuálnímu testování.
3. **Přesnost** – po manuálním provedení několika stovek testovacích případů již člověk zákonitě přestává být pozorný a začne dělat chyby, což se testovacímu nástroji stát nemůže.
4. **Neúnavnost** – testovací nástroj se neunaví a může provádět stále stejné testovací případy kdykoli a se stejným pracovním nasazením.

I přes výše uvedené klady je nutné mít na paměti, že žádný testovací nástroj není samospasný a nedokáže plně nahradit práci testera, pouze mu pomáhá odvádět práci rychleji a lépe. Vždy je potřeba software otestovat i manuálně, neboť i sebelepší sada automatizovaných testů nedokáže pokrýt všechny oblasti, kde by se v aplikaci mohly vyskytnout chyby. Automatizované testy je navíc potřeba pravidelně aktualizovat podle provedených změn v aplikaci, jediné tak mohou být výsledky testování spolehlivé a mít vypovídající hodnotu. [6 s. 176–177]

Vzhledem k tomu, že jsou v současné době vyvíjené aplikace stále komplexnější a jejich manuální testování v potřebném rozsahu by vzhledem ke snaze o co nejkratší dobu dodání produktu nebylo možné vůbec stihnout, se využívání testovacích nástrojů těší stále větší oblibě. Tyto nástroje se navíc stávají sofistikovanějšími a umožňují automatizaci stále většího rozsahu činností, které dříve zvládl provádět pouze člověk. [6 s. 175]

Přesto ale nelze říci, že je automatizované testování vhodné zavádět vždy a všude. Stejně jako v případě jiných rozhodnutí činěných v průběhu řízení projektu je potřeba porovnávat přínosy a náklady plánované změny. Vytvoření sady automatizovaných testů trvá obvykle několik týdnů a nemá tedy je smysl zavádět na malých, krátkodobých projektech, kde se aplikace rychle mění a vyvíjí. V takových případech by čas strávený nad správou testů mohl převýšit čas, který by byl věnován manuálnímu testování. Naopak vhodné je zavést automatizaci testů na větších projektech, kde je potřeba opakovaně provádět velké množství testů a testovaná aplikace se mění pouze pomalu. [6 s. 183]

Manuální testování má nespornou výhodu, že člověk dokáže vnímat mnohem širší záběr vlastností webové aplikace než jakýkoli automatizovaný test, zejména co se týče jejího vzhledu a použitelnosti. Automatizovaný test je obvykle zaměřen na zkoumání správnosti provádění určité funkcionality, například zda se po kliknutí na odkaz otevře správná stránka, ale obvykle neobsahuje další kontroly, které by pokryly celou stránku komplexně – např. barvu prvků na

stránce, jejich umístění a velikost, smysluplnost zobrazených dat apod. – jednoduše takové skutečnosti, které by tester ihned zaznamenal, aniž by na ně musel směřovat svou pozornost. Navíc si tester dokáže operativně poradit s celou řadou nestandardních situací, které by automatizovaný test reportoval jako chybu aplikace, ačkoli tomu tak není a chyba je pouze v návrhu testu, který s danou situací nepočítá – např. nelze objednat zboží, které právě není na skladě. Dobře navržené testovací scénáře navíc mohou provádět i méně zkušené testery, zatímco pro vytváření automatizovaných testů je již potřeba vysoce kvalifikovaných pracovníků. [6 s. 177–178]

V praxi se automatizace uplatňuje především při regresním testování. Využit je ale lze například i při testech jednotkových, integračních nebo testech nefunkčních (např. zátěžových). [6 s. 178]

Přehledné srovnání výše uvedených výhod i nevýhod manuálního a automatizovaného testování je uveden v následující tabulce (Tabulka 1).

Tabulka 1: Srovnání manuálního a automatizovaného testování (Zdroj: autorka)

Manuální testování	Automatizované testování
Rychlý začátek testování – začít testovat podle testovacích scénářů lze prakticky okamžitě.	Dlouhá příprava automatizovaných testů – musí se nejprve naprogramovat.
Možnost využít méně zkušené testery.	K vytváření automatizovaných testů je potřeba kvalifikovaný a zkušený tester, obvykle se znalostí programování.
Operativní řešení nestandardních situací.	Automatizovaný test vyhodnotí nestandardní situaci jako chybu, i když se o chybu aplikace nejedná.
Širší záběr testování – člověk si všimá více aspektů testované aplikace.	Automatizovaný test zkoumá pouze to, na co byl navržen, ostatní skutečnosti nevyhodnocuje.
Člověk vnímá i vlastnosti aplikace, které nelze exaktně vyjádřit – např. použitelnost a uživatelskou přívětivost aplikace (angl. usability).	Nástroje pro automatizaci testů jsou limitovány pouze na ověřování exaktních skutečností, použitelnost a uživatelskou přívětivost aplikace vyhodnotit nedokáží.
Rychlost provádění limitována fyzickými možnostmi člověka.	Provádění automatizovaných testů je několikrát rychlejší než při manuálním testování.
Únava a ztráta pozornosti testerů.	Automatizovaný test se nikdy neunaví a bude provádět testy se stále stejnou precizností.
Riziko chyb lidského faktoru.	Eliminace rizika chyb lidského faktoru.
Nejsou potřeba žádné další nástroje, pouze testery.	Náklady na testovací nástroje – cena licence atp.
Vhodné pro malé, krátkou dobu trvající projekty s rychle se měnícími požadavky.	Vhodné pro větší projekty trvající v řádu měsíců či déle, kde je potřeba opakovaně provádět velké množství testovacích případů a/nebo otestovat velké množství vstupních a výstupních dat.

Je tedy zřejmé, že každý typ testování má své výhody i nevýhody a nelze obecně říci, že je jeden z nich lepší nebo horší. Tohoto faktu by si měl být každý manažer, který plánuje a koordinuje testování, vědom a s ohledem na potřeby daného projektu zhodnotit, která varianta bude vhodnější. V mnohých případech je ideální kombinace obou možností; testování částečně automatizovat, ale využít i manuální testování, které odhalí široké spektrum dalších chyb.

3.2 Nástroje pro automatizované testování webových aplikací

Tato kapitola popisuje různé typy nástrojů pro automatizaci testování webových aplikací, které jsou v současné době dostupné na trhu. Tím naplňuje jeden z cílů této diplomové práce, kterým je kategorizace nástrojů pro automatizované testování webových aplikací.

Vzhledem ke stále vzrůstající popularitě automatizovaných testů je v současné době na trhu dostupná celá řada nástrojů.

Podle toho, zda je k testované aplikaci přistupováno z pohledu vývojáře nebo koncového uživatele, lze nástroje pro automatizované testování rozdělit do dvou základních kategorií: [52]

1. **Nástroje pro testování řízené kódem** (angl. Code-Driven Testing) – testování založené na znalosti zdrojového kódu aplikace, obvykle jde o automatizované testování veřejných rozhraní (angl. public interfaces) tříd, modulů či knihoven, kterým jsou poskytovány různé hodnoty vstupních parametrů, a následně se ověřuje, zda vrací správné výsledky. Typicky jde o nástroje pro jednotkové testování či nástroje pro vytváření testů v rámci vývoje řízeného požadavky na chování (viz níže).
2. **Nástroje pro testování grafického uživatelského rozhraní** (angl. GUI Testing) – testování aplikace z pohledu uživatele založené na znalosti požadavků zákazníka. Testovací nástroj přistupuje ke grafickému uživatelskému rozhraní testované aplikace a generuje různé události jako stisknutí tlačítka myši nad vybraným elementem webové stránky či vepsání hodnot do vstupních polí a vyhodnocuje, zda reakce testované aplikace odpovídají očekávanému chování. Typicky jde o nástroje pro funkcionální testování, které se snaží co nejvěrněji napodobit práci uživatele.

Dále lze nástroje pro automatizované testování kategorizovat také dle úrovní testování (úrovně testování jsou blíže představeny v kapitole 4 Metodika pro automatizaci testování webových aplikací), seřazeno od nejnižší úrovně po nejvyšší:

1. **Nástroje pro jednotkové testování** (testy jednotek) - jde o testování funkcionality jednotlivých tříd, metod či procedur (neboli jednotek) zdrojového kódu testované aplikace, přičemž tento typ testů mají obvykle na starosti vývojáři, neboť vyžadují dobrou znalost programování. Typicky jde o bezplatný testovací framework svázaný s některým programovacím jazykem, např. JUnit (Java), TestNG (Java), JTest (Java),

NUnit (C# .NET), CSUnit (C# .NET), Check (C), QUnit (JavaScript) a spousta dalších.

2. **Nástroje pro integrační testování** – zatímco jednotkové testování se zaměřuje na testování každé logické jednotky webové aplikace zvlášť, testování integrační stojí o stupeň výše a testuje ji napříč jednotlivými jednotkami, kterými mohou být třídy, moduly, subsystémy a další komponenty tvořící komplexní systém. Stále však jde o testování na úrovni zdrojového kódu, proto lze některé z nástrojů pro jednotkové testování stejně dobře využít i pro testování integrační. Nicméně je nutno podotknout, že nástrojů specializovaných na integrační testování je výrazně méně než nástrojů pro jednotkové testování.
 - Komerční – VectorCAST (C++), LDRA TBrun a další.
 - Zdarma – ConSol Citrus, TestNG a další.
3. **Nástroje pro systémové testování** – tato kategorie je pravděpodobně počtem nástrojů nejrozsáhlejší. Není divu, jde totiž o poslední úroveň testů před představením produktu zákazníkovi, a tudíž je zde kladen největší důraz na odhalení co největšího počtu chyb, které by mohly být příčinou nespokojenosti zákazníka či uživatele. V této kategorii lze nalézt jak nástroje pro funkcionální testování (kterých je naprostá většina), tak nefunkcionální.
 - Komerční – HP Quick Test Professional, IBM Rational Functional Tester, SmartBear TestComplete, Oracle Functional Testing, Telerik Test Studio, TestPlant eggPlant, Verisium vTest, Tellurium, Helium, Sahi a velké množství dalších.
 - Zdarma – Selenium WebDriver, Selenium IDE, Eclipse Jubula, Watir, Wetator, Capybara, Canoo WebTest a další.
4. **Nástroje pro akceptační testování** – poslední, nejvyšší úroveň testování probíhá při představení produktu zákazníkovi, který si aplikaci vyzkouší a vyhodnotí, zda naplňuje specifikované požadavky či nikoli. Tato úroveň testů obvykle není automatizována, neboť se tak často neopakuje a navíc je potřeba vyhodnotit mnohem širší spektrum aspektů, než jsou současné nástroje schopny pokrýt. Nicméně existují nástroje, které proces akceptačních testů usnadňují, jejich funkcionalita je však velmi podobná jako u nástrojů pro systémové testování, jen je více kladen důraz na to, aby byl obsah testů srozumitelný i pro zákazníka, který nemá žádnou nebo minimální znalost programování a informačních technologií obecně.
 - Komerční – TestPlant eggPlant, Cucumber a další.
 - Zdarma – FitNesse, Robot Framework a další.

Vedle toho lze nástroje pro automatizované testování rozlišovat i podle typu prováděných testů: [6 s. 185–186]

1. **Nástroje pro funkcionální regresní a dýmové (smoke) testování** – testují, zda funkcionální aplikace odpovídá požadavkům specifikovaným zákazníkem, a pomáhají zefektivnit provádění opakovaných funkcionálních testů, zejména regresních a smoke testů. Obvykle fungují na principu testování uživatelského rozhraní webových aplikací a snaží se co nejvěrněji simulovat práci uživatele s testovanou aplikací.
 - Komerční – HP Unified Functional Testing (dříve Testing Quick Test Professional), IBM Rational Functional Tester, SmartBear TestComplete, Oracle Functional Testing, Telerik Test Studio, TestPlant eggPlant, Verisium vTest, Tellurium, Helium, Sahi a velké množství dalších.
 - Zdarma – Selenium WebDriver, Selenium IDE, Eclipse Jubula, Watir, Wetator, Capybara, Canoo WebTest a další.
2. **Nástroje pro testování kompatibility webové aplikace s různými internetovými prohlížeči** (angl. Cross-Browser Testing Tools) – uživatelé obvykle používají různé internetové prohlížeče a v různých verzích, proto je potřeba otestovat, jak se v nich aplikace chová – zda je možné s ní pracovat, zda je její vzhled ve všech prohlížečích stejný apod. I tuto činnost lze zautomatizovat pomocí testovacích nástrojů.
 - Komerční – Ranorex, Ghostlab, BrowserStack, Sauce Labs, CrossBrowserTesting, Browsershots, Spoon Browser Sandbox a další.
 - Zdarma – Lunascape Orion, IETester (pouze pro různé verze prohlížeče Internet Explorer). Open-source nástrojů je v této oblasti velice málo.
3. **Nástroje pro testování výkonnosti** – monitorují a vyhodnocují rychlost a výkon testované aplikace pod vybranou úrovní zátěže, a to buď v podobě velkého počtu současně přístupujících uživatelů, nebo poskytováním velkého množství vstupních dat, které musí aplikace zpracovat.
 - Komerční – HP LoadRunner, IBM Rational Performance Tester, LoadStorm, WebLoad, Appvance PerformanceCloud, NeoLoad, Telerik Test Studio, SmartBear LoadComplete, SmartBear LoadUI, Oracle Load Testing, Load Impact, Loadster, Applause Load Testing a další.
 - Zdarma – Apache JMeter, Gatling a další. „Živých“ open-source nástrojů na testování výkonnosti je velice málo, ačkoli lze na internetu nalézt celou řadu nástrojů, většina jich nebyla několik let aktualizována a z toho důvodu byly z tohoto seznamu vyřazeny.

4. **Nástroje pro testování bezpečnosti** – umožňují definovat bezpečnostní pravidla, jejichž splnění má být testováno, a následně simulují různé formy bezpečnostních útoků a ověřují, zda je vůči nim testovaná aplikace imunní.
 - Komerční – IBM Security AppScan, HP Fortify Software Security Center, Tenable Network Security Nessus, Rapid7 Metasploit, Core Security Core Impact a další.
 - Zdarma – Pentest-Tools, W3af, OWASP Zed Attack Proxy a další. Placených nástrojů je však výrazně větší nabídka a jejich funkcionalita je širší.
5. **Nástroje pro validaci webových stránek** – kontroluje zdrojový kód webových stránek oproti nejrozšířenějším webovým standardům. Tato validace poskytuje informace o kvalitě vytvořené aplikace – pokud vývojáři dodržují standardy, pak je kód přehlednější, lépe udržovatelný a obvykle také obsahuje méně chyb. [53]
 - Komerční – Total Validator, HTML Validator Pro a další.
 - Zdarma – W3C Validator, Validome a další.
6. **Nástroje pro vývoj řízený požadavky na chování** (BDD – Behavior Driven Development) – ačkoli se může na první pohled zdát, že jde o nástroje pro podporu vývoje, lze je svým způsobem chápat i jako testovací nástroje. Jde o specifický přístup k vytváření testů, kdy se ještě před napsáním první řádky kódu testované aplikace napíšou požadavky na chování aplikace ve formě uživatelských příběhů (angl. user stories) a v průběhu vývoje aplikace se pak k těmto řádkům testu přidávají reference na již implementovanou funkcionalitu. Takto se postupně dojde až do stavu, kdy jsou všechny specifikované požadavky pokryty kódem testované aplikace a testy úspěšně prochází bez chyb. [54] Vzhledem k nutnosti provázanosti testu a kódu je tento typ automatizovaných testů obvykle doménou vývojářů.
 - Na trhu jsou k dispozici především open-source nástroje, např. Cucumber (podporuje více než 15 programovacích jazyků), JBehave (Java), Behat (PHP), Concordion (podporuje 6 programovacích jazyků) a další.

Mezi další druhy nástrojů pro testování webových aplikací patří např. nástroje pro kontrolu funkčnosti hypertextových odkazů, nástroje pro hodnocení přístupnosti webových stránek (angl. Web Accessibility Evaluation Tools), nástroje pro testování webových stránek na mobilních zařízeních atd.

Další nástroje, které proces testování usnadňují:

1. **Nástroje pro sledování stavu zaznamenaných chyb** (angl. Bug Tracking Tools) – v současné době jde o prakticky nepostradatelný nástroj pro podporu testování. Slouží především pro zaznamenávání nalezených defektů a dohled nad jejich opravou, což je jeden z hlavních cílů testování. Tyto nástroje umožňují v předem dané formě zdokumentovat nalezený defekt, přiřadit jej vývojáři, který má zajistit jeho opravu,

a následně přiřadit testerovi, který po provedení retestu zaznamená, zda byl defekt opraven či neopraven (v tom případě se pak defekt vrací zpět k vývojáři). Dokumentují tak celý životní cyklus odhalených chyb, čímž napomáhají zajistit opravu nalezených defektů a poskytují test manažerovi důležité informace pro řízení procesu testování. [18 s. 79–80] Často jsou nástroje pro sledování stavu zaznamenaných chyb součástí komplexního řešení pro řízení procesu testování (angl. Test Management Tools).

- Komerční – HP Quality Center, JIRA, IBM Rational Quality Manager, SpiraTeam, Gemini, YouTrack a další.
- Zdarma – Bugzilla, Mantis BT, Trac, Redmine a další.

2. **Nástroje pro správu testovacích případů** (angl. Test Case Management Tools) – další kategorií nástrojů, bez kterých už si mnozí testeři nedokáží svou práci ani představit, jsou nástroje pro dokumentaci testovacích případů. Umožňují zaznamenávat testovací případy ve standardizované podobě, provádět je, zaznamenávat jejich výsledky a generovat přehledné reporty o stavu testování. [55] Stejně jako nástroje pro sledování stavu zaznamenaných chyb jsou i nástroje pro správu testovacích případů součástí komplexního řešení pro řízení procesu testování (angl. Test Management Tools).

- Komerční – HP Quality Center, IBM Rational Quality Manager, TestLodge, TestRail, ApTest, Catch Enterprise Tester, QMetry, QA Complete, PractiTest, qTest, SpiraTest, TestCaddy, TestCollab, Tosca, Zephyr a další.
- Zdarma – TestLink, Tarantula, TestCube, Testopia, XQual a další. Na internetu lze však nalézt spoustu již dlouho neaktualizovaných nástrojů, pouze malá část těchto nástrojů je konkurenceschopná.

3. **Nástroje pro správu požadavků** (angl. Requirements Management Tools) – umožňují zaznamenávat požadavky zákazníka ve standardizovaném formátu, spravovat je, propojovat s testovacími případy a dalšími artefakty vznikajícími při vývoji softwaru a sledovat jejich stav – tedy, zda jsou požadavky splněny či nikoli. Tyto nástroje obvykle fungují online, díky čemuž jsou aktuální požadavky vždy dostupné celému týmu. [56]

- Komerční - HP Quality Center, JIRA, IBM Rational, SpiraTest, Seapine Software TestTrack, Visure Requirements, RequirementONE, Borland Caliber, Accompa, FogBugz, TraceCloud, ApTest a další.
- Zdarma – aNimble, rmToo. Open source nástrojů v této oblasti není mnoho.

4. **Nástroje pro spouštění automatizovaných testů** – automatizace testování je efektivní až tehdy, pokud jsou testy ne jen prováděny, ale také spouštěny automaticky, bez nutnosti zásahu fyzické osoby – testera. K tomuto účelu se výborně hodí nástroje pro kontinuální integraci softwarového projektu, které po každé potvrzené změně kódu (angl. commit) nebo ve stanovenou dobu (obvykle v noci) projekt na dedikovaném

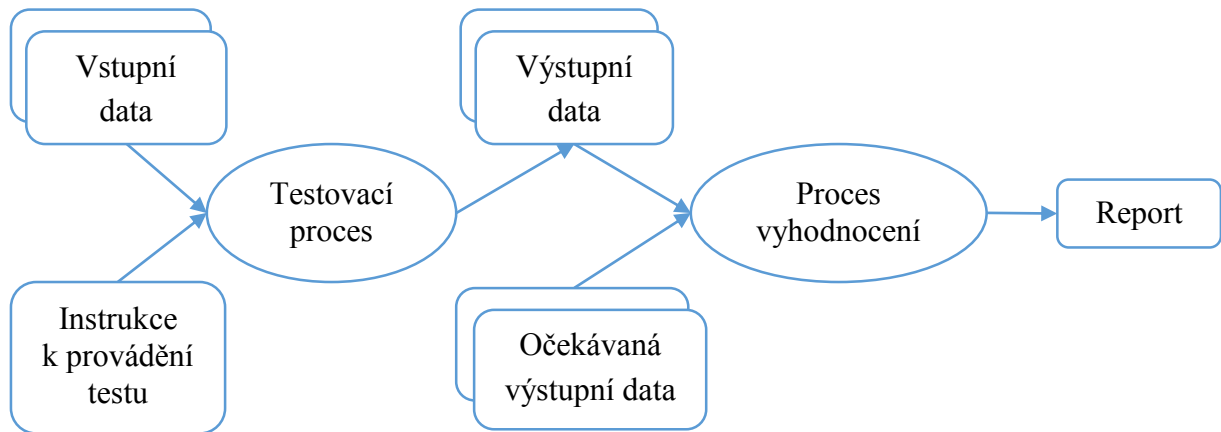
serveru sestaví a následně nad ním spustí sadu automatizovaných testů. Díky tomu je zaručena rychlá zpětná vazba a tedy i schopnost rychlejší reakce na vzniklé problémy.

- Komerční – Atlassian Bamboo, Travis CI, CircleCI a další.
- Zdarma – Jenkins/Hudson, TeamCity, CruiseControl a další.

5. **Nástroje pro analýzu pokrytí kódu testy (angl. Test Coverage Analysis Tools)** – poskytují informaci o pokrytí kódu testované aplikace automatizovanými testy. Typicky se využívá při jednotkovém testování, kde nástroj obvykle procentem a grafickým vyjádřením znázorňuje, jak velká část kódu aplikace je pokryta testy. Sem patří například nástroje JaCoCo (Java), EMMA (Java), JCover (Java), NCover (.NET), Testwell CTC++ (C a C++), TestCocoon (C, C++, C#) a další. Některé z výše uvedených nástrojů jsou však využitelné i pro analýzu pokrytí kódu u funkcionálních testů, které samy o sobě nemají přímou vazbu na kód testované aplikace. Umí totiž sledovat, jaké části kódu jsou v průběhu provádění automatizovaného testu volány, a na základě toho vypočítávají rozsah pokrytí kódu testy. Např. pro zjišťování pokrytí kódu testy napsanými v programovacím jazyce Java s využitím nástroje Selenium WebDriver lze využít kombinaci nástrojů Sonar a JaCoCo. [57]

Některé z uvedených nástrojů jsou zařazeny do více kategorií (např. HP Quality Center, IBM Rational, SpiraTest, ApTest, Telerik Test Studio a další), neboť jde o komplexní systémy pro podporu řízení celého procesu testování (tzv. Test Management Tools) obsahující více modulů.

Ačkoli na trhu existuje obrovské množství nástrojů pro automatizaci testování, prakticky všechny fungují na jednom obecném principu. Každý test je definován vstupními daty, instrukcemi, jak má být test prováděn, a očekávanými výstupními daty. Tyto tři prvky jsou pro všechny automatizované testy zásadní a ovlivňují kvalitu výstupů testování. Na základě srovnání očekávaných výstupních dat s výstupy skutečnými je pak vygenerován report, který obsahuje informace o výsledcích každého běhu testu a v případě nalezených chyb také poskytuje informaci, ve které fázi testu došlo k rozdílu mezi očekávaným a skutečným výsledkem. Celý proces je zachycen na následujícím obrázku (Obrázek 1). [19 s. 310]



Obrázek 1: Princip fungování nástrojů pro automatizaci testování (Zdroj: [19 s. 311], překresleno a přeloženo autorkou)

Jednotlivé nástroje se pak liší tím, na jakou oblast testování se zaměřují, jaké možnosti poskytují a do jaké míry jsou srozumitelné uživatelům bez znalosti programování nebo naopak zda umožňují aplikovat algoritmy blízké vývojářům.

3.3 Nástroje rodiny Selenium

Pod názvem Selenium se skrývají čtyři různé nástroje sloužící k testování webových aplikací, které v průběhu historie postupně vznikaly v rámci projektu Selenium, za kterým dnes stojí skupina více než 30 vývojářů. [58] Nástroje rodiny Selenium se liší jak datem vzniku, tak svým zaměřením a šíří možností využití. Selenium WebDriver, který je předmětem zájmu této diplomové práce, je jedním z nich.

3.3.1 Historie

Historie projektu Selenium začala roku 2004, kdy Jason Huggins, Paul Gross a Jie Tina Wang, kteří tehdy pracovali ve společnosti ThoughtWorks v Chicagu, vytvořili nástroj nazvaný JavaScript Test Runner. Byli totiž postaveni před úkol otestovat interní webovou aplikaci na několika různých prohlížečích, ale nedokázali najít žádný cenově dostupný nástroj pro automatizované testování, který by vyhovoval jejich potřebám. Rozhodli se tedy vytvořit vlastní nástroj, který by umožňoval ovládat internetový prohlížeč podobným způsobem, jako to dělá uživatel. Všechny prohlížeče, na kterých měla být aplikace testována, podporovaly JavaScript, a tak bylo možné postavit Selenium API na tomto skriptovacím jazyku. Dalším problémem, se kterým se tvůrci nástroje museli vypořádat, byl fakt, že většina testerů tehdy nedisponovala velkými programátorskými znalostmi (a popravdě toto platí dodnes). Proto se autoři nástroje snažili syntax testů co nejvíce zjednodušit a zpřehlednit. Vzniklo tak tabulkové třísloupcové uspořádání příkazů, podobné, jako lze vidět v dnešním Selenium IDE (viz Obrázek

3). [59] Jason Huggins viděl v nástroji JavaScript Test Runner velký potenciál, a tak jej zveřejnil jako open source software. Později byl nástroj přejmenován na Selenium Core. [60]

Protože byl tento testovací nástroj napsán v JavaScriptu, bylo nutné nalézt způsob, jak se vypořádat s konflikty se zabezpečením internetového prohlížeče (tzv. Same Origin Policy, která znemožňuje JavaScriptovému kódu přistupovat k elementům z jiné domény, než odkud je skript spouštěn). Kvůli těmto bezpečnostním opatřením je totiž JavaScriptový kód pouštěn z prostředí sandboxu¹⁰, odkud není možné provést celou řadu úkonů, např. přechod z HTTP do HTTPS protokolu, což se zcela běžně děje při přihlašování do aplikací. Jediným řešením bylo umístit kód testů na stejný server, na kterém běžela testovaná aplikace. To však často nebylo vždy možné, zejména pokud šlo o komerční projekt. Navíc se ukázalo, že tabulková syntax neposkytuje takové možnosti a komfort práce s kódem jako plnohodnotné programovací jazyky (např. Java) a integrované vývojové prostředí¹¹. Bylo tedy potřeba vymyslet jiné řešení. [24 s. 64]

Paul Hammant se tedy pustil do vývoje jiného nástroje, který by umožňoval psát testy v některém z populárních programovacích jazyků a využívat běžné integrované vývojové prostředí. Vznikl tak nástroj s názvem Selenium Remote Control (Selenium RC), který byl sice stále založen na Selenium Core (a šlo tedy stále o JavaScriptový kód), ale přitom zároveň využíval jazyk Java jako web server [24 s. 64] a komunikoval s testovanou webovou aplikací vzdáleně, pomocí HTTP požadavků. Hlavní myšlenkou bylo využít HTTP proxy tak, aby si prohlížeč myslel, že testovaná aplikace i testovací skript pocházejí ze stejné domény (čímž se jim podařilo obejít tolik proklínanou Same Origin Policy). [60] Díky tomuto přístupu začalo být možné psát testy prakticky v jakémkoli programovacím jazyce, který dokáže zaslat HTTP požadavek na danou URL adresu. Vznikla tak řada knihoven pro programovací jazyky jako Java, Ruby, Python, Perl, PHP a C#, které obsahovaly rozhraní se seznamem funkcí, kterými je možné z kódu spouštět speciální Selenium příkazy (označované jako „Selenese“) a ovládat tak testovanou aplikaci. [62]

Selenium RC API vycházelo z již zmíněného Selenium Core, ale zároveň bylo možné jeho metody využívat v kódu některého z podporovaných programovacích jazyků, díky čemuž se výrazně rozšířily možnosti jeho využití a testy bylo možné snáze spravovat. Zachování původní struktury Selenium příkazů však znamenalo potřebu dalšího rozvoje rozhraní, aby dokázalo pokrýt všechny úkony a situace, které v testované webové aplikaci mohou nastat. V průběhu času tak bylo neustále doplňováno o nové funkce, až jejich počet narostl na přibližně 140 metod a rozhraní se stalo poměrně nepřehledným. Tou dobou se navíc začaly objevovat HTML5 aplikace a mobilní zařízení, jejichž testování již Selenium RC nedokázalo zvládnout. [24 s. 64]

¹⁰ Sandbox – speciální prostředí internetového prohlížeče, které aplikuje bezpečnostní politiky a zabráňuje spouštění škodlivý kód na zařízení klienta. [24 s. 64]

¹¹ Integrované vývojové prostředí (Integrated Development Environment, IDE) – sada programů, která nabízí řadu funkcí usnadňujících vývoj programů. [61 s. 38]

Simon Stewart se tedy pokusil o zcela jiný přístup k ovládání prohlížeče. Ve společnosti ThoughtWorks začal v roce 2006 pracovat na novém projektu, který by odstranil dosavadní problémy a zároveň reflektoval trend vývoje od procedurálního programování směrem k objektovému. [24 s. 65] Hlavní myšlenkou bylo propojit testovací nástroj přímo s prohlížečem a ovládat jej nativně¹² a už se nemusel nechat omezovat prostředím sandboxu. Roku 2007 byla zveřejněna první část kódu nástroje WebDriver a záhy po uvolnění jej začaly podporovat dva nejpopulárnější internetové prohlížeče – Internet Explorer a Firefox. [59]

Vedle sebe tedy existovaly dva odlišné nástroje, jejichž hlavním posláním bylo v podstatě to samé – simulovat činnost uživatele na webové stránce. Architektura těchto dvou nástrojů byla velice odlišná, ale ani jeden nebyl dokonalý. Zatímco Selenium RC API mělo podobu seznamu metod v jediné třídě, WebDriver API bylo objektově orientované, rozdělené logicky do tříd. Nevýhodou nástroje WebDriver naopak bylo, že byl v té době dostupný pouze pro jazyk Java, zatímco Selenium RC podporovalo celou řadu programovacích i skriptovacích jazyků. Naopak Selenium RC bylo limitováno možnostmi HTTP požadavků, pomocí nichž testovanou aplikaci ovládal, a navíc vyžadoval existenci vlastního Selenium RC Serveru, který by fungoval jako proxy server a interpretoval příkazy Selenium Core (to vše kvůli nutnosti obejít Same Origin Policy). WebDriver přitom disponoval přímou podporou výrobců internetových prohlížečů, kteří umožnili WebDriver nativně propojit přímo s prohlížečem a obejít tak jeho bezpečnostní model. Jak je tedy zřejmé, oba dva nástroje měly své přednosti i slabé stránky. [59]

V srpnu 2009 se nakonec vývojáři dohodli, že tyto dva projekty spojí a výsledným produktem bude nový nástroj Selenium WebDriver, označovaný také jako Selenium 2.0. Díky spojení toho nejlepšího z výše uvedených projektů tak Selenium WebDriver poskytuje podporu pro jazyky Java, C#, Python a Ruby a internetové prohlížeče Internet Explorer, Firefox, Chrome, Opera a prohlížeče mobilních zařízení s operačními systémy Android a iOS. Dále existují ještě sesterské projekty podporující skriptovací jazyk Perl nebo PHP, obsahující implementaci pro prohlížeč BlackBerry nebo podporu pro tzv. „headless“ WebKit, který je užitečný zejména v případech, kdy mají být testy spouštěny na serveru pro kontinuální integraci bez nutnosti zobrazovat okno prohlížeče. [59]

Selenium 2 bylo poprvé publikováno 8. července 2011. [63] V roce 2013 pak byla publikována první pracovní verze specifikace WebDriver API na W3C. [64]

Původní Selenium RC projekt je nicméně stále udržován a umožňuje podporu pro prohlížeče, které nejsou nástrojem Selenium WebDriver podporovány. Vývoj nových funkcí však již nepokračuje, proto se všeobecně nedoporučuje v něm začínat psát nové testy. [59]

¹² Nativní – přirozený, vlastní. Ovládat aplikaci (v tomto případě prohlížeč) nativně znamená, že je k tomuto úkolu využít programovací jazyk, ve kterém je vytvořena – a který je tedy pro ní přirozený, vlastní. (Vlastní definice)

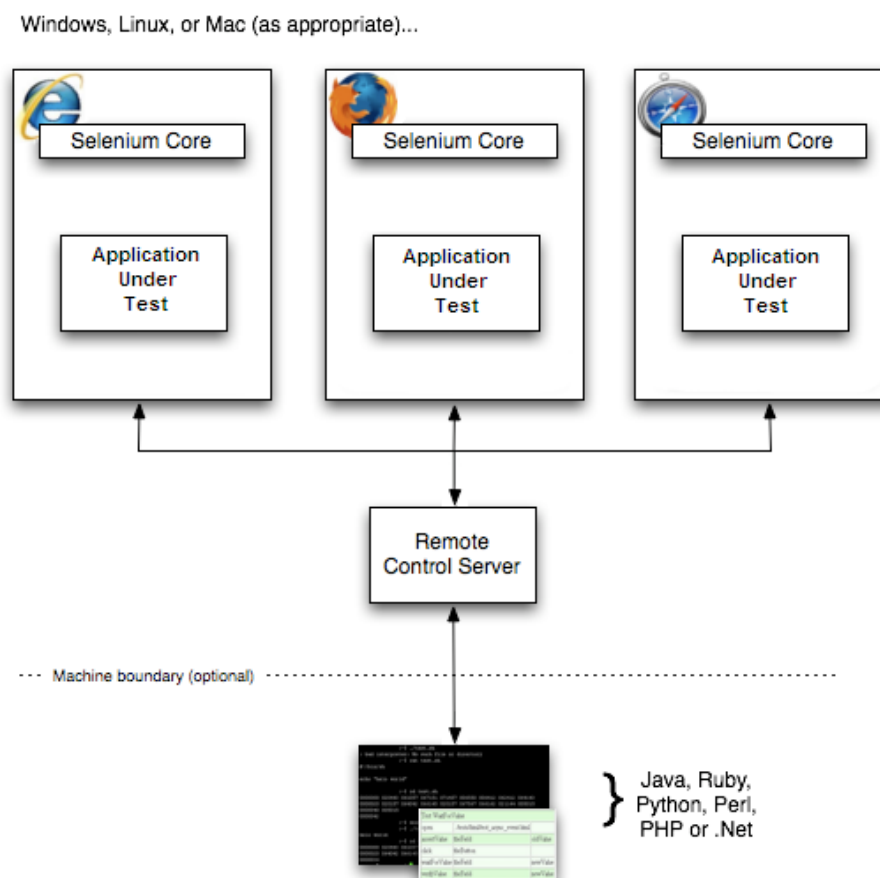
3.3.2 Selenium Remote Control

Prvním produktem z nástrojů rodiny Selenium bylo Selenium Remote Control (Selenium RC, Selenium 1), které bylo po dlouhou dobu hlavním a velice úspěšným projektem. Nicméně se potýkal s výše uvedenými nedostatky, které vedly k potřebě vytvořit zcela nový, lepší nástroj.

Selenium RC se skládá ze dvou komponent: [62]

- **Selenium RC Server** – zachycuje HTTP zprávy z testovacího skriptu, pro jehož vytváření může být použit libovolný ze sedmi podporovaných programovacích či skriptovacích jazyků (Java, JavaScript, Ruby, PHP, Python, Perl and C#). Tyto zprávy obsahují tzv. Selenese příkazy, což jsou speciální Selenium příkazy používané pro psaní automatizovaných testů. Selenium RC Server tyto příkazy interpretuje a následně spustí. Test typicky začíná spuštěním prohlížeče, do kterého je hned při spuštění automaticky vložena instance Selenium Core, prostřednictvím kterého pak Selenium RC Server testovanou aplikaci v prohlížeči ovládá.
- **Selenium Core** (Selenium knihovny) – sada JavaScriptových funkcí, které interpretují a spouští Selenese příkazy uvnitř internetového prohlížeče, a tímto způsobem pracují s testovanou aplikací.

Architektura nástroje Selenium RC je znázorněna na následujícím obrázku (viz Obrázek 2).



Obrázek 2: Architektura nástroje Selenium RC (Zdroj: [62])

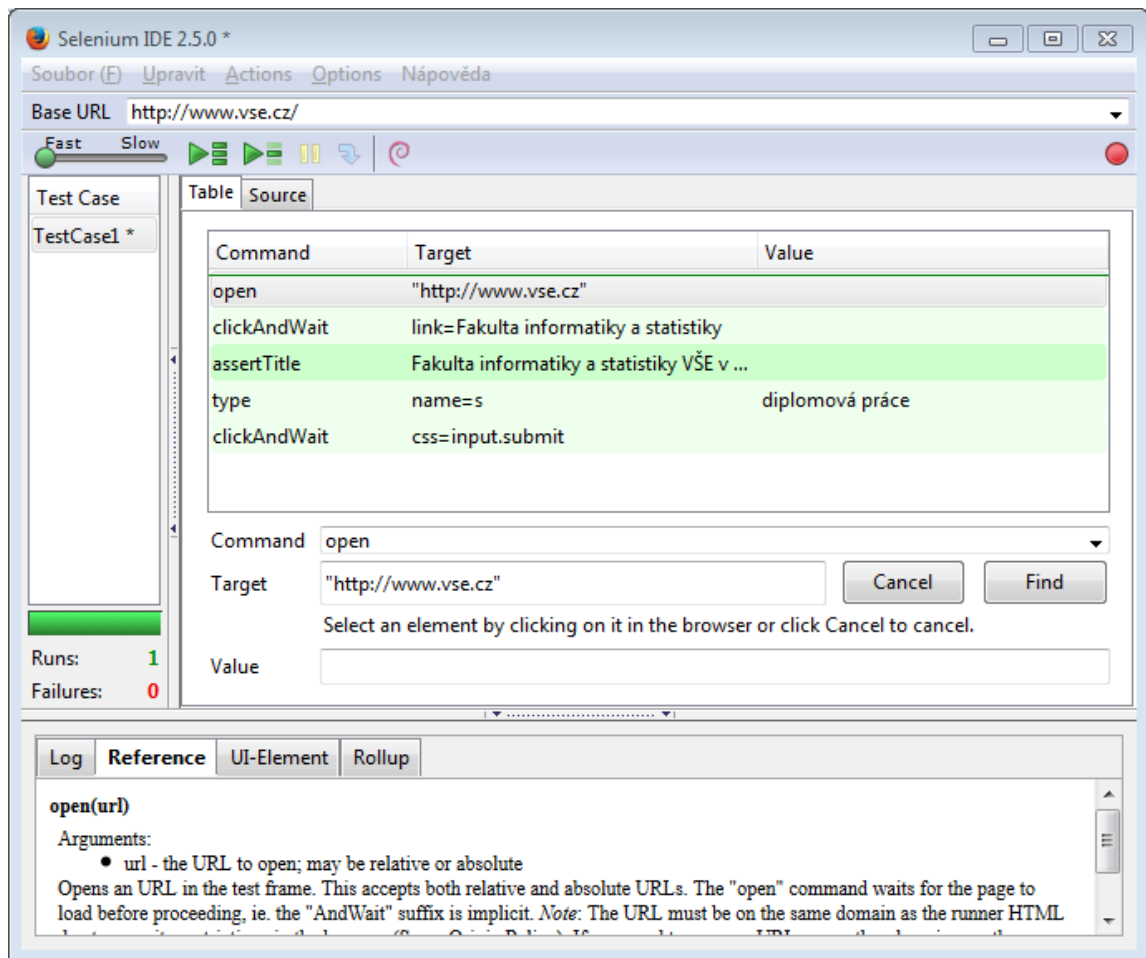
3.3.3 Selenium IDE

Selenium IDE je doplněk do internetového prohlížeče Firefox, který umožňuje nahrávat, upravovat a spouštět jednoduché automatizované testy webových aplikací. Doplněk je dostupný pouze pro prohlížeč Firefox, ostatní prohlížeče nejsou podporovány. [25 s. 3]

Selenium IDE vytvořil japonský vývojář Shinya Kasatani, jehož zaujala myšlenka nástroje JavaScript Test Runner a rozhodl se vývoj automatizovaných testů ještě více zjednodušit vytvořením nástroje s přívětivým grafickým rozhraním, který by umožňoval i začátečníkům rychle a jednoduše vytvářet testy a opakovaně je spouštět. Selenium IDE je stále ještě založeno na Selenium Core, jde tedy o JavaScriptový kód, nicméně díky geniální myšlence udělat nástroj součástí webového prohlížeče (je nainstalovaný jako doplněk) se podařilo velice elegantně obejít již zmiňovanou Same Origin Policy, která dlouho dělala vývojářům vrásky. [65]

Hlavní výhodou nástroje Selenium IDE je přívětivé uživatelské rozhraní, které umožňuje vytváření automatizovaných testů i uživatelům, kteří nemají žádné zkušenosti s programováním. Stisknutím jediného tlačítka lze začít nahrávat práci uživatele s webovou aplikací, na základě které program generuje kód, který lze poté kdykoli spustit a nechat jej pracovat namísto uživatele. Při nahrávání Selenium zaznamenává takové kroky uživatele jako otevření webové stránky na dané URL adrese, kliknutí na odkaz nebo tlačítko, vyplňování vstupních polí, otevření rozbalovací nabídky a volba některé z možností, označování zaškrtnutých políček apod. Během zaznamenávání Selenium standardně neprovádí žádné kontroly názvů nebo nadpisů stránek, tato ověřování je možné však do vygenerovaného kódu doplnit ručně nebo lze v nastavení zapnout automatické ověřování názvu (elementu *title*) každé nově otevřené webové stránky. [66]

Nahrané a případně upravené testy lze poté znovu spouštět po jednom nebo celou sadu testů najednou. Výsledky testů jsou pak zobrazeny v levém panelu – úspěšně provedené testy bez chyb jsou obarveny zelenou barvou a testy s chybami (ať už jde o chyby v návrhu testů nebo ve funkčnosti aplikace) barvou červenou.



Obrázek 3: Ukázka okna aplikace Selenium IDE

Díky své jednoduchosti však Selenium IDE není vhodné pro komplexnější testování aplikací, neboť neposkytuje takové možnosti, jaké lze využít v případě psaní automatizovaných testů v programovacím jazyku jako např. Java nebo C#. Při vytváření sady testů tedy uživatel velice brzy narazí na problémy jako nemožnost znovupoužití kódu, nekomfortní vytváření cyklů, absence polí a seznamů atp. Údržba takových testů se pak může stát bolestnou zkušeností, neboť každou změnu je nutné udělat ve všech dotčených testech zvlášť.

Kód testů je ukládán v podobě tzv. Selenium příkazů (označovány také jako Selenese), které lze následně vyexportovat do libovolného programovacího jazyka, který je podporován nástrojem Selenium WebDriver a nadále je spravovat v této podobě. [66] Ze své zkušenosti však mohu říci, že tato funkce bohužel nepracuje zcela spolehlivě a po exportu je často nutné kód dodatečně upravovat, aby pracoval správně. Pokud je tedy pravděpodobné, že bude potřeba v budoucnu přejít na robustnější Selenium WebDriver, je vhodné začít rovnou tímto nástrojem. Architektura testů se totiž u obou nástrojů liší – Selenium IDE pracuje s každým testem odděleně a nevyužívá společné části kódu, zatímco při použití nástroje Selenium WebDriver lze využívat možnosti daného programovacího jazyka a testy logicky seskupovat do tříd, využívat společné metody, dědičnost, výčtové typy, návrhové vzory atd., čímž se správa testů výrazně zjednoduší.

Z výše uvedených důvodů je tedy Selenium IDE vhodné pouze v případech, kdy pro otestování aplikace stačí jen několik málo jednoduchých testů a v průběhu času se testovaná aplikace nijak zásadně nemění.

3.3.4 Selenium Grid

Původní myšlenka dalšího nástroje – Selenium Grid – vznikla zcela mimo Selenium projekt. Postaral se o ni Patrick Lightbody, který byl svým nápadem na rozšíření funkcionality nástroje Selenium RC natolik nadšen, že dokonce opustil svou stálou práci ve firmě Jive Software a začal pracovat na projektu „Hosted QA“, jehož cílem bylo co nejvíce zkrátit čas provádění automatizovaných testů. Nástroj byl schopen pořizovat snímky obrazovky v různých stavech testované aplikace a posílat Selenium příkazy do více strojů zároveň a provádět tak testy v různých prohlížečích na různých platformách paralelně, což při spouštění několika desítek testů může představovat značnou úsporu času. [65]

Selenium Grid umožňuje spouštět testy v distribuovaném prostředí pro provádění testů (angl. distributed test execution environment). Všeobecně existují dva důvody, proč pro vytváření automatizovaných testů využít nástroj Selenium Grid: [67]

1. Je potřeba testy spouštět v různých prohlížečích, různých verzích prohlížečů a/nebo v prohlížečích běžících na různých operačních systémech.
2. Je potřeba zredukovat dobu trvání provádění celé sady testů.

Selenium Grid je v současné době distribuován v podobě .jar souboru, který v sobě obsahuje Selenium RC Server a Selenium Grid, a je označován jako Selenium Grid 2.0. [67]

3.3.5 Selenium WebDriver

Selenium WebDriver (někdy nazývaný také Selenium 2) je nástroj pro automatizaci práce s internetovým prohlížečem. V současné době je nejčastěji vnímán jako nástroj pro automatizované testování uživatelského rozhraní webových aplikací, avšak lze jej použít i pro provádění téměř jakýchkoli činností na webu, které uživatel provádí opakovaně a má smysl je automatizovat.

Selenium WebDriver je v současné době jedním z nejpoužívanějších nástrojů pro automatizované funkční testování webových aplikací. [2] K dispozici jsou knihovny pro různé programovací a skriptovací jazyky jako je Java, C#, Python nebo Ruby. Navíc existují i knihovny poskytující podporu pro další jazyky, např. Perl nebo PHP, které byly vytvořeny třetími stranami. [68]

Selenium WebDriver využívá pro simulaci práci uživatele s webovou aplikací speciální aplikační programové rozhraní¹³, které umožňuje prohlížeč nativně ovládat. Již tedy není postaven na JavaScriptu, jako tomu bylo v případě nástroje Selenium RC. [24 s. 65] WebDriver API umožňuje pracovat s HTML a JavaScript aplikacemi v internetových prohlížečích jako je např. Internet Explorer, Firefox, Google Chrome a mnoho dalších. Bohužel však neumožňuje práci s tzv. RIA technologiemi¹⁴ jako je Silverlight, Flex/Flash a JavaFX. [25 s. 3]

Detailní popis nástroje Selenium WebDriver, jeho funkcionality a možností využití je uveden v příloze této diplomové práce – viz Příloha A: Uživatelská příručka k nástroji Selenium WebDriver.

3.4 Nástroje pro zkoumání struktury webové stránky

Vedle nástrojů přímo určených pro automatizované testování aplikací se na trhu vyskytují i nástroje, které testeři využívají jen ke specifickým činnostem. Jednou z takových aktivit je určení, kde se na webové stránce nachází daný element. Pokud totiž má WebDriver provést úkon nad nějakým elementem, např. kliknout tlačítko, musí nejprve vědět, jak jej má najít. A právě tuto cestu mu musí tester poskytnout. Cestu k elementu lze samozřejmě vyhledat manuálně ve zdrojovém kódu stránky, ten je ale často nepřehledný a hledání elementu by bylo příliš zdlouhavé. Proto je vhodné si práci ulehčit využitím některého z vývojářských nástrojů pro ladění zdrojového kódu webových stránek. V následujících podkapitolách jsou nejčastěji využívané nástroje stručně představeny.

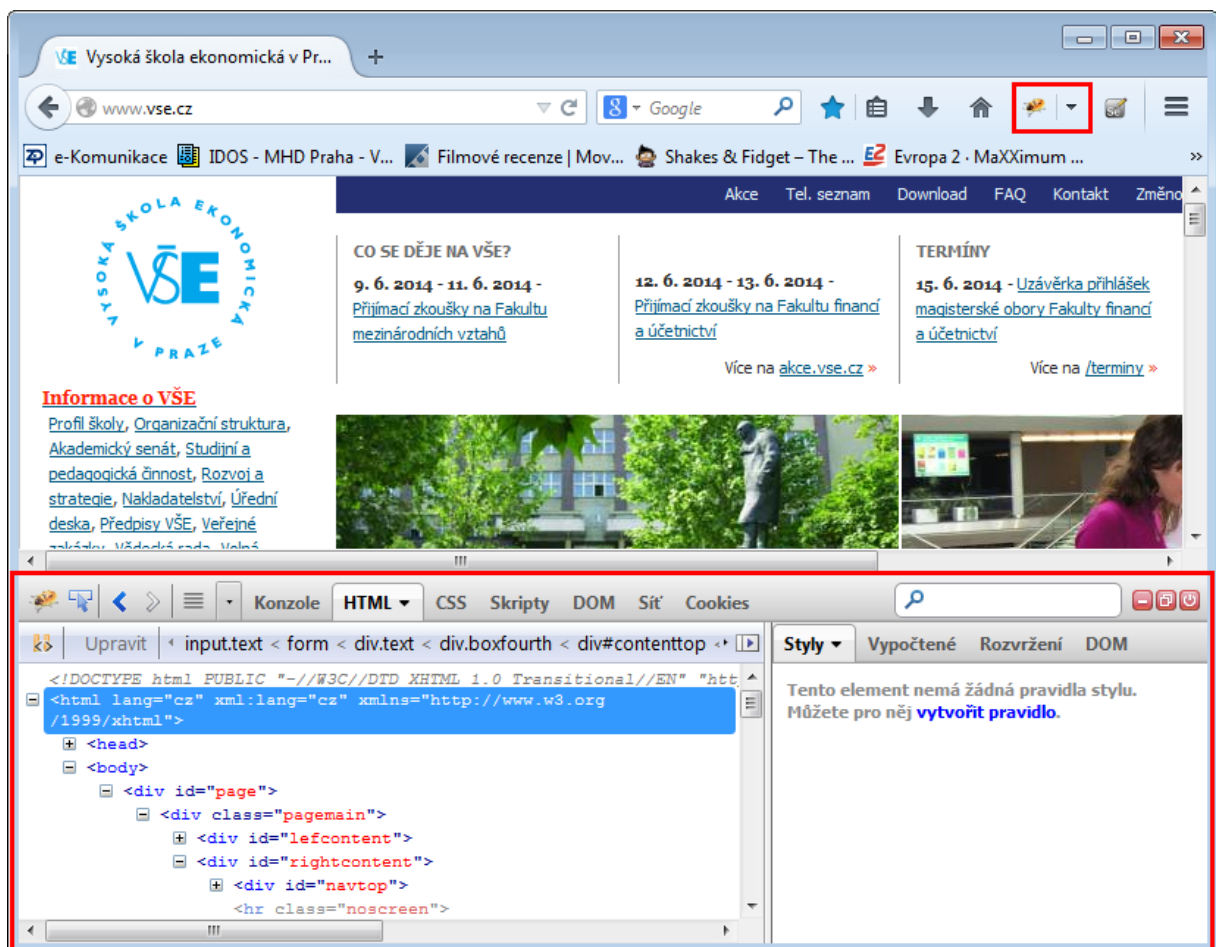
¹³ Aplikační programové rozhraní (API, Application Program Interface nebo také Application Programming Interface) – sada příkazů, funkcí a protokolů, které mohou programátoři využít při programování softwaru. [69]

¹⁴ RIA (Rich Internet Application) – moderní webové stránky, které běží ve webovém prohlížeči, ale umožňují uživateli stejný komfort práce jako běžná aplikace nainstalovaná na počítači uživatele. [70]


3.4.1 Firebug

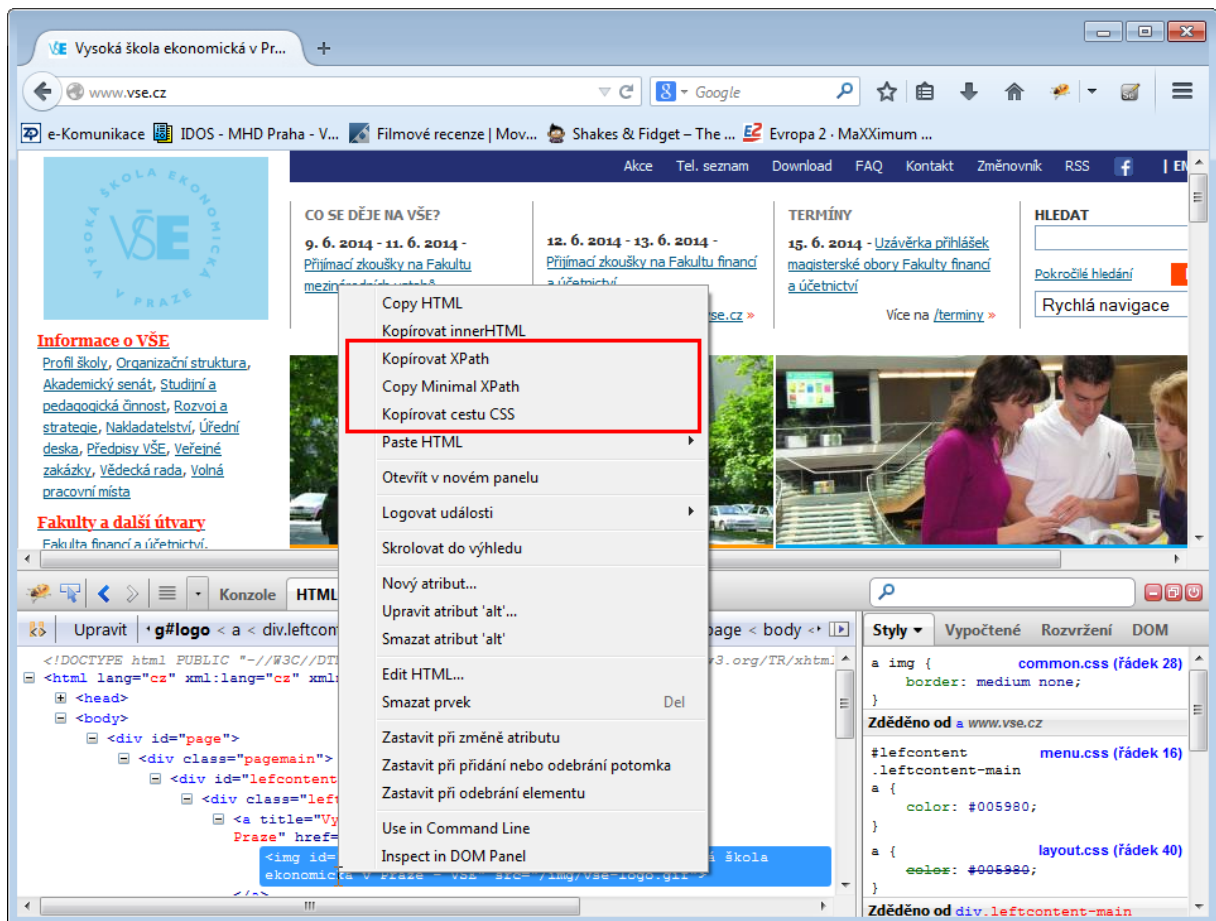
Firebug je doplněk do internetového prohlížeče Firefox (od verze 3.6 až po nejnovější [71]), který lze zdarma stáhnout z oficiálních webových stránek projektu Firebug [71]. Jde o nejoblíbenější nástroj pro ladění kódu webových stránek pro potřeby testování, neboť umožňuje snadno a rychle najít cestu k elementu mnoha různými způsoby.

Po úspěšné instalaci nástroje Firebug do prohlížeče Firefox se v pravém horním rohu zobrazí ikona brouka. Po kliknutí levým tlačítkem myši na tuto ikonu se ve spodní části obrazovky zobrazí panel aplikace.



Obrázek 4: Ikona a panel aplikace Firebug

Pro nalezení unikátního identifikátoru elementu je potřeba kliknout na tlačítko  v levém rohu panelu aplikace Firebug a poté na vybraný prvek na stránce. Část zdrojového kódu, která odpovídá zvolenému prvku, je v panelu aplikace Firebug na záložce HTML zbarvena modře. Pak již jen stačí pravým tlačítkem myši kliknout na zvýrazněnou část kódu a zvolit některou z možností *Kopírovat XPath*, *Kopírovat minimální XPath* nebo *Kopírovat cestu CSS*. Tím se do schránky uloží unikátní cesta k elementu, kterou pak lze použít v automatizovaných testech pro nalezení správného prvku na stránce.



Obrázek 5: Firebug - kopírování cesty k elementu

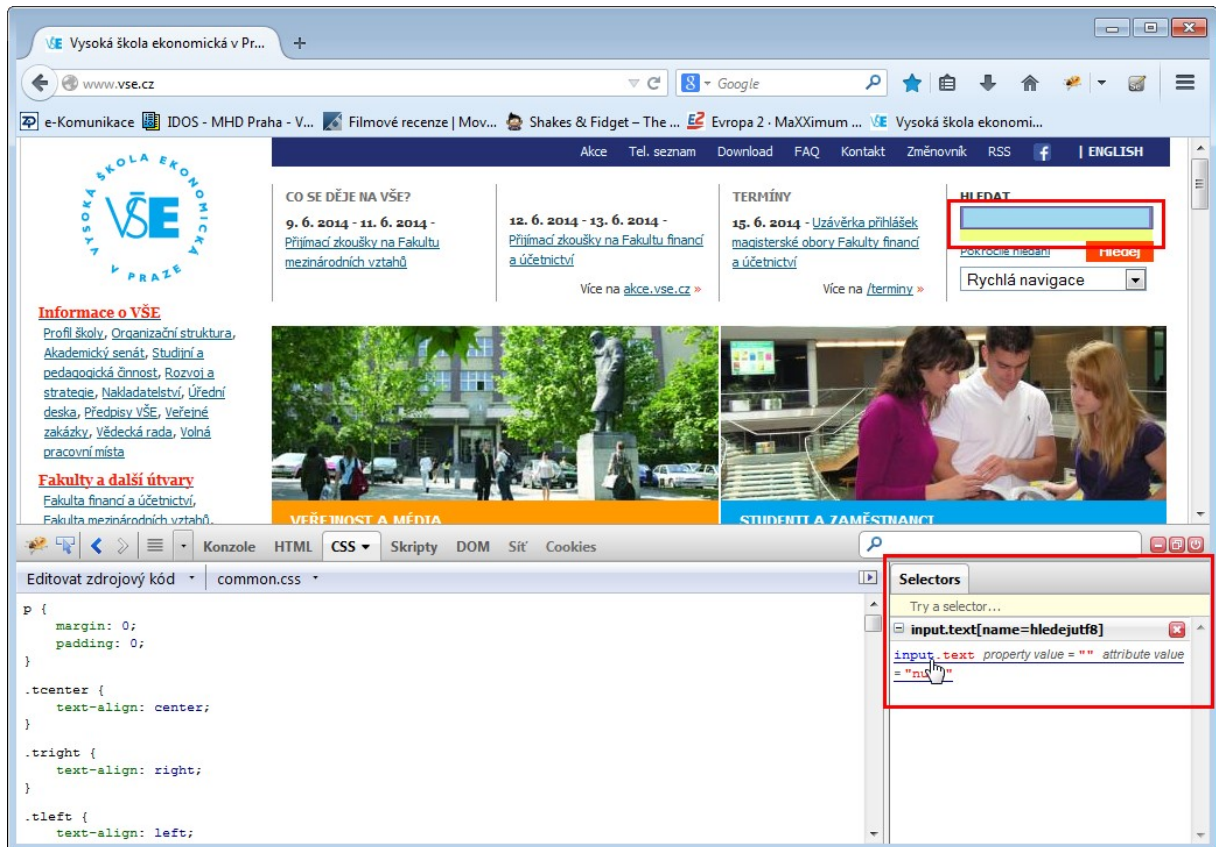
Ne vždy je však vhodné používat automaticky vygenerované cesty k elementům. V některých webových aplikacích jsou totiž atributy elementů či jejich části generovány dynamicky (např. atributy id jsou doplněny o automaticky generované číselné řady) a nelze se tak spolehnout, že bude cesta k danému elementu vždy stejná. Z toho důvodu je vhodné vytvořit cestu vlastní, která bude natolik univerzální, aby umožnila nalezení elementu, i když se jeho atributy částečně změní, a následně ověřit, zda skutečně vede ke správnému elementu. Největší předností nástroje Firebug je možnost ověřit vlastnoručně vytvořenou cestu nejen podle CSS, ale také podle XPath, což například nástroj Internet Explorer Developer Tools neumožňuje.

Ověřování CSS selektoru:

Pokud hledaný element nedisponuje atributem id, je potřeba jej identifikovat jiným způsobem. Nejčastěji je k tomuto účelu využívána identifikace pomocí CSS selektorů. Konstrukci CSS selektorů by již čtenář měl znát, proto zde nebude vysvětlována, pouze zde bude předvedeno, jak ověřovat jejich správnost.

Př.: na stránce existuje textové pole s tímto zdrojovým kódem: `<input class="text" type="text" name="hledejutf8">`. Lze jej tedy identifikovat například pomocí CSS selektoru `input.text[name=hledejutf8]`. Správnost tohoto selektoru lze ověřit v aplikaci Firebug na záložce *CSS*. Do textového pole s předvyplněným textem *Try a selector...* v pravé části panelu

aplikace stačí vyplnit výše uvedený selektor a potvrdit klávesou Enter. Pokud se aplikaci podařilo najít hledaný element nebo elementy (jednomu selektoru může odpovídat více prvků na stránce), pak odkaz na ně zobrazí pod selektorem a při umístění kurzoru nad odkaz se daný element zvýrazní modrou barvou.



Obrázek 6: Firebug - ověření CSS selektoru

Ověřování XPath selektorů

V určitých případech ale bohužel CSS selektory nestačí, zejména pokud je pro spuštění automatizovaných testů použita některá ze starších verzí prohlížeče Internet Explorer, která některé CSS selektory nepodporují (více informací viz kapitola A.3.6). Selenium WebDriver pak element nedokáže najít, i když je selektor vytvořen správně. V takovém případě je potřeba využít selektorů XPath.

Typickým příkladem je potřeba nalézt n-tý prvek daného typu na stránce – například třetí odkaz v menu. Zdrojový kód může vypadat (zjednodušeně) například takto:

Výpis 1: Zdrojový kód html stránky

```
<html>
<head>...</head>
<body>
<div class="menu">
  <p>
    <a href="/kategorie/37">Informace o VŠE</a>
  </p>
  <p>
    <a href="/kategorie/3393">Profil školy</a>
    <a href="/obecne/struktura.php">Organizační struktura</a>
    <a href="http://as.vse.cz/">Akademický senát</a>
  </p>
  ...
</div>
...
</body>
</html>
```

Odkaz *Akademický senát* lze pak nalézt například pomocí XPath selektoru `//div[@class='menu']/p[2]/a[3]`. (Podrobnosti vytváření XPath selektorů jsou součástí přílohy A, konkrétně kapitoly A.3.7 Vyhledávání elementů pomocí XPath).

Tento selektor lze ověřit v panelu aplikace Firebug na záložce *Skripty* v poli *Sledování proměnné*. XPath selektory je potřeba zadávat ve tvaru `$x("xpath výraz")`, v tomto případě tedy `$x("//div[@class='menu']/p[2]/a[3]")` a potvrdit stisknutím klávesy Enter. V případě úspěšného nalezení elementu (či elementů) je odkaz na něj zobrazen vpravo od selektoru a při umístění kurzoru myši nad odkaz se daný element zvýrazní modrou barvou.

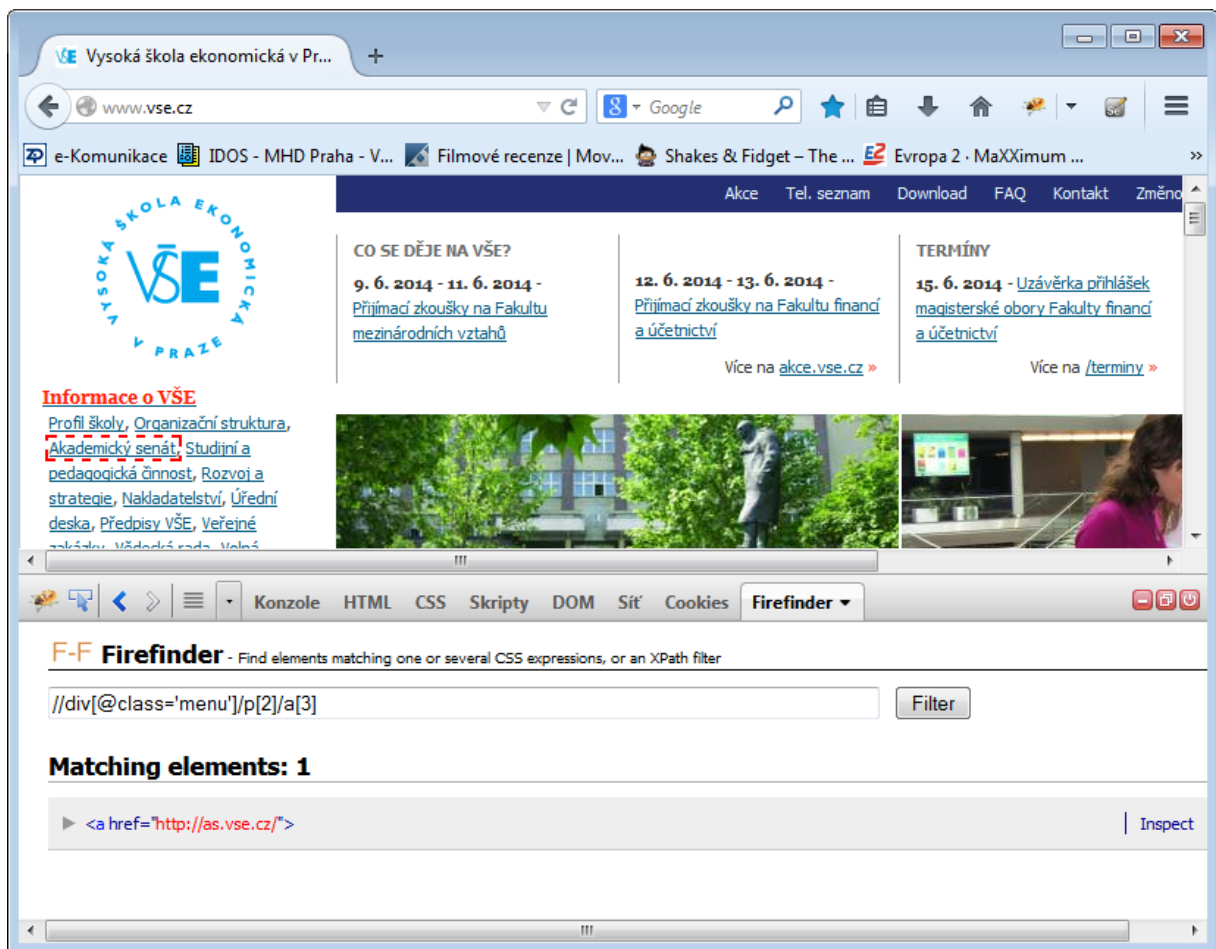


Obrázek 7: Firebug - ověření XPath selektoru

3.4.2 Firefinder

Výše uvedený nástroj Firebug lze dále doplnit o doplněk Firefinder, který umožňuje ověřovat selektory ještě komfortněji. Doplněk je možné stáhnout na webové stránce Doplnky aplikace Firefox [72] a k jeho úspěšnému spuštění je potřeba mít nejprve nainstalovaný Firebug.

Firefinder představuje další záložku v panelu aplikace Firebug s jednoduchou funkcionalitou – obsahuje pouze jediné vstupní textové pole, kam lze jednoduše vložit vytvořený CSS nebo XPath selektor a potvrdit stisknutím tlačítka *Filter*. Nalezené elementy jsou na webové stránce ohraničeny červenou přerušovanou čarou a jejich seznam lze vidět v panelu aplikace Firebug v podobě HTML kódu daného elementu. Při umístění kurzoru myši nad vybraný prvek v seznamu nalezených elementů se ohraničení daného prvku na stránce změní z červené barvy na modrou.

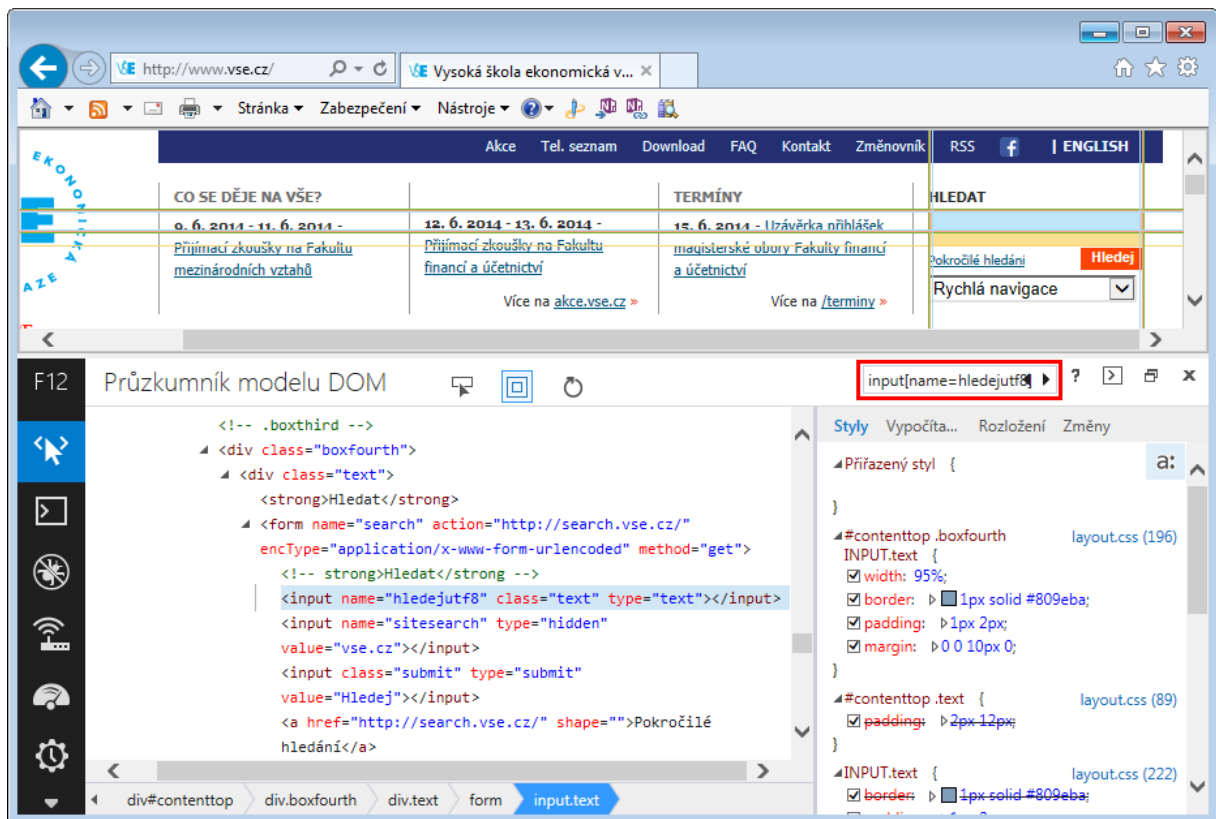


Obrázek 8: Záložka doplňku Firefinder v panelu aplikace Firebug a nalezený element

3.4.3 Internet Explorer Developer Tools

Internet Explorer Developer Tools je nástroj, který je součástí prohlížeče Internet Explorer od verze 8 až po nejnovější. [73] Pro starší verze prohlížeče jej lze stáhnout na internetových stránkách společnosti Microsoft [74]. Nástroj se spouští stisknutím klávesy F12 nebo pod volbou *Nástroje > Vývojářské nástroje F12*.

Manuálně vytvořený CSS selektor lze ověřit pomocí vyhledávacího pole umístěného v pravém horním rohu panelu nástroje Internet Explorer Developer Tools. Po vložení selektoru je potřeba volbu potvrdit stisknutím klávesy Enter.



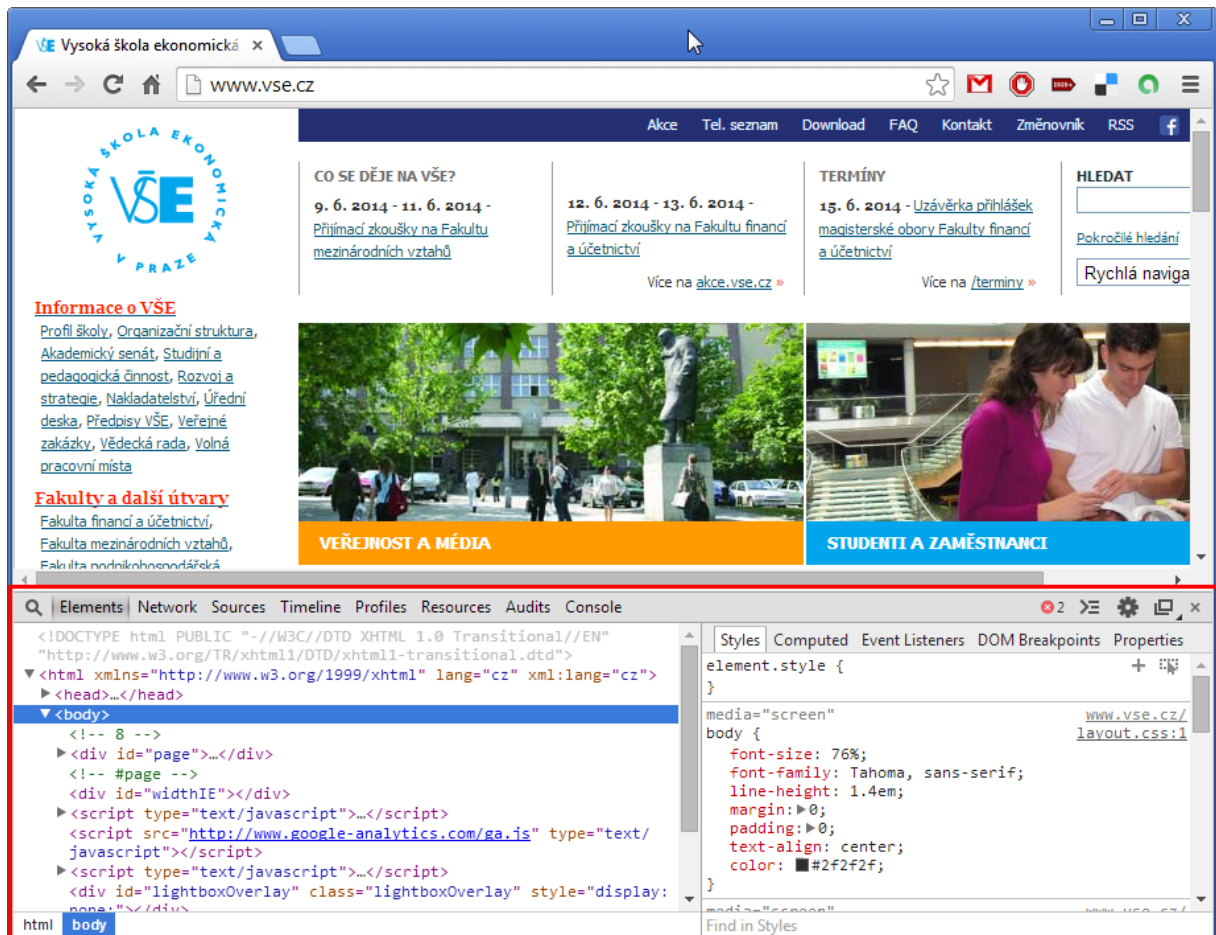
Obrázek 9: Internet Explorer Developer Tools - ověřování CSS selektoru

Práce s ním je však poměrně nekomfortní díky velice omezené velikosti vyhledávacího pole, které navíc zčásti zakrývají šipky pro pohyb mezi nalezenými elementy.


Další nevýhodou je, že na rozdíl od výše uvedených nástrojů neumožňuje ověřovat XPath selektory.

3.4.4 Chrome Developer Tools

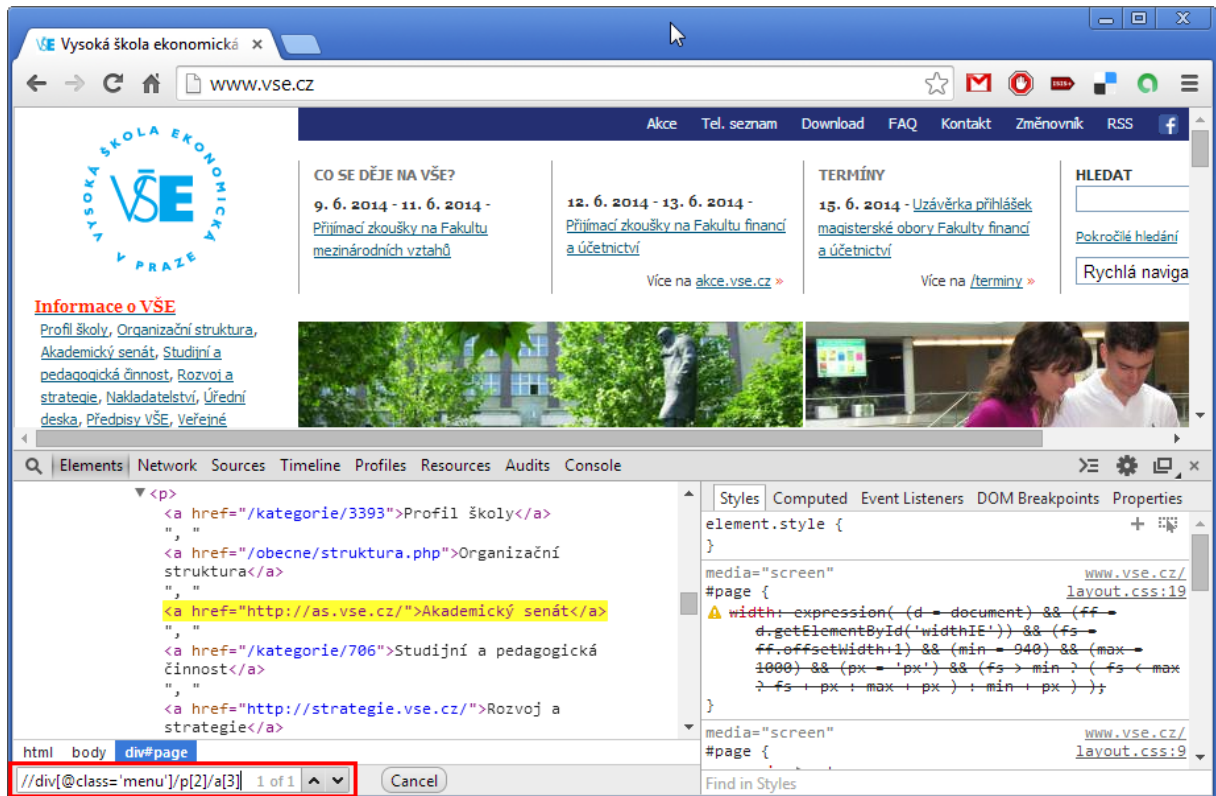
Další z nástrojů pro vývojáře, který umožňuje nalézt a ověřit cestu k elementu na stránce, je Chrome Developer Tools (zkráceně DevTools). Jde o nástroj vestavěný do prohlížeče Google Chrome. Spustit jej lze přes *Menu > Nástroje > Nástroje pro vývojáře* anebo kliknutím pravého tlačítka myši na libovolný prvek na stránce a volbou možnosti *Zkontrolovat prvek*.



Obrázek 10: Panel aplikace Chrome Developer Tools

Nalezení cesty k elementu je téměř stejné jako v nástroji Firebug – kliknutím na tlačítko  umístěné v levém horním rohu panelu aplikace Chrome Developer Tools a následně kliknutím na vybraný prvek na stránce. Jemu odpovídající část zdrojového kódu se zbarví modře a poté je možné po kliknutí pravým tlačítkem myši na tuto část kódu vybrat možnost *Copy CSS Path* nebo *Copy XPath*, čímž se automaticky vygenerovaná cesta zkopíruje do schránky a opět je možné ji použít ve zdrojovém kódu automatizovaného testu.

Ověřovat manuálně vytvořené selektory pak lze pomocí vyhledávacího pole – nejprve je potřeba kliknout levým tlačítkem myši kamkoli do zdrojového kódu stránky a poté stisknout kombinaci kláves Ctrl+F pro zobrazení vyhledávacího pole. Do něj pak lze vepsat CSS nebo XPath selektor, který se v reálném čase vyhodnocuje, a je tedy možné průběžně kontrolovat jeho správnost.



Obrázek 11: Google Chrome Developer Tools - ověření XPath selektoru

Poznámka na závěr:

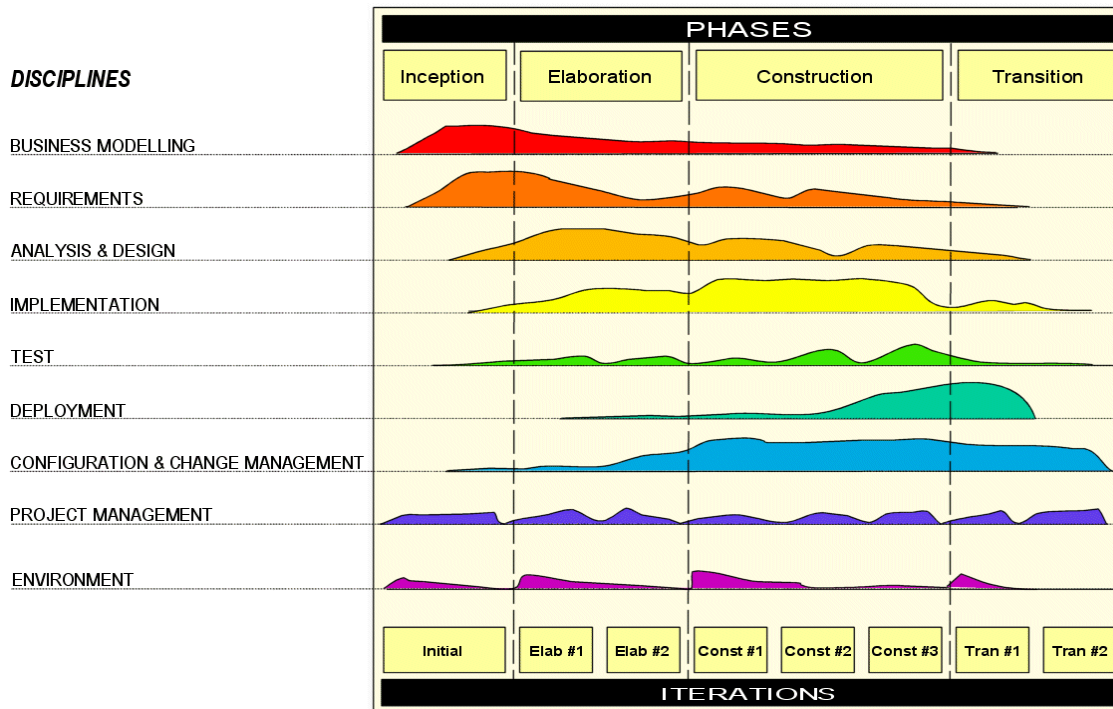
Selektory elementů není potřeba ověřovat v každém prohlížeči zvlášť, k jejich nalézání a ověřování lze využít libovolný z výše uvedených nástrojů, neboť selektory elementů by měly ve všech internetových prohlížečích fungovat shodně. Jediné, na co je potřeba si dát pozor, je kompatibilita dané verze prohlížeče s některými CSS selektory (více viz kapitola A.3.6 Vyhledávání elementů pomocí CSS).

4 Metodika pro automatizaci testování webových aplikací

Jedním z hlavních cílů této diplomové práce je vytvoření metodiky pro automatizaci testování webových aplikací, která je obsahem této kapitoly. Ještě před představením samotné metodiky je však vhodné tento pojem vyjasnit. Metodika představuje souhrn metod a postupů pro realizaci určitého úkolu, přičemž v kontextu IS/ICT jde o metody a postupy pro vývoj i provoz IS/ICT s tím, že nemusí jít nutně jen o vývoj nového softwaru, ale i o implementaci hotových řešení, integraci komponent a služeb. [75 s. 17] Metodika budování IS/ICT definuje principy, procesy, praktiky, role, techniky, nástroje a produkty používané při vývoji, údržbě a provozu informačního systému, a to jak z hlediska softwarově inženýrského, tak z hlediska řízení. [76 s. 13] Metodik budování IS/ICT existuje několik a dělí se na dvě hlavní kategorie – agilní a rigorózní. Zatímco agilní metodiky nalézají své uplatnění hlavně v projektech, kde je řešení vytvářeno velice rychle a stejně rychle se mění i požadavky zákazníků, metodiky rigorózní jsou vhodné pro projekty většího rozsahu, s předem definovanými požadavky a malou pravděpodobností jejich změn. [75 s. 65]

A proč vlastně metodiku využívat? Dle výzkumné agentury Standish Group se stále celá řada projektů potýká s velice nízkou úspěšností – v roce 2012 bylo včas, v rámci rozpočtu a se všemi požadovanými vlastnostmi a funkcemi dodáno pouze 39 % projektů. 43 % projektů pak bylo dodáno později, přesáhlo rozpočet a/nebo nenaplnilo všechny požadavky na funkcionalitu systému. A nakonec 18 % projektů skončilo zcela neúspěšně – buď byly zastaveny ještě před plánovaným dokončením, nebo jejich výsledek nebyl nikdy použit. [77] „*Příčin této skutečnosti je celá řada. Patří mezi ně zejména zvláštnosti softwaru jako produktu a specifika jeho vývoje, rychlý vývoj informačních technologií, nedocení úlohy lidí při budování informačního systému, nejasné požadavky, změny v průběhu vývoje a řada dalších.*“ [76 s. 11] Řešením pak může být využití vhodné metodiky budování IS/ICT, která by napomohla lépe zorganizovat činnosti vykonávané v rámci daného projektu, definovala role a odpovědnosti a poskytla nejlepší praktiky (angl. best practices) pro řešení problémů.

Jednou z nejznámějších metodik budování informačních systémů je **RUP** (Rational Unified Process), která byla původně zařazována mezi metodiky rigorózní, avšak od roku 2003 je doplňována i o agilní praktiky, díky čemuž jde v současné době o univerzální metodický rámec, v jehož rámci lze vytvořit různé metodiky pro různé projekty od rigorózních až po agilní. [78 s. 112] RUP dělí proces vývoje softwaru do 4 fází – počáteční (Inception), rozpracování (Elaboration), konstrukce (Construction) a nasazení (Transition), přičemž v každé z nich je zastoupeno v různé míře 9 disciplín – Business Modelling, Requirements, Analysis & Design, Implementation, Test, Deployment, Configuration & Change Management, Project Management a Environment. Jak je zřejmé z následujícího obrázku (viz Obrázek 12), i zde má testování své místo již od první fáze až po finální. [79]



Obrázek 12: Fáze a disciplíny RUP (Zdroj: [79])

Další významnou metodikou je **OpenUP** (Open Unified Process), která patří mezi metodiky agilní. I ta rozděluje proces vývoje softwaru do čtyř fází, které vycházejí z RUP, a testování přitom zaujímá svou roli ve třech z nich – rozpracování (Elaboration), konstrukce (Construction) a nasazení (Transition) – viz Obrázek 13. [80]

Iteration template patterns	Phase objectives
<ul style="list-style-type: none"> 🌀 Inception Phase Iteration <ul style="list-style-type: none"> 📁 Initiate Project 📁 Plan and Manage Iteration 📁 Identify and Refine Requirements 📁 Agree on Technical Approach 	<ul style="list-style-type: none"> ▪ Understand what to build ▪ Identify key system functionality ▪ Determine at least one possible solution ▪ Understand the cost, schedule and risks associated with the project
<ul style="list-style-type: none"> 🌀 Elaboration Phase Iteration <ul style="list-style-type: none"> 📁 Plan and Manage Iteration 📁 Identify and Refine Requirements 📁 Define the Architecture 📁 Develop Solution Increment 📁 Test Solution 📁 Ongoing Tasks 	<ul style="list-style-type: none"> ▪ Get a more detailed understanding of the requirements ▪ Design, implement, validate, and baseline an Architecture ▪ Mitigate essential risks, and produce accurate schedule and cost estimates
<ul style="list-style-type: none"> 🌀 Construction Phase Iteration <ul style="list-style-type: none"> 📁 Plan and Manage Iteration 📁 Identify and Refine Requirements 📁 Develop Solution Increment 📁 Test Solution 📁 Ongoing Tasks 	<ul style="list-style-type: none"> ▪ Iteratively develop a complete product that is ready to transition to its user community ▪ Minimize development costs and achieve some degree of parallelism
<ul style="list-style-type: none"> 🌀 Transition Phase Iteration <ul style="list-style-type: none"> 📁 Plan and Manage Iteration 📁 Develop Solution Increment 📁 Test Solution 📁 Ongoing Tasks 	<ul style="list-style-type: none"> ▪ Beta test to validate that user expectations are met ▪ Achieve stakeholder concurrence that deployment is complete

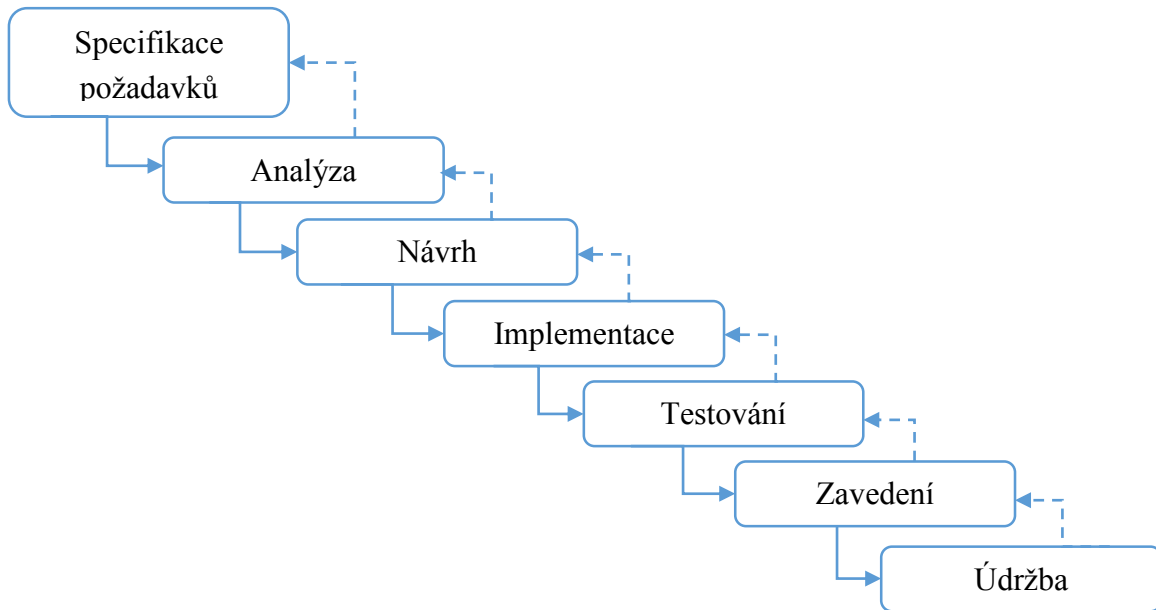
Obrázek 13: OpenUP – přehled fází projektu a jejich cílů (Zdroj: [80])

V posledních několika letech také roste popularita agilní metodiky **Scrum**, která je založena na vývoji v iteracích nazývaných Sprint, přičemž každý sprint je zaměřen na dodání minimálního produktu, který má pro uživatele přidanou hodnotu. Nedílnou součástí každého sprintu je přitom testování, neboť Scrum klade důraz na to, aby byl na konci každého sprintu zákazníkovi dodán fungující software, který bude moci používat a poskytnout dodavateli zpětnou vazbu. [81]

Mezi další agilní metodiky, které zdůrazňují význam průběžného testování, patří i **Extrémní programování**. Dle tohoto přístupu musí být programátoři neustále v kontaktu se zákazníky a pomocí testování získávat zpětnou vazbu na vyvíjený software již od prvního dne práce na projektu. I zde má tedy testování svou důležitou úlohu. [82]

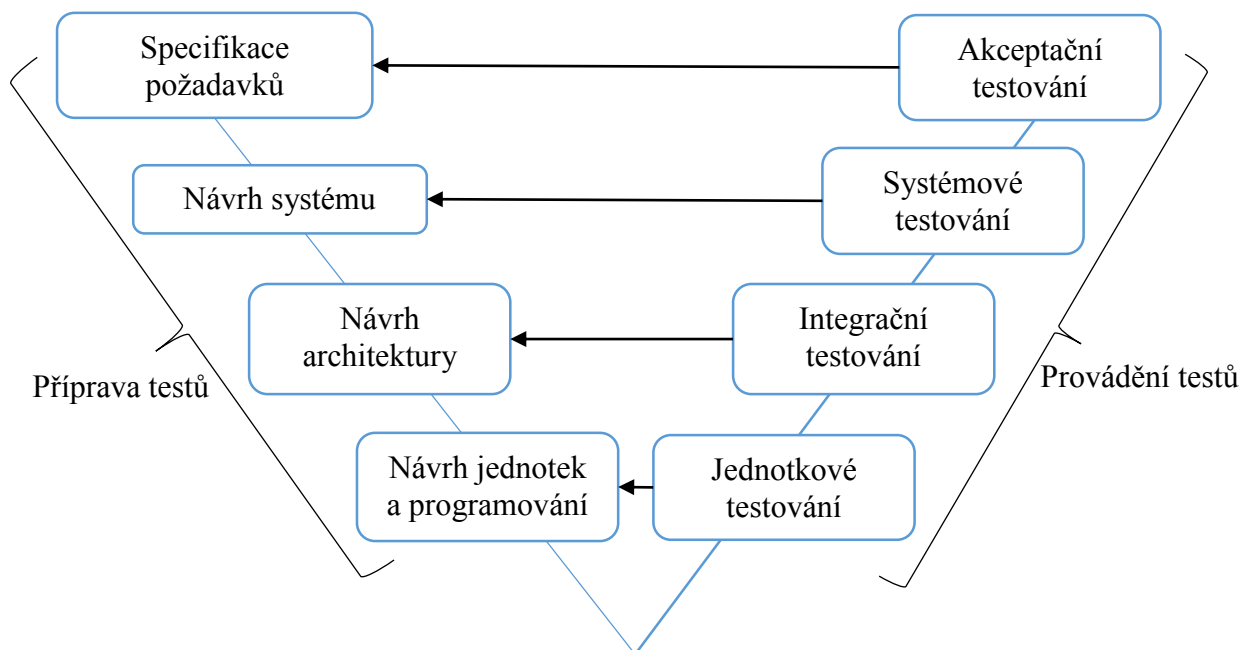
Jak tedy dokazují příklady výše uvedených metodik, testeři by měli být zapojeni v celém průběhu projektu vývoje softwaru již od jeho raného počátku a dohlížet na to, aby byly požadavky uživatele správně interpretovány a celý projekt směřoval k jejich naplnění. Jedině tak lze dosáhnout vytvoření kvalitního produktu, se kterým bude uživatel spokojen. Metodika vytvořená v rámci této diplomové práce na tento fakt bere ohled a začleňuje testovací aktivity v průběhu celého vývoje softwarového produktu.

Co se týče předmětu testování, v procesu vývoje softwaru vznikají různé artefakty, které mohou být otestovány, nemusí to být nutně jen finální produkt (software). Podle toho, jaké produkty jsou v různých fázích vývoje softwaru zkoumány a hodnoceny, lze rozlišovat několik úrovní testování. Tyto úrovně testování zachycuje tzv. **V-model**, který vychází z tradičního vodopádového modelu vývoje softwaru a snaží se odstranit jeho hlavní nedostatky. Vodopádový model vývoje softwaru totiž definuje fáze vývoje a pořadí, v jakém mají být prováděny, přičemž neumožňuje návrat o několik úrovní zpět (povolený je návrat pouze o jednu úroveň). To pak v praxi znamená, že pokud je ve fázi implementace nalezena chyba ve specifikaci požadavků, pak je nutné celý cyklus projít znovu, což může znamenat nemalé dodatečné náklady. [6 s. 26–27]



Obrázek 14: Role testování ve vodopádovém modelu vývoje softwaru (Zdroj: [75 s. 48], překresleno autorkou)

V-model je modifikací vodopádového modelu vývoje softwaru, který se snaží výše uvedeným dodatečným nákladům předejít tak, že testovací aktivity začleňuje v průběhu celého projektu vývoje softwaru. Výstupy všech fází vývoje jsou průběžně kontrolovány a vyhodnocovány, díky čemuž mohou být chyby odhaleny včas a s nižšími náklady. [6 s. 26–27]



Obrázek 15: V-model (Zdroj: [19 s. 7], přeloženo a překresleno autorkou)

V-model mimo jiné zachycuje, které činnosti mají být ve které fázi vývoje softwaru prováděny. Dokud totiž není vytvořen první funkční prototyp vyvíjené aplikace, není pochopitelně ještě možné ji otestovat. Jak je ale znázorněno na obrázku výše (Obrázek 15), již během specifikace požadavků, analýzy systému, návrhu systému a programování je možné, ba dokonce velice žádoucí, začít s přípravou testů a testovacích dat. Už v úvodní fázi projektu totiž tester může odhalit celou řadu chyb ve specifikaci požadavků, které by se v již naimplementovaném produktu odstraňovaly mnohem obtížněji. [19 s. 6]

Úrovně testování dle V-modelu:

1. **Jednotkové testování** (angl. Unit Testing) – provádí nejčastěji sám programátor, neboť tato činnost vyžaduje dobrou znalost programování. Cílem je otestovat funkcionalitu každé nejmenší stavební jednotky programu (může to být metoda, třída, procedura či funkce) samostatně, v izolaci od ostatních. Jednotkové testování je vždy automatizované, neboť jsou obvykle velice rozsáhlé a je potřeba je spouštět velice často (ideálně po každém sestavení projektu). [6 s. 61–62]
2. **Integrační testování** (angl. Integration Testing) – každá aplikace je složena z více jednotek, jejichž funkčnost byla v rámci jednotkového testování prověřena. Bohužel ale funkčnost individuálních jednotek nezaručuje funkčnost celého systému. Proto je potřeba otestovat i spojení a interakce mezi jednotlivými moduly. Integrační testování může být automatizované nebo manuální a může být prováděno testerem nebo programátorem, vždy ale vyžaduje rozsáhlejší technické znalosti než další úrovně testování. [6 s. 63–64]
3. **Systémové testování** (angl. System Testing) – klíčová fáze testování, při níž se zkoumá, zda software splňuje požadavky specifikované zákazníkem. Důležitost systémového testování spočívá zejména v tom, že jde o poslední testování před představením produktu zákazníkovi, a je tedy potřeba odhalit a zajistit opravu všech závažných chyb dříve, než je zaznamená zákazník. Naneštěstí však není možné beze zbytku otestovat každou část funkcionality softwaru, neboť existuje příliš mnoho variant vstupů a výstupů nebo cest, kterými lze aplikaci projít. Jediným řešením je správně navrhnout testovací případy a scénáře, které budou pokrývat všechny specifikované požadavky uživatele, a následně podle nich aplikaci několikrát otestovat. Do kategorie systémových testů spadají testy funkcionální i nefunkcionální (výkonnostní testování, testování použitelnosti, udržitelnosti, spolehlivosti, přenositelnosti atd.). [6 s. 65–66] [1 s. 34]
4. **Akceptační testování** (angl. User Acceptance Testing, zkr. UAT) – poslední fáze testování, jejím cíle, je zjistit, zda aplikace splňuje tzv. akceptační kritéria, což jsou zákazníkem definované měřitelné a ověřitelné podmínky pro přijetí produktu. Tato kritéria jsou samozřejmě stanovená předem a jsou obvykle součástí specifikace požadavků. V případě, že některé z akceptačních kritérií není splněno a zákazník není spokojen, může zákazník produkt odmítnout, což pro dodavatele znamená nemalý

problém. Z toho důvodu je nutné, aby bylo systémové testování provedeno důkladně a podle specifikovaných požadavků. Součástí akceptačního testování je testování nejen samotného softwaru, ale i všech doprovodných služeb a artefaktů – uživatelského manuálu, dokumentace, školení (pokud je zajišťováno dodavatelem v rámci projektu) atd. [6 s. 67]

Zde je tedy vidět nejen opět důležitost zapojení testovacích aktivit od úvodních fází projektu vývoje softwaru, ale také základní rozložení testovacích aktivit – od přípravy požadavků až do fáze implementace softwaru je to příprava testů a od chvíle, kdy je k dispozici první testovatelná verze aplikace, už se rozbíhá fáze provádění testů. I tento fakt je tedy potřeba ve vytvářené metodice zohlednit. [19 s. 6]

Metodika vytvořená v rámci této diplomové práce je zaměřena na automatizované testování webových aplikací. Vychází především z metodiky ATLM (Automated Testing Life-Cycle Methodology) prezentované v knize *Automated Software Testing: Introduction, Management, and Performance* od autorů Elfriede Dustin, Jeff Rashka a John Paul z roku 1999. [18] Informace získané z této knihy jsou aktualizovány na dnešní poměry, doplněny o poznatky z jiných metodik, například RUP, a obohaceny o osobní zkušenosti autorky.

Podle kategorizace metodik uvedené v knize *Metodiky vývoje a údržby informačních systémů: kategorizace, agilní metodiky, vzory pro návrh metodiky*, od Ing. Aleny Buchalcevé, Ph.D., z roku 2005, [76 s. 20–27] lze metodiku vytvářenou v rámci této diplomové práce charakterizovat následujícím způsobem:

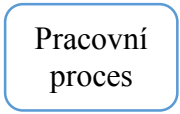



- **Kritérium zaměření metodiky:** jedná se o metodiku spíše projektovou, neboť se zaměřuje především na zavedení automatizace testování na konkrétním projektu. Přesto ale bere ohled i na kontext celé organizace a její potřeby.
- **Kritérium rozsah metodiky:** metodika pokrývá oblast testování v rámci téměř celého životního cyklu vývoje softwaru – od fáze globální analýzy a návrhu aplikace až po její zavedení. Další dimenzi této metodiky představují role – do procesu je zapojen projektový manažer, test manažer, tester, vývojář a také zástupce zákazníka.
- **Kritérium váha metodiky:** tuto metodiku bych zařadila mezi metodiky těžké, neboť se poměrně podrobně zabývá definicí procesů a popisem prováděných činností a artefaktů, které jsou v jejich průběhu využívány nebo naopak produkovány.
- **Kritérium typ řešení:** metodika se zaměřuje na testování v rámci vývoje nového řešení.
- **Kritérium doména:** metodika není nijak omezena na specifickou oblast byznysu, pro kterou by měla být aplikace určena. Lze ji aplikovat na testování široké škály webových aplikací.

Jak již bylo zmíněno výše, tato metodika klade důraz na začlenění testovacích aktivit v průběhu celého procesu vývoje softwaru, podobně jako celá řada jiných moderních metodik. Popisuje tedy, jaké činnosti testovací tým provádí v jakých fázích vývoje softwaru, definuje, které role

se těchto aktivit účastní a uvádí artefakty, které do jednotlivých aktivit vstupují či jsou v jejich průběhu produkovány.

Pro přehlednost jsou prvky metodiky nejen popsány, ale také znázorněny v grafické podobě. K tomuto účelu byla využita sada grafických komponent dle diplomové práce Marcely Šplíchalové z roku 2008 [83 s. 14].

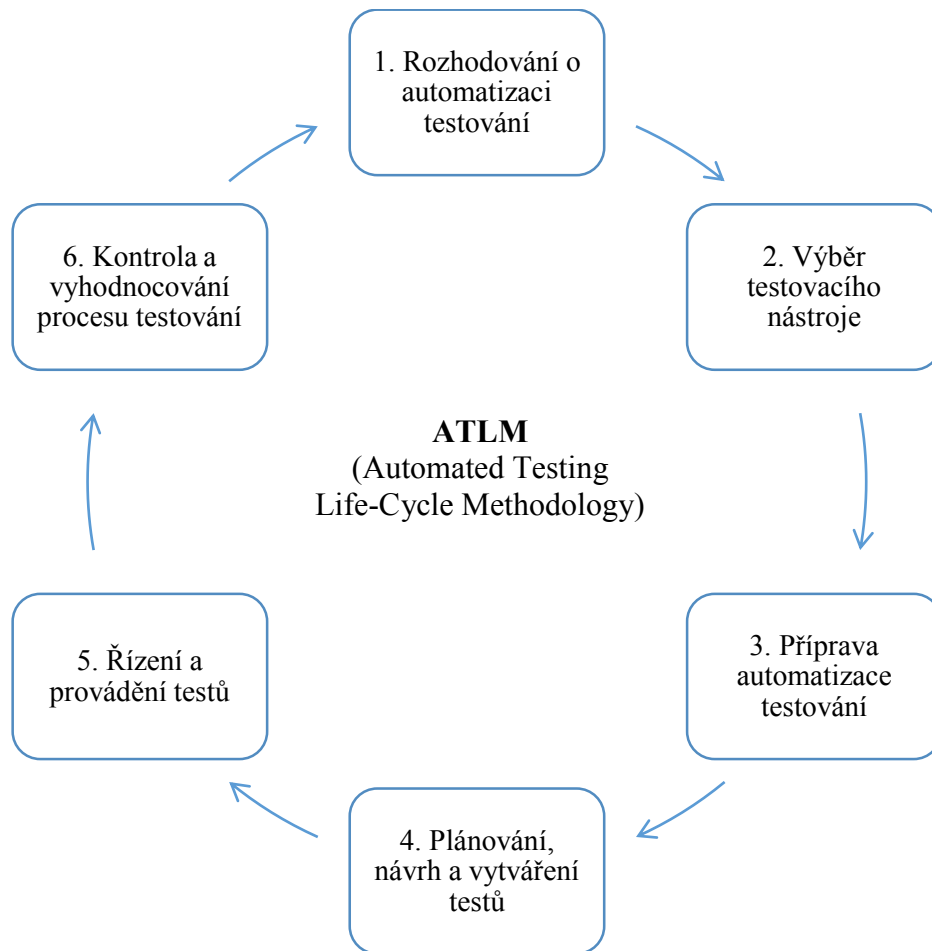
Tabulka 2: Sada grafických prvků pro popis metodiky (Zdroj: [83 s. 14], překresleno a upraveno autorkou)

Grafický prvek	Název prvku	Popis prvku
 Pracovní proces	Pracovní proces	Pracovní proces zastřešující sekvenci několika aktivit vykonávaných jednou či více osobami s definovanými rolmi, přičemž jsou využívány a/nebo produkovány určité artefakty.
 Role	Role	Formální role osoby vykonávající nějakou činnost v rámci procesu.
 Aktivita	Aktivita	Aktivita vykonávaná vybranou rolí v rámci procesu, při níž jsou využívány a/nebo produkovány vybrané artefakty.
 Artefakt	Artefakt	Vstupní či výstupní produkt (dokument, program, testovací scénář...) aktivity.

Vzhledem k tomu, že se metodika zaměřuje na testování webových aplikací, jde vždy o testování šedé skříňky – tester nezkoumá kompletní zdrojový kód testované aplikace, nicméně má přístup ke struktuře webové stránky (HTML, CSS a JavaScript) a tuto znalost využívá při vytváření automatizovaných testů.

4.1 Proces automatizace testování

Proces automatizace testování vyjadřuje nejlépe metodika ATLM: [18 s. 9]

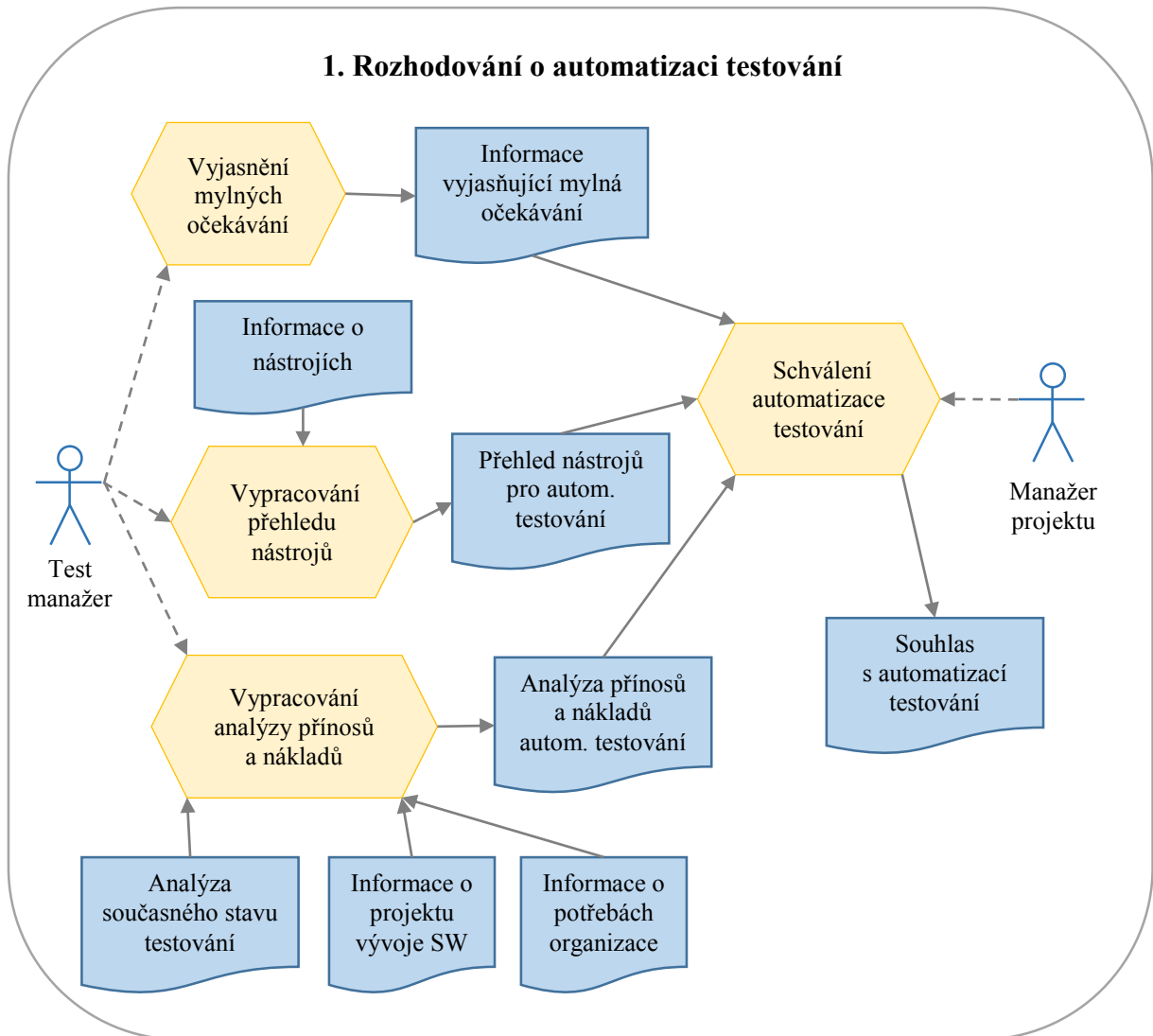


Obrázek 16: Workflow automatizace testování dle metodiky ATLM (Zdroj: [16 s. 9], překresleno a přeloženo autorkou)

Metodika ATLM popisuje 6 procesů (fází), které je pro úspěšné dokončení celého projektu automatizace testování potřeba provést. V následujících podkapitolách jsou všechny tyto fáze detailně rozebrány.

4.1.1 Rozhodování o automatizaci testování

Automatizace testování musí vždy začít procesem, který systematicky směřuje k rozhodnutí, zda automatizovat či nikoli. V této fázi je potřeba si ujasnit, co se od automatizace testování očekává a zda jsou tato očekávání reálná. Dále by se měly vyhodnotit potenciální přínosy, které by správně provedená implementace automatizace testování měla přinést, a porovnat je s očekávanými náklady. Na základě těchto předpokladů se pak lze rozhodnout, zda má automatizace smysl či nikoli. [18 s. 10]



Obrázek 17: Detail procesu rozhodování o automatizaci testování (Zdroj: autorka, inspirováno [18 s. 31])

Prvním důležitým úkolem test manažera v okamžiku, kdy je informován o možnosti zavedení automatizace testování na daném projektu, je informovat projektového manažera o tom, co může od automatizace testování očekávat a co nikoli. Často je přitom potřeba vyvrátit několik mylných představ, se kterými manažeři přicházejí, dle knihy *Automated software testing: introduction, management, and performance* [18 s. 32–37] mezi nejčastější omyly patří:

1. Představa: nástroj pro automatizaci testování umožňuje automaticky vygenerovat plán testování, návrh testů a následně i kód samotných testů.
 - Skutečnost: na trhu doposud neexistuje nástroj, který by dokázal sám, bez zásahu člověka, zautomatizovat testování jako celek. Zejména plánování a návrh testů jsou činnosti, které jsou natolik intelektuálně náročné, že je s pomocí žádného nástroje doposud nebylo možné nasimulovat.
2. Představa: jeden testovací nástroj se hodí na vše.
 - Skutečnost: bohužel ještě nebyl vytvořen takový nástroj, který by dokázal podpořit všechny fáze a druhy testování a bylo jej možné použít na všech platformách a v kombinaci se všemi IT technologiemi. Proto je potřeba pečlivě vybírat nástroj vhodný pro potřeby daného projektu.
3. Představa: automatizace přinese okamžité snížení nákladů na testování.
 - Skutečnost: manažeři rádi slyší o snížení nákladů, které automatizace testování může přinést. Často však také mylně předpokládají, že tato úspora přijde okamžitě po spuštění prvního testu. Ve skutečnosti je však vývoj automatizovaných testů aktivitou natolik náročnou, že se úspory dostaví až po několika týdnech či dokonce měsících provádění testů.
4. Představa: automatizace přinese výrazné zkrácení doby trvání testování.
 - Skutečnost: vývoj automatizovaných testů a proces automatizace testování jako celek je náročný úkol, jehož realizace vyžaduje svůj čas. Může se tedy stát, že i přesto, že byla automatizace provedena správně, harmonogram testování se tím nezmění či dokonce prodlouží. Nicméně tento čas bude využit k důkladnějšímu a efektivnějšímu otestování aplikace.
5. Představa: intuitivní a snadno použitelný nástroj.
 - Skutečnost: celá řada výrobců nástrojů pro automatizované testování se na svých webových stránkách pyšní tím, jak je jejich nástroj intuitivní a snadno použitelný. Realita je však taková, že pokud tester s daným nástrojem nikdy předtím nepracoval, musí se s ním nejprve naučit pracovat, což vyžaduje buď specializované školení, nebo čas strávený samostudiem.

6. Představa: stoprocentní pokrytí kódu testy.

- Skutečnost: existují části kódu aplikace, u nichž by stoprocentní pokrytí testy představovalo příliš mnoho úsilí v porovnání s důležitostí daného kódu. Navíc existuje nekonečné množství variant vstupů a výstupů aplikace a k tomu lze obvykle aplikaci projít několika různými způsoby, což by znamenalo testování aplikace donekonečna.

Dalším úkolem test manažera je vypracovat analýzu přínosů a nákladů automatizace testování. K tomu potřebuje mít analýzu stávajícího stavu, na základě níž byl vznesen návrh na automatizaci testování, informace o potřebách firmy či organizace, v rámci níž by měla být automatizace testování realizována, a informace o daném projektu, na kterém by měly být testy automatizovány. Analýza přínosů a nákladů by totiž měla být opřena nejen o znalost oblasti automatizace testování, ale také o potřeby daného projektu a organizace jako celku. Mezi typické přínosy automatizace testování patří dle knihy *Automated software testing: introduction, management, and performance* [18 s. 37–51]:

1. Zvýšení spolehlivosti aplikace.

- Automatizace testování umožňuje aplikaci otestovat více do hloubky a/nebo ve větším rozsahu, díky čemuž je aplikace důkladněji prověřena a pokud jsou testy správně navrženy, dokáží odhalit všechny závažné chyby aplikace. Opakovaným prováděním testů pak lze zjistit, zda již byly problémy odstraněny a po několika spuštěních testů, které všechny skončily bez chyb, lze říci, že je aplikace stabilní a neobsahuje závažné chyby.
- Automatizace navíc poskytuje kvalitativní metriky – vyhodnocováním výsledků opakovaně spouštěných testů lze zjistit spoustu informací jak o testované aplikaci, tak o testech samotných, což poskytuje dobrý základ pro rozhodování o optimalizaci.

2. Zvýšení efektivity testování.

- Automatizované testy se velice dobře hodí pro tzv. dýmové testování (angl. Smoke Testing) spouštěné automaticky po každém sestavení aplikace. Tyto testy poskytují rychlou zpětnou vazbu na nejzákladnější funkcionalitu aplikace a případné problémy lze začít hned řešit. Díky tomu se nestane, že by tester obdržel instrukce k otestování aplikace, připravil se na manuální test a následně zjistil, že aplikace spadne hned po prvním kroku.
- Automatizované testy umožňují otestovat aplikaci multiplatformně. Provést všechny manuální testy na několika různých operačních systémech, prohlížečích a následně i verzích prohlížečů je pro testera-člověka činnost velice zdlouhavá a nezábavná. Automatizovaných testů lze přitom spustit více najednou

a v různých konfiguracích – stejný test tak běží na různých platformách paralelně, čímž se výrazně ušetří čas a výsledky lze navíc snadno porovnat.

- Díky tomu, že automatizované testy prověřují základní funkcionalitu aplikace, se mohou testeři věnovat více intelektuálně náročným testovacím aktivitám – objevování nových chyb, testování uživatelské přívětivosti aplikace, přípravě dalších testovacích scénářů apod.
- Automatizované testy se obvykle spouští v době, kdy nejsou stroje využívány – typicky v noci. Tak lze významně ušetřit čas a zdroje – zatímco testeři spí, stroje pracují. Ráno pak tester zkontroluje výsledky a v případě neúspěšných testů prověří, zda je potřeba provést změny v testovacím skriptu nebo jde o chybu aplikace.

3. Úspory ze snížení náročnosti testování.

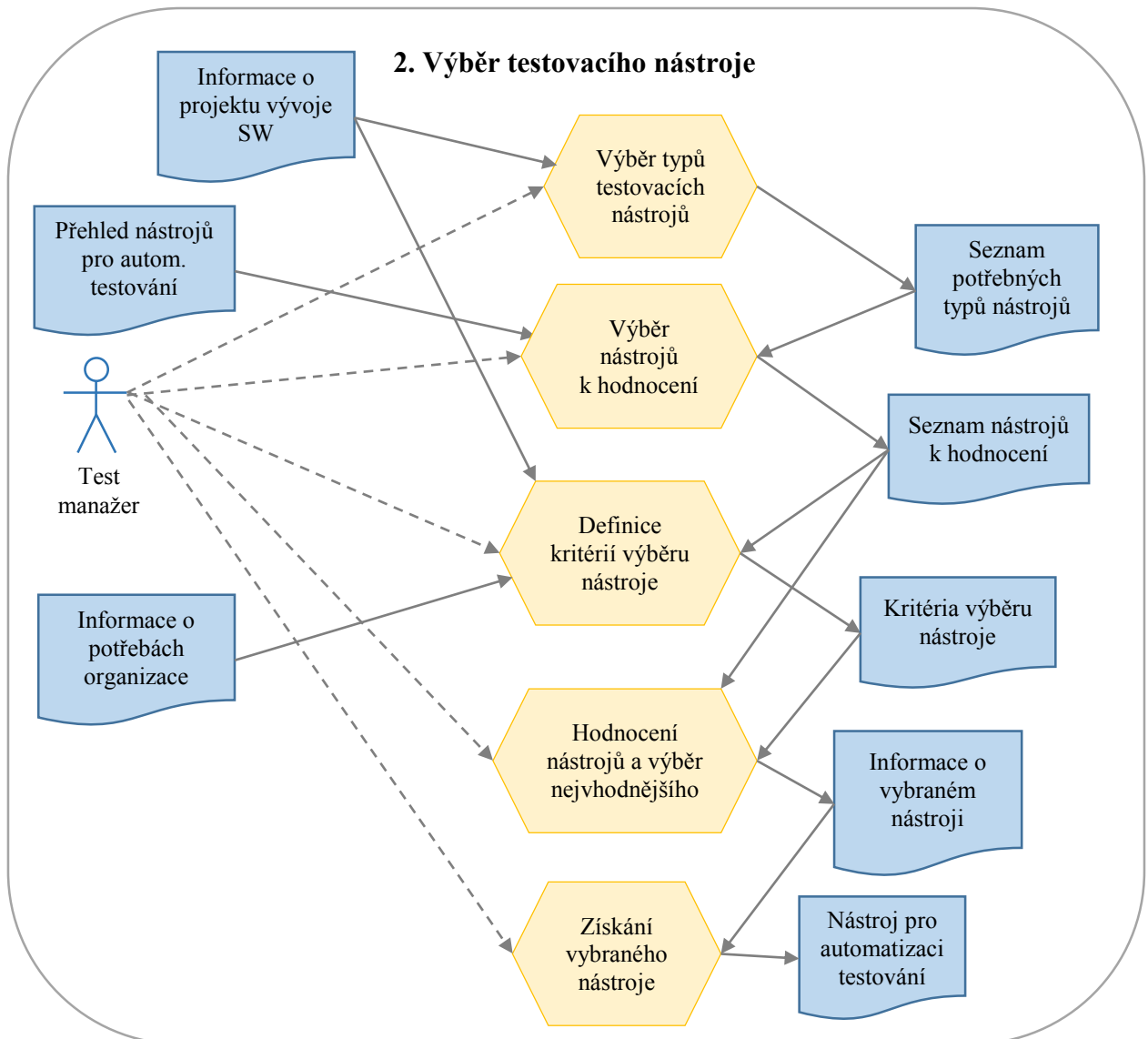
- Záhy po začátku procesu automatizace testování se může zdát, že se náročnost testování nejen že nesnížila, ale naopak zvýšila – příprava automatizovaných testů trvá dlouho a vyžaduje práci několika zkušených testerů. K úsporám pak začne docházet až ve fázi spouštění testů – automatizované testy jsou rychlejší než manuální testování a navíc jejich běh nestojí „prakticky nic“. Pouze je potřeba počítat s náklady na jejich údržbu. Ve výsledku lze automatizací dosáhnout přibližně 75 % úspory času oproti výhradně manuálnímu testování. [18 s. 50]

Posledním úkolem test manažera je pak vytvořit přehled aktuálně dostupných nástrojů, ve kterém uvede jejich charakteristiku, výhody a nevýhody, cenu licence a školení a zmíní, za jakých podmínek je vhodné či nevhodné jej použít (rozhodující jsou např. podporované platformy, technologie, charakter projektu vývoje SW apod.). Na konec by měl také alespoň přibližně stanovit kritéria výběru nástroje. [18 s. 57]

Na základě podkladů a konzultací s test manažerem (vyjasnění mylných očekávání, analýzy přínosů a nákladů automatizace testování a přehledu dostupných nástrojů) pak může manažer daného projektu rozhodnout, zda má smysl automatizaci testování začít realizovat či nikoli. [18 s. 55]

4.1.2 Výběr testovacího nástroje

Pokud již bylo rozhodnuto, že má automatizace testování v daném případě smysl, lze přistoupit k dalšímu procesu, kterým je výběr testovacího nástroje. V současné době existuje na trhu obrovské množství nástrojů, jak placených, tak volně dostupných, a je tedy vhodné proces výběru systematizovat a možné varianty pečlivě zhodnotit s ohledem na potřeby daného projektu i organizace jako celku. Hodnocení by přitom mělo probíhat podle předem připravených kritérií tak, aby bylo možné nástroje následně seřadit podle vhodnosti k implementaci. [18 s. 12]



Obrázek 18: Detail procesu výběru testovacího nástroje (Zdroj: autorka, inspirováno [18 s. 69])

Výběr typů testovacích nástrojů

V první řadě musí test manažer rozhodnout, které oblasti testování je v daném případě vhodné automatizovat. Při tomto rozhodnutí by měl respektovat zejména potřeby daného projektu – používané technologie, charakter projektu a vyvíjené aplikace, rychlost změn atd. Na základě těchto informací pak identifikuje typy nástrojů, které jsou pro automatizaci testování v daném případě potřeba – např. nástroje pro sledování stavu zaznamenaných chyb, nástroje pro regresní funkcionální testování, testování výkonnosti atd. [18 s. 68, 76]

Výběr nástrojů k hodnocení

Poté provede test manažer předvýběr nástrojů pro automatizaci testování. Z přehledu nástrojů pro automatizaci testování, který vytvořil v procesu Rozhodování o automatizaci testování, vybere pouze několik málo těch, které splňují základní minimální požadavky dané potřebami projektu a organizace, zejména co se týče typu nástroje, cen licence a školení, podporovaných technologií, poskytované funkcionality atd. Dojde tedy ke zredukování souboru nástrojů, které budou následně detailně hodnoceny, na přibližně 3-5 kandidátů. Vyhodnocování všech aktuálně dostupných nástrojů podle zadaných kritérií by bylo zcela neefektivní. [18 s. 93–94]

Definice kritérií výběru nástroje

Následně test manažer vytvoří sadu kritérií hodnocení nástrojů, přičemž bere ohled nejen na potřeby daného projektu, ale celé organizace – neboť výběr nástroje může mít dopad na chod celé firmy, ať už finanční (v případě výběru komerčního nástroje), tak procesní a organizační, neboť automatizace obvykle vyžaduje změnu v zavedených procesech testování a ke změnám dojde i v personální oblasti (nábor nových test inženýrů a/nebo školení stávajících). Kritéria výběru nástroje mohou být například: [18 s. 90–93]

1. Snadnost použití – ačkoli jde o obtížně měřitelné kritérium, mělo by být bráno v úvahu. Existují nástroje jednoduché, intuitivní a snadno použitelné a naopak také nástroje robustní, jejichž efektivní používání vyžaduje určité zkušenosti.
2. Možnost upravit nástroj dle vlastních potřeb – zda je možné funkcionality nástroje rozšířit nebo upravit podle potřeb projektu či nikoli, popř. jak je tato změna obtížná.
3. Podpora různých platforem – je nástroj vázán jen na určitou platformu nebo je multiplatformní? A v případě funkcionálních testů – umožňuje simulaci různých operačních systémů, internetových prohlížečů a jejich verzí?
4. Skriptovací jazyk – má nástroj vlastní jazyk nebo využívá některý z rozšířených programovacích jazyků (např. Java)? Jaké možnosti tento jazyk nabízí? Umožňuje využívat proměnné, cykly a znovupoužití kódu?
5. Možnost nahrávat sekvence kroků – v případě funkcionálních testů se může hodit funkcionality nahrávání a následného spouštění sekvence kroků (angl.

Record/Playback). Je ale nutné zjistit, zda lze automaticky vytvořené skripty upravovat a jaká je jejich udržovatelnost – absolutně nevhodné je celý test nahrávat znovu kdykoli se aplikace změní.

6. Možnost automatizovat i spouštění testů, nejen jejich provádění – zda nástroj umožňuje nastavit automatické spuštění testů v určitý čas nebo po určité události.
7. Možnost využití nástrojů pro spolupráci a správu verzí – zda je možné synchronizovat práci více testerů, např. pomocí Subversion, CVS, TFS atd.
8. Vytváření přehledných reportů – v jaké podobě nástroj poskytuje informaci o výsledcích testů? Popř. je možné jej integrovat s dalším nástrojem, který dokáže vytvářet přehledné a srozumitelné reporty? Lze výsledky kvantifikovat?
9. Parametrizace testů – je možné vytvářet různé varianty téhož testu s různými hodnotami parametrů, např. vstupních a výstupních dat apod.?
10. Cena licence, školení a dalších vstupních nákladů.
11. Zralost produktu a zkušenosti výrobce – jde o nový produkt začínající firmy nebo již má nástroj a jeho výrobce za sebou dlouhou historii?
12. Četnost aktualizací – jak často vycházejí nové verze nástroje? Stará se výrobce o vývoj nových funkcí a opravu chyb? (Ano, i nástroj pro automatizaci testování je software a obsahuje chyby.)
13. Dostupnost podpory, dokumentace, výukových materiálů a diskusních fór na internetu – jak snadné je sehnat informace o nástroji a řešení častých i neobvyklých problémů.
14. Popularita nástroje – jak velké procento trhu nástroj zaujímá je obvykle v korelaci s jeho kvalitou.

Dále je také potřeba se rozhodnout, v jakém rozmezí se budou hodnoty kritérií pohybovat – zda bude rozsah hodnot 1–5 nebo 1–10 a zda vyšší hodnota znamená lepší hodnocení či naopak. Jakmile jsou kritéria výběru nástroje a jejich škála definovány, lze k nim přiřadit stupeň důležitosti – váhu. Tou je hodnota kritéria vynásobena, výsledky se pro každý nástroj sečtou a nástroje se podle nich následně seřadí. [18 s. 94]

Hodnocení nástrojů a výběr nejvhodnějšího

Jak již bylo zmíněno výše, hodnocení nástrojů probíhá na základě vyhodnocování jednotlivých kritérií, jejichž hodnota je následně vynásobena vahou daného kritéria. Součtem vážených kritérií je pak celkové hodnocení daného nástroje, dle kterého lze vybrané nástroje snadno porovnávat. Pokud byla kritéria a jejich váhy správně zvoleny, nástroj s nejlepším hodnocením by měl být pro daný projekt a organizaci nejvhodnější. [18 s. 94]

Pokud je to možné, doporučovala bych vybraný nástroj ještě vyzkoušet – výrobci i v případě komerčních nástrojů obvykle nabízí zkušební verze nebo dokonce osobní prezentaci funkcionality nástroje apod., což pomůže k lepšímu rozhodnutí. Dále je také vhodné se informovat u firem, které vybraný nástroj používají, zda jsou s ním spokojeni. Ne vždy je totiž nástroj tak skvělý, jak jej výrobce prezentuje. [18 s. 98–99]

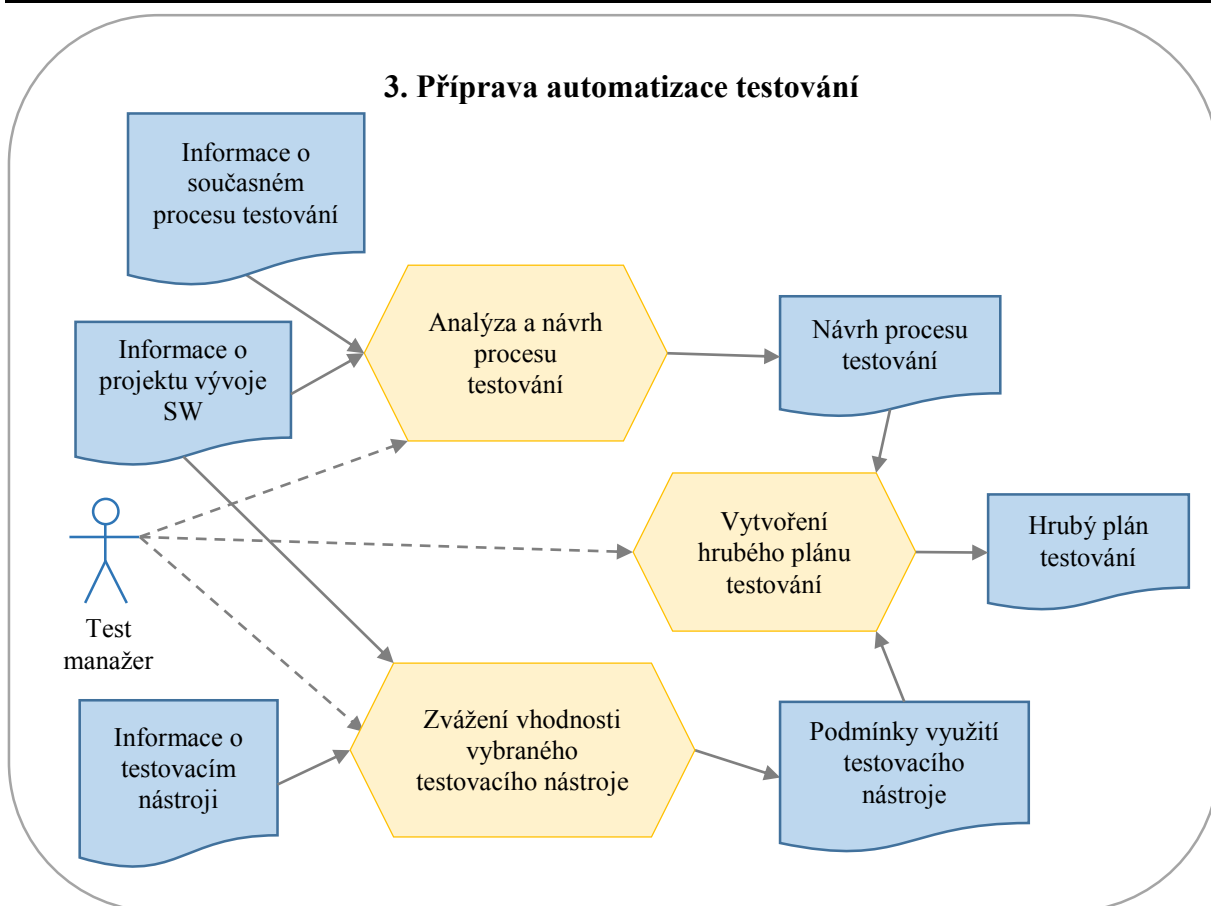
Získání vybraného nástroje

Nakonec, když je již nástroj vybrán, je potřeba jej získat – v případě komerčního nástroje uzavřít smlouvu s poskytovatelem a získat instalační soubor a licenci na jeho použití nebo v případě online nástroje přístupové údaje, zatímco volně dostupný nástroj stačí jen stáhnout z oficiálních stránek výrobce (pozor – některé nástroje jsou zdarma pouze pro nekomerční užití či za určitých podmínek).

4.1.3 Příprava automatizace testování

Po skončení výběru a úspěšném získání testovacího nástroje je potřeba proces automatizace testování s využitím vybraného nástroje důkladně připravit. Tato fáze zahrnuje analýzu stávajícího testovacího procesu, definici metrik, cílů, strategií testování, a to vše musí být zdokumentováno a představeno testovacímu týmu. Členové týmu by měli být také informováni o možnostech vybraného testovacího nástroje, zejména, jaké typy testů mohou být jeho prostřednictvím podpořeny. Dále je také potřeba analyzovat znalosti, dovednosti a zkušenosti jednotlivých členů testovacího týmu, zejména v oblasti automatizace testování, a zajistit, aby se členům týmu dostalo potřebných znalostí. [18 s. 12]

V návaznosti na proces vývoje softwaru lze v rámci této fáze začít procházet a analyzovat specifikaci požadavků od zákazníka a začít si tak utvářet představu o vyvíjené aplikaci. [18 s. 12] Už v této fázi je možné v požadavcích nalézt celou řadu nesrovnalostí, které je v tuto chvíli možné snadno vyjasnit a odstranit. Pokud by tytéž nesrovnalosti byly zaimplementovány do produktu a odhaleny až při finálním testování, mohly by způsobit celou řadu problémů, neboť odstranění chyb ve finálním produktu je časově i finančně náročnější než ve fázi specifikace požadavků.



Obrázek 19: Detail procesu přípravy automatizace testování (Zdroj: autorka, inspirováno [18 s. 109–110])

Analýza a návrh procesu testování

Prvním krokem procesu přípravy automatizace testování by měla být analýza stávajícího testovacího procesu v organizaci, který je pro potřeby daného projektu upraven a doplněn. Návrh nového procesu testování by měl splňovat následující kritéria: [18 s. 112]

1. cíle testování jsou definovány,
2. strategie testování jsou definovány,
3. potřebné nástroje jsou dostupné,
4. proces testování je dokumentován a komunikován zúčastněným osobám,
5. proces testování je měřen a vyhodnocován podle metrik,
6. uživatelé jsou zapojeni do testování softwaru,
7. testovací tým je zapojen již v rané fázi životního cyklu softwaru a jeho aktivity se prolínají celým životním cyklem softwaru,
8. harmonogram a rozpočet umožňují aplikaci dostatečně otestovat,

V případě, že některý z výše uvedených požadavků není v současném procesu testování splněn, je pro úspěch plánované automatizace testování nutné tento nedostatek napravit. Pouze správně navržený proces testování totiž může zajistit, že bude aplikace otestována řádně, včas a v rámci stanoveného rozpočtu.

Hlavním cílem testování je zvýšit pravděpodobnost, že se testovaná aplikace bude za všech okolností chovat správně a bude splňovat specifikované požadavky, a tedy uspokojí koncové uživatele. Toho je dosaženo odhalením (a zajištěním opravy) tak velkého množství defektů, jak jen je možné. Cílem automatizovaného testování je pak podpořit manuální testování v naplňování výše uvedeného cíle. Nicméně je vhodné vedle těchto obecných hlavních cílů definovat i cíle podrobnější, specifické pro daný projekt. [18 s. 116]

Strategie testování by měly pomoci defekty nejen odhalovat, ale také se jim snažit předcházet. Proto jsou strategie rozdělovány do dvou kategorií – první z nich jsou strategie pro prevenci defektů, které musí v první řadě respektovat požadavek na zapojení testovacího týmu v průběhu celého životního cyklu softwaru, dále je potřeba zvolit (nebo vytvořit) a dodržovat standardy a nakonec také využívat tzv. Quality Gates, tedy jakési milníky pro přechod z jedné fáze testování do další. [18 s. 122–125]

Druhou kategorií jsou pak strategie pro detekci defektů. Jejich součástí by mělo být pravidlo, že testovací tým zkontroluje všechny artefakty, které jsou předmětem dodávky zákazníkovi – tedy nejen samotný software, ale také specifikaci požadavků, školicí materiály, uživatelské příručky, nápovědu, příručku pro systémového administrátora atd. Další dobrou strategií pro detekci defektů je využívání nástrojů pro automatizaci testování, které podporuje a usnadňuje manuálního testování. Nezbytné je také definovat různé techniky testování využívané v různých fázích testování – např. rozdělit testování artefaktů na jednotkové, systémové a akceptační testování a pro každé z nich určit, jakým způsobem bude testování prováděno. Při testování je také potřeba identifikovat a vyhodnocovat rizika – podle důležitosti aplikace, rizika chyb v dané funkcionalitě a jejich možných dopadů atd. Pokud jde o kritickou aplikaci, jejíž funkcionalita je pro zákazníka vysoce důležitá, pak by strategie měly být definovány tak, aby riziko chyb v aplikaci minimalizovaly. [18 s. 123–133]

Neméně důležité jsou také metriky, na jejichž základě je proces testování měřen a vyhodnocován. Výsledky měření jsou důležitým zdrojem informací pro další rozvoj a zlepšování procesu testování. Zde hraje velkou roli automatizace testování – výsledky automatizovaných testů lze velice snadno měřit a vyhodnocovat jejich vývoj v čase.

Analýza a návrh využití testovacího nástroje

Poté, co je připraven obecný návrh procesu testování, je potřeba ještě jednou zvážit vhodnost využití vybraného testovacího nástroje a s tím související nároky na organizaci projektu a kvalifikaci členů testovacího týmu. Výsledkem je pak souhrn základních podmínek, za kterých by měla být automatizace testování prováděna. Součástí tohoto procesu by měly být následující kroky: [18 s. 134]

1. ověření, že nástroj vyhovuje potřebám projektu,
2. kontrola harmonogramu automatizace testování,
3. představení nástroje testovacímu týmu,
4. definice rolí a odpovědností a vytvoření testovacího týmu,
5. definice požadavků na školení.

Nejdříve je potřeba analyzovat prostředí vyvíjeného systému a zhodnotit, zda vybraný nástroj skutečně podporuje dané uživatelské prostředí, použité platformy a technologie, dokáže otestovat požadovanou funkcionalitu atd. Jednoduše řečeno, že je s testovanou aplikací kompatibilní a lze jej v daném případě a pro dané účely vůbec použít. [18 s. 133–134] Ačkoli se může zdát, že tyto aspekty již byly vyhodnocovány v předchozím procesu výběru testovacího nástroje, přesto je potřeba toto ověření učinit znovu. Důvodem je fakt, že se proces automatizace testování vyvíjí paralelně s procesem vývoje softwaru a v době definice kritérií ještě nemusela být byznys analýza a specifikace požadavků na software kompletní. [18 s. 15] Nyní je tedy vhodné se znovu ujistit, že nástroj potřebám daného projektu vyhovuje.

Následně by měl být zkontrolován také harmonogram testování, zejména zda ještě zbývá dostatek času pro přípravu automatizace testování. [18 s. 134] Nemělo by dojít k situaci, že je aplikace připravená k otestování, ale stále se čeká na dokončení testovacích skriptů.

Dále je potřeba nástroj pro testování představit projektovému týmu, a to nejen testerům, ale také vývojářům, specialistům pro zajišťování kvality (angl. Quality Assurance Specialists) a specialistům pro řízení konfigurací (angl. Configuration Management Specialists). Stejně tak jako manažeři, i tito specialisté mohou mít mylné představy o možnostech využití daného nástroje, které je potřeba objasnit. [18 s. 134]

Ani automatizace testování se však neobejde bez lidí, zejména těch, kteří mají potřebné dovednosti a zkušenosti pro práci s daným nástrojem a organizací celého procesu. Pro úspěch projektu musí test manažer definovat role, odpovědnosti, znalosti a dovednosti nezbytné pro výkon dané role, a zajistit, aby byly tyto role obsazeny kompetentními osobami. [18 s. 135]

Na základě výše uvedené definice rolí je pak potřeba sestavit tým. Nicméně i v dnešní době je velice obtížné sehnat dostatek senior testerů, kteří mají zkušenost s automatizovaným

testováním v daném nástroji, a tak je obvykle nutné členy testovacího týmu zaškolit, což samozřejmě není zadarmo. I tyto náklady tedy musí být brány v potaz. [18 s. 135]

Vytvoření hrubého plánu testování

Dalším krokem je pak spojit výše uvedené dohromady a vytvořit tak hrubý plán testování, který by měl pokrývat nejen oblast automatizovaného, ale také manuálního testování. Jeho obsahem jsou cíle, metriky a strategie testování, role a jejich odpovědnosti a přibližný harmonogram a rozpočet testování. Tento hrubý plán testování poskytuje solidní základ pro detailnější plánování v další fázi projektu.

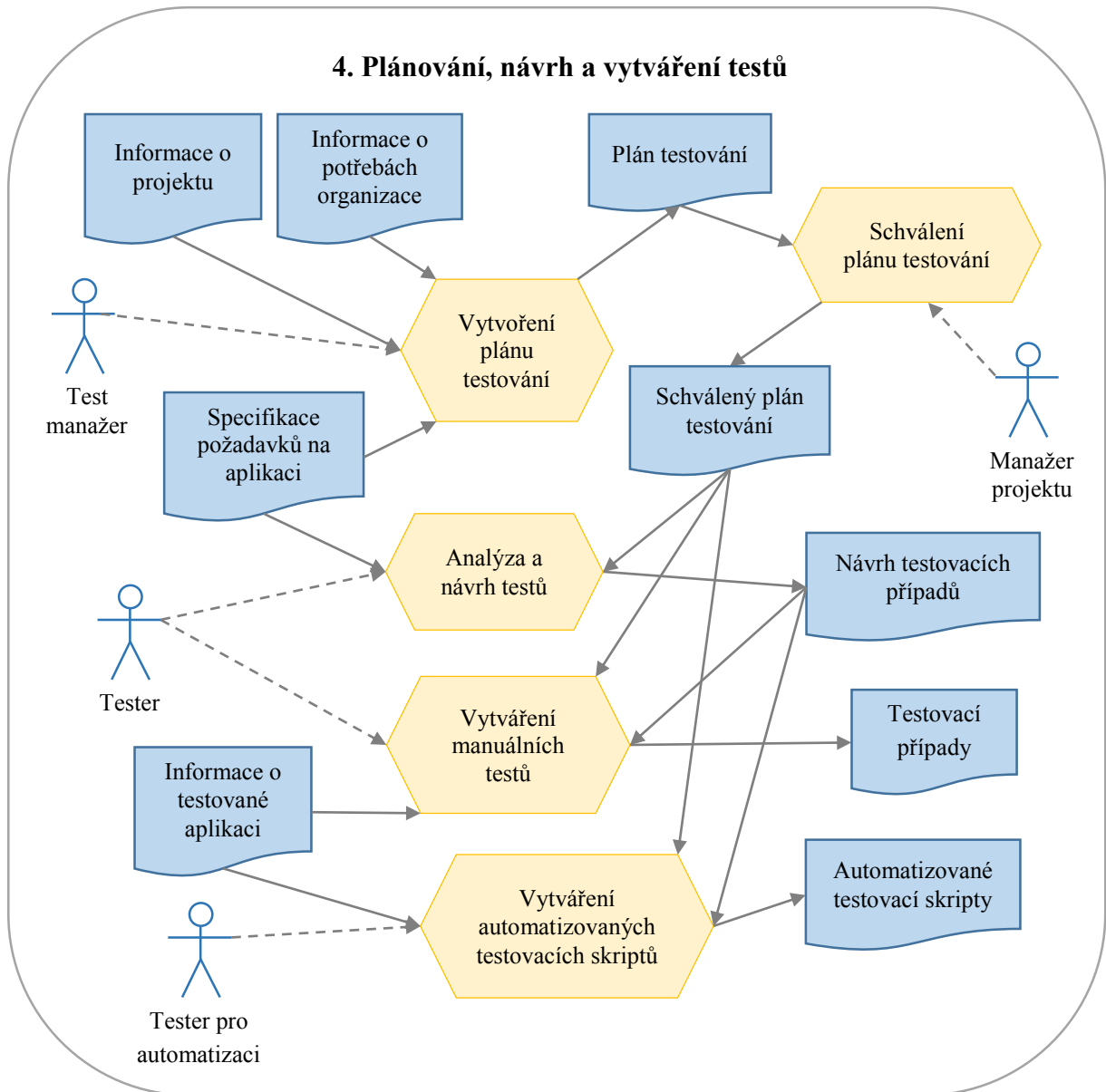
4.1.4 Plánování, návrh a vytváření testů

Po dokončení třetí fáze automatizace testování, ve které bylo důkladně prověřeno, že je k dispozici správný nástroj vhodný pro potřeby daného projektu, byl vytvořen proces testování s definovanými cíli, metrikami a strategiemi a zformován testovací tým s kompetentními osobami, je možné přistoupit k fázi čtvrté, v rámci které dochází k plánování, návrhu a následně i vytváření testů.

Fáze plánování testování zahrnuje identifikaci hardwaru, softwaru a počítačových sítí potřebných pro vytvoření testovacího prostředí, následně přípravu testovacího prostředí, definici požadavků na testovací data, detailizaci harmonogramu automatizace testování, plánování a návrh testů, identifikaci rizik a alternativních řešení, definici akceptačních kritérií, návrh způsobu řízení konfigurace testů a testovacího prostředí a nakonec i přípravu procedury pro sledování zaznamenaných defektů a s tím spojeného nástroje pro sledování chyb (angl. Bug Tracking Tool). Přílohy plánu automatizovaného testování pak mohou obsahovat jmenné konvence a jiné standardy, které budou využívány dále v průběhu automatizace testování. [18 s. 13]

Návrh testů se definuje rozsah a charakteru testů, které by měly být automatizovány, a také testovací podmínky, které by měly být testováním prověřeny. Během této fáze by také měly být také definovány standardy návrhu testů. [18 s. 13]

Poté, co jsou testy pečlivě navrženy, může začít jejich vytváření. Při vytváření automatizovaných testů musí být kladen důraz na jejich znovu-použitelnost, opakovatelnost a udržovatelnost, čemuž napomáhá využití některého ze standardů vývoje softwaru. [18 s. 13] Pokud je pro vytváření testů využito některý z běžných skriptovacích či programovacích jazyků (např. Java, C#, JavaScript apod.), pak je nasnadě zvolit a dodržovat standardy běžně používané v daném skriptovacím či programovacím jazyku. Jestliže jsou testy vytvářeny jiným způsobem (např. testovací nástroj využívá vlastní jazyk nebo speciální strukturu testů), pak je vhodné zjistit, zda nejsou pro daný nástroj k dispozici standardy vývoje a pokud ne, nebo z nějakého důvodu nejsou vyhovující, pak si stanovit vlastní.



Obrázek 20: Detail procesu plánování, návrhu a vytváření testů (Zdroj: autorka, inspirováno [18 s. 147–346])

Vytvoření plánu testování

Základním kamenem efektivního testování je jeho pečlivé naplánování, jehož smyslem je zajistit, aby činnosti, procesy, metodiky, techniky, lidé, nástroje a zařízení (hardware a software) byly organizovány a využívány efektivním způsobem. [18 s. 192]

Plán testování by měl obsahovat následující informace: [18 s. 192–197]

1. popis testované aplikace, testovacího prostředí, technologií atd.,
2. rozhodnutí o automatizaci testování – včetně příčin, které k němu vedly,
3. informace o vybraném nástroji/nástrojích a možnosti jeho/jejich využití,

4. cíle, strategie a metriky testování,
5. harmonogram a rozpočet testování,
6. rizika a alternativní řešení,
7. role, odpovědnosti a potřebné dovednosti,
8. požadavky na školení členů týmu,
9. jmenné konvence a standardy,
10. hierarchická struktura činností (angl. Work Breakdown Structure, zkr. WBS), které je potřeba v rámci procesu testování provést, včetně odhadu doby jejich trvání,
11. definice rozsahu a charakteru testování – typy testů, co a do jaké míry by mělo být otestováno atd.,
12. seznam požadavků na aplikaci a identifikace kritických částí,
13. seznam akceptačních kritérií,
14. požadavky na testovací prostředí,
15. požadavky na testovací data a způsob jejich získání,
16. definice postupu pro práci s nalezenými defekty.

Jak je zřejmé z výše uvedeného seznamu, plán testování je velice obsáhlý dokument, který v sobě shrnuje podstatné informace ze všech předchozích fází a rozšiřuje je o několik dalších.

Velice důležitou součástí plánu testování je hierarchická struktura činností, která zachycuje rozsah činností, které mají být provedeny. Jde o stromovou strukturu znázorňující rozpad hlavních činností na menší a menší úkoly, jimž je přiřazen odhad náročnosti (doby trvání) například v člověkodnech (angl. Mandays, zkr. MD). Díky tomu je pak snadnější provést celkový odhad náročnosti testování. [18 s. 194]

Definice rozsahu a charakteru testování by měla vycházet ze specifikace požadavků na aplikaci a případů užití (angl. Use Case), jsou-li k dispozici. Popisuje, jaké typy testů by měly být provedeny – nezbytně jednotkové testy, funkcionální konfirmační a regresní testy a v závěru akceptační testy, volitelně pak testy výkonnostní, bezpečnostní atd. Mimoto uvádí i základní části funkcionality aplikace a jak podrobné a rozsáhlé by mělo být jejich testování. [18 s. 204]

Součástí plánu by měl být přehled všech požadavků, které jsou na vyvíjenou aplikaci kladeny. Ideální je spravovat požadavky v takové podobě, aby k nim v dalších fázích mohla být doplněna informace, jakým testovacím případem či scénářem je daný požadavek pokryt – k tomu lze použít buď jednoduchou tabulku v Excelu nebo některý z nástrojů pro správu požadavků (např. HP Quality Center), což pomůže zajistit pokrytí všech požadavků testy. Každá část požadované

funkcionality aplikace má přitom pro zákazníka různou míru důležitosti, a tento fakt je vzhledem k omezenému času a finančnímu rozpočtu, který je pro testování vyhrazen, potřeba zohlednit. Kritičnost funkcionality lze vyhodnotit na základě čtyř faktorů – dopadu (co se stane, když požadovaná funkcionality nebude dodána?), četnosti použití (jak často budou uživatelé danou funkcionalitu využívat?), složitosti (jak moc je požadavek komplexní?) a pravděpodobnosti selhání (některé požadavky může být obtížné naplnit a mohou způsobovat problémy). [18 s. 208–211]

Dále je také potřeba určit, jak bude nakládáno s nalezenými defekty – kam a v jaké podobě se budou zaznamenávat, jaký nástroj k tomu bude využit (např. MS Excel, HP Quality Center, Bugzilla atd.), jakým způsobem se s nimi bude pracovat, odpovědné role atd. [18 s. 197]

S plánem testování by měly být seznámeny všechny zainteresované osoby, a to nejen členové testovacího týmu, ale také vývojáři a zástupci zákazníka. Dokument by měl být schválen manažerem daného projektu. [18 s. 218]

Na závěr je vhodné upozornit, že plán testování není jen výsledkem jednorázové činnosti, ale jde o živoucí dokument, který musí být v průběhu testování aktualizován a upravován tak, aby vždy reflektoval aktuální stav. [18 s. 218]

Analýza a návrh testů

Stejně tak jako vývoj softwaru, i vytváření testů vyžaduje pečlivou analýzu a návrh. Důležitou součástí analýzy požadavků na testování je prozkoumání specifikace požadavků od zákazníka, analýzy systému vytvořené v rámci životního cyklu vývoje softwaru, případů užití (jsou-li k dispozici) a dalších informací od zákazníka, analytiků a vývojářů. Po provedení důkladné analýzy lze přistoupit k návrhu testovacích případů a scénářů. Přístupů k návrhu testů je několik, vždy je však cílem zajistit, aby byly všechny požadavky pokryty testy. Součástí analýzy a návrhu testů je také rozhodnutí, která část testovacích případů by měla být prováděna manuálně a která pomocí nástroje pro automatizaci testování. [18 s. 223–224]

Existují dva základní přístupy k analýze požadavků na testování (jaké testy bude potřeba vytvořit): [18 s. 226]

- **strukturální (statický) přístup** – testy vychází ze znalosti vnitřní struktury testované aplikace. Označováno také jako testování bílé skříňky. Patří sem jednotkové a integrační testování a je obvykle prováděno vývojáři.
- **přístup z hlediska požadavků** – testy se zaměřují na prověření funkcionality definované ve specifikaci požadavků, zkoumají jen vnější projevy, chování a výstupy testované aplikace. Označováno také jako testování černé skříňky (v případě webových aplikací spíše šedé skříňky). Jde o testování systémové a akceptační a provádí je testeři (v případě akceptačních testů zákazník).

Techniky a metody vývoje testů dle strukturálního přístupu jsou mimo rozsah kompetence a odpovědnosti testerů, proto se jím tato práce nezabývá.

Při identifikaci potřebných testů na základě specifikace požadavků lze také aplikovat dva různé přístupy: [18 s. 228–230]

- **prověření vstupů a výstupů aplikace** – testy se zaměřují na prověření, zda aplikace při zadání určitých vstupních dat poskytne správný odpovídající výstup.
- **prověření chování aplikace** – tento přístup je založen na průchodu aplikací podle zadaného scénáře (angl. functional thread testing), při kterém se ověřuje, zda se aplikace chová požadovaným způsobem. Tyto testy jsou obvykle navrhovány na základě případů užití.

Pro důkladné otestování aplikace by měly být aplikovány oba přístupy.

Při návrhu testů je také nutné mít na paměti úroveň důležitosti jednotlivých požadavků, jak bylo specifikováno v plánu testování. Kritičtější části funkcionality aplikace musí být otestovány důkladněji než ty méně důležité. [18 s. 230]

Při analýze požadavků na testování je vhodné pracovat se seznamem požadavků, který byl vytvořen v rámci přípravy plánu testování, a k jednotlivým požadavkům přiřazovat techniky, pomocí kterých budou následně navrhovány testovací případy a scénáře. Na konci návrhu testů pak v seznamu požadavků nesmí zůstat žádný, který by nebyl v rámci žádného testu prověřován – každý požadavek musí být pokryt alespoň jedním testovacím scénářem. [18 s. 231]

Technik pro návrh systémových testů je několik, mezi nejčastější z nich patří: [6 s. 101]

- rozdělení do tříd ekvivalence,
- analýza hraničních hodnot,
- rozhodovací tabulky,
- graf příčin a následků,
- testování přechodů mezi stavy,
- testování s využitím ortogonálního pole,
- testování podle případů užití.

Hlubší rozebrání těchto technik již bohužel přesahuje rozsah této diplomové práce, pro podrobnější informace doporučuji nastudovat kapitolu 8 knihy *Řízení kvality softwaru* od Petra Roudenského a Anny Havlíčkové [6].

Po dokončení analýzy požadavků testování, v rámci které bylo identifikováno, jaké testy jsou potřeba a jaké přístupy a techniky budou při jejich návrhu využívány, je možné přistoupit k návrhu testů. Pro návrh testů je vhodné využívat předem danou strukturu testovacích případů, tedy určitou šablonu, která obsahuje důležité informace pro dokumentaci testování, jako například: [18 s. 273–274]

- **ID** testovacího případu,
- **název** testovacího případu,
- **autor** testovacího případu,
- **metoda ověření** (automatizovaný test, manuální test, analýza atd.),
- **instrukce k provedení testu**,
- **podmínky k provedení testu** (např. v jakém stavu musí být aplikace nebo data?),
- **závislosti** (je předtím potřeba provést nějaký jiný testovací případ?),
- **reference na požadavek/ky**, jehož/jejichž splnění je ověřováno,
- **očekávaný výsledek**,
- **skutečný výsledek**,
- **status testovacího případu** (zatím nespouštěn = angl. no run; úspěšně dokončený = angl. passed; skončen s chybou = angl. failed atd.)

Správný návrh testu by přitom neměl pokrývat pouze pozitivní, ale také negativní průběh testu – co se stane, pokud jsou aplikaci poskytnuta nesprávná vstupní data nebo pokud test probíhá za nestandardních podmínek. [18 s. 257]

Co se týče automatizace testování, zde je klíčovým faktorem úspěchu správné rozhodnutí, které testy automatizovat a které provádět manuálně. Zde je několik zásad, které je vhodné při rozhodování o automatizaci testování dodržovat: [18 s. 262–265]

- **automatizovat postupně** – nesnažit se automatizovat všechny testy najednou, ale začít od těch, u nichž je potřeba automatizace zřejmá a dále v automatizaci pokračovat podle potřeby,
- **ne všechny testy mohou být automatizovány** – i nástroje mají své limity a některé aspekty aplikace může prověřit jen člověk,
- **nezapomínat na cíle testování** – automatizované testy by měly napomáhat k dosažení cíle testování,
- **analyzovat náročnost automatizace** – některé testy je velice náročné automatizovat a jejich vývoj by mohl být nákladnější než manuální testování,

- **klást důraz na znovupoužitelnost** – automatizované testovací skripty a jejich části by měly být navrhovány tak, aby je bylo možné využít více než jednou,
- **automatizovat často opakované testy** – čím častěji je potřeba daný test provést, tím je potenciál pro jeho automatizaci vyšší, neboť právě častým prováděním automatizovaných testů dochází k úsporám času a úsilí,
- **automatizovat testy s velkým množstvím vstupních dat** – pokud je pro provedení testu potřeba poskytnout aplikaci mnoho dat a jejich vkládání představuje spoustu opakovaných úkonů, pak je vhodnější test automatizovat, čímž se zároveň minimalizuje riziko chyb při zadávání vstupních dat,
- **zvážit možnosti použití daného nástroje** – umožňuje daný nástroj otestovat požadovanou funkcionalitu? Popř. pokud je k dispozici více nástrojů, pak je namístě rozhodnout, kterým nástroj využít pro který test.
- **automatizovat testy podle rizika** – pokud je naplnění nějakého požadavku pro úspěch aplikace kritické, pak je potřeba jej důkladně otestovat, k čemuž může automatizace testu významně dopomoci.

Návrh testů by měl probíhat paralelně s implementací testované aplikace. [18 s. 267]

Vytváření manuálních testů

Po dokončení návrhu testů je testovací tým již připraven k vytváření testů. Stále by to však mělo být ve fázi, kdy vývojáři implementují a integrují požadovanou funkcionalitu – nemělo by dojít k situaci, kdy je aplikace připravena k otestování, ale testovací případy nejsou k dispozici. [18 s. 285]

V současné době je již standardem vytvářet a spravovat manuální testy pomocí nástroje pro správu testů (angl. test management tool), jako např. HP Quality Center, IBM Rational Quality Manager, Jira apod. Tyto nástroje umožňují vytvářet a uchovávat testovací případy v předem dané struktuře na serveru, kde jsou k dispozici online všem členům testovacího týmu, spravovat jejich verze, zaznamenávat výsledky jejich provádění a na základě toho generovat přehledné reporty. [17 s. 209]

Při vytváření testů je nutné dbát na jejich udržovatelnost, znovupoužitelnost, srozumitelnost a robustnost, což může být stejně náročný úkol jako vytvoření testované aplikace. Z toho důvodu je potřeba dodržovat jmenné konvence a standardy definované v plánu testování, a to jak v případě testů manuálních, tak automatizovaných. [18 s. 287]

Dále je také potřeba specifikovat, v jakém pořadí by testy měly být prováděny. Pokud provedení nějakého testu vyžaduje, aby předtím byl proveden test jiný (např. nelze testovat úpravu záznamu, když ještě žádný záznam nebyl vložen), pak tato informace musí být uvedena v popisu testovacího případu.

Vytvořené testovací případy je také vhodné nechat překontrolovat – ačkoli je testovací tým jistě složen z odborníků, přesto by měl každý testovací případ projít revizí dalšího testera, který zkontroluje, zda je test srozumitelný, kompletní a plní svůj účel. [18 s. 304]

Během vytváření testů je také potřeba připravit testovací prostředí, aby bylo v okamžiku dokončení testované aplikace možno začít testy bez prodlení provádět. [18 s. 289]

Vytváření automatizovaných testovacích skriptů

Některé nástroje pro automatizaci testování disponují možností nahrávat sekvenci kroků a automaticky tak generovat testovací skript. Obvykle však jsou takové skripty obtížně udržovatelné (při změně testované aplikace je potřeba celý test nahrát znovu). Z toho důvodu je doporučováno automatizované testy raději vytvářet od začátku ručně pomocí skriptovacího jazyka daného nástroje, jedině tak lze naplnit požadavek na udržovatelnost, znovupoužitelnost, srozumitelnost a robustnost automatizovaných testů. [18 s. 316]

Při vytváření automatizovaných testů by měla být dodržována následující pravidla:

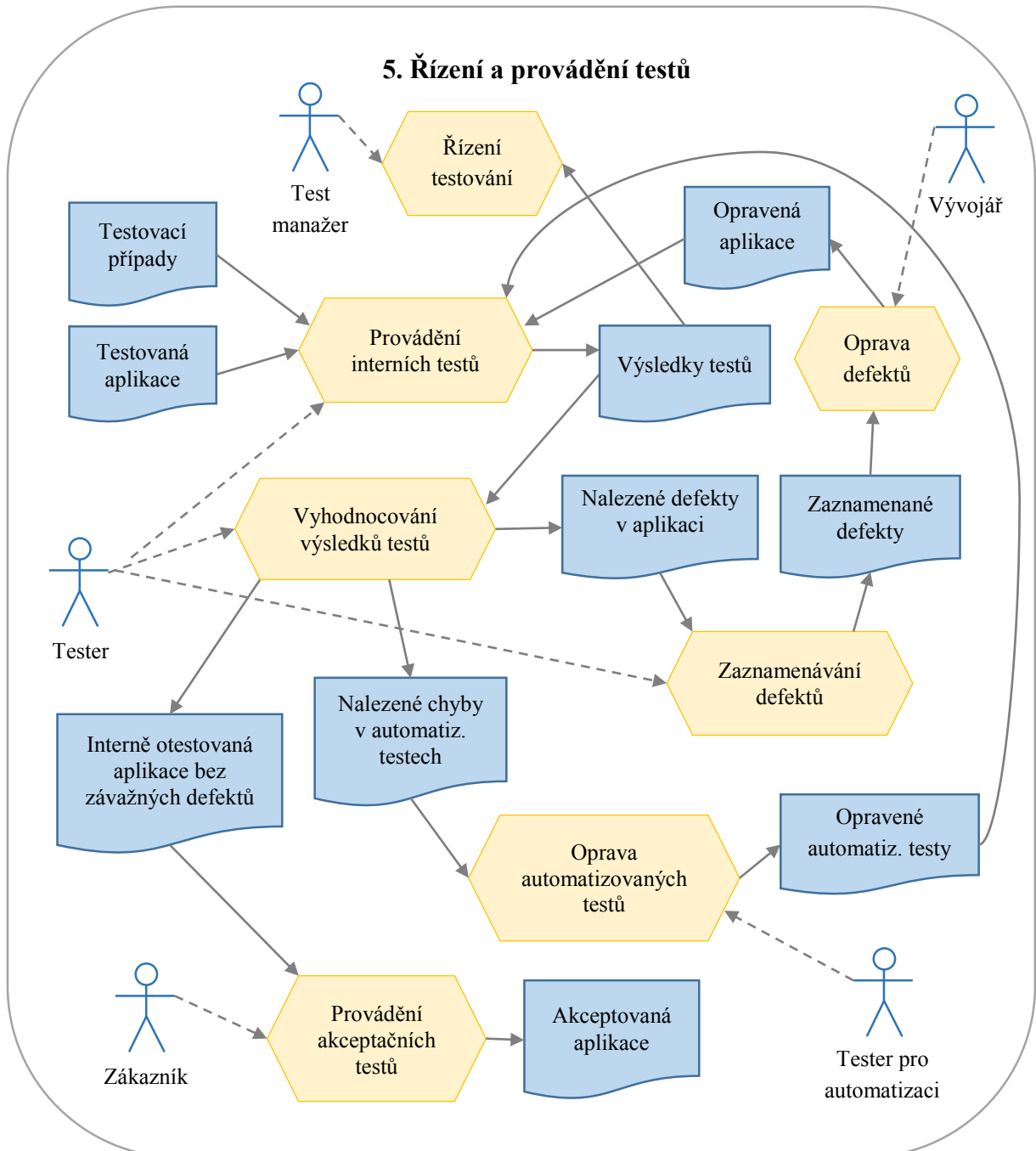
- **dodržování jmenných konvencí a standardů** definovaných v plánu testování – definují, jak by měly být testy nazvány, jaká by měla být jejich struktura, vzhled zdrojového kódu (pro zajištění čitelnosti) a další pravidla; [18 s. 317–318]
- **udržování vstupních a výstupních dat mimo kód testovacího scénáře** – vstupní a výstupní data by neměla být součástí zdrojového kódu, ale pro snadnější správu by měla být umístěna v samostatných souborech; [18 s. 310]
- **přidávat k testovacím skriptům komentáře** – testovací skript by stejně jako zdrojový kód testované aplikace měl být řádně okomentován, což významně napomáhá při dalším udržování testů; [18 s. 319]
- **rozdělovat skripty do menších logických celků** – udržovat všechny testovací skripty v rámci jednoho souboru se zdrojovým kódem je nepřehledné a neefektivní. Je tedy žádoucí skripty rozdělit do menších logických celků, které se snáze spravují. [18 s. 327–328]
- **vytvoření frameworku pro automatizované testy** – architektura testovacích skriptů by měla být jakousi knihovnou znovupoužitelných funkcí. [18 s. 346]
- **využívat cykly** – v případě opakovaných úkonů je vhodné využít cykly (angl. looping constructs), např. *For* nebo *While*, což napomáhá znovupoužitelnosti a udržovatelnosti testovacích skriptů. [18 s. 328]
- **využívat konstanty** – opět napomáhá znovupoužitelnosti a udržovatelnosti testovacích skriptů. [18 s. 328, 333]

- **využívat větvení testovacích skriptů** (angl. branching constructs) – větvení testovacích skriptů pomocí *if a else* umožňuje vytvářet robustnější testy. [18 s. 330]
- **pamatovat i na alternativní scénáře** – při běhu testu může dojít k nestandardní situaci, se kterou by se měl automatizovaný test umět vypořádat (alternativním scénářem) a správně vyhodnotit, zda se jedná o chybu či nikoli. V případě chyby by pak měl poskytnout srozumitelnou informaci o nastalém problému. [18 s. 324]

Výsledkem této fáze by měla být sada automatizovaných testů, které je možné spustit na připraveném testovacím prostředí hned, jakmile bude aplikace připravena k otestování.

4.1.5 Řízení a provádění testů

V okamžiku, kdy jsou již testy vytvořeny a připraveny a aplikace je již ve stavu, kdy ji lze otestovat, je možné začít testy konečně provádět. Vedle samotného provádění testů je potřeba také jejich výsledky vyhodnotit, zaznamenat nalezené chyby, a pokud je již aplikace ve stavu, kdy neobsahuje závažné defekty, pak ji lze předat zákazníkovi k akceptačnímu testování.



Obrázek 21: Detail procesu řízení a provádění testů (Zdroj: autorka, inspirováno [18 s. 349–378])

Provádění interních testů

Pro důkladné otestování aplikace je potřeba provést několik úrovní testů – testy jednotkové, integrační, systémové a nakonec akceptační. Testy jednotkové a integrační jsou obvykle prováděny vývojáři, proto jejich provádění není v rámci této metodiky popsáno. Poté, co aplikace úspěšně projde testy jednotkovými a integračními, přichází na řadu testovací tým, který začne provádět manuální testy podle již připravených testovacích scénářů, a začnou se spouštět testy automatizované. [18 s. 351]

Provádění manuálních testů je potřeba dokumentovat, aby bylo jasné, které z testů již byly provedeny, kdo je prováděl a s jakým výsledkem. Na základě této evidence pak lze vytvořit reporty poskytující informaci o plnění stanoveného plánu. Provedením všech testovacích případů je prověřeno splnění všech požadavků (které musí být testovacími případy pokryty, jak již bylo zmíněno v kapitole Analýza a návrh testů). [18 s. 365]

V případě automatizovaných testů je obvykle několik běhů testů neúspěšných – ať již z důvodu chyb v testované aplikaci, tak z důvodu nedokonalostí v kódu automatizovaných testů. Příčinou nesrovnalostí v kódu testů je obvykle fakt, že dokud není k dispozici první testovatelná verze aplikace, není možné testy naimplementovat tak, aby přesně kopírovaly chování aplikace. Specifikace požadavků totiž mnohdy nespécifikuje názvy tlačítek, nadpisy webových stránek, obsah dialogových oken apod. Tyto podrobnosti se tedy doladují během několika prvních dní spouštění automatizovaných testů.

Důležitou činností je také pravidelné vyhodnocování výsledků automatizovaných testů. Ačkoli většina testovacích nástrojů poskytuje přehledné reporty (popř. je lze integrovat s nástroji, které tento report vygenerují) a informace o průběhu testu, vždy je potřeba výsledky důkladně prověřit, zejména zda neúspěšný běh testu (označen jako *Failed*) skutečně představuje chybu v testované aplikaci, nebo jen neočekávanou situaci, kterou test mylně vyhodnotil jako chybu. A naopak – úspěšný průběh testu nemusí nutně znamenat, že se v dané sekvenci kroků žádná chyba nenachází. [18 s. 14] Není totiž možné v každém kroku ověřovat úplně vše, co se na stránce děje (testy by pak narostly do obrovských rozměrů a staly by se prakticky neudržovatelnými), a tak se může stát, že daná skutečnost v daném kroku nebyla prověřena a test tak chybu nezaznamenal. [18 s. 36]

V případě, že je v průběhu vyhodnocování výsledku testů objevena skutečná chyba v testované aplikaci, je potřeba o ní informovat vývojáře, což se obvykle děje prostřednictvím nástroje pro sledování defektů, který byl zvolen již ve fázi plánování testování. Obvykle jde o ten samý nástroj, který je používán pro zaznamenávání a sledování stavu chyb nalezených v průběhu manuálního testování. Na základě závažnosti a priority defektů je pak rozhodnuto, zda bude chyba opravena ihned nebo později (anebo vůbec). [18 s. 360–361]

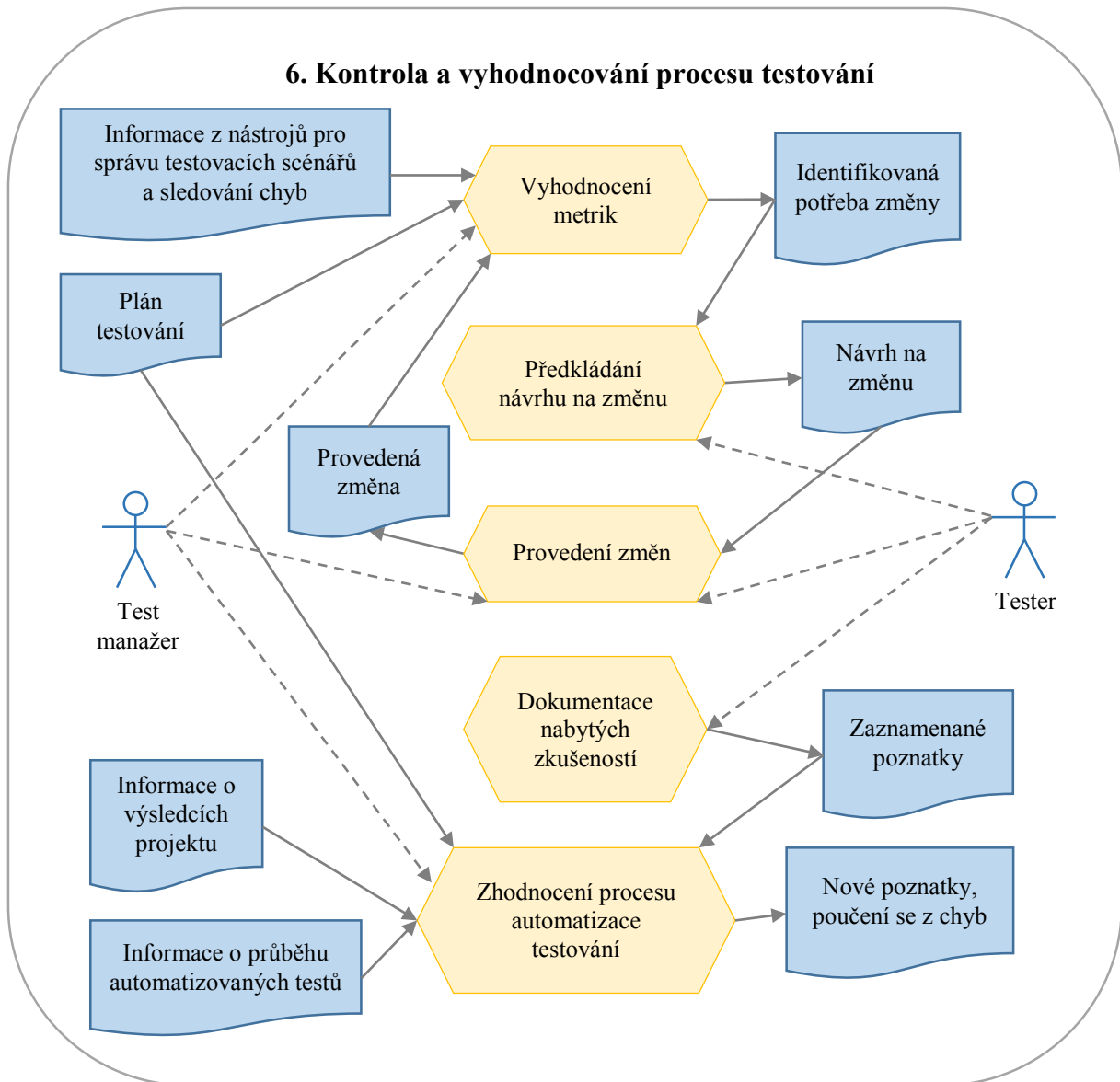
Pokud naopak nejde o chybu aplikace, ale špatně vytvořeného automatizovaného testu, pak je samozřejmě potřeba jej upravit tak, aby poskytoval správné výsledky.

Po opravení chyby je nutné provést ještě retestování (testování konfirmační), které prověří, zda byla chyba skutečně opravena. A před předáním aplikace zákazníkovi k akceptačnímu testování je vhodné se ještě jednou ujistit o správné funkcionalitě aplikace testováním regresním, které prověří, zda opravou defektu nevznikly chyby v jiné části aplikace.

Test manažer musí po celou dobu provádění testů sledovat jeho průběh a dohlížet, aby bylo testování dokončeno včas podle stanoveného harmonogramu. Organizuje, kdo bude kdy provádět jaké testy a řeší problémy, které se v průběhu testování objevují. Velice užitečnými pomocníky jsou mu přitom nástroj pro správu testovacích případů a nástroj pro sledování defektů (popř. může jít o jeden nástroj poskytující obě funkcionality), které mu poskytují užitečné informace ohledně postupu testování, na základě nichž může činit potřebná rozhodnutí. [18 s. 378]

4.1.6 Kontrola a vyhodnocování procesu testování

Proces kontroly a vyhodnocování procesu testování představuje kontinuální aktivitu, která probíhá od prvního provedení testů po zbytek vývojového cyklu testované aplikace (pokud není testování z nějakého důvodu ukončeno dříve). Cílem tohoto procesu je kontinuální zlepšování procesu, vyhodnocování metrik stanovených v plánu testování a ověřování, zda automatizace a testování jako celek poskytuje očekávané přínosy. [18 s. 14]



Obrázek 22: Detail procesu kontroly a vyhodnocování procesu testování (Zdroj: autorka, inspirováno [18 s. 379–402])

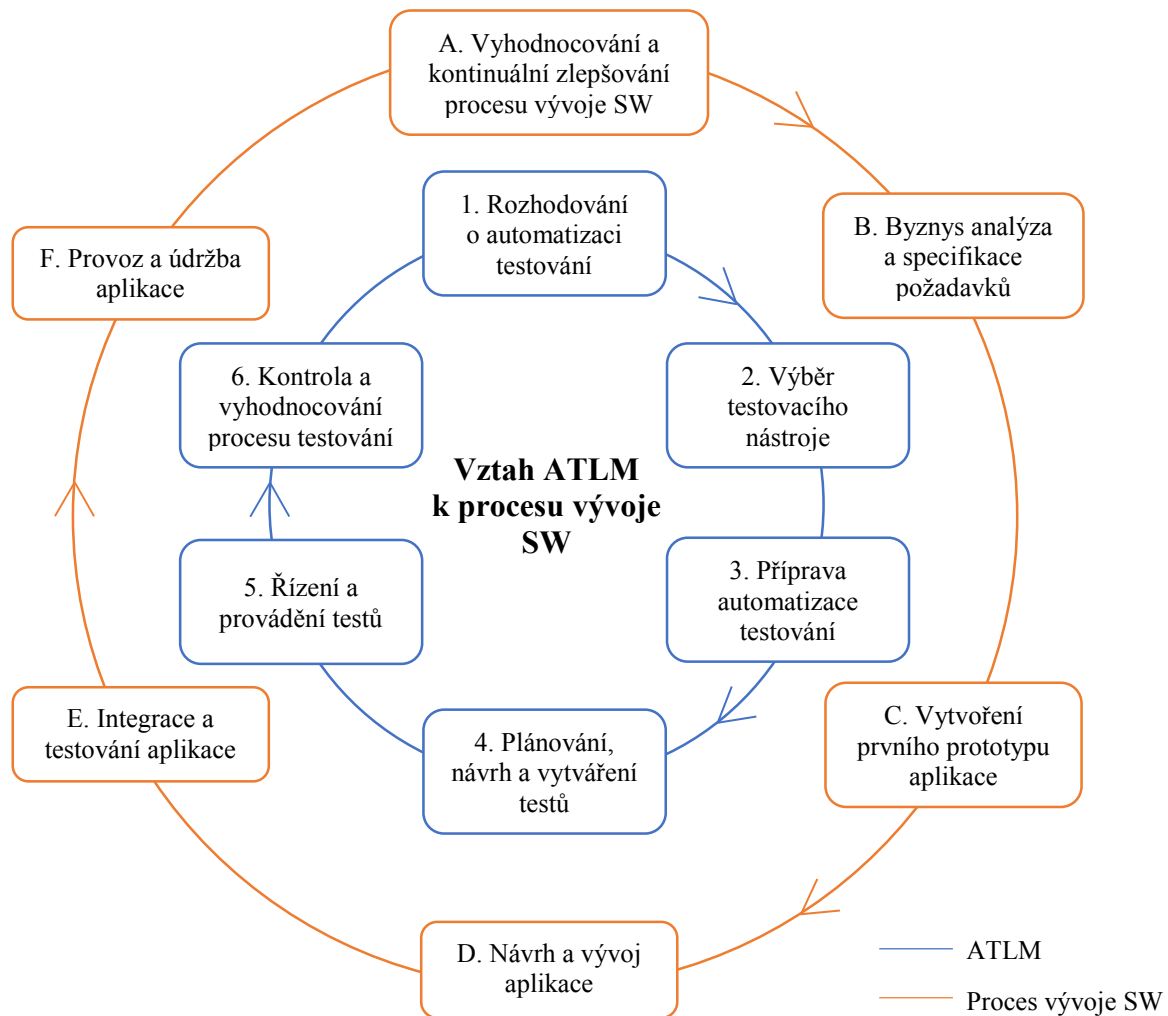
Klíčovou aktivitou v procesu kontroly a vyhodnocování testování je průběžný sběr dat o průběhu testů a pravidelné vyhodnocování metrik definovaných v plánu testování. Na základě těchto výsledků je pak možné identifikovat potřebu změny v procesu testování a předkládat návrhy pro jeho zlepšování. [18 s. 401]

Testovací tým by také měl být podporován k předkládání návrhů na zlepšení testovacího procesu a jeho členové by měli se svými kolegy sdílet své poznatky a nabyté zkušenosti (angl. lesson learned), a to ideálně nejen ústně, ale také pomocí sdílené báze znalostí. O nastalých problémech by testeři měli test manažera informovat ideálně ihned, aby bylo možné co nejdříve podniknout nápravná opatření. [18 s. 401] Veškeré návrhy na zlepšení by měly být zaznamenány, a to i v případě, že nebudou v danou chvíli realizovány. Na provedení samotných změn se pak podílí nejen test manažer, ale také testeři, neboť změny procesu se obvykle dotýkají i jejich práce.

Na konci procesu automatizace testování pak test manažer vyhodnotí, zda automatizace testování přinesla očekávané výsledky a zda se tedy vyplatila. Toto finální zhodnocení činí na základě informací o výsledcích projektu (konečná cena a doba trvání, počet nalezených defektů při akceptačních testech a v produkčním prostředí, spokojenost zákazníka a uživatelů atd.), metrik a cílů projektu definovaných v plánu testování a informací o průběhu automatizace testování (náklady a délka trvání vývoje automatizovaných testů, problémy atd.). Pokud dojde k závěru, že automatizace testování skončila neúspěchem, pak je potřeba provést analýzu příčin, které k neúspěchu vedly, a v dalších projektech se z těchto chyb poučit. [18 s. 401] Pro další zlepšování je také vhodné projít bázi znalostí, do které testeři zaznamenávali své poznatky, a nejužitečnější z nich využít i v dalších projektech.

4.2 Vztah automatizace testování a životního cyklu softwaru

Automatizace testování však nemůže probíhat nezávisle na okolí. Aby přinesla očekávané výsledky, musí být její procesy logicky navázány na odpovídající fáze životního cyklu softwaru. Tento vztah vyjadřuje následující obrázek (Obrázek 23).



Obrázek 23: Vztah ATLM k procesu vývoje SW (Zdroj: [18 s. 15], přeloženo a překresleno autorkou)

Idea automatizace testování obvykle vzniká na základě výsledků procesu vyhodnocování a kontinuálního zlepšování procesu vývoje SW (proces A). Zde dochází ke zhodnocení aktuálního stavu, a pokud je situace v oblasti testování nevyhovující, jednou z možností řešení problému je automatizace testování. Na základě analýzy současného stavu testování je tedy vydán podnět k projednání možnosti automatizace testování, čímž začíná proces rozhodování o automatizaci testování (proces 1). [18 s. 14]

Současně s byznys analýzou a přípravou specifikace požadavků v procesu vývoje SW (proces B) probíhá v procesu automatizace testování výběr vhodného testovacího nástroje (proces 2). Zde je však potřeba podotknout, že je potřeba vybrat takový nástroj, kterým bude

možné vyvíjenou aplikaci otestovat, což je obtížné, dokud o ní testovací tým nic neví. Proto je vhodné začít dostupné nástroje hodnotit až poté, co je vytvořena alespoň základní specifikace požadavků na aplikaci, a při výběru nástroje se o tyto informace opřít. [18 s. 14]

Zatímco vývojáři pracují na prvním prototypu aplikace (proces C), testovací tým v čele s test manažerem prochází specifikaci požadavků a na základě ní vytváří testovací strategie, metriky, stanovuje cíle, definuje rozsah a charakter testů apod., přičemž tyto aktivity jsou součástí úvodního procesu automatizace testování (proces 3). Už v této fázi může testovací tým poskytnout zpětnou vazbu vývojovému týmu a odhalit případné problémy a nesrovnalosti ve specifikaci požadavků. [18 s. 14]

Proces plánování, návrhu a vytváření testů (proces 4) by měl probíhat současně s procesem návrhu a vývoje testované aplikace (proces D). Samozřejmě je potřeba brát ohled na fakt, že je velmi těžké vyvíjet automatizované testy v okamžiku, kdy aplikace ještě není ani ve stavu první testovatelné verze. Proto by fáze vývoje testů měla začít až poté, co vývojáři dokončí první funkční verzi aplikace, do té doby je možné se věnovat plánování a pečlivému návrhu testů, což může významně urychlit jejich následné vytváření. [18 s. 14]

Poté, co už jsou testy připraveny a vývojáři již vytvořili daný přírůstek aplikace, je možné testy začít spouštět a řídit jejich provádění (proces 5). Tato fáze probíhá současně s manuálním testováním aplikace (proces E). Opět je na místě zdůraznit, že se automatizované testování nikdy neobejde bez testování manuálního – oba druhy testování mají svou nezastupitelnou úlohu. [18 s. 15]

Nakonec zde zbývá již jen kontrola a vyhodnocení procesu testování (proces 6). Ačkoli by tento proces měl být prováděn v průběhu celého cyklu automatizace testů a měl tak zajišťovat kontinuální zlepšování celého procesu automatizace testování, jeho výsledky jsou sumarizovány v průběhu provozu a údržby aplikace (proces F). Na jejich základě jsou pak formulovány doporučení pro další projekty. [18 s. 15]

4.3 Role

V procesu testování vystupuje několik různých rolí, které mají různé úkoly a odpovědnosti, a tedy vyžadují různé znalosti a zkušenosti. Mezi ty nejvýznamnější role, které byly zmíněny ve výše uvedené metodice pro automatizaci testování webových aplikací, patří projektový manažer, test manažer, tester, tester pro automatizaci, vývojář a nakonec zákazník. Jejich detailní popis je obsahem této kapitoly.

4.3.1 Projektový manažer

Manažer projektu, jehož úkolem je především zajistit správné naplánování, provádění, kontrola, řízení a úspěšné dokončení projektu vývoje aplikace. Jeho další zodpovědnosti a potřebné dovednosti jsou uvedeny níže. [84] [85]

- Odpovědnost a úkoly:
 - plánování, řízení a alokace zdrojů,
 - odhadování doby trvání, náročnosti a ceny projektu,
 - stanovení priorit,
 - komunikace se zákazníkem,
 - analýza, monitorování a řízení rizik,
 - řízení, motivace a usměrňování projektového týmu,
 - dohled nad plněním stanoveného harmonogramu a dodržováním rozpočtu,
 - dohled nad integritou a kvalitou vznikajících produktů.
- Znalosti a dovednosti:
 - velmi dobrá znalost životního cyklu softwaru a domény, do níž vyvíjená aplikace spadá,
 - schopnost plánovat a organizovat zdroje (lidi, čas, peníze, hardware, software...), organizovat si čas (angl. Time Management),
 - schopnost odhadnout rozsah a náročnost projektu,
 - schopnost analyzovat a řídit rizika,
 - schopnost řešit konflikty a činit rozhodnutí ve stresu,
 - prezentační a komunikační dovednosti, schopnost vyjednávat,
 - schopnost vést a utvářet tým,
 - zkušenosti s řízením projektů,
 - zaměření na dodání přidané hodnoty zákazníkovi (dodat produkt, který naplní jeho potřeby).

4.3.2 Test manažer

Vedoucí testovacího týmu. Jeho odpovědnost a znalosti potřebné pro vykonávání této role jsou uvedeny níže. [18 s. 183–184]

- Odpovědnost a úkoly:
 - komunikace se zákazníkem a dodavatelem nástroje pro automatizaci využitým v projektu,
 - nábor členů testovacího týmu, řízení a mentorování testovacího týmu,
 - vytvoření plánu testování a analýza požadavků na testování,
 - koordinace testovacích aktivit v návaznosti na průběh vývoje aplikace,
 - definice a zavedení procesu testování, sledování a řízení jeho průběhu a zajištění jeho průběžného zlepšování,
 - revize testovacích scénářů a další testovací dokumentace,
 - zajištění potřebného hardware a software (včetně testovacích nástrojů),
 - dohled nad konfigurací testovacího prostředí a nástroje pro automatizaci testování,
 - vyhodnocování metrik,
 - reporting projektovému manažerovi o stavu testování,
 - provádění změn v rámci průběžného zlepšování procesu testování.
- Znalosti a dovednosti:
 - několik let zkušeností s testováním, podrobná znalost testovacího procesu, schopnost navrhovat, vytvářet a provádět testy a zaznamenávat nalezené defekty, spravovat testovací data a řešit problémy vzniklé v průběhu testování,
 - porozumění byznys oblasti a požadavkům na aplikaci z pohledu zákazníka,
 - porozumění technické stránce vývoje aplikace,
 - schopnost formovat reálné cíle a strategie testování,
 - znalost různých testovacích nástrojů a možností jejich využití,
 - schopnost plánovat a organizovat testování a řídit členy testovacího týmu.

4.3.3 Tester

Označován také jako test inženýr, angl. Test Engineer. V souvislosti s testováním aplikací se pak mluví o testerech softwaru (angl. Software Tester, zkráceně SW Tester). Náplň jeho práce a vyžadované znalosti jsou uvedeny níže. [18 s. 185]

- Odpovědnost a úkoly:
 - vytváření testovacích případů na základě specifikace požadavků,
 - manuální provádění testovacích případů,
 - revize testovacích případů,
 - zaznamenávání objevených defektů v aplikaci,
 - retestování opravených chyb,
 - reporting test manažerovi o stavu a výsledcích testování,
 - dodržování standardů definovaných v plánu testování.
- Znalosti a dovednosti:
 - orientace a zkušenosti v oblasti testování softwaru,
 - schopnost vytvářet testovací případy,
 - znalost byznys oblasti testované aplikace,
 - znalost standardů v oblasti grafického uživatelského rozhraní a použitelnosti webových aplikací.

4.3.4 Tester pro automatizaci

Neboli tester se specializací na automatizaci testování softwaru, angl. Software Test Automation Engineer. Jeho úkolem je zejména starat se o automatizované testy, podrobnosti viz níže. [18 s. 185]

- Odpovědnost a úkoly:
 - vývoj automatizovaných testovacích skriptů na základě specifikace požadavků,
 - návrh, vývoj a provádění znovupoužitelných a udržitelných automatizovaných testovacích skriptů,
 - dodržování standardů definovaných v plánu testování (a v případě, že je k vytváření skriptů použit některý z běžných programovacích jazyků, pak i standardy pro psaní zdrojového kódu v daném jazyku),

- spouštění automatizovaných testů (pokud se z nějakého důvodu nespouští automaticky) a kontrola jejich výsledků,
- zaznamenávání defektů v aplikaci,
- reporting test manažerovi o stavu a výsledcích testování,
- Znalosti a dovednosti:
 - orientace a zkušenosti v oblasti testování softwaru,
 - zkušenosti s vytvářením a udržováním testovacích skriptů v daném testovacím nástroji,
 - znalost programování a technické stránky vývoje webových aplikací,
 - orientace ve zdrojovém kódu webových stránek (HTML, CSS, JavaScript),
 - znalost standardů vývoje aplikací.

4.3.5 Vývojář

Programátor, který vytváří aplikaci na základě požadavků od zákazníka. Seznam jeho úkolů a vyžadovaných dovedností je uveden níže. [86]

- Odpovědnost a úkoly:
 - vývoj funkcionality aplikace na základě specifikace požadavků,
 - integrace jednotlivých částí aplikace,
 - komunikace s projektovým manažerem, ostatními vývojáři, byznys analytiky, testery a dalšími členy projektového týmu,
 - vytváření a provádění jednotkových a integračních testů,
 - oprava chyb objevených během interních a akceptačních testů, popř. i chyb odhalených v produkčním prostředí,
 - optimalizace a zlepšování aplikace,
 - psaní technické dokumentace,
 - revize kódu,
 - a spousta dalších aktivit.

- Znalosti a dovednosti:
 - velmi dobrá znalost programování a technické stránky vývoje aplikací,
 - velmi dobrá znalost programovacího jazyka, který je využit pro vývoj aplikace,
 - schopnost týmové spolupráce, komunikační dovednosti,
 - orientace v nejnovějších technologiích a přístupech k vývoji aplikací,
 - znalost standardů vývoje aplikací.

4.3.6 Zákazník

Obvykle jde o zástupce zákazníka, který velice dobře zná danou oblast byznysu a projektovému týmu předává informace o potřebách a požadavcích zákazníka na vyvíjenou aplikaci. [18 s. 186]

- Odpovědnost a úkoly:
 - zastupování zákazníka ve všech jednáních v rámci projektu,
 - komunikace ohledně byznys požadavků na vyvíjenou aplikaci s test manažerem,
 - provádění akceptačních testů,
 - poskytování zpětné vazby.
- Znalosti a dovednosti:
 - výborná schopnost komunikovat s lidmi,
 - velice dobrá znalost byznysu zákazníka,
 - kompletní znalost potřeb a požadavků zákazníka,
 - alespoň základní, uživatelská orientace v IT.

Poznámka na závěr: procesu testování se může účastnit rolí více, záleží na potřebách a možnostech projektu – např. vhodné je mít specializované **test analytiky**, jejichž úkolem je především revize specifikace požadavků a identifikace potřebných testů, monitorování procesu testování a vyhodnocování celkové kvality testování. [87]

4.4 Zavedení metodiky

V předchozích podkapitolách byla představena metodika pro automatizaci testování webových aplikací, byly popsány její procesy, aktivity, artefakty a zúčastněné role. Nyní je namístě uvést, jakým způsobem metodiku uvést do praxe.

Pro úspěšné zavedení výše uvedené metodiky pro automatizaci testování je potřeba provést následující kroky:

1. zhodnotit vhodnost metodiky pro danou organizaci,
2. zhodnotit proces testování v organizaci,
3. zpřístupnit metodiku všem zainteresovaným osobám,
4. zavést metodiku a vyžadovat její dodržování,
5. být otevřený změnám a neustálému zlepšování metodiky.

V první řadě je nutné zhodnotit, zda je vůbec v dané organizaci vhodné tuto metodiku zavádět. Pokud jde o malou firmu, která se vyznačuje živelností a samo-organizací a tento způsob práce jí poskytuje požadované výsledky, pak bych se v daném případě do zavádění této rigorózní metodiky nepouštěla a raději bych doporučila některou z agilních metodik, které poskytují větší volnost a jsou založeny především na principech a nejlepších praktikách, nikoli na procesech, činnostech a artefaktech. [76 s. 24] Naopak bych tuto metodiku doporučila pro firmu většího rozsahu s početnějším testovacím týmem.

Pokud je v organizaci vyžadováno zavedení automatizovaných testů, pak je vhodné nejprve zhodnotit proces testování ve firmě obecně. Pokud je totiž samotný proces testování špatně organizovaný, nedokumentovaný a celkově chaotický, pak automatizace není dobrou volbou. Jak píše Dorothy Graham v knize *Software Test Automation*, „*Automating chaos just gives faster chaos.*“ [19 s. 11] – automatizace chaosu přináší jen rychlejší chaos.

Dalším podstatným krokem je zpřístupnění metodiky všem zainteresovaným osobám, aby se s ní mohli seznámit a pochopit koncepci realizovaného procesu automatizace testování. Nejlepším způsobem zveřejnění metodiky je sdílené úložiště nebo tzv. Wiki¹⁵, kde jsou k dispozici vždy aktuální informace, kterými se mohou členové testovacího týmu řídit.

Poté, co je metodika zpřístupněna všem zúčastněným osobám a ty jsou s ní seznámeny, je možné začít proces automatizace testování podle této metodiky zavádět a průběžně sledovat její dodržování. Tento úkol je především zodpovědností test manažera, který by měl dbát na to, aby

¹⁵ Wiki – speciální webová aplikace, která slouží pro sdílení informací na internetu. Umožňuje snadno přidávat a upravovat informace a sdílet je s ostatními uživateli. V současné době jsou tyto aplikace velice oblíbené v organizacích jako báze znalostí. [83 s. 60]

byl proces automatizace testování prováděn systematicky tak, jak jej metodika popisuje, a v případě nejasností poskytovat členům testovacího týmu podporu v jejím porozumění a dodržování.

Posledním krokem, nebo spíše zásadou, je být otevřený ke změnám a neustále hledat možnosti ke zlepšení. Žádná metodika není dokonalá a bez výjimky aplikovatelná na všechny projekty, vždy existuje možnost, jak ji vylepšit. V případě změny metodiky by měla být nová verze opět zveřejněna všem zainteresovaným osobám, jak již bylo zmíněno výše.

4.5 Shrnutí

Obsahem této kapitoly je metodika automatizace testování webových aplikací, která definuje 6 základních procesů, popisuje aktivity, které jsou v rámci nich prováděny, artefakty, které jsou při těchto aktivitách využívány nebo jsou naopak jejich výsledkem, a nakonec také zúčastněné role, jejich úkoly a požadavky na jejich znalosti a dovednosti. Následně jsou uvedeny instrukce, jak lze tuto metodiku zavést do praxe.

Metodika byla vytvořena především na základě metodiky ATLM popsané v knize *Automated Software Testing* z roku 1999 [18], která byla upravena a aktualizována na základě osobních zkušeností autorky a doplněna o informace z dalších zdrojů. Metodika nebyla prozatím ověřena v praxi, nicméně je postavena na důkladné rešerši odborné literatury vztažené k tématu automatizace testování, a proto jsem přesvědčena o tom, že ji lze v praxi aplikovat. Naskýtá se také možnost ji využít na projektech realizovaných v rámci Kompetenčního centra SQA na Vysoké škole ekonomické v Praze.

Součástí vytvořené metodiky je také výběr testovacího nástroje vhodného pro potřeby daného projektu. Jedním z nástrojů pro automatizované funkcionální testování webových aplikací je přitom Selenium WebDriver, jehož využití je podrobně popsáno v příloze této diplomové práce. Pokud je tedy pro automatizaci testování na daném projektu vybrán právě nástroj Selenium WebDriver, pak tato diplomová práce poskytuje nejen návod, jak celý proces automatizace provést, ale také příručku pro využití tohoto nástroje při vytváření samotných testovacích skriptů.

5 Závěr

Hlavním cílem této diplomové práce bylo přiblížit čtenáři problematiku automatizace testování softwaru a poskytnout systematický návod pro úspěšné zavedení automatizovaných testů webových aplikací s využitím vybraného testovacího nástroje. Tohoto cíle bylo dosaženo prostřednictvím rešerše dostupných knižních a internetových zdrojů věnovaných problematice automatizovaného testování a využití nástroje Selenium WebDriver, přičemž tyto informace byly doplněny o zkušenosti z vlastní testovací praxe. Výsledkem této je pak obsah celé této diplomové práce, která představuje komplexní metodický materiál pro realizaci automatizace testování.

Vedle hlavního cíle byly definovány také čtyři dílčí cíle. Prvním z nich bylo vymezení základních pojmů z oblasti testování, což bylo důležité zejména pro správné pochopení dalších částí práce. Tento cíl byl naplněn již v úvodu práce v rámci kapitoly 1.7 Vymezení základních pojmů, která definuje pojmy jako testování softwaru, softwarová chyba, specifikace požadavků a testovací případ a také kategorizuje typy testů.

Druhým dílčím cílem pak byla kategorizace nástrojů pro automatizované testování webových aplikací. Součástí této diplomové práce je kapitola věnovaná testovacím nástrojům – 3.2 Nástroje pro automatizované testování webových aplikací, která popisuje a kategorizuje druhy nástrojů podle různých hledisek a uvádí příklady konkrétních komerčních i nekomerčních nástrojů. Tímto tedy považuji i tento dílčí cíl za splněný.

Dalším z dílčích cílů bylo vytvořit metodiku pro automatizaci testování webových aplikací, která poskytne systematický návod pro úspěšné zavedení automatizovaných testů. Výsledkem naplnění tohoto cíle je kapitola 4 Metodika pro automatizaci testování webových aplikací, která poskytuje systematický návod, jak realizovat automatizaci testování v rámci projektu vývoje softwaru, konkrétně webové aplikace.

Posledním dílčím cílem pak bylo vytvoření příručky pro testovací nástroj Selenium WebDriver. Vzhledem k omezenému rozsahu této diplomové práce musela být tato část přesunuta do přílohy. Příloha A: Uživatelská příručka k nástroji Selenium WebDriver tedy obsahuje kompletní návod, jak s daným nástrojem pracovat, a to od vytvoření prvního testovacího skriptu až po pokročilé možnosti využití a architekturu testů. I poslední cíl byl tedy naplněn.

Tato diplomová práce tedy poskytuje dostatek informací pro realizaci automatizace testování webových aplikací s využitím vybraného nástroje, kterým je v tomto případě jeden z nejpopulárnějších nástrojů pro automatizaci funkcionálního testování webových aplikací – Selenium WebDriver. Nicméně bych na tomto místě chtěla podotknout, že automatizace testování je téma velice široké a obsáhlé a rozhodně není možné jej kompletně popsat v rámci jedné diplomové práce. Tuto skutečnost dokládá fakt, že na toto téma již bylo napsáno několik šestisetstránkových knih a i přesto žádná z nich není schopna poskytnout zcela kompletní návod, jak celý proces automatizace testování provést.

Terminologický slovník

Tabulka 3: Terminologický slovník

Termín nebo zkratka	Význam
Aplikační programové rozhraní (API, Application Program Interface nebo také Application Programming Interface)	Sada příkazů, funkcí a protokolů, které mohou programátoři využít při programování softwaru. API umožňuje programátorům používat předdefinované funkce, díky čemuž nemusí psát kód zcela od začátku. [69]
RIA (Rich Internet Application)	Moderní webové stránky, které běží ve webovém prohlížeči, ale umožňují uživateli stejný komfort práce jako běžná aplikace nainstalovaná na počítači uživatele. Na rozdíl od běžných webových aplikací probíhá celá řada výpočtů i na straně klienta, nejen na serveru. [70]
Framework	Platforma, která usnadňuje vývoj softwarových aplikací pro určitý operační systém. Typicky obsahuje předdefinované třídy a funkce, které se v aplikacích často opakují. Programátoři je pak mohou při vývoji využít jako stavebníci a nemusí tzv. „znovu vynalézat kolo“. [35]
CSS (Cascading Style Sheets, česky kaskádové styly)	jazyk, pomocí kterého lze definovat, jak mají být jednotlivé elementy na webové stránce graficky zobrazeny. Pro identifikaci elementů, na které mají být aplikovány vybrané grafické vlastnosti, se využívají tzv. CSS selektory, které pracují s HTML kódem webové stránky. [3]
XPath	jazyk, pomocí kterého lze vyjádřit cestu k XML elementům nebo atributům, zjišťovat jejich počet, provádět matematické operace nad získanými hodnotami atd. XPath selektory lze ale využívat i pro identifikaci HTML elementů na webové stránce. [4]
Sandbox	speciální prostředí internetového prohlížeče, které aplikuje bezpečnostní politiky a zabraňuje spouštění škodlivý kód na zařízení klienta. [24 s. 64]
Integrované vývojové prostředí (Integrated Development Environment, IDE)	sada programů, která nabízí řadu funkcí usnadňujících vývoj programů. [61 s. 38]
Java Development Kit (JDK)	sada nástrojů (běžové prostředí Javy, překladač, generátor dokumentace, ladící program a další.) pro vývoj aplikací v programovacím jazyku Java. Obecně se této sadě říká SDK (Software Development Kit), JDK je označení pouze pro Javu. [61 s. 38]

Objektový model dokumentu (Document Object Model, DOM)	rozhraní nezávislé na platformě ani programovacím jazyku, které popisuje strukturu dokumentu, v tomto případě webové stránky. Umožňuje programům a skriptům dynamicky přistupovat k dokumentu a upravovat jeho obsah, strukturu a styly (vzhled). Dokument je poté zpracován a výsledek je zobrazen v internetovém prohlížeči. [88] Objektový model dokumentu je nástrojem Selenium WebDriver využíván pro přístup k elementům na webové stránce.
Element	Prvek na webové stránce – např. tlačítko, hypertextový odkaz, vstupní pole, obrázek, nebo také logický blok prvků, např. horizontální menu, patička webové stránky atd. Elementy jsou definovány pomocí HTML značek a lze je vidět ve zdrojovém kódu stránky. (Vlastní definice)

Seznam použité literatury

- [1] PATTON, Ron. *Testování softwaru*. Praha: Computer Press, 2002. ISBN 9788072266364.
- [2] DRIVER, Mark, Thomas E. MURPHY, Ray VALDES, Maritess SOBEJANA, David NORTON a Nathan WILSON. Magic Quadrant for Integrated Software Quality Suites. *Gartner* [online]. 19. září 2014 [vid. 24. březen 2015]. Dostupné z: <http://www.gartner.com/technology/reprints.do?id=1-21M111D&ct=140915&st=sb>
- [3] KOSEK, Jiří. Kaskádové styly I. *Kosek.cz* [online]. 1999 [vid. 21. červen 2014]. Dostupné z: <http://www.kosek.cz/clanky/html/16.html>
- [4] BŘÍZA, Petr. Základy jazyka XPath. *Interval* [online]. 9. duben 2004 [vid. 21. červen 2014]. Dostupné z: <http://interval.cz/clanky/zaklady-jazyka-xpath/>
- [5] Platforms Supported by Selenium. *Selenium HQ: Browser Automation* [online]. [vid. 22. červen 2014]. Dostupné z: <http://docs.seleniumhq.org/about/platforms.jsp>
- [6] ROUDENSKÝ, Petr a Anna HAVLÍČKOVÁ. *Řízení kvality softwaru: průvodce testováním*. Brno: Computer Press, 2013. ISBN 9788025138168.
- [7] BEER, Armin a kolektiv. *Standard glossary of terms used in Software Testing* [online]. B.m.: International Software Testing Qualifications Board (ISTQB). 28. březen 2014 [vid. 10. červen 2014]. Dostupné z: <http://www.istqb.org/downloads/finish/20/137.html>
- [8] KANER, Cem. Exploratory Testing. In: *QAI* [online]. B.m. 17. listopad 2006 [vid. 10. červen 2014]. Dostupné z: <http://www.kaner.com/pdfs/ETatQAI.pdf>
- [9] ABRAN, Alain a kolektiv. *Guide to the Software Engineering Body of Knowledge (SWEBOK)* [online]. B.m.: The Institute of Electrical and Electronics Engineers (IEEE). 2004 [vid. 10. červen 2014]. Dostupné z: http://www.ppf.lu.lv/v.3/eduinf/files/Swebok_Ironman_June_23_2004.pdf
- [10] MÜLLER, Thomas, Debra FRIEDENBERG a kolektiv. *Certifikovaný tester: Učební osnovy pro základní stupeň* [online]. B.m.: ISTQB. 2011 [vid. 25. srpen 2014]. Dostupné z: <http://castb.org/wp-content/uploads/2013/11/ISTQB-CTFL-Syllabus-v2011-CZ-Beta1.pdf>
- [11] *Zajištění kvality programového vybavení - testování* [online]. B.m.: Jihočeská univerzita v Českých Budějovicích. [vid. 17. duben 2015]. Dostupné z: http://ecom.ef.jcu.cz/web2/download/teorie/11_zajisteni_kvality_programoveho_vybaveni.pdf
- [12] BORŮVKA, Zdeněk. *Způsoby ověření kvality aplikací a systémů (metodika, nástroje)* [online]. Praha, 2009 [vid. 25. srpen 2014]. Diplomová práce. Vysoká škola ekonomická v Praze. Dostupné z: [http://www.vse.cz/vskp/16453_zpusoby_overeni_kvality_aplikaci_a_systemu_\(metodika_nastroje\)](http://www.vse.cz/vskp/16453_zpusoby_overeni_kvality_aplikaci_a_systemu_(metodika_nastroje))

-
- [13] CHISNELL, Dana. Testování použitelnosti webu bez mýtů a předpokladů. *Interval.cz* [online]. 3. března 2010 [vid. 18. duben 2015]. Dostupné z: <https://www.interval.cz/clanky/testovani-pouzitelnosti-webu-bez-mytyu-a-predsudku/>
- [14] S, Kumar. What is Maintainability testing in software? *ISTQB Exam Certification* [online]. [vid. 7. únor 2015]. Dostupné z: <http://istqbexamcertification.com/what-is-maintainability-testing-in-software/>
- [15] S, Kumar. What is Portability testing in software? *ISTQB Exam Certification* [online]. [vid. 7. únor 2015]. Dostupné z: <http://istqbexamcertification.com/what-is-portability-testing-in-software/>
- [16] *Specifikace požadavků dle IEEE standardu* [online]. B.m.: Vysoké učení technické v Brně. 14. únor 2004 [vid. 18. duben 2015]. Dostupné z: <http://www.fit.vutbr.cz/study/courses/RPS/public/pro-vytah.html>
- [17] PAGE, Alan, Ken JOHNSTON a Bj ROLLISON. *Jak testuje software Microsoft*. Brno: Computer Press, 2009. ISBN 9788025128695.
- [18] DUSTIN, Elfriede. *Automated software testing: introduction, management, and performance*. Boston, Massachusetts, USA: Addison-Wesley, 1999. ISBN 0201432870.
- [19] FEWSTER, Mark. *Software test automation: effective use of test execution tools*. New York, London: Addison-Wesley, 1999. ISBN 0201331403.
- [20] AVASARALA, Satya. *Selenium WebDriver Practical Guide: Interactively automate web applications using Selenium WebDriver*. Birmingham, UK: Packt Pub., 2014. ISBN 9781782168850.
- [21] GUNDECHA, Unmesh. *Selenium Testing Tools Cookbook: Over 90 recipes to build, maintain, and improve test automation with Selenium WebDriver*. Birmingham, UK: Packt Publishing, 2012. ISBN 9781849515740.
- [22] VODIČKA, Petr. *Behaviour Driven Development* [online]. Praha, 2012 [vid. 8. červen 2014]. Bakalářská práce. Vysoká škola ekonomická v Praze. Dostupné z: http://www.vse.cz/vskp/32766_behaviour_driven_development
- [23] Distributed Testing - Overview. *Smart Bear* [online]. 6. červen 2014 [vid. 8. červen 2014]. Dostupné z: <http://support.smartbear.com/viewarticle/55788/>
- [24] BURNS, David. *Selenium 2 Testing Tools Beginner's Guide: Learn to use Selenium testing tools from scratch*. Birmingham, UK: Packt Publishing, 2012. ISBN 9781849518307.
- [25] GUNDECHA, Unmesh. *Instant Selenium Testing Tools Starter: A short, fast, and focused guide to Selenium Testing tools that delivers immediate results*. Birmingham, UK: Packt Publishing, 2013. ISBN 9781782165149.
- [26] GARG, Navneesh. *Test Automation using Selenium WebDriver with Java: Step by Step*
-

-
- Guide*. B.m.: Test Automation Using Selenium with Java, 2014. ISBN 0992293510.
- [27] SALUNKE, Sagar Shivaji. *Selenium Webdriver in Java: Learn With Examples*. B.m.: CreateSpace Independent Publishing Platform, 2014. ISBN 9781495450204.
- [28] SALUNKE, Sagar Shivaji. *Selenium Webdriver in C#.Net: Learn With Examples*. B.m.: CreateSpace Independent Publishing Platform, 2014. ISBN 9781495494222.
- [29] SOKOL, Martin. *Automatické testování webových aplikací* [online]. Brno, 2013 [vid. 6. červen 2014]. Diplomová práce. Masarykova Univerzita. Dostupné z: http://is.muni.cz/th/207509/fi_m/diplomova_praca.pdf
- [30] CHMURČIAK, Dávid. *Automatické testování webových aplikací* [online]. Brno, 2013 [vid. 6. červen 2014]. Diplomová práce. Masarykova Univerzita. Dostupné z: http://is.muni.cz/th/143240/fi_m/xchmurc_thesis.pdf
- [31] PIETRIK, Michal. *Automatizované testování webových aplikací* [online]. Brno, 2012 [vid. 6. červen 2014]. Diplomová práce. Masarykova Univerzita. Dostupné z: http://is.muni.cz/th/172724/fi_m/_DP_Pietrik.pdf
- [32] SKLENÁŘ, Pavel. *Testování webových aplikací za pomoci detekce změn jejího vzhledu* [online]. Praha, 2012 [vid. 6. červen 2014]. Diplomová práce. Vysoká škola ekonomická v Praze. Dostupné z: http://www.vse.cz/vskp/34828_testovani_webovych_aplikaci_za_pomoci_detekce_zmen_jejeho_vzhledu
- [33] HÚSKA, Juraj. *Automated Testing of the Component-based Web Application User Interfaces* [online]. Brno, 2012 [vid. 5. červen 2014]. Bakalářská práce. Masarykova Univerzita. Dostupné z: http://is.muni.cz/th/359345/fi_b/thesis.pdf
- [34] MAZOCH, Břetislav. *Funkční testování webových aplikací* [online]. Brno, 2012 [vid. 5. červen 2014]. Bakalářská práce. Masarykova Univerzita. Dostupné z: https://is.muni.cz/th/325233/fi_b/bmazoch-bp.pdf
- [35] CHRISTENSSON, Per. Framework Definition. *TechTerms.com* [online]. 7. březen 2013 [vid. 4. červenec 2014]. Dostupné z: <http://www.techterms.com/definition/framework>
- [36] GHAHRAI, Amir. Automation Testing Archive. *Testing Excellence* [online]. [vid. 28. březen 2015]. Dostupné z: <http://www.testingexcellence.com/category/automation-testing/>
- [37] Automated Testing: Process, Planning, Tool Selection. *Guru99* [online]. [vid. 18. duben 2015]. Dostupné z: <http://www.guru99.com/automation-testing.html>
- [38] Automated Software Testing. *Tutorialspoint* [online]. [vid. 18. duben 2015]. Dostupné z: http://www.tutorialspoint.com/software_testing_dictionary/automated_software_testing.htm

-
- [39] What is Selenium? *Selenium HQ: Browser Automation* [online]. [vid. 6. červen 2014]. Dostupné z: <http://docs.seleniumhq.org/>
- [40] Overview. *Selenium.googlecode.com* [online]. [vid. 6. červenec 2014]. Dostupné z: <http://selenium.googlecode.com/git/docs/api/java/index.html>
- [41] Selenium - Browser automation framework - Google Project Hosting. *Code.google.com* [online]. [vid. 6. červen 2014]. Dostupné z: <https://code.google.com/p/selenium/>
- [42] Newest „selenium-webdriver“ Questions - Stack Overflow. *Stack Overflow* [online]. [vid. 6. červen 2014]. Dostupné z: <http://stackoverflow.com/questions/tagged/selenium-webdriver>
- [43] Selenium Users. *Skupiny Google* [online]. [vid. 6. červen 2014]. Dostupné z: <https://groups.google.com/forum/#!forum/selenium-users>
- [44] Selenium Test Automation User Group. *LinkedIn* [online]. [vid. 6. červen 2014]. Dostupné z: <https://www.linkedin.com/groups/Selenium-Test-Automation-User-Group-961927>
- [45] Selenium 2.0 and WebDriver. *LinkedIn* [online]. [vid. 6. červen 2014]. Dostupné z: <https://www.linkedin.com/groups/Selenium-20-WebDriver-3985798>
- [46] Selenium WebDriver. *LinkedIn* [online]. [vid. 6. červen 2014]. Dostupné z: <https://www.linkedin.com/groups/Selenium-WebDriver-4067187>
- [47] RICHARDSON, Alan. Get Started With Selenium WebDriver Using Maven, IntelliJ and Java. *Selenium Simplified: automated browser testing with Selenium 2 Webdriver - made simple* [online]. [vid. 7. červen 2014]. Dostupné z: <http://seleniumsimplified.com/get-started/>
- [48] Selenium webdriver tutorials examples. *Selenium Easy: For Simple and Straightforward Tutorials* [online]. [vid. 7. červen 2014]. Dostupné z: <http://seleniueasy.com/selenium-tutorials>
- [49] Free Selenium Tutorials. *Guru 99* [online]. [vid. 7. červen 2014]. Dostupné z: <http://www.guru99.com/selenium-tutorial.html>
- [50] AKOUA, Gabiste. Selenium for Entrepreneurs: How to Use This Automation Tool. *Udemy* [online]. [vid. 28. března 2015]. Dostupné z: <https://www.udemy.com/selenium-for-entrepreneurs/>
- [51] SeleniumHQ / selenium. *GitHub* [online]. [vid. 28. března 2015]. Dostupné z: <https://github.com/SeleniumHQ/selenium>
- [52] HOODA, Ravinder Veer. *An Automation of Software Testing: A Foundation for the Future* [online]. B.m.: MNK Publication. 2012 [vid. 15. března 2015]. Dostupné z: http://www.mnkjournals.com/ijlrst_files/download/vol%201%20issue%202/401-ravinder.pdf

-
- [53] KEW, Nick. Why Validate? *W3C* [online]. [vid. 15. duben 2015]. Dostupné z: <http://validator.w3.org/docs/why.html>
- [54] BELLWARE, Scott. Behavior-Driven Development. *CODE Magazine* [online]. [vid. 15. březen 2015]. Dostupné z: <http://www.codemag.com/article/0805061>
- [55] Test Management and Test Case Management Tools. *Software Testing Tools* [online]. 6. červen 2013 [vid. 16. duben 2015]. Dostupné z: <http://www.testingtools.com/test-management/>
- [56] HELLER, Gerald. *RM Tools – what are they anyway? | The Making of Software* [online]. 4. březen 2013 [vid. 16. duben 2015]. Dostupné z: <http://makingofsoftware.com/2013/rm-tools-what-are-they-anyway>
- [57] HIDALGO, Jorge. Test Automation with Selenium WebDriver and Selenium Grid – part 3: Continuous Integration. *Dr. Macphail's trance* [online]. 2. únor 2012 [vid. 15. březen 2015]. Dostupné z: <https://deors.wordpress.com/2012/02/02/selenium-webdriver-grid-3/>
- [58] Selenium Contributors. *Selenium HQ: Browser Automation* [online]. [vid. 7. březen 2015]. Dostupné z: <http://docs.seleniumhq.org/about/contributors.jsp>
- [59] STEWART, Simon. Selenium WebDriver. *The Architecture of Open Source Applications* [online]. 2010 [vid. 9. duben 2014]. Dostupné z: <http://www.aosabook.org/en/selenium.html>
- [60] Introduction to Selenium. *Guru99* [online]. [vid. 14. březen 2015]. Dostupné z: <http://www.guru99.com/introduction-to-selenium.html>
- [61] PECINOVSKÝ, Rudolf. *Myslíme objektově v jazyku Java 5.0*. Praha: Grada, 2004. ISBN 8024709414, 9788024709413.
- [62] Selenium 1 (Selenium RC). *Selenium HQ: Browser Automation* [online]. [vid. 15. březen 2015]. Dostupné z: http://docs.seleniumhq.org/docs/05_selenium_rc.jsp
- [63] STEWART, Simon. Selenium 2.0: Out Now! *Official Selenium Blog* [online]. 8. červenec 2011 [vid. 17. květen 2014]. Dostupné z: <http://seleniumhq.wordpress.com/2011/07/08/selenium-2-0/>
- [64] STEWART, Simon a David BURNS. WebDriver. *W3C* [online]. 10. červen 2014 [vid. 10. červen 2014]. Dostupné z: <https://dvcs.w3.org/hg/webdriver/raw-file/default/webdriver-spec.html>
- [65] Selenium History. *Selenium HQ: Browser Automation* [online]. [vid. 9. duben 2014]. Dostupné z: <http://docs.seleniumhq.org/about/history.jsp>
- [66] Selenium-IDE. *Selenium HQ: Browser Automation* [online]. 14. březen 2015 [vid. 14. březen 2015]. Dostupné z: http://docs.seleniumhq.org/docs/02_selenium_ide.jsp
- [67] Selenium-Grid. *Selenium HQ: Browser Automation* [online]. 14. březen 2015 [vid. 14. březen 2015]. Dostupné z: http://docs.seleniumhq.org/docs/02_selenium_grid.jsp

-
- březen 2015]. Dostupné z: http://docs.seleniumhq.org/docs/07_selenium_grid.jsp
- [68] Selenium WebDriver. *Selenium HQ: Browser Automation* [online]. 6. červen 2014 [vid. 10. červen 2014]. Dostupné z: http://docs.seleniumhq.org/docs/03_webdriver.jsp
- [69] CHRISTENSSON, Per. API (Application Program Interface) Definition. *TechTerms.com* [online]. [vid. 4. červen 2014]. Dostupné z: <http://www.techterms.com/definition/api>
- [70] TŮMA, Jakub. RIA. *WebČesky.cz* [online]. 30. červenec 2012 [vid. 4. červen 2014]. Dostupné z: <http://www.webcesky.cz/ria/>
- [71] Download Firebug. *Firebug: Web Development Evolved* [online]. 2013 [vid. 12. duben 2014]. Dostupné z: <https://getfirebug.com/downloads/>
- [72] NYMAN, Robert. Firefinder for Firebug. *Doplňky aplikace Firefox* [online]. 27. březen 2013 [vid. 9. červen 2014]. Dostupné z: <https://addons.mozilla.org/cs/firefox/addon/firefinder-for-firebug/>
- [73] HRVATIN, John. Improved Productivity Through Internet Explorer 8 Developer Tools. *Windows Internet Explorer Engineering Team Blog* [online]. 7. březen 2008 [vid. 12. duben 2014]. Dostupné z: <http://blogs.msdn.com/b/ie/archive/2008/03/07/improved-productivity-through-internet-explorer-8-developer-tools.aspx>
- [74] Internet Explorer Developer Toolbar. *Microsoft Download Center* [online]. 6. červenec 2010 [vid. 9. červen 2014]. Dostupné z: <http://www.microsoft.com/en-us/download/details.aspx?id=18359>
- [75] BUCHALCEVOVÁ, Alena. *Metodiky budování informačních systémů*. Praha: Oeconomica, 2009. ISBN 9788024515403.
- [76] BUCHALCEVOVÁ, Alena. *Metodiky vývoje a údržby informačních systémů: kategorizace, agilní metodiky, vzory pro návrh metodiky*. Praha: Grada, 2005. ISBN 8024710757.
- [77] *Chaos Manifesto 2013* [online]. B.m.: The Standish Group International. 2013 [vid. 31. březen 2015]. Dostupné z: <http://www.versionone.com/assets/img/files/ChaosManifesto2013.pdf>
- [78] BRUCKNER, Tomáš, Alena BUCHALCEVOVÁ, Jiří VOŘÍŠEK a kolektiv. *Tvorba informačních systémů: principy, metodiky, architektury*. Praha: Grada, 2012. ISBN 9788024741536.
- [79] RUP Summary. *SUNIWE* [online]. 12. leden 2008 [vid. 25. srpen 2014]. Dostupné z: <http://projects.staffs.ac.uk/suniwe/project/rup.html>
- [80] BALDUINO, Ricardo. *Introduction to OpenUP (Open Unified Process)* [online]. 2007 [vid. 25. srpen 2014]. Dostupné z: <http://www.eclipse.org/epf/general/OpenUP.pdf>
-

-
- [81] JAMES, Michael. Scrum and Quality Assurance. *CollabNet* [online]. 24. září 2007 [vid. 25. srpen 2014]. Dostupné z: <http://blogs.collab.net/agile/scrum-and-quality-assurance>
- [82] WELLS, Don. Extreme Programming: A gentle introduction. *Extreme Programming* [online]. 8. říjen 2013 [vid. 25. srpen 2014]. Dostupné z: <http://www.extremeprogramming.org/>
- [83] ŠPLÍCHALOVÁ, Marcela. *Metodika testování webových aplikací* [online]. Praha, 2008 [vid. 31. březen 2015]. Diplomová práce. Vysoká škola ekonomická v Praze. Dostupné z: https://www.vse.cz/vskp/7634_metodika_testovani_webovych_aplikaci
- [84] HAUGHEY, Duncan. The Role of the Project Manager. *Project Smart* [online]. [vid. 14. duben 2015]. Dostupné z: <http://www.projectsart.co.uk/the-role-of-the-project-manager.php>
- [85] Role: Project Manager. *KITSCM - Rational Unified Process* [online]. [vid. 14. duben 2014]. Dostupné z: https://kitscm.vse.cz/RUP/SmallProjects/index.htm#core.base_rup/roles/rup_project_manager_363CE680.html
- [86] WILSON, Ashleigh. Systems developer. *Prospects* [online]. listopad 2013 [vid. 14. duben 2015]. Dostupné z: http://www.prospects.ac.uk/systems_developer_job_description.htm
- [87] Role: Test Analyst. *Rational Unified Process* [online]. [vid. 14. duben 2015]. Dostupné z: https://kitscm.vse.cz/RUP/SmallProjects/index.htm#core.base_rup/roles/rup_test_analyst_4637F9F0.html
- [88] HÉGARET, Philippe Le, Ray WHITMER a Lauren WOOD. Document Object Model (DOM). *W3C* [online]. 6. leden 2009 [vid. 18. duben 2015]. Dostupné z: <http://www.w3.org/DOM/>
- [89] Java SE Development Kit 8 Downloads. *Oracle* [online]. 2014 [vid. 21. červen 2014]. Dostupné z: <http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html>
- [90] WHITE, Oliver. Java Tools and Technologies Landscape for 2014. *ZeroTurnaround* [online]. 21. květen 2014 [vid. 28. březen 2015]. Dostupné z: <http://zeroturnaround.com/rebellabs/java-tools-and-technologies-landscape-for-2014/6/>
- [91] Download Latest Version of IntelliJ IDEA. *JetBrains* [online]. 26. květen 2014 [vid. 21. červen 2014]. Dostupné z: <http://www.jetbrains.com/idea/download/>
- [92] Welcome to Apache Maven. *Apache Maven Project* [online]. [vid. 22. červen 2014]. Dostupné z: <http://maven.apache.org/index.html>
- [93] BŘÍZA, Petr. Kompletní průvodce XSLT – jmenné prostory. *Interval.cz* [online]. 2. srpen 2004 [vid. 28. březen 2015]. Dostupné z:

-
- <https://www.interval.cz/clanky/kompletni-pruvodce-xslt-jmenne-prostory/>
- [94] Guide to naming conventions on groupId, artifactId and version. *Apache Maven Project* [online]. [vid. 22. červen 2014]. Dostupné z: <http://maven.apache.org/guides/mini/guide-naming-conventions.html>
- [95] Interface WebDriver. *Selenium.googlecode.com* [online]. [vid. 6. červenec 2014]. Dostupné z: <http://selenium.googlecode.com/git/docs/api/java/org/openqa/selenium/WebDriver.html>
- [96] Interface WebElement. *Selenium.googlecode.com* [online]. [vid. 6. červenec 2014]. Dostupné z: <http://selenium.googlecode.com/git/docs/api/java/org/openqa/selenium/WebElement.html>
- [97] CSS Selector Reference. *W3Schools* [online]. [vid. 19. červenec 2014]. Dostupné z: http://www.w3schools.com/cssref/css_selectors.asp
- [98] CSS Compatibility in Internet Explorer. *Microsoft Developer Network* [online]. [vid. 29. červenec 2014]. Dostupné z: [http://msdn.microsoft.com/en-us/library/hh781508\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/hh781508(v=vs.85).aspx)
- [99] KOCH, Peter-Paul. CSS selectors - desktop. *Quirksmode* [online]. 2. říjen 2013 [vid. 29. červenec 2014]. Dostupné z: <http://www.quirksmode.org/css/selectors/>
- [100] XPath Syntax. *W3Schools* [online]. [vid. 30. červenec 2014]. Dostupné z: http://www.w3schools.com/XPath/xpath_syntax.asp
- [101] XPath Axes. *W3Schools* [online]. [vid. 30. červenec 2014]. Dostupné z: http://www.w3schools.com/XPath/xpath_axes.asp
- [102] XPath, XQuery, and XSLT Functions. *W3Schools* [online]. [vid. 1. srpen 2014]. Dostupné z: http://www.w3schools.com/xpath/xpath_functions.asp
- [103] Stale Element Reference Exception. *Selenium HQ: Browser Automation* [online]. [vid. 10. srpen 2014]. Dostupné z: http://docs.seleniumhq.org/exceptions/stale_element_reference.jsp
- [104] Class Select. *Selenium.googlecode.com* [online]. [vid. 21. srpen 2014]. Dostupné z: <https://selenium.googlecode.com/git/docs/api/java/org/openqa/selenium/support/ui/Select.html>
- [105] HUME, Dean. Selenium Webdriver - Wait for an element to load. *Coding Tips & Tricks* [online]. 27. říjen 2011 [vid. 28. březen 2015]. Dostupné z: <http://deanhume.com/home/blogpost/selenium-webdriver---wait-for-an-element-to-load/64>
- [106] SNÍŽEK, Martin. AJAX - kde jsou hranice? *Snizekweb.cz: Martin Snížek píše o webu* [online]. 13. září 2005 [vid. 22. srpen 2014]. Dostupné z:
-

-
- <http://www.snizekweb.cz/clanky/ajax-kde-jsou-hranice/>
- [107] Class ExpectedConditions. *Selenium.googlecode.com* [online]. [vid. 23. srpen 2014]. Dostupné z: <https://selenium.googlecode.com/git/docs/api/java/org/openqa/selenium/support/ui/ExpectedConditions.html>
- [108] WebDriver: Advanced Usage. *Selenium HQ: Browser Automation* [online]. 23. srpen 2014 [vid. 26. srpen 2014]. Dostupné z: http://docs.seleniumhq.org/docs/04_webdriver_advanced.jsp
- [109] Interface ExpectedCondition<T>. *Selenium.googlecode.com* [online]. [vid. 24. srpen 2014]. Dostupné z: <https://selenium.googlecode.com/git/docs/api/java/org/openqa/selenium/support/ui/ExpectedCondition.html>
- [110] Class WebDriverWait. *Selenium.googlecode.com* [online]. [vid. 26. srpen 2014]. Dostupné z: <https://selenium.googlecode.com/git/docs/api/java/org/openqa/selenium/support/ui/WebDriverWait.html>
- [111] Class FluentWait<T>. *Selenium.googlecode.com* [online]. [vid. 24. srpen 2014]. Dostupné z: <http://selenium.googlecode.com/git/docs/api/java/org/openqa/selenium/support/ui/FluentWait.html>
- [112] Class Actions. *Selenium.googlecode.com* [online]. [vid. 28. březen 2015]. Dostupné z: <https://selenium.googlecode.com/git/docs/api/java/org/openqa/selenium/interactions/Actions.html#Actions-org.openqa.selenium.interactions.Keyboard->
- [113] LUKE.SEMERAU. AdvancedUserInteractions. *Code.google.com* [online]. 14. leden 2013 [vid. 28. březen 2015]. Dostupné z: <https://code.google.com/p/selenium/wiki/AdvancedUserInteractions>
- [114] Set browser width and height in Selenium Webdriver. *Selenium Easy* [online]. [vid. 8. březen 2015]. Dostupné z: <http://seleniumeasy.com/selenium-tutorials/set-browser-width-and-height-in-selenium-webdriver>
- [115] MIA, Hiro. How can switch between browser tabs in selenium webdriver using java? *Hiro Mia* [online]. leden 2015 [vid. 8. březen 2015]. Dostupné z: <http://hiromia.blogspot.cz/2015/01/how-can-switch-between-browser-tabs-in.html>
- [116] MIA, Hiro. How to open and close tab in selenium webdriver using java? *Hiro Mia* [online]. leden 2015 [vid. 8. březen 2015]. Dostupné z: <http://hiromia.blogspot.cz/2015/01/how-to-open-and-close-new-tab-in.html>
- [117] Taking Screenshot using Webdriver. *Selenium Easy* [online]. [vid. 28. březen 2015]. Dostupné z: <http://seleniumeasy.com/selenium-tutorials/take-screenshot-with-selenium-webdriver>

- [118] DIJKSTRA, Bas. Creating a video capture of your Selenium tests using Monte Screen Recorder. *On Test Automation* [online]. 9. únor 2015 [vid. 5. duben 2015]. Dostupné z: <http://www.ontestautomation.com/creating-a-video-capture-of-your-selenium-tests-using-monte-screen-recorder/>
- [119] LACKÓ, István. Recording videos of tests in Java. *WeDoQA Blog* [online]. 13. leden 2014 [vid. 5. duben 2015]. Dostupné z: <http://blog.wedoqa.com/2014/01/recording-videos-of-tests-in-java/>
- [120] RANDELSHOFER, Werner. *Constant Field Values* [online]. 6. leden 2013 [vid. 5. duben 2015]. Dostupné z: <http://www.randelshofer.ch/monte/javadoc/constant-values.html>
- [121] Continuous Integration. *BrowserStack* [online]. [vid. 5. duben 2015]. Dostupné z: <https://www.browserstack.com/automate/continuous-integration>
- [122] BALEA, Ciprian. How to Use Selenium in Continuous Integration Testing. *3Pillar Global* [online]. 2014 [vid. 5. duben 2015]. Dostupné z: <http://www.3pillarglobal.com/insights/how-to-use-selenium-in-continuous-integration-testing>
- [123] GUNDECHA, Unmesh. *Integration with Other Tools* [online]. B.m.: Packt Publishing. 2012 [vid. 5. duben 2015]. Dostupné z: https://www.packtpub.com/sites/default/files/downloads/Integration_with_Other_Tools.pdf

Seznam obrázků a tabulek

Seznam obrázků

Obrázek 1: Princip fungování nástrojů pro automatizaci testování (Zdroj: [19 s. 311], překresleno a přeloženo autorkou)	30
Obrázek 2: Architektura nástroje Selenium RC (Zdroj: [62])	33
Obrázek 3: Ukázka okna aplikace Selenium IDE	35
Obrázek 4: Ikona a panel aplikace Firebug	38
Obrázek 5: Firebug - kopírování cesty k elementu	39
Obrázek 6: Firebug - ověření CSS selektoru	40
Obrázek 7: Firebug - ověření XPath selektoru	41
Obrázek 8: Záložka doplňku Firefinder v panelu aplikace Firebug a nalezený element	42
Obrázek 9: Internet Explorer Developer Tools - ověřování CSS selektoru	43
Obrázek 10: Panel aplikace Chrome Developer Tools	44
Obrázek 11: Google Chrome Developer Tools - ověření XPath selektoru	45
Obrázek 12: Fáze a disciplíny RUP (Zdroj: [79])	47
Obrázek 13: OpenUP – přehled fází projektu a jejich cílů (Zdroj: [80])	47
Obrázek 14: Role testování ve vodopádovém modelu vývoje softwaru (Zdroj: [75 s. 48], překresleno autorkou)	49
Obrázek 15: V-model (Zdroj: [19 s. 7], přeloženo a překresleno autorkou)	49
Obrázek 16: Workflow automatizace testování dle metodiky ATLM (Zdroj: [16 s. 9], překresleno a přeloženo autorkou)	53
Obrázek 17: Detail procesu rozhodování o automatizaci testování (Zdroj: autorka, inspirováno [18 s. 31])	54
Obrázek 18: Detail procesu výběru testovacího nástroje (Zdroj: autorka, inspirováno [18 s. 69])	58
Obrázek 19: Detail procesu přípravy automatizace testování (Zdroj: autorka, inspirováno [18 s. 109–110])	62
Obrázek 20: Detail procesu plánování, návrhu a vytváření testů (Zdroj: autorka, inspirováno [18 s. 147–346])	66
Obrázek 21: Detail procesu řízení a provádění testů (Zdroj: autorka, inspirováno [18 s. 349–378])	74
Obrázek 22: Detail procesu kontroly a vyhodnocování procesu testování (Zdroj: autorka, inspirováno [18 s. 379–402])	77
Obrázek 23: Vztah ATLM k procesu vývoje SW (Zdroj: [18 s. 15], přeloženo a překresleno autorkou)	79
Obrázek 24: Vytvoření nového Maven projektu v IntelliJ IDEA	106
Obrázek 25: Nově vytvořený projekt v IntelliJ IDEA	107
Obrázek 26: Nastavení chráněného režimu v prohlížeči Internet Explorer	110
Obrázek 27: Volba názvu knihovny pro nástroj Monte Screen Recorder	147

Seznam tabulek

Tabulka 1: Srovnání manuálního a automatizovaného testování (Zdroj: autorka)	23
Tabulka 2: Sada grafických prvků pro popis metodiky (Zdroj: [83 s. 14], překresleno a upraveno autorkou)	52
Tabulka 3: Terminologický slovník	89
Tabulka 4: Přehled CSS selektorů využitelných při automatizovaném testování (Zdroj: [97])	118
Tabulka 5: Přehled XPath selektorů využitelných při automatizovaném testování (Zdroj: [100] [101])	122

Rejstřík

Ad hoc testování.....	10	Objemové testy.....	8
Akceptační testování.....	50, 67	OpenUP.....	47
Bezpečnostní testování.....	67	Plán testování.....	66
CSS.....	4, 38, 39, 45, 89, 117	Quality Gates.....	63
Dýmové testování.....	9	Regresní testování.....	21, 67
Dynamické testování.....	7	Remote Control.....	21
Exploratorní testování.....	10	Retestování.....	21
Extrémní programování.....	48	RUP.....	46
Firebug.....	38	Scrum.....	48
Firefinder.....	42	Selenium Core.....	31, 32, 33, 34
Funkcionální testování.....	7, 50, 59, 67	Selenium Grid.....	21, 36
Chrome Developer Tools.....	44	Selenium IDE.....	17, 18, 19, 21, 25, 26, 30, 34, 35, 36
Integrační testování.....	50	Selenium RC.....	15, 19, 31, 32, 33, 36
Internet Explorer Developer Tools.....	43	Selenium RC Server.....	33
JavaScript Test Runner.....	30, 31, 34	Selenium Remote Control.....	33
Jednotkové testování.....	50, 67	Softwarová chyba.....	11
Konfirmační testování.....	67	Specifikace požadavků.....	11
Metodika.....	46	Statické testování.....	7
Nástroje pro akceptační testování.....	25	Stress testy.....	8
Nástroje pro analýzu pokrytí kódu testy.....	29	Strukturální testování.....	7
Nástroje pro dýmové (smoke) testování.....	26	Systémové testování.....	50
Nástroje pro funkcionální regresní testování.....	26	Testovací případ.....	12
Nástroje pro integrační testování.....	25	Testování bezpečnosti.....	8
Nástroje pro jednotkové testování.....	24	Testování bílé skříňky.....	7
Nástroje pro sledování stavu zaznamenaných chyb.....	27	Testování černé skříňky.....	7
Nástroje pro správu požadavků.....	28	testování kompatibility s různými prohlížeči.....	10
Nástroje pro správu testovacích případů.....	28	Testování konfirmační.....	9
Nástroje pro systémové testování.....	25	Testování použitelnosti.....	9
Nástroje pro testování bezpečnosti.....	27	Testování přenositelnosti.....	9
Nástroje pro testování grafického uživatelského rozhraní.....	24	Testování regresní.....	9
Nástroje pro testování kompatibility webové aplikace s různými internetovými prohlížeči.....	26	Testování spolehlivosti.....	9
Nástroje pro testování řízené kódem.....	24	Testování šedé skříňky.....	8
Nástroje pro testování výkonnosti.....	26	Testování udržitelnosti.....	9
Nástroje pro validaci webových stránek.....	27	Úrovně testování.....	48
Nástroje pro vývoj řízený požadavky na chování.....	27	V-model.....	48, 49, 50
Nefunkcionální testování.....	7, 8, 25, 50	Vodopádový model vývoje softwaru.....	48
		Výkonnostní testování.....	8, 67
		WebDriver API.....	15, 32, 111, 112
		XPath.....	39, 40, 41, 45
		Zátěžové testy.....	8

Příloha A: Uživatelská příručka k nástroji Selenium WebDriver

Posledním z dílčích cílů této diplomové práce bylo vytvoření příručky k nástroji Selenium WebDriver. Vzhledem k omezenému rozsahu práce jsem se rozhodla tuto část přesunout do přílohy, což ovšem neznamená, že by tato část byla méně důležitá. Pro úspěšnou realizaci automatizace testování je totiž potřeba, aby byl celý proces nejen dobře řízen, ale aby také testy samotné byly kvalitní a udržitelné, a tedy plnily svou funkci. První podmínku úspěchu lze naplnit prováděním procesu automatizace podle kapitoly 4 Metodika pro automatizaci testování webových aplikací a druhá podmínka je splněna, pokud tester zodpovědný za automatizaci testů dobře zvládá práci s daným nástrojem. A s tím by měla pomoci právě tato příručka.

Tato kapitola popisuje práci s nástrojem Selenium WebDriver při vytváření automatizovaných testů webových aplikací od vytvoření prvního testovacího skriptu až po pokročilé funkce a doporučení k architektuře testů. Kód automatizovaných testů je přitom psán v programovacím jazyku Java s využitím frameworku JUnit.

A.1 Jak začít

Ještě před napsáním první řádky kódu automatizovaného testu je potřeba si stáhnout a nainstalovat několik softwarových nástrojů a připravit vývojové prostředí, ve kterém budou automatizované testy vytvářeny a spouštěny. Pro vytváření automatizovaných testů jsou v této diplomové práci využity následující softwarové balíky a nástroje:

1. Java Development Kit (Standard Edition)
2. IntelliJ IDEA Community Edition
3. Maven
4. JUnit
5. Selenium WebDriver

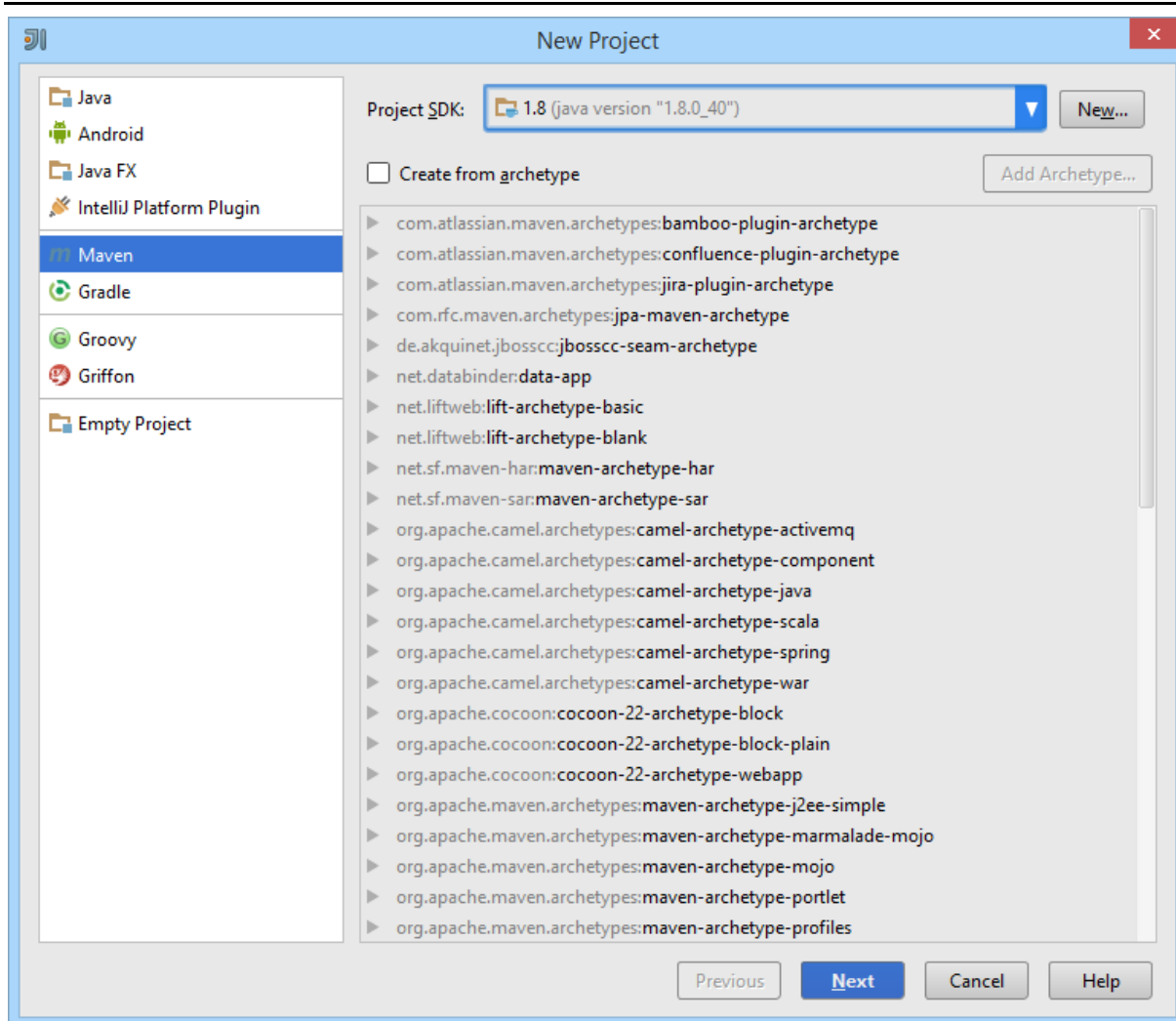
Vzhledem k tomu, že pro vytváření automatizovaných testů s využitím nástroje Selenium WebDriver byl v této diplomové práci zvolen programovací jazyk Java, je nutné začít stažením balíčku Java Development Kit¹⁶ z oficiálních stránek společnosti Oracle [89]. Pro účely vytváření automatizovaných testů postačuje verze SE (Standard Edition). Před stažením je potřeba odsouhlasit licenční podmínky společnosti Oracle (*Accept License Agreement*) a poté již lze stáhnout JDK (např. *Java SE Development Kit 8u40*) pro vybraný operační systém a nainstalovat jej.

Pro komfortní práci se zdrojovým kódem automatizovaných testů se standardně využívá některého z integrovaných vývojových prostředí. V této diplomové práci jsem zvolila IntelliJ IDEA Community Edition, neboť jde o jedno z nejrozšířenějších integrovaných vývojových prostředí pro jazyk Java [90] a navíc je dostupné zdarma. Stáhnout jej lze na oficiálních stránkách společnosti JetBrains, která tento nástroj vyvíjí. [91] Pokud však má čtenář zkušenost s jiným vývojovým prostředím, např. NetBeans nebo Eclipse, může jej samozřejmě využít namísto IntelliJ IDEA. Následující postup je popisován v prostředí IntelliJ IDEA, analogicky jej však lze aplikovat i v jiných vývojových prostředích.

Po spuštění vývojového prostředí uživatel vytvoří nový projekt (*Create New Project*) a z nabídky zvolí možnost *Maven*¹⁷. Na zobrazené stránce stiskne tlačítko *New* pro přidání reference na SDK (v tomto případě JDK, neboť je využit jazyk Java). Poté je potřeba zvolit cestu k nainstalovanému JDK, typicky *C:\Program Files\Java\jdk1.8.0_40* (číslo verze samozřejmě odpovídá zvolené instalaci), a potvrdit tlačítkem *OK*. Následně uživatel ponechá nezaškrtnuté políčko *Create from archetype* a stiskne tlačítko *Next*.

¹⁶ Java Development Kit (JDK) – sada nástrojů (běhové prostředí Javy, překladač, generátor dokumentace, ladící program a další.) pro vývoj aplikací v programovacím jazyku Java. Obecně se této sadě říká SDK (Software Development Kit), JDK je označení pouze pro Javu. [61 s. 38]

¹⁷ Apache Maven – nástroj pro řízení softwarového projektu. Umožňuje snadno sestavovat projekt (angl. build), spravovat závislosti a zásuvné moduly (angl. plugins) používané v projektu a vytvářet reporty a dokumentaci projektu. [92]



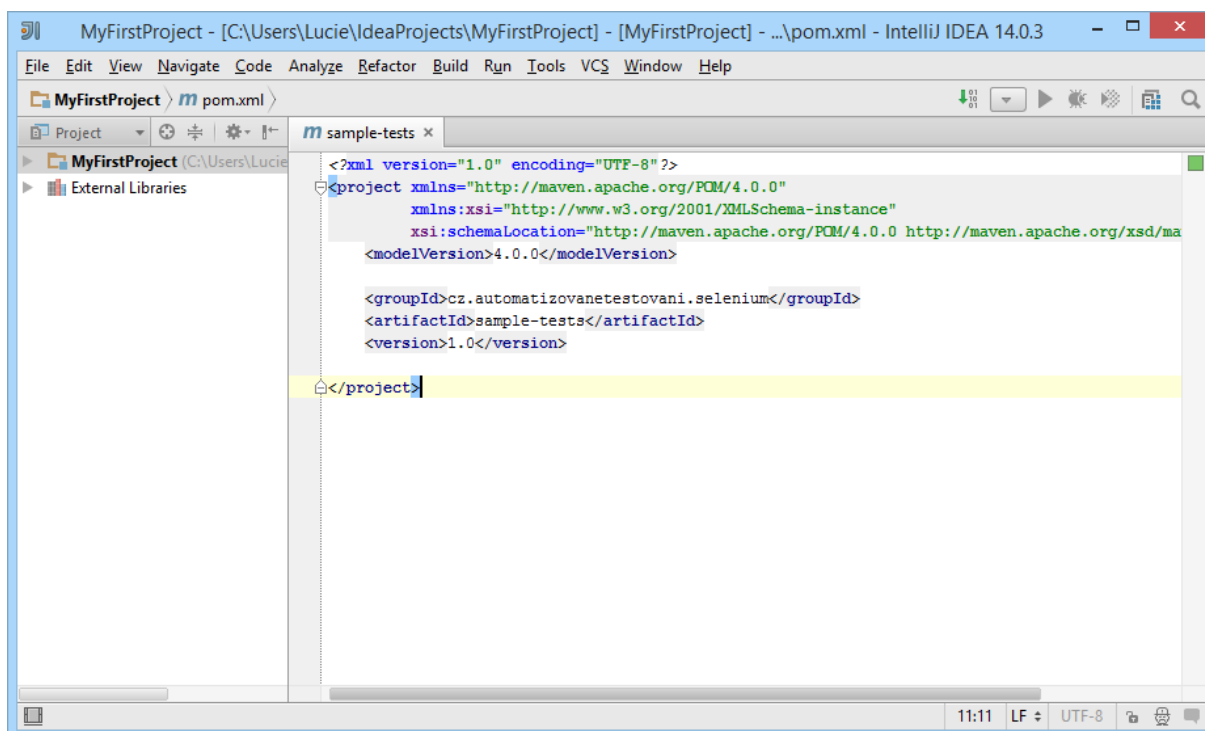
Obrázek 24: Vytvoření nového Maven projektu v IntelliJ IDEA

Na další stránce je dotázán na *GroupId*, *ArtifactId* a *Version*. *GroupId* představuje unikátní identifikátor projektu, přičemž vytváří tzv. jmenný prostor (naming schema)¹⁸.

- Pro *GroupId* je vhodné zvolit název webové domény, kterou uživatel (popř. společnost, pro kterou pracuje) vlastní, a to v obráceném tvaru – nejprve název domény nejvyššího řádu, poté název domény 2. řádu a poté případně další řády (které mohou být pouze pro účely vytvoření jmenného prostoru testů, reálně existovat nemusí), např. *cz.automatizovanetestovani.selenium*.
- *ArtifactId* představuje jméno *jar* souboru, který má být vytvořen, např. *sample-tests*.
- *Version* pak označuje verzi; v případě nového projektu tedy bude mít hodnotu *1.0*. [94]

¹⁸ Jmenný prostor (angl. naming schema) - množina jmen, které jsou si významově blízké. Jmenný prostor umožňuje elementy či atributy v daném projektu zařadit do jedné společné skupiny, obvykle definované pomocí URL adresy, díky čemuž je možné je jednoznačně identifikovat i v celosvětovém měřítku a nedochází tak ke konfliktu jejich názvů. [93]

V dalším kroku je už jen potřeba zvolit jméno projektu, např. *MyFirstProject*, stisknout tlačítko *Finish* a počkat, až IntelliJ IDEA vytvoří projekt.



Obrázek 25: Nově vytvořený projekt v IntelliJ IDEA

Po vytvoření projektu se otevře soubor pom.xml, kde jsou uvedeny základní informace o projektu, které byly uživatelem zadány v předchozích krocích. V pravém horním rohu se zobrazuje upozornění „Maven projects need to be imported“ s možnostmi *Import Changes* a *Enable Auto-Import*. Kliknutím na možnost *Enable Auto-Import* budou všechny závislosti projektu, které budou v tomto souboru uvedeny, automaticky naimportovány a spravovány, což představuje značné usnadnění práce s externími knihovnami, frameworky atp.

Nyní již zbývá jen doplnit závislosti projektu na externích knihovnách, které budou pro vytváření automatizovaných testů potřeba: JUnit a Selenium WebDriver. Pro tento účel je potřeba do souboru pom.xml doplnit element `<dependencies>` se dvěma vnořenými elementy `<dependency>`, ve kterých budou uvedeny informace o výše uvedených knihovnách. Kód souboru pom.xml pak bude vypadat přibližně takto:

Výpis 2: Zdrojový kód souboru pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>cz.automatizovanetestovani.selenium</groupId>
  <artifactId>sample-tests</artifactId>
```

```
<version>1.0</version>

<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.seleniumhq.selenium</groupId>
    <artifactId>selenium-java</artifactId>
    <version>2.45.0</version>
  </dependency>
</dependencies>
</project>
```

Verze knihoven (např. JUnit 4.12) lze samozřejmě jednoduše změnit pouhým přepsáním čísla verze (poslední verzi frameworku/knihovny lze zjistit na oficiálních webových stránkách projektu) a Maven už se sám postará o stažení potřebných souborů z internetu.

Pak už stačí jen nechat Maven automaticky stáhnout vybrané externí knihovny a vše je připraveno pro napsání prvního automatizovaného testu s využitím nástroje Selenium WebDriver.

V adresáři *MyFirstProject/src/test* se nachází složka *java*, která je připravena pro testovací třídy. Uživatel klikne pravým tlačítkem myši na složku *java* a vybere možnost *New* → *Java Class*. Poté zvolí libovolný název testovací třídy, např. *MyFirstSeleniumTest*, a potvrdí tlačítkem *OK*.

Do nově vytvořené testovací třídy lze napsat první testovací metodu. Pro úvod do fungování nástroje Selenium WebDriver jsem zvolila velice jednoduchý test, který otevře webovou stránku na URL adrese www.vse.cz, zkontroluje její titulek a poté prohlížeč zavře.

Kód této jednoduché testovací metody je následující:

Výpis 3: Zdrojový kód jednoduché testovací metody

```
@Test
public void basicTest(){
    WebDriver driver = new FirefoxDriver(); //spuštění prohlížeče Firefox
    driver.get("http://www.vse.cz"); //otevření webové stránky na dané URL adrese
    Assert.assertTrue(driver.getTitle().contains("Vysoká škola ekonomická
v Praze")); //kontrola, zda hodnota elementu Title obsahuje očekávaný text
    driver.quit(); //zavření okna prohlížeče
}
```

Pro správné fungování výše uvedených příkazů je samozřejmě potřeba nainportovat do testovací třídy odpovídající knihovny, pokud to již integrované vývojové prostředí neprovedlo automaticky:

Výpis 4: Seznam importovaných balíčků

```
import org.junit.Assert;
import org.junit.Test;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.firefox.FirefoxDriver;
```

Poté stačí jen kliknout pravým tlačítkem myši na název testovací metody a zvolit možnost *Run 'basicTest()'*. Vývojové prostředí projekt sestaví a spustí zvolený test. V příštích několika sekundách se otevře okno prohlížeče Firefox, provedou se výše uvedené kroky a nakonec se okno prohlížeče uzavře. V okně vývojového prostředí je test označen zelenou barvou signalizující, že proběhl úspěšně a bez chyb.

Jak jistě uživatel postřehl, po sestavení proběhl test velice rychle a dokázal tak zjistit přítomnost a ověřit správnost hodnoty elementu dokonce rychleji, než by to dokázal sám uživatel. Právě tato rychlost je hlavním přínosem automatizovaných testů.

A.1.1 Spouštění testů v prohlížeči Internet Explorer

Prohlížeč Internet Explorer má v tomto ohledu na rozdíl od ostatních podporovaných prohlížečů svá specifika. Pro hladký průběh testů je potřeba nejprve provést tři kroky:

1. Stáhnout soubor IEDriverServer.exe, umístit jej na disk a v kódu automatizovaných testů k němu uvést cestu;
2. Nastavit chráněný režim ve všech bezpečnostních zónách prohlížeče Internet Explorer;
3. Nastavit velikost stránky v prohlížeči Internet Explorer na 100 %.

Na rozdíl od jednoduchého vytvoření instance v případě prohlížeče Firefox je pro spouštění prohlížeče Internet Explorer nejprve potřeba stáhnout soubor IEDriverServer.exe (k dispozici na oficiálních stránkách projektu Selenium – <http://www.seleniumhq.org/download/>), uložit jej na disk a v kódu automatizovaných testů k němu ještě před vytvořením první instance prohlížeče Internet Explorer specifikovat cestu, např.: [20 s. 97]

```
public class MyFirstSeleniumTest {
    @Test
    public void basicTest() {
        System.setProperty("webdriver.ie.driver",
"C:\\Users\\<username>\\Documents\\IEDriverServer.exe"); //specifikace cesty
k souboru IEDriverServer.exe - nahraďte skutečnou cestou k souboru na vašem disku
        WebDriver driver = new InternetExplorerDriver(); //spuštění prohlížeče
Internet Explorer
    }
}
```

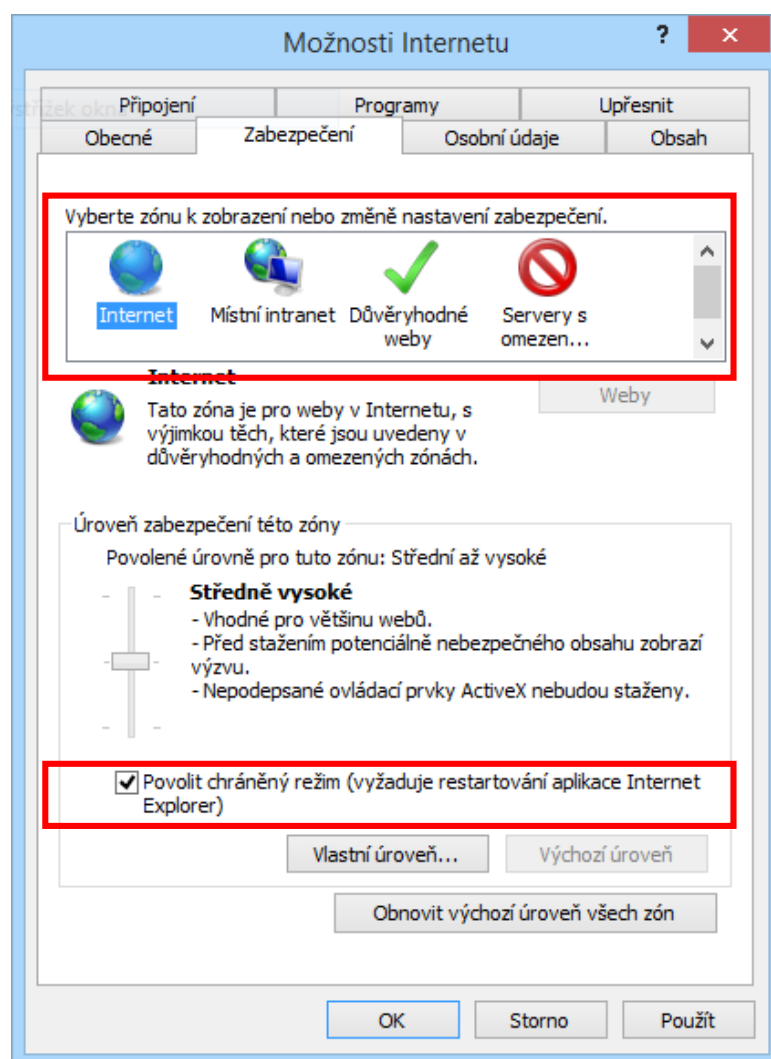
```

//další kroky testu
driver.quit();
}
}

```

Cestu C:\\Users\\<username>\\Documents\\IEDriverServer.exe je samozřejmě nutné nahradit cestou, na které se soubor IEDriverServer.exe skutečně nachází.

Následně je nutné nastavit v prohlížeči Internet Explorer pro všechny zóny zabezpečení tzv. chráněný režim. [20 s. 99] Postup je následující: otevřete prohlížeč Internet Explorer, vpravo nahoře stisknete možnost *Nástroje*, poté vyberte *Možnosti Internetu* a na záložce *Zabezpečení* projděte všechny čtyři zóny a u všech zaškrtněte možnost *Povolit chráněný režim*.



Obrázek 26: Nastavení chráněného režimu v prohlížeči Internet Explorer

Jako poslední krok je potřeba nastavit velikost stránky na základní hodnotu, tedy 100 % - v okně prohlížeče Internet Explorer klikněte na možnost *Stránka*, vyberte položku *Velikost* a následně zvolte 100 %.

Po těchto opatřeních už by měl být WebDriver schopen spustit instanci prohlížeče Internet Explorer a provést v něm další kroky testu.

A nyní, když už bylo přiblíženo základní nastavení a obsah testů, je možné se již začít věnovat dalším funkcím, kterými Selenium WebDriver disponuje. V první řadě jsou představeny různé možnosti vyhledávání elementů na webové stránce. Následuje výklad možností práce s nalezenými elementy, přiblížení různých způsobů řízení průběhu testů, pokročilých funkcí nástroje Selenium WebDriver a také neméně důležité architektury testů a možností integrace s dalšími nástroji.

A.2 WebDriver API

WebDriver API lze charakterizovat jako seznam tříd a metod nástroje Selenium WebDriver, které lze v kódu automatizovaného testu využít. Hlavními dvěma objekty jsou *WebDriver* a *WebElement*. Zatímco *WebDriver* reprezentuje webový prohlížeč, *WebElement* představuje element na stránce, se kterým lze dále pracovat (např. na něj kliknout). [24 s. 66]

S objekty *WebDriver* a *WebElement* se bude čtenář při vytváření automatizovaných testů střídat velice často, je tedy vhodné si představit jejich metody. Nejčastěji používanými metodami rozhraní *WebDriver* jsou: [95]

- *get(String url)* – otevře webovou stránku na zadané URL adrese.
- *close()* – zavře aktuálně otevřené okno prohlížeče. Pokud jde o jediné otevřené okno, pak zavře celý prohlížeč.
- *quit()* – zavře všechna okna prohlížeče.
- *findElement(By by)* – vyhledá element podle zadaných kritérií (by).
- *findElements(By by)* – vyhledá skupinu elementů podle zadaných kritérií (by).
- *getTitle()* – vrátí hodnotu elementu title webové stránky.
- *getPageSource()* – vrátí zdrojový kód naposledy načtené webové stránky.
- *getCurrentUrl()* – vrátí URL adresu webové stránky, která je právě zobrazena.
- *manage()* – umožňuje spravovat některé možnosti, které internetový prohlížeč nabízí – např. příkaz `manage().deleteAllCookies()`, který vymaže všechny uložené cookies nebo `manage().timeouts().pageLoadTimeout(10000, TimeUnit.SECONDS)`, který nastaví vypršení časového limitu načítání stránky na 10 sekund.
- *navigate()* – umožňuje přístup k historii prohlížeče, znovunačtení stránky nebo přechod na zadanou URL adresu. Často využívané jsou příkazy `navigate().back()`, který

simuluje kliknutí na tlačítko *Zpět* v internetovém prohlížeči a načte tak předchozí URL adresu, nebo naopak `navigate().forward()`, který přejde na následující stránku z historie procházení stránek. Dále je k dispozici metoda `navigate().refresh()` vyvolávající znovunačtení aktuální stránky nebo `navigate().to(String url)`, který umožňuje načíst zadanou URL adresu v aktuálně zobrazeném okně prohlížeče.

Výše uvedený výčet zmiňuje pouze často používané metody, kompletní seznam lze nalézt v oficiální dokumentaci rozhraní *WebDriver* [95].

Rozhraní *WebElement* pak také jmenuje celou řadu metod. Nejčastěji volanými metodami pracujícími s konkrétním elementem jsou: [96]

- `click()` – klikne na daný element.
- `sendKeys(CharSequence value)` – vloží hodnotu do vybraného vstupního pole.
- `submit()` – pokud je element formulářem nebo se uvnitř nějakého formuláře nachází, pak lze příkazem `submit()` tento formulář odeslat.
- `getText()` – vrátí zobrazenou textovou hodnotu elementu (včetně textové hodnoty jeho případných potomků).
- `getAttribute(String name)` – vrací hodnotu vybraného atributu daného elementu.
- `isEnabled()`, `isDisplayed()`, `isSelected()` – metody zjišťující, zda je element aktivní (lze na něj kliknout), zobrazený nebo označený. Vrací hodnotu *boolean*.
- `findElement(By by)`, popř. `findElements(By by)` – umožňují vyhledat element, resp. skupinu elementů podle zadaných kritérií (*by*) v rámci vybraného nadřazeného elementu.

Opět jde pouze o výběr nejpoužívanějších metod, kompletní výčet metod lze nalézt v oficiální dokumentaci rozhraní *WebElement* [96].

Aby byla zajištěna dokonalá souhra nástroje Selenium WebDriver s podporovaným internetovým prohlížečem, existuje pro každý z nich speciální implementace *WebDriver API* – *Firefox Driver*, *Internet Explorer Driver*, *Chrome Driver*, *Opera Driver*, *HtmlUnit Driver*, *Android Driver* a *iOS Driver*. [68] Mimoto lze vytvořit vlastní implementaci rozhraní *WebDriver* a rozšířit tak obsah předdefinovaných metod o další funkce.

A.3 Vyhledání elementů na webové stránce

Pro práci s elementem je potřeba v první řadě určit, jakým způsobem a kde má aplikace daný element hledat. Až poté je možné začít s elementem pracovat – např. kliknout na tlačítko, vepsat text do vstupního pole nebo ověřit správnost nadpisu webové stránky. Způsobů, jak identifikovat element na stránce, existuje hned několik, a často záleží jen na uvážení testera, který z nich v daném případě využije.

Způsoby, jak identifikovat element na webové stránce, jsou následující: [21 s. 14–16]

- podle atributu *id*,
- podle atributu *name*,
- podle atributu *class*,
- podle značky elementu (angl. tag name),
- pomocí textu odkazu,
- pomocí CSS,
- pomocí XPath,
- pomocí JQuery selektorů.

Výše uvedenými způsoby lze samozřejmě nalézt nejen jednotlivý element, ale také celou skupinu elementů. Pro nalezení jediného elementu se využívá metoda *findElement(By by)*, která vrací instanci třídy *WebElement*, popř. vyhodí výjimku *NoSuchElementException*, pokud žádný element zadaným kritériím neodpovídá. [95]

Výpis 5: Ukázka vyhledání jednoho elementu

```
WebElement vyhledavaciTlacitko = driver.findElement(By.id("searchButton"));
```

Při vyhledávání elementů je potřeba mít na paměti, že metoda *findElement(By by)* prochází objektový model dokumentu¹⁹ a vrací odkaz na první element, který odpovídá kritériím vyhledávání. Pokud tedy kritéria vyhledávání splňuje více elementů na dané webové stránce, je potřeba se ujistit, že se hledaný element nachází v objektovém modelu dokumentu skutečně na prvním místě. K tomuto účelu lze využít podpůrné nástroje pro zkoumání struktury webové stránky představené v kapitole 3.4.

¹⁹ Objektový model dokumentu (Document Object Model, DOM) – rozhraní nezávislé na platformě ani programovacím jazyku, které popisuje strukturu dokumentu, v tomto případě webové stránky. Umožňuje programům a skriptům dynamicky přistupovat k dokumentu a upravovat jeho obsah, strukturu a styly (vzhled). Dokument je poté zpracován a výsledek je zobrazen v internetovém prohlížeči. [88]

Naopak pokud je cílem nalézt více elementů, které odpovídají daným kritériím vyhledávání, pak je nasnadě využít metodu `findElements(By by)`, která vrací seznam instancí třídy `WebElement`. Pokud zadaným kritériím vyhledávání žádný element neodpovídá, vrátí prázdný seznam. [95]

Výpis 6: Ukázka vyhledání více elementů najednou

```
java.util.List<WebElement> tlacitka = driver.findElement(By.tagName("button"));
```

Jak je z výše uvedeného popisu zřejmé, tyto dvě metody se neliší jen tím, kolik elementů dokáží vrátit, ale i tím, co se stane v případě, že se hledaný element na stránce nenachází. Zatímco metoda `findElement(By by)` vyhodí výjimku a provádění testovací metody je tak ukončeno s výsledkem `Test failed`, metoda `findElements(By by)` pouze vrátí prázdný seznam a nijak nebrání dalšímu provádění testu. Tato vlastnost je užitečná zejména v případě, kdy chce tester ověřit přítomnost nějakého elementu, který být na webové stránce může, ale i nemusí, a na základě této informace řídit průběh testu. Kód v takovém případě může vypadat následovně:

Výpis 7: Ověřování přítomnosti elementu na stránce

```
if (findElements(By.id("searchButton")).isEmpty()) {  
    //kód, který se má provést, pokud element není na stránce přítomen  
} else {  
    //kód, který se má provést, pokud je element na stránce přítomen  
}
```

A.3.1 Vyhledávání elementů pomocí atributu id

V programátorské praxi je dobrým zvykem každému elementu přiřazovat atribut `id`, který je v rámci jedné webové stránky unikátní a s jeho pomocí tak lze prvek zcela přesně identifikovat, i když se jeho umístění na stránce změní. Pokud tedy hledaný prvek disponuje atributem `id`, pak je nejjednodušším způsobem vyhledání elementu pomocí `id`. [21 s. 16]

Výpis 8: Ukázka vyhledání elementu pomocí atributu id

```
WebElement vyhledavaciTlacitko = driver.findElement(By.id("searchButton"));
```

Problém však může nastat ve chvíli, kdy existuje v testované aplikaci několik elementů se stejným atributem `id`. Ačkoli se vyskytují na různých stránkách (podle výše uvedeného pravidla, že prvek s daným atributem `id` může být na stránce pouze jeden), může tento fakt při provádění testů způsobovat problémy – například pokud se některý z kroků testu neprovede správně (ať už z důvodu změny v testované aplikaci nebo špatně napsaného testu) a pohled se nachází na jiné než očekávané webové stránce a zároveň se na této stránce shodou okolností nachází element s daným `id`, WebDriver jej použije a test se tak může vydat zcela jiným směrem, než bylo původně zamýšleno. Z toho důvodu doporučuji doplnit identifikaci o více údajů – např. `id` nadřazeného elementu `div`, jak je specifikováno v následujícím textu popisujícím metody vyhledávání elementů na stránce pomocí CSS a XPath.

Pozor také na automaticky generované atributy *id*, které se mohou změnit vždy se změnou struktury webové stránky nebo dokonce i dynamicky během používání webové aplikace. Takové atributy *id* lze často poznat podle toho, že obvykle obsahují nějaké číslo, spolehlivější je však se zeptat přímo vývojářů testované aplikace. V případě, že jde skutečně o automaticky generovanou hodnotu, je lepší se identifikaci elementů pomocí *id* vyhnout a použít některý ze způsobů uvedených v následujícím textu.

A.3.2 Vyhledávání elementů pomocí atributu name

Další možností, jak element na stránce nalézt, je pomocí atributu *name*. Syntaxe je velice podobná výše uvedenému příkladu, pouze namísto lokátoru *By.id* je zde použito *By.name*.

Výpis 9: Ukázka vyhledání elementu pomocí atributu name

```
WebElement vyhledavaciTlacitko = driver.findElement(By.name("searchButton"));
```

Atribut *name* však na rozdíl od atributu *id* nemusí být na stránce nutně unikátní. Pokud se tedy na webové stránce nachází více elementů se stejným atributem *name* (nebo existuje vysoká pravděpodobnost, že se takový element v budoucnu objeví), pak je nutné lokaci omezit pouze na určitou oblast webové stránky. Např. pokud se element nachází uvnitř jiného elementu (např. menu), lze nejprve najít tento nadřazený element a až v rámci něj pomocí atributu *name* identifikovat hledaný element:

Výpis 10: Řetězení příkazů pro vyhledání elementů

```
WebElement kontakty =  
driver.findElement(By.id(„left_menu“)).findElement(By.name(„contacts“));
```

Nebo raději použít metodu vyhledávání elementů pomocí CSS nebo XPath, jak je specifikováno dále, neboť je jejich zápis přehlednější a také kratší.

A.3.3 Vyhledávání elementů podle atributu class

Řada elementů na webové stránce také disponuje atributem *class*, který je hojně využíván jako selektor při aplikování grafických pravidel definovaných v CSS. Atribut *class* však lze využít i při tvorbě automatizovaných testů. [21 s. 18]

Výpis 11: Ukázka vyhledání elementu pomocí atributu class

```
WebElement menu = driver.findElement(By.class("menu"));
```

Opět je však potřeba mít na paměti, že stejně jako atribut *name* ani atribut *class* nemusí být nutně unikátní (a velice často také není). Opět tedy doporučuji raději element specifikovat blíže pomocí CSS nebo XPath.

A.3.4 Vyhledávání elementů podle značky elementu

Každá webová stránka je tvořena elementy opatřenými různými značkami (angl. tag), které poskytují internetovému prohlížeči informaci, jak je má zobrazit nebo jak s nimi pracovat.

Například nadpis nejvyšší úrovně je ve zdrojovém kódu webové stránky opatřen značkou *h1*:

Výpis 12: Zápis nadpisu první úrovně v html kódu

```
<h1>Nadpis stránky</h1>
```

Takový element lze vyhledat pomocí následujícího kódu:

Výpis 13: Vyhledání nadpisu první úrovně podle značky elementu

```
WebElement nadpis1Urovne = driver.findElement(By.tagName("h1"));
```

I značka elementu však není zcela jednoznačně unikátním identifikátorem prvku na stránce. Proto bych doporučovala pomocí značky vyhledávat jen některé elementy, např. *title*, *h1*, *h2* atd., u nichž existuje jistá pravděpodobnost, že se na stránce nevyskytují vícekrát.

A.3.5 Vyhledávání elementů pomocí textu odkazu

Klíčovým elementem každé webové aplikace jsou odkazy, které umožňují se pohybovat mezi jednotlivými stránkami. I ty lze na stránce při automatizovaném testování snadno identifikovat, a to dokonce dvěma způsoby – pomocí celého textu odkazu nebo pomocí jeho části. [21 s. 20]

Odkaz je v html kódu webové stránky typicky opatřen značkou *a*, např.:

Výpis 14: Zápis hypertextového odkazu v html kódu

```
<a href="/kategorie/628">Fakulty a další útvary</a>
```

Takový element lze nalézt pomocí celého textu odkazu následujícím způsobem:

Výpis 15: Vyhledání hypertextového odkazu pomocí textu odkazu

```
WebElement odkazNaFakulty = driver.findElement(By.linkText("Fakulty a další útvary"));
```

Tentýž element lze však nalézt i pomocí části textu odkazu (*By.partialLinkText*):

Výpis 16: Vyhledání hypertextového odkazu pomocí části textu odkazu

```
WebElement odkazNaFakulty = driver.findElement(By.partialLinkText("Fakulty"));
```

Druhý způsob vyhledávání odkazů je užitečný zejména v případech, kdy jsou jejich texty z části generovány dynamicky – například odkaz na složku s doručenými e-maily v aplikaci Gmail, který zobrazuje i počet nepřečtených zpráv, např. „*Doručená pošta (4)*“. [21 s. 21]

A.3.6 Vyhledávání elementů pomocí CSS

Všechny běžně používané internetové prohlížeče podporují formátování vzhledu webových stránek pomocí kaskádových stylů, které pro identifikaci elementů na stránce využívají CSS selektory. Původní idea zachycuje CSS selektory jako velice rychlý a flexibilní způsob, jak v objektovém modelu dokumentu identifikovat elementy, na které má být vybrané grafické pravidlo aplikováno. Lze vytvořit jak jednoduché, obecné selektory, které ve svém výběru zahrnují velké množství elementů, tak selektory velice komplexní, zaměřené na jeden konkrétní element nacházející se v přesně definovaném kontextu. Díky velice rozšířené podpoře internetových prohlížečů lze CSS selektory využít i pro účely automatizovaného testování. [21 s. 22]

Podoba zdrojového kódu pro vyhledání elementu pomocí CSS selektoru může být např.:

Výpis 17: Vyhledání elementu pomocí CSS selektoru

```
WebElement logo = driver.findElements(By.cssSelector("div.menu li"));
```

Ačkoli tato diplomová práce již u čtenáře předpokládá určitou základní znalost konstrukce CSS selektorů, považuji za vhodné si některé z nich představit blíže.

CSS selektor, který vybírá element dle absolutní cesty, může vypadat například takto: `html body div div img`. Popř. lze selektor ještě více upřesnit pomocí znaku `>`, který znázorňuje přímý vztah předek – potomek: `html > body > div > div > img`. Zatímco první způsob říká, že se element `img` nachází kdekoli uvnitř elementu `div`, druhý způsob vyžaduje, aby byl element `img` přímým potomkem elementu `div`. Všeobecně se však selekce pomocí absolutní cesty využívá jen zřídka, neboť by takový selektor v případě změny struktury webové stránky mohl přestat fungovat, popř. by ukazoval na jiný element. Vhodnější je tedy zvolit selektory s relativní cestou k elementu, které jsou méně závislé na struktuře webové stránky. [21 s. 23]

CSS selektor vyhledávající element pomocí relativní cesty v sobě neobsahuje kompletní výčet nadřazených prvků, ale snaží se nalézt co nejpřesnější identifikaci daného elementu bez ohledu na jeho umístění na webové stránce. Takový selektor může vypadat následujícím způsobem: `div.header > img[alt='Logo VŠE']`. Jak je zřejmé, selektor neobsahuje kompletní cestu, ale pouze její část, přičemž elementy dostatečně specifikuje pomocí atributů tak, aby byly spolehlivě nalezeny, i když se objektový model dokumentu změní. V uvedeném příkladu jde o element `img`, který má atribut `alt="Logo VŠE"` a nachází se uvnitř elementu `div` s atributem `class="header"`.

Přehled CSS selektorů, které mají pro účely automatizovaného testování smysl, je uveden v následující tabulce (Tabulka 4).

Tabulka 4: Přehled CSS selektorů využitelných při automatizovaném testování (Zdroj: [97])

Syntax	Příklad	Popis	Verze CSS
element	img	Vybere všechny elementy se značkou img.	1
element element	div img	Mezera symbolizuje vztah předek – potomek, a to bez ohledu na případné mezistupně. V tomto příkladu tedy může být vybrán i element img, který je přímým potomkem jiného elementu, který je vnořen do elementu div.	1
element > element	div > img	Znaménko > označuje přímý vztah předek – potomek, tedy že element img musí být přímým potomkem (dítětem) elementu div.	2
element, element	a, button	Pomocí čárky lze vyjádřit, že mají být vybrány oba elementy (popř. více elementů).	1
element1 + element2	input + button	Znak + vyjadřuje, že má být vybrán element2, který se nachází bezprostředně za specifikovaným elementem1, přičemž oba mají stejného rodiče (jsou tedy sourozenci). V tomto případě bude tedy nalezen každý element button, kterému bezprostředně předchází element input, přičemž mají oba stejného rodiče.	2
element1 ~ element2	input ~ button	Velice podobnou funkci má i znak ~, který vybere element2, který se nachází za specifikovaným elementem1. Opět musí být elementy sourozenci, ale už nemusí být umístěny bezprostředně po sobě. V tomto případě tedy stačí, aby byl element button umístěn za elementem input, přičemž mezi nimi může být i další element.	3
element#id	#logo	Znak # označuje atribut id. Uvedený selektor tedy najde element s atributem id="logo". Lze použít ve tvaru img#logo či pouze #logo.	1
element.class	div.menu	Tečka znázorňuje atribut class. Uvedený selektor tedy vybere atributy div, které mají atribut class="menu". Selektor lze použít i bez určení značky elementu (tedy pouze .menu), pak je ale namísto CSS selektoru vhodnější zvolit vyhledání elementu pomocí atributu class (viz kapitola A.3.3).	1

Syntax	Příklad	Popis	Verze CSS
[atribut]	img[alt]	Najde všechny elementy img, které mají definován atribut alt (bez ohledu na jeho hodnotu).	2
[atribut=hodnota]	img[alt='logo']	Najde všechny elementy img, které mají hodnotu atributu alt="logo".	2
[atribut^=hodnota]	img[alt^='Vysoká']	Najde všechny elementy img, jejichž hodnota atributu alt začíná řetězcem "Vysoká".	3
[atribut\$=hodnota]	[src\$='.pdf']	Najde všechny elementy, jejichž hodnota atributu src končí řetězcem ".pdf".	3
[atribut*=hodnota]	a[src*='vse']	Najde všechny elementy a, jejichž hodnota atributu src obsahuje řetězec "vse".	3
:checked	input:checked	Najde všechny elementy input, které mají atribut checked – což znamená, že jsou zaškrtnuty. Může jít buď o zaškrťovací políčko (angl. checkbox) nebo přepínač (angl. radio).	3
:disabled	input:disabled	Najde všechny elementy input, které mají atribut disabled. Typicky jde o needitovatelná formulářová vstupní pole, zaškrťovací políčka, přepínače, tlačítka atp.	3
:empty	p:empty	Najde všechny elementy p, které nemají žádného potomka ani neobsahují žádný text.	3
:enabled	input:enabled	Najde všechny elementy input, které nemají atribut disabled a jsou tedy editovatelné. Typicky jde o formulářová vstupní pole, zaškrťovací políčka, přepínače, tlačítka atp.	3
:first-child	tr:first-child	Najde všechny elementy tr, které jsou prvním dítětem svého rodiče. Pozor však na správné pochopení funkcionality selektoru – aby byl daný element vybrán, musí být splněny obě podmínky – musí být prvním dítětem a musí mít značku tr. Neznamená to, že bude vybrán první element tr v rámci daného rodiče!	2
:first-of-type	p:first-of-type	Na rozdíl od předchozího selektoru dokáže first-of-type nalézt první element se značkou p v rámci daného rodičovského elementu aniž by tento element musel být nutně zároveň prvním dítětem.	3
:invalid	input:invalid	Najde všechny elementy input, které obsahují nevalidní hodnotu. Může jít např. o vstupní pole určené pro číselnou hodnotu, do které byl vepsán text.	3

Syntax	Příklad	Popis	Verze CSS
:last-child	td:last-child	Najde všechny elementy td, které jsou posledním dítětem svého rodiče. Opět platí pravidlo splnění obou podmínek jako u selektoru first-child.	3
:last-of-type	p:last-of-type	Najde všechny element, které jsou posledním elementem se značkou p v rámci daného rodičovského elementu. Opět to neznamená, že element musí být nutně i posledním dítětem.	3
:link	a:link	Najde všechny nenavštívené odkazy.	1
:not(selector)	div:not(.menu)	Najde všechny elementy div, které nemají atribut class="menu".	3
:nth-child(n)	td:nth-child(3)	Najde každý element td, který je 3. dítětem svého rodiče.	3
:nth-last-child(n)	td:nth-last-child(2)	Najde každý element td, který je předposledním (tedy 2. od konce) dítětem svého rodiče.	3
:nth-last-of-type(n)	p:nth-last-of-type(5)	Najde každý element p, který je 5. elementem značkou p v rámci daného rodičovského elementu, počítáno od konce.	3
:nth-of-type(n)	p:nth-of-type(2)	Najde každý element p, který je druhým elementem se značkou p v rámci daného rodičovského elementu.	3
:only-of-type	input:only-of-type	Najde každý element, který je jediným elementem se značkou input v rámci daného rodičovského elementu.	3
:only-child	p:only-child	Najde každý element p, který je jediným dítětem svého rodiče.	3
:optional	input:optional	Najde všechny elementy input, které nemají atribut required, a tedy jejich vyplnění ve formuláři není povinné.	3
:required	input:required	Najde všechny elementy input, které mají atribut required, a tedy je nutné je před odesláním formuláře vyplnit, jinak tato akce není umožněna.	3
:visited	a:visited	Najde všechny navštívené odkazy.	1

Kompletní seznam CSS selektorů lze nalézt na stránkách W3Schools. [97] Pro účely vytváření automatizovaných testů by však měl být tento seznam zcela dostačující.

CSS selektory lze kombinovat mnoha různými způsoby, čímž lze docílit nalezení téměř jakéhokoli elementu, který se na webové stránce nachází. Nicméně i přesto existují případy,

kdy je potřeba namísto CSS využít XPath selektor, a to zejména z důvodu chybějící kompatibility prohlížeče s některými CSS selektory. Nejčastěji se takové problémy vyskytují u prohlížeče Internet Explorer verze 8 a starších, ale lze nalézt i selektory, které nejsou podporovány ani nejnovější verzí tohoto prohlížeče. Kompletní přehled kompatibility CSS selektorů v prohlížeči Internet Explorer je k dispozici na webových stránkách Microsoft Developer Network [98].

Internet Explorer však není jediným prohlížečem, kde lze na problémy s kompatibilitou narazit, s některými selektory CSS verze 3 si neporadí ani prohlížeče Firefox, Chrome, Safari či Opera. Přehlednou tabulku kompatibility CSS selektorů v pěti nejpoužívanějších prohlížečích lze nalézt na webu Quirksmode [99].

A.3.7 Vyhledávání elementů pomocí XPath

Ačkoli je vyhledávání elementů pomocí CSS rychlejší a efektivnější [21 s. 8], existují případy, kdy nezbyvá nic jiného, než použít XPath selektor. Ten se hodí nejen v případech problémů s kompatibilitou prohlížeče s CSS selektorem, ale také umožňuje využít několik dalších funkcí, kterými CSS ani ve verzi 3 nedisponuje. Jednou z takových velice často využívaných funkcí je možnost vyhledat element, který ve svém textu či hodnotě atributu obsahuje určitý řetězec. Selenium WebDriver sice umožňuje vyhledat odkaz podle části textu, ale v praxi je často potřeba nalézt i jiný element než jen odkaz, např. tlačítko nebo odstavec textu. V takovém případě je jediným řešením využít XPath selektor s klíčovým slovem *contains*. Jeho použití je spolu s dalšími selektory vysvětleno v dalším textu.

Stejně jako u CSS selektorů lze XPath selektor definovat buď pomocí absolutní, nebo relativní cesty. Absolutní cesta, která může mít např. podobu `/html/body/div/div/img`, vyjmenovává kompletní hierarchii elementů od kořene dokumentu (ve webové stránce je jím vždy element `html`) až po hledaný element (v uvedeném příkladu element `a`). Absolutní cesta vždy začíná jedním lomítkem, který vyjadřuje, že první element je zároveň kořenem dokumentu.

Relativní cesta naopak obsahuje jen ty elementy, které je nutné pro nalezení správného elementu uvést, čímž se do jisté míry zbavuje závislosti na struktuře webové stránky. Relativní cesta vždy začíná dvěma lomítky, čímž dává najevo, že první zmíněný element nemusí být kořenem a může se nacházet kdekoli v objektovém modelu dokumentu, např. `//div/img[@id='logo']`. Stejně jako u CSS selektorů, i zde je všeobecně doporučováno využívat relativní cesty, a to především z důvodu menší závislosti na struktuře objektového modelu dokumentu.

Přehled XPath selektorů využitelných při psaní automatizovaných testů je uveden v následující tabulce (viz Tabulka 5).

Tabulka 5: Přehled XPath selektorů využitelných při automatizovaném testování (Zdroj: [100] [101])

Syntax	Příklad	Popis
//element	//img	Najde všechny elementy se značkou img kdekoli na webové stránce.
element	p	Najde všechny elementy p, které se nacházejí uvnitř aktuálně vybraného elementu. Užitečné zejména při použití XPath funkcí – viz níže.
//element1//element2	//div//img	Dvě lomítka mezi elementy značí, že se druhý element nachází někde uvnitř elementu prvního, tedy, že je jeho potomkem, ale nemusí být nutně jeho přímým potomkem (dítětem). V tomto případě tedy bude nalezen každý element img, který je potomkem elementu div.
element1/element2	div/a	Najde všechny elementy a, které mají za rodiče element div, který se nachází v aktuálně vybraném elementu. Podrobnější popis funkcionality tohoto selektoru je uveden pod tabulkou.
.	//div[contains(.,'VŠE')]	Tečka vyjadřuje aktuálně vybraný element a všechny jeho potomky. Tento znak je užitečný zejména při použití XPath funkcí – viz níže.
..	//p/..	Pomocí dvou teček lze vybrat rodiče aktuálně vybraného uzlu. Uvedený selektor tedy najde rodiče všech elementů p.
element[n]	//div[1]	Najde první výskyt elementu div. Tento selektor je výbornou alternativou v případě problémů s kompatibilitou internetového prohlížeče s CSS selektorem :nth-child(n).
element[last()]	//a[last()]	Najde poslední element a na webové stránce. Jde o alternativu CSS selektoru :last-of-type.
element[last()-n]	//a[last()-1]	Najde předposlední element a na webové stránce. Jde o alternativu CSS selektoru :nth-last-of-type(n).
element[position()<n]	//div/p[position()<3]	Najde první dva elementy p, které jsou dětmi elementu div. Samozřejmě lze použít jak se znakem < (menší než), tak > (větší než).
element[@atribut='hodnota']	//div[@class='menu'] //img[@alt]	Znak @ vyjadřuje, že půjde o atribut. První selektor najde všechny elementy div, které mají atribut class="menu". Je ale možné najít i element, který má určitý atribut, ale není známa jeho hodnota – jako v případě druhého uvedeného selektoru.

Syntax	Příklad	Popis
element[podmínka and podmínka]	<code>//div[@class='header' and @name='contacts']</code>	Pomocí operátoru and lze vytvářet složené podmínky. Uvedený selektor najde všechny elementy div, které mají atribut class="header" a zároveň také atribut name="contacts". Lze také zapsat jako <code>//div[@class='header'][@name='contacts']</code> .
*	<code>//*[@alt='logo'] //p[@*] //div[@class='menu']/*</code>	Znak * zastupuje značku libovolného elementu nebo atributu. První uvedený selektor tedy najde jakýkoli element, který má atribut alt="logo". Druhý selektor najde všechny elementy p, které mají specifikován jakýkoli atribut. Třetí selektor pak najde všechny děti elementu div, který má atribut class="menu".
element element	<code>//div/a //div/p</code>	Najde všechny elementy a, které jsou dětmi elementu div, a zároveň také všechny elementy p, které jsou dětmi elementu div. Jde tedy o sjednocení dvou množin nalezených elementů.
ancestor::element	<code>//a/ancestor::div //a/ancestor::div[@class='menu']</code>	Klíčové slovo ancestor označuje předka. První uvedený selektor tedy najde všechny elementy div, které jsou předky (nemusí být přímo rodiči) elementu a. Druhý uvedený selektor pak ještě přidává podmínku, že element div musí mít atribut class="menu".
descendant::element	<code>//div/ancestor::p</code>	Klíčové slovo descendant označuje potomka. Uvedený selektor tedy najde všechny elementy p, které jsou potomky (tedy nejen dětmi) elementu div. Lze použít i zkrácenou formu v podobě dvou lomítek - např. <code>//div//p</code> , jak již bylo uvedeno výše.
following::element	<code>//div/following::p</code>	Najde všechny elementy p, které se v dokumentu vyskytují po koncové značce (angl. closing tag) elementu div.
following-sibling::element	<code>//p/following-sibling::a</code>	Najde všechny elementy a, které mají stejného rodiče jako element p (jsou tedy sourozenci) a nacházejí se v objektovém modelu dokumentu až za ním.
parent::element	<code>//div/parent::*</code>	Najde všechny rodiče elementů div.
preceding::element	<code>//p/preceding::div</code>	Pomocí klíčového slova preceding lze nalézt všechny elementy, které se nacházejí v objektovém modelu dokumentu před vybraným elementem, s výjimkou jeho předků.

Syntax	Příklad	Popis
<code>preceding-sibling::element</code>	<code>//p/preceding-sibling::a</code>	Najde všechny elementy a, které jsou sourozenci elementu p a nachází se v objektovém modelu dokumentu před ním.

Kód automatizovaného testu, který vyhledává element pomocí XPath selektoru, může vypadat následovně:

Výpis 18: Vyhledání elementu pomocí XPath selektoru

```
WebElement menu = driver.findElements(By.xpath("//div[@class='menu']"));
```

V případě řetězovitého použití funkce `findElement` nebo `findElements` (např. `driver.findElement(By.id("menu")).findElement(By.xpath("p/a"))`), jak již bylo zmíněno v kapitole A.3.2) je však potřeba dávat pozor na správnost XPath selektoru. Uvedený selektor najde všechny elementy a, které jsou dětmi elementu p a nachází se někde uvnitř elementu s atributem `id="menu"`. Pokud by měl ale XPath selektor podobu `//p/a`, pak jsou výsledkem hledání všechny elementy a, které jsou dětmi elementu p, a to v celém dokumentu, nikoli jen v menu. Aby tedy byl nalezen správný element, který se nachází uvnitř elementu nalezeného v předchozím kroku, je nutné použít selektor bez úvodních lomítek.

XPath zároveň umožňuje v selektorech použít celou řadu užitečných funkcí, jako např.: [102]

- *contains(string,substring)*
- *starts-with(string,substring)*
- *ends-with(string,substring)*
- *not(podmínka)*

Zřejmě nejpoužívanější XPath funkcí při psaní automatizovaných testů je příkaz *contains()*. Nejenže na rozdíl od všech dříve zmíněných způsobů vyhledávání elementů umožňuje nalézt jakýkoli element obsahující určitý text, ale umožňuje vyhledat element i podle části textu v hodnotě atributu, v některém ze svých potomků atd. Syntax funkce *contains()* totiž umožňuje jako atribut *string* využít XPath selektor nebo další funkci. Díky tomu lze vytvořit i velice komplexní selektor, který dokáže nalézt zcela libovolný element nacházející se na webové stránce, který by jiným způsobem nebylo možné přesně nalézt.

Nejčastěji využívanou variantou je zřejmě vyhledávání elementu podle části textu, který obsahuje, např. `//p[contains(text(),'škola')]`. Tento selektor umožňuje nalézt element *p*, který ve svém textu obsahuje řetězec "škola". Funkce *text()* totiž vrací řetězec umístěný mezi počáteční a koncovou značkou elementu, přičemž ale vynechává vnořené elementy. Jde tedy výhradně jen o text daného elementu a nikoli jeho potomků.

V případech, kdy je známa celá textová hodnota hledaného elementu, a nikoli jen její část, je možné použít selektor `//p[text()='Výuka']`, který najde všechny elementy *p*, které mají textovou hodnotu "Výuka".

Pokud je však cílem nalézt takový element, v němž se daný řetězec vyskytuje bez ohledu na to, zda jde o text přímo daného elementu nebo některého z jeho potomků, pak je možné selektor formulovat jako `//div[contains(.,'VŠE')]`, kde tečka zastupuje aktuálně vybraný element a všechny jeho potomky. Uvedený selektor tedy najde všechny elementy *div*, které někde ve svém nitru obsahují text (ale nikoli atribut) s řetězcem "VŠE".

Opět lze také vyhledávat element podle celé textové hodnoty daného elementu nebo jeho potomků, a to pomocí selektoru ve tvaru `//p[.='Výuka']`, který najde všechny elementy *p*, jejichž textová hodnota nebo některého z jejich potomků odpovídá řetězci "Výuka".

Dále existuje i možnost zúžit vyhledávání podle části textu i jen na určité potomky – např. pokud je potřeba nalézt řádek tabulky, který obsahuje buňku s určitou hodnotou: `//tr[contains(td,'OK')]`. Namísto tečky tedy stačí uvést značku libovolného podřizovaného elementu nebo i celý selektor: `//div[contains(p/span[@class='gdate'],'4. 7. 2014')]`.

Další možností je vyhledávat element podle části hodnoty jeho atributu, např. `//div[contains(@class,'menu')]`. Tento selektor najde každý element *div*, jehož hodnota atributu *class* obsahuje řetězec "menu". Totéž lze ale učinit i pomocí CSS selektoru, proto má smysl tuto funkci využít jen jako součást komplexnějšího XPath selektoru.

Analogicky lze využívat i funkce *starts-with()* a *ends-with()*, které podobným způsobem nacházejí elementy začínající, popř. končící daným řetězcem. Tyto XPath funkce jsou užitečné zejména v případech, kdy jsou atributy *id* generovány zčásti dynamicky, např. u ASP.NET aplikací. Atribut *id* pak může mít např. podobu "searchButton_12" (klíčové slovo je na začátku a doplněno automaticky generovaným kódem) nebo "a_1_userName" (klíčové slovo je na konci řetězce). Pomocí funkcí *starts-with()* a *ends-with()* pak lze i tyto elementy snadno nalézt, a to selektory ve tvaru `//button[starts-with(@id,'searchButton_')]` a `//input[ends-with(@id,'_userName')]`.

Při používání funkcí *contains()*, *starts-with()* a *ends-with()* je však mít potřeba na paměti, že jde o funkce velice náročné na výpočetní výkon. V případě jejich aplikace na několik desítek elementů se tak může stát, že se při provádění automatizovaného testu objeví výjimka *TimeoutException*, neboť provádění kódu trvalo příliš dlouho. Z toho důvodu je vhodné použít co nepřesnější selektor a danou funkci aplikovat jen na několik málo elementů.

Další užitečnou funkcí je `not(podmínka)`, která vyjadřuje negaci – umožňuje tedy nalézt elementy, které nespĺňují danou podmínku. V případě selektoru `//div[not(@class='menu')]` tak budou nalezeny všechny elementy *div*, které nemají atribut *style* s hodnotou "menu".

Další funkce lze nalézt na stránkách W3Schools v sekci věnované XPath [102].

A.4 Práce s elementy

V předchozí kapitole bylo vysvětleno, jakým způsobem lze na webové stránce nalézt určitý element. Nalézt jej však nestačí, pro správné otestování aplikace je potřeba nad daným elementem provést nějakou akci – např. kliknout na něj, zjistit jeho vlastnosti, vepsat do něj určitou hodnotu apod. V této kapitole je uveden návod, jak pomocí nástroje Selenium WebDriver provádět nejčastější úkony potřebné pro řádné otestování webové aplikace.

A.4.1 Kliknutí na element

Nejčastějším úkonem prováděným při používání jakékoli webové aplikace je bezpochyby kliknutí na nějaký prvek na stránce. Ať už jde o odkaz, tlačítko či položku menu, příkaz má v automatizovaném testu vytvářeném s pomocí nástroje Selenium WebDriver ve všech případech stejnou podobu:

```
driver.findElement(By.id("searchButton")).click();
```

Příkaz lze samozřejmě rozdělit i do dvou řádků – nejprve element nalézt a uložit jeho instanci do proměnné a až poté nad ním provést požadovanou akci, např.:

```
WebElement vyhledavaciTlacitko = driver.findElement(By.id("searchButton"));  
vyhledavaciTlacitko.click();
```

Tento způsob však zejména ve webových aplikacích využívajících JavaScript může způsobovat problémy s výjimkou *StaleElementReferenceException*. Tato výjimka je způsobena změnou objektového modelu dokumentu, ke které došlo v čase mezi nalezením elementu a jeho použitím. K takové změně dochází obvykle při obnovení nebo překreslení stránky nebo její části, a to mnohdy aniž by si toho uživatel všiml. Ačkoli vizuálně k žádným změnám na stránce nemuselo dojít, objektový model dokumentu byl v důsledku aktivity JavaScriptu změněn, což Selenium WebDriver zaznamená a nedovolí již dříve nalezený element použít, neboť již nemůže zaručit, že je nově vykreslený element shodný s původním elementem. V takovém případě tedy vyhodí výjimku *StaleElementReferenceException*. Nejlepším způsobem, jak se takovým problémům vyhnout, je spojit nalezení a použití elementu do jednoho řádku kódu. [103]

A.4.2 Dvojklik

Při práci s některými webovými aplikacemi však někdy jen jednoduché kliknutí nestačí a pro provedení určité funkcionality aplikace je potřeba na element kliknout dvakrát rychle po sobě. V nástroji Selenium WebDriver lze tento úkon provést pomocí instance třídy *Actions* a příkazu *doubleClick()*, následovaným příkazem *perform()*: [20 s. 57]

Výpis 19: Provádění dvojkliku nad elementem

```
Actions builder = new Actions(driver); //vytvoření instance třídy Actions
builder.doubleClick(driver.findElement(By.id("message"))).perform(); //provedení
dvojkliku nad elementem s atributem id="message"
```

Po volání metody `doubleClick()` musí následovat příkaz `perform()`. Tento krok je velice důležitý, neboť bez něj by nebyl dvojklik proveden, neboť teprve příkazem `perform()` je akce skutečně provedena.

Důležité upozornění: nutnost zakončení akce příkazem `perform()` je společná pro všechny metody třídy `Actions`.

Vzhledem k tomu, že třída `Actions` není součástí základní knihovny nástroje Selenium WebDriver (`org.openqa.selenium.WebDriver`), která byla zmíněna v kapitole A.1, je potřeba do dané testovací třídy nejprve naimportovat knihovnu `org.openqa.selenium.interactions.Actions`.

A.4.3 Vyplňování formulářových polí

Dalším častým úkonem uživatele na webové stránce je vyplňování vstupních polí formulářů a jejich odesílání. I vyplnění vstupního pole lze velice jednoduše automatizovat, a to pomocí příkazu `sendKeys()`:

Výpis 20: Vyplňování vstupního pole formuláře

```
driver.findElement(By.name("hledejutf8")).sendKeys("Studijní řád");
```

Uvedený kód najde element s atributem `name="hledejutf8"` a vepíše do něj hodnotu "Studijní řád". Pro vložení textového řetězce je pochopitelně potřeba, aby byl element typu vstupního pole, tedy měl značku `input`. V opačném případě nebude řetězec vložen, ale kód proběhne bez vyhození výjimky.

Po vyplnění formuláře je obvykle potřeba jej odeslat. V HTML se pro vytváření formulářů používají standardizované elementy, přičemž tlačítko pro odeslání vložených dat formuláře typicky nese atribut `type="submit"`. Díky tomu je možné toto tlačítko v rámci formuláře snadno identifikovat, neboť může být v daném formuláři pouze jedno. Stačí tedy pomocí WebDriveru nalézt daný formulář (v HTML struktuře má značku `form`) a nad ním zavolat metodu `submit()`.

Výpis 21: Odeslání formuláře

```
driver.findElement(By.cssSelector("form[name='search']")).submit();
```

A.4.4 Získávání textové hodnoty elementu

V některých situacích je naopak potřeba získat textovou hodnotu elementu a následně ověřit její správnost. K tomuto úkonu slouží příkaz `getText()`:

Výpis 22: Získávání textu z elementu

```
String odstavec = driver.findElement(By.cssSelector(".main p")).getText();
```

Jak je zřejmé z výše uvedeného příkladu, metoda `getText()` vrací řetězec znaků (*string*). Jeho obsahem je přitom textová hodnota nejen daného elementu, ale i všech jeho potomků.

Po extrakci textu z elementu s ním lze pracovat stejně jako s kterýmkoli běžným řetězcem – ověřovat jeho hodnotu (např. `odstavec.equals("Akademický rok 2015");`), zjišťovat, zda obsahuje nějaký řetězec (např. `odstavec.contains("rok");`), rozdělovat jej na části s využitím regulárních výrazů (např. `odstavec.split("\\s")[0];` – získání pouze prvního slova z věty) atd.

Následně lze textovou hodnotu elementu vyhodnotit pomocí metody `Assert.assertTrue()`, resp. `Assert.assertFalse()`, která ověřuje, zda je splněna, resp. nesplněna určitá podmínka, a v opačném případě vyhodí výjimku a ukončí provádění testu s výsledkem *Test failed*. Kód ověřující textovou hodnotu elementu může vypadat následujícím způsobem:

Výpis 23: Ověřování hodnoty nadpisu první úrovně s očekávanou hodnotou

```
String nadpis = driver.findElement(By.tagName("h1")).getText();  
Assert.assertTrue("Nadpis webové stránky není správný!", nadpis.equals("Informace  
o VŠE"));
```

Nejprve je získána textová hodnota nadpisu se značkou `h1`, poté se zjišťuje, zda odpovídá řetězci "Informace o VŠE" a v případě, že nikoli, pak je provádění testu ukončeno a do konzole je vypsán text "Nadpis webové stránky není správný!". Pokud text nadpisu odpovídá zadané hodnotě, pak test pokračuje dále.

A.4.5 Získávání hodnoty atributu elementu

Stejně tak snadno, jako lze vyextrahovat z elementu jeho textovou hodnotu, lze snadno získat i hodnotu kteréhokoli z jeho atributů, a to pomocí metody `getAttribute()`:

Výpis 24: Získávání hodnoty atributu `alt` z elementu

```
String logoAlt = driver.findElement(By.id("logo")).getAttribute("alt");
```

Analogicky lze z elementu získat i informaci o jeho CSS vlastnostech pomocí metody `getCssValue()`:

Výpis 25: Získávání CSS vlastností z elementu

```
String velikostNadpisu1 = driver.findElement(By.tagName("h1")).getCssValue("font-size");
```


A.4.6 Kontrola stavu elementu

Ne vždy je nutné s elementem fyzicky provádět nějakou akci, existují případy, kdy postačuje zjistit o elementu určité informace a porovnat je s očekávanými hodnotami.

Často je například pro hladký průběh testu potřeba nejprve zjistit, zda je prvek vůbec zobrazen. Může se totiž stát, že je element sice na stránce přítomen, ale v danou chvíli je skryt a z toho důvodu na něj například nelze kliknout. Ověření zobrazení elementu lze provést pomocí metody `isDisplayed()`: [20 s. 40]

Výpis 26: Zjišťování, zda je element zobrazen

```
boolean jeTlacitkoHledejZobrazeno =  
driver.findElement(By.cssSelector("input[value='Hledej']")).isDisplayed();
```

Stejně snadno lze také zjistit, zda je prvek (typicky tlačítko nebo vstupní pole) aktivní či nikoli. K tomuto účelu slouží metoda `isEnabled()`, která vrací hodnotu `true` nebo `false` podle toho, zda je element aktivní či neaktivní. [20 s. 41] Příklad použití v kódu testu pak vypadá následovně:

Výpis 27: Zjišťování, zda je element aktivní

```
boolean jeTlacitkoHledejAktivni =  
driver.findElement(By.cssSelector("input[value='Hledej']")).isEnabled();
```

A.4.7 Práce s rozevíracími a výběrovými seznamy

Řada webových aplikací obsahuje formulář s rozevíracím seznamem (angl. drop-down list) nebo výběrovým seznamem (angl. list), jejichž funkcionalitu je samozřejmě potřeba také otestovat. Proto se tvůrci nástroje Selenium WebDriver rozhodli práci s nimi zjednodušit vytvořením třídy `Select`. Tato třída je určena výhradně pro práci s elementy s html značkou `select`, což jsou typicky rozevírací nebo výběrové seznamy, ze kterých je možné vybrat jednu či více hodnot, které pak mají značku `option`. [21 s. 56]

Výpis 28: Příklad HTML kódu rozevíracího seznamu

```
<select name="languages">  
  <option value="cs" index=0>Čeština</option>  
  <option value="en" index=1>Angličtina</option>  
  <option value="de" index=2>Němčina</option>  
</select>
```

Kód automatizovaného testu pak může mít například následující podobu:

Výpis 29: Kód automatizovaného testu pracujícího s rozevírácím seznamem

```
Select jazyky = new Select(driver.findElement(By.name("languages")));
Assert.assertEquals(3, jazyky.getOptions().size()); //ověření, že rozevírací seznam
obsahuje 3 možnosti
Assert.assertFalse(jazyky.isMultiple()); //ověření, že lze vybrat pouze jednu
možnost
jazyky.selectByValue("en"); //výběr možnosti s atributem value="en"
Assert.assertEquals("Angličtina", jazyky.getFirstSelectedOption().getText());
//ověření, zda vybraná možnost nese správnou textovou hodnotu
```

Pro úspěšné sestavení kódu je potřeba nejdříve nainportovat třídu `org.openqa.selenium.support.ui.Select`;

Třída *Select* umožňuje vybírat možnosti třemi způsoby – pomocí textu, který zobrazují, nebo pomocí hodnot atributů *value* či *index*. Výběr možnosti pomocí textu lze provést pomocí metody *selectByVisibleText()*, např. `make.selectByVisibleText("Němčina")`. Atribut *value* lze použít pro výběr možnosti metodou *selectByValue()*, např. `make.selectByValue("de")`. Poslední metodou je pak výběr možnosti pomocí atributu *index*, ke které slouží metoda *selectByIndex()*, např. `make.selectByIndex(0)`. Nutno podotknout, že *index* je zde skutečně míněn jako hodnota atributu *index* daného elementu, nikoli jeho pořadí v seznamu. Dále je potřeba si dát pozor na dynamicky generované hodnoty atributu *index*, které mohou následně způsobovat problémy s výběrem správné možnosti.

Stejně tak, jako je možné možnosti vybírat pomocí textu nebo atributu *value* nebo *index*, lze i možnosti odznačovat (tedy zrušit jejich označení), a to metodami *deselectByVisibleText()*, *deselectByValue()* nebo *deselectByIndex()*. Popř. lze odznačit všechny vybrané možnosti najednou pomocí metody *deselectAll()*.

Pro ověření správnosti výběru možností z rozevíracího seznamu lze pak použít metody *getFirstSelectedOption()* nebo *getAllSelectedOptions()*. Jak jejich název napovídá, první z nich vrátí instanci třídy *WebElement* první zvolené možnosti ze seznamu, zatímco druhá poskytne seznam instancí třídy *WebElement*, které reprezentují všechny zvolené možnosti v seznamu. Kompletní seznam možností, které rozevírací nebo výběrový seznam nabízí, lze pak získat pomocí metody *getOptions()*.

Další užitečnou metodou je *isMultiple()*, která poskytuje informaci, zda je v seznamu možné zvolit více možností najednou. Metoda vrací hodnotu typu *boolean* – pokud vrátí *true*, pak seznam podporuje vícenásobnou selekci, a naopak pokud vrátí *false*, pak je možné ze seznamu vybrat jen jedinou možnost.

Kompletní seznam metod využitelných při práci s rozevíracími a výběrovými seznamy lze nalézt v oficiální dokumentaci třídy *Select* [104].

A.4.8 Práce se zaškrťovacími políčky a přepínači

Součástí formulářů ve webových aplikacích, které je potřeba otestovat, jsou poměrně často také zaškrťovací políčka (angl. checkbox) nebo přepínače (angl. radio). Nástroj Selenium WebDriver s nimi umožňuje pracovat pomocí metod `click()` a `isSelected()`. Označit nebo odznačit zaškrťovací políčko či přepínač lze jednoduše použitím metody `click()`. Druhá ze jmenovaných metod, `isSelected()`, pak poskytuje informaci (`true/false`), zda je zaškrťovací políčko nebo přepínač označeno či nikoli. [21 s. 66]

Výpis 30: Část zdrojového kódu webové stránky s přepínačem

```
<form>
  <input type="radio" name="sex" value="male">
  <input type="radio" name="sex" value="female">
</form>
```

Příklad kódu automatizovaného testu pak může mít následující podobu:

Výpis 31: Kód automatizovaného testu pracujícího s přepínači

```
driver.findElement(By.cssSelector("input[value='female']")).click();
Assert.assertTrue(driver.findElement(By.cssSelector("input[value='female']"))
    .isSelected()); //ověření, zda je označen přepínač s hodnotou "female"
Assert.assertFalse(driver.findElement(By.cssSelector("input[value='male']"))
    .isSelected()); //ověření, zda není označen přepínač s hodnotou "male"
```

Pozn.: rozdíl mezi zaškrťovacími políčky a přepínači je nejen v jejich vzhledu, ale také v tom, že zaškrťvacích políček lze typicky označit více najednou, zatímco přepínač může být označen vždy maximálně jeden (pokud uživatel označí některý z neoznačených přepínačů, pak je zároveň odznačen dříve označený přepínač). Tento poznatek je důležitý pro správné otestování formuláře se zaškrťovacími políčky a přepínači.

A.4.9 Provádění úkonu táhni-a-pust'

Některé webové aplikace lze ovládat i technikou táhni-a-pust' (angl. drag-and-drop), tedy přetahováním některých objektů po obrazovce z jednoho místa na jiné. I takovou funkcionalitu je potřeba otestovat, což lze v automatizovaném testu s využitím nástroje Selenium WebDriver provést pomocí speciální třídy `Actions`, která byla zmíněna již v kapitole A.4.2.

Nejprve je tedy potřeba tento balíček do dané testovací třídy naimportovat, neboť není součástí základní knihovny nástroje Selenium WebDriver:

Výpis 32: Import balíčku třídy `Actions`

```
import org.openqa.selenium.interactions.Actions;
```

Samotný kód automatizovaného testu pak může vypadat například takto: [21 s. 50]

Výpis 33: Ukázka kódu automatizovaného testu, který simuluje akci táhni-a-pust'

```
WebElement source = driver.findElement(By.id("draggable"));
WebElement target = driver.findElement(By.id("droppable"));
Actions builder = new Actions(driver);
builder.dragAndDrop(source, target).perform();
```

Metoda *dragAndDrop()* uchopí element, který je prvním parametrem, a přetáhne jej na místo, kde leží element uvedený jako druhý parametr. Po volání metody *dragAndDrop()* musí následovat příkaz *perform()*, jinak nebude akce vůbec provedena. Tato vlastnost je společná pro všechny metody třídy *Actions*.

A.5 Řízení průběhu testu

Při vytváření automatizovaných testů se lze nečekaně setkat s problémem, že testovaná aplikace neodpovídá tak rychle, jak by bylo žádoucí pro hladký průběh automatizovaného testování. Příčinou těchto problémů může být velký objem dat stahovaných spolu s webovou stránkou, rychlostí internetového připojení, výkonem serveru nebo také výkonem koncové stanice, tedy počítače, na kterém je stránka zobrazována. Ať je již příčina zpoždění jakákoli, WebDriver je nastaven tak, že implicitně (tedy již ze svého základního nastavení) čeká na úplné načtení webové stránky a až poté se snaží provést další kroky. Díky tomu se u klasických aplikací využívajících kombinaci technologií HTML, CSS a PHP nemůže stát, že by WebDriver hledal na stránce element, který se ještě nestihl načíst.

V současné době se však v mnohem větší míře vyskytují aplikace, které vedle zmíněných technologií využívají také JavaScript. A právě zde nastává problém, se kterým se s největší pravděpodobností setkal každý uživatel nástroje Selenium WebDriver – u těchto aplikací se totiž objektový model dokumentu mění i bez načítání stránky. Často se tedy stává, že je stránka načtena, ale data se stále ještě stahují a webová stránka se překresluje, což může trvat ještě několik dalších sekund po samotném načtení stránky, tedy v době, kdy se již WebDriver snaží provádět další kroky. Tento fakt je velice často příčinou vyhození výjimky *StaleElementReferenceException*, která upozorňuje na to, že byl objektový model dokumentu mezitím změněn a WebDriver tak nemůže zaručit, že bude element nalezen správně [103], či výjimky *NoSuchElementException*, která je vyhozena v případě, že se element v danou chvíli na stránce vůbec nenachází (protože v době, kdy byl hledán, ještě nebyl načten) [105].

AJAX (Asynchronous JavaScript and XML) je technologie, která vznikla spojením XML, JavaScript, HTTP a (X)HTML. Tento přístup umožňuje webové stránce kontaktovat server a obdržet od něj data ve formátu XML, aniž by musela být stránka znovu načtena – jednoduše se překreslení pouze ta část webové stránky, která se v důsledku potřeby zobrazení nových dat musela změnit. Díky tomu je webová aplikace rychlejší a také se minimalizují přenosy dat, neboť není potřeba posílat celý zdrojový kód stránky, jako v případě klasických webových

stránek, ale posílají se jen změněná data, kterých je podstatně méně. [106] Avšak ačkoli má AJAX svůj nesporný kladný vliv na rychlost a výkonnost aplikace, pro osoby, které vytvářejí nebo spravují automatizované testy těchto aplikací, znamená tato technologie nemalou komplikaci.

Tato kapitola představuje několik způsobů, jak řídit průběh testů tak, aby nedocházelo k jejich neočekávaným pádům testů z důvodu překreslování testované webové aplikace. Správným nastavením automatizovaných testů lze docílit toho, aby nástroj Selenium WebDriver vždy nejprve počkal, až bude překreslování testované webové stránky dokončeno, a až poté provedl požadovaný úkon. K tomuto účelu slouží třída *WebDriverWait*, jejíž využití je vysvětleno v následujících podkapitolách. [21 s. 74]

A.5.1 Implicit wait

Nejjednodušším způsobem, jak se vyhnout problémům při delším překreslování testované webové aplikace, je prodloužení doby čekání na přítomnost elementu, a to pomocí takzvaného implicitního čekání (angl. *implicit wait*). Jde o nastavení doby, po kterou se Selenium WebDriver snaží opakovaně nalézt na stránce daný element, dokud jej nenajde nebo dokud tento časový limit nevyprší. [21 s. 74]

Implicitně (tedy ze základního nastavení) je tato hodnota nastavena na nulu, tzn., že se Selenium WebDriver pokusí element nalézt pouze jednou, a pokud ten v danou chvíli není v objektovém modelu dokumentu přítomen, vyhodí výjimku *NoSuchElementException*. Nastavením hodnoty implicitního čekání na vyšší hodnotu, např. 10 vteřin, se doba čekání na element prodlouží na 10 vteřin. Neznamená to však, že by Selenium WebDriver před každým zavoláním metody *findElement()* nebo *findElements()* čekal po dobu 10 sekund, ale po celý tento časový úsek se snaží element nalézt a v případě, že jej najde, čekání okamžitě ukončí a test pokračuje dále bez jakékoli časové prodlevy. Pokud však element není v objektovém modelu přítomen ani po uplynutí pěti vteřin, vyhodí výjimku *NoSuchElementException*. [20 s. 72]

Nastavenou hodnotu doby implicitního čekání si v sobě instance WebDriveru nese po celý zbytek své existence, stačí ji tedy nastavit už na začátku automatizovaného testu hned po inicializaci instance WebDriveru a poté již zůstává platná až do okamžiku zavření okna internetového prohlížeče (příkazem *driver.quit()*, popř. *driver.close()*). [24 s. 104]

Výpis 34: Použití implicitního čekání v kódu automatizovaného testu

```
WebDriver driver = new FirefoxDriver();
driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);
driver.get("http://www.vse.cz");
```

V průběhu testu lze hodnotu doby implicitního čekání samozřejmě měnit, a to jednoduše nastavením nové hodnoty.

Implicitní čekání však nedokáže předejít vyhození výjimky *StaleElementReferenceException*, která již byla zmíněna výše. Může se totiž stát, že WebDriver daný element najde, ale během zlomku sekundy zjistí, že se objektový model dokumentu změnil (protože se v danou chvíli stránka právě překresluje), a tak vyhodí uvedenou výjimku a test tak skončí s chybou, ačkoli testovaná aplikace funguje správně. Řešením tohoto problému je tzv. Fluent wait, který je představen v kapitole A.5.3 Fluent wait.

A.5.2 Explicit wait

Existují případy, kdy implicitní čekání nestačí a je potřeba místo něj použít jiné řešení. Jedním z nich je tzv. explicitní čekání (angl. *explicit wait*), které se od předchozího způsobu čekání odlišuje především tím, že jde o jednorázové čekání na splnění určité podmínky, zatímco implicitní čekání je aplikováno při každém vyhledávání elementu na webové stránce. [20 s. 73] Další výhodou explicitního čekání je pak možnost nastavit čekání na libovolnou událost, nikoli jen na přítomnost elementu v objektovém modelu dokumentu, jak je tomu v případě implicitního čekání. [21 s. 78]

Explicitní čekání se hodí zejména v situacích, kdy se v testované aplikaci vyskytuje několik málo problémových míst s delší dobou odezvy, zatímco zbytek aplikace funguje svižně. Právě před tato problémová místa je vhodné umístit explicitní čekání, který po zadanou dobu počká, dokud není splněna zadaná podmínka (např. dokud se nezobrazí určitá varovná zpráva, dokud se určité tlačítko nestane aktivním apod.) a až poté může test pokračovat dále.

Nejprve je potřeba vytvořit instanci třídy *WebDriverWait*, která definuje časový limit v sekundách, jak dlouho může maximálně čekání trvat:

Výpis 35: Nastavení hodnoty čekání

```
WebDriverWait wait = new WebDriverWait(driver, 10);
```

WebDriverWait není součástí standardních knihoven WebDriver, a tak je potřeba do dané testovací třídy naimportovat balíček `org.openqa.selenium.support.ui.WebDriverWait`.

Dále je potřeba definovat podmínku, na jejíž splnění má WebDriver čekat. K tomu slouží třída `ExpectedConditions` a rozhraní `ExpectedCondition<T>`, které jsou představeny v následujícím textu.

Třída `ExpectedConditions`

Pro definici podmínky, která má být pro úspěšné pokračování v provádění testu splněna, je určena třída `ExpectedConditions`, která poskytuje celou řadu metod, které reprezentují podmínky, které je potřeba splnit pro další postup v testování.

Mezi nepoužívanější metody třídy `ExpectedConditions` patří: [107]

- `elementToBeClickable(By locator)`, popř. `elementToBeClickable(WebElement element)` – zjišťuje, zda je možné na element kliknout.
- `elementToBeSelected(By locator)`, popř. `elementToBeSelected(WebElement element)` – zjišťuje, zda je element označen.
- `presenceOfElementLocated(By locator)` – zjišťuje, zda je element přítomen v objektovém modelu dokumentu.
- `textToBePresentInElement(WebElement element, String text)` – zjišťuje, zda element ve své textové hodnotě obsahuje zadaný text.
- `visibilityOfElementLocated(By locator)`, popř. `visibilityOf(WebElement element)` – zjišťuje, zda je element zobrazen.
- `titleIs(String title)`, popř. `titleContains(String title)` – zjišťuje, zda titulek webové stránky odpovídá, resp. obsahuje určitý text.

Kompletní seznam metod třídy `ExpectedConditions` je k dispozici v oficiální dokumentaci nástroje Selenium WebDriver [107].

Kód automatizovaného testu pak může vypadat například následujícím způsobem:

Výpis 36: Využití třídy `ExpectedConditions` pro definici podmínky, na jejíž splnění se má čekat

```
WebDriver driver = new FirefoxDriver();
WebDriverWait wait = new WebDriverWait(driver, 10); //inicializace instance třídy
WebDriverWait s hodnotou 10 sekund
driver.get("http://www.vse.cz"); //otevření webové stránky
driver.findElement(By.linkText("Informace o VŠE")).click(); //kliknutí na odkaz
wait.until(ExpectedConditions.visibilityOfElementLocated(By.tagName("h1")));
//čekání na zobrazení nadpisu 1. úrovně
Assert.assertTrue(driver.findElement(By.tagName("h1")).getText().equals("Informace
o VŠE")); //ověření hodnoty nadpisu 1. úrovně
```

I zde je pro správnou funkcionalitu potřeba naimportovat odpovídající knihovnu – `org.openqa.selenium.support.ui.ExpectedConditions`.

Uvedený příklad automatizovaného testu tedy otevře okno prohlížeče Firefox, nastaví hodnotu čekání na 10 sekund, otevře zadanou URL adresu, klikne na odkaz Informace o VŠE, počká, dokud se nezobrazí nadpis 1. úrovně a následně ověří jeho textovou hodnotu.

V případě, že element se značkou `h1` není zobrazen, Selenium WebDriver opakuje jeho hledání a ověřování, zda je zobrazen, každých 500 milisekund, až do vypršení časového limitu 10 sekund nebo dokud element není zobrazen. V případě, že se ani po 10 vteřinách element nezobrazí, je vyhozena výjimka *TimeoutException*. [108]

Metody *ExpectedConditions* vrací hodnotu typu *boolean*, tedy *true* (element byl nalezen a je zobrazen) nebo *false* (element buď nebyl nalezen, nebo není na stránce viditelný). Metoda *wait.until()* pak na základě této *boolean* hodnoty vyhodnocuje, zda je potřeba provádění kódu zopakovat (v případě hodnoty *false*) nebo jestli už je podmínka splněna (hodnota *true*) a lze pokračovat v provádění dalších kroků testu. [108]

Při používání metod třídy *ExpectedConditions* je však nutné mít na paměti rozdíl mezi metodou využívající jako parametr instanci *WebElementu* a metodou využívající lokátor. Metoda *visibilityOfElementLocated(By locator)* zjišťuje, zda je element odpovídající zadanému lokátoru přítomen na stránce a zda je zobrazen. V případě, že daný element v danou chvíli není přítomen v objektovém modelu elementu, se jej pokusí vyhledat znovu a tento postup se opakuje tak dlouho, dokud se element buď neobjeví, nebo nevyprší časový limit. Až poté zjišťuje, zda je zobrazen či nikoli (některé elementy totiž mohou být přítomny v objektovém modelu dokumentu, ale jejich zobrazení je na základě CSS pravidla zakázáno). Naopak metoda *visibilityOf(WebElement element)* vyžaduje, aby byl již element v objektovém modelu dokumentu přítomen, a zjišťuje pouze jeho viditelnost na webové stránce. Pokud je tedy spuštěn následující kód:

Výpis 37: Čekání na zobrazení nadpisu první úrovně

```
wait.until(ExpectedConditions.visibilityOf(driver.findElement(By.tagName("h1"))));
```

a element se značkou `h1` není v danou chvíli v objektovém modelu dokumentu přítomen, pak je již při prvním pokusu o provedení kódu `driver.findElement(By.tagName("h1"))` vyhozena výjimka *NoSuchElementException* a test skončí neúspěšně, aniž by bylo čekání na element aplikováno. Z tohoto důvodu je vhodnější používat metody s parametrem *By locator*.

Rozhraní `ExpectedCondition<T>`

Vedle předdefinovaných metod třídy `ExpectedConditions` je možné v rámci explicitního čekání použít i podmínku, kterou je nutno splnit pro postup k dalším krokům testování. K tomuto účelu slouží rozhraní `ExpectedCondition<T>`, kde `T` označuje typ návratové hodnoty. [109]

Její použití v praxi může vypadat následujícím způsobem:

Výpis 38: Využití rozhraní `ExpectedCondition<T>` pro pokročilé možnosti nastavení čekání na splnění podmínky

```
WebDriver driver = new FirefoxDriver();
WebDriverWait wait = new WebDriverWait(driver, 10); //inicializace instance třídy
WebDriverWait s hodnotou 10 sekund
driver.get("http://www.vse.cz");
driver.findElement(By.linkText("Informace o VŠE")).click();
wait.until(new ExpectedCondition<Boolean>(){
    public Boolean apply(WebDriver d) {
        return d.findElement(By.tagName("h1")).isDisplayed();
    }
}); //čekání na zobrazení nadpisu 1. úrovně
Assert.assertTrue(driver.findElement(By.tagName("h1")).getText().equals("Informace
o VŠE")); //ověření hodnoty nadpisu 1. úrovně
```

Pro úspěšné sestavení kódu je potřeba opět nejprve naimportovat knihovnu `org.openqa.selenium.support.ui.ExpectedCondition`.

Rozhraní `ExpectedCondition<T>` deklaruje metodu `apply(WebDriver driver)`, která se používá pro ověření splnění podmínky. Ve výše uvedeném příkladu tato metoda obsahuje kód, který vyhledá element `h1`, zjistí, jestli je zobrazen, a následně vrátí hodnotu typu `boolean` (`true`, pokud je zobrazen, nebo `false`, pokud zobrazen není). V případě, že element se značkou `h1` není v objektovém modelu přítomen nebo není zobrazen, je kód opakovaně spouštěn každých 500 milisekund [110], dokud není element přítomen a zobrazen nebo nevyprší časový limit 10 sekund.

Výše uvedený příklad je pouze jednoduchou ukázkou použití rozhraní `ExpectedCondition<T>` pro vytvoření podmínky, která musí být splněna před provedením dalšího kroku automatizovaného testu, pro ověření zobrazení elementu je samozřejmě vhodnější a přehlednější použít metodu `visibilityOfElementLocated(By locator)` třídy `ExpectedConditions`, která byla zmíněna výše. Existují nicméně případy, kdy již předdefinované metody `ExpectedConditions` nestačí a nezbyvá než použít rozhraní `ExpectedCondition<T>`.

A.5.3 Fluent wait

Dalším, ještě pokročilejším, způsobem, jak řídit průběh testu pomocí čekání na určitou událost nebo stav, je tzv. souvislé čekání (angl. *fluent wait*). To totiž na rozdíl od výše uvedené třídy *WebDriverWait* umožňuje vedle časového limitu nastavit i další charakteristiky – např. v jakých časových intervalech má být podmínka znovu vyhodnocována či jaké druhy výjimek mají být ignorovány.

Nejčastěji využívané metody třídy *FluentWait* jsou: [111]

- *withTimeout(long, TimeUnit)* – definuje, jak dlouho může maximálně čekání trvat.
- *pollingEvery(long, TimeUnit)* – za jaký časový úsek má být podmínka znovu vyhodnocena.
- *ignoring(exceptionClass)*, popř. *ignoring(exceptionClass1, exceptionClass2)*, popř. *ignoreAll(Collection<exceptionClass>)* – jaká výjimka, popř. výjimky mají být ignorovány.
- *withMessage(String)* – jaká zpráva má být vypsána do konzole, pokud časový limit vyprší a podmínka stále není splněna.

Tyto vlastnosti jsou užitečné zejména v situacích, kdy testovaná webová aplikace využívá JavaScript. Taková stránka se totiž často po samotném načtení ještě několik sekund vykresluje, popř. může dojít k překreslení, a tedy i změně v objektovém modelu dokumentu, i kdykoli v průběhu práce s aplikací, což způsobuje problémy při běhu automatizovaného testu. Pokud totiž při práci s elementem dojde k jakékoli změně objektového modelu dokumentu (ať už se daného elementu přímo týkala či ne), Selenium WebDriver tuto změnu zaznamená a vyhodí *StaleElementReferenceException*, díky čemuž test okamžitě skončí s chybou, ačkoli aplikace funguje správně. Právě zde je více než příhodné využít souvislé čekání, které bude výjimku tohoto typu ignorovat a pokusí se podmínku vyhodnotit znovu.

Kód automatizovaného testu, který je ošetřen proti chybám způsobeným aktivitou JavaScriptu, pak může mít následující podobu:

Výpis 39: Pokročilé možnosti nastavení souvislého čekání na splnění podmínky

```
WebDriver driver = new FirefoxDriver();
Wait wait = new FluentWait(driver)
    .withTimeout(10, TimeUnit.SECONDS)
    .pollingEvery(500, TimeUnit.MILLISECONDS)
    .ignoring(NoSuchElementException.class, StaleElementReferenceException.class);
//vytvoření instance třídy FluentWait a nastavení jejích vlastností - časový limit,
časový interval opakování a ignorované výjimky
driver.get("http://www.vse.cz"); //otevření webové stránky
driver.findElement(By.linkText("Informace o VŠE")).click(); //kliknutí na odkaz
wait.until(ExpectedConditions.visibilityOfElementLocated(By.tagName("h1")));
//čekání na zobrazení nadpisu 1. úrovně
Assert.assertTrue(driver.findElement(By.tagName("h1")).getText().equals("Informace
o VŠE")); //ověření hodnoty nadpisu 1. úrovně
driver.quit(); //zavření okna prohlížeče
```

Před spuštěním kódu je potřeba nainportovat tyto knihovny:

Výpis 40: Seznam importovaných balíčků

```
import org.openqa.selenium.support.ui.FluentWait;
import org.openqa.selenium.support.ui.Wait;
import org.openqa.selenium.support.ui.ExpectedCondition;
import org.openqa.selenium.NoSuchElementException;
import org.openqa.selenium.StaleElementReferenceException;
import java.util.concurrent.TimeUnit;
```

A.6 Pokročilé možnosti

V některých případech je nutné při testování webové aplikace k ovládání využít nejen myš, ale také klávesnici. Při jiných testech pak může být potřeba využívat více oken či záložek prohlížeče zároveň. A nakonec může dojít i k situaci, kdy žádná ze tříd a metod nástroje Selenium WebDriver nevyhovuje potřebám automatizovaných testů, a je tedy nutné si vytvořit vlastní skript (např. v JavaScriptu), který potřebnou akci provede. To vše jsou pokročilé možnosti nástroje Selenium WebDriver, které jsou v této kapitole předvedeny.

5.1.2 Ovládání aplikace pomocí klávesnice

Při testování webové aplikace je někdy potřeba využít i jiný způsob ovládání než myš, a tehdy přichází ke slovu klávesnice. Selenium WebDriver pro tyto případy poskytuje třídu *Actions*, která již byla představena v kapitole A.4.2 Dvojklik, ovšem v jiném kontextu. Mimo jiné totiž obsahuje následující metody: [20 s. 59–60] [112]

- *keyDown(Keys key)*, resp. *keyDown(WebElement element, Keys key)* – simuluje stisknutí a podržení vybrané klávesy na klávesnici. Zatímco první z uvedených metod již předpokládá, že je daný element již nalezen a vybrán (popř. to není v danou chvíli relevantní), druhá metoda nejprve vyhledá zadaný element a až poté provede stisknutí dané klávesy. Na výběr jsou 3 různé klávesy - *Keys.SHIFT* (klávesa Shift), *Keys.ALT* (klávesa Alt) nebo *Keys.CONTROL* (klávesa Ctrl). Je ale potřeba mít na paměti, že součástí tohoto příkazu není uvolnění stisknuté klávesy, pro tento účel je potřeba vyvolat další akci – *keyUp()* nebo *sendKeys()*, které jsou představeny níže.
- *keyUp(Keys key)*, resp. *keyUp(WebElement element, Keys key)* – simuluje uvolnění vybrané klávesy na klávesnici (stejně jako u metody *keyDown()*, i zde existuje varianta s výběrem elementu i bez). Opět aplikovatelné pouze na klávesy (Shift, Alt nebo Ctrl, jak bylo zmíněno výše).
- *sendKeys(CharSequence keysToSend)*, resp. *sendKeys(WebElement element, CharSequence keysToSend)* – slouží k vepsání zadané textové hodnoty do vstupního pole. Pokud si čtenář všiml podobnosti s metodou *sendKeys()* volanou nad instancí třídy *WebElement*, jako tomu bylo v kapitole A.4.3 Vyplňování formulářových polí, je potřeba upozornit, že se jedná o dvě různé metody se stejnou funkcionalitou, ale trochu jiným způsobem zápisu. Využití metody *sendKeys()* v kódu automatizovaného testu je předvedeno na následujícím příkladu.

Výpis 41: Použití metody *sendKeys()* třídy *Actions* pro vyplnění vstupního pole formuláře

```
WebDriver driver = new FirefoxDriver(); //spuštění prohlížeče Firefox
driver.get("http://vse.cz/"); //otevření webové stránky na dané URL adrese
Actions builder = new Actions(driver); //vytvoření instance třídy Actions
builder.sendKeys(driver.findElement(By.name("hledejutf8")), "Studijní
řád").perform(); //nalezení elementu s atributem name="hledejutf8" a vepsání textu
"Studijní řád"
driver.findElement(By.name("search")).submit(); //odeslání formuláře
driver.quit();
```

Ještě pro připomenutí importu odpovídajícího balíčku:

Výpis 42: Import balíčku třídy *Actions*

```
import org.openqa.selenium.interactions.Actions;
```

Jak lze vidět na výše uvedeném příkladu, volání metody *sendKeys()* se následovano ještě voláním metody *perform()*, bez něj nebude akce vůbec provedena. Důvodem pro toto chování je fakt, že je někdy potřeba provést více akcí najednou, např.: [113]

Výpis 43: Provedení více akcí najednou

```
Actions builder = new Actions(driver); //vytvoření instance třídy Actions
WebElement odkaz = driver.findElement(By.linkText("Informace o VŠE")); //získání
instance třídy WebElement, v tomto případě hypertextového odkazu
builder.keyDown(Keys.CONTROL).click(odkaz).keyUp(Keys.CONTROL).build().perform();//
stisknutí klávesy Ctrl, kliknutí na element a uvolnění klávesy Ctrl - sekvence
kroků pro otevření odkazu na nové záložce
```

Nutnost zakončit akci příkazem *perform()* je společná pro všechny metody třídy *Actions*.

V případě provádění více akcí najednou je zároveň nutné před zavoláním metody *perform()* ještě metodu *build()*, která akce zkompileje tak, aby mohly být provedeny najednou – tak, jako ve výše uvedeném příkladu.

A.6.1 Práce s okny prohlížeče

Selenium WebDriver umožňuje pracovat nejen s testovanou aplikací, ale také se oknem samotného webového prohlížeče. Tato kapitola popisuje, jak okno prohlížeče maximalizovat, změnit jeho velikost, jak pracovat se záložkami, jak otevřít odkaz v novém okně a jak pracovat s pop-up okny.

Maximalizace okna prohlížeče

Okno lze jednoduše maximalizovat pomocí příkazu: [21 s. 56]

Výpis 44: Maximalizace okna prohlížeče

```
driver.manage().window().maximize();
```

Změna velikosti okna prohlížeče

Minimalizaci okna WebDriver neumožňuje, což ale při provádění automatizovaných testů nepředstavuje žádný problém – minimalizace okna by nepřinesla žádný užitek. Pomocí metody *setSize()* lze však nastavit velikost okna, což se hodí například pro ověření, že se aplikace zobrazuje správně i na menších obrazovkách. Využití metody *setSize()* je následující: [114]

Výpis 45: Nastavení přesných rozměrů okna prohlížeče

```
driver.manage().window().setSize(new Dimension(800,700));
```

Přičemž první číselná hodnota označuje výšku a druhá šířku okna.

Práce se záložkami v okně prohlížeče

Další užitečnou funkcionalitou je možnost přepínání mezi záložkami prohlížeče – v některých případech je totiž k otestování aplikace potřeba pracovat na více záložkách zároveň. Tuto funkcionalitu však WebDriver zatím nepodporuje, a tak je potřeba si vypomoci simulací klávesových zkratk pro ovládání prohlížeče. Otevření nové prázdné záložky lze provést následovně: [115]

Výpis 46: Otevření nové prázdné záložky v okně prohlížeče

```
driver.findElement(By.tagName("html")).sendKeys(Keys.CONTROL + "t");
```

Přičemž vyhledání elementu html slouží pouze k tomu, aby bylo možné aplikovat metodu *sendKeys()*, která simuluje stisknutí klávesy Ctrl a písmene T, což je klávesová zkratka pro otevření nové záložky ve webovém prohlížeči. WebDriver se zároveň na nově otevřenou záložku přepne a lze tak plynule pokračovat v práci.

Pro návrat na původní záložku, popř. přepínání mezi záložkami, pak lze využít další klávesovou zkratku – Ctrl + Tab. Jejich využití v kódu automatizovaného testu je následující: [115]

Výpis 47: Přepínání mezi záložkami v okně prohlížeče

```
driver.findElement(By.tagName("html")).sendKeys(Keys.CONTROL + "\t");
```

Kde `\t` označuje klávesu Tab.

Zavřít aktuální záložku pak lze pomocí stisknutí kláves Ctrl a písmene W. Kód automatizovaného testu pak vypadá takto: [116]

Výpis 48: Zavření aktuálně otevřené záložky v okně prohlížeče

```
driver.findElement(By.tagName("html")).sendKeys(Keys.CONTROL + "w");
```

Otevření odkazu v novém okně prohlížeče

Existují případy, kdy je při testování aplikace potřeba otevřít nějaký odkaz ve zcela novém okně prohlížeče. Tuto akci lze provést pomocí instance třídy *Actions*, která již byla představena výše. Její využití v tomto případě je uvedeno v následujícím kódu:

Výpis 49: Otevření odkazu v novém okně prohlížeče

```
WebDriver driver = new FirefoxDriver(); //spuštění prohlížeče Firefox
driver.get("http://www.vse.cz"); //otevření webové stránky www.vse.cz
String puvodniOkno = driver.getWindowHandle(); //získání identifikátoru původního
okna
WebElement odkaz = driver.findElement(By.linkText("Informace o VŠE")); //nalezení
odkazu
Actions builder = new Actions(driver);
builder.keyDown(Keys.SHIFT).click(odkaz).keyUp(Keys.SHIFT).build().perform();
//otevření odkazu v novém okně
```

```
String noveOkno = driver.getWindowHandle(); //získání identifikátoru nového okna
...
driver.switchTo().window(puvodniOkno); //přepnutí do původního okna
...
driver.switchTo().window(noveOkno); //přepnutí do nového okna
driver.quit();
```

Příkaz `driver.getWindowHandle()` vrací textový řetězec jednoznačně identifikující aktivní okno. Tento identifikátor lze pak využít pro pohyb mezi otevřenými okny, neboť slouží jako parametr při volání metody `driver.switchTo().window(String identifikátor)`, která zajistí přepnutí na dané okno.

Dále je potřeba mít na paměti, že po otevření odkazu v novém okně se *driver* (instance třídy `WebDriver`) automaticky přepne do nového okna a veškeré následující akce jsou dále prováděny v něm. Poté se lze samozřejmě kdykoli přepnout do jiného okna.

Jakékoli z otevřených oken lze zavřít příkazem `driver.close()`, který zavře aktivní okno, ale ostatní okna ponechá otevřená. Je pouze potřeba si ohlídat, ve kterém okně se právě *driver* nachází, což lze měnit pomocí již zmíněného příkazu `driver.switchTo().window(String identifikátor)`.

Práce s pop-up okny

Některé aplikace navíc využívají tzv. pop-up okna, což jsou automaticky otevřená okna poskytující nějakou informaci nebo vyžadující vstup od uživatele. Práce s nimi je podobná jako v předchozí kapitole věnované otevírání odkazu v novém okně prohlížeče, zde ale není potřeba třídy *Actions* pro simulaci stisku klávesy Shift, neboť se pop-up okno v testované aplikaci otevře buď samovolně, nebo po nějaké jiné běžné akci (např. kliknutí na tlačítko). Práce s pop-up okny je ukázána na následujícím modelovém příkladu: [20 s. 65–66]

Výpis 50: Ukázka práce s pop-up okny

```
WebDriver driver = new FirefoxDriver(); //spuštění prohlížeče Firefox
    driver.get("http://www.idnes.cz/"); //otevření webové stránky
    String puvodniOkno = driver.getWindowHandle(); //získání identifikátoru
původního okna
    driver.findElement(By.linkText("Práce v mediální skupině")).click();
//kliknutí na odkaz, který vyvolá otevření pop-up okna
    Set<String> allWindows = driver.getWindowHandles(); //získání
identifikátorů všech otevřených oken
    Assert.assertTrue("Pop-up okno nebylo otevřeno!", allWindows.size()>1);
//ověření, zda bylo otevřeno nové okno
    for (String windowId : allWindows) { //procházení všech otevřených oken
        if (driver.switchTo().window(windowId).getTitle().equals("Nabídka práce
- jobDNES.cz")) { //pokud se titulek stránky shoduje s očekávanou hodnotou
            driver.close(); //zavření pop-up okna
            break; //ukončení cyklu
```

```
    }  
  }  
  driver.switchTo().window(puvodniOkno); //přepnutí driveru do původního okna  
  Assert.assertTrue(driver.getTitle().equals("iDNES.cz - zprávy, kterým  
můžete věřit")); //ověření správnosti titulku stránky  
  driver.quit(); //uzavření prohlížeče
```

Princip fungování je tedy stejný jako při otevírání odkazu v novém okně. V obou případech je také nutné nezapomenout, že je potřeba po uzavření jednoho z oken přepnout *driver* do jiného, právě otevřeného okna, jinak je vyhozena výjimka *NoSuchWindowException*.

Práce s dialogovými okny

Často se lze také setkat s tím, že testovaná aplikace v určitých situacích zobrazuje dialogová okna s nějakou informací (potvrzovací nebo chybovou hláškou), dotazem vyžadujícím akci od uživatele (obvykle stisknutí některého ze zobrazených tlačítek) nebo vstupními poli. I na tyto případy Selenium WebDriver pamatuje a k práci s nimi poskytuje rozhraní *Alert*. Jeho využití je následující:

Výpis 51: Přepnutí driveru na otevřené dialogové okno

```
Alert alert = driver.switchTo().alert();
```

Pro úspěšné spuštění testu využívajícího rozhraní *Alert* je potřeba nainportovat následující knihovnu: `org.openqa.selenium.Alert`.

V případě, že se na stránce v danou chvíli žádné dialogové okno nenachází, je vyhozena výjimka *NoAlertPresentException*.

Nad instancí *alert* pak lze zavolat čtyři různé metody:

- *accept()* – stiskne tlačítko *OK* v daném výstražném okně;
- *dismiss()* – stiskne tlačítko *Cancel* v daném výstražném okně;
- *getText()* – vrací textovou hodnotu výstražného okna;
- *sendKeys(String keysToSend)* – vepíše zadaný textový řetězec do vstupního pole výstražného okna;

Nyní je navíc ve vývoji také metoda *authenticateUsing(Credentials credentials)*, která umožňuje do dialogového okna zadávat přihlašovací jméno a heslo, např. `alert.authenticateUsing(new UserAndPassword("login", "password"))`; V tomto případě je potřeba nainportovat knihovnu `org.openqa.selenium.security.UserAndPassword`.

A.6.2 Provádění JavaScript kódu

Nástroj Selenium WebDriver umožňuje vedle všech výše uvedených metod rozhraní WebDriver také spouštět zdrojový kód napsaný v jazyku JavaScript a ovládat tak webovou aplikaci i tímto alternativním způsobem. Pro tento účel slouží rozhraní JavascriptExecutor, které může být použito pro spuštění libovolného JavaScript kódu uvnitř internetového prohlížeče, přičemž jeho nespornou výhodou je možnost provádět pokročilé operace, které doposud nejsou rozhraním Selenium WebDriver přímo podporovány. [21 s. 51]

Jednoduchou ukázkou použití JavaScript kódu v automatizovaném testu lze vidět na následujícím příkladu:

Výpis 52: Provádění JavaScript kódu v automatizovaném testu nástroje WebDriver

```
driver.get("http://www.vse.cz"); //otevření URL adresy
JavascriptExecutor js = (JavascriptExecutor) driver; //vytvoření instance
JavascriptExecutoru
String titulekStranky = (String) js.executeScript("return document.title");
//spuštění JavaScript kódu, který vrací titulek webové stránky
Assert.assertTrue(titulekStranky.equals("Vysoká škola ekonomická v Praze | VŠE"));
//ověření, zda titulek webové stránky nese správnou textovou hodnotu
```

Pro úspěšné sestavení zdrojového kódu je však potřeba nejprve naimportovat knihovnu `org.openqa.selenium.JavascriptExecutor`.

Uvedený příklad je samozřejmě pouze ilustrativní, získat titulek webové stránky lze velice jednoduše prostřednictvím příkazu `driver.getTitle()`. Existují však situace, ve kterých již standardní funkcionality nástroje Selenium WebDriver nedostačuje, a je tedy potřeba sáhnout po alternativním řešení využívajícím JavaScript.

Dalším způsobem, jak nalézt element na webové stránce, je pomocí JQuery selektorů. Jejich výhodou je, že jsou založeny na CSS selektorech verze 1 až 3, které doplňují o další užitečné selektory, přičemž je lze využívat nezávisle na použitém internetovém prohlížeči – tedy i v případech, kdy vybraný internetový prohlížeč daný CSS selektor nepodporuje. [21 s. 35]

Výpis 53: Využití JQuery selektoru pro nalezení elementu

```
JavascriptExecutor js = (JavascriptExecutor) driver; //vytvoření instance
JavascriptExecutoru
List<WebElement> elements = (List<WebElement>) js.executeScript("return
jQuery.find(':checked')"); //spuštění JavaScript kódu, který vrací všechna označená
zaškrťávací políčka nebo přepínače
```

A.7 Záznam průběhu testů

Pro účely analýzy výsledků již proběhlého testu se často může hodit nějaká dodatečná informace o tom, co se vlastně v testované aplikaci dělo a zda test probíhal očekávaným způsobem. V případě testů, jejichž běh skončil chybou, pak je nutné zjistit, zda šlo skutečně o chybu v testované aplikaci, nebo byla chyba na straně testu. Vedle standardní chybové hlášky s informací o typu výjimky, která v průběhu testu nastala, je užitečné mít k dispozici i obraz stavu aplikace ve chvíli, kdy k výjimce došlo. V následujícím textu této kapitoly je představen postup, jak v určitém momentu průběhu testu pořídit snímek obrazovky a dále také jak vytvořit videozáznam, kterým je možné zachytit průběh celého testu.

A.7.1 Snímek obrazovky

Jak již bylo zmíněno výše, mnohdy je užitečné mít přehled o tom, co se v průběhu automatizovaného testu v okně prohlížeče dělo (zejména, pokud je test spouštěn automaticky v průběhu noci, kdy jsou stroje méně vytížené, což je obvyklá praxe). K tomu lze využít snímky pořízené v různých etapách testu či v okamžiku vyhození výjimky. Pořízení snímku obrazovky je ilustrováno na následujícím příkladu: [117]

Výpis 54: Pořízení snímku obrazovky

```
WebDriver driver = new FirefoxDriver(); //spuštění prohlížeče Firefox
driver.get("http://vse.cz/"); //otevření webové stránky
try {
    File snimek = ((TakesScreenshot) driver).getScreenshotAs(OutputType.FILE);
    //pořízení snímku obrazovky
    String nazevSouboru = "Snimek " + new SimpleDateFormat("yyyy-mm-dd hh-mm-ss").format(new Date()).toString(); //definice názvu výstupního souboru
    FileUtils.copyFile(snimek, new File("C:\\Users\\<username>\\Pictures\\Test Screenshots\\" + nazevSouboru + ".png")); //uložení souboru na zadané místo na disku
}
catch (IOException e) {
    e.printStackTrace();
    Assert.fail();
}
driver.quit(); //ukončení prohlížeče
```

O samotné pořízení snímku se stará jediný řádek kódu: `File snimek = ((TakesScreenshot) driver).getScreenshotAs(OutputType.FILE);`

Snímku je ale vhodné přiřadit název, který v sobě obsahuje datum a čas vytvoření, což v uvedeném příkladu zajišťuje řádek `String nazevSouboru = "Snimek " + new SimpleDateFormat("yyyy-mm-dd hh-mm-ss").format(new Date()).toString();` Zde je však potřeba dát pozor na formát času v názvu souboru – ačkoli je zvykem jej psát ve formátu hh:mm:ss, kód v takovém případě vyhodí výjimku *IOException* s upozorněním na nesprávný název souboru, proto jsou v uvedeném příkladu dvojtečky nahrazeny pomlčkou.

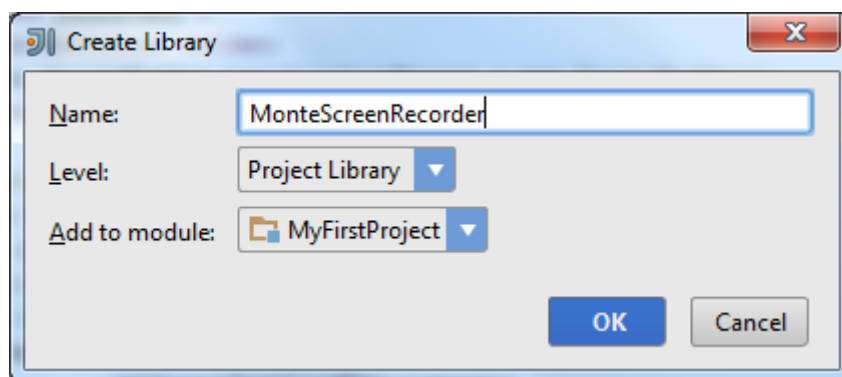
A nakonec již jen zbývá soubor uložit na nějaké zadané místo na disku (v tuto chvíli je totiž umístěn ve složce s dočasnými soubory, typicky `C:\Users\\AppData\Local\Temp\`), a to pomocí následujícího řádku kódu: `FileUtils.copyFile(snimek, new File("C:\\Users\\<username>\\Pictures\\Test Screenshots\\"+"navezSouboru+".png"))`; Cestu je samozřejmě možné nadefinovat vlastní. V případě, že některá složka na zadané cestě neexistuje, program si ji při ukládání souboru automaticky vytvoří.

A.7.2 Videozáznam

Dalším, ještě podrobnějším způsobem jak zaznamenat, co se v průběhu testu v testované aplikaci dělo, je pořízení videozáznamu. Vzhledem k tomu, že automatizované testy obvykle probíhají bez dohledu člověka, je pro následnou analýzu výsledků užitečné mít k dispozici důkaz, jak test probíhal a zda nedošlo k nějaké neočekávané situaci. Tato informace je také užitečná pro reprodukci případných chyb. Videozáznam však nemusí sloužit jen pro účely testování – může být využit i jako výukové video nebo pro prezentaci produktu zákazníkovi. [21 s. 257]

Nástroj Selenium WebDriver sice nedisponuje funkcionalitou pro záznam videa, nicméně jej lze rozšířit pomocí open-source nástrojů, které již videa pořídít dokáží. V této kapitole je popsán návod, jak této funkcionality dosáhnout pomocí nástroje Monte Media Library pro jazyk Java. Existují však i řešení pro další programovací jazyky, např. pro .NET je k dispozici Microsoft Expression Encoder a pro Python lze využít nástroj Castro. [21 s. 257]

Nejprve je potřeba ze stránek <http://www.randelshofer.ch/monte/> stáhnout soubor `ScreenRecorder.jar` a umístit ho kamkoli do složky s projektem. V prostředí IntelliJ Idea pak uživatel na soubor klikne pravým tlačítkem myši a zvolí možnost *Add as Library...* a v následujícím dialogovém okně zvolí název knihovny a modul, kterým je název aktuálního projektu (viz Obrázek 27).



Obrázek 27: Volba názvu knihovny pro nástroj Monte Screen Recorder

Zdrojový kód testu využívajícího nástroj Monte Screen Recorder pro pořízení videozáznamu průběhu testu je následující: [21 s. 258–261][118]

Výpis 55: Pořízení videozáznamu testu s pomocí nástroje Monte Screen Recorder

```
import org.junit.*;
import org.monte.media.Format;
import org.monte.media.math.Rational;
import org.monte.screenrecorder.ScreenRecorder;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.firefox.FirefoxDriver;

import java.awt.*;

import static org.junit.Assert.fail;
import static org.monte.media.AudioFormatKeys.*;
import static org.monte.media.VideoFormatKeys.*;

public class Nahravani {
    private WebDriver driver;
    private StringBuffer verificationErrors = new StringBuffer();
    private ScreenRecorder screenRecorder;

    @Before
    public void setUp() throws Exception {
        GraphicsConfiguration gc = GraphicsEnvironment
            .getLocalGraphicsEnvironment()
            .getDefaultScreenDevice()
            .getDefaultConfiguration();
        screenRecorder = new ScreenRecorder(gc,
            gc.getBounds(),
            new Format(MediaTypeKey, MediaType.FILE, MimeTypeKey, MIME_AVI),
            new Format(MediaTypeKey, MediaType.VIDEO, EncodingKey,
ENCODING_AVI_TECHSMITH_SCREEN_CAPTURE,
                CompressorNameKey, ENCODING_AVI_TECHSMITH_SCREEN_CAPTURE,
                DepthKey, 24, FrameRateKey, Rational.valueOf(15),
                QualityKey, 1.0f,
                KeyFrameIntervalKey, 15 * 60),
            new Format(MediaTypeKey, MediaType.VIDEO, EncodingKey, "black",
FrameRateKey, Rational.valueOf(30)),
            null,
            new File("C:\\Users\\<username>\\Desktop"));
        driver = new FirefoxDriver();
        screenRecorder.start();
    }

    @Test
    public void basicTest() {
        driver.get("http://www.vse.cz");
    }
}
```

```

Assert.assertTrue(driver.getTitle().contains("Vysoká škola ekonomická"));
driver.findElement(By.name("hledejutf8")).sendKeys("Studijní řád");
driver.findElement(By.name("search")).submit();
Assert.assertTrue(driver.getTitle().equals("Vyhledávání"));
}

@After
public void tearDown() throws Exception {
    driver.quit();
    screenRecorder.stop();
    String verificationErrorString = verificationErrors.toString();
    if (!"".equals(verificationErrorString)) {
        fail(verificationErrorString);
    }
}
}

```

Kód je rozdělen na 3 části - `@Before`, `@Test` a `@After`. Tyto anotace vyjadřují, že kód označený `@Before` se má spustit před každým spuštěním testovací metody označené `@Test` a anotace `@After` naopak označuje kód, který se má spustit po každém běhu testovací metody. To je užitečné zejména v případě, kdy v projektu existuje více testovacích metod – není tak potřeba stále opakovat ten samý kód v každé testovací metodě.

A nyní již k vysvětlení částí výše uvedeného kódu. Nejprve je potřeba získat informace o grafické konfiguraci obrazovky, na které bude průběh testu nahráván. Implicitně se nahrává obraz z první obrazovky. [21 s. 261]

Výpis 56: Získání informací o grafické konfiguraci obrazovky

```

GraphicsConfiguration gc = GraphicsEnvironment
    .getLocalGraphicsEnvironment()
    .getDefaultScreenDevice()
    .getDefaultConfiguration();

```

Pokud je však počítač připojen k více než jednomu monitoru, může nastat situace, že je potřeba pořídit záznam obrazovky druhé (či kterékoli jiné). Volby obrazovky lze docílit následujícím kódem: [21 s. 262]

Výpis 57: Pořízení videozáznamu druhé obrazovky

```

GraphicsConfiguration gc = GraphicsEnvironment//
    .getLocalGraphicsEnvironment()//
    .getScreenDevices()[1]
    .getDefaultConfiguration();

```

Kde [1] označuje index pořadí obrazovky (0 = první obrazovka, 1 = druhá obrazovka atd.).

Následuje iniciace instance `ScreenRecorderu`. Zde lze definovat typ výstupního formátu (AVI, MOV, MP4 a další), barvu kurzoru, formát nahrávaného audia (v následujícím příkladu `null` = bez nahrávání audia, bohužel se mi nepodařilo zjistit, jak jej zprovoznit) a složku, do které se

má vytvořený videozáznam uložit (zde `C:\Users\<username>\Videos\Videozaznamy testu`, cestu je samozřejmě potřeba upravit). [118]

Výpis 58: Vytvoření instance `ScreenRecorder`

```
screenRecorder = new ScreenRecorder(gc,
    gc.getBounds(),
    new Format(MediaTypeKey, MediaType.FILE, MimeTypeKey, MIME_AVI),
    new Format(MediaTypeKey, MediaType.VIDEO, EncodingKey,
ENCODING_AVI_TECHSMITH_SCREEN_CAPTURE, CompressorNameKey,
ENCODING_AVI_TECHSMITH_SCREEN_CAPTURE, DepthKey, 24, FrameRateKey,
Rational.valueOf(15), QualityKey, 1.0f, KeyFrameIntervalKey, 15 * 60),
    new Format(MediaTypeKey, MediaType.VIDEO, EncodingKey, "black",
FrameRateKey, Rational.valueOf(30)),
    null,
    new File("C:\\Users\\<username>\\Videos\\Videozaznamy testu"));
```

Pro nahrávání videa ve formátu MOV stačí ve výše uvedeném kódu vyměnit `MIME_AVI` za `MIME_QUICKTIME` a `ENCODING_AVI_TECHSMITH_SCREEN_CAPTURE` za `ENCODING_QUICKTIME_JPEG`. [119]

Kompletní seznam možných nastavení lze nalézt v oficiální dokumentaci nástroje Monte ScreenRecorder: <http://www.randelshofer.ch/monte/javadoc/constant-values.html> [120]

Pro přehrání videa je však potřeba speciální kodek – Techsmith Screen Capture Codec (TSCC Codec), který lze stáhnout na adrese <https://www.techsmith.com/download.html> (nachází se až téměř na konci stránky pod názvem TSCC® Codec). Po instalaci kodeku již lze video spustit v jakémkoli přehrávači, který podporuje daný formát videa. [21 s. 258]

A.8 Architektura automatizovaných testů

V předchozích kapitolách bylo vysvětleno, jak pomocí nástroje Selenium WebDriver pracovat s webovou aplikací a jak řešit případné problémy. Čtenář by již tedy měl být v tuto chvíli schopen vytvořit celou řadu komplexních automatizovaných testů. Na řadu ale přichází otázka, jak tyto testy strukturovat, aby jejich následná údržba nebyla příliš nákladná. Často se totiž stává, že se testovaná aplikace změní, testy tak přestanou plnit svou funkci a je potřeba je upravit. Pokud se například změní vlastnosti (např. atribut `Id`) určitého elementu, který je používán v několika desítkách či stovkách testů, je nutné změnit selektor na všech místech kódu, kde je element vyhledáván. A to znamená spoustu práce. [20 s. 191]

Naštěstí však existuje způsob, jak selektory pro vyhledávání elementů soustředit na jednom místě a v případě změny vlastností prvku na webové stránce je pak potřeba změnit jen jeden řádek kódu. K tomu slouží návrhový vzor `PageObject` v kombinaci se třídou `PageFactory`. Hlavní ideou tohoto přístupu je oddělit od sebe kód automatizovaného testu a popis webové stránky. Každá webová stránka dané aplikace tak má svou vlastní třídu, v níž jsou na jednom

místě uvedeny selektory pro vyhledání elementů na dané webové stránce a akce, které je možné na webové stránce provést, jsou poskytovány jako veřejné (angl. public) metody. Tyto metody jsou pak dle potřeby volány z kódu automatizovaných testů, umístěných v jiné třídě, dedikované pro popis sekvence kroků testu. [20 s. 196]

Pro popis elementů na webové stránce se využívá anotace `@FindBy`, kterou lze zapsat dvěma různými způsoby: [20 s. 197]

Výpis 59: První varianta zápisu vyhledání elementu na webové stránce pomocí anotace `@FindBy`

```
@FindBy(id="logo")
WebElement logo;
```

nebo

Výpis 60: Druhá varianta zápisu vyhledání elementu na webové stránce pomocí anotace `@FindBy`

```
@FindBy(how=How.ID, using="logo")
WebElement logo;
```

Oba dva způsoby zápisu jsou správné a fungují stejným způsobem, je tedy jen na uživateli, který z nich preferuje.

Aby však došlo k použití výše uvedených anotací, je potřeba při vytváření instance dané třídy, která obsahuje popis elementů na webové stránce, využít `PageFactory`:

Výpis 61: Inicializace elementů na webové stránce s využitím `PageFactory`

```
Stranka stranka = PageFactory.initElements(driver, Stranka.class);
```

Pokud by byla instance stránky vytvořena pouze pomocí konstruktoru (`new Stranka(driver)`), test by při pokusu o využití některého elementu vytvořeného pomocí anotace `@FindBy` vyhodil výjimku `NullPointerException`, neboť nedošlo k iniciaci dané proměnné a ta tedy obsahuje hodnotu `NULL`. Proto je nutné nezapomínat na volání metody `PageFactory.initElements()`.

Využití návrhového vzoru `PageObject` a třídy `PageFactory` je předvedeno na následujícím příkladu, skládajícím se z jedné třídy popisující hlavní stránku webu VŠE a druhé třídy obsahující testovací skript:

Výpis 62: Třída s popisem několika elementů na hlavní stránce webu VŠE a metodami pro práci s nimi

```
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.support.FindBy;

public class MainPageVSE {

    private WebDriver driver;

    @FindBy(id = "logo")
```

```
private WebElement logo;

@FindBy(name = "hledejutf8")
private WebElement vyhledavaciPole;

@FindBy(css = "form[name='search']")
private WebElement vyhledavaciFormular;

public MainPageVSE(WebDriver driver) {
    this.driver = driver;
}

public boolean jeLogoZobrazeno() {
    return logo.isDisplayed();
}

public boolean vyhledej(String hledanyVyras) {
    vyhledavaciPole.sendKeys(hledanyVyras);
    vyhledavaciFormular.submit();
    return driver.getTitle().equals("Vyhledávání");
}
}
```

Výpis 63: Třída s testovacím skriptem

```
import org.junit.Assert;
import org.junit.Test;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.firefox.FirefoxDriver;
import org.openqa.selenium.support.PageFactory;

public class BasicTest {

    @Test
    public void basicTest() {
        WebDriver driver = new FirefoxDriver();
        driver.get("http://www.vse.cz");
        MainPageVSE mainPageVSE = PageFactory.initElements(driver,
MainPageVSE.class); //inicializace elementů třídy MainPageVSE
        Assert.assertTrue(mainPageVSE.jeLogoZobrazeno()); //kontrola, zda je
zobrazeno logo
        Assert.assertTrue(mainPageVSE.vyhledej("Studijní řád")); //test vyhledání
textu Studijní řád
        driver.quit();
    }
}
```

Jak je vidět na výše uvedených příkladech, popis elementů webové stránky je striktně oddělen od testovacího skriptu a veškerá práce s nimi je prováděna prostřednictvím metod třídy

popisující webovou stránku. Z toho důvodu jsou proměnné nesoucí elementy označené jako soukromé (angl. private), aby k nim nebylo možné přistupovat přímo.

Při návrhu architektury testů je vhodné následovat nejlepší praktiky (angl. best practices), podle kterých by se mělo na třídu popisující webovou stránku nahlížet jako na poskytovatele služeb. Těmito službami jsou myšleny metody, prostřednictvím kterých pak může testovací skript získávat potřebné informace a provádět požadované akce. [20 s. 199]

Při návrhu testů je tedy potřeba se v první řadě na webovou stránku podívat a zamyslet se, jaké služby uživateli poskytuje. Může to být vyhledání nějakého výrazu, poskytnutí kontaktních informací, zápis kurzu apod. Podle těchto služeb pak lze připravit odpovídající metody. Při návrhu testů je ale nutné nezaměňovat služby s akcemi uživatele, mezi které patří např. kliknutí na tlačítko, vepsání textu do pole, kliknutí na odkaz, vybrání možnosti z rozevíracího seznamu apod. To jsou již příliš detailní kroky, které by měly být před testovacím skriptem skryty a raději by měly být součástí celé služby. Třída s testovacím skriptem by tedy nikdy neměla obsahovat kroky jako „vyplň text ‚Studijní řád‘ do vyhledávacího pole a klikni na tlačítko Hledej“, ale pouze volání metody pro vyhledávání zadaného textu. Tímto přístupem lze významně redukovat komplexitu testovacích skriptů a usnadnit tak jejich udržování. [20 s. 204]

Hlavním smyslem návrhového vzoru PageObject je oddělit od sebe popis webové stránky a kód automatizovaných skriptů. Pokud tedy dojde ke změně v testované aplikaci, ale požadavky na chování aplikace zůstanou stejné, měla by být upravena pouze třída popisující webovou stránku, nikoli testovací skript. Kód samotných testů by se tedy měl měnit pouze na základě změny požadavků od zákazníka. [20 s. 207]

A.9 Automatizované spouštění testů

Automatizace testování je dovedena k dokonalosti až tehdy, když testy nejen že probíhají, ale také se spouští bez nutnosti zásahu člověka. V ideálním případě by se měly testy spouštět buď po každém sestavení projektu (angl. build) testované aplikace nebo alespoň 1x denně v době, kdy stroje nejsou vytíženy – typicky v noci. Ať v prvním či druhém případě by však měly být testy spuštěny automaticky. K tomuto účelu slouží nástroje pro kontinuální integraci, pro jazyk Java je velmi oblíbený Jenkins. Využít lze ale také jiné nástroje – např. Travis, Circle, Bamboo a další. [121] [122]

Popis nastavení nástroje Jenkins pro automatické spouštění Selenium testů je však již mimo rozsah této diplomové práce, pro více informací doporučuji nastudovat dokument, který je online přílohou knihy *Selenium Testing Tools Cookbook* od autora Unmesh Gundecha vydané v roce 2012. Tento dokument, který popisuje možnosti propojení nástroje Selenium WebDriver s dalšími nástroji, např. pro kontinuální integraci aplikace, je k dispozici na adrese https://www.packtpub.com/sites/default/files/downloads/Integration_with_Other_Tools.pdf [123]