VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

AUTOMATICKÉ TESTOVÁNÍ PROJEKTU JAVASCRIPT RESTRICTOR

AUTOMATIC TESTING OF JAVASCRIPT RESTRICTOR PROJECT

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. MARTIN BEDNÁŘ**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. LIBOR POLČÁK, Ph.D.**
SUPERVISOR

BRNO 2020

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2019/2020

Zadání diplomové práce

Student: **Bednář Martin, Bc.**
Program: Informační technologie Obor: Informační systémy
Název: **Automatické testování projektu JavaScript Restrictor**
Automatic Testing of JavaScript Restrictor Project
Kategorie: Web

Zadání:

1. Seznamte se s projektem JavaScript Restrictor.
2. Seznamte se s problematikou testování, projektem Selenium, nástrojem Jasmine a dalšími nástroji vhodnými pro testování webových prohlížečů. U testovacích nástrojů se zaměřte také na podporu různých operačních systémů, verzí prohlížeče a možnosti paralelizace testování (např. Selenium Grid).
3. Nastudujte metody procházení nejnavštěvovanějších stránek a automatizovaného zjišťování očekávaného chování.
4. Navrhněte automatizované testy pro projekt JavaScript Restrictor. Zaměřte se jak na jednotkové testy kódu rozšíření JavaScript Restrictor, tak na zachování funkcionality nejčastěji navštěvovaných webových stránek.
5. Testy implementujte kvalitním a komentovaným kódem, aby byl vedoucím práce akceptován do projektu JavaScript Restrictor. Pro testování zachování funkcionality nejčastěji navštěvovaných webových stránek implementujte distribuované paralelní testování.
6. Práci vyhodnoťte a navrhněte možná zlepšení.

Literatura:

- TIMKO, Martin. *Vylepšení rozšíření pro omezení volání JavaScriptu*. Brno, 2019. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií.
- BASTL, Vojtěch. *Automatizace webového prohlížeče*. Brno, 2019. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií.
- SLÁMOVÁ, Hana. *Refaktorizace síťového forenzního nástroje Netfox Detective*. Brno, 2018. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií.

Při obhajobě semestrální části projektu je požadováno:

- Body 1 až 4 zadání.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Polčák Libor, Ing., Ph.D.**
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2019
Datum odevzdání: 31. července 2020
Datum schválení: 8. ledna 2020

Abstrakt

Cílem této práce bylo navrhnout, implementovat a vyhodnotit výsledky automatických testů pro projekt JavaScript Restrictor, který je vyvíjen jako rozšíření do webových prohlížečů. Testy jsou rozděleny do tří úrovní – jednotkové, integrační a systémové. Jednotkové testy ověřují chování jednotlivých funkcí, integrační testy ověřují správné obalování koncových bodů rozhraní prohlížeče a systémové testy kontrolují, zda rozšíření nepotlačuje chtěnou funkcionalitu webových stránek. Systémové testy jsou implementovány pro paralelní spouštění na distribuovaném prostředí, čímž se podařilo dosáhnout téměř přímo úměrného snížení časové náročnosti vzhledem k počtu testovacích uzlů. Přínosem této práce je odhalení dosud neznámých chyb v rozšíření JavaScript Restrictor a poskytnutí potřebných informací, díky nimž bylo možné část zjištěných chyb již opravit.

Abstract

The aim of the thesis was to design, implement and evaluate the results of automatic tests for the JavaScript Restrictor project, which is being developed as a web browser extension. The tests are divided into three levels – unit, integration, and system. The Unit Tests verify the behavior of individual features, the Integration Tests verify the correct wrapping of browser API endpoints, and the System Tests check that the extension does not suppress the desired functionality of web pages. The System Tests are implemented for parallel execution in a distributed environment which has succeeded in achieving an almost directly proportional reduction in time with respect to the number of the tested nodes. The benefit of this work is detection of previously unknown errors in the JavaScript Restrictor extension and provision of the necessary information that allowed to fix some of the detected bugs.

Klíčová slova

testování, paralelní, distribuované, automatické, JavaScript, Restrictor, Selenium, automatizace, integrační testy, podvrhnutí, webový prohlížeč, Google Chrome, Mozilla Firefox, WebDriver, Grid, Selenese, systémové testy, funkční testování, uživatelského rozhraní, Jasmine, jednotkové testy, nejnavštěvovanější webové stránky, žebříček, Alexa, Cisco Umbrella, Majestic, Quantcast, TRANCO, NetMonitor, záznamy v konzoli, Levenshteinova vzdálenost, Kosinová podobnost, snímky obrazovky, porovnání

Keywords

testing, parallel, distributed, automatic, JavaScript, Restrictor, Selenium, automatization, integration tests, spoofing, web browser, Google Chrome, Mozilla Firefox, WebDriver, Grid, Selenese, system tests, functional testing, user interface, Jasmine, unit tests, the most visited website, page rank, Alexa, Cisco Umbrella, Majestic, Quantcast, TRANCO, NetMonitor, logs, Levenshtein distance, Cosine similarity, screenshots, comparison

Citace

BEDNÁŘ, Martin. *Automatické testování projektu JavaScript Restrictor*. Brno, 2020. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Libor Polčák, Ph.D.

Automatické testování projektu JavaScript Restrictor

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Libora Polčáka Ph.D. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Martin Bednář
30. července 2020

Poděkování

Rád bych vyjádřil velkou vděčnost a poděkování vedoucímu své diplomové práce Ing. Liboru Polčákovi Ph.D. za jeho odbornou konzultaci, ale i za jeho ochotu a časovou flexibilitu. Velké poděkování patří mým rodičů, kteří mě celé studium podporovali.

Obsah

Seznam obrázků

Kapitola 1

Úvod

Testování, kterým se tato práce zabývá, má nezastupitelné místo při každém vývoji softwarového produktu. Cílem testování softwaru je ověření, že se produkt podařilo nejen správně naprogramovat, ale také i to, že byl vytvořen správný produkt.

S rozvojem softwarového inženýrství v 70. letech minulého století se začalo systematičtěji přistupovat i k testování výsledných produktů. Vznikaly metodiky a doporučení, začaly se vyvíjet nástroje pro testování. Od té doby zaznamenalo testování progresivní rozvoj – v dnešní době existují nástroje a rámce pro testování nabízející pokročilé možnosti pravidelného automatického spouštění a provádění testů.

Na FIT VUT v Brně je vyvíjen výzkumný projekt JavaScript Restrictor, který slouží k omezení možností sledování uživatele při prohlížení webových stránek (sledování jeho pohybu na internetu). Tento projekt, publikovaný jako rozšíření do webových prohlížečů, funguje na principu, že zamezuje navštíveným webovým stránkám v přístupu k informacím o počítači a uživateli tím, že buď podvrhne webové stránce falešná data nebo neposkytne data žádná.

Cílem práce je navrhnout a implementovat automatické testy pro výzkumný projekt JavaScript Restrictor, které se budou zaměřovat na otestování, že chování rozšíření odpovídá specifikaci a že nedochází k omezení chtěné funkcionality webových stránek. V praxi by mělo použití testů vypadat tak, že se pravidelně a zejména před vydáním nové verze rozšíření spustí automatické testy, které ověří chování rozšíření specifikované v dokumentaci a také zjistí případné problémy při načítání webových stránek, které může způsobovat JavaScript Restrictor. Výstupem bude zpráva, která informuje vývojáře o úspěšnosti nebo neúspěšnosti proběhlých testů a v případě selhání testu také o podrobnějších informacích vedoucích na možnou příčinu selhání.

Díky automatickým testům, které vzniknou v rámci této práce, bude možné vyvíjet JavaScript Restrictor s větší jistotou, že nově vydaná verze nezpůsobí neočekávané omezení chtěné funkcionality na webových stránkách a že údaje z otisku prohlížeče sloužící k identifikaci uživatele jsou skutečně maskované dle nastavení.

Práce je rozdělena do několika kapitol, které postupně popisují řešení problému testování projektu JavaScript Restrictor. V následujících dvou kapitolách je rozvedena problematika testování softwaru. Čtvrtá kapitola představuje žebříčky nejnavštěvovanějších webových stránek používaných jako vzorek webu. V páté kapitole jsou představeny metody pro testování webových stránek. Na základě teoretických znalostí z těchto kapitol je v následující šesté kapitole zhodnocen současný stav, identifikován problém a je navrženo jeho řešení v podobě automatických testů. V další kapitole je popsána implementace. Práce končí vyhodnocením výsledků testů, popisem přínosu a shrnutím celé práce v závěru.

Kapitola 2

Obecná problematika testování softwaru

Pojem *testování* (anglicky *testing*) v kontextu softwarového inženýrství definuje norma ISO/IEC/IEEE 29119-1:2013(E) jako *„soubor činností prováděných za účelem usnadnění objevování a/nebo hodnocení vlastností jedné nebo více testovaných položek“* [41].

Na normu navazuje odborná literatura, kde se můžeme setkat s definicemi, které jsou více popisné a více zaměřené na praxi. Ron Patton ve své knize *Testování softwaru* vymezuje testování jako aktivity prováděné softwarovým testerem za účelem včasného vyhledání chyby a zajištění její nápravy. Co je projevem chyby (myšleno softwarové chyby) lze určit pomocí 4 podmínek. Pokud platí alespoň jedna podmínka, jedná se pravděpodobně o projev chyby v programu [61]:

- Software nedělá něco, co by podle specifikace produktu dělat měl.
- Software dělá něco, co by podle údajů specifikace produktu dělat neměl.
- Software dělá něco, o čem se produktová specifikace nezmiňuje.
- Software dělá něco, o čem se produktová specifikace nezmiňuje, ale měla by se zmiňovat.

Výčet podmínek pro definici projevu chyby můžeme ještě rozšířit o pátý bod, který se zaměřuje na testování UX (User Experience) [61]:

- Software je obtížně srozumitelný, těžko se s ním pracuje, je pomalý, nebo – podle názoru testera softwaru – jej koncový uživatel nebude považovat za správný.

Z prvních čtyřech podmínek lze vyčíst, že testování neslouží pouze k verifikaci toho, že software dělá to, co je zadáno ve specifikaci (co se od softwaru očekává), ale testování zároveň slouží i k ověření, že software nedělá něco, co ve specifikaci není nebo co by dokonce mohlo negativně ovlivnit celý systém, jiný program nebo obecně spokojenost uživatelů.

Pro demonstraci účelu testování využiji právě projekt JavaScript Restrictor, na který se tato práce zaměřuje a který je představen v sekci 6.1. Testování by mělo ověřit, že software zamezuje sledování koncového uživatele pomocí určitých nástrojů – tedy, že software dělá to, co se od něj očekává. Zároveň by však testování mělo ověřit i to, že JavaScript Restrictor neomezuje funkčnost stránky – tedy že software nedělá nic nežádoucího navíc, než co od něj požadujeme.

Určitá forma testování má své nezastupitelné místo snad v každém životním cyklu softwaru. Ať už je aplikován V model, W model, spirálový model nebo agilní přístup při vývoji, testování je vždy nezbytnou součástí a pokud je zanedbáno, může to vést k neočekávanému zvýšení nákladů v době údržby produkční verze programu nebo dokonce k zásadnímu ohrožení funkčnosti celého systému.

Pro pozdější jasnější vymezení pojmu „testování projektu JavaScript Restrictor“ je nyní uvedena kategorizace, ve které testování softwaru můžeme rozdělit:

dle kritéria automatičnosti na

- manuální a
- automatické,

dle kritéria dynamičnosti na

- statické a
- dynamické,

dle kritéria opakovatelnosti testu na

- jednorázové a
- regresní,

dle kritéria úrovně testování na [16]

- jednotkové,
- integrační,
- systémové testy a
- Alfa, Beta a akceptační testy,

dle kritéria znalosti vnitřní struktury a funkčnosti softwaru na

- testování černé skříňky,
- testování bílé skříňky
- testování šedé skříňky [60] – tento pojem používá zejména Alan Page ve své knize *Jak testuje software Microsoft*,

nebo dle typu požadavku k otestování na

- funkční a
- nefunkční. (Do této kategorie patří testování požadavků na rychlost, výkon, škálovatelnost, udržitelnost, spolehlivost, použitelnost, dostupnost, rozšiřitelnost, modifikovatelnost, spravovatelnost, přenositelnost, bezpečnost, testovatelnost a další.)

V této kapitole je nejdříve uvedena terminologie používaná v následujících kapitolách technické zprávy a následně je shrnuta motivace pro důsledné testování softwarového produktu. Za ní následuje podobnější rozebrání různých druhů testování představených v kategorizaci výše.

2.1 Terminologie

V celé práci jsou používány pojmy dle slovníku *Glossary of Testing Terms* [42] vydaného asociací ISTQB (International Software Testing Qualifications Board), která vznikla v roce 2002 a nyní je světovým lídrem v certifikaci softwarových testerů. ISTQB má v České a Slovenské republice regionálního zástupce – organizaci CaSTB (Czech and Slovak Testing Board), která zajistila překlad slovníku své mateřské organizace do češtiny [19]. Tento český slovník jsem v práci použil pro správné spárování pojmů v české a anglické odborné literatuře a k rozlišení jazykových odlišností v českých termínech. Dalším rozšiřujícím zdrojem termínů použitých v práci je slovník *IEEE Standard Glossary of Software Engineering Terminology* [40].

Následují nejdůležitější definice pojmů pro tuto práci doplněné o rozšiřující výklad z jiných publikací než výše uvedených slovníků.

Chyba

Softwarovou chybou se rozumí takové selhání člověka, které se promítne do programu. Chybu může udělat analytik, vývojář či jiný člověk při práci na svém artefaktu softwarového produktu (fragment kódu, kapitola analýzy, diagram) [16]. Pokud se jedná o chybu při programování (tj. chybu v kódu), označujeme ji také anglickým pojmem *bug* [44].

Původ pojmu *bug* je v oblasti IT spojen s označením defektu způsobeného skutečným hmyzem (*bug* lze do češtiny přeložit jako *moucha*, *štěnice* nebo obecně *brouk*). Jeden z nejznámějších případů je zápis o molu zachyceném na relé počítače Mark II dne 9. září 1947. Nejedná se však o první použití termínu v tomto kontextu, čemuž odpovídá i zápis (v překladu): „První skutečný případ nalezeného bugu“ [88].

Defekt

Defekt (někdy označován i českým pojmem *chyba*, který se však poté překrývá s předchozí definicí) je odchylka aktuálního způsobu fungování softwaru od očekávaného [16]. Defekt je projevem chyby. Jelikož se jedná o široký pojem, který označuje určitý problém v softwaru, lze ho přesněji vymezit pomocí pěti podmínek uvedených na začátku této kapitoly.

Selhání

Selhání (nebo také *porucha*) je neschopnost softwarového systému nebo komponenty plnit požadované funkce v rámci stanovených požadavků [17].

Testování

Testování označuje proces hledání defektů (případně přímo chyb) nejen v programu, ale i ve specifikaci a návrhu [61].

Do procesu testování se však již nezahrnuje opravení chyby. Opravení chyby je úkolem vývojáře, nikoliv testera [61]. Tester může být odpovědný za zajištění opravy chyby, neměl by však opravu sám provádět, aby poté mohl objektivně znovu otestovat, zda došlo k nápravě chyby.

Verifikace

Verifikace je potvrzení získané zkoumáním a doložením objektivních důkazů, že specifikované požadavky byly naplněny [16]. Verifikace znamená ptát se: „Vytváříme produkt správně?“.

Verifikace je tedy proces, jehož cílem je potvrdit, že software vyhovuje zadané specifikaci [61].

Validace

Validace je potvrzení získané zkoumáním a doložením objektivních důkazů, že skutečné požadavky pro určité zamýšlené použití či nasazení byly naplněny [16]. Validace znamená ptát se: „Vytváříme správný produkt?“.

Validace je tedy proces, jehož cílem je potvrdit, že software vyhovuje potřebám uživatele [61].

2.2 Motivace pro testování softwaru

Testování tvoří nezanedbatelnou část nákladů na vývoj softwaru (dle průzkumu britské společnosti Prolifics Testing připadá průměrně 1 tester na 3 programátory [38]). Nabízí se otázka, zda testování je opravdu nutnou součástí vývojového procesu? Zda je vůbec potřeba testovat? Nestačilo by zaměstnat zkušenější a zodpovědnější programátory?

Historie ukazuje, že testování je zatím nenahraditelnou součástí vývoje softwaru, protože základní vlastností sebelepšího programátora je, že dělá chyby. A i kdybychom si představili, že budeme mít dokonalé programátory nebo budeme kód automaticky generovat na základě návrhových diagramů, stále se nezbavíme lidské chyby třeba už při specifikaci nebo návrhu. Dokonce větší množství chyb vzniká již při specifikaci a návrhu softwaru než při samotném programování [61]. Z toho důvodu do vývojového procesu vstupuje tester, jehož základní lidskou vlastností by měla být zvědavost a který se snaží najít ve specifikaci, návrhu i programu chyby.

Ani testování však nedokáže zajistit, že výsledný produkt bude bezchybný. Základním paradigmatem je, že každý software obsahuje chybu nehledě na množství testů [61]. Cílem testera je zkusit zajistit, aby software neobsahoval chyb mnoho a hlavně, aby neobsahoval chyby kritické, které by vedly k ohrožení daleko cennějších věcí, jakými jsou třeba firemní data (v případě bezpečnostní chyby) nebo dokonce lidské životy.

Nabízí se otázka, jaká je hodnota testování? Je možné ji zkusit vyjádřit prostřednictvím chyb nalezených v průběhu testování. Nejdříve by bylo nutné ocenit nalezené chyby. Cena chyby je hodnota chybou způsobené ztráty vynásobená počtem výskytů dané chyby během celého života softwaru. [16] Pokud bychom tedy uvažovali software, který poběží 10 let, v němž je chyba způsobující ztrátu 10 000,- Kč a tato chyba se může projevit jednou ročně, dostáváme cenu chyby rovnou 100 000,- Kč. Hodnota chybou způsobené ztráty může být vyčíslena jako hodnota ztracených dat, hodnota přidělané práce nebo třeba jako hodnota dobrého jména, pokud by došlo k jeho poškození vlivem chyby. Stačí takto ocenit všechny chyby, sečíst získané ceny a dostaneme celkovou hodnotu testování.

2.3 Manuální a automatické testování

Automatické testování se od manuálního liší tím, že jej vykonává nějaký program místo člověka. Takový program simuluje předem definované kroky a typicky to dokáže daleko rychleji, než by tester zvládl manuálně. Tím se dostáváme k výhodám automatického testování, ke kterým patří:

- rychlost,
- možnost jednoduše opakovat test,
- každé opakování je stejné,
- vykonání testu je zdarma (pokud zanedbáme náklady na elektřinu apod.).

Poslední bod zmiňuje cenu za test. Samotné vykonání testu je z hlediska personálních nákladů opravdu zdarma, protože pro vykonání testu již nepotřebujeme žádného pracovníka. Přesto je třeba si uvědomit, že i v tomto případě lze mluvit o nákladech na provedení testu, protože daný test musel tester manuálně vytvořit, a to typicky zabere daleko více času, než kdyby test jednou sám manuálně provedl.

Vyplatí se dělat automatické testy? Jsou ekonomicky výhodné? Na otázky neexistuje univerzální odpověď, ač v mnoha případech odpovíme bez dlouhého přemýšlení, že ano. Automatické testování má rozhodně smysl pro softwarový produkt, u kterého je plánovaný dlouhodobý provoz a u kterého se čeká nižší počet změn. U takového produktu dojde k rychlému návratu nákladů investovaných do automatických testů [16].

Pokud je systém dlouhodobě upravován a výrazně měněn, je nutné si uvědomit, že bude třeba upravovat i samotné testy. Může se tak stát, že návratnost investice do automatických testů se vrátí za mnohem delší dobu.

Poslední nevýhodou automatického testování, kterou zmíním, je omezená možnost analýzy chybného chování testovaného systému. Zatímco tester při manuálním testování dokáže vnímat systém z pohledu uživatele a všimnout si tak například přetékajícího okna nebo špatně zarovnaného obrázku, možnosti automatického testu jsou v tomto směru omezené a automatický test kontroluje pouze to, co je explicitně v testu napsáno. A s tím souvisí i to, že automatické testování je schopno si jen v omezené míře dávat jednotlivé informace do širších souvislostí.

Určitě by šly najít další důvody, proč manuální testování může být upřednostněno před tím automatickým, avšak možnost opakovaně a zdarma spouštět automatické testy předčí u většiny softwarových projektů všechny protiargumenty, a i proto vývojové společnosti upřednostňují u většiny větších projektů tvorbu automatických testů namísto manuálního testování [16]. I v případě rozšíření JavaScript Restrictor dává jednoznačně smysl zvolit automatické testy, které bude možné v budoucnu opakovat po každé úpravě nástroje a také v pravidelných intervalech z důvodu proměnlivosti webových stránek.

2.4 Statické a dynamické testování

Statické testování probíhá na softwarovém produktu, aniž by byl výsledný software spuštěn [16]. Naopak dynamické testování znamená testování softwaru za běhu [60].

Typickým příkladem statického testování je analýza kódu nebo testování jednotlivých dokumentů vzniklých v projektu. Statické testování je možné v určitých případech nazvat

také pojmem „kontrola“ [61]. Testování dokumentů je zejména důležité z pohledu manažerů – při tomto typu testování se ověřuje úplnost, správnost, relevantnost, jednoznačnost, konzistence a formální úprava dokumentů. Takové testování může probíhat například formou odškrtávání položek kontrolního seznamu. Statické testování kódu může mít formu revize kódu nebo párového programování, kdy jeden vývojář píše zdrojový kód a druhý ho sleduje a kontroluje (tzn. staticky testuje správnost vznikajícího kódu). Při statickém testování kódu se nejen ověřuje, zda kód zjevně bude fungovat dle specifikace, ale kontroluje se také volba vhodných datových typů, návrhových vzorů a dalších aspektů, které mají zajistit dobrou budoucí udržovatelnost a rozšiřitelnost aplikace. Mohlo by se zdát, že statické testování nelze automatizovat a onu kontrolu vždy musí provést člověk. Avšak i bez využití pokročilých nástrojů umělé inteligence, která do jisté míry dokáže nahradit inteligentní chování člověka při statickém testování, lze jednoduše automaticky a staticky testovat kód na základě měření vybraných metrik. Takovými metrikami mohou být [16]:

- počet řádků kódu na třídu/metodu,
- cyklomatická složitost,
- provázanost tříd,
- pokrytí kódu testy.

Dalším příkladem statického testování kódu může být také automatická analýza kódu vývojovým prostředím (IDE) [60], které případně upozorňuje programátora, pokud objeví chyby v datových typech, volání neexistující metody třídy a podobně.

Dynamické testování je to, co bývá běžně chápáno pod pojmem testování [77]. Jedná se o provádění testů nad běžící programem. Pro každý test nebo skupinu testů je testovaný program spuštěn se vstupními testovacími daty. Následně je proveden testovací scénář, jehož výstupem je informace o uspění nebo selhání testu a případné doplňující informace. Výběr vstupních dat pro testování je velice důležitý. Často je vhodné volit mezní hodnoty pro ověření správně nastavených hranic a podmínek v programu. Při dynamickém testování rozlišujeme testování černé, bílé a případně šedé skříňky. Tyto přístupy jsou dále rozvedeny v samostatné sekci 2.10.

2.5 Regresní testování

Regresní testování označuje testování dříve otestovaného programu po jeho modifikaci s cílem ověřit, že jako důsledek změn nebyly do nemodifikovaných částí softwaru zaneseny nové defekty nebo nebyly odkryty již existující [42]. Regresní testování je prováděno při změně softwaru nebo prostředí, v němž je spuštěn. Takovou změnou softwaru může být i oprava chyby, přičemž v takovou chvíli slouží regresní testy k ověření, že oprava chyby nezpůsobila nebo neodhalila jiný defekt. Regresní testování je úzce spojeno s automatickým testováním, protože automatické testování je levnější než manuální při častém opakování stejných testů [17].

Z hlediska terminologie je důležité rozlišit pojmy „konfirmační testování“ a „regresní testování“. Zatímco konfirmační testování označuje opětovné spuštění pouze takových testovacích případů, které v předchozím spuštění skončily chybou, regresní testování znamená spouštění takových testovacích případů, které předtím skončily úspěšně, a případně i spuštění dříve neúspěšných testovacích případů, pokud by mohly nalézt i jinou chybu, než kterou

minule odhalily [16]. Konfirmační testování tak dokáže maximálně potvrdit pouze to, že minule nalezené chyby jsou opravené. Regresní testování se snaží potvrdit to, že při opravě chyby nebyla do softwaru zanesena nebo se neprojevila jiná chyba (klidně i v úplně jiné části systému). Konfirmační testování se také nazývá *přetestování* [77]. V praxi se provádí konfirmační a regresní testování souběžně. Zároveň se tak spustí celá sada testů, ve které jsou jak testy, které neuspěly (konfirmační testy), tak testy, které uspěly (regresní testy). Po proběhnutí se zjistí výsledek – zda minule neúspěšné testy již nyní projdou a zda minulé úspěšné testy naopak neodhalí novou nebo předtím skrytou chybu.

Regresní testování se neváže k žádné konkrétní úrovni testování (jednotkové, integrační nebo systémové testy). Naopak je vhodné regresní testy se správně zvolenou intenzitou opakovaně spouštět na všech třech úrovních.

2.6 Jednotkové testy

Jednotka (v jednotkovém testování) je nejmenší možná testovatelná softwarová komponenta [17]. Jednotkové testování (anglicky *unit testing*) je tedy nejnižší úroveň testování, při kterém se ověřuje správná funkčnost jednotlivých metod a funkcí. Jednotkové testy spadají do kategorie testů bílé skříňky (tester musí znát jednotlivé metody zdrojového kódu).

Princip jednotkového testování je založen na vytvoření dalšího programu, který bude automaticky testovat jednotlivé metody vyvíjeného software. Ač by to bylo možné, není potřeba takový testovací program vytvořit od nuly, ale lze využít hojně rozšířené rámce, které práci velmi urychlí (např. JUnit nebo TestNG pro Javu, nUnit nebo MbUnit pro .NET – všechny zmíněné rámce vychází z architektury xUnit [36], jejímž cílem je tuto oblast standardizovat). Výstupem takového testovacího programu je zpravidla přehled testů s informacemi o úspěšném nebo neúspěšném proběhnutí.

Jednotkové testy jsou v naprosté většině psány ve stejném programovacím jazyku jako testovaný program. Soubor jednotkového testu se skládá ze tří hlavních částí: *set-up* metody, testovací metody a *tear-down* metody [16]. *Set-up* metody se volají před testem a slouží k inicializaci testovacího prostředí (např. načtení testovacích dat). *Tear-down* metody slouží k „úklidu“ prostředí po testu (např. smazání nově vzniklých záznamů v databázi). Testovací metody vykonávají samotné testy nad programem a v jednom souboru jich může být více. Testovací metoda má typickou strukturu zvanou *arrange-act-assert* [16]:

- *arrange*: vytvoření testovaného objektu, jeho počáteční nastavení, definování očekávaného výsledku testu,
- *act*: volání testované metody nad testovacími daty,
- *assert*: kontrola, zda skutečný výsledek metody odpovídá očekávanému výsledku.

V jednotkovém testování se používají metriky pro vyjádření, jaká část zdrojového kódu je pokryta testy. Typickou metrikou tohoto typu je *Pokrytí řádků* (poměr počtu řádků kódu vykonaných testem k celkovému počtu řádků kódu) nebo pokročilejší metrika *Pokrytí cest* (poměr počtu větví v řídící struktuře programu vykonaných testem k celkovému počtu těchto větví).

V agilním vývoji existují přístupy, u nichž je obráceno pořadí psaní zdrojového kódu programu a tvorby jeho testů. Při takových přístupech jsou nejdříve vytvořeny testy a teprve poté je psán kód aplikace, ale jenom do té míry, aby testy uspěly. Nejstarší z těchto přístupů je Vývoj řízený testy (*Test-driven development*, zkráceně TDD), následně na jeho

základu vznikly další přístupy: Vývoj řízený akceptačními testy (*Acceptance test–driven development*, zkráceně ATDD) a Vývoj řízený chováním (*Behaviour-driven development*, zkráceně BDD).

2.7 Integrační testy

Každý systém se skládá z částí. Nemusí to být přímo moduly, ale stačí složení z jednotek popsaných v předchozí kapitole. Cílem integračního testování je nalezení chyb na rozhraní jednotek, modulů i celých subsystémů. Integrační testování následuje po jednotkovém testování a mělo by být provedeno až ve chvíli, kdy všechny jednotkové testy uspějí [17]. Zatímco jednotkové testy hledají chyby jen uvnitř jednotek, integrační testy předpokládají otestované jednotky a hledají chyby ve spolupráci a komunikaci mezi jednotkami, případně většími částmi.

Integrační testy mohou být vytvářeny na 2 základních úrovních [16]:

- lokální rozhraní,
- vzdálené rozhraní.

Lokálním rozhraním je myšleno rozhraní mezi jednotkami nebo mezi objekty obsahující jednotky, případně mezi celými moduly programu. Lokální rozhraní je implementováno typicky na úrovni programovacího jazyka, zřídka kdy se používá specializovaný protokol. Testování je možné realizovat často stejným rámcem, který byl používán pro tvorbu jednotkových testů.

Vzdálené rozhraní je rozhraní, které systém vystavuje svému okolí. Tímto okolím může být jenom několik málo ostatních systémů v rámci rozsáhlého podnikového IT řešení skládajícího se z několika spolupracujících systémů nebo i tisíce koncových uživatelů, kteří vzdálené rozhraní využívají jako službu. Takové rozhraní bývá v dnešní době typicky založeno na architektuře REST. Na tento typ integračních testů lze úspěšně využít specializované nástroje (SoapUI, Katalon Studio, Postman apod. [21, 2]) a je vhodné je upřednostnit před rámcem, ve kterém jsou tvořeny jednotkové testy a integrační testy na úrovni lokálního rozhraní.

2.8 Systémové testy

Systémové testy přichází na řadu ve chvíli, kdy je sestaven kompletní systém (respektive jeho nejnovější verze při inkrementálním vývoji). Zatímco předchozí úrovně testování – jednotkové a integrační testy – byly založeny na testování bílé skříňky, takže tester se na systém díval z pohledu vývojáře, systémové testy odráží spíše pohled koncového uživatele, který systém vnímá jako černou skříňku [44]. Cílem systémových testů je ověření komplexní funkčnosti systému, do které spadají nejen scénáře využití systému, ale i odolnost systému vůči vysokému vytížení, škálovatelnost, udržovatelnost, přívětivost nebo bezpečnost. Obecně lze systémové testování rozdělit na funkční a nefunkční podle toho, jaké požadavky ověřuje.

Cílem systémových testů je zejména detekovat problémy související s rozmanitými hardwarovými zařízeními, na nichž program může běžet a odhalit chyby v rozhraní programu, přičemž hlavním rozhraním bývá grafické uživatelské rozhraní [17]. Ač to může být obtížnější oproti jednotkovému nebo integračnímu testování, systémové testy se také dají automatizovat pomocí vybraných nástrojů.

Testování funkčních požadavků

Funkční požadavky definují funkcionalitu systému [16]. Tedy množinu funkcí, které bude moci používat uživatel systému. Funkční požadavky určují, *co* má systém umět.

Jeden funkční systémový test typicky provádí ucelený elementární scénář použití systému. Může se jednat například o scénář vytvoření nové faktury. Systémové funkční testy se zaměřují na to, zda pro zadané vstupy vrací systém správné výstupy.

Automatizace provedení takového scénáře je jednodušší pro webové aplikace než pro klasické aplikace, a to z důvodu jednotného běhového prostředí – webového prohlížeče, pro který existují automatizační nástroje. Nejvýraznějším nástrojem pro automatizaci webového prohlížeče je Selenium, který je představený v následující kapitole v sekci 3.1.

Testování nefunkčních požadavků

Nefunkční požadavky definují kvalitativní kritéria a omezení, jež se vztahují k celému systému, k jeho dílčím částem anebo jen ke skupině jiných požadavků [16]. Nefunkční požadavky určují, *jak* má být systém navržen, implementován a nasazen, aby bylo dosaženo zadané míry kvality systému. V případě některých systémových nefunkčních testů neplatí striktně princip testování černé skříňky, protože pro vybrané účely testování potřebujeme znát i vnitřní strukturu a fungování (např. bezpečnostní audit v rámci bezpečnostních testů).

Nefunkční vlastnosti (rychlost, výkon, škálovatelnost, udržitelnost, spolehlivost, použitelnost, dostupnost, rozšiřitelnost, modifikovatelnost, spravovatelnost, přenositelnost, bezpečnost, testovatelnost a další) zpravidla není možné přímo měřit. Místo toho jsou hodnoty odhadovány pomocí nepřímých veličin: spolehlivost se například odvozuje od chybovosti, testovatelnost se měří pomocí cyklomatické složitosti [60].

Významnou kategorii takových testů tvoří *zátěžové testy*. Výsledná aplikace může bezvadně fungovat, když při testování do aplikace přistupuje pouze jediný uživatel, ale co když se přístupu k objektu bude v jeden okamžik dožadovat několik tisíc uživatelů? Při zátěžových testech jsou vytvářeny takové podmínky, aby testovaný program byl výrazně vytížen. Zátěžové testy se snaží najít takové chyby, které se primárně vyskytnout ve chvíli největšího vytížení systému.

Na podobném principu fungují *stresové testy*. Jediným rozdílem je, že běh programu není ztížen vysokým množstvím požadavků, ale ubráním dostupných zdrojů programu – například paměti a diskového prostoru. Program při stresovém testu může do jisté míry omezit svoji funkčnost (zejména se očekává zpomalení), ale neměl by „spadnout“ nebo provádět operace chybně [61].

Bezpečnostní testy ověřují tři pilíře počítačové bezpečnosti: dostupnost, důvěrnost a integritu. Při těchto testech se zjišťuje odolnost systému proti neoprávněnému přístupu do systému nebo k datům, ochrana proti odepření služeb a podobné [77]. Testy mohou být prováděny etickým hackerem (anglicky *white hat*), který se staví do role útočníka a snaží se prolomit zabezpečení nově vytvořeného systému. Do bezpečnostních testů nespadá pouze snaha prolomit zabezpečení systému, ale také bezpečnostní audit, který může kupříkladu obsahovat: kontrolu požadavků na hesla uživatelů (délka, kombinace malých a velkých písmen a čísel, pravidelná změna hesla atd.), vynucení dvou-faktorové autentizace, šifrování úložiště dat a komunikace mezi systémy a kontrola práv jednotlivých skupin uživatelů.

Cílem *testů obnovení* je předem vyzkoušet, jak by vypadalo obnovení systému ze záloh při jeho napadení škodlivým softwarem (anglicky *malware*) nebo obecně při fatálním selhání testovaného systému. Pokud testy obnovení skončí neúspěšně, znamená to, že systém

by v případě vážného selhání nebylo možné ze záloh obnovit. Pokud jsou testy obnovení zanedbány nebo úplně vynechány, zjistí se často až při skutečném napadení systému, že obnovení není možné. Konkrétním příkladem může být úspěšný útok vyděračského softwaru (anglicky *ransomware*), po kterém je potřeba obnovit data ze záloh.

Testy konfigurace ověřují plnohodnotnou funkčnost testovaného systému na různých verzích operačního systému, v různých jazycích, na různých hardwarových platformách a podobně [77]. Na prvním místě bývá ověření, že výsledný systém bude schopný pracovat s různými kombinacemi hardwaru. V dnešní době webových aplikací je třeba velkou výzvou ohledně testování konfigurace široká škála zařízení – od mobilních telefonů až po velké monitory. Pokud vytváříme webovou aplikace, často požadujeme, aby byla přívětivá pro všechna zařízení. Další výzvou pro webové aplikace je vytvoření přívětivého rozhraní nejen pro zařízení ovládaná myší, ale i pro zařízení s dotykovými displeji. To si vyžaduje rozsáhlé testování konfigurace pro různé displeje.

Jako poslední oblast nefunkčního testování jsou více rozvedeny *testy použitelnosti*, které se zaměřují na to, jak snadno dokáže uživatel pochopit uživatelské (grafické) rozhraní programu a jak snadno a přirozeně s ním dokáže pracovat [60]. Testování použitelnosti spočívá v ověření, zda lze všechny funkce aplikace snadno najít a zda se chovají tak, jak uživatel předpokládá. Zároveň se také ověřuje, že zpětná vazba uživateli od systému je vždy jasná, dobře čitelná a ideálně okamžitá. Na použitelnost mají vliv i ostatní kategorie nefunkčního testování. Testování použitelnosti zpravidla neprovádí testeři. Doporučovaných postupem je přizvání externí člověka, který bude mít za úkol v aplikaci provést určitý úkol a na základě záznamu pohybu myši, stisku kláves, výrazů v obličeji a podobně, je následně vyhodnocena celková uživatelské přívětivost. Jako takový externí spolupracovník může být vybrán potenciální budoucí uživatel programu, netechnický pracovník firmy (např. účetní, sekretářka) nebo je možné najmout naprosto nezávislého externího člověka.

I některé systémové nefunkční testy lze automatizovat. Takové nástroje se pak často zaměřují nejen na jednorázové testy, ale i na dlouhodobý monitoring provozu systému – příkladem může být nástroj *Digital experience monitoring* od společnosti Dynatrace, který se zaměřuje na analýzu celkové uživatelské přívětivosti systému [23]. Pro některé nefunkční systémové testy lze úspěšně využít i zmíněnou sadu nástrojů Selenium (např. testování výkonu aplikace prostřednictvím měření procesorového času nutného pro vykonání vybrané akce) [69].

2.9 Alfa, beta a akceptační testy

Beta testy a akceptační testy jsou z technického hlediska speciální kategorií systémových testů, které ale probíhají na zařízeních zákazníka a provádí je sám zákazník, nikoliv tester [17]. Alfa testy probíhají ještě před beta testy a také se jedná o speciální kategorii systémových testů. Od Beta testů se liší tím, že je ještě neprovádí koncoví uživatelé, ale přímo zaměstnanci vývojové společnosti nebo malá skupina externích spolupracovníků.

Akceptační testy se používají pro ověření splnění specifikace v případě vývoje softwaru na zakázku. Po dokončení vývoje a všech testů je výsledný systém nasazen u zákazníka a zákazník s možnou účastí projektového manažera provede předem domluvené akceptační testy. Akceptační testy jsou významné zejména z obchodního hlediska, protože slouží pro vymezení podmínek, za jakých bude výsledný systém akceptován zákazníkem. Právě až po úspěšném proběhnutí akceptačních testů dochází typicky k platbě za vyvinutý systém.

Akceptační testy probíhají zejména při agilním vývoji na speciálně vyhrazeném prostředí, které by ale mělo odpovídat svými vlastnostmi produkčnímu prostředí. Ve výsledku

tak potřebujeme 3 prostředí: vývojové (DEV), akceptační (UAT) a produkční. DEV prostředí slouží pro vývoj, UAT prostředí (z anglické zkratky *User Acceptance Testing*) slouží jako testovací prostředí. Díky existenci UAT prostředí je možné na produkční prostředí přenášet jen řádně otestované změny a zároveň máme jistotu, že při testování nedojde ke změně produkčních dat.

Pojem alfa a beta testování se vyskytuje zejména u systémů, u kterých neprobíhá vývoj na zakázku, ale u kterých se jedná o generický software, který má mnoho různých zákazníků. V takovém případě se nepoužívají mechanismy akceptačního testování, protože ověřovaná specifikace není dána zákazníky, ale společností, která systém vyvíjí. Aplikuje se alfa a beta testování. Alfa testování probíhá interně ve vývojové společnosti typicky tak, že sami zaměstnanci výsledný program používají. Následně se program uvolní beta testerům, což jsou většinou dobrovolníci, kteří stojí o technologické novinky i za cenu toho, že nejnovější verze softwaru může obsahovat větší množství chyb. Po alfa a beta testování a případném odstranění nalezených chyb je nový software nabídnut zákazníkům buď ve formě nové aktualizace nebo nové verze k zakoupení.

2.10 Testování černé a bílé skříňky

Testování černé, bílé a případně šedé skříňky rozlišuje, jak moc známe vnitřní strukturu a vnitřní fungování testovaného softwaru při vytváření testů. Testování černé skříňky označuje případ, kdy nemáme o vnitřní struktuře a vnitřním fungování žádné informace. Celý softwarový produkt se nám tak jeví jako neprůhledná černá skříňka. Testování černé skříňky reprezentuje pohled koncového uživatele na software, který neví nic o vnitřním fungování a jsou pro něho důležité jenom korektní výstupy na zadané vstupy.

Opakem je testování bílé skříňky, které staví na kompletní znalosti vnitřní struktury a fungování programu. V případě bílé skříňky mohou být testovací scénáře velice podrobné a specificky zaměřené, protože známe přesnou část kódu, který chceme daným testem ověřit.

V případě testování černé skříňky jsou testy vytvářeny pouze na základě popisu softwaru (specifikace, požadavky apod.). V případě testování bílé skříňky jsou testy vytvářeny na základě znalosti vnitřní struktury a fungování (typicky na základě znalosti zdrojového kódu [61]).

U společnosti Microsoft se lze setkat i s pojmem testování šedé skříňky [60], který představuje kombinaci testování černé a bílé skříňky. Pojem testování šedé skříňky označuje testování softwaru, o němž máme informace o vnitřní struktuře a fungování, ale testy jsou navrhovány podle uživatelských scénářů, a ne podle jednotlivých částí kódu. Je to prostřední cesta mezi testováním černé a bílé skříňky. U testování šedé skříňky by se mělo podařit vytvořit testy odpovídající uživatelským scénářům s dostatečně podrobnou kontrolou programu na základě znalosti vnitřní struktury a fungování.

Kapitola 3

Nástroje pro testování rozšíření do webových prohlížečů

V této kapitole jsou popsány nástroje, které lze využít pro automatické testování rozšíření do webových prohlížečů. Zmíněné nástroje však nejsou limitovány pouze na tento případ testování a lze je úspěšně použít i pro obecnější účely. Slovem testování bude nejen v této kapitole myšleno automatické testování. Na manuální testování se tato práce nezaměřuje.

Testování rozšíření do webových prohlížečů se zdá být obtížnější než testování klasické počítačové aplikace. Při testování rozšíření do webových prohlížečů se musíme vypořádat s několika specifickými problémy:

- Rozšíření typicky bývá určeno pro více než jen jeden prohlížeč. A ve všech podporovaných prohlížečích musí fungovat ideálně stejně nebo alespoň obdobně.
- Webový prohlížeč je neustále vyvíjen, to znamená, že pořád přicházejí jeho nové verze a rozšíření s aktualizacemi prohlížeče musí držet krok. Při testování nám ale nestačí otestovat funkčnost rozšíření jen oproti nejnovější verzi vybraného prohlížeče, ale většinou potřebujeme testovat funkčnost rozšíření v několika posledních verzích prohlížeče.
- Rozšíření pracuje se stránkami – mění je, vylepšuje, interaguje s nimi. V době vzniku této diplomové práce existuje přibližně 1,5 miliardy veřejných webů [48, 49]. Celkový počet indexovaných webových stránek dostupných přes vyhledávače (vyhledávání Google, vyhledávač Bing apod.) je vyčíslen na přibližně 6 miliard [51], a to dle metodiky, kterou v roce 2016 představil nizozemský vědec Maurice de Kunder se svými kolegy [13]. Rozšíření prohlížečů by měla korektně fungovat ideálně na každé stránce. S tím souvisí potřeba testování na mnoha různých webových stránkách.
- Weby jsou tvořeny typicky z více než jediné stránky (jednoho souboru ve formátu `html`, `php`, `aspx` apod.). Když v automatickém testu otevřeme hlavní stránku webu, na kterou se lze dostat pomocí doménového jména druhého řádu, je třeba počítat s nutností projití i podstránek, pokud chceme web kompletně otestovat. Nalezení všech podstránek však klade před testera výzvu jejich nalezení, což není triviální záležitost. Odkazy na podstránky lze nalézt například pomocí získání všech jedinečných atributů `href` u elementů typu `<a>` z DOM. Úkol průchodu všech podstránek je i tak náročný z důvodu, že ne všechny podstránky musí být odkazované pomocí elementu `<a href="odkaz">`.

- Webové aplikace obsahují často významnou část své funkcionality dostupnou až po vytvoření účtu a přihlášení. Pokud bychom chtěli automatizovaně testovat kompletně webové aplikace, které se vyznačují právě nutností vlastního účtu a přihlášení, bylo by velmi problematické až téměř nemožné vytvářet automatizovaně účty pro potřeby testování. A kdybychom účty vytvořili manuálně, bude počáteční množství práce neúměrně vysoké, nehledě na nutnost řešit bezpečné uchovávání přístupových údajů pro běh automatických testů. Obdobným problémem jsou privátní části webových stránek třeba jenom pro správce webových aplikací (tzv. administrátorské sekce), do kterých přístup pro testování pravděpodobně nezískáme ani registrací, pokud se tedy nebude jednat o webovou aplikaci, kterou sami vyvíjíme.

3.1 Selenium

Selenium je sada softwarových nástrojů, publikovaných ve formě otevřeného kódu (známější je anglické označení *opensource*) pod licencí *Apache License 2.0*, nabízející automatizaci aplikací běžících ve webovém prohlížeči [46]. Základ celého nástroje vytvořil v roce 2004 Jason Huggins, ke kterému se postupem času přidávali další vývojáři, až vznikla celá komunita vyvíjející nástroj Selenium [14].

Selenium nabízí automatizaci webových prohlížečů, která může být využita i pro jiné účely než testování [12]. V praxi je však Selenium využíváno zejména pro automatické funkční dynamické testování uživatelského rozhraní [34], ale dokáže nabídnout také například nástroje pro výkonnostní testy, které spadají do kategorie testování nefunkčních požadavků.

Selenium se dělí na 4 projekty: *Selenium IDE*, *Selenium Remote Control*, *Selenium WebDriver* a *Selenium Grid* [46]. Každý projekt nabízí jiné možnosti pro automatizaci webových prohlížečů. Na zmíněné čtyři projekty z nástroje Selenium lze nahlížet také jako na čtyři komponenty, které se vzájemné mohou doplňovat a spolupracovat mezi sebou.

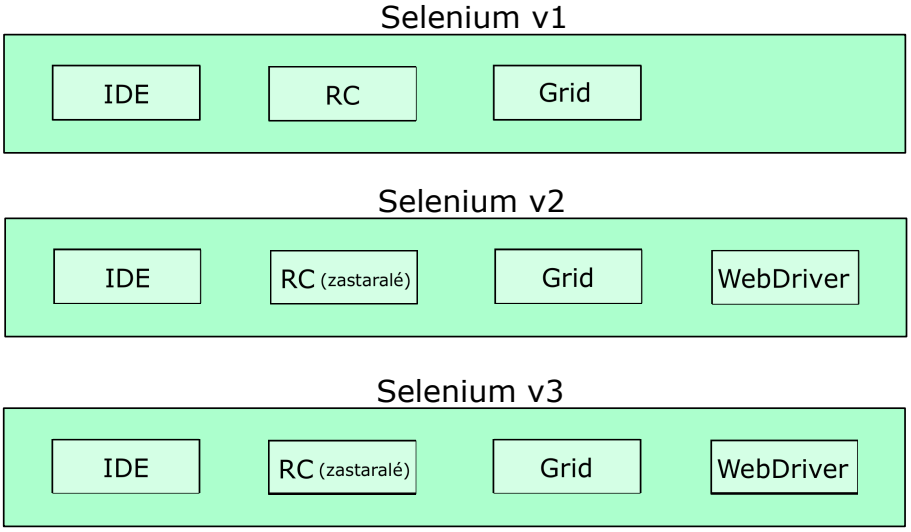
Selenium se postupně rozrůstalo a bylo neustále vylepšováno. Od druhé verze přibyl do rodiny Selenium nástroj Selenium WebDriver, a naopak zastaralým a dále nerozvíjeným se stal Selenium RC (název pochází z anglického sousloví *„Remote Control“*) [1]. Obrázek 3.1 zobrazuje přehled verzí a jednotlivých nástrojů v nich obsažených.

Následuje postupné představení jednotlivých nástrojů (komponent) nástroje Selenium.

Selenium IDE

Selenium IDE (z anglického názvu *Integrated Development Environment*) označuje rozšíření do prohlížeče, které bylo poprvé představeno v roce 2006 a nejdříve bylo dostupné pouze pro prohlížeč Mozilla Firefox. V roce 2018 došlo k modernizaci rozšíření, aby splňovalo formát *Web Extension*, a tím byl nástroj Selenium IDE dostupný nově i pro Google Chrome [20, 35, 71].

Selenium IDE představuje nejjednodušší způsob, jak se Seleniem vytvořit automatické testy. Jejich záznam probíhá formou nahrávání akcí uživatele. Samotná instalace nástroje Selenium IDE je také velice jednoduchá. Stačí nainstalovat dané rozšíření do prohlížeče Google Chrome přes Internetový obchod Chrome nebo do prohlížeče Mozilla Firefox přes Firefox Add-ons. Grafické uživatelské rozhraní nástroje Selenium IDE včetně ukázkového testu je zachyceno na obrázku 3.2. Ukázkový test provede automatické otevření webové stránky `https://www.seznam.cz` a prostřednictvím vyhledávače Seznam nalezne výsledky hledání na dotaz „FIT VUT“.



Obrázek 3.1: Přehled verzí sady nástrojů Selenium.

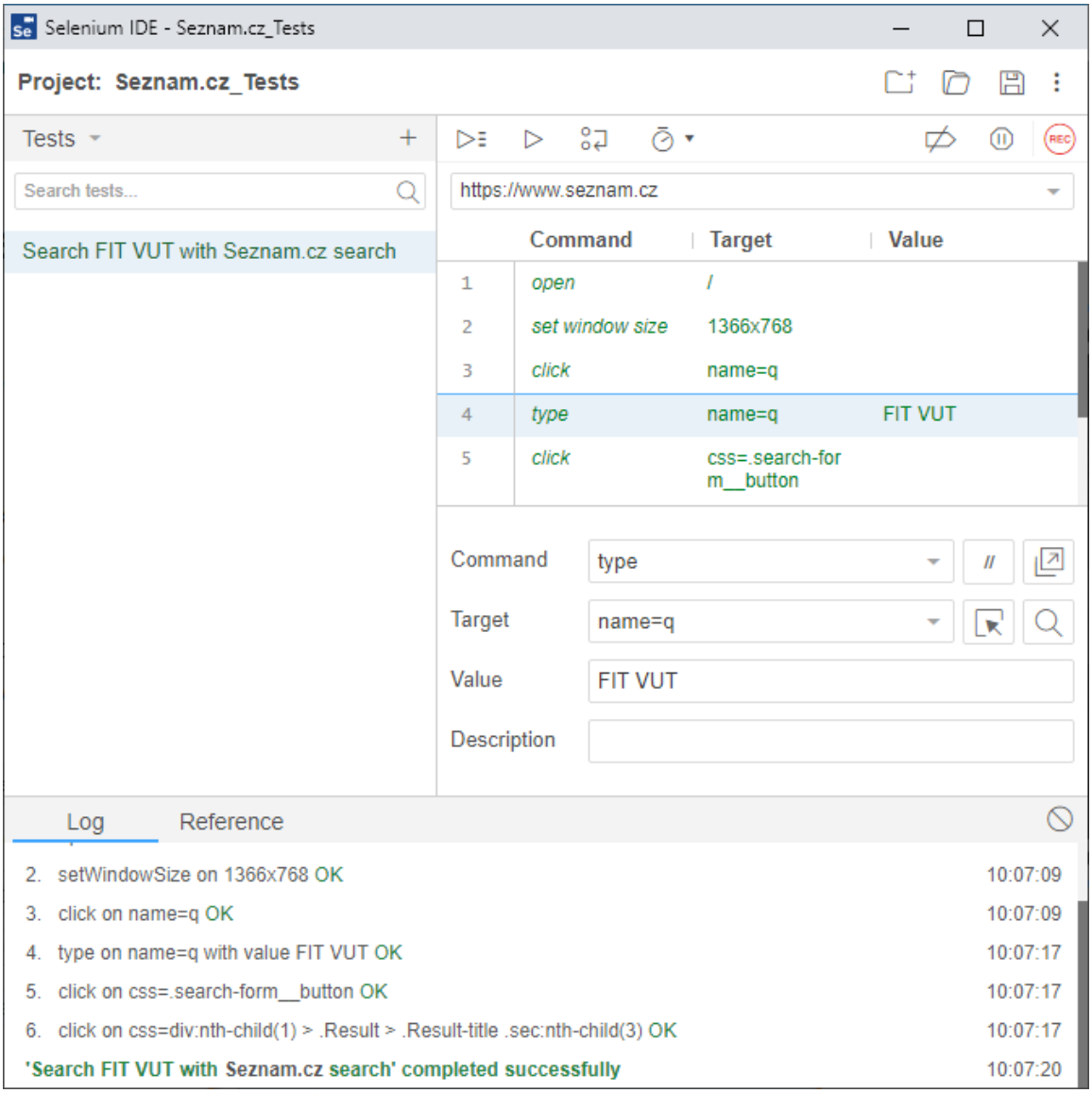
Pro reprezentaci nahraných akcí je využíván jazyk *Selenese*, ve kterém se každá akce (pokyn od uživatele) reprezentuje pomocí 3 informací [15]:

1. Příkaz.
2. Cíl příkazu (element cílený příkazem).
3. Argument (volitelná hodnota, která může například upřesňovat příkaz nad elementem).

Selenese je jazyk vytvořený přímo pro Selenium a slouží k reprezentaci příkazů. Uživatel nemusí Selenese rozumět. Každá akce, kterou nahraje v uživatelském rozhraní rozšíření Selenium IDE se do jazyku Selenese převede automaticky. Pokud však jazyku Selenese uživatel rozumí, může si příkazy přímo psát anebo upravovat. Rozšíření přímou editaci příkazů umožňuje.

Testy se sdružují do projektu, který lze uložit do souboru s koncovkou *.side*. Po vytvoření testů pomocí nahrávání lze testy opakovaně spouštět. Je možné spustit postupně všechny testy z projektu nebo pouze konkrétní vybraný test. V konzoli *Log* je možné zjistit výsledky testů.

Na první pohled se zdá, že Selenium IDE nabízí vše potřebné k plnohodnotné tvorbě testů. Existuje zde dokonce i podpora pro práci s proměnnými. Pomocí příkazu `store` je možné uložit hodnotu do proměnné a později ji použít. Také příkazy pro selekci i iteraci v testu jsou k dispozici. Nicméně využití těchto pokročilých konstrukcí není uživatelsky přívětivé a složitější testy by se staly brzy nepřehlednými [70]. Nehledě na to, že složitější příkazy (uložení do proměnné, selekce, iterace) již nenahrajeme, ale je potřeba znalost jazyku Selenese a zadání těchto příkazů manuálně. Selenium IDE je primárně nástroj, který umožňuje komukoliv bez znalosti programování nahrát a spustit jednoduché sekvenční testy. Pokud potřebujeme složitější konstrukce testů, je vhodné zvolit některý nástroj z rodiny Selenium představený v následujících sekcích.



Obrázek 3.2: Uživatelské rozhraní rozšíření Selenium IDE včetně ukázky příkazu v jazyku Selenese.

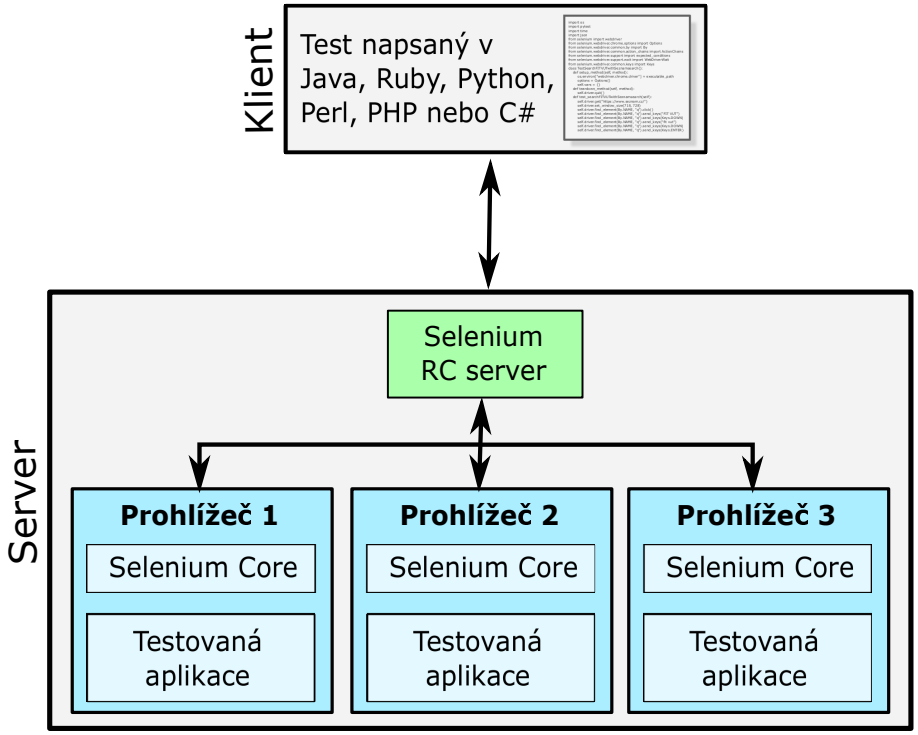
Selenium Remote Control

Selenium RC (z anglického názvu *Remote Control*) je nástroj typu klient-server. Skládá se ze dvou částí:

- Selenium Remote Control Server.
- Klientské knihovny, které poskytují rozhraní pro ovládání Selenium RC serveru.

Test běžící na klientovi posílá HTTP GET/POST požadavky na Selenium RC Server. Test na klientovi může být napsán v libovolném jazyku, který umožňuje zasílání HTTP požadavků (mezi podporované patří: Java, Ruby, Python, Perl, PHP nebo C#) [46]. Selenium RC Server, který může, ale nemusí, běžet na jiném fyzickém zařízení, přijímá HTTP poža-

davky od klienta. V HTTP požadavcích jsou definovány jednotlivé příkazy pomocí jazyka Selenese. Selenium RC Server tyto požadavky provádí nad testovanou aplikací v prohlížeči prostřednictvím programu Selenium Core, který dokáže příkazy v jazyku Selenese interpretovat ve webovém prohlížeči jako příkazy v jazyku JavaScript. Po proběhnutí testu zasílá server na klienta výsledný report, který informuje o úspěšnosti nebo selhání testu. Architekturu klient-server nástroje Selenium RC a směry komunikace zobrazuje obrázek 3.3.



Obrázek 3.3: Diagram architektury znázorňující klienta a Selenium RC server. Klient zasílá testovací instrukce na server, který test provádí.

Výhodou je, že testování nemusí probíhat pouze v prohlížečích Google Chrome nebo Mozilla Firefox, jako tomu bylo u nástroje Selenium IDE. Lze využít i další prohlížeče jako jsou například Opera, Safari, Edge nebo Internet Explorer [12]. Vytváření testů pro Selenium RC poskytuje více volnosti díky možnosti výběru z podporovaných programovacích jazyků. Zatímco v nástroji Selenium IDE se testy vytvářely primárně nahráváním akcí uživatele, v Selenium RC probíhá tvorba testů v programovacích jazycích, které umožňují vytvářet složitější a komplexnější testy. Zároveň tím vzniká ale nutnost ovládat nějaký programovací jazyk, což u rozšíření Selenium IDE nebylo nutné.

Selenium RC přestal být od druhé verze Selenia dále vyvíjen, ač zůstal stále podporován. Důvodem bylo uvedení nového nástroje Selenium WebDriver, který dokáže komunikovat s prohlížeči příměji – bez nutnosti použití Selenium RC serveru jako prostředníka. Pro vývojové týmy tak došlo ke zjednodušení infrastruktury nutné pro testování.

Selenium WebDriver

Selenium Web Driver označovaný také názvem *Selenium 2.0* [27] byl představen jako nástupce nástroje Selenium RC. Přichází zejména s jednodušší architekturou testovacího prostředí. Oproti nástroji Selenium RC nepotřebuje Selenium WebDriver spuštěný samostatný

server, který provádí testy. Selenium WebDriver komunikuje přímo s prohlížečem na počítači testera [46]. Pokud však tester chce, může využít s nástrojem Selenium WebDriver i Selenium Standalone Server, takže samotné testování neběží na jeho počítači, ale na serveru jako tomu bylo u Selenium RC – toho se využije zejména u větších projektů. Zároveň zůstaly zachovány výhody nástroje Selenium RC, jakými jsou například podpora široké škály programovacích jazyků, možnost testovat v různých prohlížečích nebo přenositelnost mezi různými platformami [69].

Drobné změny oproti nástroji Selenium RC můžeme u Selenium WebDriver najít ve zprávách o výsledcích testů. Zatímco Selenium RC dokázal sám vytvářet a zasílat zprávy o výsledcích testů, v nástroji Selenium WebDriver je tato funkcionalita přenechána vlastnímu řešení programátora nebo častěji spíše nějakému testovacímu rámci (anglicky *framework*) – například JUnit, NUnit, TestNG nebo Pytest (dle zvoleného jazyka pro psaní testů) [28]. V praxi je využití Selenia v kombinaci s testovacím rámcem naprosto typické a běžné [27, 74]. Taková kombinace nástrojů funguje tím způsobem, že Selenium zajišťuje pouze automatizaci prohlížeče (poskytuje testerovi příkazy, kterými může prohlížeč ovládat) a veškerou agendu okolo testování zajišťuje vybraný testovací rámec. Takovou agendou může být příprava prostředí před spuštěním testu, inicializace a spuštění testu, úklid a vrácení změn po proběhnutí testu nebo vytvoření zprávy o úspěšnosti proběhnutého testu.

Ze zkušeností s použitím Selenia pro automatizaci webových prohlížečů mohu zmínit velkou výhodu nástroje Selenium WebDriver oproti Selenium RC, kterou je zjednodušení a zmenšení instrukčního setu. Zatímco Selenium RC obsahuje na první pohled redundantní příkazy, Selenium WebDriver tyto příkazy sjednocuje. Redundance příkazů u Selenium RC je ale naneštěstí jenom zdánlivá, protože různé prohlížeče zdánlivě stejné příkazy vyhodnocují odlišně. Příkladem může být příkaz pro zadávání textu pomocí klávesnice:

- Selenium RC: type nebo typeKeys
- Selenium WebDriver: sendKeys()

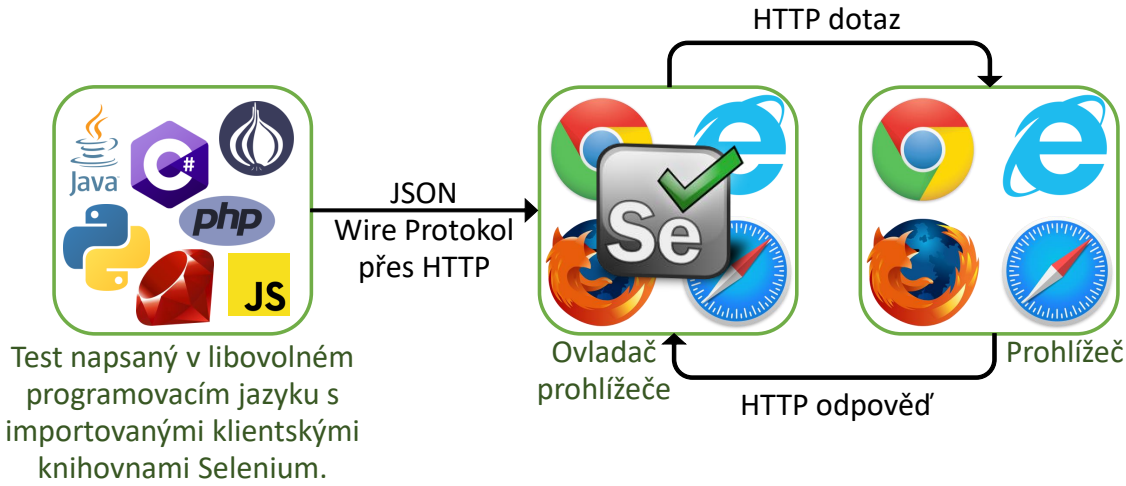
nebo příkaz pro kliknutí myší:

- Selenium RC: click, mouseDown, or mouseDownAt
- Selenium WebDriver: click().

Nevýhodou nástroje Selenium WebDriver oproti Selenium RC je nutnost mít pro každý prohlížeč, ve kterém má být webová aplikace testována, stažený ovladač (anglicky *driver*) [46]. To však také znamená, že pro daný prohlížeč musí být ovladač k dispozici. Pokud není, nelze použít Selenium WebDriver pro automatizaci prohlížeče. Tato vlastnost souvisí s rozdílným přístupem nástrojů Selenium RC a Selenium WebDriver k automatizaci prohlížeče. Selenium RC využívá program Selenium Core napsaný v jazyku JavaScript, který je zaveden do jakéhokoliv prohlížeče s podporou JavaScriptu a provádí akce definované v jazyku Selenese. Oproti tomu nástroj Selenium WebDriver funguje na jiném principu. Nedochází k přebrání kontroly nad funkcemi JavaScriptu, ale je využívána nativní podpora automatizace prohlížeče pomocí jeho ovladače, který většinou vydává sám výrobce prohlížeče. Z toho také vyplývá, že prohlížeč, ke kterému neexistuje ovladač, nelze pomocí nástroje Selenium WebDriver automatizovat. Při testování v praxi nás však tato skutečnost neomezí, protože pro většinu běžně používaných prohlížečů existují jejich ovladače: ChromeDriver, EventFiringWebDriver, GeckoDriver, HtmlUnitDriver, InternetExplorerDriver, PhantomJSDriver, RemoteWebDriver, SafariDriver.

S rozdílným přístupem k automatizaci webových prohlížečů souvisí i větší realističnost testování pomocí nástroje Selenium WebDriver oproti Selenium RC. Představme si, že máme ve formuláři políčko, do kterého nelze vpisovat (je zakázané). Selenium RC však bude schopný při testování do tohoto políčka vpisovat prostřednictvím JavaScriptového programu Selenium Core, kterého atribut `disabled` nezastaví. Selenium WebDriver se v tomto případě bude při testování chovat více jako reálný uživatel a do zakázaného políčka hodnotu nebude schopen doplnit [69].

Na obrázku 3.4 je znázorněna architektura nástroje Selenium WebDriver. Ač to na první pohled nemusí být jasně viditelné, jedná se opět o architekturu klient-server, kde klientem jsou knihovny Selenia a serverem je ovladač prohlížeče [46]. Komunikace mezi jednotlivými komponenty nástroje Selenium WebDriver funguje následovně. Po spuštění testu jsou příkazy klientskými knihovnami převedeny do formátu JSON a zaslány pomocí protokolu HTTP ovladači prohlížeče. Každý ovladač prohlížeče má jako svoji část HTTP server, který HTTP dotaz obdrží a zpracuje. Ovladač následně přijatou akci provede v odpovídajícím prohlížeči. Jak je zobrazeno na obrázku 3.4, komunikace mezi ovladačem a prohlížečem probíhá také prostřednictvím HTTP [86].



Obrázek 3.4: Diagram architektury znázorňující 4 hlavní části nástroje Selenium WebDriver: jazykově-specifické knihovny, JSON Wire protokol, ovladač prohlížeče a webový prohlížeč.

Velice příjemná pro testera je spolupráce nástroje Selenium WebDriver s rozšířením Selenium IDE. Selenium IDE nabízí možnost exportovat nahraný test do zdrojového kódu vybraných jazyků (Java, JavaScript, Python), který dokáže poté použít Selenium WebDriver. Tester si tak může ušetřit práci s psaním jednoduchého testu nebo menších součástí složitějších testů a pouze si je nahraje a exportuje do zvoleného jazyka. Případně může takto jednoduše zjistit, jaký příkaz je nutné zadat pro provedení odpovídající akce.

Při porovnání je určitě nutné zmínit i rychlost, ve které jednoznačně vede Selenium WebDriver [28]. To je dáno přímým ovládáním prohlížeče bez nutnosti komunikace přes Selenium RC server. A roli také hraje, že nástroj Selenium WebDriver ovládá přímo prohlížeč, zatímco Selenium RC server předává příkazy do Selenium Core, který teprve akce ve webovém prohlížeči provádí.

Selenium Grid

Selenium Grid je nástroj umožňující paralelní provádění testů v distribuovaném prostředí, které se skládá z jednoho zařízení v roli *Hub* (testovací server) a z libovolného počtu zařízení v roli *Node* (testovací uzly) [47]. Testovací server řídí celé testování. Testovací uzly mohou být zařízení s různými operačními systémy a s různými prohlížeči v různých verzích. Z topologického hlediska je testovací prostředí fyzická nebo virtuální počítačová síť, kde na jednom vybraném počítači běží Selenium Standalone Server v roli *Hub* a na ostatních běží v roli *Node*. Prostředí pro testování Selenium Grid využijeme například ve chvíli, kdy program musí být otestován na posledních 3 verzích prohlížečů Google Chrome, Mozilla Firefox a Safari, které běží na platformách Windows, Linux a MacOS. A se Selenium Grid ušetříme čas, protože testy na všech zařízeních poběží současně [9].

Selenium Grid nelze vnímat jako samostatný nástroj oddělený od dříve představených nástrojů z rodiny Selenium. Přesnější pojetí je spíše jako rozšíření nástroje Selenium WebDriver o možnost testování v distribuovaném prostředí. Pro běh testů paralelně potřebujeme kromě nástroje Selenium Grid použít ještě vybraný testovací rámec (např. TestNG), který bude testování řídit a zajistí paralelní spuštění testů. Architektura distribuovaného prostředí pro testování pomocí nástroje Selenium Grid je znázorněna na obrázku 3.5

3.2 Jasmine

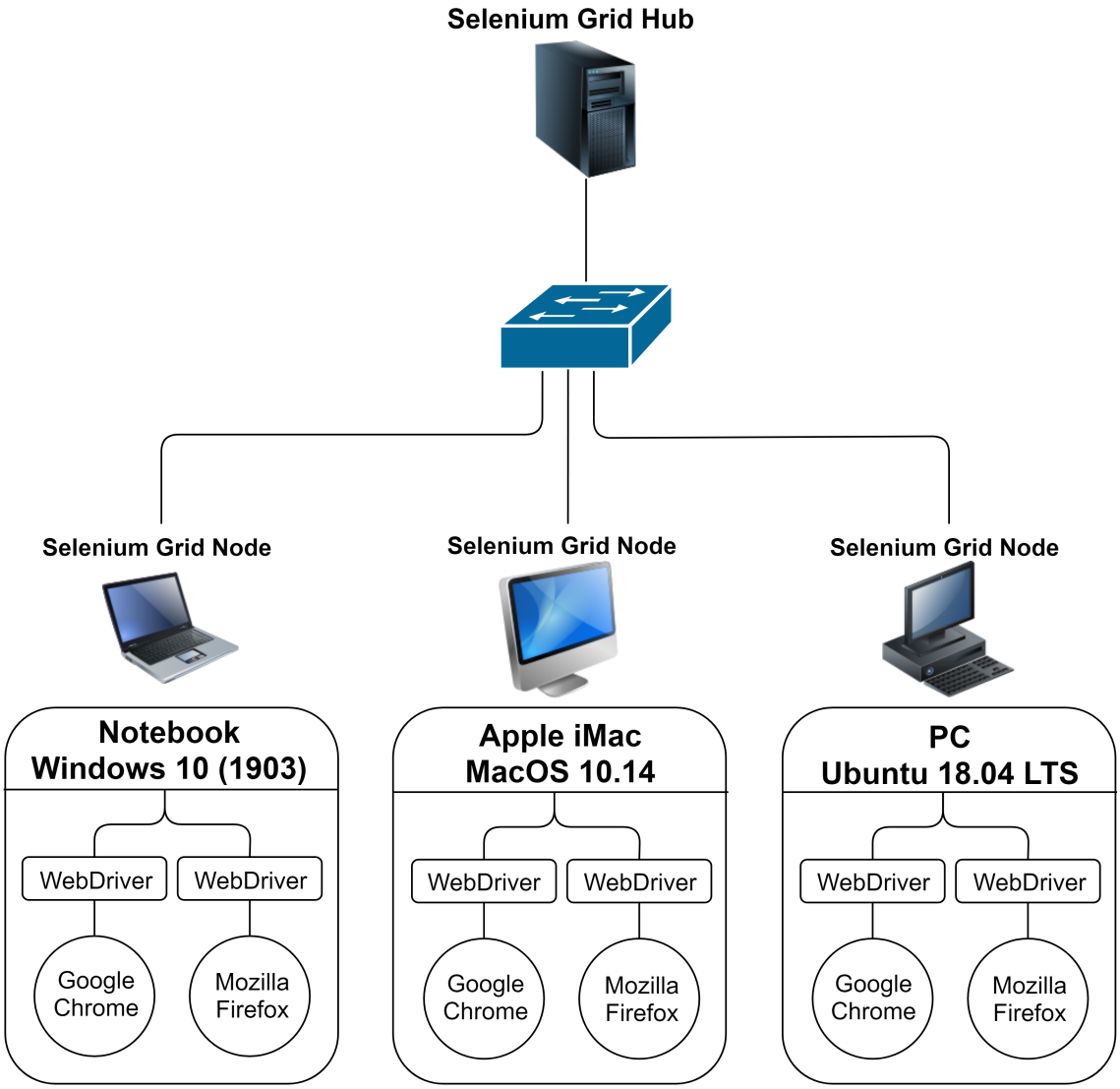
Jasmine je testovací rámec určený pro psaní jednotkových testů v jazyku JavaScript. Tento rámec se výborně hodí při vývoji řízeném požadavky na chování (anglicky *Behavior-Driven Development*, zkráceně *BDD*) [37], což je jeden z agilních přístupů pro vývoj softwaru, jenž se vyvinul z vývoje řízeného testy (anglicky *Test-Driven Developement*, zkráceně *TDD*) a jenž je založen na vytváření testovacích scénářů, kterým rozumí i zákazník, takže mohou sloužit do jisté míry i jako akceptační testy [22].

Nejnovější verzi Jasmine je možné stáhnout z GitHub[1]. V době psaní této práce byla poslední verze 3.5.0 s datem vydání 21. září 2019. Pokud stáhneme samostatnou (anglicky *standalone*) verzi rámce, nalezneme v adresáři mimo jiné i dvě složky. Jedna je pojmenovaná `src`. V této složce Jasmine očekává zdrojové soubory v jazyku JavaScript, které chceme otestovat. Ve složce `spec` se očekává definování testů dle dokumentace k tomuto nástroji. Soubor `SpecRunner.html` slouží ke spuštění testů a zobrazení výsledků testování.

Jasmine se vyznačuje přímým a čistým přístupem k tvorbě testů i k samotnému testování. Tento rámec nevyžaduje instalaci žádných dalších závislostí a ač Jasmine existuje i jako balíček pro vybraná prostředí (např. Node.js), sám o sobě nevyžaduje žádné speciální běhové prostředí. Spuštění testů i zobrazení výsledků je možné v libovolném prohlížeči podporující JavaScript.

Definice testů je založena na hierarchicky uspořádaných podmínkách, které se vztahují k elementárním částem zdrojového kódu. Taková podmínka, nazývaná v rámci Jasmine také jako *očekávání*, porovnává dvě hodnoty – skutečnou (např. návratovou hodnotu z testované funkce s daným vstupem) a očekávanou (hodnota definovaná testerem). Pro každý jednotkový test platí, že úspěšně projde, pokud je dané očekávání při spuštění testů splněno.

[1] https://github.com/jasmine/jasmine/releases



Obrázek 3.5: Diagram znázorňující příklad distribuovaného prostředí pro paralelní testování webové aplikace v prohlížečích Google Chrome a Mozilla Firefox na různých platformách s využitím nástroje Selenium Grid.

3.3 Add-ons Linter

Add-ons Linter je nástroj vyvinutý společností Mozilla sloužící k analýze vytvořeného rozšíření pro webový prohlížeč Mozilla Firefox [59]. Add-ons Linter lze použít lokálně při vývoji rozšíření ještě před jeho nahráním do obchod Firefox Browser Add-ons. Tento nástroj je publikovaný jako balíček pro Node.js [33].

Proces analýzy testovaného rozšíření nástrojem Add-ons Linter do jisté míry připomíná počáteční fáze práce překladače. Zpracování jednotlivých zdrojových souborů testovaného rozšíření zahajují takzvané *skenery*. Pro každý vybraný typ souborů existuje skener, který načte dokumenty příslušného typu a provede lexikální analýzu. Následně je takto předzpra-

covaný dokument porovnáván vůči pravidlům interně definovaných v nástroji. Při nesplnění nějakého pravidla je na modul *kolektor* zaslána zpráva. Kolektor posbírá všechny zprávy a předá je poslednímu modulu, který zajistí textový výstup analýzy ve formátu JSON.

Add-ons Linter dokáže identifikovat a rozlišit přibližně 100 problémů z různých oblastí vývoje – od syntaktické chyby v CSS stylech, přes různé chyby v manifestu, až po běhové chyby jazyka JavaScript (zkráceně *JS*). Příkladem problémů v rozšíření, na které dokáže nástroj upozornit, jsou [32]:

- Varování: Nedoporučovaná verze JS knihovny.
- Chyba: Tato verze JS knihovny je zakázána z bezpečnostních důvodů.
- Varování: Zastaralé API.
- Varování: Použití `document.write` se důrazně nedoporučuje.
- Varování: Dočasná ID mohou způsobit chybu se `storage.sync`.
- Varování: API není kompatibilní s `applications.gecko.strict_min_version`.
- Chyba: CSS selektor nemůže být vnořený.
- Varování: CSP zakazuje vložené skripty.

3.4 Mocha

Mocha je testovací rámec pro prostředí Node.js [18]. Svým zaměřením spadá zejména do kategorie rámců pro tvorbu jednotkových testů, ale lze ho úspěšně využít i pro testování na vyšších úrovních. Rámec Mocha je vhodný zejména pro projekty vyvíjených pro prostředí Node.js, protože testy vytvořené v rámci Mocha lze jednoduše do projektu napojit jako jeho součást a spouštět příkazem `npm test`.

Testy se zapisují v jazyku JavaScript a jejich struktura je velmi podobná hierarchickému uspořádání testů v rámci Jasmine. Porovnávání jednotlivých hodnot lze v testech provádět příkazem `assert` nebo lze nainstalovat specializovanou knihovnu (např. Chai, Should.js nebo Express.js) a využít pokročilou syntaxi porovnávacích příkazů dané knihovny.

Knihovna Chai, často využívána jako podpůrná knihovna pro testy v rámci Mocha, nabízí možnost zapisovat příkazy `assert` v pokročilejší notaci, která je výrazně přehlednější a je dobře čitelná například i pro zadavatele projektu. Knihovnu Chai lze tak úspěšně využít při vývoji řízeném požadavky na chování (*Behaviour-driven development*, zkráceně BDD).

3.5 Shrnutí

V této kapitole byly představeny vybrané nástroje, které lze použít pro testování rozšíření do webových prohlížečů. Cílem nebylo představit všechny dostupné nástroje (to by bylo vzhledem k jejich počtu nemožné), ale ukázat vybrané nástroje, které by mohly být využity při implementaci automatických testů pro projekt JavaScript Restrictor.

Výsledný výběr vhodných nástrojů pro implementaci testů závisí na mnoha faktorech. Důležitým kritériem může být, aby testy celkově zapadly do projektu. Pokud tedy projekt není vyvíjen pro prostředí Node.js, nedává smysl vytvářet testy v rámci fungujícím výhradně nad tímto prostředím, ale bude lepší vybrat rámec, který lze začlenit do stávající prostředí nebo samostatný rámec (anglicky *standalone*) nezávislý na běhovém prostředí.

Kapitola 4

Žebříčky nejnavštěvovanějších webových stránek

Odpověď na otázku, jaké stránky patří mezi nejvíce navštěvované, mohou poskytnout veřejně publikované žebříčky nejnavštěvovanějších webových stránek. Nejnavštěvovanější webové stránky je možné využít jako určitý vzorek webu pro účely testování. V podsekcích níže jsou představeny vybrané žebříčky včetně zdrojů dat k analýze, které používají, a metodiky, které slouží k výpočtu pozice webové stránky v žebříčku. U čtyřech nejpopulárnějších žebříčků (Alexa, Cisco Umbrella, Majestic a Quantcast [52]) je také doplněno hodnocení důvěryhodnosti, které se opírá o práci Le Pochata [52]. Na konci každé podsekce jsou rozepsány cenové podmínky, v nichž se jednotlivé žebříčky výrazně liší.

4.1 Alexa

Alexa Internet, Inc. je společnost zabývající se analýzou webového provozu. Byla založena již v roce 1996 [3] a v roce 1999 ji odkoupila společností Amazon [24], jejíž součástí je dodnes.

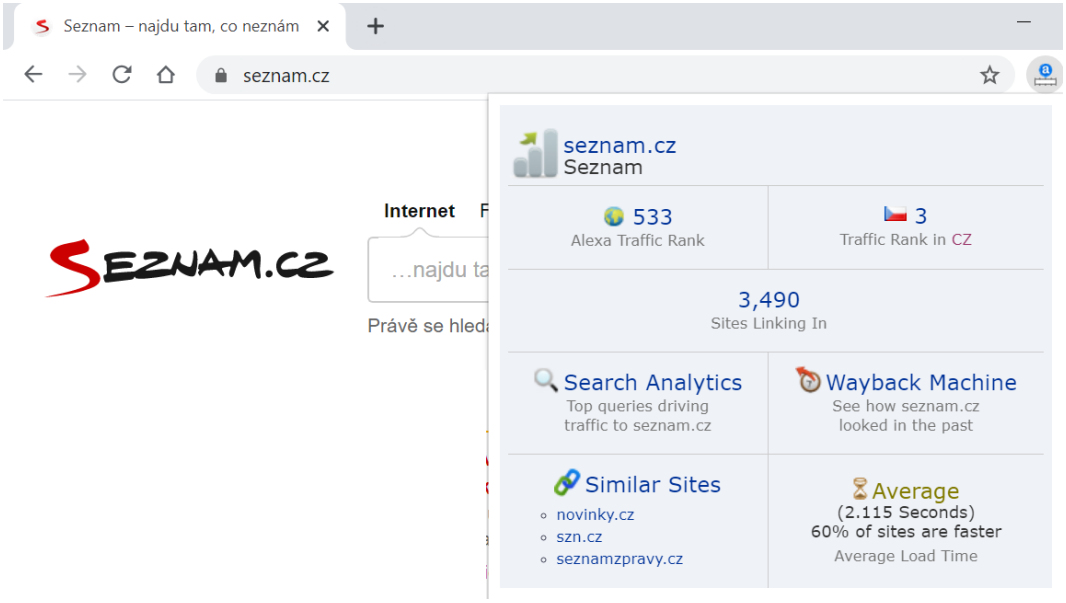
Data pro analýzu webového provozu jsou získávána primárně z několika rozšíření webových prohlížečů, které mají nainstalovány milióny uživatelů [3]. Rozšíření webového prohlížeče zaznamenává navštívené stránky a odesílá tyto informace společnosti Alexa. Na základě zaslaných dat jsou poté sestavovány žebříčky[1] a další analýzy webového provozu. Žebříček nejnavštěvovanějších webových stránek je sestaven na základě posledního tříměsíčního průměru počtu návštěv webových stránek [6]. Tento průměr je aktualizován každý den. U počtu návštěv se rozlišuje počet unikátních návštěvníků a počet opakovaných návštěv, přičemž vyšší váhu má počet unikátních návštěv. Žebříček od Alexa zobrazuje položky na úrovni doménových jmen druhého a třetího řádu, nikoliv na úrovni jednotlivých stránek [6]. Přístupy na podstránky se nerozlišují a započítávají se k příslušné doméně.

Konkrétní počet uživatelů, od kterých jsou data o navštívených webových stránkách získávána, lze jen odhadovat. Na oficiálních webových stránkách Alexy jsou uvedeny pouze řády, a to jsou již zmíněné miliony [3]. Počet instalací oficiálního rozšíření do prohlížečů se může pohybovat kolem jednoho milionu, přičemž údaje o počtu stažení jsou dostupné pouze v Internetovém obchodě Chrome [4], protože rozšíření pro prohlížeč Mozilla Firefox není distribuováno přes oficiální obchod s doplňky, a proto není k dispozici ani počet instalací v tomto prohlížeči. Počet stažení pro Google Chrome je přibližně 500 000. Oficiální rozšíření

[1] https://s3.amazonaws.com/alexa-static/top-1m.csv.zip

webových prohlížečů *Alexa Traffic Rank* již dnes není jediným zdrojem informací. Alexa využívá data i z jiných spolupracujících rozšířeních a také data ze sledovacích serverových skriptů spuštěných na vybraných spolupracujících serverech [6].

Motivací pro instalaci rozšíření Alexa do webového prohlížeče může pro uživatele být možnost podílet se svojí aktivitou na sestavování žebříčků návštěvnosti webových stránek. Častější motivací však pravděpodobně je možnost zjištění pozice své vlastní nebo navštívené webové stránky v žebříčku, přímý odkaz na historické verze webové stránky prostřednictvím služby *WayBack Machine* [45] a informace o podobných stránkách, které by uživatele mohly zajímat. Vzhled otevřeného rozšíření v prohlížeči Google Chrome pro doménu `seznam.cz` lze vidět na obrázku 4.1.



Obrázek 4.1: Rozšíření *Alexa Traffic Rank* verze 4.0.4 ve webovém prohlížeči Google Chrome.

Nevýhodou tohoto žebříčku může být geografická neobjektivnost, protože se zdá, že od května 2018 je zastaveno získávání analytických dat z počítačů z oblasti EHP[2] z důvodu nařízení GDPR [52]. Výsledná pozice webové stránky v žebříčku na základě počtu navštívení je tak zkreslená. Žebříček Alexa je vhodný pro analýzu provozu v lokálním měřítku konkrétního státu mimo prostor EHP, nikoliv však v celosvětovém měřítku.

Další slabinou žebříčku Alexa je relativně snadná ovlivnitelnost žebříčku. Ač společnost Alexa Internet tvrdí ve své dokumentaci, že žebříček je chráněn proti cílenému ovlivnění [5], Le Pochatovi se podařilo provést úspěšný útok s cílem zařadit novou doménu do žebříčku na významnější místo. S pomocí prohlížeče s rozšířením *Alexa Traffic Rank* se opakovaným automatickým navštěvováním podařilo posunout tuto novou doménu až na 370 461. místo v žebříčku nejnavštěvovanějších webových stránek Alexa [52].

Žebříček obsahuje jeden milion nejnavštěvovanějších stránek, avšak bez předplatného je možné z webových stránek nástroje Alexa zobrazit pouze prvních 50 míst z žebříčku [8]. Kompletní žebříček je dostupný pouze uživatelům s předplatným, jehož levnější varianta stojí $149/měsíc [7]. Data z žebříčku jsou poskytována i přes API, ale opět pouze za po-

[2]Evropský hospodářský prostor

platek, který v tomto případě činí $0,0025 za získání jedné pozice (jedné domény) z žebříčku [72].

4.2 Cisco Umbrella

Cisco Umbrella je bezpečnostní řešení od společnosti Cisco, které nabízí včasné hodnocení rizika webové stránky, na kterou se uživatel snaží připojit. Řešení je založeno na hodnocení DNS dotazu. Cisco provozuje vlastní DNS servery. Služba, kterou poskytují se nazývá OpenDNS. Administrátor firemní sítě nastaví adresy DNS serverů na firemních zařízeních, aby směřovaly právě na OpenDNS servery. Tyto DNS servery vyhodnotí reputaci domény na základě dříve získaných informací již při překladu doménového jména na IP adresu a přístup na webovou stránku buď povolí nebo zablokují [73]. Toto řešení je vhodné zejména v případě, kdy se zaměstnanci z pracovních notebooků připojují k veřejné síti mimo kanceláře. Uvnitř firemní sítě mají administrátoři díky firewallu dostatek informací o tom, jaké stránky uživatelé navštěvují, a dokáží je velmi efektivně blokovat. Avšak ve veřejné síti nemají nad filtrováním síťového provozu kontrolu a případné blokování domén lze zajistit jen komplikovanými cestami [89].

Na základě služby Cisco Umbrella existuje od roku 2016 žebříček *Cisco Umbrella 1 Million*[3]. Ten obsahuje jeden milion nejnavštěvovanějších domén (ne jenom druhého řádu [85]) na základě DNS dotazů z OpenDNS serverů. Cisco uvádí, že žebříček je založen na více než 100 miliardách DNS dotazů za den od přibližně 65 milionů unikátních uživatelů z více než 165 zemí světa [39]. Samotný výpočet pozice domény v žebříčku není dán jednoduše porovnáním součtů počtu DNS dotazů na domény. Při výpočtu hraje také roli počet unikátních návštěvníků, který pozici v žebříčku zvyšuje. Ve výsledku je tedy důležité nejen to, kolik přišlo dotazů na překlad doménového jména, ale také z kolika unikátních zařízeních tyto dotazy přišly [39]. Žebříček je aktualizován každý den.

Žebříček *Cisco Umbrella 1 Million* vznikl jako reakce na zpoplatní získávání dat z žebříčku Alexa [39]. Společnost Cisco tak v roce 2016 nabídla svůj vlastní žebříček dostupný zdarma. Získávání dat z DNS dotazů způsobuje, že jsou započítávány do statistik i přístupy na domény, které nevedou z prohlížečů. Jedná se o domény, na které je přistupováno například programy běžícími na pozadí. To se v žebříčku projevuje tak, že se v něm občas objeví i nevalidní domény – např.: interní domény jako *.ec2.internal získané z DNS dotazů na službu v Amazon EC2 (zkratka *Elastic Compute Cloud*) [52]. Žebříček také obsahuje domény, které mohly být zadány ve webovém prohlížeči uživatelem do adresního řádku, ale jednalo se o chybu. Projevem takových chyb je to, že průměrně 28 % domén z žebříčku je nedosažitelných a dalších 20 % webových stránek vrací při pokusu o přístup serverovou chybu (kód 4xx) [52].

4.3 Majestic

Anglická společnost Majestic se zabývá analýzou a mapováním internetu, zejména jeho části *World Wide Web*. V rámci své činnosti vytvořila databázi *Link Intelligence* [56], ve které je obsažena a pravidelně aktualizována „mapa webu“. Dle Majestic je každý den navštíven a automaticky analyzován obsah přibližně 450 miliard URL [55] za účelem aktualizace dat v databázi. Hlavní oblastí analýzy jsou zpětné odkazy na webu, tedy zjišťování, kolik

[3] https://s3-us-west-1.amazonaws.com/umbrella-static/top-1m.csv.zip

webových stránek odkazuje na danou doménu. Nashromážděné informace slouží například majitelům webových stránek při SEO optimalizaci a jiných marketingových aktivitách [56].

Na základě každodenní analýzy webu je od října 2012 sestaven a aktualizován žebříček *Majestic Million*[4], který prezentuje milion nejnavštěvovanějších domén druhého a třetího řádu [58]. Domény jsou seřazeny na základě počtu zpětných odkazů, které na ně vedou. Přesněji, domény jsou seřazeny dle počtu podsítí třídy C (IPv4 /24), které na ně odkazují alespoň jednou [43]. Pozice domén v žebříčky jsou aktualizovány na základě jejich posledního 120 denního průměru [55].

Specifikem žebříčku *Majestic Million* je, že obsahuje pouze doménová jména, na které vedou zpětné odkazy [52]. Pokud bychom tedy měli hojně navštěvovanou doménu, která je v síti osamocena, uživatelé na ni přistupují přímo přes odkaz, a tak na ni nevedou žádné zpětné odkazy, nebude do žebříčku vůbec zahrnuta.

Slabou stránkou žebříčku *Majestic Million* je relativně vysoký počet domén, na nichž provozované webové stránky obsahují potenciálně nebezpečný obsah nebo provádí sociální inženýrství (např. phishing) dle kontroly službou *Google Safe Browsing* [30]. V celém žebříčku bylo nalezeno více než 2 000 domén s takovým nebezpečným obsahem, z toho jedna doména byla zařazena mezi 10 000 nejvyšších pozic žebříčku [52].

Majestic Million se společně s *Cisco Umbrella 1 Million* řadí k žebříčkům nejnavštěvovanějších stránek, které data poskytují zdarma. *Majestic Million* nabízí kompletní seznam zdarma pod licencí *Creative Commons Attribution 3.0 Unported License* [58]. Pokročilé služby od Majestic zahrnující například SEO analýzu a optimalizaci webu jsou však dostupné už pouze předplatitelům. Předplatné služeb začíná na částce $49,99/měsíc [57].

4.4 Quantcast

Quantcast je americká technologická společnost, která se zaměřuje na analýzy webového provozu, zejména na měření návštěvnosti webových stránek včetně zjišťování údajů o návštěvnících (pohlaví, věk, lokalita, vzdělání, zájmy apod.) [66].

Společnost Quantcast byla založena v roce 2006 [65] a již v roce 2007 [52] zveřejnila svůj vlastní žebříček nejnavštěvovanějších webových stránek[5]. Žebříček je však relevantní pouze pro USA, protože zohledňuje pouze americký internetový provoz [52]. Ač je žebříček pojmenován *Top Million U.S. Web Sites* (v překladu *Milion nejnavštěvovanějších amerických webů*), obsahuje aktuálně pouze kolem 500 000 domén druhého a třetího řádu [67]. Seznam obsahuje také takzvané „skryté profily“, což jsou záznamy domén, které jsou měřeny a analyzovány, ale jejichž jméno je z určitého důvodu v žebříčku skryté.

Žebříček *Top Million U.S. Web Sites* je sestaven na základě tří zdrojů [52]. Prvním zdrojem jsou skripty sloužící k analýze a monitoringu návštěvnosti na webových stránkách zákazníků. Quantcast uvádí, že monitoruje přibližně 100 milionů domén [65]. Dalším zdrojem dat jsou záznamy webového provozu od spolupracujících amerických poskytovatelů internetového připojení (zkráceně *ISP* z anglického označení *Internet Service Provider*). Posledním zdrojem jsou partnerská rozšíření do prohlížečů podobně jako v případě žebříčku Alexa. Ač jsou data o návštěvnosti webů sbírána a ukládána každý den, samotný žebříček *Top Million U.S. Web Sites* je aktualizován měsíčně a pozice domén v žebříčku jsou založeny právě na měření z posledního měsíce. Žebříček je možné stáhnout a používat zdarma při dodržení podmínek užití [68].

[4] http://downloads.majestic.com/majestic_million.csv
[5] https://ak.quantcast.com/quantcast-top-sites.zip

4.5 TRANCO

V předcházejících podsekcích byly představeny čtyři nejpopulárnější žebříčky nejnavštěvovanějších webových stránek včetně zdroje dat a metodik, jaké jsou pro výpočet pozice domény v žebříčku použity. Každý žebříček staví na jiném zdroji dat a nelze se divit, že mezi jednotlivými žebříčky existují často výrazné odlišnosti. Tabulka 4.1 zobrazuje prvních 25 míst ze všech žebříčků pro porovnání. Le Pochat se zabýval otázkou důvěryhodnosti žebříčků [52]. Vlastním výzkumem objevil několik skutečností o čtyřech zmíněných žebřících nejnavštěvovanějších webových stránek:

- Podobnost: Žebříčky se shodují průměrně pouze v 2.48 % populárních domén. To znamená, že všechny čtyři žebříčky, které dohromady obsahují přes 2,82 milionů domén, se shodují na pouze přibližně 70 000 domén.

	Alexa	**Cisco Umbrella**	**Majestic**	**Quantcast**
1	google.com	google.com	facebook.com	google.com
2	youtube.com	microsoft.com	google.com	facebook.com
3	tmall.com	www.google.com	youtube.com	amazon.com
4	baidu.com	netflix.com	twitter.com	youtube.com
5	qq.com	api-global.netflix.com	instagram.com	wikipedia.com
6	facebook.com	safebrowsing.googleapis.com	linkedin.com	yelp.com
7	sohu.com	facebook.com	microsoft.com	giphy.com
8	login.tmall.com	doubleclick.net	apple.com	twitter.com
9	taobao.com	windowsupdate.com	wikipedia.org	fandom.com
10	360.cn	clients4.google.com	plus.google.com	dailymail.co.uk
11	jd.com	g.doubleclick.net	en.wikipedia.org	quizlet.com
12	yahoo.com	ctldl.windowsupdate.com	adobe.com	hidden profile
13	wikipedia.org	data.microsoft.com	youtu.be	hidden profile
14	amazon.com	live.com	itunes.apple.com	buzzfeed.com
15	sina.com.cn	clientservices.googleapis.com	googletagmanager.com	realtor.com
16	weibo.com	googleads.g.doubleclick.net	vimeo.com	espn.com
17	pages.tmall.com	www.googleapis.com	pinterest.com	hidden profile
18	live.com	google-analytics.com	play.google.com	yahoo.com
19	reddit.com	settings-win.data.microsoft.com	goo.gl	msn.com
20	vk.com	googleusercontent.com	maps.google.com	urbandictionary.com
21	netflix.com	events.data.microsoft.com	wordpress.com	hidden profile
22	xinhuanet.com	skype.com	blogspot.com	thehill.com
23	okezone.com	officeapps.live.com	bit.ly	cheatsheet.com
24	blogspot.com	edge.skype.com	player.vimeo.com	hidden profile
25	bongacams.com	config.edge.skype.com	github.com	whitepages.com

Tabulka 4.1: 25 prvních záznamů ze 4 porovnávaných žebříčků nejnavštěvovanějších webových stránek.

- Stabilita: Stabilita žebříčku vyjadřuje průměrné procentuální množství změn za jeden den. Nejstabilnějšími žebříčky jsou Majestic a Quantcast, ve kterých se denně změní pouze přibližně 1 % záznamů. Umbrella dosahuje průměrné změny 10 % záznamů za den a v žebříčku Alexa se denně změní téměř 50 % všech záznamů.

- Dosažitelnost: Dosažitelnost označuje průměrnou procentuální část domén z žebříčku, které jsou dosažitelné a při HTTP dotazu na kořenovou stránku vrací kód `2xx` (`Success`). Žebříčky Alexa a Quantcast mají dosažitelných přes 90 % domén, což je nejvíce v porovnání s ostatními. Žebříček Majestic má jen o něco málo méně – přibližně 85 % domén z žebříčku je dosažitelných. Naopak žebříček Umbrella nemá ani 50 % záznamů dosažitelných. Důvody takto vysoké nedosažitelnosti jsou rozebrány v podsekci věnované žebříčku Umbrella.

- Bezpečnost: Bezpečnost žebříčku je vyjádřena prostřednictvím počtu potenciálně nebezpečných webových stránek nacházejících se na doménách z žebříčku. Potenciální nebezpečnost byla zjišťována pomocí *Google Safe Browsing*. Čím nižší počet odkazů z žebříčku označil *Google Safe Browsing* za nebezpečné buď z důvodu škodlivého softwaru na webových stránkách nebo z důvodu provozování sociálního inženýrství, tím bezpečnější je žebříček vnímán. Nebezpečnějším žebříčkem se na základě analýzy staly Quantcast a Alexa, které obsahovaly kolem 500 nebezpečných odkazů v přepočtu na milion záznamů v žebříčku. Žebříček Umbrella skončil s dvojnásobným počtem nebezpečných odkazů na třetím místě a jako nejméně bezpečný žebříček byl vyhodnocen žebříček Majestic, který překonal hranici 2 000 nebezpečných odkazů.

- Manipulovatelnost: Manipulovatelnost žebříčku vyjadřuje množství úsilí nutné vyvinout k tomu, aby útočník zařadil svoji vlastní doménu na významnější pozice v žebříčku. Tato veličina není exaktní a také není vždy porovnatelná, ale spíše popisuje množství času a financí nutných k ovlivnění žebříčku. Jako nejzranitelnější vůči manipulaci se ukázal žebříček Alexa, u kterého stačilo zaslat pouze 12 HTTP dotazů na vybranou doménu z prohlížeče s nainstalovaných bezplatným oficiálním rozšířením, aby se doména dostala na 370 461. místo v žebříčku. Ač si ostatní žebříčky vedly lépe, náchylné k manipulaci jsou také. Nejlépe dopadl žebříček Majestic, k jehož manipulování by bylo nutné vyvinout vysoké úsilí a investovat značné množství peněz, aby na doménu, kterou chce útočník v žebříčku posunout nahoru, začal vést co nejvyšší počet zpětných odkazů.

Z důvodu zjištěných problémů u čtyřech analyzovaných žebříčků se výzkumná skupina rozhodla vytvořit nový žebříček pojmenovaný TRANCO[6], který by svojí vypovídající hodnotou více vyhovoval potřebám výzkumu [63]. Žebříček TRANCO nedisponuje vlastním zdrojem dat, na jejichž základu by mohl být vystavěn žebříček nejnavštěvovanějších webových stránek, ale kombinuje všechny čtyři představené žebříčky [62]. TRANCO je aktualizován každý den a ve výchozím nastavení je pozice domén v žebříčku počítána na základě všech čtyř žebříčků za posledních 30 dnů [52]. Tím TRANCO dosáhlo výrazně vyšší stability než zdrojové žebříčky. Denní proměnlivost žebříčku TRANCO se pohybuje pod 0,6 % [52]. Kromě samotné kombinace žebříčků nabízí TRANCO pokročilé možnosti filtrování a výběru zdrojů dat. Výchozí hodnota 30 dnů, za které probíhá kombinování žebříčků, lze libovolně měnit a lze také některé žebříčky vyřadit, aby se nepodílely na výsledném seznamu domén. Zároveň lze aplikovat například filtry pro odebrání nereagujících domén (domény bez odpovědi) nebo domén s potenciálně nebezpečným obsahem. TRANCO obsahuje již předdefinované kombinace filtrů, které stačí vybrat, nebo je možné si velice podrobně definovat filtry vlastní.

[6] https://tranco-list.eu/#download

4.6 NetMonitor

NetMonitor je projekt realizovaný polskou společností Gemius S.A., jehož cílem je poskytnout informace o návštěvnosti internetu a sociodemografickém profilu jeho návštěvníků v České republice [79]. Zadavatelem projektu je SPIR neboli Sdružení pro internetový rozvoj v České republice, z.s.p.o., které působící v oblasti monitoringu a analýzy internetu v České republice od roku 2000 [80].

V rámci projektu NetMonitor je publikován žebříček[7] návštěvnosti domén druhého řádu, které jsou zapojeny do projektu. Umístění domény druhého řádu v žebříčku nelze brát absolutně vzhledem k veškerému webovému provozu v ČR, neboť žebříček poskytuje pouze porovnání v rámci monitorovaných domén. Pokud se tedy například doména `aktualne.cz` nachází na 6. místě v žebříčku s přibližně jedním milionem jedinečných návštěvníků za den (údaj k 27. 2. 2020) [26], nelze pozici brát tak, že `aktualne.cz` je šestou nejnavštěvovanější doménou v ČR, ale z pozice vyplývá pouze to, že je šestou nejnavštěvovanější doménu mezi doménami analyzovanými v rámci projektu NetMonitor. V době vzniku této práce je do projektu NetMonitor zapojeno 464 českých domén [26]. Projekt slouží například pro oficiální srovnání návštěvnosti různých online médií [50]. Na rozdíl od dříve představených žebříčků totiž poskytuje NetMonitor absolutní čísla návštěvnosti, a to díky skriptům umístěných přímo na webových serverech, které se do projektu zapojily [25]. NetMonitor tak nabízí relativně omezený, ale o to přesnější pohled na webový provoz v České republice.

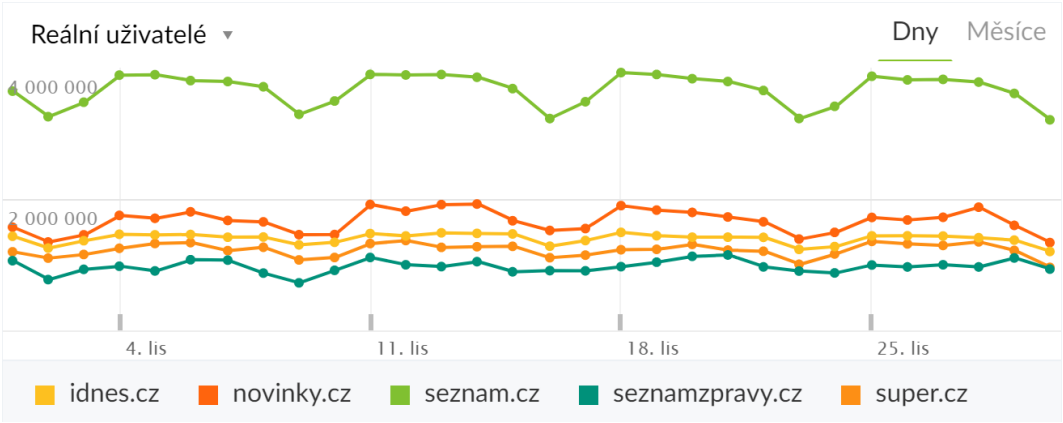
Výsledný žebříček nejnavštěvovanějších webových stránek je poskytován široké veřejnosti zdarma [26]. Reklamním agenturám nebo jiným společnostem, které požadují kompletní detailní údaje nebo které chtějí žebříček využívat komerčně, jsou data z měření poskytována za úplatu. Aktuální jednorázový poplatek za kompletní naměřená data činí 18 400,- Kč [81]. Webové servery, které chtějí být součástí žebříčku musí za členství platit měsíční poplatek a spustit na svém serveru monitorovací skript. Měsíční poplatek je závislý na množství webového provozu na serveru a dalších podmínkách. Pokud například provozovatel webového serveru souhlasí se zobrazováním dotazníků v rámci projektu NetMonitor svým návštěvníkům, získá slevu 25 % [81].

Zdrojem dat k sestavení žebříčku jsou již zmíněné serverové skripty měřící návštěvnost přímo na webových serverech zapojených do projektu. Není to však jediný zdroj. Další data jsou získávána přímo od uživatelů prostřednictvím výzkumného panelu [25]. Do výzkumného panelu se uživatel zapojuje vyplněním dotazníku. Poté je monitorována jeho aktivita na webových stránkách zapojených do projektu NetMonitor. Pokud si ještě k tomu uživatel nainstaluje aplikaci NetSoftware, poskytuje údaje o kompletním procházení webu včetně návštěv webových stránek, které do projektu zapojeny nejsou.

Data je možné zobrazit za jednotlivé dny, týdny a měsíce dle výběru. Při zobrazení za časový úsek delší než jeden den, nedochází k průměrování počtu návštěv domény jako u dříve představených žebříčků, ale data jsou sčítána. Dostáváme tak sumu všech návštěvníků na jednotlivých doménách. Podle této sumy jsou domény seřazeny v řebříčku. Návštěvnost lze zobrazit i graficky prokládaným bodovým nebo sloupcovým grafem. Vývoj počtu reálných návštěvníků pěti nejnavštěvovanějších domén v jednotlivých dnech za měsíc listopad roku 2019 je zobrazen na obrázku 4.2. Z grafu lze vyčíst, že na návštěvnost má vliv víkend, kdy návštěvnost klesá. Nejnavštěvovanější doménou z analyzovaných je doména `seznam.cz`, která dosahuje 4 milionů návštěvníků za den. V grafu je na ose y počet reálných návštěvníků. NetMonitor rozlišuje mezi reálných návštěvníkem a unikátním návštěvníkem. Reálný návštěvník odpovídá skutečnému člověku, který daný webový server ve zkoumaném ob-

[7] `http://www.netmonitor.cz/online-data-ola`

dobí navštívil. Unikátní uživatel pak bývá ekvivalentem pro identifikátor uložený v *cookie*, kterým je určen konkrétní prohlížeč. Jeden člověk (reálný uživatel) může používat více počítačů/prohlížečů, své cookies mazat a může tedy být identifikován jako více unikátních uživatelů [78].



Obrázek 4.2: Vývoj počtu reálných návštěvníků pěti nejnavštěvovanějších domén v jednotlivých dnech za měsíc listopad roku 2019 dle projektu NetMonitor [26].

Kapitola 5

Podpůrné metody pro automatické testování webových stránek

Pro efektivní testování webových stránek existují metody pro manipulaci s webovou stránkou. Tato kapitola se zaměřuje na metody procházení webových stránek do šířky i do hloubky a na automatické zjišťování očekávaného chování. Teoretické poznatky z této kapitoly jsou dále využity při návrhu a implementaci zejména systémových testů.

5.1 Metody procházení webových stránek

Pro přesnější vymezení termínu „*webová stránka*“ jsou použity anglické pojmy „*website*“ a „*webpage*“. Pojem „*website*“ označuje celou webovou prezentaci nebo webovou aplikaci skládající se z jednotlivých webových stránek označovaných anglicky „*webpage*“. Při procházení webových stránek do šířky je primárně užíván pojem webová stránka ve smyslu anglického slova „*website*“ a při procházení do hloubky je pojem webová stránka používán hlavně ve smyslu „*webpage*“.

Procházení do šířky

Procházením webových stránek do šířky se rozumí postupné navštěvování domén nebo konkrétních webových stránek z určitého seznamu. Takovým seznamem může být například žebříček nejnavštěvovanějších webových stránek.

Postup procházení webových stránek do šířky je vcelku přímočarý. Je nutné mít určitý zdroj webových adres. Po načtení prvních `n` záznamů ze zdrojového souboru do pole slouží toto pole jako datová struktura, kterou lze v cyklu procházet a postupně navštěvovat webové stránek na daném indexu pole.

Procházení do hloubky

Procházení do hloubky označuje detailní projití jednotlivých webových stránek (anglicky *webpages*) na testované webové prezentaci (anglicky *website*) nebo na testované doméně. Po otevření webové prezentace při procházení do šířky je třeba zjistit strukturu této webové prezentace a najít odkazy na dílčí webové stránky. K automatickému nalezení podstránek lze využít několik zdrojů:

- položky rozcestníku (element `<nav>` [53]) – HTML5 definuje novou značku `<nav>`, která slouží k vytvoření hlavního rozcestníku. Jejími položkami jsou odkazy (elementy

<a>) a z nich lze získat adresu odkazované stránky z atributu href. Odkazy nemusí být přímými následníky elementu <nav>, ale mohou být ještě zapouzdřeny například do seznamu (element <ul>). Značka <nav> slouží ke sjednocení označení hlavních rozcestníků, avšak ne všechny weby tento standard HTML5 již implementovaly.

- odkazy (element <a> [53]) – obecnějším řešením je procházet DOM a sbírat všechny elementy <a> a z nich následně využít atribut href. Nevýhodou může být velké množství získaných odkazů a také, že se nemusí jednat pouze o podstránky, ale i o odkazy na jakýkoliv obsah daného webu, případně o odkazy vedoucí pryč z aktuální domény.

V obou případech je třeba každý odkaz zkontrolovat, jestli směřuje na webovou stránku na stejné doméně. Odkazy, které vedou na jinou doménu je možné zahazovat, protože cílem procházení do hloubky je pouze analýza aktuální webové prezentace, nikoliv hledání vazeb mezi jakýmikoliv webovými stránkami.

Získávání odkazů (elementů <a>) je možné v případě potřeby důkladnější víceúrovňové analýzy volat rekurzivně pro každou nově otevřenou webovou stránku na dané doméně a nově získané odkazy ukládat do stromové struktury nebo obecně do grafu. Aby graf neobsahoval duplicitní hodnoty, je nutné porovnávat získaný odkaz s odkazy již uloženými. Výhodou sestavení stromu pro webovou prezentaci je hierarchický přehled webových stránek a možnost využít stromové algoritmy pro procházení jednotlivých webových stránek na jedné doméně. Ukládání do obecného grafu sice přináší výhodu uchování informace, jaká webová stránka odkazuje na jakou, ale zase se vytrácí možnost jednoduchého procházení grafu, jako je tomu u stromu.

5.2 Automatizované zjišťování očekávaného chování

Pro testování interaktivní webové stránky je potřeba umět rozpoznat jednotlivé změny, které uživatelem provedené události vyvolají. Změnu webové stránky na provedenou událost označme jako očekávané chování. Příkladem může být událost kliknutí na odkaz, jejíž očekávaným chováním je přesměrování na jinou stránku. Každý interaktivní element má svoje očekávané chování a jedním z cílů testování webových rozšíření je zjistit, jak tato rozšíření ovlivňují chování a jestli chování s instalovaným rozšířením odpovídá očekávanému chování.

Aby bylo možné aktuální chování elementu nebo celé webové stránky porovnat s očekávaným chováním, je potřeba nejdříve očekávané chování získat, rozpoznat a alespoň dočasně uložit. Tato sekce se zabývá právě otázkou, jak automaticky zjistit očekávané chování webové stránky a konkrétních elementů.

Rozsah očekávaného chování

Očekávané chování lze zjišťovat u webové stránky jako celku i u jednotlivých interaktivních elementů (<button>, <input>, <select>, <a> a další).

Pokud se pohybujeme v rozsahu celé webové stránky, tak základním očekávaným chováním je, že při HTTP GET požadavku na webovou stránku dojde k získání potřebných zdrojových souborů, aby bylo možné webovou stránku vykreslit následně v prohlížeči.

Na úrovni elementů je událostmi, které by měly vyvolat očekávané chování elementu, chápána běžná interakce uživatele s webovou stránkou, jako je kliknutí myši, vepsání textu do elementu pomocí klávesnice nebo odeslání formuláře klávesou *Enter*. Na úrovni elementů platí pro očekávané chování určité zažité zvyklosti, které pomáhají uživateli v interakci

s webovou stránkou. Kliknutí na tlačítko pod formulářem typicky vyvolá uložení nebo odeslání vyplněného formuláře, najetí myší nad ikonu s otazníkem typicky zobrazí nápovědu a kliknutí na šipku směrem dolů u pole ve formuláři typicky rozbalí možnosti, ze kterých uživatel může vybírat. Nejen na těchto příkladech lze vidět, že očekávané chování na úrovni elementů je do velké míry dáno zvyklostmi v grafických uživatelských rozhraních.

Ani u jednotlivých elementů, ani u webové stránky jako celku se však nelze spolehnout pouze na ustálené zvyklosti v očekávaném chování a rozpoznávat očekávané chování jenom na základě ustálených pravidel u daného elementu. Následující odstavce proto popisují dynamické i statické metody pro získání očekávaného chování.

Dynamické získávání očekávaného chování

Dynamické metody jsou založeny na otevření webové stránky v prohlížeči a aktivní interakcí s ní. Označení „dynamické“ vychází z nutnosti stránku otevřít (spustit) a teprve za běhu zjišťovat potřebné informace.

Dynamické zjišťování očekávaného chování je založeno na provedení vybrané události a zachycení změny bez předchozích znalostí o testovaném elementu nebo webové stránce. Provedenou událostí může být například kliknutí na odkaz nebo stisknutí klávesy *Enter*. Pro zachycení nového stavu je možné použít některou z metod:

- pořízení snímku obrazovky,
- uložení DOM,
- uložení webové stránky (HTML souboru a dalších souborů s obsahem stránky).

Všechny tři metody mají společné to, že získáme nějaký obraz stránky po provedení vybrané události. Testování rozšíření pro webové prohlížeče je v případě dynamických metod poté založeno na dvou krocích:

1. Otevření webové stránky bez instalovaného rozšíření, provedení události a zachycení výsledku.
2. Znovuotevření stejné webové stránky s instalovaným rozšíření, provedení stejné akce a porovnání aktuálního stavu se zachyceným výsledkem z předchozího bodu.

Statické získávání očekávaného chování

Dynamické metody při testování vyžadovaly, aby jedna webová stránka byla otevřena dvakrát a dvakrát byla také provedena stejná událost. To znamená až dvojnásobný čas pro otestování jedné webové stránky než u statických metod. Statické metody odstraňují nevýhodu prvního provedení události na webové stránce jenom pro získání jejího výsledku. Snahou je získat výsledek události ještě před provedením samotné události, a to metodou analýzy zdrojového kódu webové stránky, ve kterém by měla být informace o očekávaném chování obsažena.

Získání očekávaného chování statickými metodami směřuje na poměrně náročné zpracování zdrojového kódu webové stránky. Může mít podobu posloupnosti pravidel, která zjišťují, jestli je daná podmínka splněna a pokud ano, vrací očekávanou změnu. Příkladem pro element `<a>` je splnění podmínky, že element obsahuje atribut `href` a není k elementu navázán žádný kód v jazyku JavaScript, který by popisoval chování pro událost `onClick()`. Pokud je taková podmínka splněna, je možné za očekávané chování na událost kliknutí

vybrat přesměrování na webovou stránku, která je v hodnotě elementu `href`. Ne vždy je možné očekávané chování získat pouze z HTML kódu. Může se stát, že chování je popsáno funkcí v jazyku JavaScript, což by vedlo na velké množství možných pravidel nebo velmi náročnou analýzu.

Přes náročnost a omezení statického získávání dat, je možné alespoň v základu kategorizovat očekávané chování daného elementu nebo celé webové stránky. Ukázkový test pak může mít podobu dvou kroků:

1. Zjištění základního očekávaného chování ze zdrojového kódu webové stránky.
2. Provedení události v prohlížeči s instalovaným rozšířením JSR a ověření, že došlo k očekávané akci. Například ověření, že opravdu došlo k přesměrování na jinou webovou stránku, lze zjistit kontrolou, že se změnila adresa URL.

Kapitola 6

Zhodnocení současného stavu a návrh testů

Na začátku kapitoly je nejdříve představen projekt JavaScript Restrictor (zkráceně JSR) a jsou identifikovány problémy při jeho vývoji. Následně jsou navrženy vhodné automatické testy, pomocí kterých by bylo možné identifikované problémy vyřešit. Navržené komplexní řešení automatických testů je ještě dekomponováno na jednotlivé úrovně testování dle teoretických znalostí o problematice testování shrnutých v kapitole 2 a jednotlivé úrovně testování jsou blíže rozebrány v samostatných sekcích včetně návrhu testů na daných úrovních a vytvoření odpovídajících diagramů sloužících k ujasnění a názornému představení návrhu.

6.1 Představení projektu JavaScript Restrictor

JavaScript Restrictor je rozšíření do webových prohlížečů, jehož cílem je dát uživateli možnost kontroly nad přístupem navštíveného webu k aplikačnímu rozhraní (API) webového prohlížeče [83].

Webová stránka s povoleným kódem v jazyku JavaScript může běžně přistupovat k jakémukoliv API, které jí webový prohlížeč poskytne. Uživatel nemusí ani vědět, jaká data webová stránka získává, a také má jen omezené možnosti, jak přístupu webové stránky k API webového prohlížeče zamezit. A právě omezení přístupu k API webového prohlížeče zajišťuje rozšíření JavaScript Restrictor (zkráceně JSR). Toto rozšíření webových prohlížečů funguje jako firewall, který filtruje příchozí požadavky na poskytnutí dat a dokáže přístupu zamezit nebo v odpovědi na požadavek podvrhnout falešná data [64].

Aktuálně neexistuje systematický přístup k testování rozšíření JSR. Výsledky provedených změn jsou ověřovány manuálně (vizuální kontrolou hodnot) na testovací stránce projektu[1]. Chyby jsou zjišťovány z hlášení od uživatelů nebo na základě používání rozšíření samotnými vývojáři (ve stylu α-testování).

Při manuálním testování JSR na testovací stránce projektu můžeme ověřit, jaká data poskytuje aktuální sestavení JSR ve zvoleném prohlížeči testovací webové stránce při volání vybraných koncových bodů API. Jak však ovlivňuje JSR celkový vzhled a chování ostatních webových stránek? Neomezuje chtěnou funkcionalitu? Jak se chová JSR v dalších prohlížečích? To jsou otázky, které aktuálně není možné jednoduše odpovědět nebo je lze odpovědět jen v omezené míře, která je dána rozsahem možností manuálního testování.

[1] https://polcak.github.io/jsrestrictor/test/test.html

Při aktuálních možnostech manuálního testování je také problematický pokračující vývoj, který probíhá už v takovém rozsahu, že na něm spolupracuje tým diplomantů. Při schvalování žádosti o změny v kódu rozšíření se můžeme ptát: Nezpůsobila změna v kódu narušení v podvrhování dat? Nebo neovlivnila negativně chtěnou funkcionalitu webových stránek? Tyto otázky patří společně s těmi v předcházejícím odstavci k otázkám, které je problematické nebo dokonce nemožné zodpovědět, přestože se jedná o důležité otázky, jejichž odpovědi jsou před publikací nové verze JSR klíčové.

Poslední kategorií, která může působit potíže, je aktualizace samotných webových prohlížečů a vydávání nových verzí webových stránek. Po roce od vydání první verze JSR vyvstaly otázky: Funguje stále JSR správně i v nejnovější verzi webového prohlížeče? Jestliže před rokem JSR neomezoval chtěnou funkcionalitu vybrané webové stránky, je tomu tak stále i po jejím přechodu na novější verzi?

Zmíněné otázky ohledně ověřování funkčnosti, které vyvstávají při vývoji i udržování JSR, jsem se pokusil shrnout do několika bodů. V aktuálním přístupu k vývoji JSR chybí způsob, jak automaticky ověřit, že:

- JSR obaluje vybrané koncové body API různých webových prohlížečů a podvrhuje správná data vzhledem k aktuálnímu nastavení JSR.
- JSR neomezuje chtěnou funkcionalitu webových stránek.
- Změny ve zdrojovém kódu JSR nezpůsobují chybné podvrhování dat nebo neomezují chtěnou funkcionalitu webových stránek (tzn.: při vývoji nedošlo k zanesení nové významné chyby nebo se neprojevila již existující závažná chyba).
- Po vydání aktualizace webového prohlížeče nedošlo k narušení funkčnosti JSR.
- Po publikování nové verze webové stránky nedochází k omezení chtěné funkcionality této webové stránky.

Řešením pro automatické ověřování správné funkčnosti rozšíření JSR a zachování chtěné funkcionality webových stránek je vytvoření sady automatických testů, které na různých úrovních budou ověřovat požadavky na JavaScript Restrictor.

6.2 Úrovně testování

Komplexní problém otestování rozšíření JSR je vhodné rozdělit do tří úrovní testování: jednotkové, integrační a systémové testy. Každá úroveň testování by měla být spustitelná samostatně a nezávisle na testech nebo podpůrných nástrojích z ostatních úrovní. To však nevylučuje, že úrovně testování mohou sdílet společný kód. Už při návrhu je jisté, že ke zdrojovému kódu JSR budou přistupovat všechny tři druhy testů. I přes sdílení společného kódu je možné zajistit samostatnou spustitelnost pouze jedné úrovně testování.

Nezávislost jednotlivých úrovní testování také umožní využití různých technologií pro každou úroveň. Je možné předpokládat, že každá úroveň testování bude vyžadovat jiné technologie vhodné právě jen pro danou úroveň, a proto má smysl již v návrhu počítat s nutností vzájemné nezávislosti jednotlivých úrovní testování. Výběr vhodných technologií je však záležitostí až následující kapitoly 7, která se zabývá samotnou implementací.

Každá úroveň testování musí mít vymezenou svoji odpovědnost – co má ověřit a co už není předmětem jejích testů. Odpovědnosti jednotlivých úrovní jsou podrobně definovány v samostatných sekcích této kapitoly v popisu dané úrovně testování.

Bylo zmíněno, že jednotlivé úrovně testování nesou odpovědnost za ověření určitých vlastností nebo požadavků. Dle teorie o testování však nelze pomocí testů nikdy ověřit, že výsledný program neobsahuje chybu a že daný požadavek je tedy bez výjimky splněn. Pro kompletní ověření zdrojového kódu vůči nějakému požadavku slouží metody formální verifikace. Proto slovo „ověření“ používané při návrhu testů bude mít význam spíše „otestování“, neboli maximální možné ověření proveditelné prostřednictvím automatických testů.

Diagram v příloze na obrázku A.1 zachycuje návrh balíčků, do kterých lze testy rozdělit, a vazby balíčků na zdrojový kód rozšíření JSR. Za povšimnutí stojí, že mezi jednotlivými balíčky testů (úrovněmi testování) neexistuje žádná závislost, jak bylo diskutováno v předchozích odstavcích. Jednotkové testy mají přímou závislost na zdrojový kód rozšíření JSR, což je dáno tím, že jednotkové testy přistupují typicky přímo ke zdrojovému kódu a přímo zkoušejí volat jednotlivé funkce. V návrhu integračních a systémových testů se nepočítá s přímou interakcí se zdrojovým kódem. Prostředníkem mezi těmito testy a rozšířením JSR by byly skripty obsažené v `common files`, které by zajistily sestavení rozšíření JSR ze zdrojového kódu a případné nastavení testovacího prostředí. Samotné integrační a systémové testy by pracovaly už pouze s hotovým balíčkem JSR (např. archivem `zip`), nikoliv se zdrojovým kódem.

6.3 Jednotkové testy

Zodpovědnost: *Ověření, že jednotlivé funkce jsou definovány a že vrací očekávanou hodnotu na zadaný vstup (včetně případného vyvolání výjimky, pokud vstup nepatří do žádného intervalu možných vstupních hodnot).*

Při návrhu automatických testů dává smysl začít od nejnižší úrovně testování – od jednotkových testů. Pokud je totiž chyba v základní funkcionalitě dané funkce, je možné ji pomocí jednotkových testů odhalit ještě před samotným sestavením JSR a přesně chybu ve zdrojovém kódu lokalizovat, protože jednotkové testy vždy testují jenom elementární části zdrojového kódu.

Cílem jednotkového testování bude takzvaně pokrýt kód jednotkovými testy. Kompletní pokrytí kódu jednotkovými testy znamená vytvořit testy ověřující správnou funkčnost každé definované funkce. Ověření správné funkčnosti je poté založeno zejména na opakovaném volání testované funkce s různými testovacími vstupními hodnotami a kontrolování výstupních hodnot. Každá funkce má určitou množinu či intervaly očekávaných vstupních hodnot. Také existují hodnoty, které leží mimo tuto množinu a intervaly, a takové hodnoty budou označovány jako nepovolené. Pro sjednocení označení vstupních hodnot bude dále používán pojem „intervaly vstupních hodnot“, které budou označovat všechny množiny a intervaly s očekávanými (tj. povolenými) vstupními hodnotami.

Za účelem pokrytí zdrojového kódu JSR by pro každou funkci měla vzniknout sada jednotkových testů, která bude ověřovat následující:

- Funkce je definována.
- Funkce vrací očekávaný datový typ (např.: číslo, textový řetězec, objekt).
- Funkce vyvolá výjimku nebo vrátí hodnotu s významem chyby pro nepovolenou vstupní hodnotu (Hodnoty mimo intervaly možných vstupních hodnot. U každé funkce jiné. Např.: nedefinovaný vstup, nesmyslná vstupní hodnota, příliš vysoké/nízké číslo).

- Funkce vrací očekávané hodnoty pro:
  - běžné vstupní hodnoty (hodnoty z vnitřku intervalů možných vstupních hodnot),
  - krajní hodnoty (hodnoty z krajů intervalů možných vstupních hodnot).

Jelikož ne ke všem funkcím existuje detailní dokumentace popisující intervaly možných vstupních hodnoty a očekávané návratové hodnoty, bude třeba tyto informace případně vyčíst a odvodit ze zdrojového kódu. Očekávané návratové hodnoty by mělo být možné získat pomocí názvu funkce, který by měl být dostatečně popisný, aby v základu uváděl, co funkce provádí a co vrací. Intervaly povolených vstupních hodnot by mělo být možné odhadnout z míst ve zdrojovém kódu, odkud je testovaná funkce volána. Z těchto míst ve zdrojovém kódu by mělo být možné vyčíst, jaké hodnoty jsou funkci předávány a jaké mohou být intervaly povolených vstupních hodnot. V obou případech mohou být velkým vodítkem názvy proměnných, ať už názvy předávaných parametrů nebo název proměnné, která je z funkce vracena. A samozřejmě komentáře, sloužící částečně jako dokumentace, dokáží poskytnou podrobnější informace o vstupních a návratových hodnotách funkce, pokud tyto informace obsahují.

Diagram na obrázku A.2 graficky znázorňuje návrh struktury jednotkových testů. Cílem diagramu je zobrazit způsob rozdělení jednotkových testů do komponent a vazby těchto komponent na zdrojový kód rozšíření JSR. V diagramu není obsažena celá struktura jednotkových testů, ale pouze ukázka, jakým způsobem bude přistupováno k následné implementaci. Bylo by neúčelné, aby diagram obsahoval všechny komponenty jednotkových testů z důvodu velikosti a neustálého opakování podobné struktury testů v každé komponentě, nehledě na fakt, že JSR se stále vyvíjí, a proto by diagram mohl rychle zastarávat, protože přibývají komponenty nové a ty stávající se mění.

Z diagramu v příloze na obrázku A.2 lze vyčíst, že každému zdrojovému souboru JSR odpovídá komponenta (soubor) obsahující testy k dané zdrojové komponentě. Pro dvě komponenty (`helpers_tests` a `url_tests.js`) je zpracován detailnější pohled, který obsahuje jednotlivé elementární testy zachycené v pseudokódu podobném jazyku JavaScript. Testy v jedné komponentě vychází z výčtu bodů na začátku této sekce, ve kterých bylo shrnuto, co by měly jednotkové testy otestovat pro každou funkci (je definována, vrací očekávaný datový typ, ...).

Hlavní komponentou v balíčku jednotkových testů je spouštěč (komponenta `runner`), který má přístup ke všem ostatním komponentám jednotkových testů a zajišťuje spuštění a provádění jednotkového testování. Jeho úkolem je také zobrazení výsledků testování.

6.4 Integrační testy

Zodpovědnost: *Ověření, že rozšíření JSR lze sestavit do balíčku schopného importu do testovaného webového prohlížeče. Následně ověření, že JSR dle aktuálního nastavení správně podvrhne falešné hodnoty nebo nebrání poskytnutí reálných hodnot při volání vybraných koncových bodů API prohlížeče.*

Stěžejním bodem automatického testování budou integrační testy, které by se při spuštění pokusily nejdříve sestavit nejaktuálnější verzi JSR ze zdrojového kódu a tím by vyzkoušely, zda lze ze zdrojových kódů JSR vůbec vytvořit publikovatelný balíček rozšíření do webových prohlížečů. Pokud by sestavení balíčků JSR pro testované prohlížeče proběhlo úspěšně, následovalo by spuštění samotných integračních testů, které by ověřily, zda

ve vybraných prohlížečích dochází k obalení žádaných koncových bodů API a tím pádem k podvrhovaní dat dle aktuálního nastavení.

Smyslem integračních testů je nahradit manuální kontrolu dosud prováděnou při sestavení nové verze JSR. Manuální kontrola spočívá v otevření testovací stránky projektu a vizuální kontrole hodnot vypsaných na stránce. Vypsané hodnoty jsou manuálně porovnávány vůči očekávaným hodnotám uvedených v dokumentaci[2].

Cílem integračních testů není vyzkoušet, že rozšíření JSR funguje správně na všech webových stránkách. Cílem je zkusit, že je možné rozšíření nainstalovat (integrovat) do prohlížeče a že JSR poskytuje očekávané hodnoty (že integrace proběhla v pořádku a JSR se na testovací stránce chová dle očekávání).

Při návrhu stojí za zvážení, zda jednotlivé integrační testy neprovádět paralelně. Pokud ale uvážíme celkovou dobu běhu integračních testů, kterou je možné dle dosavadních experimentů odhadnout na maximálně několik minut, lze vyvodit, že paralelní provádění testů by nepřineslo výraznější zrychlení, a naopak by to neúměrně zvýšilo režii integračního testování.

Vývojový diagram v příloze na obrázku A.3 zachycuje návrh procesu automatického integračního testování. Ten se skládá ze tří vnořených cyklů. Cyklus na nejvyšší úrovni iteruje přes všechny testované prohlížeče. Cyklus na druhé úrovni iteruje přes všechny výchozí úrovně v nastavení JSR (každá výchozí úroveň má jiné očekávané hodnoty). Poslední cyklus – na třetí úrovni – načítá z prohlížeče hodnoty testovaných proměnných a porovnává je s očekávanými hodnotami (tedy iteruje přes testované hodnoty). Očekávanou hodnotou může být buď předem daná falešná hodnota (např. počet logický procesorů = 2) nebo reálná hodnota, pokud v aktuálně testované úrovni JSR není u této hodnoty vyžadováno její podvrhnutí falešnou hodnotou. Případně může očekávanou hodnotou být také skutečná číselná hodnota zaokrouhlená na vyžadovanou přesnost (tento speciální způsob podvrhnutí dat se využívá zejména u lokalizačních údajů a u času). Integrační testy tak zjistí nejen to, že hodnoty, které mají být podvrženy, jsou opravdu podvržené, ale také zjistí, zda hodnoty, které nemají být podvržené, odpovídají skutečným hodnotám nebo případně splňují vyžadovanou přesnost.

6.5 Systémové testy

Zodpovědnost: *Ověření, že rozšíření JSR neomezuje chtěnou funkcionalitu webových stránek.*

Systémové testy obecně ověřují, zda výsledný program funguje jako celek. V případě JSR je v návrhu počítáno s tím, že funkčnost rozšíření jako celku (tj. podvrhnutí falešných hodnot dle aktuálního nastavení) ověří integrační testy, jak bylo představeno v předchozí sekci. Dosud však v návrhu neexistují testy, které by zjišťovaly, zda rozšíření korektně interaguje s různými webovými stránkami. Po provedení integračních testů budeme moci říci, že vybrané funkce se obalují, ale stále bude otázkou, jak se budou chovat webové stránky ovlivněné rozšířením JSR po jeho instalaci. A právě ověření, zda nedochází k omezení chtěné funkcionality webových stránek bude zodpovědností systémových testů.

Systémové testy by měly do velké míry simulovat chování uživatelů při prohlížení webu a přímo interagovat s webovými stránkami, když bude rozšíření JSR aktivní. Na základě

[2] https://polcak.github.io/jsrestrictor/levels.html

různých ukazatelů by testy sbíraly zpětnou vazbu z prohlížení s cílem identifikovat potlačení chtěné funkcionality.

Výběr vzorku webu k testování

Vzhledem k velikosti webu je nutné vzít pouze jeho určitý vzorek a nad ním systémové testy spouštět. Jako vzorek by mohly dobře posloužit nejnavštěvovanější webové stránky. Problematika získávání žebříčků nejnavštěvovanějších webových stránek byla představena v kapitole 4 a poznatky, které v ní byly shrnuté, poslouží pro získání vzorku webu pro systémové testy. Nejnavštěvovanější webové stránky lze vnímat jako dostatečně reprezentativní vzorek webu z několika důvodů:

- Uživatel s nainstalovaným doplněk JSR s největší pravděpodobností navštíví některé z nejnavštěvovanějších webových stránek a bude očekávat jejich správnou funkčnost i s nainstalovaným rozšířením JSR.
- Nejnavštěvovanější webové stránky jsou typicky aktivně vyvíjeny, dlouhodobě udržovány a pravidelně aktualizovány. Je tedy velká šance, že když se objeví nové technologické řešení, které nebude s JSR kompatibilní, brzy se na těchto webových stránkách vyskytne a systémové testy tak případnou nekompatibilitu odhalí.
- Vzhledem k neustálému vývoji a aktualizaci nejnavštěvovanějších webových stránek lze také očekávat, že nebudou obsahovat zastaralé přístupy, nepodporované technologie, neaktuální verze knihoven apod. Mohlo by se zdát, že právě toto omezuje reprezentativnost vybraného vzorku webu, avšak toto omezení nevadí při úvaze, že cílem JSR není snaha o zachování funkcionality zastaralých technologií. Pokud by při vývoji měly být podporovány například zastaralé verze knihoven jazyka JavaScript, je možné, že by neprošel vyšší počet systémových testů, protože by byly identifikovány a reportovány chyby, která jsou způsobeny spíše zastaralými technologiemi než rozšířením JSR samotným. Podpora zastaralých technologií by tak mohla zvýšit počet falešných hlášení chyb (anglicky *False Positive*). Pokud při systémových testech budeme brát v potaz zejména aktuální a udržované webové stránky, eliminujeme taková falešná hlášení chyb souvisejících se zastaralými technologiemi.

Pro získání reprezentativního vzorku všech webových stránek je také podstatné, kolik nejnavštěvovanějších webových stránek bude tento vzorek obsahovat. Pokud bychom do vzorku zařadili pouze prvních deset webových stránek, jen stěží bychom mohli mluvit o dostatečné reprezentativnosti vybraného vzorku. Na druhou stranu je nutné uvažovat i o časové náročnosti systémového testování. Pokud bychom spustili testy bez paralelního provádění, tak dle vlastního časového odhadu se dá očekávat, že časová složitost bude přibližně lineárně závislá na počtu testovaných webových stránek. Základní otestování jedné webové stránky může trvat přibližně jednu minutu. Pokud bychom tak chtěli otestovat jeden milion webových stránek při sekvenčním provádění testů, trvalo by to téměř dva roky.

Při výběru vhodného počtu nejnavštěvovanějších webových stránek k otestování je nutné brát v potaz dostatečnou reprezentativnost vzorku, ale také celkový čas potřebný k provedení systémových testů. Ideálním počtem webových stránek k otestování se zdají být jednotky tisíců, v případě časově náročnějších systémových testů raději jenom stovky webových stránek, přičemž webovou stránkou je myšlena doména obsahující komplexní webovou prezentaci nebo webovou aplikaci.

Návrh běhu systémových testů

Základem systémových testů bude program procházející nejnavštěvovanější webové stránky (anglické označení *websites*) do šířky. Kostra systémových testů je zachycena na následujících vývojových diagramech. Celé systémové testování je možné rozdělit do dvou fází, které musí proběhnout v tomto pořadí:

1. Získávání dat z testovaných webových stránek (diagramy v příloze na obrázcích A.4 a A.5).

2. Analýza získaných dat (diagramy v příloze na obrázcích A.6 a A.7).

Diagram na obrázku A.4 začíná sestavením rozšíření JSR ze zdrojového kódu. Následují kroky nastavení testovacího prostředí pro nástroj Selenium Grid. Architektura nástroje Selenium Grid je založena na principu klienti (testovací uzly) a server (zajišťuje řízení testování). Nastavení testovacího prostředí je zahájeno spuštěním Selenium serveru a testovacích uzlů. Počet testovacích uzlů je dán konfigurací při spuštění. Následně je voláno získávání dat z testovaných webových stránek (návrh tohoto procesu je zachycen na samostatném diagramu na obrázku A.5). Získávanými daty jsou:

- záznamy v konzoli – textová informace o chybách a varováních při otevření testované webové stránky,

- snímky obrazovky – zachycení vizuální informace o výsledku pokusu načíst testovanou webovou stránku.

Po projití všech webových stránek k otestování a po získání dat z nich je ukončeno testovací prostředí nástroje Selenium Grid. V rámci toho je nejprve postupně zastaven běh testovacích uzlů a následně i Selenium serveru.

Další dva diagramy navazují na fázi získávání dat z testovaných webových stránek a představují návrh procesů pro analýzu získaných dat. Diagram na obrázku A.6 zachycuje návrh procesu pro analýzu záznamů z konzole webového prohlížeče. Cílem této analýzy je identifikovat záznamy z konzole, které se vyskytly pouze v prohlížeči s aktivním rozšířením JSR. U takových záznamů se dá předpokládat, že jejich vznik byl zapříčiněn narušením chtěné funkcionality rozšířením JSR. Pro porovnání záznamů z konzole je v návrhu počítáno s využitím více než jen jedné metody, kvůli zvýšení spolehlivosti identifikace přidaných logů.

Proces analýzy snímků obrazovky získaných v první fázi systémových testů je zobrazen na obrázku A.7 opět ve formě vývojového diagramu. Porovnávání snímků obrazovky získaných bez aktivního a s aktivním rozšířením JSR je založeno na metodě odečtení jednoho snímku od druhého (získání rozdílu snímků).

Paralelní testování v distribuovaném prostředí

Při zvážení počtu testovaných webových stránek a časové náročnosti jednoho testu by vhodným přístupem pro zefektivnění systémových testů mohlo být zavedení paralelního provádění testů. Pokud by paralelní testování bylo spuštěno v distribuovaném prostředí, ve kterém se nachází jeden testovací server řídící testování a několik testovacích klientů provádějících testy, dávalo by takové prostředí možnost spouštět testy na různých zařízeních (stolní počítač, laptop, případně i mobilní telefon apod.) s různými operačními systémy (Windows 8.1, Windows 10, MacOS Catalina, Ubuntu 20.04 LTS, CentOS 8 apod.) s různými webovými prohlížeči (Google Chrome, Mozilla Firefox, Opera apod.) v různých verzích. Pokud by však

všechny testovací uzly byly shodné (stejná platforma se shodnými verzemi prohlížečů), bylo by možné rozdělit testy jednotlivých stránek mezi všechny testovací uzly a tím snížit čas nutný k provedení všech testů. Paralelní distribuované testování dokáže tedy naplnit jeden z následujících cílů:

- Zkrácení celkového času potřebného k provedení testů tím, že celková práce je rozdělena mezi několik stejných testovacích uzlů a každý uzel za paralelního běhu vykoná přiřazené testy.
- Souběžné otestování JSR na různých zařízeních, na různých operačních systémech a v různých verzích webových prohlížečů. Na každém testovacím uzlu tak proběhnou všechny testy. Tím je dosaženo otestováních v různých prostředích, ale nedojde k urychlení systémového testování.

Vzhledem k očekávané časové náročnosti systémového testování by bylo prvořadým cílem zkrácení celkového času potřebného k provedení testů a tím zrychlení celého systémového testování. Ideálním prostředím pro naplnění takového cíle by byla síť totožných počítačů, ve které by jeden počítač měl roli serveru a ostatní roli klienta. Pokud by nebyla k dispozici taková síť, stačí pro urychlení testování spustit paralelně na jednom zařízení testovací server i všechny uzly provádějící samotné testy. Při spuštění na zařízení s více-jádrovým procesorem anebo s více procesory dojde také ke zkrácení celkového času testování jako v případě distribuovaného prostředí, i když v případě vyššího počtu klientů nebude časová úspora tak výrazná z důvodu dosažení hranice maximálního výkonu jader procesorů.

Systémové testování je rozděleno na dvě fáze: získávání dat (záznamy z konzole a snímky obrazovky) a analýza dat. Dle prvotních odhadů je fáze získávání dat časově výrazně náročnější než fáze druhá. Z toho důvodu by dávalo smysl provádět paralelně pouze první fázi (získávání dat) a po jejím dokončení provést analýzu získaných dat už pouze sekvenčně.

Pro koordinaci paralelního získávání dat je nutná komunikace mezi kontroléry, testovacími uzly a Selenium serverem. Diagramy v příloze na obrázcích A.8 a A.9 zachycují komunikaci v rámci této první fáze systémových testů. Tyto dva diagramy sekvence nabízejí jiný pohled na návrh procesu získávání dat, který je zachycen na dříve představených vývojových diagramech na obrázcích A.4 a A.5.

Kapitola 7

Implementace

V této kapitole je popsána implementace testů dle návrhu z předchozí kapitoly 6. Struktura kapitoly kopíruje rozdělení testů při návrhu na tři úrovně a implementace každé úrovně testování je detailně rozebrána v samostatné sekci. V každé sekci jsou uvedeny technologie použité při implementaci včetně zdůvodnění jejich výběru. Představeny jsou použité přístupy k řešení problémů. Zmíněné jsou také komplikace, které se vyskytly při implementaci, a jejich řešení, které bylo zvoleno.

Při implementaci vznikl nejen samotný kód, ale také další podpůrné soubory pro uživatele jako například návody pro spuštění, které jsou přiloženy k této technické zprávě jako přílohy a jsou odkazovány z této kapitoly z jednotlivých sekcí, ke kterým náleží.

7.1 Jednotkové testy

Výběr vhodného rámce

Před samotným zahájením implementace jednotkových testů bylo nutné vybrat vhodný rámec pro jednotkové testování. Protože jednotkové testy přistupují přímo ke zdrojovému kódu programu (k jednotlivým funkcím a proměnným), je žádoucí a do velké míry i nutné, aby jednotkové testy byly psány ve stejném programovacím jazyku jako samotné rozšíření JSR – tedy v jazyku JavaScript. Rámců pro jednotkové testy v jazyku JavaScript existuje celá řada. Několik vybraných, které by mohly být vhodné pro testování JSR, bylo představeno v kapitole 3. Na základě předchozí analýzy rámců byl pro jednotkové testování JSR vybrán rámec Jasmine.

Pro výběr rámce Jasmine existuje několik důvodů:

- Jedná se o minimalistický (anglické označení *lightweight*) rámec, který poskytuje opravdu jenom základní kostru a knihovny pro testování, což pro účely JSR naprosto dostačuje.
- Zdrojové kódy programu i testů se načítají jednotným způsobem, který je totožný s načítáním skriptů do webové stránky pomocí značky `<script>`. To umožňuje jednoduše řídit postupné načítání skriptů, vybírat, které skripty se mají importovat a které ne a případně lze také velice jednoduše přidat vlastní kód v jazyku JavaScript, který nemusí být ani součástí rozšíření ani součástí jednotkových testů.
- Rámec Jasmine poskytuje přehledný a interaktivní výpis výsledků testů. Výstup se zobrazuje v prohlížeči jako webová stránka, což umožňuje detailní grafické rozlišení

úspěšných a neúspěšných testů, zobrazování struktury testů pomocí vnořených seznamů a také vkládání odkazů, které vedou z celkového přehledu na jednotlivé testy a u neúspěšných testů zobrazují detailní informace, kde došlo k chybě.

- Vzhledem k tomu, že testy jsou prováděny v prohlížeči, je snadné případné odhalování chyb v testech pomocí obdobných nástrojů, které se používají při samotném vývoji rozšíření – např. konzole pro vývojáře, kontrolní výpisy v podobě záznamů v konzoli nebo kontrolní výpisy přímo do webové stránky. To znamená, že v budoucnu na vývojáře není kladen požadavek, aby se naučili pracovat s dalšími technologiemi, protože si vystačí pouze s těmi, které již používají při vývoji JSR. To může vést k ušetření času a rychlejšímu vývoji.

Jasmine je vydáván jako balíček pro různá prostředí: Node.js, Ruby, Python. Zároveň je však vydáván ještě jako samostatná verze (anglicky *standalone*) schopná fungovat bez podpůrného prostředí. Pro implementaci jednotkových testů jsem zvolil právě samostatnou verzi. Implementace v jazycích Python nebo Ruby s využitím odpovídajících balíčků nedávala smysl vzhledem k cíli zachovat stejný jazyk pro vývoj JSR a psaní testů. Využití balíčku Node.js by sice vedlo k používání jazyku JavaScript, který je také využíván při vývoji JSR, avšak použití Node.js by dávalo smysl, pouze pokud by i samotné rozšíření JSR bylo vyvíjeno jako Node.js projekt, což není a využití Node.js pro jednotkové testy by působilo problémy při importu zdrojových kódů JSR do testů. Vybranými technologiemi pro implementaci jednotkových testů se tak stal rámec Jasmine Standalone verze 3.5.0 v kombinaci s programovacím jazykem JavaScript.

Emulace `browser.storage`

Při zahájení tvorby jednotkových testů se objevil problém, který je specifický pouze pro testy rozšíření webových prohlížečů. Tímto problémem je nemožnost v testech přistupovat k API úložiště `browser.storage` (respektive `chrome.storage` v prohlížeči Google Chrome).

Úložiště `browser.storage` je dostupné pouze sestavenému rozšíření, které je nainstalováno v prohlížeči. Vzhledem k tomu, že jednotkové testy probíhají před samotným sestavením rozšíření a jeho instalací do prohlížeče, je toto úložiště při testování nedefinované (chyba `undefined`). Pokud se tedy například sada testů `levels_tests.js` pokusí načíst a přistoupit k odpovídajícímu testovanému skriptu `levels.js`, dojde k chybě `undefined`, protože neexistuje `browser.storage`, který je ve skriptu `level.js` využíván.

Řešením by bylo vytvořit vlastní úložiště s identickým API, které by nahradilo skutečné úložiště `browser.storage`. API je podrobně popsáno v dokumentaci prohlížečů [10, 29], takže by bylo možné odhadnout základní vnitřní funkcionalitu a tu poté implementovat. Jelikož se však jedná o typický problém při jednotkovém testování rozšíření do webových prohlížečů, existují už i implementované nástroje, které dokáží úložiště `browser.storage` nahradit. Níže jsou zmíněny tři různé nástroje, které dokáží zajistit emulaci úložiště prohlížeče při testování:

- Sinon-chrome [84]
- Mock Browser [87]
- WebExtensions API Fake [82]

Nástroje *Mock Browser* a *WebExtensions API Fake* jsou distribuovány pouze jako balíčky pro Node.js. Nástroj *Sinon-chrome* je také distribuován jako balíček pro Node.js,

ale zároveň je možné stáhnout jeho samostatnou (anglicky *standalone*) edici. Všechny tři zmíněné nástroje poskytují srovnatelnou funkcionalitu, takže rozhodujícím faktorem byla možnost stažení samostatné edice vzhledem k tomu, že při implementaci bylo preferováno vytvoření řešení nezávislého na Node.js. Pro emulaci úložiště `browser.storage` při implementaci jednotkových testů byl vybrán nástroj *Sinon-chrome* v samostatné edici.

7.2 Integrační testy

Výběr vhodných technologií

Z návrhu integračních testů vyplývá, že při implementaci je nutné zajistit přímou interakci s webovými prohlížeči. Nejrozšířenějším prostředkem pro automatizaci webových prohlížečů je dříve představená sada nástrojů Selenium. Ta je aktuálně dostupná ve třech různých edicích: IDE, WebDriver a Grid. Selenium IDE je relativně omezený nástroj, který nedokáže nabídnout všechnu potřebnou funkcionalitu pro navržené integrační testy. Selenium Grid sice obsahuje vše potřebné, ale vzhledem k tomu, že u integračních testů není požadavek na paralelní provádění testů, je zbytečné využívat takto robustní nástroj. Jako vhodné řešení pro automatizaci webových prohlížečů pro účely integrační testování byl vybrán Selenium WebDriver, který nabízí potřebnou funkcionalitu, a navíc umožňuje výbornou spolupráci s testovacími rámci.

Další otázkou, po výběru nástroje pro automatizaci webových prohlížečů, je výběr programovacího jazyka. Pro vybraný programovací jazyk musí být k dispozici knihovny nástroje Selenium WebDriver. Selenium WebDriver však podporuje celou řadu programovacích jazyků – např.: Java, C#, Ruby, Python, JavaScript, Perl, PHP. Pro první pokusy byl vybrán implementační jazyk JavaScript, jelikož se jedná o programovací jazyk použitý při vývoji rozšíření JSR a shodovaly by se tak programovací jazyky pro vývoj rozšíření a psaní testů. Po prvním experimentech jsem však JavaScript opustil z několika důvodů:

- Selenium WebDriver pro jazyk JavaScript je velmi úzce spojen s Node.js. Jelikož rozšíření JSR není vyvíjeno jako Node.js projekt, nedávalo smysl vytvářet samostatný Node.js projekt pouze pro testy.
- Ne všechna funkcionalita je nástrojem Selenium WebDriver podporována v každém programovacím jazyku. Při použití programovacího jazyku JavaScript je velmi komplikovaná kontrola nad webových prohlížečem – například instalace rozšíření do prohlížeče nebo nastavení profilu webového prohlížeče uloženého v lokálním úložišti zařízení, na kterém jsou testy spouštěny.
- Podpora pro Selenium WebDriver v různých programovacích jazycích není srovnatelná. Zejména na fórech převažují rady pro jazyky Java a Python. Ostatní jazyky jsou spíše v minoritním zastoupení.

Na základě zkušeností s kombinací Selenium WebDriver a jazyku JavaScript jsem se rozhodl zvolit jiný programovací jazyk. Vybraným implementačním jazykem se stal Python, u kterého jsou vhodně vyřešeny problematické body zjištěné u jazyku JavaScript.

Poslední technologií k výběru byl testovací rámec pro vybraný programovací jazyk Python. Pro integrační testy byl vybrán rámec pytest zejména z důvodu širokých možností pro *SetUp* a *TearDown* metody na různých úrovní. Pytest umožňuje definovat zmíněné metody na úrovni celého testování, jednotlivých modulů (tj. skupiny souvisejících testů) nebo na úrovni jednotlivých testů. Výslednou kombinací vybraných technologií se staly:

- Programovací jazyk: Python
- Nástroj pro automatizaci webových prohlížečů: Selenium WebDriver
- Testovací rámec: pytest

Definování očekávaných hodnot

Základem integračních testů je porovnávání očekávaných hodnot s hodnotami získaných z prohlížeče při různých úrovních JSR. Hodnoty z prohlížeče jsou získávány přes příslušné JavaScript API. Očekávané hodnoty jsou definovány testerem na základě specifikace z oficiální dokumentace[1] rozšíření JSR [64]. Určení očekávané hodnoty však není triviální záležitostí, protože je třeba uvažovat, že očekávaná hodnota nemusí být jenom předem dané číslo (např. `navigator.hardwareConcurrency: 2`), ale může to být také reálná hodnota s určitou přesností (např. `Date` a `performance` zaokrouhleny na desetiny sekund) nebo různá hodnota pro různé webové prohlížeče (např. `navigator.deviceMemory: 4` pro Google Chrome a `navigator.deviceMemory: None` pro Mozilla Firefox).

Očekávané hodnoty jsou definovány v souboru *values_expected.py*. Pro každou výchozí úroveň JSR (*Level 0*, *Level 1*, *Level 2* a *Level 3*) je v tomto souboru vytvořen objekt ze třídy `TestedValues`, která určuje, jaké všechny hodnoty se testují. Pokud je třeba rozšířit testování o další hodnoty, je třeba je přidat jako vlastnost do této třídy. Při vytvoření objektů v souboru *values_expected.py* jsou definovány hodnoty vlastností. Vlastnosti mohou nabývat následujících hodnot:

- `'REAL VALUE'` – pokud je očekávána skutečná hodnota (tj. taková hodnota, která by byla získána bez aktivního rozšíření JSR).
- `{'value': 'REAL VALUE','accuracy': 0.1}` – pokud je očekávána skutečná hodnota s určitou přesností. Přesnost může být libovolné číslo typu `float`, doporučuje se však používat pouze násobky deseti z důvodu přehlednosti. Definovaná přesnost je maximální povolená přesnost. Pokud by získaná hodnota byla přesnější, test skončí chybou.

  Speciálním případem je dvojice `{'value': 'REAL VALUE','accuracy': 'EXACTLY'}` využívaná zejména pro očekávané hodnoty při JSR na úrovni 0. Taková kombinace znamená, že není povoleno zaokrouhlování hodnoty. Testy jsou nastaveny tak, aby dokázaly identifikovat případné zaokrouhlení a v takovém případě by test skončil chybou. Přesnost `'EXACTLY'` však neznamená úplnou rovnost získané a očekávané hodnoty. Při testu je počítáno například s tím, že pro čas nastane odchylka mezi očekávanou hodnotou a skutečnou hodnotou, která je dána dobou běhu příslušných příkazů. Dvojice `{'value': 'REAL VALUE','accuracy': 'EXACTLY'}` tak neznamená přesnou rovnost získané a očekávané hodnoty, ale pouze vyžaduje, že získaná hodnota nesmí být zaokrouhlena. Že nedošlo k zaokrouhlení je zjištěno v cyklu, který získává opakovaně hodnoty té samé vlastnosti a každou hodnotu kontroluje, že je dostatečně přesná (např. na pozici jednotek není nula). Jakmile je nalezena první hodnota splňující přesnost, test prošel. Pokud po předem daném počtu opakování není nalezena ani jedna přesná hodnota, test selže.
- konstanta – nejpřímějším způsobem definování očekávané hodnoty je udání její přesné konstantní hodnoty elementárního datového typu. Příkladem může být testovaná

[1] https://polcak.github.io/jsrestrictor/levels.html

vlastnost `'hardware_concurrency': 2`, jejíž očekávaná hodnota je v tomto případě nastavena na celočíselnou konstantu 2. Konstantou může být libovolné číslo typu `int`, pravdivostní hodnota typu `boolean`, znak typu `char` nebo textový řetězec typu `char[]`. Při testech je taková hodnota porovnávána na přesnou shodu se získanou hodnotou. Proto nejsou jako očekávané hodnoty podporovány číselné konstanty typu `float`, protože porovnání na přesnou shodu je u nich problematické. Pokud by bylo třeba porovnávat desetinné číslo, je vhodné ho porovnat jako textový řetězec.

- `{BrowserType.FIREFOX: None, BrowserType.CHROME: 4}` – poslední možností je definování různé hodnoty pro různé prohlížeče. Hodnotou pro konkrétní prohlížeč může být obecně jakákoliv hodnota z předchozích bodů, ale typicky se jedná o konstantu. Při testování jsou rozlišovány jednotlivé webové prohlížeče a tato možnost umožňuje rozlišit očekávané hodnoty dle prohlížeče. Příkladem použití takové speciální hodnoty je například vlastnost `device_memory`, u níž platí, že v prohlížeči Google Chrome musí mít při podvrhnutí hodnotu 4, zatímco v prohlížeči Mozilla Firefox není definovaná. Právě případ, že hodnota není v nějakém prohlížeči definovaná je typickým případem, kdy použít rozlišení prohlížečů v očekávaných hodnotách.

7.3 Systémové testy

Výběr vhodných technologií

Stejně jako u integračních testů bude základem pro systémové testy výběr nástroje pro automatizaci webového prohlížeče, protože dle návrhu systémových testů je vyžadováno procházení nejnavštěvovanějších webových stránek, interakce s nimi a získávání záznamů z konzole a snímků obrazovky načtených webových stránek. Na rozdíl od integračních testů je však u systémových testů nutné zajistit možnost paralelního provádění testů v distribuovaném prostředí z důvodu zajištění dokončení testů v přijatelném čase a případných dalších důvodů diskutovaných při návrhu systémových testů. Takovým požadavkům vyhovuje z rodiny aktuálně podporovaných nástrojů Selenium pouze Selenium Grid. Ze všech nástrojů rodiny Selenium nabízí Selenium Grid nejpokročilejší možnosti testovacího prostředí. Implementace nastavení testovacího prostředí před samotným spuštěním testů i využívání pokročilých API pro interakci s webovou stránkou vychází z doporučení v knize *Selenium Testing Tools Cookbook* [34].

Selenium Grid stejně jako další nástroje z rodiny Selenium podporuje řadu programovacích jazyků. Na základě zkušeností z implementace integračních testů jsem opět zvolil programovací jazyk Python z důvodu sjednocení programovacího jazyka integračních a systémových testů a také z důvodu, že pro Python existuje celá řada knihoven, a to i knihoven specializovaných na analýzu a porovnávání textu, což může být u systémových textů využito při analýze záznamů z konzole. Také knihovní podpora pro zpracování obrazu je v prostředí Python dostatečně bohatá a je možné využít již implementované metody pro analýzu a porovnávání snímků obrazovky.

Posledním bodem by mohl být ještě výběr vhodného testovacího rámce. Při zvážení charakteru systémových testů však nedává využití testovacího rámce smysl. Systémové testy nemají podobu desítek elementárních testovacích scénářů. Testovanými hodnotami jsou dle návrhu pouze záznamy v konzoly a snímky obrazovky. Aby výsledky systémových testů byly dobře interpretovatelné, je nutné zvolit vlastní výstupní grafický formát v podobě HTML souboru, který je případně i interaktivní. Bylo by náročné upravit předdefinovaný výstup

nějaké testovacího rámce tak, aby odpovídal potřebám výstupu systémových testů. Jednodušší se zdá být kompletní definice vlastního výstupu. Selenium Grid je specifické prostředí a integrace s testovacích rámcem, zejména v případě paralelního testování v distribuovaném prostředí, by mohla být velmi náročná. Při zvážení všech výhod a omezeních testovacím rámcem byl testovací rámec z vybraných technologií pro systémové testy vynechán. Řízení testů a prezentace výsledků testů zůstane čistě ve vlastní režii.

Výslednou kombinací vybraných technologií se staly:

- Programovací jazyk: Python
- Nástroj pro automatizaci webových prohlížečů: Selenium Grid

Metody systémových testů

Výsledkem systémových testů jedné webové stránky by měla být informace, zda a případně jak a jak moc se změnila webová stránka při načtení ve webovém prohlížeči s aktivním rozšířením JSR oproti načtení v prohlížeči bez aktivního rozšíření JSR. Pro zjištění takové informace je nutné najít v první řadě vhodné vlastnosti, které vypovídají o stavu webové stránky a o chybách při jejím načítání, a následně je třeba najít vhodné metody, kterými lze takové vlastnosti porovnávat a určit míru změny.

Jako vhodné vlastnosti pro reprezentování stavu webové stránky a informování o chybách při načítání byly již při návrhu zvoleny záznamy v konzoli (anglicky *logs*) a snímky obrazovky načtené webové stránky. V následujících částech sekce jsou představeny metody pro analýzu a porovnávání daných dat.

Testování záznamů v konzoli

Záznam v konzoli (anglicky *log*) slouží jako textová informace o chybách, varováních nebo jiných upozorněních při otevření webové stránky. Bez nutnosti pohledu na grafickou podobu webové stránky lze ze záznamů v konzoli zjistit, zda došlo k chybám při načítání a o jaké chyby se jedná.

Při systémových testech není ani tak podstatné, zda došlo k nějaké chybě nebo varování při načítání webové stránky, důležité je zejména, jaké chyby a varování se vyskytly pouze při aktivním rozšířením JSR. Základem testování záznamů v konzoli je tedy získat množinu záznamů z prohlížeče bez aktivního rozšíření JSR a množinu záznamů z prohlížeče s aktivním rozšířením JSR pro tu stejnou webovou stránku a tyto dvě množiny následně porovnat. Porovnání takových dvou množin není triviální záležitostí, protože se jedná o porovnávání struktur a v nich obsažených textových informací. Cílem je najít dostatečně odlišné záznamy z konzole, ne pouze takové, kde se změnila třeba jen část adresy URL. Pro takový cíl již nestačí absolutní porovnání dvou textových řetězců, ale je nutné kvantitativně vyjádřit informaci o podobnosti dvou záznamů z konzole a pouze pokud je podobnost dostatečně malá pro všechny záznamy získané bez aktivního rozšíření JSR, je možné označit záznam získaný s aktivním rozšířením JSR jako přidaný. Takový přidaný záznam může signalizovat nechtěné ovlivnění webové stránky rozšířením JSR, což naplňuje celkový cíl systémových testů – zjistit, jak byla webová stránka ovlivněna rozšířením JSR.

Záznam v konzoli je struktura. Pro konkrétní webové prohlížeče se různí složky této struktury. Jako referenční struktura je brán záznamu v konzoli z prohlížeče Mozilla Firefox. Jeho složky jsou níže popsány na základě informací z oficiální dokumentace společnosti Mozilla [11]:

- časové razítko (*timestamp*) – přirozené číslo
- kategorie (*category*) – hodnota z množiny: {Network request, CSS warning/error/log, JavaScript warning/error, Security warning/error, Server logs, Console API messages, Input/Output from the interactive command line interpreter}
- typ (*type*) – hodnota z množiny: {error, warning, informational log}
- zpráva (*message*) – textový řetězec

Pro zjištění, zda mohl být záznam přidán rozšířením JSR, je třeba každý záznam získaný z prohlížeče s rozšířením JSR porovnat s každým záznamem z prohlížeče bez rozšíření JSR, když je v obou otevřena totožná webová stránka. Pokud se pro záznam z prohlížeče s rozšířením JSR najde alespoň jeden záznam z prohlížeče bez rozšíření JSR, který s ním je stejný (nebo velmi podobný), dá se o analyzovaném záznamu z prohlížeče s rozšířením JSR prohlásit, že se nejedná o záznam přidaný rozšířením JSR.

Aby dva záznamy z konzole byly vyhodnoceny jako shodné, musí se přesně shodovat kategorie a typ. Časové razítko při porovnávání nehraje roli. Dále se musí buď shodovat zprávy nebo být alespoň velmi podobné. Pro porovnání zpráv můžeme využít jednu z metod:

- Přesná shoda (rovnost textových řetězců).
- Levenshteinova vzdálenost je vyšší než stanovená úroveň.
- Kosinová podobnost je vyšší než stanovená úroveň.

Přesná shoda

Přesná shoda je triviální metoda, při které se pouze posunuje ukazatel po obou textových řetězcích zároveň a porovnávají se znaky na stejných pozicích. Textové řetězce jsou vyhodnoceny jako shodné, jestliže se u obou řetězců došlo s ukazatelem až na konec a nebyly nalezeny rozdílné znaky na stejných pozicích.

Levenshteinova vzdálenost

Levenshteinova vzdálenost mezi dvěma textovými řetězci je vypočítána jako minimální počet změn znaků (vkládání, mazání, záměna), aby z jednoho řetězce vznikl druhý [54]. Pro využití při porovnávání logů je ještě třeba získané absolutní číslo c_a převést do relativní reprezentace c_r vztahem 7.1.

$$c_r = \frac{c_a}{max(len(string1), len(string2))}, \tag{7.1}$$

kde $len(str)$ je délka textového řetězce str.

Vypočtená hodnota r_c může být z intervalu $< 0, 1 >$, kde 0 značí úplnou shodu textových řetězců a 1 značí naprostou rozdílnost textových řetězců (tj. bylo nutné zaměnit všechny znaky, aby z prvního řetězce vznikl druhý).

Nakonec stačí číslo c_r porovnat s nastaveným prahem c_t a pokud je c_r větší než nastavený práh, znamená to, že záznamy z konzole jsou pravděpodobně dostatečně odlišné na to, aby měly stejnou příčinu vzniku. Dle experimentů se vhodná hodnota prahu pro porovnávání záznamů pohybuje kolem čísla $0,6$.

Kosinová podobnost

Kosinová podobnost vyjadřuje míru podobnosti dvou nenulových vektorů, která se získá výpočtem kosinu úhlu těchto vektorů [76]. Pro tuto metodu je nutné porovnávaný text předzpracovat tak, aby vektory, u kterých se počítá míra podobnosti, reprezentovaly četnost jednotlivých slov.

V rámci předzpracování textového řetězce je nejdříve nutné odstranit všechna interpunkční znaménka (respektive je nahradit mezerou). Pokud by interpunkce v textu zůstala, budou slova „loading“ a „loading.“ identifikována jako dvě rozdílná slova. Ze stejného důvodu je také třeba převést celý textový řetězec pouze na malá písmena (anglicky *lowercase*). Dále je vhodné z textu odstranit frekventovaná slova bez významu (anglicky *stopwords*). Ta jsou specifická pro každý jazyk. Jelikož jsou záznamy z konzole v angličtině, stačí uvažovat pouze anglická slova, do kterých patří například *it*, *this*, *which*. Pokud by se takováto slova neodstranila, může počet dimenzí vektorů výrazně narůst a pravděpodobně se tím zvětší i úhel mezi vektory a tím by se zmenšila podobnost.

Po předzpracování textu přichází na řadu výpočet samotných vektorů. Pro dva porovnávané textové řetězce je třeba vytvořit dva n-dimenzionální vektory, kde n je počet unikátních slov v obou řetězcích. V posledním kroku je třeba získané vektory dosadit do rovnice 7.2, která uvádí matematický výpočet kosinové podobnosti dvou vektorů t a e, které svírají úhel θ [76].

$$cs = similarity(\mathbf{t}, \mathbf{e}) = \cos(\theta) = \frac{\mathbf{te}}{\|\mathbf{t}\|\|\mathbf{e}\|} = \frac{\sum_{i=1}^{n} \mathbf{t}_i \mathbf{e}_i}{\sqrt{\sum_{i=1}^{n} (\mathbf{t}_i)^2} \sqrt{\sum_{i=1}^{n} (\mathbf{e}_i)^2}}, \tag{7.2}$$

kde t_i a e_i jsou složky vektoru t, respektive vektoru e.

Po dokončení výpočtu získáme číslo cs z intervalu $< 0, 1 >$, kde 1 znamená úplnou shodu textových řetězců (vektory jsou totožné) a 0 znamená naprostou odlišnost. Výsledek cs lze opět porovnat s nastaveným prahem cs_t a určit, zda jsou záznamy z konzole dostatečně odlišné. Dle experimentů se vhodná hodnota prahu pro porovnávání záznamů pohybuje opět kolem čísla $0,6$.

Vizuální analýza webových stránek

Snímek obrazovky slouží jako grafická informace o stavu webové stránky po jejím načtení. Tato vizuální reprezentace webové stránky odpovídá tomu, co vnímá uživatel. Proto je porovnání snímků obrazovky tak důležité. Zatímco přidané záznamy v konzoli informují zejména vývojáře o chybách a varováních a pro běžného uživatele nejsou podstatné, dokud se nějak neprojeví přímo na funkčnosti webové stránky, rozdíl snímků obrazovky dokáže přesně informovat, jaký element se nenačetl, jaký element změnit svůj vzhled a jak.

Vstupem algoritmu pro výpočet rozdílového snímku jsou dva snímky obrazovky téže webové stránky – jeden snímek z prohlížeče bez JSR a druhý snímek z prohlížeče s JSR. Oba vstupní snímky jsou reprezentovány jako 2D matice A a B o rozměrech $(m \times n)$, ve které každá pozice odpovídá hodnotě jednoho pixelu. Oba snímky musí mít stejné rozlišení, aby odvozené matice byly stejně velké a mohlo být následně vypočítán jejich rozdíl tak, že jsou od sebe odečteny pixely na stejných indexech $[i, j]$, kde $i, j \in \mathbb{N}_0$. Tím vznikne nová matice $Diff$, která vyjadřuje rozdílový snímek. Výpočet rozdílového snímku popisuje rovnice 7.3.

$$\forall i < m, \forall j < n : Diff[i, j] = A[i, j] - B[i, j] \tag{7.3}$$

Pokud jsou dva pixely $A[i,j]$ a $B[i,j]$ shodné, odpovídající pixel $Diff[i,j]$ v rozdílové matici bude mít nulovou hodnotu, což v odvozeném snímku bude znamenat černou barvu. V opačném případě získáme pixel s nenulovou hodnotou, který vyjadřuje rozdíl jednotlivých barevných složek pixelů.

Odečtením matic získáme novou matici a z ní rozdílový snímek, který podává informaci o tom, jak se vizuálně změnila testovaná webová stránka, když bylo do webového prohlížeče nainstalováno rozšíření JSR. Jako doplnění této metody by bylo užitečné umět vyjádřit rozdíl snímků číslem z určitého rozsahu. Taková hodnota by se mohla použít například pro filtrování rozdílových snímků a zjištění, které webové stránky ze všech otestovaných byly nejvíce ovlivněny rozšířením JSR.

Pro získání rozdílové hodnoty je nutné převést diferenční obrázek z barevného modelu do odstínů šedi. Tím získáme novou matici $Diff'$, ve které každý pixel bude mít hodnotu od 0 do 255. Matice $Diff'$ má opět rozměry $(m \times n)$. Takovou matici lze agregovat do jediné hodnoty tak, že vypočteme průměrnou hodnotu pixelu. Pokud diferenční snímek byl celý černý (nedošlo k žádné změně mezi webovými stránkami), bude mít průměrná hodnota jednoho pixelu rozdílového snímku ve stupních šedi hodnotu 0. Maxima, kterého může průměrná hodnota pixelů rozdílového snímku dosáhnout je 255. Získali jsme tak číselné ohodnocení změny mezi webovými stránkami, které se pohybuje v intervalu $< 0, 255 >$ a které je možné využít pro filtrování. Výpočet průměrné hodnoty pixelu je popsán rovnicí 7.4.

$$\overline{Diff'} = \frac{\sum_{i=0}^{m-1} \sum_{j=0}^{n-1} Diff'[i,j]}{\sum_{i=0}^{m-1} \sum_{j=0}^{n-1} 1} \tag{7.4}$$

Kapitola 8

Výsledky testů

Tato kapitola shrnuje výsledky provedených testů. Je v ní popsáno, na jakém prostředí a s jakou konfigurací testy proběhly. Cílem kapitoly je shrnout úspěšnost testů a upozornit na testovací případy, které skončily chybou.

Výsledky testování jsou specifické pro jednotlivé úrovně. Proto je kapitola rozdělena na tři části, které odpovídají úrovním při návrhu a implementaci testů. Každá úroveň testování je navržena tak, aby dokázala identifikovat chyby spadající do odpovědnosti dané úrovně, čemuž odpovídá i vyhodnocení výsledků.

8.1 Jednotkové testy

Jednotkové testy slouží pro ověření existence jednotek a jejich správných návratových hodnot v případě funkcí, zároveň však slouží částečně jako dokumentace, která udává, jaký výstup je očekáván pro daný vstup funkce. V jednotkových testech se ověřují správné návratové hodnoty na korektní vstupy funkcí, ale také se ověřuje, že pro nesmyslný nebo neočekávaný argument je vyvolána ve funkci výjimku.

Spuštění jednotkových testů je možné v jakémkoliv prohlížeči s podporou skriptů v jazyku JavaScript na jakékoliv platformě pouhým otevřením souboru *SpecRunner.html*. Tabulka 8.1 uvádí, v jakých prostředích byly jednotkové testy spuštěny a ověřeny. Výsledky se shodovaly na různých platformách i v různých prohlížečích. Díky tomu, že je možné spustit jednotkové testy v různých prohlížečích pouze tím, že v nich je otevřen soubor *SpecRunner.html*, je možné jednotkové testy spouštět i v prohlížečích s instalovaným rozšířením JSR, ač to v případě jednotkových testů nedává příliš smysl, protože jednotkové testy by měly být spuštěné ještě před sestavením rozšíření a jeho instalací do prohlížeče. Zároveň je ale možné spouštět jednotkové testy i s různým nastavením prohlížeče a s jinými rozšířeními a sledovat, jaký vliv má nastavení a jiné rozšíření na elementární funkcionalitu JSR.

Následující podsekce uvádí několik vybraných výsledků jednotkových testů. Výsledky testů jsou reprodukovatelné. Jednotkové testy lze spustit otevřením souboru *SpecRunner.html* v libovolném webovém prohlížeči, který podporuje běh skriptů v jazyku JavaScript.

Chyba extrakce domény

Problémem, na který jednotkové testy upozornily a který se již podařilo odstranit, byla chyba související s extrakcí domény z URL ve funkci `extractRootDomain`. Tato funkce zajišťovala extrakci domény před uložením záznamu určujícího specifickou úroveň JSR pro

Operační systém	Webový prohlížeč
Windows 10	Google Chrome (v. 84)
	Mozilla Firefox (v. 68.10.0 ESR)
Ubuntu 20.04 LTS	Google Chrome (v. 84)
	Mozilla Firefox (v. 68.10.0 ESR)
CentOS 7	Google Chrome (v. 84)
	Mozilla Firefox (v. 68.10.0 ESR)

Tabulka 8.1: Prostředí, ve kterých bylo jednotkové testování spuštěno.

danou doménu. Díky této funkcionalitě je možné konkrétní doméně přiřadit odlišnou úroveň JSR (např. úroveň 0), i když ostatní domény mají výchozím nastavením danou jinou úroveň (např. úroveň 2). Funkce `extractRootDomain` vždy extrahovala pouze domény prvního a druhého řádu, což sice odpovídalo jejímu názvu, ale nekorespondovalo to se způsobem použití této funkce při jejím volání z funkce `getCurrentLevelJSON`. Při použití JSR je však potřebné dokázat rozlišit i subdomény. Když je nebylo možné rozlišit, byla nastavena stejná úroveň JSR všem těmto webovým stránkám, protože se shodovaly jejich domény prvního a druhého řádu:

- sites.google.com
- code.google.com
- docs.google.com
- support.google.com
- apod.

Podobných příkladů, kdy subdoména obsahuje kompletně nezávislou webovou prezentaci nebo aplikaci, u které je potřeba nastavovat úroveň JSR samostatně nezávisle na dalších subdoménách, by bylo možné najít více. V tabulce 8.2 je příklad pro další webové stránky včetně zachycení očekávaných a získaných hodnot.

Argument funkce pro extrakci domény	Očekávaná hodnota	Získaná hodnota
"fit.vutbr.cz"	"fit.vutbr.cz"	"vutbr.cz"
"wis.fit.vutbr.cz"	"wis.fit.vutbr.cz"	"vutbr.cz"
"eva.fit.vutbr.cz"	"eva.fit.vutbr.cz"	"vutbr.cz"
"netfox-hyperv.fit.vutbr.cz"	"netfox-hyperv.fit.vutbr.cz"	"vutbr.cz"

Tabulka 8.2: Očekávané a získané hodnoty při testu funkce pro extrakci domény.

Na základě výsledků jednotkových testů byla funkce `extractRootDomain` kompletně nahrazena funkcí `extractSubDomains`, která má podobné chování, jen s tím rozdílem, že místo jednoho řetězce vrací pole textových řetězců, které postupně konkretizují doménové jméno. Při určování úrovně JSR je pak pole procházeno v obráceném směru, takže konkrétní úroveň

je přiřazena na základě toho nejkonkrétnějšího existujícího záznamu. Příkladem návratové hodnoty pro vstup *"wis.fit.vutbr.cz"* je pole *["vutbr.cz","fit.vutbr.cz","wis.fit.vutbr.cz"]*.

Chyba zpracování IPv4 adresy

Podobná chyba se projevuje ve stejné funkci při zpracování webových stránek, které nemají doménové jméno, ale přistupuje se na ně pouze přes (veřejnou) IP adresu. Příkladem takové stránky, či webové aplikace může být `http://89.45.196.133/paneln/Login.aspx`.

Funkce se pro IPv4 adresy zachová stejně jako pro doménová jména a rozdělí adresy podle teček. Problém se týká pouze IPv4 adres, protože v IPv6 adrese se jako oddělovač používá dvojtečka nikoliv tečka.

Spíše, než o chybu ve funkci `extractSubDomains` se jedná o chybné použití této funkce, protože funkce `extractSubDomains` má v dokumentaci uvedeno, že jejím argumentem může být pouze platné doménové jméno. Funkce je však použita tak, že může být volána i s IP adresou získanou z URL.

Chyba se neprojevuje při používání JSR a to zejména proto, že kolize částí IPv4 adres je velice nízká. Samotný počet webových stránek, na které se přistupuje výhradně přes IPv4 adresu, je také velice nízký. Proto tato chyba má nízkou důležitost. Jedná se spíše o upozornění na nesrovnalost v použití funkce `extractSubDomains`, která by s IP adresou neměla být vůbec volána. Pravděpodobnost projevu chyby při používání JSR je velice nízká.

8.2 Integrační testy

Integrační testy vznikly pro dvě různé verze JSR. Nejdříve pro verzi 0.2 a následně pro verzi 0.3, která v sobě obsahuje novou funkcionalitu dodanou vývojářským týmem v průběhu akademického roku 2019/2020. Ač je kostra integračních testů shodná pro obě verze, testy se přesto liší nejen v očekávaných hodnotách, ale částečně také v definicích jednotlivých testů.

Řádky tabulky 8.3 uvádí kombinace prostředí a nastavení JSR, pro které byly integrační testy spuštěny. Tabulka platí pro testování obou verzí JSR (0.2 i 0.3). Po proběhnutí všech testů na všech uvedených prostředích bylo zjištěno, že na výsledky testů neměla změna operačního systému vliv. Na všech testovaných operačních systémech se výsledky shodovaly. To však již nelze říci o prohlížečích, u kterých bylo zjištěno, že některé testy v prohlížeči Google Chrome v pořádku projdou, zatímco v prohlížeči Mozilla Firefox selžou. Bližší výsledky jsou představeny dále.

Operační systém	Webový prohlížeč	JSR úrovně
Windows 10	Google Chrome (v. 84)	0, 1, 2, 3
	Mozilla Firefox (v. 68.10.0 ESR)	0, 1, 2, 3
Ubuntu 20.04 LTS	Google Chrome (v. 84)	0, 1, 2, 3
	Mozilla Firefox (v. 68.10.0 ESR)	0, 1, 2, 3
CentOS 7	Google Chrome (v. 84)	0, 1, 2, 3
	Mozilla Firefox (v. 68.10.0 ESR)	0, 1, 2, 3

Tabulka 8.3: Prostředí, ve kterých bylo integrační testování spuštěno, a nastavené úrovně JSR, které byly v daném prostředí otestovány.

V následujících sekcích jsou pro danou verzi shrnuté výsledky testů v podobě tabulek s hodnotami „OK“ a „CHYBA“. „OK“ značí, že test byl úspěšně dokončen (tj. hodnota získaná z prohlížeče odpovídá očekávané hodnotě), zatímco „CHYBA“ značí selhání testu (tj. získaná hodnota neodpovídá očekávané hodnotě). Jedna tabulka obsahuje výstup testování z jednoho webového prohlížeče. Z tabulek lze také přesně zjistit, jaké všechny vlastnosti byly testovány. Jak již bylo zmíněno, na všech třech operačních systémech byly výsledky testů shodné (získané hodnoty shodné nebyly, ale výsledky testů se shodovaly), proto jsou výsledky pro daný prohlížeč uvedeny v jedné tabulce bez rozlišení operačního systému.

Za přehledem výsledků testování v podobě tabulek následuje podrobnější rozbor těch testů, které neprošly, a jsou komentovány odhalené chyby. Výsledky testů jsou reprodukovatelné. Postup pro spuštění integračních testů je popsán v příloze C.

JSR verze 0.2

Tabulky 8.4 a 8.5 obsahují výsledky integračních testů pro rozšíření JSR ve verzi 0.2 v prohlížečích Google Chrome a Mozilla Firefox.

V obou prohlížečích selhal test ověřující podvrhnutí hodnot elementu `canvas` (plátno). Podvrhnutí hodnot při čtení z elementu `canvas` je aktivní až na úrovních 2 a 3. Na obou úrovních v obou prohlížečích test selhal. Prázdné plátno lze reprezentovat 2D polem (maticí), které obsahuje na všech pozicích pouze čísla 0. Neprázdné plátno obsahuje v takovém 2D poli i nenulové hodnoty, které nesou informace o grafických prvcích nakreslených na plátně. Pokud ale chceme podvrhnout toto 2D pole, je nutné všechny jeho hodnoty nastavit na nulu, aby plátno vypadalo jako prázdné. Rozšíření JSR však při podvrhnutí hodnot způsobilo, že vrácené 2D pole mělo na všech pozicích pouze hodnoty 255. Všechny hodnoty tak byly invertované. Takové unikátní plátno však vede k zanechávání téměř jedinečného otisku. Namísto vrácení prázdného plátna, které by se ztratilo mezi plátny dalších uživatelů s prázdným elementem `canvas`, bylo vráceno bílé plátno, což vedlo ke zvýšení unikátnosti otisku webového prohlížeče. Pomocí testů se na tento problém podařilo upozornit a v následující verzi byla chyba odstraněna. Jelikož se nyní již jedná o známou a vyřešenou chybu, byl test elementu `canvas` označen příznakem `xfailed`, který značí, že je u něho očekáváno selhání a není nutné tuto chybu reportovat ve výsledcích testů.

Druhý odhalený problém se týká pouze prohlížeče Mozilla Firefox a jedná se o nefunkční zaokrouhlování hodnot milisekund. Na úrovních 1, 2 a 3 rozšíření JSR jsou postupně zaokrouhlovány hodnoty času a časových razítek na desítky, stovky a tisíce milisekund. Zatímco zaokrouhlování `performance.now()` a `geolocation.timestamp` probíhá v pořádku, k zaokrouhlení milisekund při volání `getTime()` na objektu typu `Date` nedochází. Lze si toho povšimnout na ukázkových získaných hodnotách v tabulce 8.6.

Test hodnoty `Date().getTime()` byl laděn za účelem nalezení chyby, ale po kontrole a ověření funkčnosti a po několika opakování testu na různých zařízeních bylo toto selhání testů považováno za upozornění na možnou neznámou chybu. Jelikož dosud není jisté, co je příčinou, je test ponechán ve stavu, kdy končí chybou, dokud nebude příčina zjištěna a odstraněna.

JSR verze 0.3

Tabulky 8.7 a 8.8 obsahují výsledky integračních testů pro rozšíření JSR ve verzi 0.3 v prohlížečích Google Chrome a Mozilla Firefox.

V důsledku změn ve verzi 0.3 je v rozšíření JSR dočasně nefunkční podvrhnutí pozice. V návaznosti na to jsou deaktivované také testy GPS.

Testovaná hodnota	Chrome (Level 0)	Chrome (Level 1)	Chrome (Level 2)	Chrome (Level 3)
navigator.userAgent	OK	OK	OK	OK
navigator.appVersion	OK	OK	OK	OK
navigator.platform	OK	OK	OK	OK
navigator.vendor	OK	OK	OK	OK
navigator.language	OK	OK	OK	OK
navigator.languages	OK	OK	OK	OK
navigator.cookieEnabled	OK	OK	OK	OK
navigator.doNotTrack	OK	OK	OK	OK
navigator.oscpu	OK	OK	OK	OK
navigator.deviceMemory	OK	OK	OK	OK
navigator.hardwareConcurrency	OK	OK	OK	OK
document.referrer	OK	OK	OK	OK
geolocation.accuracy	OK	OK	OK	OK
geolocation.altitude	OK	OK	OK	OK
geolocation.altitudeAccuracy	OK	OK	OK	OK
geolocation.heading	OK	OK	OK	OK
geolocation.latitude	OK	OK	OK	OK
geolocation.longitude	OK	OK	OK	OK
geolocation.speed	OK	OK	OK	OK
geolocation.timestamp	OK	OK	OK	OK
performance.now()	OK	OK	OK	OK
Date()	OK	OK	OK	OK
Date().getTime()	OK	OK	OK	OK
canvas.getImageData()	OK	OK	CHYBA	CHYBA

Tabulka 8.4: Výsledky integračních testů ve webovém prohlížeči Google Chrome s rozšířením JSR ve verzi 0.2.

Verze 0.3 přinesla opravu podvrhnutí dat elementu `canvas`, takže v případě úrovně JSR 2 nebo 3 je vráceno 2D pole tvořené samými nulami, které má význam prázdného plátna. V případě prohlížeče Google Chrome s instalovaným rozšířením JSR na úrovni 3 selhává test plátna. Není to však z důvodu chybného podvrhnutí dat z plátna, ale v důsledku zdánlivě nesouvisející chyby, která brání interakci nástroje Selenium s webovou stránkou. Test neselže z důvodu získání jiné než očekávané hodnoty, ale selže z důvodu vyvolání výjimky v běhovém prostředí jazyka JavaScript při pokusu o interakci nástroje Selenium s webovou stránkou za účelem získání dat z plátna. Ve výsledcích integračních testů je upozorněno na případy, jako je tento, chybovým hlášením, že test selhal z důvodu, že nebylo možné získat testovanou hodnotu. Další experimenty navíc ukázaly, že zkoumaná výjimka je vyvolána pouze při konfiguraci úrovně 3 v JSR v prohlížeči Google Chrome a nesouvisí

Testovaná hodnota	**Firefox (Level 0)**	**Firefox (Level 1)**	**Firefox (Level 2)**	**Firefox (Level 3)**
navigator.userAgent	OK	OK	OK	OK
navigator.appVersion	OK	OK	OK	OK
navigator.platform	OK	OK	OK	OK
navigator.vendor	OK	OK	OK	OK
navigator.language	OK	OK	OK	OK
navigator.languages	OK	OK	OK	OK
navigator.cookieEnabled	OK	OK	OK	OK
navigator.doNotTrack	OK	OK	OK	OK
navigator.oscpu	OK	OK	OK	OK
navigator.deviceMemory	OK	OK	OK	OK
navigator.hardwareConcurrency	OK	OK	OK	OK
document.referrer	OK	OK	OK	OK
geolocation.accuracy	OK	OK	OK	OK
geolocation.altitude	OK	OK	OK	OK
geolocation.altitudeAccuracy	OK	OK	OK	OK
geolocation.heading	OK	OK	OK	OK
geolocation.latitude	OK	OK	OK	OK
geolocation.longitude	OK	OK	OK	OK
geolocation.speed	OK	OK	OK	OK
geolocation.timestamp	OK	OK	OK	OK
performance.now()	OK	OK	OK	OK
Date()	OK	OK	OK	OK
Date().getTime()	OK	CHYBA	CHYBA	CHYBA
canvas.getImageData()	OK	OK	CHYBA	CHYBA

Tabulka 8.5: Výsledky integračních testů ve webovém prohlížeči Mozilla Firefox s rozšířením JSR ve verzi 0.2.

pouze se získáváním dat z plátna, ale obecně s téměř jakoukoliv interakcí. Chybová zpráva vyvolané výjimky je „*Failed to execute 'getRandomValues' on 'Crypto': parameter 1 is not of type 'ArrayBufferView'*“. Vzhledem k tomu, že výjimka není vyvolána bez instalovaného rozšíření JSR, ač je prováděn totožný test, je nutné hledat příčinu této chyby v rozšíření JSR. Ve výsledcích systémových testů, které jsou rozebrány v následující sekci, si lze povšimnout stejné chyby v záznamech z konzole u celé řady webových stránek.

Stále přetrvává chyba způsobující vracení přesných hodnot milisekund, ač by měly být zaokrouhlené. To by mohlo být způsobeno tím, že se daný koncový bod API správně neobaluje. Zatímco v Google Chrome se tato chyba projevuje pouze s JSR na úrovni 3, v Mozilla Firefox se tato chyba projevuje i na úrovních 1 a 2 jako tomu bylo i v JSR ve verzi 0.2.

JSR úroveň	Date().getTime()	
	Očekávaná hodnota	**Získaná hodnota**
1	1594901867740 (skutečná hodnota zaokrouhlená na desítky)	1594901867741 (CHYBA: nezaokrouhleno)
2	1594901885800 (skutečná hodnota zaokrouhlená na stovky)	1594901885763 (CHYBA: nezaokrouhleno)
3	1594901903000 (skutečná hodnota zaokrouhlená na tisíce)	1594901902939 (CHYBA: nezaokrouhleno)

Tabulka 8.6: Očekávané a získané hodnoty při testu zaokrouhlování milisenkud ve webovém prohlížeči Mozilla Firefox pro úrovně JSR 1 až 3.

Testovaná hodnota	**Chrome (Level 0)**	**Chrome (Level 1)**	**Chrome (Level 2)**	**Chrome (Level 3)**
navigator.userAgent	OK	OK	OK	OK
navigator.appVersion	OK	OK	OK	OK
navigator.platform	OK	OK	OK	OK
navigator.vendor	OK	OK	OK	OK
navigator.language	OK	OK	OK	OK
navigator.languages	OK	OK	OK	OK
navigator.cookieEnabled	OK	OK	OK	OK
navigator.doNotTrack	OK	OK	OK	OK
navigator.oscpu	OK	OK	OK	OK
navigator.deviceMemory	OK	OK	OK	OK
navigator.hardwareConcurrency	OK	OK	OK	OK
document.referrer	OK	OK	OK	OK
performance.now()	OK	OK	OK	CHYBA
Date()	OK	OK	OK	OK
Date().getTime()	OK	OK	OK	CHYBA
canvas.getImageData()	OK	OK	OK	CHYBA

Tabulka 8.7: Výsledky integračních testů ve webovém prohlížeči Google Chrome s rozšířením JSR ve verzi 0.3.

Ukázkou mohou být opět výsledky konkrétních běhů integračních testů zachycených v tabulkách 8.9 a 8.10.

Testovaná hodnota	**Firefox (Level 0)**	**Firefox (Level 1)**	**Firefox (Level 2)**	**Firefox (Level 3)**
navigator.userAgent	OK	OK	OK	OK
navigator.appVersion	OK	OK	OK	OK
navigator.platform	OK	OK	OK	OK
navigator.vendor	OK	OK	OK	OK
navigator.language	OK	OK	OK	OK
navigator.languages	OK	OK	OK	OK
navigator.cookieEnabled	OK	OK	OK	OK
navigator.doNotTrack	OK	OK	OK	OK
navigator.oscpu	OK	OK	OK	OK
navigator.deviceMemory	OK	OK	OK	OK
navigator.hardwareConcurrency	OK	OK	OK	OK
document.referrer	OK	OK	OK	OK
performance.now()	OK	OK	OK	CHYBA
Date()	OK	OK	OK	OK
Date().getTime()	OK	CHYBA	CHYBA	CHYBA
canvas.getImageData()	OK	OK	OK	OK

Tabulka 8.8: Výsledky integračních testů ve webovém prohlížeči Mozilla Firefox s rozšířením JSR ve verzi 0.3.

JSR úroveň	Date().getTime()		performance.now()	
	Očekávaná hodnota	**Získaná hodnota**	**Očekávaná hodnota**	**Získaná hodnota**
3	1594981059000 (skutečná hodnota zaokrouhlená na tisíce)	1594981058616 (CHYBA: nezaokrouhleno)	3000 (skutečná hodnota zaokrouhlená na tisíce)	3445.784999974265 (CHYBA: nezaokrouhleno)

Tabulka 8.9: Očekávané a získané hodnoty při testu zaokrouhlování milisenkud ve webovém prohlížeči Google Chrome pro úroveň JSR 3.

JSR úroveň	Date().getTime()		performance.now()	
	Očekávaná hodnota	**Získaná hodnota**	**Očekávaná hodnota**	**Získaná hodnota**
1	1594970990990 (skutečná hodnota zaokrouhlená na desítky)	1594970990991 (CHYBA: nezaokrouhleno)	OK	
2	1594971008100 (skutečná hodnota zaokrouhlená na stovky)	1594971008094 (CHYBA: nezaokrouhleno)	OK	
3	1594971019000 (skutečná hodnota zaokrouhlená na tisíce)	1594971019435 (CHYBA: nezaokrouhleno)	2000 (skutečná hodnota zaokrouhlená na tisíce)	2331 (CHYBA: nezaokrouhleno)

Tabulka 8.10: Očekávané a získané hodnoty při testu zaokrouhlování milisenkud ve webovém prohlížeči Mozilla Firefox pro úrovně JSR 1 až 3.

8.3 Systémové testy

Systémové testy jsou univerzální pro jakoukoliv verzi JSR. Ve výchozím nastavení se při zahájení testů sestavuje rozšíření z dostupných zdrojových kódů. Řádky tabulky 8.11 uvádí, na jakých prostředích a v jaké konfiguraci bylo rozšíření JSR testováno. Ohledně konfigurace je nutné zmínit, že pro systémové testy nebylo možné využít prohlížeč Mozilla Firefox ze dvou důvodů:

- Rozhraní pro zachytávání výstupu z konzole není v dosavadních verzích ovladače *geckodriver* podporováno [31]. Od verze 65 prohlížeče Mozilla Firefox sice existuje způsob, jak tento problém částečně obejít [75], ale toto obejití vyžaduje importování existujícího profilu při spuštění webového prohlížeče, což naráží na omezení nástroje Selenium Grid, které je popsáno v následujícím bodě.
- V nástroji Selenium Grid je problematická instalace rozšíření ze souboru v distribuovaném prostředí pro prohlížeč Mozilla Firefox. Stejně tak je problematický import existujícího profilu do prohlížeče Mozilla Firefox při jeho spuštění. Zatímco pro prohlížeč Google Chrome existuje rozhraní pro nahrání balíčku rozšíření, jeho zakódování do *Base64*, předání zakódovaného souboru testovacímu uzlu a následné dekódování a instalace rozšíření z originálního souboru, pro prohlížeč Mozilla Firefox zatím podobné rozhraní neexistuje a zdá se být jedinou možností naprogramovat si vlastní nástroj pro vzdálenou instalaci rozšíření do prohlížeče Mozilla Firefox a takový program ještě integrovat do nástroje Selenium Grid. Obdobě platí, že vlastní nástroj by bylo potřeba naprogramovat i pro import existujícího profilu.

V nástroji Selenium Grid je znatelná rozdílná úroveň podpory jednotlivých prohlížečů a po uvážení předchozích dvou bodů byl pro spouštění systémových testů vybrán pouze prohlížeč Google Chrome. Základní řídicí algoritmus systémových testů však už nyní počítá i s možností pozdějšího testování ve vícero různých prohlížečů a již nyní z konfigurace bere pole prohlížečů, ve kterých má testování proběhnout, nikoliv jen jedinou hodnotu. V případě potřeby testovat v budoucnu i v dalších prohlížečích nebude vyžadována úprava algoritmu, ale pouze doplnění konfiguračního souboru a některých vstupně-výstupních operací.

Operační systém	**Webový prohlížeč**	**JSR úrovně**	**Počet zařízení**
Windows 7	Google Chrome (v. 84)	0, 1, 2, 3	1, 1+1, 1+2, 1+4, 1+8
Ubuntu 20.04 LTS	Google Chrome (v. 84)	0, 1, 2, 3	1
CentOS 7	Google Chrome (v. 84)	0, 1, 2, 3	1, 1+1, 1+2, 1+4, 1+8

Tabulka 8.11: Prostředí, ve kterých bylo systémové testování spuštěno, a nastavené úrovně JSR, které byly v daném prostředí otestovány.

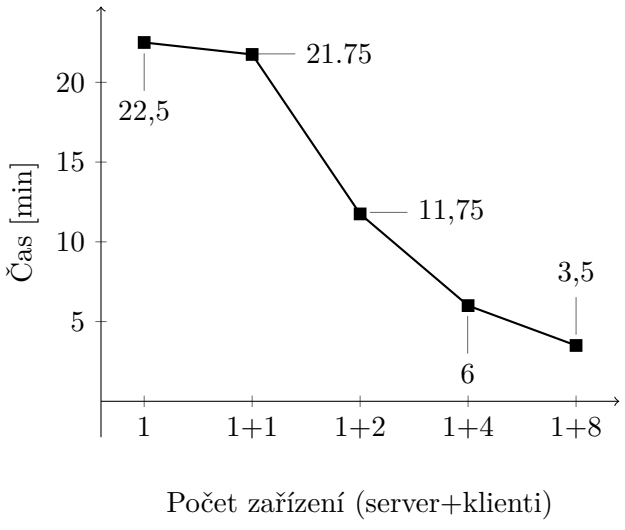
Systémové testy byly spuštěny pro prvních 500 webových stránek z žebříčku nejnavštěvovanějších webových stránek TRANCO [63] vytvořeného dne 14. 7. 2020[1]. V případě vlastního spuštění systémových testů je možné stáhnout novější verzi žebříčku TRANCO nebo jakéhokoliv jiného žebříčku, který má stejný formát jako žebříček TRANCO.

[1] https://tranco-list.eu/list/K3ZW

Paralelní testování v distribuovaném prostředí

Systémové testy jsou založeny na architektuře typu server a klienti, která umožňuje spouštění testů v distribuovaném prostředí. Zároveň je ale podporováno i spouštění pouze na jediném zařízení. Proto tabulka 8.11 má oproti jiným úrovním testování ještě sloupec *Počet zařízení*, který udává, pro jaký počet zařízení v distribuovaném prostředí byly testy spouštěny a ověřeny. Ze všech zařízení je vždy jedno postaveno do role Selenium serveru a ostatní slouží jako testovací uzly. Zápis 1+2 znamená, že byly v distribuovaném prostředí paralelně běžící 3 zařízení, z nichž jedno bylo Selenium server a zbývající dvě zařízení byly testovacími uzly. Zařízení distribuovaného prostředí byla vždy se stejným operačním systémem a se stejnou verzí prohlížeče za účelem urychlení systémových testů, nikoliv pokrytí širší škály platforem a verzí prohlížečů.

V grafu na obrázku 8.1 je zachycena doba běhu získávání dat (první fáze systémových testů) v závislosti na počtu zařízení v distribuovaném prostředí. Graf vychází z provedených experimentů v laboratoři C304 na FIT VUT. Získávání dat bylo vždy spuštěno pro prvních sto webů z žebříčku TRANCO. Ze spojnicového grafu lze vyčíst, že doba běhu na jediném zařízení (tj. paralelismus bez distribuovaného prostředí) je o málo vyšší, než doba běhu v distribuovaném prostředí, kde je jeden server a jeden klient. To je dáno rozložením zátěže mezi dvě různá zařízení v distribuovaném prostředí 1+1, což vede k mírnému urychlení získávání dat. Postupným přidáváním počtu klientů do distribuovaného prostředí docházelo ke znatelnému zrychlení získávání dat. Lze si všimnout, že každým zdvojnásobením počtu klientů došlo k urychlení běhu téměř dvakrát. Důvodem, proč nedošlo při každém zdvojnásobení počtu klientů na snížení doby běhu na polovinu, je vzrůstající režie při vzrůstajícím počtu zařízení v distribuovaném prostředí. Závislost počtu klientů na času potřebného k získávání dat ze stejného počtu webových stránek by se dala přibližně vyjádřit funkcí nepřímé úměrnosti – v uvedeném konkrétním případě $y = \frac{24}{x}$, kde x je počet klientů a y je doba běhu získávání dat ze sto webových stránek v minutách.



Obrázek 8.1: Doba běhu získávání dat v závislosti na počtu zařízení v distribuovaném prostředí s využitím nástroje Selenium Grid.

Výsledky systémových testů

V následujících odstavcích jsou shrnuty výsledky systémových testů. Výsledky testů jsou reprodukovatelné. Postup pro spuštění systémových testů je popsán v příloze D.

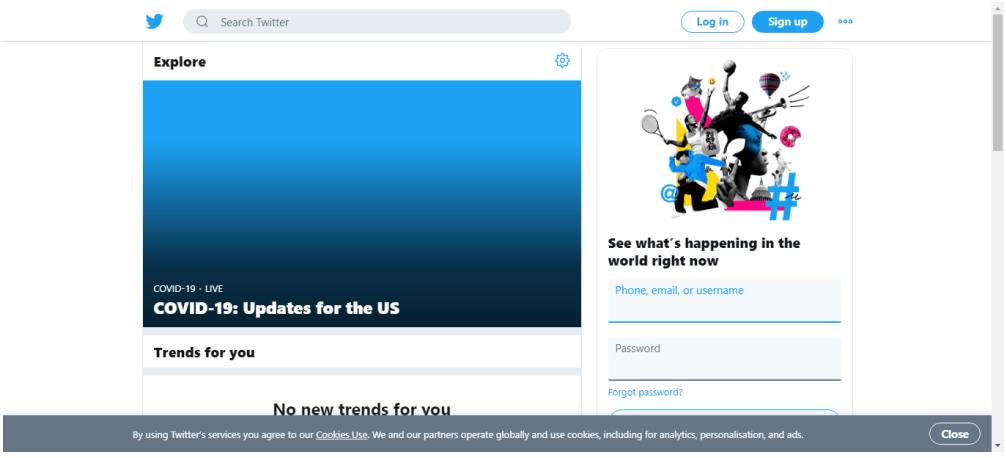
Výstupem testů je porovnání dat ze stejné webové stránky načtené ve stejných prohlížečích, přičemž v jednom bylo nainstalováno rozšíření JSR a ve druhém nikoliv. Porovnávanými daty jsou záznamy z konzole a snímky obrazovky. Podrobné výsledky systémových testů si lze prohlédnout v souborech na paměťovém médiu, které je přiloženo k této práci. V rámci technické zprávy je vybráno pouze několik ukázkových výsledků.

Výsledky porovnání snímků obrazovky

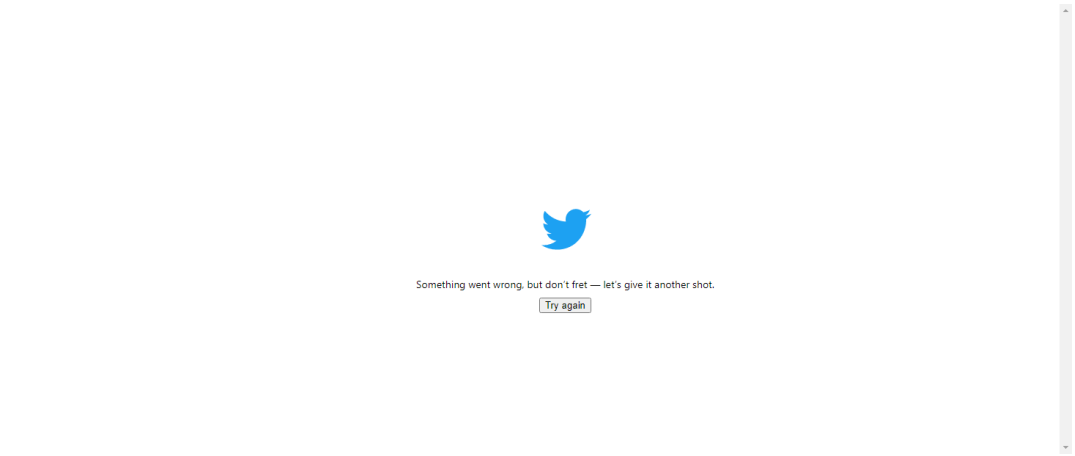
Výstupem testování snímků obrazovky je HTML soubor definující webovou stránku, na které jsou dle pořadí v žebříčku seřazeny snímky obrazovky testovaných webových stránek. U každé webové stránky jsou dva snímky obrazovky (jeden získaný bez nainstalovaného JSR, druhý snímek získaný s nainstalovaným JSR), rozdílový snímek a číselné vyjádření rozdílu snímků webové stránky.

V záhlaví celé stránky je posuvník sloužící k filtrování zobrazených webových stránek. Posuvník nastavuje prahovou hodnotu rozdílu snímku bez JSR a s JSR. Webové stránky, které mají příslušnou hodnotu pod tímto prahem, jsou skryté. Díky tomu je možné filtrovat výsledky dle potřeby a zobrazit si tak jenom webové stránky, které byly nejvíce ovlivněny rozšířením JSR.

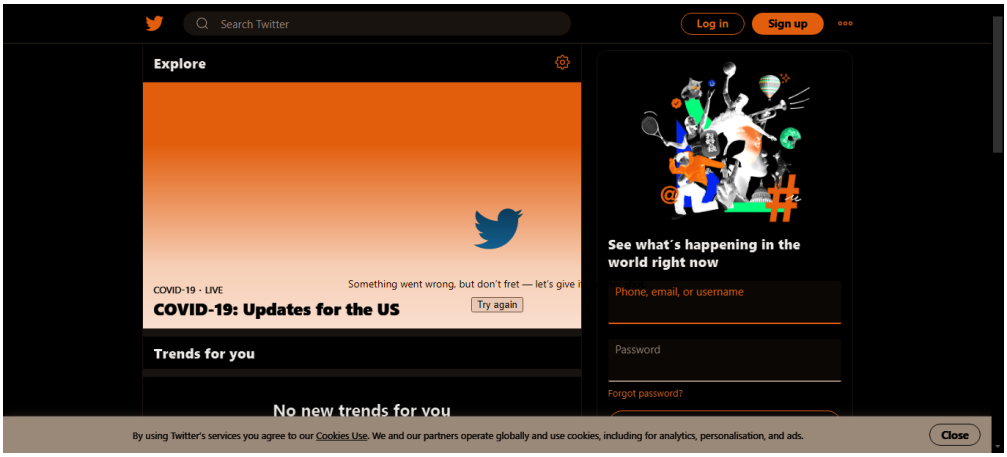
Nejvýraznější dopad z pohledu uživatele, jaký může rozšíření JSR na načítanou webovou stránku mít, je úplné nebo téměř úplné zabránění v jejím vykreslení. Pro úroveň 3 rozšíření JSR opakovaně odhalily systémové testy tento problém například u webových stránek `twitter.com` a `instagram.com`, které se dlouhodobě pohybují v první desítce nejnavštěvovanějších webových stránek. Na obrázku 8.2 je zachycena načtená webová stránka `twitter.com` v prohlížeči Google Chrome bez JSR, na obrázku 8.3 je zachycena stejná webová stránka ve stejném prohlížeči jenom s nainstalovaným JSR nastaveným na úroveň 3 a na posledním obrázku 8.4 je graficky zobrazen rozdíl snímků, který numericky odpovídá hodnotě přibližně 58,483.



Obrázek 8.2: Webová stránka twitter.com načtená v prohlížeči Google Chrome bez nainstalovaného rozšíření JSR.



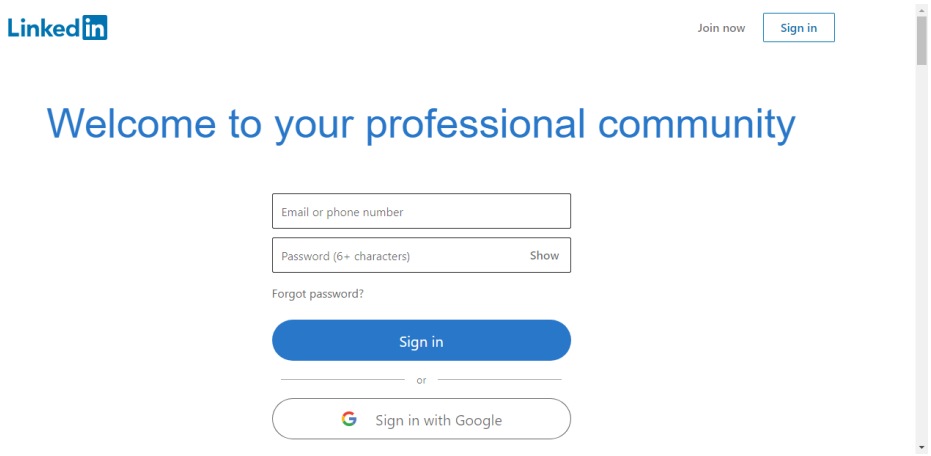
Obrázek 8.3: Webová stránka twitter.com načtená v prohlížeči Google Chrome s nainstalovaným rozšíření JSR nastaveným na úroveň 3.



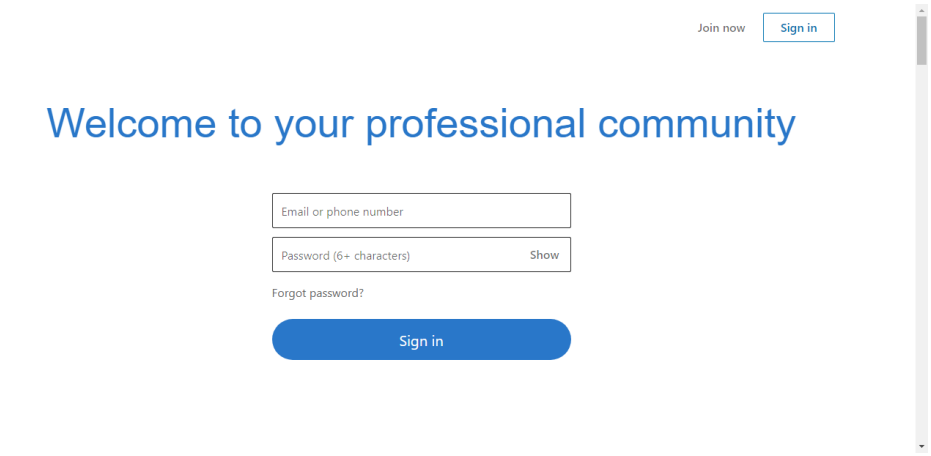
Obrázek 8.4: Rozdíl snímků webové stránky twitter.com.

Dalším chybovým případem, který je dle provedených testů nejčastější, je nenačtení nebo chybné načtení pouze určitého elementu na webové stránce. Příkladem může být webová stránka `linkedin.com`, u které stojí za pozornost zejména nenačtení loga stránky v prohlížeči Google Chrome s instalovaným rozšířením JSR nastaveným na úroveň 3. Trojice obrázků 8.5, 8.6 a 8.7 opět zachycuje originální snímky a jejich rozdíl, jehož numerická hodnota je přibližně 0,705.

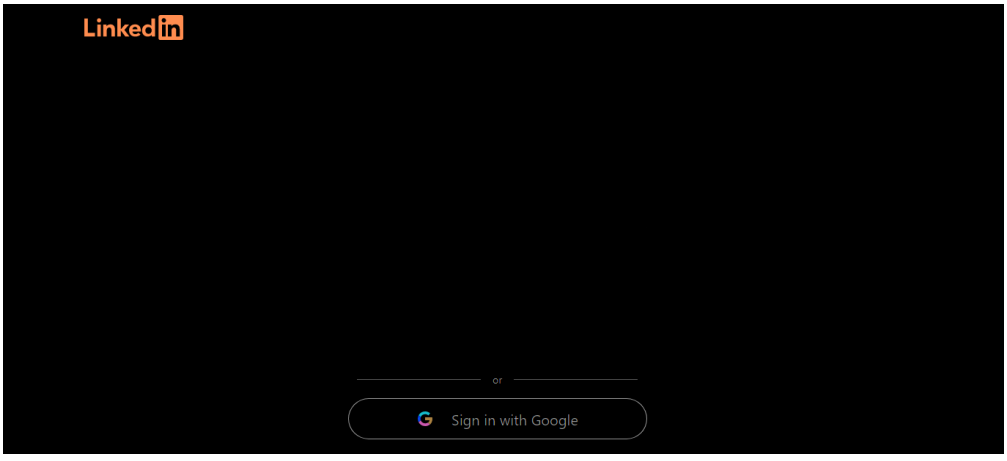
V předchozích dvou typech výsledků se jednalo o případy anglicky označované jako *True Positive*. Tedy případy, ve kterých testy správně identifikovaly problém s načítáním webové stránky a upozornily na něho zvýšeným rozdílovým číslem snímků. Kromě těchto případů se ve výsledcích porovnávání snímků objevují i takzvané *False Positive* hlášení. Jedná se o webové stránky, u nichž je identifikován problém, ač žádný nenastal. Typicky se jedná o webové stránky, které obsahují dynamicky se měnící obsah. Ukázkovým příkladem může být webová stránka `youtube.com`, u které se všechny elementy načtou správně, pouze je změněn obsah – konkrétně jsou zaměněna určitá videa, což je korektní chování nezpůsobené rozšířením JSR. Přesto porovnávání snímků označí webovou stránku nenulovým rozdílovým ukazatelem, protože vychází pouze z vizuálních informací, které se změnily.



Obrázek 8.5: Webová stránka linkedin.com načtená v prohlížeči Google Chrome bez nainstalovaného rozšíření JSR.

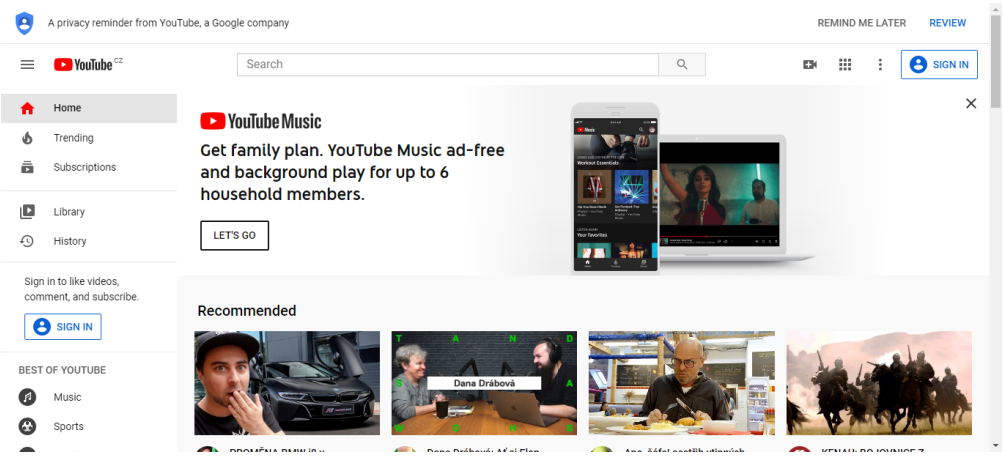


Obrázek 8.6: Webová stránka linkedin.com načtená v prohlížeči Google Chrome s nainstalovaným rozšíření JSR nastaveným na úroveň 3.

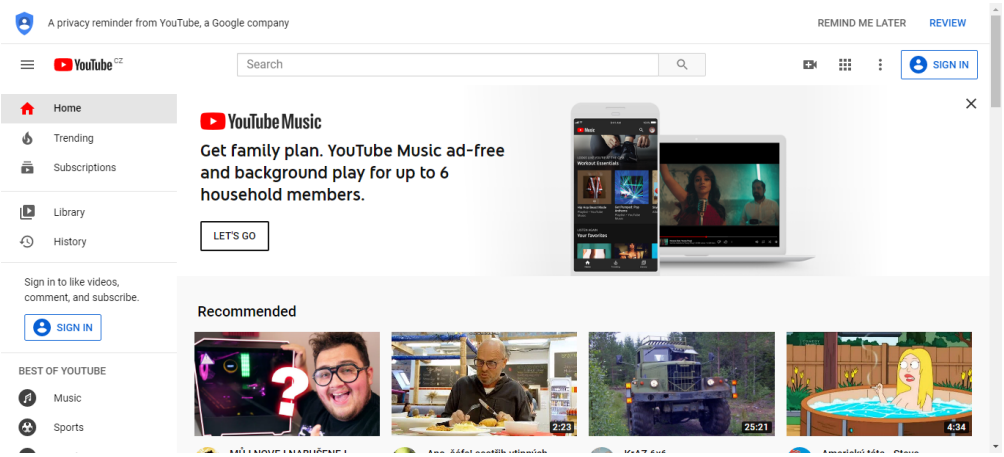


Obrázek 8.7: Rozdíl snímků webové stránky linkedin.com.

V kontextu těchto možných falešných poplachů je třeba vnímat analýzu snímků pouze jako přehledný informativní nástroj a při vyhodnocování testů přihlédnout i k výstupům porovnávání záznamů z konzole a obecného posouzení vizuálních změn. Trojice obrázků 8.8, 8.9 a 8.10 zachycuje originální snímky webové stránky youtube.com a jejich rozdíl, jehož numerická hodnota je přibližně 8,105.



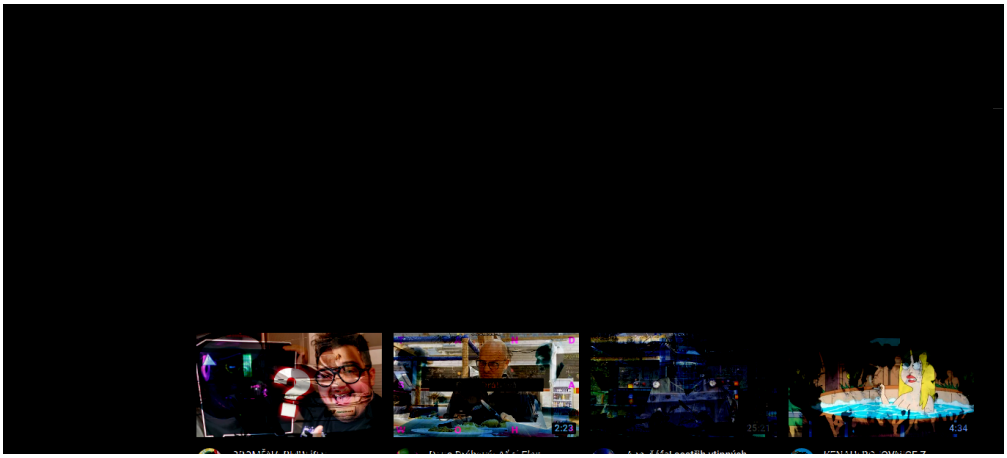
Obrázek 8.8: Webová stránka youtube.com načtená v prohlížeči Google Chrome bez nainstalovaného rozšíření JSR.



Obrázek 8.9: Webová stránka youtube.com načtená v prohlížeči Google Chrome s nainstalovaným rozšíření JSR nastaveným na úroveň 3.

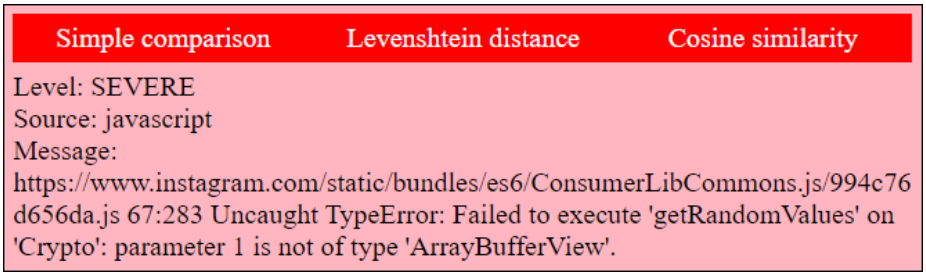
Výsledky porovnání záznamů z konzole

Identifikované záznamy z konzole, které se objevily až s instalovaným rozšířením JSR, mohou být upřesňující informací k rozdílu snímků obrazovky poskytující detailnější odůvodnění, proč například vůbec nedošlo k vykreslení webové stránky. Jednotlivé záznamy jsou označeny červenými štítky podle toho, jaké metody identifikovaly záznam jako přidaný. Záznam je podbarven celý růžově, pokud alespoň jedna metoda identifikovala záznam jako přidaný. Příklad pro záznam identifikovaný všemi třemi metodami jako přidaný rozšířením



Obrázek 8.10: Rozdíl snímků webové stránky youtube.com.

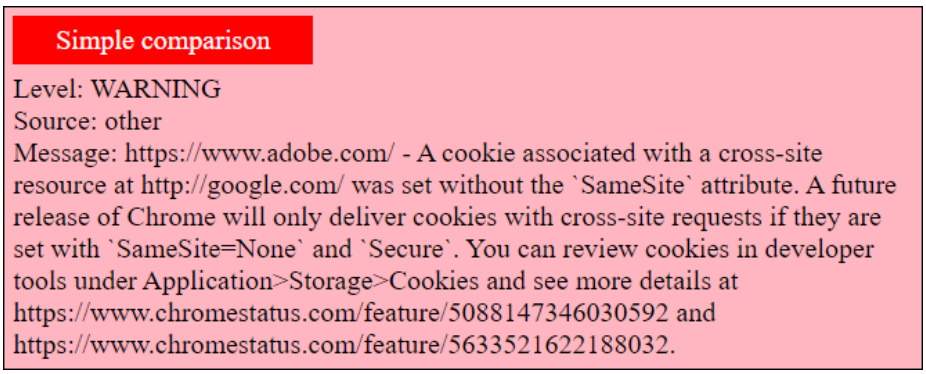
JSR na úrovni 3 z webové stránky `instagram.com` je zobrazen na obrázku 8.11. Za pozornost stojí, že na stejnou chybu, jako je na obrázku 8.11, upozornily již integrační testy, když selhal test elementu `canvas`.



Obrázek 8.11: Záznam z konzole identifikovaný všemi třemi metodami jako přidaný rozšířením JSR na úrovni 3.

Na základě výsledků systémových testů, které byly spuštěny pro prvních 500 webových stránek z žebříčku nejnavštěvovanějších webových stránek TRANCO [63] vytvořeného dne 14. 7. 2020, byly rozšířením JSR na úrovni 3 přidány průměrně 2 záznamy pro každou testovanou stránku. Nejčastěji zastoupenou chybou byla chyba se zprávou „*Failed to execute 'getRandomValues' on 'Crypto'*“ zachycená na obrázku 8.11, která tvořila přibližně 17 % všech záznamů přidaných rozšířením JSR.

I při analýze záznamů z konzole se vyskytly falešné poplachy (anglicky *False Positive*), které však byly hlášeny téměř výhradně pouze jednoduchou metodou, která porovnává dva záznamy z konzole na absolutní shodu. Pokročilejší metody porovnání již počítají s určitou odchylkou podobnosti a takové záznamy neoznačily jako přidané rozšířením JSR. Metody Levenshteinova vzdálenost a Kosinová podobnost se ve více než 90 % případů shodly na stejném výsledku. Příklad falešného poplachu, který nahlásila jednoduchá metoda, ale pokročilejší metody nikoliv, je záznam na obrázku 8.12 z webové stránky `adobe.com`. Tento záznam byl získán z prohlížeče s instalovaným rozšířením JSR a jednoduchou metodou byl označen jako přidaný rozšířením JSR přesto, že v prohlížeči bez instalovaného JSR byl zachycen téměř totožný záznam.



Obrázek 8.12: Záznam z konzole identifikovaný pouze jednoduchou metodou jako přidaný rozšířením JSR na úrovni 3.

Kapitola 9

Závěr

V práci se mi podařilo navrhnout, implementovat a vyhodnotit výsledky automatických testů pro projekt JavaScript Restrictor (JSR). Testy jsou rozděleny do tří úrovní – jednotkové, integrační a systémové. Jednotkové testy ověřují chování jednotlivých funkcí, integrační testy ověřují správné obalování koncových bodů rozhraní prohlížeče a systémové testy kontrolují, zda JSR nepotlačuje chtěnou funkcionalitu webových stránek.

Prostřednictvím testů se mi v projektu podařilo objevit a popsat dosud nezjištěné chyby, některé z nich jsou nyní zmíněny. Jednotkové testy odhalily chybu související s extrakcí domény z URL, která byla na základě výsledků testů opravena. Integrační testy odhalily v JSR ve verzi 0.2 chybné vracení bílého plátna namísto prázdného, což vedlo k nežádoucímu zvýšení unikátnosti otisku webového prohlížeče. V následující verzi 0.3 byla tato chyba odstraněna a konfirmačními testy bylo její odstranění ověřeno. Systémové testy upozornily na webové stránky, které se v prohlížeči s instalovaným rozšířením JSR nevykreslí. Jedná se například o webové stránky `twitter.com` a `instagram.com`, které se dlouhodobě pohybují v první desítce nejnavštěvovanějších webových stránek. Díky automatické analýze záznamů z vývojářské konzole prohlížeče v rámci systémových testů se podařila zjistit i možná příčina nevykreslení některých webových stránek, která spočívá v předávání chybného parametru funkci `getRandomValues`.

Dosavadní manuální testování je díky výsledkům práce možné do velké míry nahradit nově vytvořenými automatickými testy. Práce doplňuje výsledky diplomantů pracujících na rozvoji projektu JSR tím, že ověřuje stávající i nově implementovanou funkcionalitu. Bylo schváleno přidání automatických testů vytvořených v rámci této práci do oficiálního repozitáře projektu JSR na platformě GitHub.

Systémové testy jsou navrženy, implementovány a vyzkoušeny pro paralelní běh v distribuovaném prostředí. Dle provedených experimentů je čas potřebný k otestování určitého počtu webových stránek přibližně nepřímo úměrný k počtu testovacích uzlů v distribuovaném prostředí. Díky tomu lze i tisíce webových stránek otestovat řádově za desítky minut, což je při automatickém sekvenčním testováním nebo při použití manuálních testovacích metod nedosažitelné.

Na výsledky automatických testů by mohla navázat práce zaměřená na odstranění zjištěných chyb. Dalším možným rozvojem samotných testů je pokrývání v budoucnu nově implementované funkcionality, což bude vyžadovat pouze přidání konkrétních testů do již připraveného testovacího projektu. Architektura automatických testů je také připravena pro možné rozšíření testování na další prohlížeče a pro rozšíření typů prováděných testů.

Literatura

[1] Akanksha. *Selenium Tutorial For Beginners | What Is Selenium? | Selenium Automation Testing Tutorial | Edureka* [YouTube]. Bangalore, India: "edureka!", 2017. [Online; navštíveno 12.09.2019]. Dostupné z: https://www.youtube.com/watch?v=5FUdrBq-WFo.

[2] Aldaine, A. *Top 10 API Testing Tools in 2020 (Details & Updates Done!)* [Medium]. Květen 2018. [Online; navštíveno 8.1.2020]. Dostupné z: https://medium.com/@alicealdaine/top-10-api-testing-tools-rest-soap-services-5395cb03cfa9.

[3] Alexa. *About Us* [Alexa]. 2019. [Online; navštíveno 20.2.2020]. Dostupné z: https://www.alexa.com/about.

[4] Alexa. *Alexa Traffic Rank* [Chrome web store]. Srpen 2019. [Online; navštíveno 20.2.2020]. Dostupné z: https://chrome.google.com/webstore/detail/alexa-traffic-rank/cknebhggccemgcnbidipinkifmmegdel.

[5] Alexa. *Can the Alexa rankings be cheated?* [Alexa Support]. 2019. [Online; navštíveno 20.2.2020]. Dostupné z: https://support.alexa.com/hc/en-us/articles/200449754-Can-the-Alexa-rankings-be-cheated-.

[6] Alexa. *How are Alexa's traffic rankings determined?* [Alexa Support]. 2019. [Online; navštíveno 20.2.2020]. Dostupné z: https://support.alexa.com/hc/en-us/articles/200449744-How-are-Alexa-s-traffic-rankings-determined-.

[7] Alexa. *Choose the right plan for your goals* [Alexa]. 2020. [Online; navštíveno 20.2.2020]. Dostupné z: https://www.alexa.com/plans.

[8] Alexa. *The top 500 sites on the web* [Alexa]. 2020. [Online; navštíveno 20.2.2020]. Dostupné z: https://www.alexa.com/topsites.

[9] Altaf, I., Dar, J. A., Rashid, F. U. a Rafiq, M. Survey on selenium tool in software testing. In: *2015 International Conference on Green Computing and Internet of Things (ICGCIoT)*. IEEE, 2015, s. 1378–1383. ISBN 978-1-4673-7909-0.

[10] Bamberg, W., Scholz, F., Galaurumadn, Z. et al. *storage* [MDN Web Docs]. Leden 2020. [Online; navštíveno 29.5.2020]. Dostupné z: https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/API/storage.

[11] Bamberg, W., Swisher, J., Smith, I. et al. *Console messages* [MDN Web Docs]. Duben 2020. [Online; navštíveno 10.7.2020]. Dostupné z: https://developer.mozilla.org/en-US/docs/Tools/Web_Console/Console_messages.

[12] Bastl, V. *Automatizace webového prohlížeče*. Brno, 2019. Diplomová práce. Vysoké učení technické v Brně. Fakulta informačních technologií.

[13] Bosch, A., Bogers, T. a Kunder, M. de. Estimating search engine index size variability: a 9-year longitudinal study. *Scientometrics*. Dordrecht: Springer Netherlands. 2016, sv. 107, č. 2, s. 839–856. ISSN 0138-9130.

[14] Brown, C. T., Gheorghiu, G. a Huggins, J. *An Introduction to Testing Web Applications with twill and Selenium*. Sebastopol, California, USA: O'Reilly Media, Inc., 2007. ISBN 978-0-596-52780-8.

[15] Bruns, A., Kornstadt, A. a Wichmann, D. Web Application Tests with Selenium. *IEEE Software*. Los Alamitos: IEEE Computer Society. 2009, sv. 26, č. 5, s. 88–91. ISSN 0740-7459.

[16] Bureš, M., Renda, M., Doležal, M. et al. *Efektivní testování softwaru : klíčové otázky pro efektivitu testovacího procesu*. 1. Praha: Grada, 2016. Profesionál. ISBN 978-80-247-5594-6.

[17] Burnstein, I. *Practical software testing*. Vyd. 1. New York: Springer, 2003. ISBN 0-387-95131-8.

[18] Byun, J., Hiller, C., Coe, B. E. et al. *Mocha* [MochaJS]. Leden 2020. [Online; navštíveno 3.2.2020]. Dostupné z: https://mochajs.org/.

[19] CASTB. *ISTQB Glossary of Testing Terms / Slovník pojmů ISTQB [CZ]* [CASTB]. 2018. [Online; navštíveno 6.1.2020]. Dostupné z: http://castb.org/wp-content/uploads/2018/09/ISTQB_CZ_Glossary_20180831.pdf.

[20] Colantonio, J. *It's back! Selenium IDE Reborn with Dave Haeffner* [Testing Podcast]. 2018. [Online; navštíveno 13.09.2019]. Dostupné z: https://testingpodcast.com/230-its-back-selenium-ide-reborn-with-dave-haeffner/.

[21] Colantonio, J. *11 top open-source API testing tools: What your team needs to know* [TechBeacon]. Leden 2020. [Online; navštíveno 8.1.2020]. Dostupné z: https://techbeacon.com/app-dev-testing/11-top-open-source-api-testing-tools-what-your-team-needs-know.

[22] Diepenbeck, M. Behavior driven development for tests and verification. In: *Formal Modeling and Verification of Cyber-Physical Systems: 1st International Summer School on Methods and Tools for the Design of Digital Systems, Bremen, Germany, September 2015*. Springer Fachmedien, 2015, s. 275–277. ISBN 978-3-6580-9994-7.

[23] Dynatrace. *Dynatrace digital experience monitoring* [Dynatrace]. Březen 2017. [Online; navštíveno 10.1.2020]. Dostupné z: https://assets.dynatrace.com/en/docs/fs/digital-experience-monitoring-fact-sheet.pdf?_ga=2.136823539.2102501253.1580929417-2080574161.1580929417.

[24] Feuerstein, A. *E-commerce loves Street: Critical Path plans encore* [San Francisco Business Times]. Květen 1999. [Online; navštíveno 20.2.2020]. Dostupné z: https://www.bizjournals.com/sanfrancisco/stories/1999/05/24/newscolumn4.html.

[25] Gemius. *Metodika NetMonitoru měření návštěvnosti internetu* [NetMonitor]. 2016. [Online; navštíveno 28.2.2020]. Dostupné z: https://www.netmonitor.cz/sites/default/files/prilohy/190926_Metodika_NetMonitor.pdf.

[26] Gemius. *Online data (OLA)* [NetMonitor]. 2020. [Online; navštíveno 28.2.2020]. Dostupné z: http://www.netmonitor.cz/online-data-ola.

[27] Gojare, S., Joshi, R. a Gaigaware, D. Analysis and Design of Selenium WebDriver Automation Testing Framework. *Procedia Computer Science*. Elsevier B.V. 2015, sv. 50, s. 341–346. ISSN 1877-0509.

[28] Gojare, S., Joshi, R. a Gaigaware, D. Analysis and Design of Selenium WebDriver Automation Testing Framework. *Procedia Computer Science*. Elsevier B.V. 2015, sv. 50, s. 341–346. ISSN 1877-0509.

[29] Google. *chrome.storage* [Chrome Developers]. 2019. [Online; navštíveno 29.5.2020]. Dostupné z: https://developer.chrome.com/apps/storage.

[30] Google. *Google Safe Browsing* [Google]. 2020. [Online; navštíveno 26.2.2020]. Dostupné z: https://safebrowsing.google.com/.

[31] Grainger, T. *Support for Selenium's logging interface* [GitHub]. Říjen 2016. [Online; navštíveno 12.7.2020]. Dostupné z: https://github.com/mozilla/geckodriver/issues/284.

[32] Grebs, C., Colville, S., MacPherson, M. R. et al. *Linter Rules* [GitHub]. Září 2019. [Online; navštíveno 3.2.2020]. Dostupné z: https://mozilla.github.io/addons-linter/.

[33] Grebs, C., Colville, S., MacPherson, M. R. et al. *addons-linter* [npm]. Leden 2020. [Online; navštíveno 3.2.2020]. Dostupné z: https://www.npmjs.com/package/addons-linter.

[34] Gundecha, U. *Selenium Testing Tools Cookbook*. Olton: Packt Publishing, Limited, 2012. ISBN 978-1-8495-1574-0.

[35] Haeffner, D. *Selenium IDE Is Dead, Long Live Selenium IDE!* [Official Selenium Blog]. 2018. [Online; navštíveno 13.09.2019]. Dostupné z: https://seleniumhq.wordpress.com/2018/08/06/selenium-ide-tng/.

[36] Hamill, P. *Unit Test Frameworks: Tools for High-Quality Software Development*. 1. vyd. O'Reilly Media, 2004. O'Reilly Series. ISBN 978-0-5960-0689-1.

[37] Hazem, S. Jasmine. In: *JavaScript Unit Testing*. Packt Publishing, 2013, s. 1–1. ISBN 978-1-7821-6062-5.

[38] Helsby, S. *Optimal Tester to Developer Ratios* [Prolifics Testing]. 2018. [Online; navštíveno 6.1.2020]. Dostupné z: https://www.prolifics-testing.com/blog-articles/optimal-tester-to-developer-ratios.

[39] Hubbard, D. *Cisco Umbrella 1 Million* [Cisco Umbrella | Blog]. Prosinec 2016. [Online; navštíveno 24.2.2020]. Dostupné z: https://umbrella.cisco.com/blog/2016/12/14/cisco-umbrella-1-million/.

[40] *IEEE Standard Glossary of Software Engineering Terminology (610.12-1990)*. New York, USA: IEEE, 1990. 1–84 s. ISBN 978-0-7381-0391-4.

[41] *ISO/IEC/IEEE International Standard – Software and systems engineering – Software testing – Part 1: Concepts and definitions (29119-1-2013)*. USA: IEEE, 2013. ISBN 978-0-7381-8597-2.

[42] ISTQB. *Glossary of Testing Terms* [ISTQB]. 2012. [Online; navštíveno 6.1.2020]. Dostupné z: https://glossary.istqb.org/.

[43] Jones, D. *Majestic Million CSV now free for all, daily.* [Majestic Blog]. Říjen 2012. [Online; navštíveno 25.2.2020]. Dostupné z: https://blog.majestic.com/development/majestic-million-csv-daily/.

[44] Jorgensen, P. C. *Software testing : a craftsman's approach*. 3. vyd. Boca Raton: Auerbach Publications, 2008. ISBN 978-0-8493-7475-3.

[45] Kahle, B. *Internet Archive WayBack Machine* [Internet Archive]. 2020. [Online; navštíveno 20.2.2020]. Dostupné z: https://archive.org/web/.

[46] Kolektiv autorů. *Selenium documentation* [SeleniumHQ Browser Automation]. [Online; navštíveno 10.09.2019]. Dostupné z: https://www.seleniumhq.org/docs/.

[47] Kolektiv autorů. *Selenium wiki* [GitHub]. [Online; navštíveno 20.09.2019]. Dostupné z: https://github.com/SeleniumHQ/selenium/wiki.

[48] Kolektiv autorů. *Total number of Websites* [Internet Live Stats]. 2019. [Online; navštíveno 11.10.2019]. Dostupné z: https://www.internetlivestats.com/total-number-of-websites/.

[49] Kolektiv autorů. *Web Server Survey* [Netcraft]. 2019. [Online; navštíveno 11.10.2019]. Dostupné z: https://news.netcraft.com/archives/category/web-server-survey/.

[50] Krupka, J. *NetMonitor, nebo Google Analytics? Přibývá webů, které nectí jednotnou měnu* [Mediář]. Březen 2015. [Online; navštíveno 28.2.2020]. Dostupné z: https://www.mediar.cz/netmonitor-nebo-google-analytics-pribyva-webu-ktere-necti-jednotnou-menu/.

[51] Kunder, M. de. *The size of the World Wide Web (The Internet)* [WorldWideWebSize.com]. 2019. [Online; navštíveno 11.10.2019]. Dostupné z: https://www.worldwidewebsize.com/.

[52] Le Pochat, V., Van Goethem, T., Tajalizadehkhoob, S. a Joosen, W. TRANCO: A Research-Oriented Top Sites Ranking Hardened Against Manipulation. *Network and Distributed Systems Security (NDSS) Symposium 2019*. 2019, NDSS. DOI: 10.14722/ndss.2019.23386.

[53] Leithead, T., Moon, S., Faulkner, S., Danilo, A. a Eicholz, A. *HTML 5.2*. W3C Recommendation. W3C, prosinec 2017. [Online; navštíveno 12.7.2020]. Dostupné z: https://www.w3.org/TR/2017/REC-html52-20171214/.

[54] Levenshtein, V. I. *Binary Codes Capable of Correcting Deletions, Insertions and Reversals.* Soviet Physics Doklady. *Únor 1966, sv. 10, č. 8, s. 707. ISSN 1028-3358.*

[55] Majestic. *Majestic launch a Bigger Fresh Index [Majestic Blog]. Květen 2018. [Online; navštíveno 25.2.2020]. Dostupné z:* https://blog.majestic.com/company/majestic-launch-a-bigger-fresh-index/.

[56] Majestic. *About Majestic [Majestic]. 2019. [Online; navštíveno 25.2.2020]. Dostupné z:* https://majestic.com/company/about.

[57] Majestic. *Get Started with Majestic [Majestic]. 2019. [Online; navštíveno 25.2.2020]. Dostupné z:* https://majestic.com/plans-pricing.

[58] Majestic. *The Majestic Million [Majestic]. Únor 2020. [Online; navštíveno 25.2.2020]. Dostupné z:* https://majestic.com/reports/majestic-million.

[59] Neiman, *C. Improving the Add-ons Linter [The Mozilla Blog]. Březen 2018. [Online; navštíveno 3.2.2020]. Dostupné z:* https://blog.mozilla.org/addons/2018/03/27/improving-add-ons-linter/.

[60] Page, *A.*, Johnston, *K. a* Rollison, *B.* Jak testuje software Microsoft. *1. Brno: Computer Press, 2009. ISBN 978-80-251-2869-5.*

[61] Patton, *R.* Testování softwaru. *1. Praha: Computer Press, 2002. Programování. Pro každého uživatele. ISBN 80-7226-636-5.*

[62] Pochat, *V. L. Tranco: A Research-Oriented Top Sites Ranking Hardened Against Manipulation [GitHub]. Únor 2019. [Online; navštíveno 27.2.2020]. Dostupné z:* https://github.com/DistriNet/tranco-list.

[63] Pochat, *V. L.*, Goethem, *T. V.*, Tajalizadehkhoob, *S.*, Korczyński, *M. a* Joosen, *W. TRANCO: A Research-Oriented Top Sites Ranking Hardened Against Manipulation [TRANCO]. Únor 2019. DOI: 10.14722/ndss.2019.23386.*

[64] Polčák, *L. a* Timko, *M. JavaScript Restrictor [GitHub]. 2019. [Online; navštíveno 25.5.2020]. Dostupné z:* https://polcak.github.io/jsrestrictor/.

[65] Quantcast. *ABOUT US | Helping Brands Grow in the AI Era. [Quantcast]. 2020. [Online; navštíveno 26.2.2020]. Dostupné z:* https://www.quantcast.com/about-us/.

[66] Quantcast. *QUANTCAST MEASURE | Know your audience. [Quantcast]. 2020. [Online; navštíveno 26.2.2020]. Dostupné z:* https://www.quantcast.com/products/measure-audience-insights/.

[67] Quantcast. *Quantcast Top Sites [Quantcast]. 2020. [Online; navštíveno 26.2.2020]. Dostupné z:* https://ak.quantcast.com/quantcast-top-sites.zip.

[68] Quantcast. *Website Terms of Use [Quantcast]. 2020. [Online; navštíveno 26.2.2020]. Dostupné z:* https://www.quantcast.com/learning-center/quantcast-terms/website-terms-of-use/.

[69] Ramya, *P.*, Sindhura, *V. a* Sagar, *P. V. Testing using selenium web driver. In:* 2017 Second International Conference on Electrical, Computer and Communication Technologies (ICECCT). *IEEE, 2017, s. 1–7. ISBN 978-1-5090-3238-9.*

[70] Reš, *R. Zajištění kvality webových aplikací pomocí nástrojů automatického testování. Brno, 2014. Diplomová práce. Vysoké učení technické v Brně. Fakulta informačních technologií.*

[71] Sargent, *A. 16 reasons why to use Selenium IDE in 2019 (and 2 why not) [applitools BLOG]. 2019. [Online; navštíveno 13.09.2019]. Dostupné z: https://applitools.com/blog/why-selenium-ide-2019.*

[72] Services, *A. W. Alexa Top Sites [aws marketplace]. 2020. [Online; navštíveno 20.2.2020]. Dostupné z: https://aws.amazon.com/marketplace/pp/B07QK2XWNV.*

[73] Shah, *B. Cisco Umbrella: A Cloud-Based Secure Internet Gateway (SIG) On and Off Network.* International Journal of Advanced Research in Computer Science. *Udaipur: International Journal of Advanced Research in Computer Science. 2017, sv. 8, č. 2. ISSN 0976-5697.*

[74] Sheth, *H. Test Automation Using Pytest and Selenium WebDriver [LAMBDATEST]. 2019. [Online; navštíveno 19.09.2019]. Dostupné z: https://www.lambdatest.com/blog/test-automation-using-pytest-and-selenium-webdriver/.*

[75] Skupin, *H. New preference in the upcoming Firefox 65 release [GitHub]. Leden 2019. [Online; navštíveno 12.7.2020]. Dostupné z: https://github.com/mozilla/geckodriver/issues/284#issuecomment-458305621.*

[76] Soyusiawaty, *D. a* Zakaria, *Y. Book Data Content Similarity Detector With Cosine Similarity (Case study on digilib.uad.ac.id). In:* 2018 12th International Conference on Telecommunication Systems, Services, and Applications (TSSA). *IEEE, 2018, s. 1–6. ISBN 978-1-5386-6940-2.*

[77] Spillner, *A.*, Linz, *T. a* Schaefer, *H.* Software testing foundations : a study guide for the certified tester exam : foundation level, ISTQB compilant. *Santa Barbara: Rocky Nook, 2007. Computing. ISBN 1-933952-08-3.*

[78] SPIR. *FAQ – často kladené dotazy [NetMonitor]. 2016. [Online; navštíveno 28.2.2020]. Dostupné z: http://www.netmonitor.cz/sites/default/files/faq_netmonitor_0.pdf.*

[79] SPIR. *O projektu [NetMonitor]. 2016. [Online; navštíveno 27.2.2020]. Dostupné z: http://www.netmonitor.cz/o-projektu.*

[80] SPIR. *O sdružení [SPIR]. 2016. [Online; navštíveno 27.2.2020]. Dostupné z: https://www.spir.cz/o-sdruzeni.*

[81] SPIR. *Ceník NetMonitor platný od 1.1.2019 [NetMonitor]. Leden 2019. [Online; navštíveno 28.2.2020]. Dostupné z: https://www.netmonitor.cz/sites/default/files/prilohy/cenik_NM_1_2019.xlsx.*

[82] Stoically. *WebExtensions API Fake [GitHub]. Leden 2020. [Online; navštíveno 29.5.2020]. Dostupné z: https://github.com/stoically/webextensions-api-fake.*

[83] Timko, *M.* Vylepšení rozšíření pro omezení volání JavaScriptu. *Brno, 2019. Diplomová práce. Vysoké učení technické v Brně. Fakulta informačních technologií.*

[84] Tsvetkov, A., Potapov, V. et al. *Sinon-chrome [GitHub]. Listopad 2019. [Online; navštíveno 29.5.2020]. Dostupné z:* https://github.com/acvetkov/sinon-chrome.

[85] Umbrella, C. *Umbrella Popularity List [Cisco]. 2020. [Online; navštíveno 24.2.2020]. Dostupné z:* http://s3-us-west-1.amazonaws.com/umbrella-static/index.html.

[86] Vaidya, N. *All You Need to Know About Selenium WebDriver Architecture [edureka!]. 2019. [Online; navštíveno 19.09.2019]. Dostupné z:* https://www.edureka.co/blog/selenium-webdriver-architecture/.

[87] West, D. a Walton, S. *Mock Browser [GitHub]. Březen 2017. [Online; navštíveno 29.5.2020]. Dostupné z:* https://github.com/darrylwest/mock-browser.

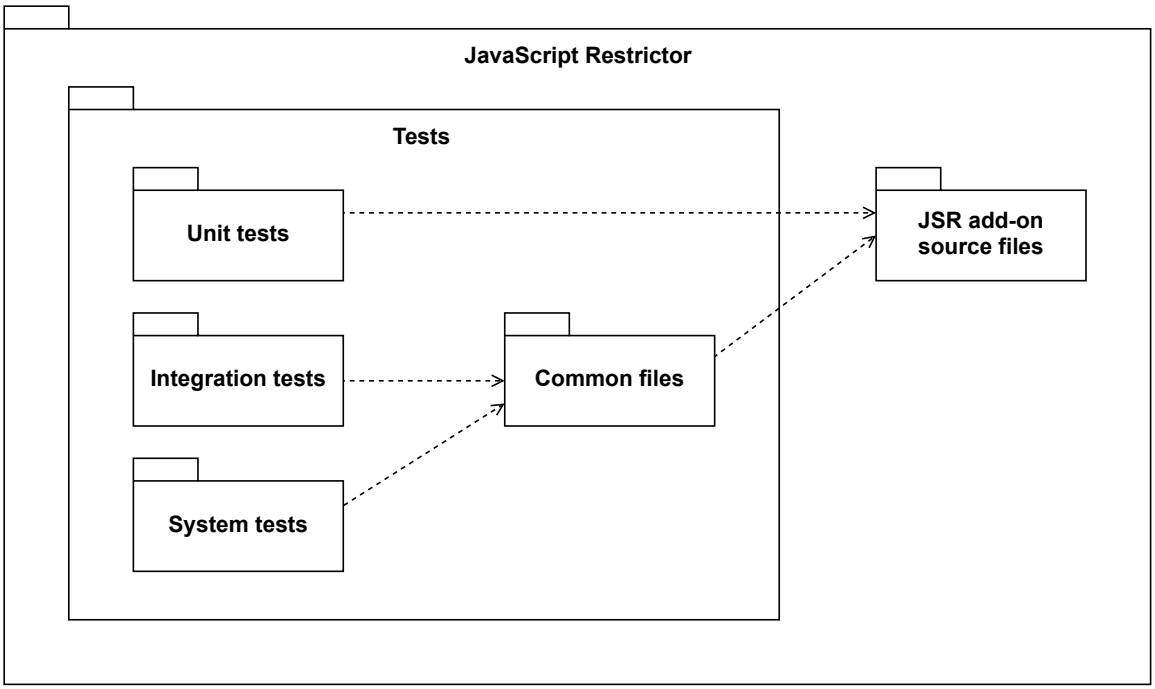
[88] Zanten, B. V. van. *The very first recorded computer bug [TNW]. 2013. [Online; navštíveno 6.1.2020]. Dostupné z:* https://thenextweb.com/shareables/2013/09/18/the-very-first-computer-bug/.

[89] Šnajdr, P. *Blokujte nebezpečný obsah pomocí Cisco Umbrella [ALEF Distribution CZ]. Prosinec 2018. [Online; navštíveno 24.2.2020]. Dostupné z:* https://www.alef.com/cz/blokujte-nebezpecny-obsah-pomoci-cisco-umbrella.c-455.html.

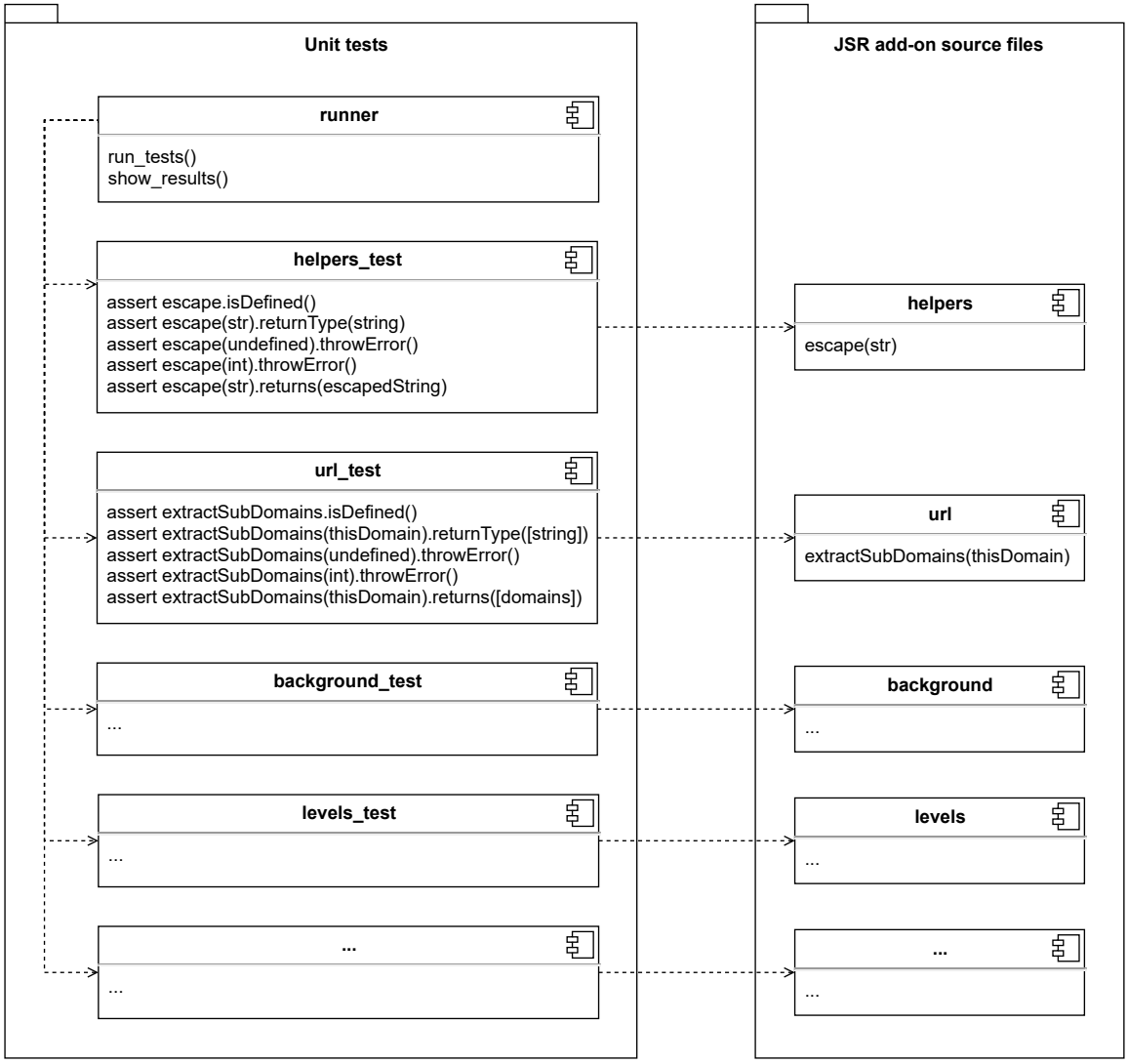
Přílohy

Příloha A

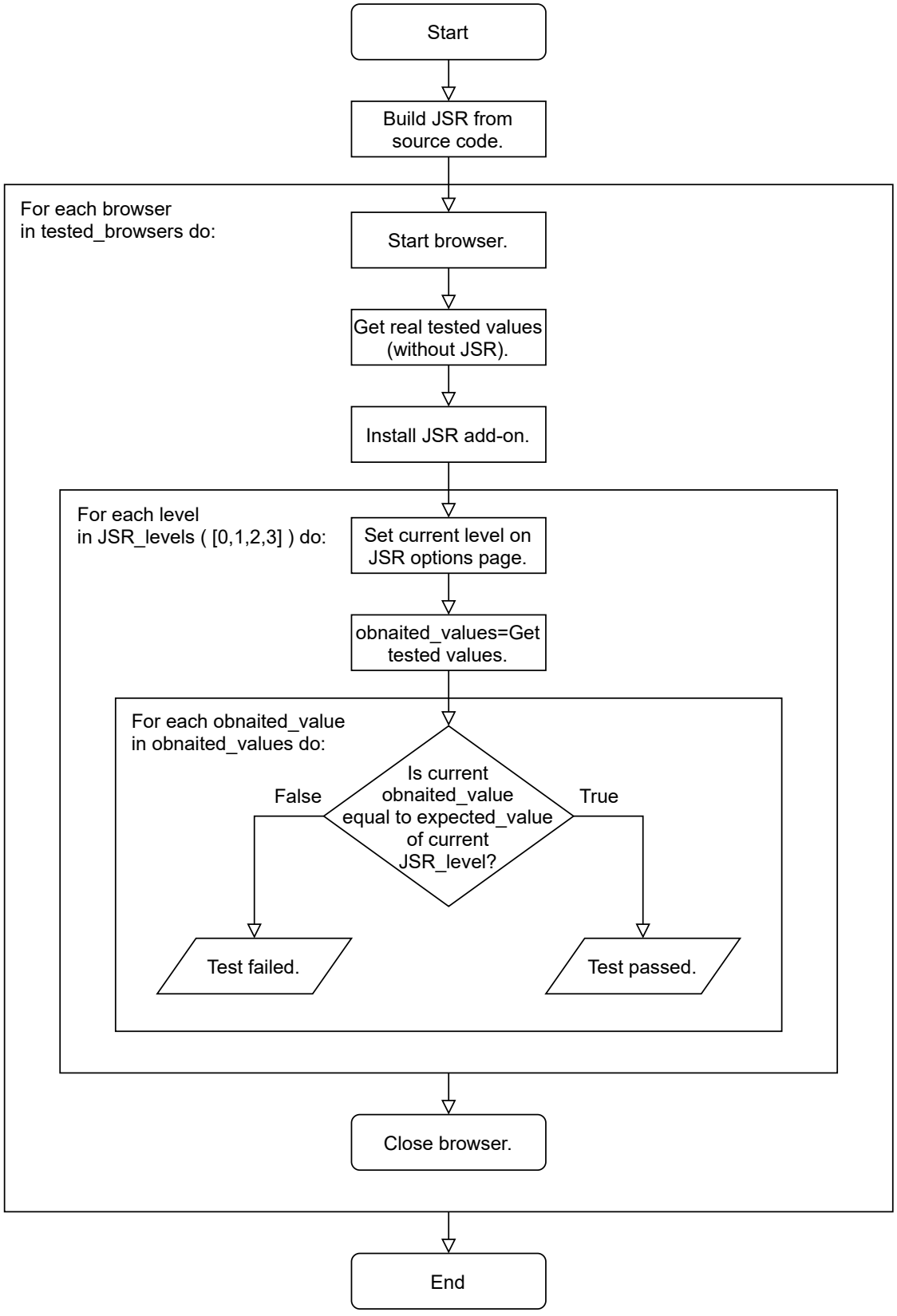
Diagramy k návrhu testů



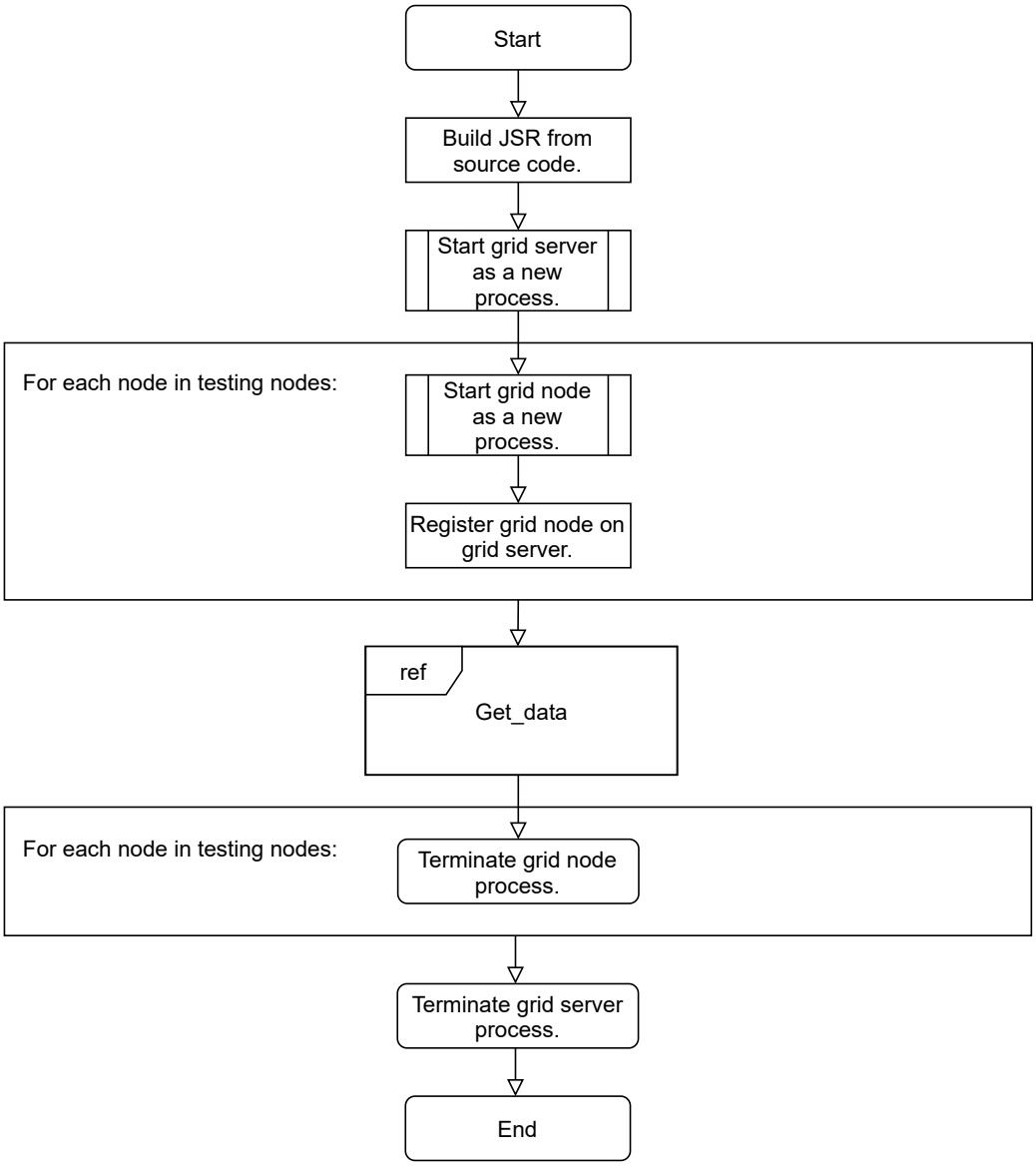
Obrázek A.1: Návrh balíčků, do kterých lze testy rozdělit, a vazby balíčků na zdrojový kód rozšíření JSR.



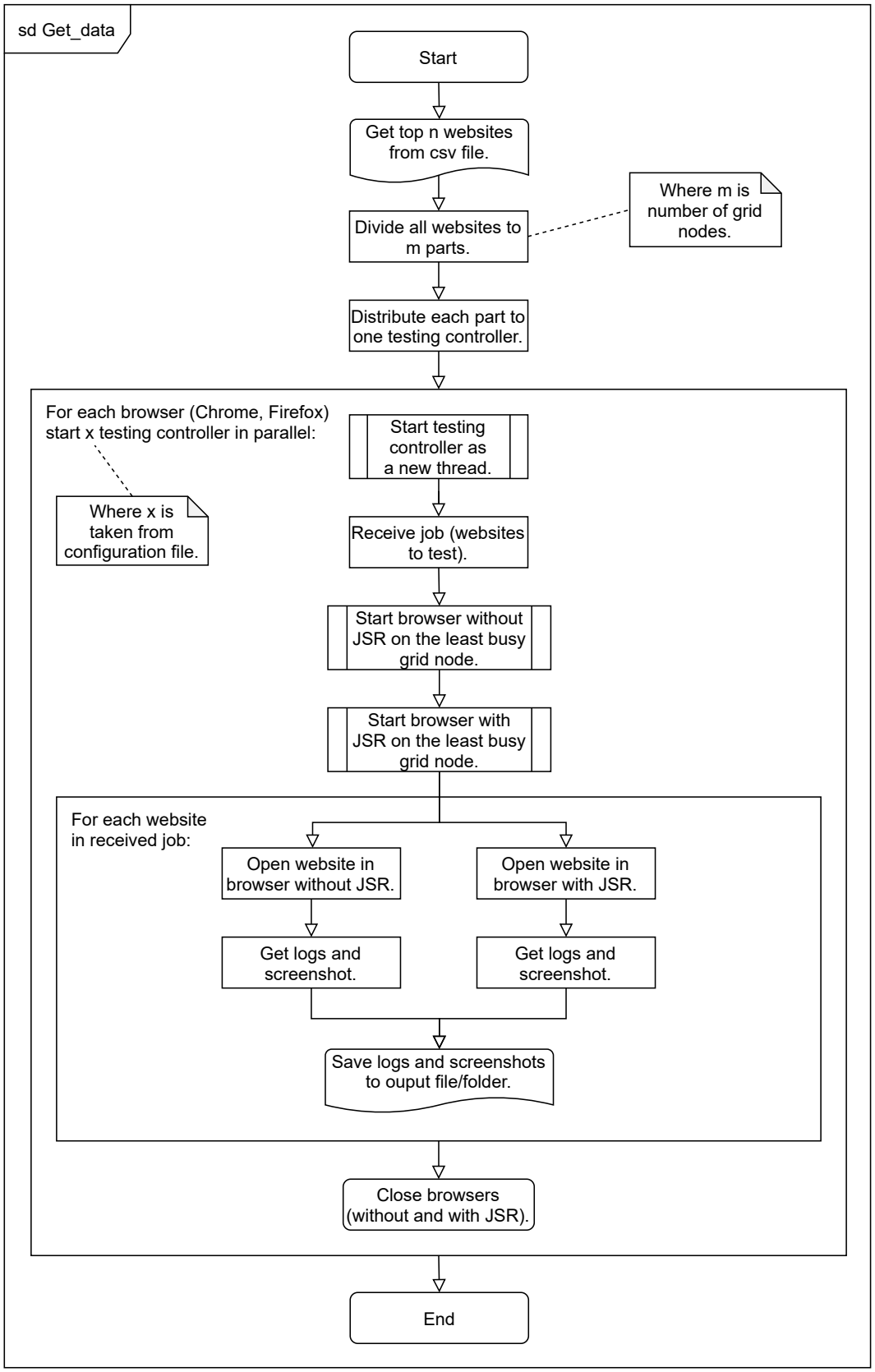
Obrázek A.2: Návrh komponent jednotkových testů a jejich vazeb na zdrojový kód rozšíření JSR. Detail pro vybrané komponenty `helpers_tests` a `url_tests.js`.



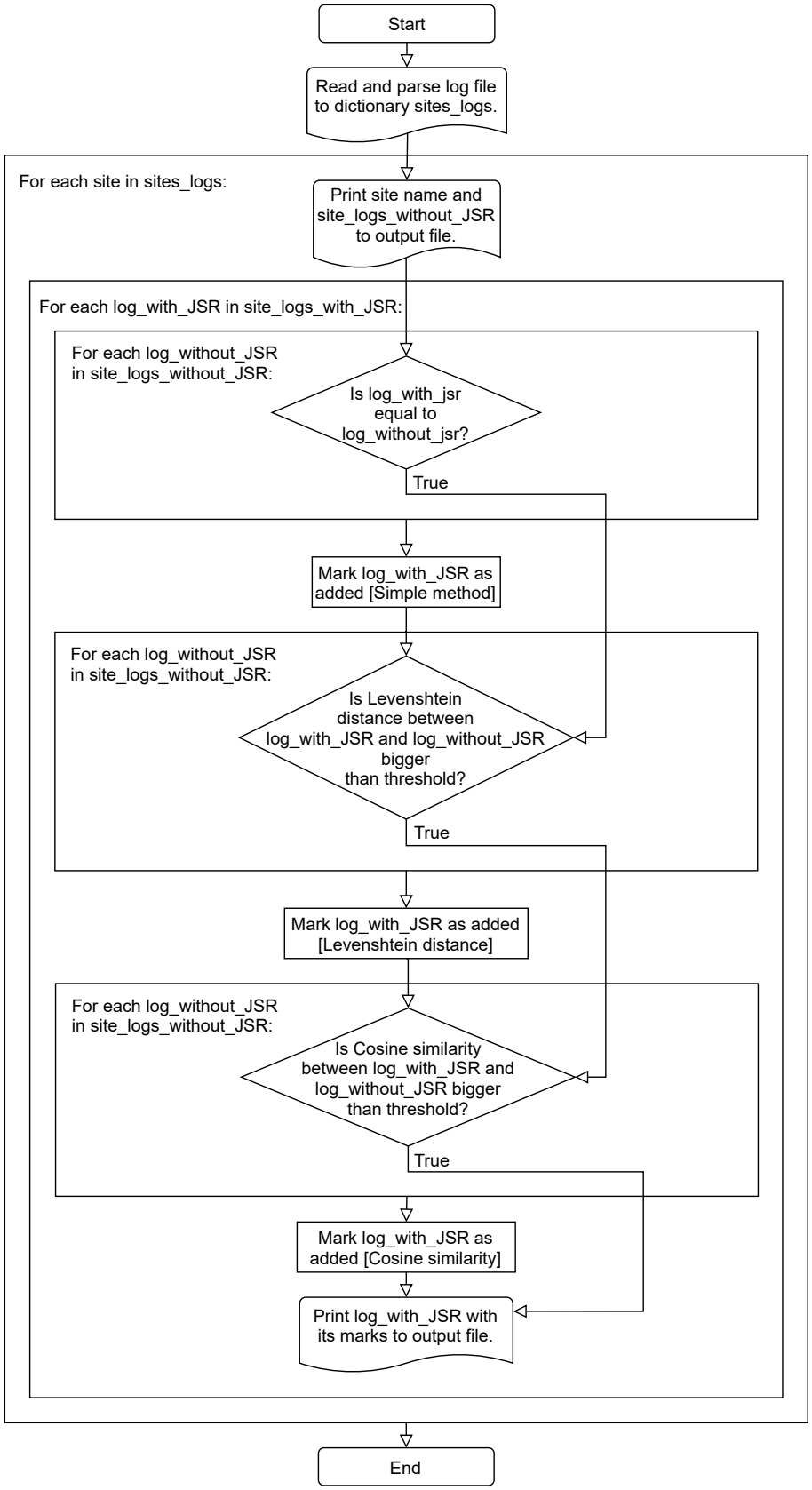
Obrázek A.3: Návrh procesu automatického integračního testování.



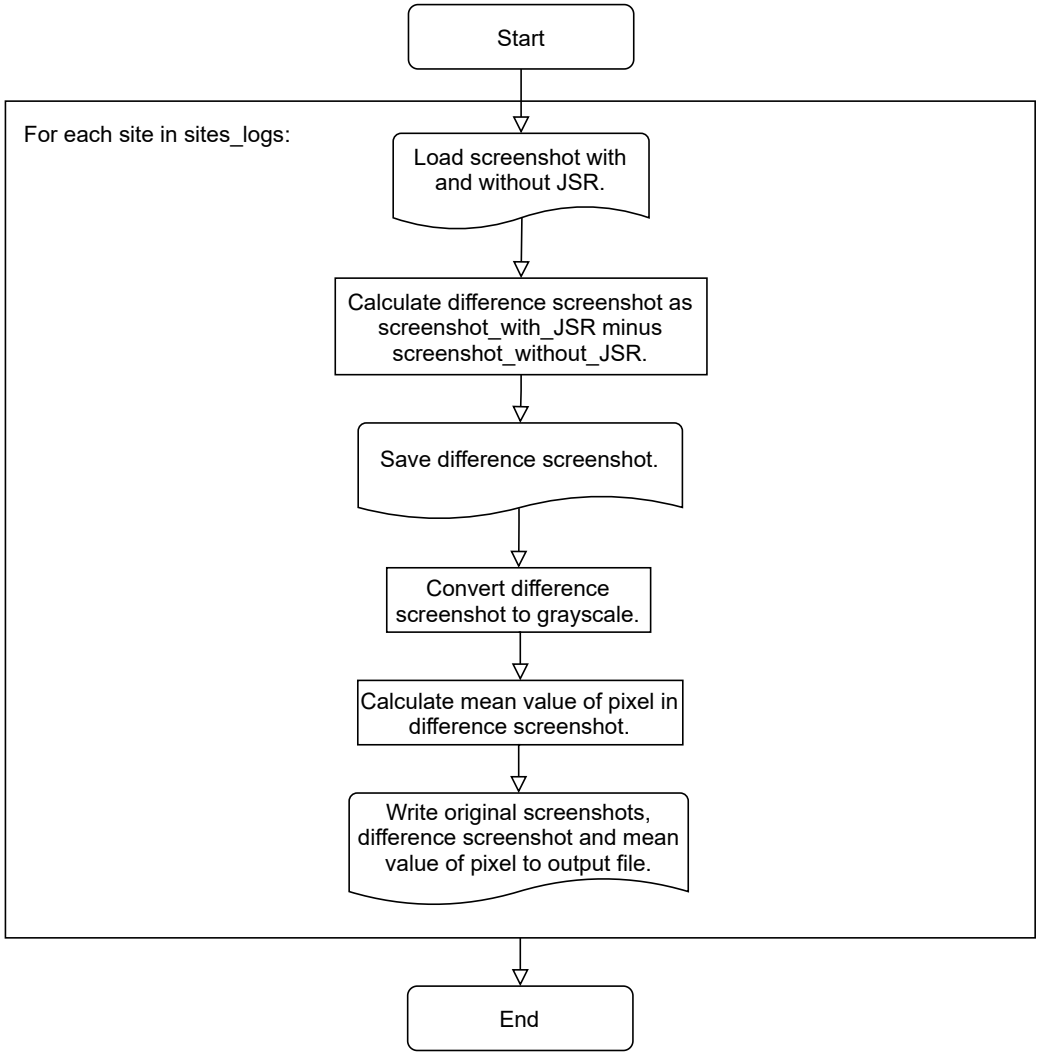
Obrázek A.4: Návrh procesu nastavení testovacího prostředí nástroje Selenium Grid.



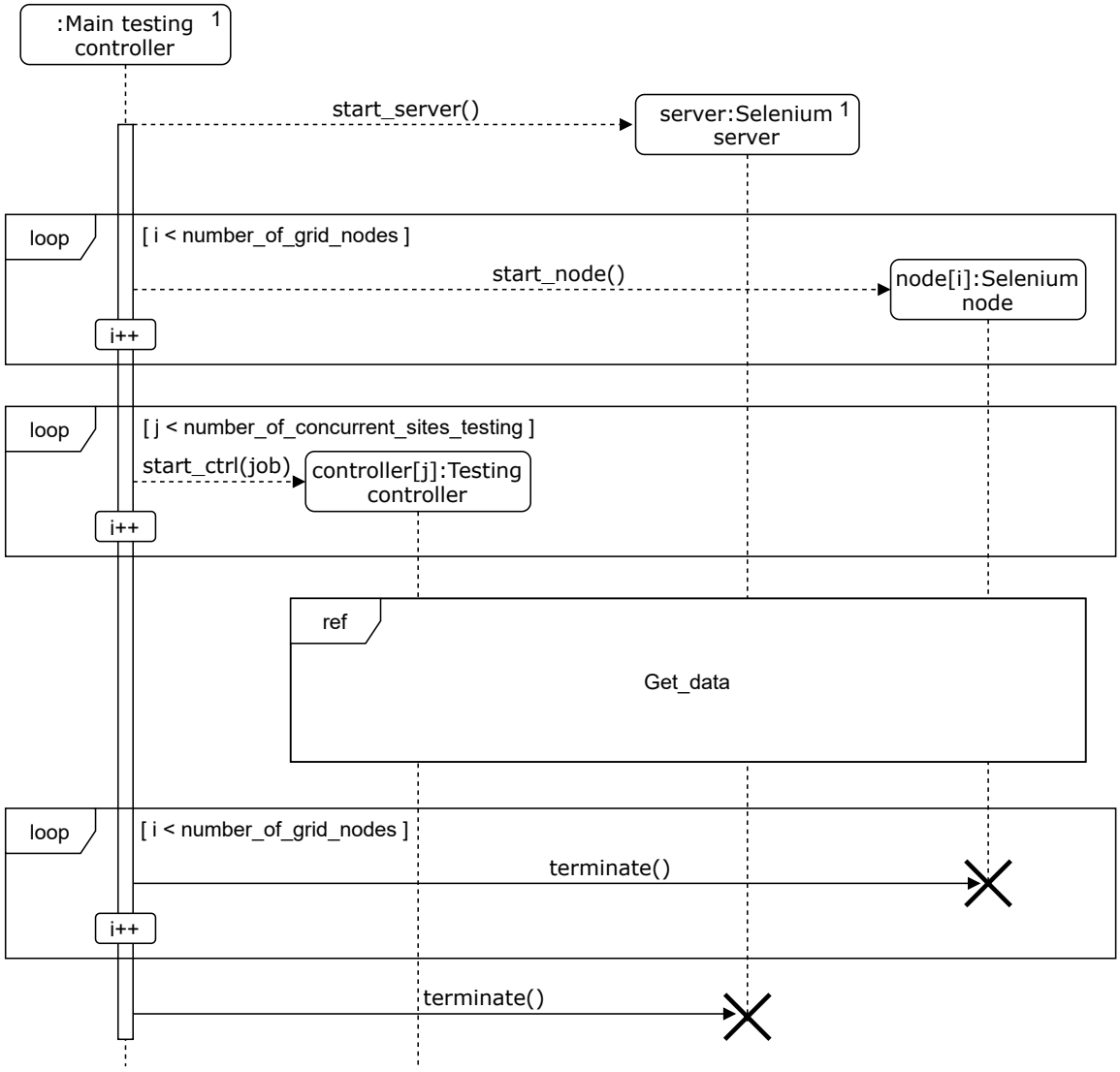
Obrázek A.5: Návrh procesu získání dat testovaných webových stránek.



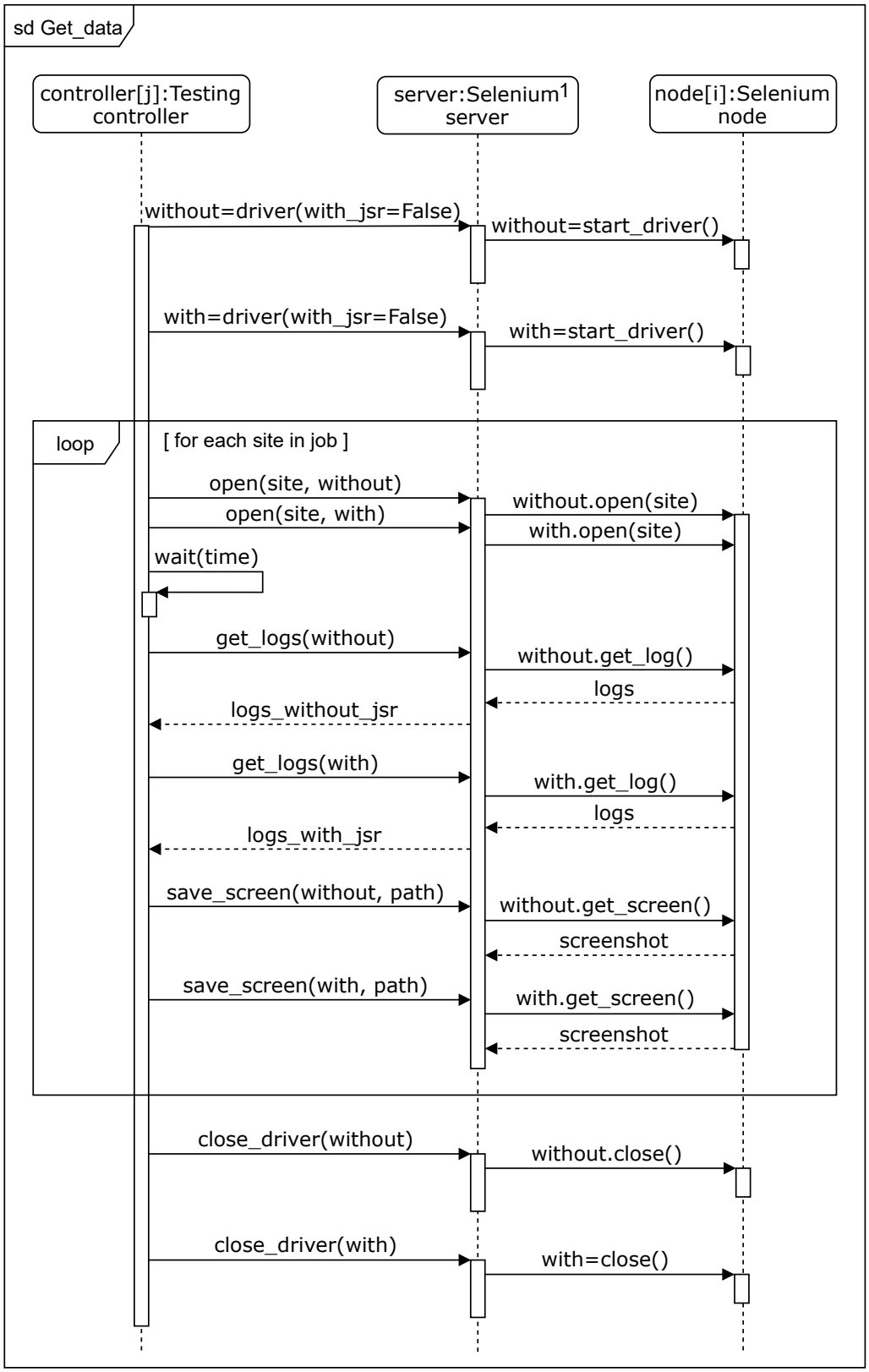
Obrázek A.6: Návrh procesu analýzy získaných záznamů z konzole.



Obrázek A.7: Návrh procesu analýzy získaných snímků obrazovky.



Obrázek A.8: Návrh časově uspořádané interakce mezi objekty při nastavení testovacího prostředí nástroje Selenium Grid.



Obrázek A.9: Návrh časově uspořádané interakce mezi objekty při získání dat testovaných webových stránek.

Příloha B

Obsah přiloženého paměťového média

```
DP_MartinBednar_CD
├── jsrestrictor-devel0.2
│   ├── ...
│   └── tests
│       ├── common_files
│       └── integration_tests
├── jsrestrictor-devel0.3
│   ├── ...
│   └── tests
│       ├── common_files
│       ├── integration_tests
│       ├── system_tests
│       └── unit_tests
├── Technická zpráva
│   ├── DIP_xbedna60.zip
│   └── DIP_xbedna60.pdf
└── Výsledky testů
    ├── JSR devel0.2
    │   ├── integration_tests
    │   └── system_tests
    └── JSR devel0.3
        ├── integration_tests
        ├── system_tests
        └── unit_tests
```

Ve stromu shrnujícího obsah paměťového média jsou explicitně uvedeny pouze ty soubory, jichž jsem autorem. Na paměťovém médiu jsou však uložené celé repozitáře, které obsahují i zdrojové soubory JSR, protože automatické testy jsou na nich závislé.

Na paměťovém médiu jsou v samostatných složkách uloženy dvě verze JSR (verze 0.2 a 0.3) a k nim také odpovídající testy. Systémové testy jsou univerzální pro obě verze.

Paměťové médium obsahuje kompletní výsledky automatických testů. Je tak možné se s výsledky testů pro aktuálně nejnovější verzi JSR seznámit bez nutnosti vlastního spuštění testů.

Příloha C

Manuál pro spuštění integračních testů

Pozn.: Následující manuál je ve formě README souboru přiložen také k integračním testům na paměťovém médiu.

Integration tests for the web browser extension Javascript Restrictor automatically verify that the required JavaScript API is wrapped and conversely, that the non-wrapped JavaScript API provides real values.

It is necessary to partially set up manually a test environment before the first test run!

SET UP TEST ENVIRONMENT

Install required programs and tools

These programs and tools are required to be installed:

- Python 3.5+
- Python package `pytest`
- Python package `selenium`
- Google Chrome – Install really Google Chrome, Chromium is not supported.
- Mozilla Firefox ESR – Be careful, ESR (eventually Developer or Nightly edition) is required. Standard edition is not supported.

No other versions of Google Chrome and especially Mozilla Firefox may be installed on the same machine. If you already have installed Mozilla Firefox Standard Edition, uninstall it before instalation of Mozilla Firefox ESR starts. The web browser driver automatically selects the installed version of the web browser so it is better to have installed only one correct version of each web browser. Web browsers may not have installed the JSR extension. The python script will install it itself before running tests. If you already have installed JSR in a web browser, delete JSR setting from the Options page and then remove JSR extension from the browser.

How to install Mozilla Firefox ESR on Linux

If you have problems with installing Mozzila Firefox ESR on Linux, try this way:

```
sudo add-apt-repository ppa:jonathonf/firefox-esr
sudo apt-get update
sudo apt-get install firefox-esr
```

Or download the archiv with Firefox ESR from the official page, extract the archiv to `/opt/firefox` and create a symbolic link `firefox` in `/usr/bin` pointing to `/opt/firefox/firefox`. In case of problems with installing Firefox ESR, follow this tutorial.

Setup web browsers

Open Mozilla Firefox ESR and change the preference `xpinstall.signatures.required` to `false` in the Firefox Configuration Editor (`about:config` page). You can follow official Mozilla support.

Open the testing page and click on the button *Show GPS data*. Firefox will ask you if you want to enable the page to access the location. Check the option *Remember this decision* and then click *Allow*.

Google Chrome is already prepared in a default state for testing JSR, the Chrome settings do not need to be changed.

Download web browser drivers

Download web browser drivers needed for controlling web browsers by tests. Download drivers for both web browsers - Google Chrome and Mozilla Firefox - and for both platform - Windows and Linux.

For Google Chrome download the ChromeDriver from the download page. Select the version coresponding to the version of your Google Chrome web browser. If you download an incompatible version, you will see an error during starting tests. Download the correct ChromeDriver to the folder `../common_files/webbrowser_drivers` with the name `chromedriver.exe` (for Windows) or `chromedriver` (for Linux).

For Mozilla Firefox download the GeckoDriver from the download page. Select the version coresponding to the version of your Mozilla Firefox web browser (typically the newest version). If you download an incompatible version, you will see an error during starting tests. Download the correct GeckoDriver to the folder `../common_files/webbrowser_drivers` with the name `geckodriver.exe` (for Windows) or `geckodriver` (for Linux).

RUN TESTS

on Windows OS

Open PowerShell in the folder *integration_tests* and run the command:
`.\setup_buildJSR_runTests.ps1`

The script may ask you for the path into the directory, where the file chrome.exe is stored and where the files of Firefox ESR default profile are stored.

The default location of the directory, where chrome.exe is stored, is:
`C:\Program Files (x86)\Google\Chrome\Application`

The default location of directory, where the files of Firefox ESR default profile are stored, is:
`C:\Users\<username>\AppData\Roaming\Mozilla\Firefox\Profiles\<profilename>.default-esr`

If the script does not find needed files into the default locations, it will prompts you to input the path, where the file(s) is/are saved.

When the script execution starts for the first time, OS Windows may ask you to allow Firewall Exception for this script (for Python). Click *Allow*.

on Linux OS

Open Terminal in the folder *integration_tests* and run the command:
`./setup_buildJSR_runTests.sh`

The script may ask you for the the path into directory, where the files of Firefox ESR default profile are stored.

The default location of the directory, where the files of Firefox ESR default profile are stored, is: `/home/<username>/.mozilla/firefox/<profilename>.default-esr`

If the script does not find needed files into the default location, it will prompts you to input the path into the directory, where the files are saved.

Integration tests configuration

You can change selected browsers, which integration tests run in, and you can change JSR levels, which are tested, by modifing the file `./testing/configuration.py`.

Příloha D

Manuál pro spuštění systémových testů

Pozn.: Následující manuál je ve formě README souboru přiložen také k systémovým testům na paměťovém médiu.

System tests for the web browser extension Javascript Restrictor (JSR) automatically checks how JSR affets tested websites.

It is necessary to partially set up manually a test environment before the first test run!

SET UP TEST ENVIRONMENT

Install required programs and tools

These programs and tools are required to be installed:

- Python 3.5+
- Python package `numpy`
- Python package `selenium`
- Visual C++ build tools – required by `python-Levenshtein` on Windows. More information.
- Python package `python-Levenshtein`
- Python package `sklearn`
- Python package `nltk`
- Nltk stopwords – install them by running the command

  ```
  python -m nltk.downloader stopwords
  ```

- Google Chrome - Install really Google Chrome, Chromium is not supported.

Download the Selenium server

Download the Selenium server (Grid) standalone from the download page. Save it to the folder `./get_data/selenium/` with the name `selenium-server-standalone.jar`.

Download the top sites list

Download the latest TRANCO top sites list from the download page. Save it to the folder `./get_data/top_sites/` with the name `tranco.csv`.

Download the Chrome driver

Download the Chrome driver from the download page. Select the version coresponding to the version of your Google Chrome web browser. If you download an incompatible version, you will see an error during starting tests.

Download the correct ChromeDriver to folder `../common_files/webbrowser_drivers` with the name `chromedriver.exe` (for Windows) or `chromedriver` (for Linux).

RUN TESTS

Open the file `./get_data/configuration.py` and check if all paths and other properities are right.

The results of system tests will be stored in the folder `./data` after finishing tests.

on Windows OS

Open PowerShell in the folder *system_tests* and run the command:
`.\setup_buildJSR_runTests.ps1`.

When the script execution starts for the first time, OS Windows may ask you to allow Firewall Exception for this script (for Python). Click *Allow*.

on Linux OS

Open Terminal in the folder *system_tests* and run the command:
`./setup_buildJSR_runTests.sh`