Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačních technologií

Bakalářská práce

Automatizované testování softwaru

Mihail Vozian

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Mihail Vozian

Informatika

Název práce

Automatizované testování softwaru

Název anglicky

Automated software testing

Cíle práce

Hlavním cílem bakalářské práce je vývoj sady automatizovaných testů a vyhodnocení vhodnosti jejich nasazení vzhledem ke zvoleným požadavkům. Dílčí cíle bakalářské práce:
- Analýza dostupných odborných informačních zdrojů
- Návrh testovacích scénářů a specifikace požadavků na testování
- Vyhodnocení vhodnosti a úspěšnosti testů na základě měření ve zvoleném časovém období

Metodika

Teoretická část práce je založena na studiu odborné literatury týkající se automatizovaného testování softwaru. V rámci praktické části bude vypracován základový object model pro vytváření a údržbu automatizovaných testů v programovacím jazyce Scala, automatizované testy budou vytvářeny podle vhodně navržených testovacích případů. Na základě dlouhodobých výsledků testů bude stanovena vhodnost zvolených postupů a formulovány závěry práce.

Doporučený rozsah práce

40-50

Klíčová slova

Automatizované testování, Vývoj softwaru, Webové aplikace, Scala

---

Doporučené zdroje informací

BUREŠ, M. – RENDA, M. – DOLEŽEL, M. – Efektivní testování softwaru, 2016. ISBN 978-80-247-5594-6.
Cocchiaro C. – Selenium framework design in data-driven testing : build data-driven test frameworks using Selenium WebDriver, AppiumDriver, Java, and TestNG, 2018. ISBN 978-1788473576.
Daniel J. Mosley – Bruce A. Posey – Just Enough Software Test Automation, 2002. ISBN 978-0130084682.
Kleijn – Roy de a Gundecha – Unmesh – Selenium Testing Tools Cookbook, 2012. ISBN 9781849515740.
Ron Patton – Software testing, 2002. ISBN 80-7226-636-5.

Předběžný termín obhajoby

2021/22 LS – PEF

Vedoucí práce

Ing. Jan Pavlík

Garantující pracoviště

Katedra informačních technologií

Elektronicky schváleno dne 17. 8. 2021

doc. Ing. Jiří Vaněk, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 5. 10. 2021

Ing. Martin Pelikán, Ph.D.

Děkan

V Praze dne 13. 03. 2022

Čestné prohlášení

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

V Praze dne 15.03.2022 _____

Poděkování

Rád bych touto cestou poděkoval vedoucímu práce Ing. Janu Pavlíkovi, za cenné rady a odborné vedení při zpracování této bakalářské práce.

Automatizované testování softwaru

Abstrakt

Cílem bakalářské práce je vývoj sady automatizovaných testů dle vhodných a logických testovacích scénářů. Teoretická část je zaměřena na základy testování softwaru a možnosti automatizace za použití testovacích nástrojů. Teoretická část se zabývá hlavně nástrojem Selenium a dalšími nástroji automatizace. Praktická část se zaměřuje na implementaci nového projektu, vypracování object modelu, návrhu a implementaci automatizovaných testů. Na základě dlouhodobých výsledků testů bude stanovena vhodnost zvolených postupů a formulovány závěry práce.

Klíčová slova: automatizované testování, vývoj softwaru, webové aplikace, Scala, testování, Selenium, Xpath, jQuery

Automated software testing

Abstract

The purpose of this bachelor thesis is to develop a set of automated tests in accordance with suitable and logical testing scenarios. The theoretical part focuses on the basics of software testing and the possibilities of automation using testing tools. The theoretical part deals mainly with the tool called Selenium and other tools of automatization. The practical part focuses on the implementation of the new project, the development of an object model, design and implementation of the automated testing. Based on the long-term test results, the suitability of the chosen procedures will be set, and the conclusions of the thesis will be formulated.

Keywords: automated testing, development of software, web app, Scala, testing, Selenium, Xpath, jQuery

Obsah

Seznam obrázků

1 Úvod

Firem vyvíjející softwarový produkt by v dnešní době měly mít sekci manuálního testování. Testování je nezbytnou součástí vývoje produktu, zejména pro zajištění kvality. Mnoho společností dává přednost manuálnímu testování, i když to není nejlepší a nejefektivnější přístup, zejména z důvodů vytváření složitějších, komplikovanějších a všeobecně komplexnějších propojení aplikací, u kterých se mnohonásobně zvyšují nároky na kvalitu a rychlost testování.

Manuální testování by mělo být co nejvíce minimalizováno, a proto dalším logickým krokem je automatizované testování jako jeden ze způsobů zvýšení efektivity, kvality a rychlosti provádění testů. Automatizované testování by mělo firmě ušetřit peníze, protože na testování produktu je vynaloženo méně prostředků. Do budoucna bychom v ideálním případě neměli provádět žádné manuální testování. V průběhu celého projektu to pak může znamenat velký rozdíl, jak ze strany financí, tak ze strany časové efektivity a kvality otestování.

Manuální testování omezuje počet testů, které můžeme ověřit. Díky automatizaci bychom měli trávit více času psaním nových testů a jejich přidáváním do již existujících sad. To zvyšuje testovací pokrytí produktu, takže je řádně otestováno více funkcí. Automatizované testování nám otevírá i možnost psát komplexní testovací případy, kde můžeme psát zdlouhavé testy, kterým se při ručním testování vyhýbáme. Jednou z mnoha výhod automatizace je to, že běh testů můžeme provádět bez dozoru a dokážeme je použít opakovaně. I ten nejlepší testovací technik může při manuálním testování udělat chybu. Zejména pak při testování složitého případu je více než pravděpodobné, že bude docházet k chybám. Na druhou stranu automatizované testy dokážou provádět testy se 100% přesností, protože při každém spuštění poskytují stejný výsledek. Automatizované testování nám poskytuje i více funkcí, například simulaci tisíců uživatelů, kteří nějakým způsobem interagují s webovou aplikací, abychom viděli, jak se aplikace v daném okamžiku chová. Manuálním testováním je naopak zcela nemožné tento druh chování simulovat.

Cílem této práce je tedy vývoj testovacích sad automatizovaných testů podle navržených testovacích scénářů, které dokážeme tvořit i manuálně. Na základě výsledků se budeme snažit ověřit efektivitu a to, jakých rozdílů mezi manuálním a automatizovaným testováním dokážeme docílit, a to jak z pohledu časové efektivity, tak i z pohledu kvality.

2 Cíl práce a metodika

2.1 Cíl práce

Hlavním cílem bakalářské práce je vývoj sady automatizovaných testů dle navržených logických testovacích scénářů. Na základě dlouhodobých výsledků automatizovaných testů bude stanovena jejich vhodnost, účinnost, pracnost a vyhodnocení časové návratnosti.

Vedlejším cílem bude vytvořit projekt a vypracovat základový object model pro vývoj a údržbu automatizovaných testů v objektově orientovaném programovacím jazyce Scala.

2.2 Metodika

Teoretická část se bude zabývat základy testování a vysvětlení výhod a úskalí automatizovaného testování. Na základě informací získaných z odborné literatury, která se zabývá testováním softwaru, budou dosažené znalosti uplatněny k vytvoření vhodných testovacích scénářů. V souladu se scénáři budou automatizované testy implementovány do základového object modelu, který bude sloužit pro vývoj i údržbu všech testovacích sad.

3 Teoretická východiska

3.1 Testování softwaru

Testování hraje velmi důležitou roli při dosahování kvality softwaru. Na jedné straně zlepšujeme kvalitu softwaru a na druhou stranu zjišťujeme, jak dobrý náš systém je, když se během vývoje produkt testuje. Testování je proces ověřování, hodnocení a zlepšování kvality softwaru (Naik and Tripathy, 2008). Testování je založené na úzké spolupráci vývojového týmu a testerů, na porozumění nejen samotnému produktu, ale hlavně zákazníkovi, pro kterého je produkt vytvářen. Za vzájemné spolupráce se tým snaží vyvinout kvalitní produkt, který bude zákazníkovi přinášet prospěch. Na úplném počátku vývoje je potřeba stanovit daný cíl a měřítka kvality, zvolit takový přístup a připravit implementaci, abychom měli jistotu, že nejen samotný výsledný produkt, ale i celý proces jeho vývoje dosahuje požadované kvality. Nejprve se musí stanovit záběry testování, podle nich se pak vybírají testy, sbírají data a také se určí to, jsou-li potřeba další nástroje, které tým k testovaní potřebuje. Je třeba zkontrolovat, zda jsou všechny požadavky na produkt v takovém stavu, aby bylo možné ověřit jejich jednoznačné splnění. Testování přináší mnoho výhod, které organizacím ušetří čas a peníze, protože síla testování spočívá v tom, že dokážeme zjistit chybu dříve, než se produkt dostane k samotnému zákazníkovi. Produkt pak zákazníka naplňuje uspokojením, že vše funguje tak, jak má, bez jakýchkoliv chyb, a sám zákazník projevuje produktu určitou důvěru a vytváří si k němu dobrý vztah. Samotné testování i přes stanovení veškerých cílů naráží na mnoho omezujících překážek (*Naik and Tripathy, 2008*).

Další problémy, se kterými se běžně setkáváme, jsou:

- Ve většině případů není možné otestovat veškeré možné případy zadání.
- Pokud se ve specifikaci na něco zapomene, v pokročilejší fázi vývoje je složité tuto chybu dohledat.
- U některých prvků není specifikace jednoznačná nebo může chybět.
- Každé testování musí být cíleno na jeden určitý produkt, nemůžeme vícero produktů testovat stejným způsobem.
- V průběhu vývoje se může stát, že se některé testy stanou neefektivními. Proto je nutné testy přizpůsobovat podle toho, jak se samotný produkt mění a rozvíjí.
- Tři odlišné chyby se navenek mohou projevovat stejně, nebo se také jedna chyba může projevovat několika způsoby.

K zajištění dobré kvality softwaru však nestačí jen vyhledávat chyby. Na první pohled to nemusí být patrné, ale nalezením chyby začíná další důležitá práce testera, a to je vypracování důkladného a přehledného reportu, který je pak předán vývojáři. Čím lépe je report vypracovaný, tím lépe a rychleji vývojář pochopí, o jaký problém se jedná (*Naik and Tripathy, 2008*).

3.1.1 Termíny používané v testování

Bug – chyba softwaru, která může být definována jako chyba zdrojového kódu, jež způsobuje nečekaný defekt, závadu nebo chybu. Jinými slovy, pokud program nefunguje tak, jak bylo zamýšleno, jedná se s největší pravděpodobností o chybu.
Error – neshoda mezi programem a specifikací, která vede k chybě v programu.
Defect – defekt je odchylka určitého atributu v produktu, mohou to být nesprávné, chybějící nebo doplňující údaje. Může se jednat o dva typy defektů: jedním z nich je vada na výrobku, druhým pak odchylka od očekávání zákazníka. Jedná se o chybu, která nemá žádný dopad, dokud neovlivní zákazníka.
Failure – selhaní je vada, která způsobuje chybu v provozu. Negativně ovlivňuje zákazníka, nebo některého z uživatelů.
Quality Assurance – zajišťování kvality softwaru hraje velkou roli ve zlepšení celkového výkonu procesů. Hlavním úkolem je zajistit, abychom docílili toho, aby všichni účastníci zabývající se projektem dodržovali postupy a standardy. Zároveň zajišťuje shromažďování různých softwarových opatření, které jsou nezbytně nutné k hodnocení standardů, procesů a postupů, které jsou v organizaci implementovány.
Quality Control – kontrola kvality nebo řízení jakosti je soubor opatření, který se snaží zajistit, aby nebyly vyráběny vadné služby a aby projekt splňoval požadavky na výkon (*Java T Point, 2022*).

3.1.1.1 Verifikace a validace

Verifikace a validace jsou pojmy, které se v softwarovém testování vyskytují velmi často. Mnoho lidí tyto pojmy zaměňuje, nebo neví, co přesně znamenají.

Verifikace je proces, který slouží ke zjišťování možných selhání softwaru ještě před zahájením testovací fáze. Celý tento proces zahrnuje kontroly, inspekce, schůzky, kontroly kódu a specifikace. Verifikace většinou probíhá bez zadavatele. Zjišťujeme, zda vyvíjíme produkt správně.

Validace je proces, který probíhá až po verifikaci. Validace probíhá se zadavatelem, zjišťujeme, zda softwarový produkt skutečně splňuje potřeby zákazníka, nebo ne. Tento proces nám pomáhá zjistit, jestli software splňuje požadované použití ve vhodném prostředí. Zjišťujeme tedy, zda vyvíjíme správný produkt (*Naik and Tripathy, 2008*).

3.1.2 Kvalita softwaru

Kvalitu softwaru chápeme každý jinak, pro různé lidi může kvalita znamenat různé věci, a bývá to velmi závislé na kontextu. Kvalita softwaru je vnímána různými způsoby v různých oblastech a oborech. Kvalitu softwaru můžeme rozdělit do pěti úhlů pohledu, a to díky Kitchenhamovi a Pfleegerovi, kteří ilustrují práci založenou na rozboru vnímání kvality softwaru:

1. „*Transcendentální pohled: Díváme se na kvalitu jako na něco, co můžeme rozpoznat, ale je pro nás těžké to definovat.*
2. *Pohled uživatele: Uživatel vnímá kvalitu jako produkt, který splňuje potřeby a očekávání uživatelů.*
3. *Výrobní pohled: Uživatelé zkoumají produkt zvenčí, vnímají kvalitu jen do jisté míry, a to, jestli se shoduje s rozsahem požadavků. Koncept výrobního modelu je založený na úvaze, že produkty mají být vyráběny ihned ze začátku, aby se náklady na vývoj a údržbu snížily.*
4. *Pohled na produkt: Jedná se o pohled na produkt z pohledu samotné konstrukce. To znamená, že posuzujeme vlastní vlastnosti produktu, jako jsou vnitřní a inherentní vlastnosti.*
5. *Pohled na základě hodnoty: Tento pohled vnímáme jako rovnocennou kvalitu za to, co je zákazník ochoten zaplatit“* (*B. Kitchenham and S. L. Pfleeger, 1996*).

3.1.3 Spolehlivost softwaru

Spolehlivost softwaru je definována jako pravděpodobnost bezporuchového provozu softwarového systému pro zadaný čas v určitém prostředí. Úroveň spolehlivosti závisí na vstupech, které způsobují, že koncoví uživatelé mohou zpozorovat selhání softwaru. Spolehlivost softwaru je možné odhadnout pomocí náhodného testování. Spolehlivost je specifická pro určité prostředí, je třeba čerpat testovací data přímo z distribuce vstupů, aby se nejvíce podobala budoucímu využití systému (*Naik and Tripathy, 2008*).

3.1.4 Chyby

Softwarová chyba nebo také *bug* je brána jako vada softwarového programu, který může vracet nesprávné výsledky, nebo brání samotnému programu v korektním běhu. V softwarovém testování jakýkoliv *bug* nemusí být brán jako chyba programu. Má se za to, že cokoli, co ovlivňuje kvalitu softwaru, je bráno jako chyba. Chyba samotná se může skrývat pod různými jmény, jako například: závada, problém, incident, porucha nebo rozptyl. Pokud se software chová jinak, než je psáno v samotné specifikaci, ve většině případů se jedná o chybu. Sem patří například i případy, pokud software neodpovídá, nebo softwaru není možné porozumět, či má těžkopádné kroky, nebo pomalé reakce. Všechny tyto příklady jsou brány jako softwarová chyba (*Naik and Tripathy, 2008*).

3.1.4.1 Druhy chyb

Když různé skupiny vývojářů pracují na vývoji jednotného programu, vznikají softwarové chyby, které se vyskytují bez ohledu na velikost programu. Většina těchto chyb se pak dá rozdělit do určitých skupin:

1. Matematické chyby: Dělení nulou, přetečení nebo podtečení, nesprávné zaokrouhlení, nebo zkrácení desetinných hodnot.
2. Logické chyby: Použití špatné logiky, nekonečné smyčky, nebo nekonečná rekurze, špatné použití podmínky nebo také přerušení.
3. Chyby zdrojů: Přetečení vyrovnávající paměti, narušení přístupu, použití proměnných, které nebyly nikdy inicializovány.

4. Chyby společného programování: Většinou se jedná o chyby souběžnosti, dva vývojáři pracují na stejném kódu a při nasazení daného kódu se budou navzájem blokovat.
5. Chyby týmové práce: Není napsaná dokumentace k danému kódu, má zastaralé komentáře, neshoduje se s dokumentací nebo se neshodují samotné soubory, které mohou být neaktuální a zastaralé, dochází propojení programu s nesprávnými soubory.

Časté faktory, kvůli kterým v programu dochází k mnoha chybám:

1. Lidský faktor: Osoba, která vyvíjí samotný software, je „jen“ člověk, a tak vždy existuje riziko, že může dojít k nějaké chybě. Software je náchylný k chybám od té doby, kdy celý proces vývoje zahrnuje lidi.
2. Špatná komunikace: Pokud lidé nedostatečně komunikují, nebo pokud jsou v různých fázích vývoje podávány nesprávné informace, dochází k nedostatečnému informování týmu o požadavku, návrhu a úpravách – to vše může vést k chybám.
3. Nerealistický časový rámec vývoje: Někdy se musí určité softwary vyvíjet mnohonásobně rychleji, kvůli splnění požadavku na dodání, nebo kvůli samotným zdrojům, které mohou být nedostatečné. To však má negativní vliv jak na samotný vývoj, tak i na testování. Proto se pak musejí dělat různé kompromisy, jako například změny designu softwaru či nedostatečné otestování softwaru.
4. Špatný kód: Špatné postupy v kódu, jako jsou překlepy v klíčových slovech, nesprávné ověření proměnných, nebo inicializování proměnných, které se nakonec ani nepoužívají a jen samotný produkt zpomalují. Některé editory a kompilátory nám mohou v kódu způsobovat problémy, protože kvůli nim může být kód špatně čitelný.
5. Nedostatek kvalifikovaných testerů: Pokud v procesu testování softwaru existují nějaké nevýhody, pak jednou z nich jsou určitě nedostatečně kvalifikovaní testeři, který nemají dostatečné zkušenosti, a hrozí, že mnoho chyb přehlédnou.
6. Změny na poslední chvíli: U změn na poslední chvíli záleží na jejich rozsahu a na tom, jestli je možné během krátké chvíle otestovat veškeré funkcionality, kterých se změna mohla dotknout. Takové změny můžou vést k vážným problémům

v aplikaci, pokud se nestihne dostatečně otestovat (*Software Testing Help Types of software Errors, 2022*).

3.1.4.2 Cena chyb

Jakákoliv chyba, která je součástí softwaru a dostane se k zákazníkovi, může pro organizaci znamenat značné finanční ztráty. Může dojít i ke ztrátě reputace softwaru nebo i samotné organizace, což může vést k tomu, že se zákazník rozhodne pro jiný software a přejde ke konkurenci. Pokud by jakákoliv chyba způsobila újmu zákazníkovi, nebo majetku zákazníka, může to vést k ohromným pokutám pro organizaci, která daný software vyvinula. Každá chyba, která se během testování najde, bude mít za následek narušení časové osy vývoje a zvýšení nákladů, jelikož bude nějakou dobu trvat, než se chyba opraví. Pokud jsou chyby detekovány a opraveny již ve fázi vývoje softwaru, může to mít za následek vyšší návratnost investic s ohledem na čas i náklady. Náklady na opravu chyby se liší podle fáze vývoje.

1. Fáze požadavků: Nalezení chyb v požadavcích nebo přepsání některých požadavků může organizaci stát nějaký čas, v této fázi jsou ale chyby snadno dohledatelné a opravitelné.
2. Fáze kódování: Vývojář stráví opravou nějaký čas, který se bude lišit podle závislosti a složitosti chyby. Většinou se chyba přiděluje vývojáři, který má danou sekci na starost a měl by v této fázi problém snadno vyřešit, protože kód psal on sám.
3. Fáze integrace: Vyřešení problému v takové fázi bude vyžadovat více času, protože se jedná o vyšší úroveň, která vyžaduje čas na kontrolu části kódu nebo konfigurace, jež může být špatná a zabere další čas na opravu. Na vývoji v této fázi se podílejí vývojáři a další systémoví inženýři.
4. Fáze testování: Nalezení chyby v této fázi znamená, že na vyřešení chyby se musí podílet tester, vývojář, systémový inženýr a projektový manažer. Je to iterační proces, který vyžaduje mnoho času a úsilí za účelem opravy chyby. Zvýší se náklady na projekt z hlediska lidské síly a způsobí celkové zpoždění dodacích lhůt.
5. Produkční fáze: Nalezení chyby v produkci se považuje za nejzávažnější. Organizaci to stojí čas a investice, ve srovnání s chybou zjištěnou při testování to vyžaduje stanovení priorit a podrobné plánování etap. Organizace musí zařídit, aby se chyba odstranila za co nejkratší možnou dobu, čím déle je chyba v produkci, tím

větší ztráty to pro organizaci může znamenat, samozřejmě záleží na závažnosti chyby. To ale nemění nic na tom, že tyto chyby zhoršují i pověst společnosti (*Codes Kitchen Cost of fixing vs preventing, 2022*).

3.1.5 Úrovně testování a typy testů

Jedním z počátečních problémů, se kterým se setkáváme, je to, jaký způsob testování vlastně zvolit. Na počátku se musíme rozhodnout, jaké druhy testů budou použity, kdy a do jaké míry budou aplikovány a jaké nástroje a data je potřeba připravit. Rozsah možných testů je velmi široký, proto je třídíme do kategorií podle různých hledisek (*Naik and Tripathy, 2008*).

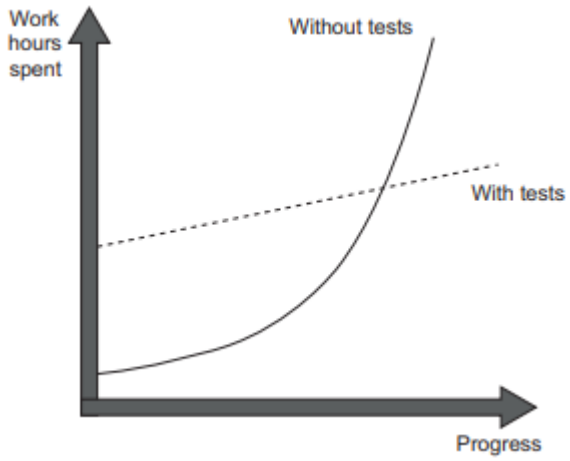
3.1.5.1 Statické a dynamické testování

Statické testování je bráno jako analýza softwaru, která se provádí bez jeho spuštění. Jedná se o techniku bílé skříňky, kdy vývojář kontroluje zdrojový kód, aby v něm nalezl chyby. Provádí se tedy statické testování, které kontroluje hlavně správnost a logiku kódu, zatímco dynamické testování kontroluje odezvu systému na předdefinované vstupy. Celý proces testování statických testů je velmi únavný, protože každý řádek kódu musí kontrolovat sám vývojář. Pro řešení tohoto problému bylo vyvinuto mnoho nástrojů, které umožňují vývojářům, aby prováděli statické testování mnohonásobně rychleji. Aby se v softwaru vyskytovalo co nejméně chyb, je důležité provádět statické i dynamické testování (*Naik and Tripathy, 2008*).

3.1.5.2 Unit testy

Cílem unit testů je umožnit udržitelný růst softwarového projektu. Softwarový projekt je snadno udržitelný, pokud začínáme od nuly, ale mnohem těžší je udržet tento růst spolu s rozvíjejícím se projektem. Unit testy hlídají formu kódu, který má tendenci se zhoršovat. Pokaždé, když se v kódu něco změní, pravděpodobnost množství nepořádku se zvyšuje. Postupem času se může stát, že opravou jedné chyby v kódu se zavádí více chyb a je to jako dominový efekt: místo jedné chyby musí programátor opravovat několik chyb. Unit testy pomáhají tuto tendenci zvrátit, pomáhají zajistit funkčnost kódu. Do unit testů jsou implementovány jednotlivé jednotky, do kterých se snažíme posílat různé variabilní vstupy, a chceme ověřit, zda získáme očekávané výstupy. Na grafu 1 můžeme vidět rozdíl růstu projektu, kdy na jedné straně máme unit testy, a na druhé je nemáme. Projekt bez testů má zprvu náskok,

ale pak rychle zpomaluje, a to až do té míry, že další pokrok je velmi pomalý, a nakonec je těžké nějaký další krok vůbec vytvořit (*Khorikov, 2020*).



Obrázek č. 1 – Projekt s unit testy vs. bez unit testů v závislosti na růstu projektu (Khorikov, 2020)

3.1.5.3 Integrační testy

Ověření softwarových komponent izolovaně je důležité, ale stejně důležité je zkontrolovat, jak tyto komponenty fungují v integraci s externími systémy. V projektu máme unit testy a všechny nám úspěšně projdou, ale aplikace stále nefunguje, v tuto chvíli přicházejí na řadu integrační testy. Musíme ověřit, jak se různé části integrují navzájem s externími systémy, např. databáze a sběrnice zpráv. Integrační testy se provádějí proti modulům (vnější část aplikace). Pokud máme aspoň dva moduly, které nám tvoří podsystém, můžeme použít integrační testy. V integračních testech se využívají falešné moduly. Jde o simulaci komunikace ostatních modulů, ale bez jejich funkčnosti. Díky tomu můžeme ověřit, zda modul umí správně odesílat i přijímat komunikaci proti dalším modulům (*Khorikov, 2020*).

3.1.5.4 Systémové testy

Testování systému je považováno za proces zkoumání funkčnosti systému a identifikace poruch v systému. Během tohoto testování je aplikace testována jako funkční celek. Takové testy používáme v pokročilejších fázích vývoje. Ověřujeme aplikace z pohledu zákazníka. Simulujeme různé akce a kroky, které mohou v praxi nastat. Obvykle tyto testy probíhají v několika kolech. Pokud se najdou nějaké chyby, tak se musejí opravit a v dalších kolech jsou tyto scénáře znovu testovány. Testování se neomezuje jen na funkční testy, ale také na

nefunkční testy. Toto testování se provádí před předáním produktu zákazníkovi (*Ammann and Offutt, 2008*).

3.1.5.5 Akceptační testy

Akceptační testy provádí tester, který bývá pověřený klientem. Má za úkol ověřit, zda software splňuje klientovy požadavky. Klient si obecně vyhrazuje právo odmítnout převzetí produktu, pokud produkt neprojde akceptačními testy. Kritéria přijetí musí být definována a dohodnuta mezi dodavatelem a zákazníkem. Zákazník může pověřit firmu třetí strany, která může navrhnout plán akceptačních testů. Akceptační testy pak bývají součástí smlouvy. Nejedná se pouze o ověření použitelnosti softwaru, ale také o ostatní služby, jako jsou dokumentace, manuály, nebo také řešení celého návrhu softwaru; tester pak může vyhodnotit přijatelnost návrhu softwaru z hlediska grafického uživatelského rozhraní, zpracování chyb a řízeného přístupu (*Naik and Tripathy, 2008*).

3.1.5.6 Black box

Testování černé skříňky, označované také jako testování chování, kontroluje, zda software funguje podle požadavků nebo specifikace. Důvod, proč se tomuto procesu říká testování černé skříňky, je ten, že tester provádí testy bez znalostí vnitřní logiky toho, jak je software naprogramovaný nebo jak přesně funguje. Soustředí se pouze na výstupy, které jsou generované v reakci na vybrané vstupy, a popřípadě další podmínky provedení. Velmi důležitý je vývoj testovacích případů. Tester nemá žádné znalosti interního fungovaní programu, musí se zcela spolehnout na analýzu a transformaci vstupů na výstupy, na jejichž základě pak vyvodí, zda se jedná o chybu softwaru, či nikoliv. Testování černé skříňky nám pomáhá ověřit odpověď na otázku, zda stavíme správný software.

Výhody:

- Tester nemusí rozumět internímu fungování softwaru, testy lze snadno vytvořit.
- Testeři se zabývají hlavně výstupem grafického uživatelského rozhraní, a tím se ušetří časová analýza vnitřních rozhraní. Proto lze testovací případy vyvíjet snadno a rychle.
- Už po dokončení specifikace produktu je možné navrhnout testovací případy.
- Pomáhá odhalit jakékoliv nejasnosti nebo nesrovnalosti ve specifikacích.

Nevýhody:

- Tester může testovat jen menší počet možných vstupů, je nemožné otestovat veškeré možné scénáře.
- Pokud specifikace není jasná a stručná, je velmi obtížné navrhnout testovací případy.
- Pokud dochází ke špatné komunikaci mezi testerem a vývojářem, může se stát, že tester není dostatečně informován, a mohou nastat takové situace, jako je zbytečné opakování testovacích případů na vstupy, které vývojář již otestoval.
- Tento typ testování nelze zaměřit na konkrétní funkční segmenty, které mohou být velmi složité, mohly by totiž nastat situace, kdy by tester nedokázal odhalit chybu (*Java T Point, 2022*).

3.1.5.7 White box

Tester se při testování bílé skříňky zaměřuje na strukturu softwarového kódu a vyvíjí testovací případy tak, aby zkontroloval jeho logické fungování. Tester musí vyvinout testovací případy k jednotlivým modulům, ale také musí ověřit, jak moduly interagují mezi sebou při spuštění softwaru. Všechny testy jsou prováděny na úrovni zdrojového kódu. Tester kontroluje všechny parametry kódu, jako je účinnost napsaného kódu, větvící se příkazy, vnitřní logika, organizace paměti a srozumitelnost kódu. Proto, aby byly pokryty důležité vnitřní části aplikace, je nutné, aby testovací případy byly navrženy velice pečlivě.

Výhody:

- Jelikož tester má znalosti programování, není pro něj těžké vyvinout testovací případy, které budou efektivní a pokryjí širší oblast.
- Díky tomu, že se testování provádí na úrovni kódu, často může docházet k tomu, že tester pomáhá k optimalizaci kódu.

Nevýhody:

- Je nemožné zkontrolovat veškerý kód a zjistit skryté chyby.
- K provedení takovýchto testů jsou nutní velice zkušení testeři, kteří budou rozumět daným programovacím jazykům, což může zvýšit náklady.

- Pokud máme složitý software, je k vývoji a provedení testů potřeba více času, a proto se nám zvyšují i náklady na jednotlivé testovací případy (*Java T Point, 2022*).

3.1.5.8 Gray Box

Testování šedé skříňky je kombinací testování černé a bílé skříňky. Účelem testování šedé skříňky je najít a identifikovat chyby způsobené nesprávnou strukturou kódu, nebo také nesprávným používáním aplikace. Většinou se identifikují chyby specifické pro kontext, které úzce souvisejí s webovým softwarem. Zvyšuje se tak pokrytí na všechny vrstvy jakéhokoli složitého systému. K testování šedé skříňky není zapotřebí přístup testera ke zdrojovému kódu, testy jsou navrženy na základě znalosti algoritmů, architektury nebo jiného chování programu na vysoké úrovni. Můžeme testovat jak prezentační vrstvu aplikace, tak i určité části kódu. Primárně se využívá při penetračních a integračních testech. Tento typ testování je vhodný spíše pro testování GUI, hodnocení zabezpečení, webové aplikace a různé webové služby (*Java T Point, 2022*).

3.1.5.9 Smoke testy

Smoke testy považujeme za neúplné testování, které nám zajišťuje kontrolu nejdůležitějších funkcí systému po nasazení nového buildu. Testujeme pouze nejzákladnější vlastnosti produktu, pokud tyto testy úspěšně projdou, můžeme pokračovat v dalším procesu testování, kde se více zaobíráme jemnějšími detaily. Jedná se o jednoduché testy, které nám pomáhají určit, zda je sestavení chybné, díky tomu se ušetří čas a zdroje. Pokud však některé testy padnou, nebudeme pokračovat dále a vrátíme produkt vývojovému týmu, který musí udělat nezbytné opravy, aby veškeré testy procházely. Tyto testy bývají v dnešní době plně automatizovány a pouští se automaticky po novém buildu, nebo mohou být kombinací manuálního a automatizovaného testování. Výhodou zautomatizování smoke testů je to, že vývojáři mohou nasadit novou verzi, kdykoliv budou chtít, a okamžitě dostanou zpětnou vazbu, zda je základní funkcionalita produktu v pořádku, nebo je potřeba něco opravit *(Hamilton, 2022*).

3.1.5.10 Regresní testy

Hlavním účelem regresních testů je ověřit, zda nedošlo k žádné závadě v určité části softwaru, když byly provedeny změny v jiných částech. Během testování se odhalí mnoho závad a kód je podroben úpravám, aby tyto vady opravil. V důsledku úpravy kódu mohou nastat čtyři scénáře:

- Chyba byla opravena.
- I když bylo vyvinuto úsilí, závadu se nepodařilo opravit.
- Chyba byla opravena, ale něco, co dříve fungovalo, selhává.
- I když bylo vyvinuto velké úsilí, závadu se nepodařilo opravit a kód, který dříve fungoval, selhává.

Pokud bychom při každé opravě nebo úpravě kódu museli provádět veškeré testy, abychom zjistili, zda určitá změna v kódu nezavinila závadu, byl by to neúměrně drahý proces. Navíc nová verze softwaru obsahuje i mnoho nových funkcionalit. Regresní testování se proto řadí do nákladných procesů, proto z existujících testovacích sad je pečlivě vybrána podmnožina testovacích případů pro maximalizaci pravděpodobnosti odhalení nových vad a otestování správné funkčnosti softwaru, zároveň s ohledem na snížení nákladů testování. Regresní testy se proto většinou provádějí před nasazením verze k samotným zákazníkům. Stojí za to zdůraznit, že by se měl klást velký důraz na zautomatizování regresních testů, a to kvůli stále se rozšiřujícímu softwaru. Čím více se bude software rozšiřovat, tím více funkcionalit bude potřeba otestovat (*Java T Point, 2022*).

3.2 Automatizované testování

Za poslední dekádu se v testování softwaru stalo nejrozšířenější změnou právě zvýšené využívání automatizovaného testování. Testování softwaru může být velmi nákladné a náročné na pracovní sílu, proto důležitým cílem je automatizovat jej co nejvíce. Automatizované testy totiž nejen snižují náklady na testování, ale zároveň snižují lidské pochybení a usnadňují regresní testování tím, že umožňují opakované pouštění testů pouhým stisknutí jednoho tlačítka *(Ammann and Offutt, 2008)*. O automatizaci je důležité přemýšlet jako o strategickém aktivu. Takovéto strategické aktivum potřebuje v začátcích velkou podporu, jinak s největší

pravděpodobností selže kvůli nedostatku financí a času. Na automatizaci se můžeme dívat jako dlouhodobou investici, je to totiž neustálý proces, kterého nelze dosáhnout za krátké časové období. Než se organizace rozhodne pro zařazení automatizace, je třeba vzít v úvahu několik předpokladů a posoudit, jestli je vše připraveno pro implementování samotné automatizace (*Naik and Tripathy, 2008*):

- Systém je dostatečně stabilní, veškeré jeho funkce jsou dobře definované, aby měla automatizace smysl. Pokud se software neustále mění a často padá, náklady na údržbu automatizovaných testovacích sad budou poměrně vysoké, aby testovací sady byly aktuální se softwarem.
- Všechny testovací případy jsou jednoznačně definované a připravené pro automatizaci. Vývoj automatizovaných testů bude komplikovaný, a ne příliš úspěšný, pokud nebudou k dispozici podrobné testovací postupy.
- Máme k dispozici potřebné testovací nástroje a infrastrukturu.
- Máme přidělený adekvátní rozpočet na nákup softwarových nástrojů.
- Pracovníci v oblasti vývoje automatizovaných testů mají dostatečné zkušenosti. Vývoji automatizovaných testů se nebude moc dařit, pokud pracovníci nemají potřebné programátorské dovednosti, nebo je nechtějí rozvíjet. Využití dodavatele je jednou z možností pro urychlení vývoje testů, ale je potřeba mít na zřeteli, že dodavatel nebude schopen průběžně udržovat všechny testovací sady.

Pokud nejsou tyto podmínky splněny, s největší pravděpodobností bude mít tento projekt problémy a bude stagnovat, což může vést k jeho ukončení.

Díky automatickým testům snížíme potřebu manuálního a celkově lidského zapojení do stále se opakujících úkolů.

Výhody automatizovaného testování:

- Díky těmto testům můžeme pokrýt větší rozsah testů, což vybízí organizaci k častějšímu a úplnějšímu testování.
- Automatické testování je oproti manuálnímu testování mnohokrát rychlejší, můžeme například manuální testování, které trvá dny, převést na automatizované, které bude trvat pár hodin, a to nejen díky samotné rychlosti automatu, ale i díky

možnosti pouštět testy v paralelním běhu a klidně na více počítačích/virtuálních strojích najednou.

- I nejpečlivější tester se při monotónním ručním testování může dopustit chyby, automatizované testy provádějí kontroly naprosto přesně, bez jakéhokoliv zaváhání při každém spuštění.
- Automatizované testy nám pomáhají rozšířit testování softwaru, díky němuž zvyšujeme hloubku a rozsah testů, které nám pomáhají zlepšit kvalitu softwaru.
- S příchodem automatizovaných testů se začaly více řešit složité funkce, které byly pro manuální testování příliš komplikované a zdlouhavé (*Naik and Tripathy, 2008*).

3.2.1 Typy scénářů pro automatizaci

Existuje mnoho typů scénářů, kdy lze manuální testování automatizovat. Musíme analyzovat aplikaci a vypracovat kompletní testovací plán, který pokrývá všechny funkční oblasti a funkce, které byly doposud testovány manuálně. Tato analýza pak musí demonstrovat, které testovací scénáře by se měly automatizovat, a které by naopak měly zůstat v oblasti manuálního testování (*Java T Point, 2022*).

3.2.1.1 Exception or Negative

Negativní testování je proces, ve kterém se navrhují testovací případy, které mají za úkol zjistit, zda nám aplikace dokáže zpracovat neplatné vstupy, a vrací tak správné výjimky. Takové testovací případy jsou úspěšné jen v případě, že je vyvolán konkrétní typ očekáváné výjimky. Je potřeba zajistit všechny situace, kdy uživatel zadává špatné vstupy, a zabránit tak selhávání aplikace. Díky tomuto testování zlepšíme kvalitu aplikace a šanci na nalezení slabých míst (*Java T Point, 2022*).

3.2.1.2 Stress

Stresové testování je proces, kde je testována účinnost softwaru při nepříznivých a extrémních podmínkách. Ty mohou zahrnovat velké zatížení síťového provozu, nebo zatížení hardwaru, kde se snažíme jít za hranice výkonnostních zdrojů, aniž by se v takových krizových situacích systém zhroutil. Stresové testování pomáhá odhadnout úrovně robustnosti a spolehlivosti. S ohledem na software, který pracuje v kritických situacích nebo v reálném čase, je stresové testování považováno za životně důležité. Hlavním účelem stresového

testování je zajistit, aby se systém dokázal po selhání zotavit; tomuto procesu se říká obnovitelnost (*Java T Point, 2022*).

3.2.1.3 Performance

Výkonnostní testování je proces sloužící ke stanovení účinnosti softwaru. Tento proces zahrnuje provádění velkého množství testů, jako měření doby odezvy v závislosti na počtu provedených instrukcí za sekundu. Výkonnostní testování se často provádí pomocí zátěžových testů; díky tomu můžeme zjistit, zda hardware splňuje specifikace uvedené výrobcem. Výkonnostní testy mohou porovnávat dvě i více zařízení nebo programy, a to z takových hledisek, jako je celková rychlost, rychlost přenosu dat, šířka pásma, účinnost či spolehlivost (*Java T Point, 2022*).

3.2.1.4 Load

Zátěžové testování je proces, ve kterém je výkon softwarové aplikace testován při konkrétním zatížení určitého zařízení. Většinou se jedná o pracovní úroveň, kde se hardware blíží limitům svých specifikací. Zjišťujeme například, jak se aplikace chová při přístupu více uživatelů současně. Díky tomu získáme představu o výkonu v reálném světě (*Java T Point, 2022*).

3.3 Nástroje pro automatizované testování

3.3.1 Selenium

Selenium je bezplatná testovací sada nástrojů pro automatizaci testů. Používá se k ověřování aplikací na různých platformách a prohlížečích. Skládá se z nástrojů Selenium WebDriver, Selenium IDE a Selenium Grid, každá sada je pak určena pro specifické využití. K vytváření skriptů pro Selenium můžeme využít mnoho populárních programovacích jazyků jako Java, C#, Python, JavaScript, Scala a mnoho dalších. Selenium používá k definování každého prvku na stránce koncept známý jako elementy, které známe také pod názvem lokátory. Na stránkách jsou lokátory uloženy v samotných základech a jejich podtřídách, k definování lokátoru používá DOM atributy, jako jsou ID, class, name, tag, link, Css, Xpath a další (*The Selenium Browser Automation Project, 2022*).

3.3.1.1 Selenium IDE

Selenium IDE je nástroj pro záznam a přehrávání kroků. Je to jednoduchý nástroj, který nevyžaduje žádné velké zkušenosti a může jej použít i úplný začátečník. Jedná se o doplněk do prohlížeče, který je podporovaný pouze v prohlížečích Chrome a Firefox. Stačí pustit záznam a vykonávat úkony v prohlížeči, poté záznam uložíme a automatizovaný test je hotový. Po uložení záznamu můžeme jednotlivé kroky upravovat podle svých potřeb. Selenium IDE neumí navázat spojení s databází, nepodporuje pořizování snímků a neexistují žádné funkce pro generování zpráv. Z důvodu své jednoduchosti by tento nástroj měl být používaný spíše jako nástroj k vytváření prototypů, nikoliv jako celkové řešení pro vývoj a údržbu komplexních testů (*Selenium IDE Tutorial : Definition, Features, Benefits, 2022*).

3.3.1.2 Selenium WebDriver

WebDriver ovládá prohlížeč přímou nativní komunikací, a proto není potřeba používat žádný další nástroj k pouštění testů. Neumí porovnávat věci, vyhodnocovat, jestli test prošel, nebo selhal. Má pouze jednu úlohu, a tou je komunikace s webovým prohlížečem. Voláním prohlížeče napřímo vzniká povinnost při každém pouštění skriptů, kde musíme říct, který prohlížeč chceme pouštět, a konfigurovat každý separátně. Webdriver můžeme používat i na vzdáleném zařízení (*The Selenium Browser Automation Project, 2022*).

3.3.1.3 Selenium Grid

Selenium Grid nám umožňuje pouštět paralelně skripty WebDriveru na vzdálených počítačích, a to odesíláním příkazů přes klienta do vzdálených instancí prohlížeče. Díky Selenium Grid můžeme centrálně spravovat různé verze prohlížeče a jejich konfigurace namísto toho, abychom ho nastavovali při každém jednotlivém testu. Funguje jako centrální bod pro všechny testy, které pak můžeme delegovat na zařízení, kde se mají pouštět, můžeme řídit to, kolik testů se maximálně může pouštět na jednom zařízení, a pokud máme více zařízení, vyrovnává mezi nimi zátěž (*The Selenium Browser Automation Project, 2022*).

3.3.2 IBM Rational Functional Tester

Rational functional tester je nástroj pro automatizované testování vyvinutý organizací IBM, který je schopen provádět automatizování testování funkčních, regresních, GUI a testování založené na datech. Podporuje širokou škálu aplikací a protokolů, jako jsou HTML, Java, .Net, Visual Basic a mnoho dalších. Díky pokročilé technologii Script Assure se učí charakteristikám uživatelského rozhraní a aplikuje je na nové verze softwaru, což šetří čas strávený vytvářením nových testovacích scriptů. Umožňuje vývojářům vytvářet skripty s klíčovými slovy, což poskytuje možnost snadného opětovaného použití. Poskytuje rovněž pokročilé možnosti debuggování (IBM Docs, 2022).

3.3.3 Cucumber

Cucumber je testovací nástroj, který podporuje Behavior Driver Development (dále jen „BDD“). BDD je proces vývoje softwaru. Spojuje proces vývoje a testování softwaru. Zajišťuje vývoj, ale také správu softwaru. Překládá jednotlivé kroky do srozumitelného jazyka, kterému dokáže porozumět každý, bez ohledu na své technické znalosti. Cucumber čte spustitelné specifikace napsané prostým textem a ověřuje, že software dělá to, co tyto specifikace říkají. Specifikace se skládají ze scénářů, kde je napsán seznam kroků, které musí vypracovat. Pro každý scénář pak vygeneruje zprávu, jestli daný scénář uspěl, nebo neuspěl. Cucumber musí dodržovat základní pravidla syntaxe, aby porozuměl scénářům, které se nazývají Gherkin. Gherkin používá sadu speciálních klíčových slov, které struktuře dodávají význam ve spustitelném scénáři. Většina řádků pak začíná klíčovým slovem a popiskem, co se v daném kroku děje (*Cucumber - Cucumber Documentation, 2022*).

Příklad použití klíčových slov:

Scenario: Pan Novák zkusí okomentovat blog a neuspěje.

Given: Jsem přihlášený jako Novák.

When: Zkusím přidat komentář: „Tento blog se mi moc nelíbí.“

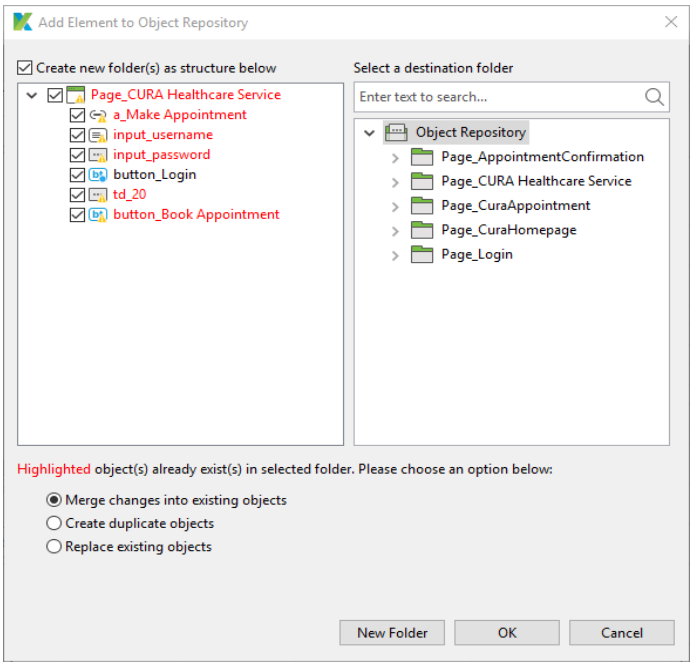
Then: Měl by vidět: „Tento blog nemůžete komentovat.“

3.3.4 Kobiton

Kobiton je komerční software, který využívá cloudovou platformu, jež poskytuje uživatelům skutečné mobilní zařízení, ke kterému mají plnou vzdálenou kontrolu. Zpřístupňuje velké množství modelů zařízení iOS a Android, které můžeme použít na manuální, nebo automatizované testování. Kobiton podporuje k automatizovanému testování Selenium a Appium. Podporuje detailní zaznamenávání a analýzu uživatelských akcí, díky tomu lze problémy rychleji identifikovat a vyřešit. Ke každému scénáři poskytuje video a zároveň snímek obrazovky každé akce. Kobiton nám poskytuje informace o zařízení jako monitorování vytížení procesoru, sítě, baterie, paměti a podrobné logování (*Kobiton, 2022*).

3.3.5 Katalon

Katalon je komplexní sada nástrojů pro automatizaci webových aplikací, API, mobilních a desktopových aplikací. Tento nástroj obsahuje velký balíček výkonných funkcí, díky nimž lehce překonáme běžné výzvy v automatizaci, například vyskakovací okno, nebo čekání na různé prvky. Tester nemusí mít žádné programovací znalosti. Uživatel může přetahovat, vybírat klíčová slova, testovat objekty a rychle vytvářet testovací kroky. Podporuje i vývojové možnosti, jako je například návrh nebo úprava kódu. Podporuje paralelní běh testovacích sad najednou pomocí kolekce testovacích sad. Umožňuje uživatelům zefektivnit proces zachycování a ukládání objektů stránky a poté je organizovat do konkrétních tříd, aby bylo možné je opakovaně použít i později. Můžeme zachytit jakýkoliv prvek všemi možnými lokátory, díky tomu můžeme vybrat a použít nejvíce vyhovující lokátor, který nám pokryje i dynamicky se měnící prvky. Katalon automaticky detekuje podobné existující objekty v úložišti objektů a požádá uživatele o další akci. Díky tomu jsou všechny objekty uspořádány a nenastane situace, že bychom měli duplicitní objekty (*Welcome to Katalon Docs, 2022*).



Obrázek č. 2 – Katalon – hlídání duplicitních objektů (Welcome to Katalon Docs, 2022)

3.4 Používané technologie

3.4.1 Document Object Model

Document object model (dále jen „DOM“) si při každém načtení stránky analyzuje HTML a vygeneruje z ní objekty DOM. DOM je programovací API, a to pro dokumenty HTML a XML. Jedná se o stromovou strukturu, která nám reprezentuje veškeré elementy a jejich atributy na webových stránkách. Určuje nám logickou strukturu a také to, jakým způsobem budeme k dokumentu přistupovat a manipulovat s ním. XML a HTML jsou používány jako způsob, jak reprezentovat mnoho různých druhů informací, které pak můžeme uložit v různých systémech (*Cocchiaro, 2018*).

3.4.2 Application Programming Interface

Application programming interface (dále jen „API“) je rozhraní pro programování aplikací, které umožňuje společnostem, aby data a funkce jejich aplikací mohli otevírat externí vývojáři třetích stran, obchodní partneři, nebo jiná oddělení v jejich společnosti. API umožňuje službám a produktům komunikovat mezi sebou navzájem a využívat vzájemně svá data a funkce, prostřednictvím zdokumentovaného rozhraní. Vývojáři, testeři nebo jiní zaměstnanci, který pracují s API, nemusejí vědět, jak je implementováno. Kdokoliv z organizace může jednoduše používat rozhraní ke komunikaci s jinými produkty a službami. Rozhraní API je sada

definovaných pravidel, která vysvětlují, jak spolu aplikace komunikují. Mezi aplikací a webovým serverem funguje API jako prostřední vrstva, která zpracovává přenos dat mezi jednotlivými systémy (*What is an API?,* 2022). Fungování API bychom si mohli ilustrovat ve čtyřech jednoduchých krocích:

1. Některá z klientských aplikací zahájí volání API, a to za účelem načtení informací. Tento požadavek je zpracován a poslán na webový server.
2. Po obdržení správného požadavku provede API volání na konkrétní externí aplikaci.
3. Příslušný server nebo aplikace, na které byl požadavek poslán, odešle na API odpověď s požadovanými informacemi.
4. Rozhraní API přenese data zpět do počáteční aplikace, odkud byl požadavek odeslán.

3.4.3 XML Path Language

XML Path Language (dále jen „Xpath“) je dotazovací jazyk, který se používá k jedinečné identifikaci nebo pro adresování částí XML dokumentu. Poskytuje nám možnost manipulace s řetězci, čísly nebo booleány. Xpath modeluje dokument XML jako hierarchickou stromovou strukturu uzlů. Podporuje mnoho typů uzlů, prvků, atributů a textových řetězců. Xpath definuje způsob, jak vypočítat hodnotu daného řetězce pro každý typ uzlu. Plně podporuje jmenné prostory XML (*Working with XPath Queries, 2022*).

3.4.4 CSS Selectors

Css selectors je řetězec určený k identifikaci jednoho nebo více prvků na webové stránce, kde kombinuje zadávání atributů nebo „rodičů“ v DOM. Je to způsob, jak najít jeden nebo více prvků na stránce, kde selektor kombinuje element a jeho hodnotu. Css selektory můžeme použít k vyhledávání elementů bez použití ID, class nebo name, kde se můžeme řídit pomocí hierarchie (*CSS Selectors in Selenium : Example to locate elements, 2022*).

3.4.5 Jsoup

Jedná se o Java knihovnu, která se používá k analýze HTML dokumentů. Jsoup poskytuje API pro extrahování a manipulaci s daty z URL nebo HTML souboru. Používají se metody DOM, CSS a Jquery pro extrakci a manipulaci se soubory.

Jsoup načte HTML a vytvoří odpovídající strom DOM. Tento strom funguje stejně jako DOM v prohlížeči a nabízí podobné metody jQuery pro výběr, manipulaci s textem, HTML, atributy a přidávání nebo odebírání prvků (*Jsoup HTML Parser, 2022*).

3.4.6 Intellij IDEA

Multiplatformní integrované vývojové prostředí pro programovací jazyky Java a další, které mají značnou část založenou na Java, avšak nově přidává i mnoho jiných jazyků, které na ní nejsou založené. Poskytuje chytré dokončování kódu, analýzu statického kódu a refaktorování, umožňuje automatizaci rutinních a opakujících se procesů. Nabízí plně funkční integraci s Maven, která umožní automatizovat stavební proces, balení spouštění, nasazení a mnoho dalších aktivit. Po otevření nového projektu jej automaticky detekuje a stáhne všechna požadovaná úložiště a pluginy. Poskytuje i integraci s nástroji pro správu verzí, jako je Git. I kdybychom neměli zřízenou žádnou správu verzí, Intellij funguje jako lokální systém správy verzí, který automaticky zaznamenává revize projektu spuštěnými různými událostmi, když nasazujeme novou verzi aplikace, pouštíme testy nebo i samotné úpravy kódu (*JetBrains IntelliJ IDEA, 2022*).

3.4.7 Maven

Jedná se o oblíbený open-source nástroj, který pomáhá řídit všechny procesy, jako je budování, dokumentace, uvolňování a distribuce v rámci projektů pro lepší správu. Tento nástroj umožňuje vytvářet a dokumentovat rámec životního cyklu. Maven uchovává všechny informace týkající se projektu a podrobnosti o konfiguraci v POM (Project Object Model), což je soubor XML. Tento soubor se nachází v domovském adresáři projektu. Maven poskytuje výchozí nastavení pro konfigurace, takže nemusíme přidávat každou konfiguraci do souboru pom.xml. Můžeme například hlídat kompatibilitu Javy, Scaly a dalších programovacích jazyků nebo i pluginů v celém projektu, aby se nestalo, že někdo z týmu má odlišnou verzi a některé věci mu nefungují z důvodu změn ve frameworku. Díky tranzitivním závislostem můžeme do našeho projektu načíst projektové soubory vzdálených úložišť, nebo také knihovny třetích stran. Obecně platí, že v našem projektu jsou použity všechny závislosti, tak jako všechny, které dědí od svých rodičů, a tak dále, počet úrovní získání závislostí není nijak omezen (*Maven – Maven Documentation, 2022*).

3.4.8 Xpath

Xpath slouží k definování částí dokumentu XML pro navigaci ve struktuře HTML stránek. Jedná se o jazyk pro nalezení jakéhokoliv prvku na webové stránce pomocí výrazu cesty XML. Lze použít pro dokumenty HTML i XML k nalezení umístění libovolného prvku na webové stránce pomocí struktury HTML DOM (*XML Path Language, 2022*).

3.4.9 BitBucket

Jedná se o nejběžněji používaný systém pro správu verzí. Git sleduje změny, které v souborech provedeme, takže máme záznam o tom, co bylo provedeno, a můžeme se v případě potřeby vrátit ke konkrétním verzím. Usnadňuje spolupráci a umožňuje sloučit změny provedené více lidmi do jednoho zdroje. Obecně se používá pro správu zdrojového kódu ve vývoji softwaru. Podporuje nelineární vývoj. BitBucket jako cloudové úložiště slouží k hostování těchto repositářů (*Atlassian, 2021*).

3.4.10 TestNG

Je aplikační rámec pro testování automatizace, kde NG znamená „Next Generation“. Je navržen tak, aby usnadnil komplexní testování. WebDriver nemá žádný mechanismus pro generování reportů. Pomocí TestNG můžeme vygenerovat přehledný report, kde můžeme snadno zjistit, kolik testovacích případů bylo úspěšných, neúspěšných, či bylo přeskočeno nebo nebylo puštěno vůbec. Neúspěšné testovací případy pak můžeme pustit samostatně. TestNG umožňuje nastavení priorit testovacích sad, které by měly být provedeny jako první. Jeden test může být puštěn vícekrát bez použití jakýchkoliv smyček, stačí použít v parametru anotace[1] klíčové slovo „invocationCount“ a přiřadit počet, kolikrát chceme, aby se daný test pustil. Většina anotací je velmi snadno srozumitelných, např.: @BeforeMethod, @BeforeTest, @Ignore, @Test. Testovací případy můžeme seskupovat do skupiny, a tím rozdělovat jednotlivé testy. TestNG umožňuje paralelní běh testů. Jednou z důležitých funkcí TestNG jsou tzv. „Listeners“ (dále jen „posluchači“). Jde o rozhraní, které naslouchá předdefinovaným událostem v testovacích scriptech a umožňuje upravovat výchozí chování nástroje. Existuje mnoho posluchačů, které TestNG využívá, a každý obsahuje různé metody

[1] Anotace v TestNG jsou řádky kódu, které řídí, jak bude provedena metoda pod nimi. Vždy se před nimi píše znak „@“ (*TestNG, 2022*).

vyvolané k provedení konkrétního úkolu. Když například testovací případ selže, nebo je přeskočen, můžeme chtít pořídit snímek obrazovky konkrétního testovacího případu, abychom přesněji viděli, co se pokazilo. V takovém případě můžeme vyvolat metodu „onTestFailure()“ v rámci „ITestListener“, přesunout běh do konkrétního bloku a provést konkrétní událost (*TestNG, 2022*).

4 Vlastní práce

4.1 Návrh testovacích scénářů

Automatizované testy jsou navrhovány na míru pro sázkovou společnost, která nechtěla být jmenována. Celkový objem sekcí a funkcionalit, kterých firma ve svém produktu používá, je mnoho a jsou pravidelně zvyšovány, vylepšovány a upravovány. Proto roste i množství testovacích scénářů. Pravidelným rozvojem produktu se znatelně zvyšuje i doba potřebná na kvalitní otestování během vývoje a regresních testů. Sekce manuálního testování disponuje více jak tisíci testovacími scénáři, přičemž testování scénářů se pohybuje v rozmezí od pár minut až po několik hodin.

Za pomoci vícekriteriální analýzy jsou analyzovány jednotlivé manuální testovací scénáře a vyhodnoceny všechny specifikace pro automatizaci, abychom dosáhli co nejrychlejší návratnosti automatizovaných testů. Vývojář automatizovaných testů pracuje v úzké spolupráci s manuálními testery, ale i se samotnými vývojáři webové aplikace. Manuální tester dodává bližší a detailnější informace k jednotlivým scénářům a díky spolupráci s vývojáři webové aplikace dokážeme zefektivnit a vylepšit robustnost automatizovaných testů, a to hlavně proto, že nám tato spolupráce umožňuje specifikaci jednotlivých prvků na pozadí aplikace k lepší identifikaci elementů a přesnému definování lokalizace.

4.2 Požadavky

Veškeré testy vyvinuté v rámci bakalářské práce budou zahrnuty do již existujících a budou pouštěny v běžném harmonogramu. Testy budou v projektu rozděleny do sekcí, kde v aplikaci figurují pro lepší přehlednost. Budou upřednostněny testy pro sekci komunity, testy budou vyvíjeny na úrovních front-end, kde se bude využívat z velké části kontrola proti databázi a jako pomocník nám poslouží vystavené API pro volání back-end metod, které nám zjednoduší vývoj a kontroly. Všechny testy budou vyvíjeny a testovány na speciálním testovacím prostředí, aby jednotlivé testy nenarušovaly produkční verzi. U testovacích scénářů se klade velký důraz na srozumitelnost jednotlivých kroků a logický postup v jednotlivých krocích. Pokud nenarušíme logickou návaznost kroků, budou složitější testovací scénáře rozděleny do více testů, abychom snížili pravděpodobnost selhání testů na minimum a zároveň abychom kvalitně otestovali jednotlivé funkcionality.

4.3 Praktické použití technologií

V této kapitole budou popsány všechny technologie použité pro vývoj automatizovaných testů.

4.3.1 Vytvoření projektu ve vývojovém prostředí

K vývoji a spravování automatizovaných testů je potřeba vytvořit ve vývojovém prostředí nový projekt. Víme dopředu, že budeme využívat Maven, zvolíme tedy při vytváření nového projektu, že chceme vytvářet nový projekt, který už bude mít základy Maven, abychom pak nemuseli Maven zvlášť přidávat.

Obrázek č. 3 – Základní nastavení Mavenu při vytváření nového projektu [Vlastní zdroj]

Po vytvoření projektu je potřeba nastavit globální jazykovou sadu pro daný projekt, díky které bude IDE vědět, který programovací jazyk se používá a který kompilátor by měla vlastně použít pro kontrolu správné syntaxe kódu. Pro účely projektu bude nastavena Scala verze 2.16. Zároveň musíme nastavit Java JDK[2]. K pouštění testů v TestNG budeme potřebovat i JRE[3] komponentovou sadu, kterou TestNG potřebuje, proto se v projektu zvolila verze Java 11 JDK, která disponuje i JRE. Výhodou Intellij IDEA je to, že nemusíme na internetu dohledávat konkrétní verzi, ale můžeme ji stáhnout rovnou v rámci vývojového prostředí, kde se nám pak automaticky nastaví všechny cesty ke konkrétní Java.

[2] JDK – Sada nástrojů k vývoji pro konkrétní softwarový rámec (*Oracle Lesson*, 2022).
[3] JRE – Spouštěcí prostředí Java, používá se ke pouštění bajtového kódu (*Oracle Lesson*, 2022).

4.3.2 Maven projekt

```
<project>
    <groupId>org.scalaHelpers</groupId>
    <artifactId>czuProjekt</artifactId>
    <version>1.0-SNAPSHOT</version>
    <name>object-Model</name>
    <description>objective page model for autotests</description>
    <packaging>jar</packaging>

    <properties>
        <encoding>UTF-8</encoding>
        <java.version>12</java.version>
        <scala.tools.version>2.12</scala.tools.version>
        <scala.version>2.12.6</scala.version>
    </properties>
    <dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.scala-lang</groupId>
            <artifactId>scala-reflect</artifactId>
            <version>2.12.10</version>
        </dependency>
        <dependency>
            <groupId>org.testng</groupId>
            <artifactId>testng</artifactId>
            <version>6.14.3</version>
        </dependency>
        <dependency>
            <groupId>com.oracle</groupId>
            <artifactId>ojdbc6</artifactId>
            <version>12.1.0.2.0</version>
        </dependency>
    </dependencies>
    </dependencyManagement>
</project>
```

Obrázek č. 4 – Ukázka zdrojového kódu: Ukázka závislostí v projektu a získávání knihoven třetích stran [Vlastní zdroj]

GroupId – je jedinečné v rámci projektu, jedná se o relativní adresářovou strukturu základního úložiště. Tam, kde je v našem příkladu „main.autotest.pom“ skupina, je v adresáři $(mavenRoot)/main/autotest/pom.

ArtifactId – jedná se o obecný název, pod kterým je projekt znám. Společně s groupId vytváří unikátní klíč, který odděluje tento projekt od ostatních projektů.

Properties – fungují jako obecné popisy, které se ukládají do proměnných. Jsou dostupné v celém projektu, například „<java.version>12</java.version>“ můžeme získat kdekoliv

v projektu přes ${java.version} nebo získáním seznamu všech properties přes metodu „java.lang.System.getProperties()“.

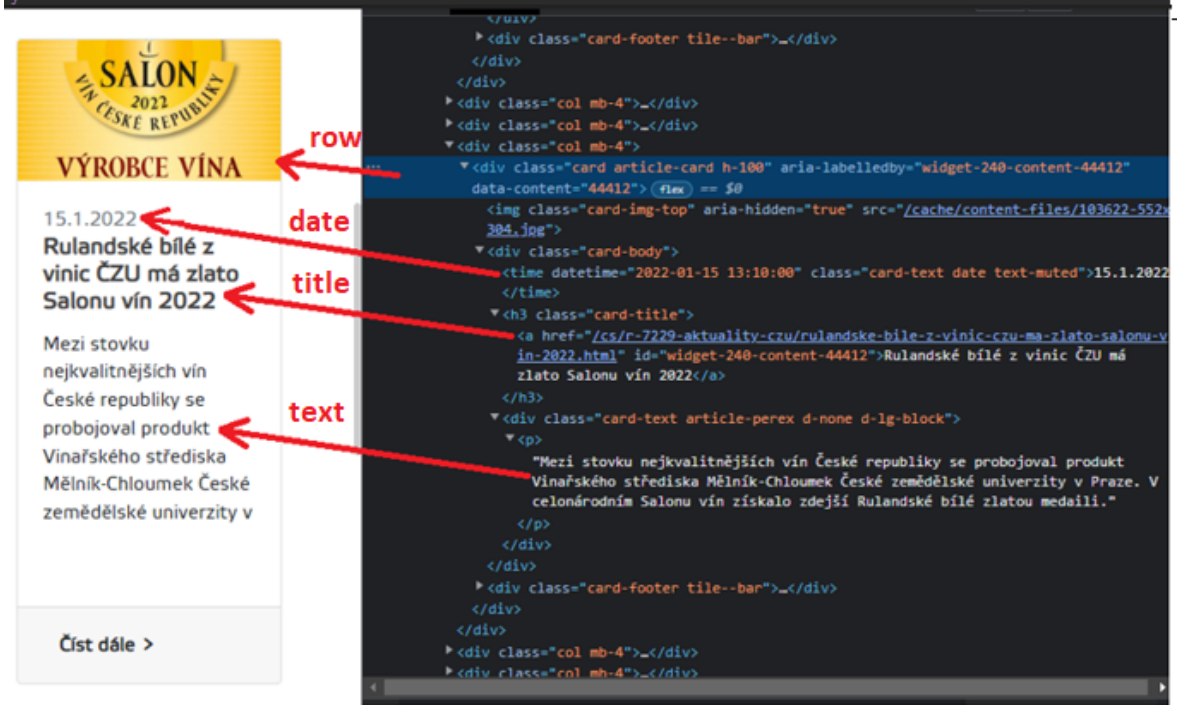
Dependencies – neboli závislosti, Maven stahuje a propojuje dependencies v závislosti na kompilaci a také dalších cílech, které mohou být vyžadovány. Pomocí prvku Scope odkazujeme na cestu ke třídě dané úlohy, díky tomu můžeme omezit přechodnost závislostí. Všechny prvky scope jsou ve výchozím stavu „Kompilace“, pokud nevložíme do závislosti prvek scope, automaticky je mu přiřazen výchozí stav „Kompilace“.

4.3.3 BitBucket

Na již existujícím projektu na serveru pomocí příkazu git clone $url_Daného_projektu si naklonujeme projekt v konkrétní složce, kde se nacházíme. Při prvním dotazování na server budeme muset ověřit naše credentials neboli přístupové údaje, nejběžněji se používají username a heslo. Git si tyto údaje zapamatuje a už se nás na ně nebude dotazovat. Podle pravidel přístupu k danému úložišti může být nastavena přísnější politika pro ověření, kdy se ověřuje i jméno daného uživatele a email. Tyto údaje můžeme jednoduše nastavit. Nastavení jména i příjmení – [git config user.name ‘‘MéJméno MéPříjmení‘‘]. Nastavení emailu skoro stejný postup – [git config user.email ‘‘můjEmail@email.com‘‘]. Pomocí příkazu [git config --list] si můžeme vypsat všechny nastavené konfigurace pro ověření, že jsme vše nastavili správně.

4.3.4 Jsoup parsování

```
case class CzuCardArticle(date: String, title: String, text: String)

def getCZUNewsArticles: List[CzuCardArticle] = {
 val pageSource = driver.getPageSource
 val doc = Jsoup.parse(pageSource)
 val rows = doc.select(".news-header + div > div[class^='col'] >
.card").asScala.toList
 val czuNews: List[CzuCardArticle] = for (row <- rows) yield {
  val date = row.select(".card-body time").attr("datetime")
  val title = row.select(".card-title").text()
  val text = row.select(".card-title + .card-text").text()
  CzuCardArticle(date, title, text)
 }
 czuNews
}
```



Obrázek č. 5 – Ukázka použití Jsoup k získání elementů [Vlastní zdroj]

Pomocí driveru získáme zdroj HTML stránky, na které se aktuálně nacházíme, naparsujeme zdroj pomocí metody „parse()“, abychom získali dokument, ze kterého můžeme číst data, nebo s nimi manipulovat pomocí zpřístupněných metod. Jsoup používá výhradně Jquery selektory. Jako ukázku použití Jsoup se snažíme naparsovat ze stránky czu.cz všechny novinky dostupné na stránce. Na obrázku (Obrázek 5 – Ukázka použití Jsoup k získání elementů) nám proměnná rows slouží k získání všech novinek, které musí mít jasný a přesný identifikátor, abychom omylem nenaparsovali na stránce jiné nebo další elementy než ty, které doopravdy chceme. Rows nám slouží jako rodič všech dalších elementů. Kombinací for cyklu a yield v programovacím jazyce Scala docílíme toho, že pro každého rodiče dokážeme vybrat

potřebná data, která potřebujeme, a uložit je jako kolekci pro další manipulaci nebo kontrolu. Pro každou novinku získáváme její datum, nadpis a text.

4.3.5 Xpath syntaxe

Je to základní syntaxe k nalezení prvku umístěného na webové stránce. Standardní syntaxe Xpath - „Xpath=//tagName[@attribute=“value“]“.

- **//** - Vybrat aktuální uzel
- **TagName –** název konkrétního uzlu {class, div, section, p, table, h1, h2, header, article, a, script}
- **@** - Vyber atribut
- **Attribute** – název daného atributu {@id, @class, @role, @style, @alt, @width, @status}
- **Value** – hodnota atributu

Existují dva typy Xpath:

1. **Absolutní –** je přímá cesta k nalezení prvku, ale její nevýhodou je to, že pokud dojde k nějakým změnám v cestě prvku, Xpath selže. Klíčovou vlastností je to, že začíná jedním lomítkem „/“, což znamená, že můžeme vybrat prvek z kořenového uzlu.
2. **Relativní –** začíná uprostřed struktury HTML DOM. Začíná dvojitým lomítkem „//“. Umí vyhledávat prvky kdekoliv na webové stránce, není potřeba psát dlouhou syntaxi a můžeme začít rovnou od středu struktury HTML DOM. Xpath je ve většině případů vždy podporována, protože nejde o úplnou cestu z kořenového prvku.

4.4 Vývoj a implementace

4.4.1 WebDriver

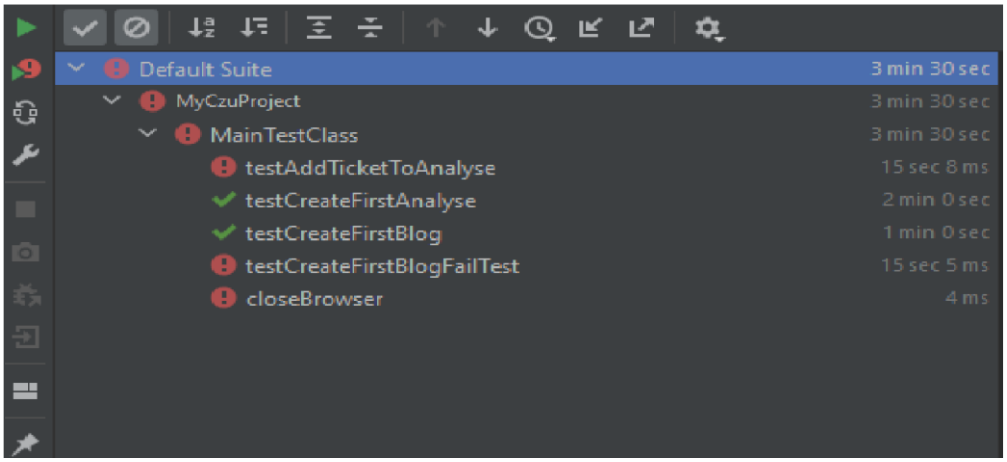
```
abstract class WebDriverInit extends TestNGSuiteLike {

 implicit var driver: WebDriver = _


 protected def startBrowser(): Unit = {
  val browserOptions: ChromeOptions = new ChromeOptions
  browserOptions.setBinary("C:\\Program Files\\Google\\Chrome
Dev\\Application\\chrome.exe")
  browserOptions.addArguments("start-maximized")
  browserOptions.addArguments("net-log-capture-mode=IncludeSocketBytes")
  browserOptions.addArguments("--enable-automation")

  val webDriver: WebDriver = new ChromeDriver(browserOptions)
  webDriver.manage().timeouts().pageLoadTimeout(60, TimeUnit.SECONDS)
  webDriver.get("https://www.czu.cz")
  driver = webDriver
 }

 @AfterTest()
 protected def closeBrowser(): Unit = {
  Try(driver.close()) match {
   case Failure(exception) =>
    driver.quit()
    throw new Exception(exception)
   case Success(_) => Log.info("Prohlížeč byl úspěšně uzavřen.")
  }
}
```

Obrázek č. 6 – Ukázka zdrojového kódu: Nastavení prohlížeče a zapnutí [Vlastní zdroj]

V této bakalářské práci se zaměříme na nejrozšířenější prohlížeč Google Chrome. Pro realizaci automatizace musíme mít nainstalovaný chromeDriver a Selenium WebDriver. ChromeDriver je open source nástroj pro automatické testování webových aplikací. Poskytuje možnosti pouštění JavaScriptu, navigaci na webových stránkách, hledání elementů, vstup uživatele a další. Jedná se o samostatný server, který se používá k pouštění prohlížeče a pro komunikaci mezi prohlížečem a Selenium knihovnou.

Abychom se nemuseli starat o pouštění a celé nastavování prohlížeče v každém testu zvlášť, vytvoříme si abstraktní třídu, kde definujeme metodu právě pro start a nastavení samotného prohlížeče. Abstraktní třídu volíme z toho důvodu, že nechceme a ani nepotřebujeme vytvářet žádnou instanci této třídy. Z této třídy budeme pouze dědit a využívat některé její části. Pomocí anotací TestNG, jejíž knihovnu získáváme a máme ji přístupnou v celém našem projektu díky Maven, můžeme říct, jaká metoda nebo funkce se mají kdy vykonat. Pomocí anotace _@BeforeTest“ říkáme, ať se nám tato metoda zavolá před každou

metodou, která nese anotaci „@Test“, neboli před každým puštěným testem, protože všechny testy musejí mít anotaci „@Test“. Pokud tuto anotaci nebudou mít, nebudou mezi testy zařazeny a nepustí se. V případě, že metoda, která se má pouštět před testem v dané třídě, neprojde, všechny testy v této třídě budou přeskočeny a ani jeden z nich se neprovede; nebyl by pro to totiž důvod, protože v našem případě by se nepustil prohlížeč. Do těchto metod většinou dáváme důležité funkce, bez kterých se testy neobejdou, může se jednat o puštění prohlížeče, přípravu dat, vytváření uživatelů, spojení s databází a mnoho dalších. Když budeme dědit od této třídy, docílíme tím toho, že se nám tyto metody pokaždé zavolají v pořadí a ve chvílích, které udávají jednotlivé anotace, aniž bychom museli cokoliv definovat nebo vytvářet navíc. Všechny tyto metody se dají přetížit, a tím přepsat celou danou logiku a nastavit nové postupy podle potřeb testů v dané třídě, nebo ponechat danou logiku a jen ji rozšířit.



Obrázek č. 7 – Ukázka spuštění testovací sady [Vlastní zdroj]

startBrowser() – otevře nové okno prohlížeče jako novou instanci, které nese unikátní ID, podle něhož můžeme měnit okno, které zrovna ovládáme. ChromeDriver nabízí mnoho možností přizpůsobení a konfigurací relace k WebDriveru přidávání argumentů pomocí třídy[4] „ChromeOptions“. Pokud nemáme nastavenou cestu v proměnných systému k „ChromeDriver.exe“, můžeme použít metodu „setBinary“, u které do parametru vložíme cestu ke spustitelnému souboru prohlížeče „chrome.exe“. Přidáním argumentu pomocí „browserOptions.addArguments("start-maximized")“ se budou všechna okna prohlížeče pouštět přes celou obrazovku daného zařízení, kde je prohlížeč pouštěný. Poté předáme

[4] Třída je šablona, ze které se vytvářejí jednotlivé objekty (*Oracle Lesson, 2022*).

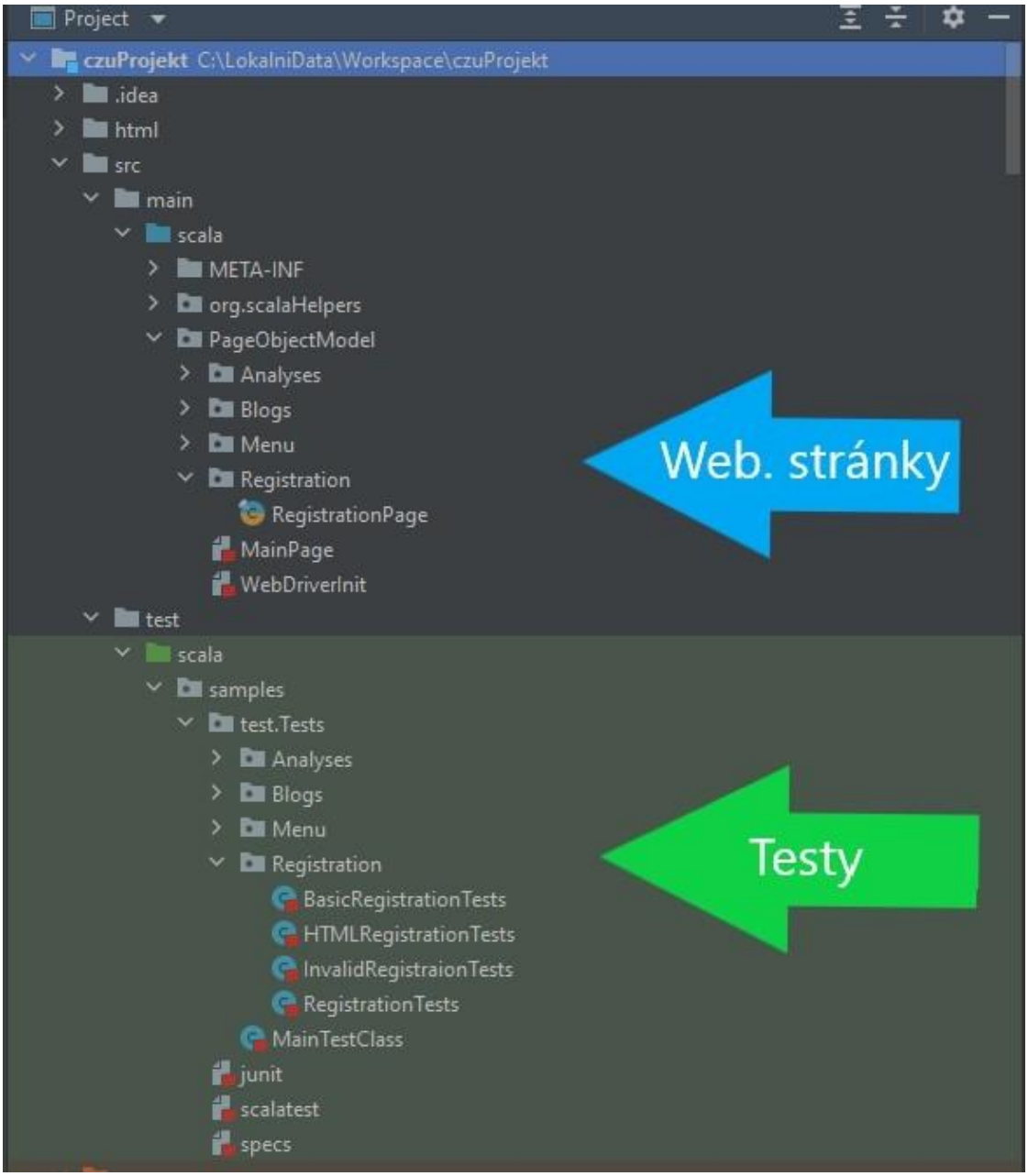
vytvořený objekt[5] ChromeOptions do konstruktoru ChromeDriver, čímž vytvoříme plnohodnotný objekt ChromeDriver se všemi nastaveními a pustí se nám prohlížeč. Pomocí konvertování přiřadíme ChromeDriveru interface[6] WebDriver, a tím získáme objekt Selenium WebDriver, čímž si zpřístupníme veškeré dostupné metody pro práci a ovládání prohlížeče. WebDriver (dále jen „driver“) si uložíme do implicitní proměnné na úrovni třídy, abychom ho mohli předávat dál a používat v testech pro komunikaci s puštěným prohlížečem.

closeBrowser() – metoda, která se díky anotaci „@AfterTest“ pustí po každém testu, který doběhne. Nezáleží na tom, jestli úspěšně, nebo neúspěšně, tato metoda se zavolá v jakémkoliv případě. Pokusí se uzavřít aktivní okno prohlížeče pomocí metody „close()“, které driver pustil. Pokud by se z nějakého důvodu toto okno nepodařilo uzavřít – například více otevřených záložek, metoda“quit()“ uzavře všechna aktivní okna prohlížeče a bezpečně uzavře všechny probíhající relace.

4.4.2 Rozdělení projektu

Celý projekt bude rozdělen na Page object model, ve kterém budou obsaženy metody, na rozšíření s ním spojené, a na testy. Pro každou skupinu testů je vytvořena třída, která v sobě bude obsahovat jen testy z určité oblasti a pro lepší přehlednost – abychom nezaplnili jednu třídu mnoha testy, ve kterých bychom se sice ze začátku dobře orientovali, ale postupem času, až by bylo testů více, by nám již mohlo dělat potíže najít, kde jaký test máme. Kdybychom pak chtěli pustit jen určitou část testů, museli bychom dělat úpravy v dané třídě, což by pro nás bylo velice neefektivní z hlediska vynaloženého času. Díky tomuto rozdělení budeme moci využít i paralelního běhu testů a docílíme lepší dynamiky, kdy budeme moci využít paralelismus na úrovni samotných tříd. Jak můžeme vidět na obrázku (Obrázek 8 – Ukázka rozdělení Page object modelu), pro registrace máme jednu třídu v Page object modelu, ale v testech tuto třídu využívají tři sekce testů na registrace. Získali jsme velkou úsporu času v následných opravách daných testů, díky tomu, že pokud se nám změní lokátor u jednoho elementu, který se používá ve většině testů, opravíme ho na jednom místě, a tím se nám opraví i všechny testy, které tento element používají.

[5] Softwarový objekt koncepčně podobný objektům z reálného světa, které se skládají ze stavu a souvisejícího chování (*Oracle Lesson, 2022*).
[6] Interface je rozhraní, které definuje objektům svou interakci s vnějším světem prostřednictvím metod, které odhaluje (*Oracle Lesson, 2022*).



Obrázek č. 8 – Ukázka rozdělení Page obect modelu [Vlastní zdroj]

4.4.3 Page Object Model

Protože už dopředu víme, že se nám mnoho kroků v testech bude opakovat (například kliknutí na tlačítko „Přihlásit se“, nebo vyplňování textového pole pro přihlášení), museli bychom u každého kroku, který bychom automatizovali, hledat a lokalizovat každý element, se kterým bychom chtěli pracovat, a vznikalo by tak mnoho duplicitních elementů, což je v rámci vývoje velice neefektivní. Proto jsem se rozhodl použít Page object model, který je velmi populární při používání automatizace testování a který vytváří repozitář objektů pro prvky webového uživatelského rozhraní. Velkou výhodou tohoto modelu je, že výrazně snižuje duplicitu kódu, a zlepšuje tak údržbu testů. Pro každou webovou stránku ve webové aplikaci by měla existovat odpovídající třída stránky v našem kódu. Tato třída stánky bude reprezentovat jednotlivé elementy dané webové stránky a také metody, které provádějí operace s těmito elementy k usnadnění vývoje. Název těchto metod by měl být úzce spojen s funkcionalitou, kterou daná webová stránka nabízí. Například když budeme chtít registrovat klienta, budeme mít vytvořenou třídu „RegistrationPage“ pro stránku registrace, kde si uděláme za pomoci elementů přístupných na stránce metodu pro registraci klienta, která by měla mít blízký název, například „registerClient()“. Díky tomu docílíme vysoké efektivnosti, protože se nebudeme muset odkazovat na jednotlivé elementy, ale na jedinou metodu, která nám klienta zaregistruje.

```
final class RegistrationPage() extends WebDriverInit
 with RegistrationPage.Elements
 with RegistrationPage.Helpers

object RegistrationPage {

 def apply(): RegistrationPage = new RegistrationPage()

 trait Loc {
  protected val Name: By = By.xpath("//*[@name='name-Input']")
  protected val SurrName: By = By.id("lastNameInput")
  protected val DateOfBirth: By =
By.cssSelector("input[date='dateOfBrith']")
  protected val MobileNumber: By = By.id("mobilePhoneWithoutPrefix")
  protected val Email: By =
By.cssSelector("input[name='text'][class='emailInput']")
  protected val RegisterButton: By =
By.cssSelector("button[type='submit'][class^='registerBtn']")
  protected val Dialog: By = By.cssSelector(".mainDialog div[@class='reg-
Dialog']")
 }

 sealed protected trait Elements extends Loc {
  this: RegistrationPage =>

  def nameElem: WebElement = driver.findElement(Name)
  def surrNameElem: WebElement = driver.findElement(SurrName)
  def dateOfBirthElem: WebElement = driver.findElement(DateOfBirth)
  def mobileNumberElem: WebElement = driver.findElement(MobileNumber)
  def emailElem: WebElement = driver.findElement(Email)

  def registerBtnElem: WebElement = driver.findElement(RegisterButton)
  def registrationDialog: List[WebElement] =
driver.findElements(Dialog).asScala.toList
 }

 sealed trait Helpers {
  this: RegistrationPage =>

  private case class Client(name: String, surrName: String, dateOfBirth:
String, mobile: Int, email: String)
  def registerClient(name: String, surranme: String, dateOfBrith: String,
mobile: Int, email: String): Client = {
   nameElem.sendKeys(name)
   surrNameElem.sendKeys(surranme)
   dateOfBirthElem.sendKeys(dateOfBrith)
   mobileNumberElem.sendKeys(s"$mobile")
   emailElem.sendKeys(email)

   registerBtnElem.click()

   if(registrationDialog.nonEmpty) {
    registrationDialog.head.getText should be ("Vaše registrace byla
úspěšná.")
    Client(name, surrName = surranme, dateOfBirth = dateOfBrith, mobile =
mobile, email = email)
   } else throw new Exception("Nepodařilo se vytvořit nového clienta")
  }
```

Obrázek č. 9 – Ukázka návrhu Page Object Modelu registrace [Vlastní zdroj]

Jak můžeme vidět na obrázku (Obrázek č. 9 – Ukázka návrhu Page Object Modelu registrace), vytvořili jsme si třídu RegistrationPage, která slouží výhradně pro účely registrace. V samotné třídě pak smí být jen elementy a metody, které se týkají konkrétní stránky na webu, nesmíme je míchat s jinými, jinak bychom porušili zásady page object modelu. Rozdělením lokátorů/elementů získáme ve třídě všeobecnou přehlednost a budeme se lépe orientovat; když budeme potřebovat upravit metodu, nebo cestu k jednotlivým elementům, budeme hned vědět, kde musíme hledat a co přesně upravit. V případě, že musíme změnit lokátor, změníme ho jen na jednom místě. Element vytvořený z daného lokátoru, který pak může být použit na více místech, bude vždy odkazovat na jediný lokátor. Jednou změnou lokátoru opravíme všechna místa, kde je použit, a nemusíme tedy po jednom elementu opravovat na každém místě zvlášť. Ve třídě si můžeme vytvořit i další, rozšiřovací sekce, jako v našem případě „Helpers“, která má pomoci s usnadněním práce s danými elementy, kde si můžeme vytvářet různé balíky příkazů.

4.4.4 Běh testů

Veškeré automatizované testy v rámci bakalářské práce se pouštějí každou neděli po nasazení nové verze aplikace. Pokud je v rámci týdne nasazena nová verze, jsou testy pouštěny ihned po nasazení pomocí TestNG souboru XML, který má připravené veškeré testovací sady a je nakonfigurovaný pro konkrétní běh testů v určitém prostředí, kdy testy můžeme pouštět sériově, nebo je můžeme pouštět paralelně. Testy jsou tímto způsobem rozděleny z toho důvodu, že by se některé testy mohly navzájem ovlivňovat, a tak by docházelo ke kolizi samotných testů a dat. V případě, že u testovacích sad víme, že se nemohou nijak ovlivnit, volíme paralelní běh pro větší úsporu času; pokud se však testy mohou ovlivňovat, přidáváme testy do sériového běhu. Musíme brát v potaz i výkon zařízení, na kterém budou konkrétní testy pouštěny. Když budeme pouštět testy na zařízení s menším výpočtovým výkonem, můžeme si větším paralelismem testů spíše uškodit, než abychom ušetřili čas.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd">
<suite name="DefaultSuite" parallel="tests" configfailurepolicy="continue"
verbose="1" thread-count="10">

 <test name="Test Runner" parallel="classes" thread-count="4">
  <packages>
   <package name="samples.test.Tests.Analyses"/>
   <package name="samples.test.Tests.Blogs"/>
   <package name="samples.test.Tests.Registration"/>
   <package name="samples.test.Tests.Menu"/>
  </packages>
 </test>

</suite>
```

Obrázek č. 10 - Ukázka souboru XML pro pouštění jednotlivých testovacích sad [Vlastní zdroj]

```
@Test()
def testCreateBlogAndVerify(): Unit = {
 /** Jdi do sekce blogů */
 val blogPage = mainPage.menu.goToBlogs()
 /** Klikni na tlačítko napsat nový blog */
 blogPage.leftMenu.writeNewBlogBtn.click()
 /** Vytvoř nový blog se zadanými náležitostmi */
 val createdBlog = blogPage.create.createBlog(user, "BlogName")
 val blogText = createdBlog.createdBlogText
 val blogTitle = createdBlog.blogTitle
 val blogId = createdBlog.id
 /** Počkej dokud nebude vkládání textu ve filtrování blogů dostupné */
 webDriverWait(blogPage.filter.searchInput.isEnabled)
 /** Vlož do vyhledávače název blogu */
 blogPage.filter.searchInput.sendKeys(blogTitle)
 /** Klikni na tlačítko vyhledat  */
 blogPage.filter.filterBtn.click()
 /** Naparsuj blogy a vyber první v seznamu */
 val currentBlog = blogPage.main.parse.head
 /** Naparsovaný blog musí mít stejný název jako vytvořený blog */
 currentBlog.blogName should be (blogTitle)
 /** Jdi do detailu daného blogu kliknutím na název */
 blogPage.main.blogIdElem(blogId).click()
 /** Počkej dokud nebude v detailu blogu dostupný text */
 webDriverWait(blogPage.detail.blogTextElem.isDisplayed)
 /** Text ve detailu blogu musí být stejný jako text u vytvořeného blogu */
 blogPage.detail.blogTextElem.text should be (blogText)
 /** Detail blogu musí mít stejné id jako vytvořený blog */
 blogPage.detail.getBlogId() should be (blogId)
}
```

Obrázek č. 11 – Ukázka testu vytvoření blogu a verifikace vytvoření [Vlastní zdroj]

4.4.5 Reportování

K reportování o výsledcích testů nám poslouží TestNG, který v projektu vytvoří HTML soubor o všech výsledcích běhu ve složce „test-output“ s názvem „emailable-report“. TestNG generuje zprávy HTML po provedení všech testovacích případů, které máme v běžící sestavě. Tyto metody nám pomohou identifikovat stav testů v projektu. TestNG v Selenium má tři metody: passTest, failTest a skipTest ke kontrole dat testovacích případů. Když se podíváme na výsledný report, můžeme snadno zjistit, kolik testovacích případů prošlo, selhalo nebo bylo přeskočeno. Díky reportu můžeme vidět, v jakém stavu se projekt nachází podle úspěšnosti testů, a všeobecný přehled jednotlivých sekcí udává informaci o tom, jak dlouho daný test běžel v milisekundách. Po doběhnutí celé sady nám vrátí celkový čas běhu sady, tyto údaje nám mohou posloužit jako porovnání sériového a paralelního běhu sad testů. Pokud nějaký test spadne, v reportu se nám uloží základní informace o dané chybě, zapříčinění chyby a údaj, na kterém místě v projektu a na kterém řádku v kódu tato chyba nastala. Jednoduchým způsobem můžeme přidat informace o protokolu při testování za použití „Reporter“ třídy, která je přítomna v TestNG. Přidávání informací k jednotlivým krokům nám může při opravě nějakého testu snadno dát informaci o konkrétním problému, aniž bychom test pouštěli znovu, anebo v kódu dohledávali, co automat na daném řádku vykonává za funkci. Reporter nám zároveň může rychle dát informaci o tom, jaká je hlavní podstata testu a čeho se v daném testu pokoušíme dosáhnout. Samotné zprávy reporteru si můžeme upravit podle vlastních představ, tj. jaký budou mít obsah, jakou úroveň a jaký bude způsob ukládání.

```
@Test()
def checkCZUNewsArticles(): Unit = {
 Reporter.log("1) Získání pageSource")
 val pageSource = driver.getPageSource
 Reporter.log("2) Parse a Získání všech cards [news-header]")
 val czuNews = parseNewsCards(pageSource)
 val cardsSize = czuNews.size
 Reporter.log(s"3) Celkový počet cards je ${cardsSize}, základní počet má
být 8.")
 cardsSize should be (8)
}
```

Command line test

samples.test.Tests.MainTestClass#checkCZUNewsArticles

Messages
1) Získání pageSource 2) Parse a Získání všech cards [news-header] 3) Celkový počet cards je 8, základní počet má být 8.

Obrázek č. 12 – Ukázka použití Reporteru a jeho výstupu [Vlastní zdroj]

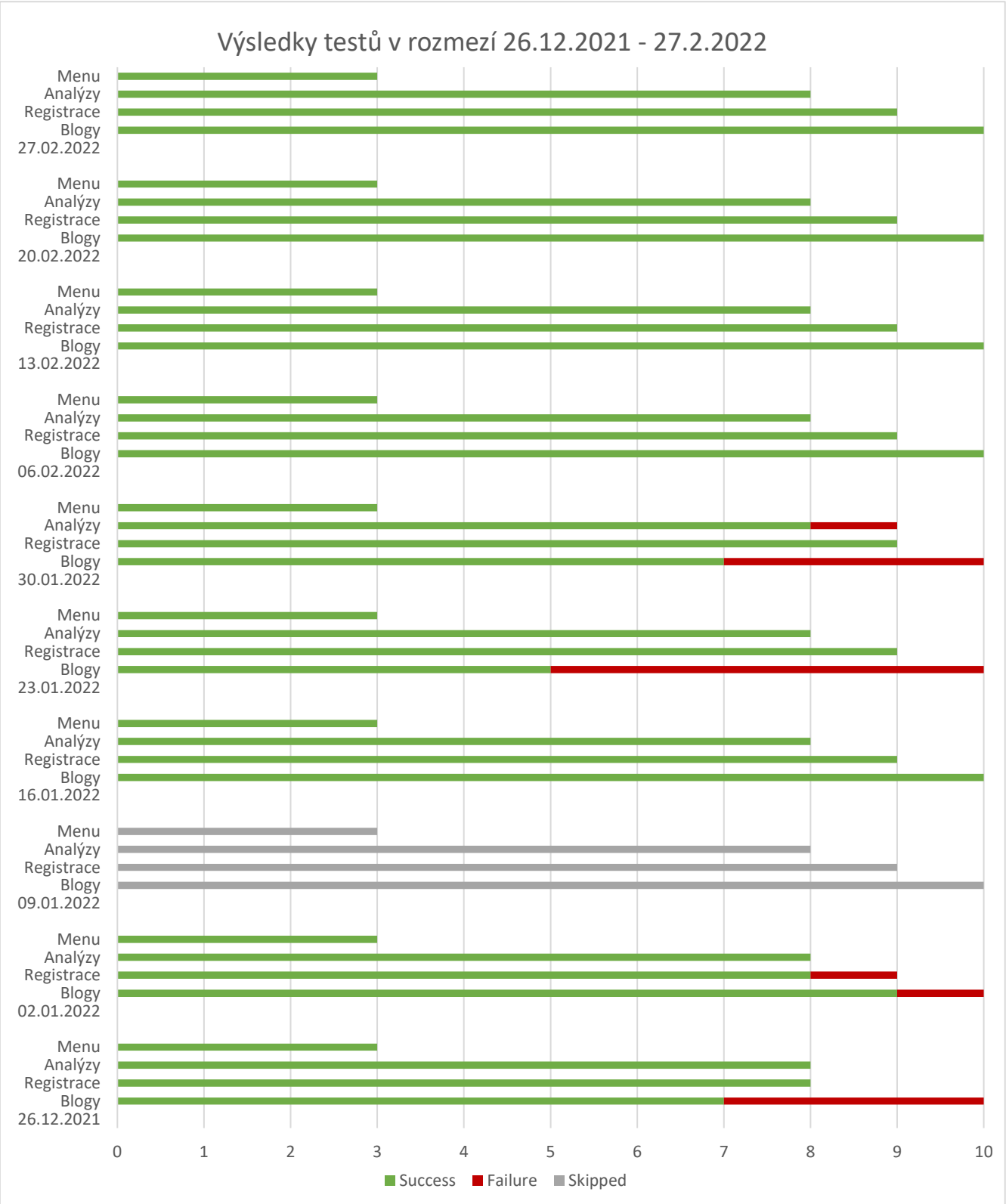
Test	# Passed	# Skipped	# Failed	Time (ms)	Included Groups	Excluded Groups
DefaultSuite						
Test Runner	9	1	2	1 218 083		

Class	Method	Start	Time (ms)
DefaultSuite			
Test Runner — failed (configuration methods)			
samples.test.Tests.Registration.BasicRegistrationTests	startBrowser	1646592412152	8696
Test Runner — failed			
samples.test.Tests.Registration.BasicRegistrationTests	testClientRegistrationHtmlInjection	1646592431898	2224
samples.test.Tests.Registration.RegistrationTests	testClientRegistration	1646592420875	2653
Test Runner — skipped			
samples.test.Tests.Registration.BasicRegistrationTests	testClientRegistration	1646592420860	0
Test Runner — passed			
samples.test.Tests.Analyses.AnalyseTests	testAnalyseFailCreation	1646592412152	530023
	testAnalyseSuccCreation	1646592412162	313156
	testAnalysesCreation	1646592412162	90420
samples.test.Tests.Blogs.BlogBasicTests	testBlogFailCreation	1646592412164	60788
	testBlogSuccCreation	1646592412164	31156
samples.test.Tests.Menu.MenuTests	testBasicCheckSize	1646592412165	15656
	testRefreshing	1646592412166	15599
	testRefreshingTimeOut	1646592412167	61593
samples.test.Tests.Registration.RegistrationTests	userRegistration	1646592433523	86119

Obrázek č. 13 – Ukázka TestNG emailable – report [Vlastní zdroj]

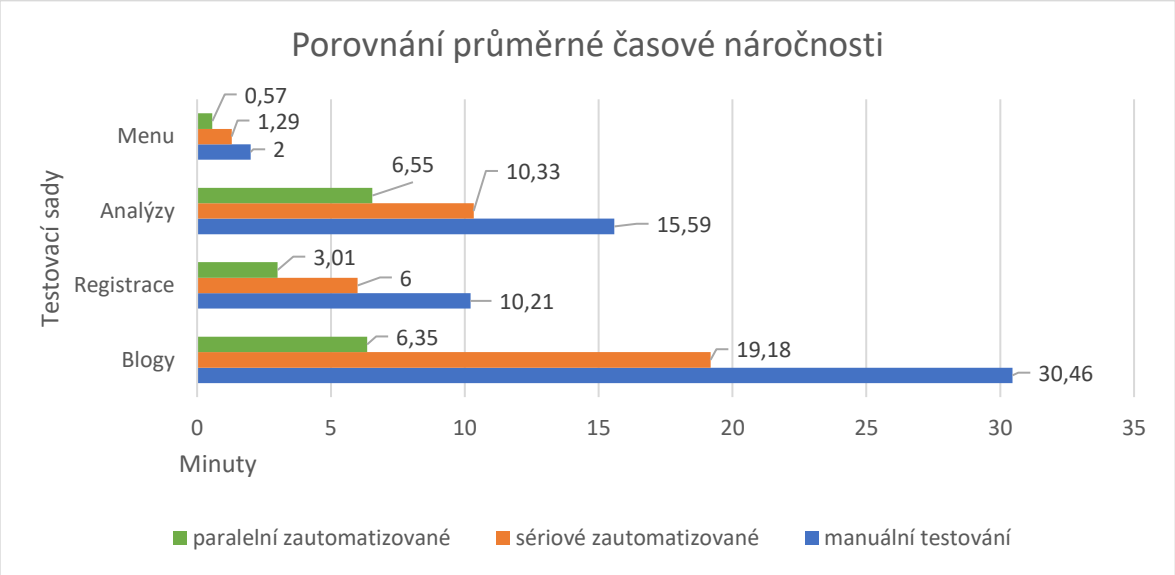
5 Výsledky

5.1 Výsledky testů



Obrázek č. 14 – Dlouhodobé výsledky testů [Vlastní zdroj]

5.1.1 Časová náročnost testů



Obrázek č. 15 – Porovnání časové náročnosti testů v různých režimech běhů [Vlastní zdroj]

5.2 Výsledky práce

Výsledkem práce jsou 4 testovací sady, které obsahují dohromady celkem 30 automatizovaných testů, které pokrývají přibližně hodinu manuálního testování. Samotný vývoj testů trval přibližně 25 hodin. Návratnost tohoto balíku testů se očekává během 27,3 týdnů za předpokladu, že se testy pouští každou neděli a že nebudou provedeny žádné velké změny aplikace v oblasti působení testů, kde by byla pozměněna logika nebo lokátory. Pokud by se testy pouštěly častěji, můžeme očekávat rychlejší návratnost investic v závislosti na četnosti pouštění testovacích sad. Byl vytvořen logický page object model, který se může dál rozšiřovat do dalších oblastí aplikace. Všechny testy byly pouštěny vždy jako skupina sad testů na stejné verzi aplikace a ve stejný čas, aby výsledky za jeden běh byly relevantní k dané verzi aplikace. Z výsledků testů a postupného vývoje můžeme konstatovat, že automatizované testy nemusejí být po svém nasazení zcela stabilní, protože ne vždy můžeme při vývoji eliminovat všechny negativní vlivy. Postupem vývoje a při snaze o eliminování všech překážek můžeme získat velice stabilní testy, které budou robustní i proti menším změnám v aplikaci, ale abychom se k tomuto cíli mohli přiblížit, je velice důležitá komunikace vývojáře aplikace nebo manuálního testeru s vývojářem automatizovaných testů tak, aby testovací sady mohly co nejrychleji reagovat na větší změny v aplikaci. Nasazení testů se za dané testovací období osvědčilo jako velice efektivní, hlavně z hlediska časové úspory, kdy se v některých testovacích sadách dosahovalo v paralelním běhu úspory až 80 % oproti manuálnímu testování a zároveň spolehlivosti, kdy ze strany manuálního testování došlo k pochybení, ale ze strany

automatizovaných testů nikoliv. Díky návrhu page object modelu dokážeme rychle opravit elementy, které se využívají na mnoha místech, a díky tomu strávíme minimální čas opravami lokátorů. V bakalářské práci je využíváno vícevláknového běhu testů. Díky paralelismu můžeme docílit velké časové efektivity, musíme však brát v potaz i to, na jakém zařízení dané testy běží a jak výkonově náročné jsou. Čím více testů poběží souběžně v jeden moment, tím větší důraz bude kladen na výkon zařízení. Pokud na zařízení pustíme více testů, než je zařízení schopno za normálního běhu odbavit, může se vícevláknový běh stát velice neefektivním, až kontraproduktivním. Pokud budeme pouštět na jednom zařízení 10 testů paralelně, které v sériovém běhu zařízení dokáže odbavit za 7 minuty a v paralelním běhu za 3 minuty, je to nejlepší možný scénář, jaký je od vícevláknového běhu očekáván. Avšak pokud pustíme na stejném zařízení 20 testů v sériovém běhu, zařízení je dokáže odbavit za zhruba 15 minut, zatímco pokud pustíme testy v paralelním běhu, na zařízení je vyvinuto příliš velké zatížení a nedokáže všechny testy odbavovat, některé testy kvůli snížení výkonu zařízení mohou selhávat z důvodu nedostatečně rychlého postupu v jednotlivých krocích a ve výsledku nám testy v paralelním běhu mohou trvat až dvojnásobek času. Zároveň se zvyšuje pravděpodobnost selhání testů kvůli nedostatku výpočetního výkonu, než je tomu v sériovém běhu, kde jsou podmínky takřka ideální.

6 Závěr

Hlavním cílem této bakalářské práce byl vývoj sady automatizovaných testů podle vhodně navržených testovacích scénářů. Byly vytvořeny 4 testovací sady, které se z dlouhodobého hlediska osvědčily jak z pohledu časové úspory, tak z pohledu kvality a spolehlivosti. Počáteční investice do vývoje automatizovaného testování může být ze začátku větší a nějakou dobu bez žádné návratnosti z důvodu implementace technologií, vytváření projektu, zabezpečení kvality a optimalizace. Avšak v průběhu vývoje a v celém životním cyklu samotného softwaru se nám investice postupně mnohonásobně vrátí. Výhodou použití page object modelu je jednoduchá, rychlá údržba elementů a přehlednost celého projektu. Projekt může být dále rozvíjen a implementován novými testovacími scénáři. Automatizované testování by mělo být v dnešní době součástí každého vývoje softwaru z důvodu vytváření složitějších a komplikovaněji propojených aplikací, u kterých se zvyšují jejich nároky na kvalitu a rychlost testování.

První, teoretická část byla věnována obecnému seznámení s testováním softwaru. Druhá část je především věnována automatizovanému testování a představení používaných technologií. Některé z těchto technologií byly pak použity v praktické části.

Praktická část se zabývá vytvořením projektu pro automatizované od úplného počátku a postupným popisem využití technologií a jejich implementace.

Tato práce poskytne čtenáři základní přehled o automatizovaném testování a možnostech použití technologií k jeho realizování. Navržená metodika pak může sloužit jako výchozí zdroj pro postup vytvoření projektu, vývoj a údržbu automatizovaných testů.

Seznam použitých zdrojů

1. Ammann, P. and Offutt, J. (2017) *Introduction to Software Testing*. 2017th edn. [cit. 14.02.2022]. ISBN 978-1-107-17201-2.

2. Atlassian. „Learn Git- Git Tutorials, Workflows and Commands | Atlassian Git Tutorial". [cit. 17.09.2021] Dostupné z: https://www.atlassian.com/git.

3. B. Kitchenham and S. L. Pfleeger (1996) Software quality: the elusive target [special issues section]. Dostupné z: https://ieeexplore.ieee.org/.

4. *CSS Selectors in Selenium : Example to locate elements BrowserStack.* [cit.14.02.2022]. Dostupné z: https://browserstack.wpengine.com/guide/css-selectors-in-selenium/.

5. Cocchiaro, C. (2018) *Selenium Framework Design in Data-Driven Testing*. 2018th edn. Packt Publishing. ISBN 978-1788473576.

6. Codes Kitchen Cost of fixing vs preventing. [cit. 14.02.2022]. Dostupné z: https://www.coderskitchen.com/cost-of-fixing-vs-preventing-bugs/.

7. *Cucumber - Cucumber Documentation*. [cit. 14.02.2022]. Dostupné z: https://cucumber.io/docs/cucumber/

8. Hamilton, T. '*What is Smoke Testing? How to do with EXAMPLES*'. [cit. 14.02.2022]. Dostupné z: https://www.guru99.com/smoke-testing.html

9. *IBM Docs* (2021). [cit. 14.02.2022]. Dostupné z: https://prod.ibmdocs-production-dal-6099123ce774e592a519d7c33db8265e-0000.us-south.containers.appdomain.cloud/docs/en/rtw/9.2.0?topic=tester-release-notes-version-920

10. Java T Point. [cit. 14.02.2022]. Dostupné z: https://www.javatpoint.com/

11. *JetBrains IntelliJ IDEA*. [cit 14.02.2022]. Dostupné z: https://www.jetbrains.com/idea/.

12. *Jsoup HTML Parser*. [cit. 14.02.2022]. Dostupné z: https://jsoup.org/.

13. Khorikov, V. (2020) *Unit Testing Principles, Practices, and Patterns*. ISBN 9781617296277.

14. *Kobiton*. [cit. 14.02.2022]. Dostupné z: https://support.kobiton.com/hc/en-us.

15. *Maven – Maven Documentation*. [cit 14.02.2022]. Dostupné z: https://maven.apache.org/guides/.

16. Naik, K. and Tripathy, P. (2008) *Software Testing and Quality Assurance: Theory and Practise*. [cit. 14.02.2022]. ISBN: 0471789119.

17. *Oracle Lesson: Object-Oriented Programming Concepts*. cit. [14.02.2022]. Dostupní z: https://docs.oracle.com/javase/tutorial/java/concepts/index.html.

18. *Selenium IDE Tutorial : Definition, Features, Benefits BrowserStack*. [cit. 14.02.2022]. Dostupné z: https://browserstack.wpengine.com/guide/what-is-selenium-ide/.

19. Software Testing Help Types of software Errors. [cit. 14.02.2022]. Dostupné z: https://www.softwaretestinghelp.com/types-of-software-errors/.

20. *TestNG*. [cit. 14.02.2022]. Dostupné z: https://testng.org/doc/documentation-main.html.

21. *The Selenium Browser Automation Project Selenium*. [cit. 14.02.2022]. Dostupné z: https://www.selenium.dev/documentation/.

22. *Welcome to Katalon Docs* (2021) *https://docs.katalon.com*. [cit. 14.02.2022]. Dostupné z: https://docs.katalon.com/katalon-studio/docs/index.html.

23. *What is an API?*. [cit. 14.02.2022]. Dostupné z: https://www.redhat.com/en/topics/api/what-are-application-programming-interfaces.

24. *Working with XPath Queries*. [cit. 14.02.2022]. Dostupné z: https://docs.oracle.com/cd/E35691_01/doc.121/e35689/xpath.htm

25. *XML Path Language (XPath)*. [cit. 14.02.2022]. Dostupné z: https://www.w3.org/TR/1999/REC-xpath-19991116/.