Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačních technologií

Bakalářská práce

Automatizace testování softwaru

Kateřina Mašková

© 2023 ČZU v Praze

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Kateřina Mašková

Informatika

Název práce

Automatizace testování softwaru

Název anglicky

Software testing automation

Cíle práce

Bakalářská práce se věnuje automatizovanému testování aplikací. Hlavním cílem je analyzovat stávající situaci manuálního testování, navrhnout vhodnou automatizace a na základě její implementace porovnat oba přístupy.

Dílčí cíle jsou:
- analýza postupů při automatizaci testování softwaru na základě vybrané odborné literatury
- analýza stávající situace, definice požadavků na zavedení automatizace
- navrhnout a implementovat automatizaci testování a ověřit její vhodnost na základě porovnání s manuálním testováním

Metodika

Metodika teoretické části bakalářské práce je založená na studiu odborných knižních a elektronických zdrojů z oblasti testování a kvality softwaru. Metodika praktické části spočívá v analýze stávající situace používající manuální testování. Na základě definice specifických požadavků pro automatizaci bude zvolen konkrétní automatizační nástroj a postup bude aplikován v praxi. Pomocí porovnání obou přístupů bude ověřena vhodnost zavedené automatizace. Na základě poznatků z praktické a teoretické části budou formulovány závěry práce.

Doporučený rozsah práce

40-50

Klíčová slova

Automatické testování, manuální testování, testovací scénáře, agilní vývoj, software

---

Doporučené zdroje informací

CRISPIN, Lisa a Janet GREGORY. Agile testing: a practical guide for testers and agile teams. New Jersey: Addison-Wesley, c2009. The Addison-Wesley signature series. ISBN 978-0321534460.

PATTON, Ron. Testování softwaru. Praha: Computer Press, 2002. Programování. ISBN 80-7226-636-5.

ROMAN, Adam. A study guide to the ISTQB (R) foundation level 2018 syllabus. Springer International Publishing, 2018. ISBN13 9783319987392

ROUDENSKÝ, Petr a Anna HAVLÍČKOVÁ. Řízení kvality softwaru: průvodce testováním. Brno: Computer Press, 2013. ISBN 9788025138168.

Předběžný termín obhajoby

2022/23 LS – PEF

Vedoucí práce

Ing. Jan Pavlík

Garantující pracoviště

Katedra informačních technologií

Elektronicky schváleno dne 14. 7. 2022

doc. Ing. Jiří Vaněk, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 27. 10. 2022

doc. Ing. Tomáš Šubrt, Ph.D.

Děkan

V Praze dne 15. 03. 2023

Čestné prohlášení

Prohlašuji, že svou bakalářskou práci "Automatizace 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(ka) uvedené bakalářské práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne 15.3.2023 ______________________________

Poděkování

Ráda bych vyjádřila své díky všem, kteří mi pomohli při psané této bakalářské práce. Nejprve bych ráda poděkovala vedoucímu této práce, panu Ing. Janovi Pavlíkovi, Ph.D. za odborné vedení a cenné rady. Díky patří i vedení firmy Quanti s.r.o. za možnost realizace praktické části a kolegům za trpělivost při poskytování konzultací a zpětné vazby. V neposlední řadě bych chtěla poděkovat své rodině a přátelům za podporu během celého studia.

Automatizované testování softwaru

Abstrakt

Tato bakalářská práce se zaměřuje na problematiku automatizovaného testování softwaru a jeho ekonomickou výhodnost. Cílem práce je implementovat automatizované testování na vybraný projekt a na základě porovnání výsledků s výsledky manuálního testování vyhodnotit vhodnost provedené automatizace.

V teoretické části práce jsou představeny základní principy vývoje a testování softwaru a pojmy důležité pro tuto oblast.

V praktické části práce jsou sepsány testovací scénáře a na jejich základě je realizována automatizace testování vybrané webové aplikace určené ke správě firemních financí, projektů a zaměstnanců. Poté jsou změřeny výsledky automatizovaného a manuálního testování v oblasti časové náročnosti a chybovosti. Na základě naměřených hodnot je dopočítána finanční náročnost provedené automatizace. Výsledky jsou poté porovnány a je zhodnocen přínos a návratnost provedené automatizace oproti manuálnímu testování.

Výstupem práce je vyhodnocení výhodnosti a na jeho základě jsou sepsána doporučení pro použití automatizovaného testování na podobných projektech.

Klíčová slova: Automatické testování, manuální testování, případy užití, testovací scénáře, testovací nástroje, regresní testování, testovací dokumentace, agilní vývoj, software, QA, testovací nástroje

Software testing automation

Abstract

This bachelor thesis focuses on the issue of automated software testing and its cost-effectiveness. The aim of the thesis is to implement automated testing on a selected project and to evaluate the suitability of the automation based on the comparison of the results with the results of manual testing.

In the theoretical part of the thesis the basic principles of software development and testing and concepts important for this area are presented.

In the practical part of the thesis, test scenarios are written and based on them the automation of testing of a selected web application designed to manage company finances, projects and employees is implemented. Then, the results of automated and manual testing are measured in terms of time and error rates. Based on the measured values, the financial intensity of the automation is calculated. The results are then compared and the benefit and payback of automation versus manual testing is assessed.

The output of the work is a benefit evaluation and based on it, recommendations are made for the use of automated testing on similar projects.

Keywords: Automated testing, manual testing, use cases, test cases, test tools, regression testing, test documentation, agile development, software, QA, test tools

Obsah

1 Úvod **10**

2 Cíl práce a metodika **11**

2.1 Cíl práce 11

2.2 Metodika 11

3 Teoretická část práce **12**

3.1 Vývoj softwaru 12

3.1.1 Životní cyklus vývoje softwaru 12

3.1.2 Vztah mezi vývojem a testováním v rámci životního cyklu softwaru 13

3.2 Agilní vývoj 13

3.2.1 Scrum 15

3.3 Kvalita softwaru 15

3.4 Testování softwaru 16

3.4.1 Axiomy testování softwaru 17

3.4.2 Proces testování softwaru 18

3.4.3 Význam testování v procesu vývoje softwaru 18

3.4.4 Role testera 19

3.5 Chyby softwaru 20

3.5.1 Kořenové příčiny 21

3.5.2 Klasifikace chyb 22

3.6 Testovací dokumentace 23

3.6.1 Testování dokumentace 24

3.6.2 Testovací plán 24

3.6.3 Případ užití (use case) 25

3.6.4 Testovací případ (test case) 26

3.7 Kategorie testů 26

3.7.1 Testování dle fáze vývoje 26

3.7.2 Black box vs white box 27

3.7.3 Regresní testování 29

3.8 Automatizované testování softwaru 29

3.8.1 Význam automatizace 29

3.8.2 Jak automatizovat 31

3.8.3 Nástroje pro automatizaci testování softwaru 31

3.8.4 Selenium 32

3.8.5 Cypress 33

4 Praktická část práce **34**

4.1 Představení aplikace a důvod automatizace 34

4.2 Požadavky na automatizaci 35
4.3 Výběr testovacího nástroje 35
4.4 Případy užití 37
4.4.1 Případ užití: Vytvoření projektu 37
4.4.2 Případ užití: Úprava projektu 38
4.4.3 Případ užití: Připojení faktury k projektu 39
Název případu užití: Připojení faktury k projektu 39
4.5 Testovací scénáře 39
4.5.1 Testovací scénář: Vytvoření budget main projektu 41
4.5.2 Testovací scénář: Úprava budget projektu 42
4.5.3 Testovací scénář: Připojení income faktury k budget main projektu 43
4.6 Vývoj automatizovaných testů 45
4.6.1 Organizace repozitáře 45
4.6.2 Testy 46
4.6.3 Handlers 48
4.6.4 Fixtures 49
4.6.5 Builder a entity 49
4.7 Měření manuálního testování 50
4.8 Měření automatizovaného testování 52
4.9 Porovnání měření manuálního a manuálního testování 53
4.9.1 Časová úspora 53
4.9.2 Finanční návratnost 54
4.9.3 Personální náročnost 56
4.9.4 Chybovost 57

5 Výsledky a diskuse 58

6 Závěr 60

7 Seznam použitých zdrojů 61

8 Seznam obrázků, tabulek, grafů a zkratek 66
8.1 Seznam obrázků 66
8.2 Seznam tabulek 66
8.3 Seznam grafů 66
8.4 Seznam zdrojových kódů 66

1 Úvod

Rychlý vývoj výpočetních technologií spojený s koncem 20. století zapříčinil, že se postupem času staly běžnou součástí lidského života. S výpočetními technologiemi se setkáváme napříč všemi průmyslovými odvětvími, v lékařství, ve finanční sféře, ve vzdělávání, i v běžném životě. Staly se nejenom běžnou, ale především takřka nezbytnou součástí našich životů. S výpočetními technologiemi je neodmyslitelně spjatý software, bez kterého by mnohé z nich nemohly fungovat.

Testování softwaru, kterému se věnuje tato bakalářská práce, je takřka elementární součástí procesu vývoje softwaru. Testováním funkčnosti softwaru před jeho vydáním lze odhalit a odstranit případné chyby a nedostatky, čímž lze předejít často i fatálním problémům. Stejně jako vývoj je i testování složitý proces, ke kterému lze přistupovat různými způsoby.

2 Cíl práce a metodika

2.1 Cíl práce

Hlavním cílem bakalářské práce je navržení a implementace řešení automatizovaného testování webové aplikace vyvíjené ve zvolené firmě a následné vyhodnocení efektivity a profitability ve srovnání s původním řešením.

Dílčí cíle jsou:

- analyzovat postupy při automatizaci testování softwaru na základě vybrané odborné literatury
- analyzovat stávající situaci a definovat požadavky na zavedení automatizace
- vybrat vhodný testovací nástroj
- navrhnout a implementovat automatizaci testování
- ověřit vhodnost implementace na základě porovnání s předchozím manuálním testováním

2.2 Metodika

Metodika teoretické části bakalářské práce je založená na studiu odborných knižních a elektronických zdrojů z oblasti testování a kvality softwaru. Metodika praktické části spočívá v analýze stávající situace používající manuální testování. Na základě definice specifických požadavků pro automatizaci bude zvolen konkrétní automatizační nástroj a postup bude aplikován v praxi. Pomocí porovnání obou přístupů bude ověřena vhodnost zavedené automatizace. Na základě poznatků z praktické a teoretické části budou formulovány závěry práce.

3 Teoretická část práce

3.1 Vývoj softwaru

Vývoj softwaru je komplexním procesem zahrnujícím mnoho činností za účelem vytvoření softwarového produktu. Nejdůležitějšími jsou: specifikace, návrh, implementace, tvorba dokumentace, testování a oprava chyb. Většina vývoje softwaru probíhá ve větších či menších týmech. Je tedy potřeba koordinovat nejenom jednotlivé činnosti, ale i týmy, které na nich pracují. Důležitým prvkem v procesu vývoje softwaru je modelování. To slouží k návrhu jednotlivých částí výsledného systému, jejich propojení a jako podpora při tvorbě dokumentace. Dostatečně propracovaným modelováním lze předejít chybám v pozdějších částech vývoje (1).

3.1.1 Životní cyklus vývoje softwaru

Životní cyklus vývoje softwaru (Software Development Life Cycle, SDLC) je proces, kterým se vytváří a vylepšuje software. Dle (7) ho můžeme definovat také jako *„návaznost jednotlivých etap procesu vývoje"*. Každá jednotlivá etapa vývoje je přesně definovaná a nezbytná pro výsledný softwarový produkt.

Existuje mnoho různých metodologií pro SDLC, ale všechny zahrnují následující fáze (18):

1. **Analýza požadavků:** Zjišťování potřeb a požadavků uživatelů a stanovení cílů projektu.
2. **Návrh:** Navrhování architektury a rozvržení softwaru (specifikace funkcí, rozvržení uživatelského rozhraní, databáze, ...) tak, aby splnil veškeré požadavky.
3. **Implementace:** Programování a implementace návrhu.
4. **Testování:** Testování softwaru, aby se ověřilo, že splňuje požadavky uživatelů a že je bezchybný.
5. **Integrace a testování:** Integrace jednotlivých částí softwaru do celku a testování celého systému. Po této fázi je software považován za připravený k nasazení do produkčního prostředí.
6. **Maintenance:** Údržba a opravy softwaru po jeho nasazení včetně oprav chyb.

Některé metodologie, jako Agile nebo Scrum, se zaměřují na krátké cykly vývoje, kde se průběžně vytváří a testuje funkční software. Jiné metodologie, jako Waterfall, se zaměřují na pečlivé plánování a dokončení jednotlivých fází před přechodem na další fázi.

3.1.2 Vztah mezi vývojem a testováním v rámci životního cyklu softwaru

Vývoj softwaru a jeho testování jsou dva klíčové prvky životního cyklu softwaru. Vývoj se zaměřuje na vytváření softwarového produktu, zatímco testování se zaměřuje na ověřování, zda produkt splňuje požadavky a specifikace. Tyto dva prvky jsou vzájemně propojené a navzájem ovlivňují.

V průběhu vývoje se vytváří softwarový produkt od základu. V této fázi se rozhoduje o architektuře, návrhu a implementaci softwarového produktu. Vývojový tým musí při vývoji softwaru zohledňovat funkční a nefunkční požadavky. Funkční požadavky jsou specifikace, které určují, co software musí dělat, zatímco nefunkční požadavky se zaměřují na to, jak software musí fungovat, aby splnil požadované specifikace (18).

Během vývoje softwaru se také vytváří testovací plán, který se zaměřuje na ověření, zda software splňuje specifikace a požadavky. Testování se provádí v několika fázích životního cyklu softwaru – část probíhá během vývoje, část po jeho skončení.

V rámci testování se používají různé metody testování, jako jsou manuální testování, automatizované testování a testování v reálném prostředí. Cílem testování je odhalit chyby a nedostatky v softwaru a zajistit, že software plní funkční i nefunkční požadavky.

Je důležité, aby vývojový tým spolupracoval s týmem testování, aby zajistil, že software splňuje specifikace a požadavky. Testování by mělo být zahrnuto do každé fáze životního cyklu softwaru a mělo by být průběžné a iterativní, aby bylo možné odhalit chyby a nedostatky co nejdříve a minimalizovat tak náklady na opravy (19).

3.2 Agilní vývoj

Pro účely této práce považuji za důležité blíže přiblížit pojmy „Agilní vývoj“ a „Scrum“, protože se jedná o způsob vývoje softwaru ve společnosti Quanti s.r.o, kde je realizována praktická část práce.

Agilní vývoj je moderní a stále populárnější přístup k vývoji softwaru. Jedná se o iterativní způsob vývoje, kdy je práce rozdělena do menších celků. Ačkoliv bylo možné setkat se s některými agilními metodami i v minulosti, agilní vývoj jako samostatný přístup

byl definován v roce 2001 v Manifestu agilního vývoje. Hlavní myšlenkou agilního vývoje je přizpůsobivost a schopnost pružně reagovat na změny (4).

Základní hodnoty agilního vývoje shrnuje již zmíněný Manifest agilního vývoje (6):
„Jednotlivci a interakce před procesy a nástroji
Fungující software před vyčerpávající dokumentací
Spolupráce se zákazníkem před vyjednáváním o smlouvě
Reagování na změny před dodržováním plánu"

Manifest zároveň definuje 12 principů, na kterých Agilní vývoj stojí:

1. *„Naší nejvyšší prioritou je vyhovět zákazníkovi časným a průběžným dodáváním hodnotného softwaru."*
2. *„Vítáme změny v požadavcích, a to i v pozdějších fázích vývoje. Agilní procesy podporují změny vedoucí ke zvýšení konkurenceschopnosti zákazníka."*
3. *„Dodáváme fungující software v intervalech týdnů až měsíců, s preferencí kratší periody."*
4. *„Lidé z byznysu a vývoje musí spolupracovat denně po celou dobu projektu. "*
5. *„Budujeme projekty kolem motivovaných jednotlivců. Vytváříme jim prostředí, podporujeme jejich potřeby a důvěřujeme, že odvedou dobrou práci."*
6. *„Nejúčinnějším a nejefektnějším způsobem sdělování informací vývojovému týmu z vnějšku i uvnitř něj je osobní konverzace."*
7. *„Hlavním měřítkem pokroku je fungující software."*
8. *„Agilní procesy podporují udržitelný rozvoj. Sponzoři, vývojáři i uživatelé by měli být schopni udržet stálé tempo trvale."*
9. *„Agilitu zvyšuje neustálá pozornost věnovaná technické výjimečnosti a dobrému designu."*
10. *„Jednoduchost-umění maximalizovat množství nevykonané práce-je klíčová."*
11. *„Nejlepší architektury, požadavky a návrhy vzejdou ze samo-organizujících se týmů."*
12. *„Tým se pravidelně zamýšlí nad tím, jak se stát efektivnějším, a následně koriguje a přizpůsobuje své chování a zvyklosti."*

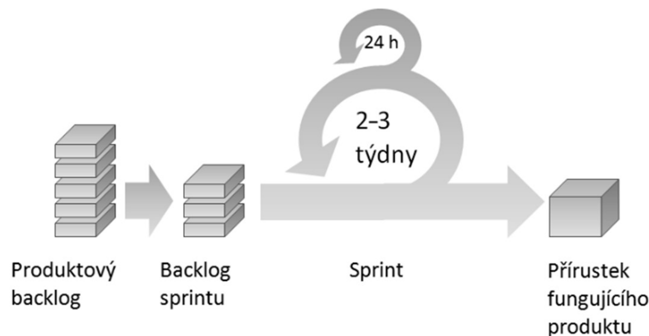
3.2.1 Scrum

Scrum je v praxi často využívaná agilní metoda vývoje softwaru. Jejím základem je úzká spolupráce mezi členy vývojového týmu. Práce se zde rozděluje do menších úseků nazývaných „sprinty“. Délka trvání jednoho sprintu je obvykle 1-4 týdny, v závislosti na konkrétním projektu. Nejčastěji jsou využívány dvoutýdenní sprinty. Výsledkem sprintu je „demo“ – nová část aplikace, která je předložena klientovi ke schválení a případným připomínkám.

Na začátku každého sprintu probíhá schůzka celého vývojového týmu nazývaná „sprint planning“. Zde se rozděluje práce na další sprint. Jednotlivým úkolům je přiřazována priorita dle požadavků zákazníka. Omezením je zde cena a časová náročnost vycházející z odhadu náročnosti jednotlivých úkolů. Úkoly, které nemohou být z kapacitních důvodů v daném sprintu splněny se ukládají do produktového backlogu (2).

V průběhu spritu probíhají krátké schůzky nazývané „stand-upy“, kde se členové týmu vzájemně informují o aktuálním stavu jejich práce, dalších plánech a případných problémech. Stand-upy v závislosti na konkrétním projektu probíhají každý den nebo v určité dny v týdnu. Uvedené schéma scrumu zobrazuje obrázek 1.

Obrázek 1 Schématické zobrazení Scrumu



Zdroj: (2)

3.3 Kvalita softwaru

Quality Assurance (QA) je proces nebo aktivita, která se zaměřuje na ověření a zajištění kvality produktu nebo služby. Primárním cílem tedy je předcházení situacím, kdy se produkt s neodhalenými nedostatky nebo chybami dostane k zákazníkům/uživatelům.

V oblasti vývoje softwaru QA zahrnuje procesy a metody, které se zaměřují na ověření kvality softwaru před jeho nasazením do produkčního prostředí. Tyto procesy mohou kromě samotného testování zahrnovat i například analýzu kódu, rizik, dokumentace a jiné. QA tým může také provádět audit a revizi procesů vývoje softwaru, aby se zajistilo, že jsou splněny mezinárodní standardy kvality (26).

QA má několik cílů:

1. Identifikace chyb a nedostatků v softwaru
2. Zlepšení kvality softwaru
3. Zvýšení spokojenosti uživatelů
4. Snížení nákladů spojených s opravami chyb

QA je důležitý proces, který se provádí během celého životního cyklu vývoje softwaru. To umožňuje včasnou detekci a opravu chyb, což vede k vyšší kvalitě a spolehlivosti softwaru.

3.4 Testování softwaru

Software je nedílnou součástí životů lidí v 21. století. Se softwarem se setkáváme na denní bázi ať už jde o mobilní telefony, počítače, nebo různá drobná elektronická zařízení. Software se zároveň stal součástí kritické infrastruktury – najdeme ho v nemocnicích, v elektrárnách, v dopravě a dalších oborech. Pro správné fungování softwaru je důležité, aby byl dodáván bez chyb. V případě, že v softwaru zůstanou odhalené chyby, následky můžou být fatální, jak jsme se mohli nejednou přesvědčit v minulosti (2).

Testování softwaru je dynamický a velmi různorodý obor. Je zároveň takřka nenahraditelnou součástí vývoje veškerého softwaru, protože chybám ve vývoji se nelze vyhnout. Testováním softwaru se pokoušíme zajistit požadovanou kvalitu výsledného produktu a jeho bezpečnost. Dle (19) je kvalita *softwaru míra, do které softwarový produkt splňuje stanovené a implicitní potřeby, je-li používán za stanovených podmínek.*

Proces testování zároveň zahrnuje tvorbu testovací dokumentace, analýzu, vytváření testovacích dat a reporting výsledků (1).

3.4.1 Axiomy testování softwaru

Základní principy (axiomy) testování se pokusilo definovat více autorů. Nejznámějším z nich je Ron Patton (1), který přišel ve své knize Testování softwaru s 10 základními axiomy testování, rozvedu zde 4 nejdůležitější z nich:

1. *„Žádný netriviální program není možné otestovat úplně."* V praxi se mnohokrát setkáme s názorem, že produkt lze plně otestovat a odhalit tak všechny chyby. Bohužel něco takového není možné, protože nelze předem předpokládat všechny možné stavy programu a zařízení, přechody mezi nimi, vlivy prostředí, vstupní hodnoty, chovní uživatele apod.
2. *„Testování je věcí odhadu rizika."* Protože není možné otestovat všechny kombinace situací, které mohou nastat, postupuje tester dle svého uvážení – tedy dle toho, v jakých kombinacích vidí potencionální riziko.
3. *„Testování může prokázat jen přítomnost defektů, nikoli jejich absenci."* Navazuje na první axiom – protože nejsme schopni otestovat u komplexních aplikací všechny varianty, bez ohledu na množství prostředků věnovaných testování nedokážeme odhalit všechny chyby a proto ani nelze dokázat jejich nepřítomnost.
4. *„Čím více defektů je nalezeno, tím více jich v produktu je."* Tento axiom pracuje s předpokladem, že se chyby vyskytují v určitých skupinách, protože lidé často dělají stejné chyby a jejich kvalita práce je proměnlivá.

V (3) jsou kromě výše zmíněných axiomů definovány další 3 důležité axiomy:

1. *„Včasné testování šetří čas a peníze."* Čím dříve je chyba odhalena, tím nižší jsou náklady na její opravu. Je proto důležité začít s testováním už v ranných fázích vývoje, aby se předešlo komplikacím v budoucnu.
2. *„Vyvarování se pesticidnímu paradoxu."* Pesticidní paradox popisuje jev, kdy při použití stále stejných metod nepřicházíme s novými výsledky, v tomto případě s nově objevenými chybami. Proto je důležité udržovat testy aktuální a obměňovat testovací data.
3. *„Testování je závislé na kontextu."* Proces testování softwaru je potřeba tvořit s ohledem na konkrétní projekt, zákazníka a metodiku vývoje.

3.4.2 Proces testování softwaru

Testování softwaru je komplexní proces. Zpravidla zahrnuje několik základních kroků:

1. Plánování: stanovení cílů testování, vytvoření testovacího plánu
2. Návrh: návrh testovacích případů na základě dodané dokumentace, vytváření testovacích scénářů
3. Testování: samotný proces testování a reporting průběžných výsledků
4. Vyhodnocení: analýza výsledků, zhodnocení aktuálního stavu aplikace
5. Oprava: opravy nalezených chyb
6. Regresní testování: testování pro ověření, zda nově implementované opravy nezpůsobily další chyby
7. Ukončení: ukončení testování a předání aplikace do produkčního prostředí

Konkrétní nastavení procesu se zpravidla odvíjí od typu daného projektu a fáze vývoje. Menší odlišnosti lze najít i mezi jednotlivými firmami a týmy. Z toho důvodu je uvedený přehled pouze orientačním základem, jak proces testování softwaru může vypadat.

3.4.3 Význam testování v procesu vývoje softwaru

Vyhnout se chybám při vývoji softwaru je takřka nemožné, viz podkapitola Příčiny vzniku softwarových chyb. V minulosti, kdy nebyla testování softwaru věnována přílišná pozornost, jsme byli svědky mnohdy fatálních selhání. V dnešní době, kdy je software běžnou součástí našich životů včetně kritické infrastruktury, je dostatečné testování důležitější než kdy dřív. Vhodně nastaveným procesem testování lze ale výrazně snížit množství chyb, které se dostanou do provozu a zvýšit tak kvalitu výsledného produktu. Důležitost procesu testování lze shrnout do třech hlavních důvodů: kvality, bezpečnosti a splnění požadavků zákazníka. V následujících odstavcích budou tyto důvody blíže vysvětleny včetně uvedení příkladů selhání v historii (2).

Kvalita

Kvalita softwaru úzce souvisí se splněním požadavků, ale zároveň ji částečně přesahuje. Ověřením kvality softwaru během vývoje se myslí ověření korektního fungování všech funkcionalit. Patří sem rovněž testování hraničních hodnot, zátěžové testování a testování na různých zařízeních. Pro důležitost testování produktu na různých zařízeních lze najít ukázkový příklad v historii.

V roce 1994 vydala společnost Disney multimediální hru na CD-ROM The Lion King Animated Storybook. Jednalo se o první hru z produkce Disney. Její propagaci byly věnovány značné prostředky, čemuž odpovídaly výsledné prodeje. Bohužel hra nebyla otestována na různých v té době dostupných osobních počítačích a operačních systémech. Toto pochybení způsobilo, že hra byla na mnoha osobních počítačích nefunkční a společnost Disney přišla v důsledku této chyby o značné zisky (1).

Bezpečnost

Bezpečnost se částečně překrývá s kvalitou softwaru, neboť zpravidla k jejím nedostatkům dochází vlivem nedostatečné kvality. Zvláštní prostor je jí ale věnován, protože se jedná o velmi důležitý aspekt, který testování softwaru zajišťuje. Na bezpečnost je kladen stále větší důraz, neboť se dnes setkáváme s různými typy softwaru takřka ve všech odvětvích včetně kritické infrastruktury.

Příkladem softwarové chyby, která narušila bezpečnost celého zařízení a bohužel skončila tragicky byl software obranného raketového systému Patriot v roce 1991. Jednalo se o systém obrany proti iráckým raketám během války v Perském zálivu. V systému byla kumulující se softwarová chyba v časování, která v důsledku vedla k velké nepřesnosti navádění a nedokázala tak zabránit dopadu raket. Následky byly fatální a stály život 28 amerických vojáků (1).

Splnění požadavků zákazníka

Na začátku procesu vývoje softwaru (viz kapitola Proces vývoje softwaru) jsou definovány požadavky na výsledný produkt. S těmi dále pracuje projektový manažer a rozděluje jednotlivé úkoly členům svého týmu. Při dělení práce do drobnějších celků snadno dojde vlivem lidského faktoru k drobným chybám a vývoj se tak mírně odchýlí od požadavků definovaných na začátku. V tomto případě je rolí softwarového testera, aby odchylky od zadání odhalil a výsledný produkt tak odpovídal všem požadavkům definovaným na začátku vývoje.

3.4.4 Role testera

Tester je důležitou součástí vývojového týmu. Jeho hlavní rolí je testování – tedy odhalování chyb a nedostatků vyvíjeného produktu. Tester musí být na základě dokumentace schopen identifikovat potenciální nedostatky a problémy. Na základě toho provede samotné

testování pro ověření, zda software pracuje korektně a splňuje definované požadavky. Proces testován může zahrnovat automatizované i manuální testy a jejich různé druhy.

Tester musí být také schopen komunikovat s ostatními členy týmu, aby mohl pochopit požadavky na software a poskytnout relevantní a užitečné informace o problémech a nedostatcích, které identifikuje (2). Kromě toho tester hraje důležitou roli při zlepšování procesu vývoje, například pomocí návrhu a implementace metod pro detekci chyb a analýzy výsledků testů.

V krátkosti, role testera je klíčová pro zajištění kvality a stability softwaru, pomáhá v týmu rozpoznávat a řešit problémy a spolupracuje na zlepšení procesu vývoje.

3.5 Chyby softwaru

Softwarová chyba je chyba způsobující nesprávné, nežádoucí nebo neočekávané výsledky chování programu. Chyby na sebe zpravidla navazují – chyba ve specifikaci zapříčiňuje chybu v kódu apod. Chybám se ve vývoji softwaru prakticky nelze vyhnout.

V české literatuře se v kontextu softwaru setkáme většinou s označením „chyba“, „defekt“, případně „vada“. V anglické literatuře je softwarová chyba zpravidla označována jako „bug“ nebo „error“.

Lze ale nalézt i detailnější rozdělení. Příkladem je rozdělení dle (2) na *defekty (faults), chyby (errors) a selhání (failures),* kde autor zdůrazňuje nezaměnitelnost těchto pojmů a nevhodné použití v praxi. Defekt je v tomto pojedí následkem lidského pochybení a je příčinou chyby. Chyba je stav systému, který může vést k selhání. Selhání je nekonzistence mezi aktuálním stavem systému a jeho specifikací. V této práci nicméně budeme pro zjednodušení používat pouze české označení „chyba“.

Nejčastější příčiny chyb dle (3):

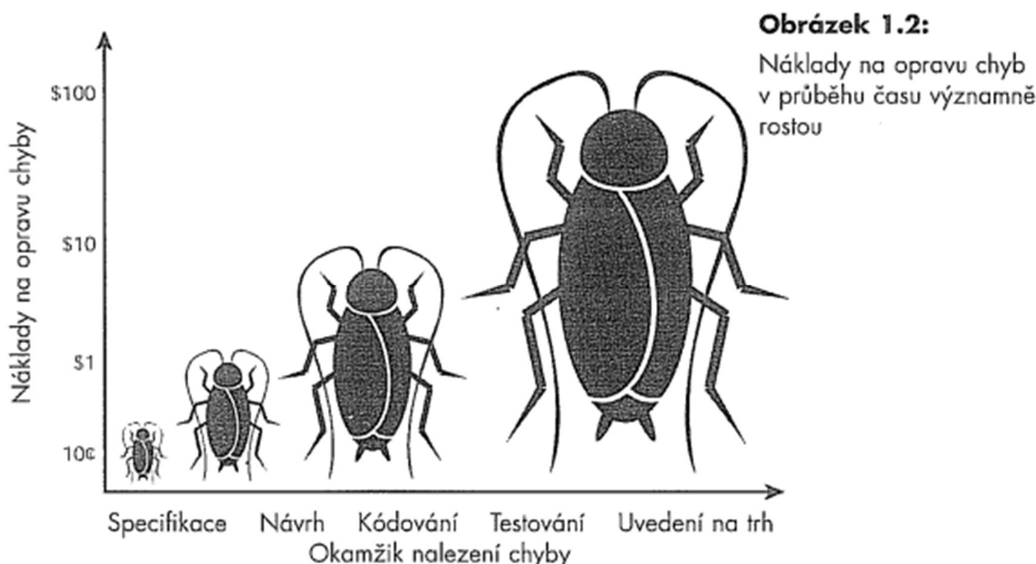
- Časová tíseň
- Lidská omylnost
- Nedostatečná kvalifikace/zkušenosti
- Nesprávná komunikace (v týmu nebo se zákazníkem)
- Složitost návrhu, kódu, architektury nebo technologie
- Neočekávané vnější okolnosti (nejčastěji ovlivňuje hardware)

3.5.1 Kořenové příčiny

Výše vyjmenované časté příčiny chyb jsou v mnoha případech pouze výsledkem chyb vzniklých hlouběji v procesu vývoje softwaru. Těmto tomuto prvopočátku chyb říkáme „kořenové příčiny".

Jejich zpětná analýza je důležitá především kvůli možnosti předcházení dalším podobným chybám v budoucnu a díky tomu šetření prostředků na vývoj. Prostředky zároveň

Obrázek 2 Graf zobrazující stoupající náklady na opravu chyby v průběhu času



Zdroj: (1)

významně šetří včasné odhalení chyby (viz obrázek 2). Její objevení může urychlit právě analýza kořenových příčin.

Ačkoliv by se mohlo zdát, že kořenovou příčinou chyb bude samotný kód, opak je pravdou. Dle (1) (viz obrázek 3) je nejčastější kořenovou příčinou chyb specifikace – především z důvodu jejího nedostatečného pokrytí jednotlivých funkcionalit výsledného produktu nebo její nepochopení (zde se kombinuje s lidským faktorem).

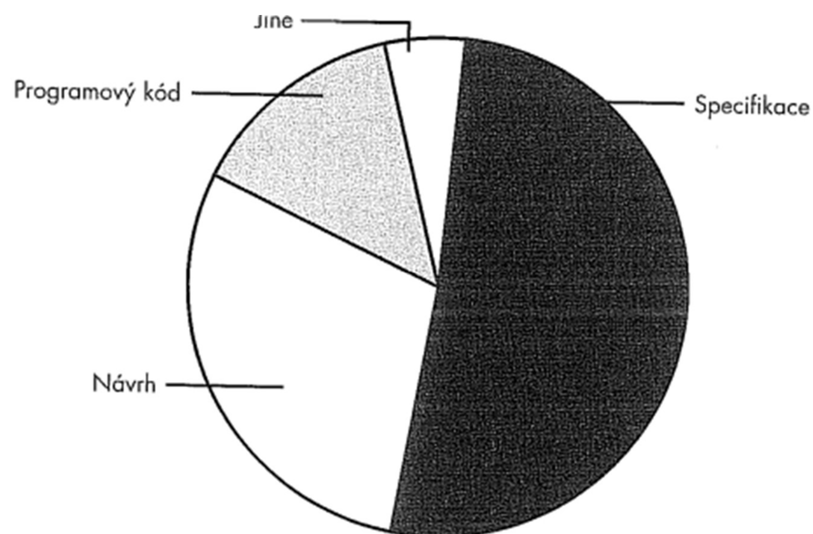
Druhým nejčastějším důvodem je návrh, který vytváří softwarový architekt. V praxi, především v agilním vývoji, dochází k častým změnám specifikace. Tyto změny nemusí být dostatečně reflektovány v návrhu, podle kterého následně programátoři postupují a výsledné chování aplikace tak neodpovídá požadavkům.

Vzhledem k tomu, že kořenová příčina je často specifikace nebo návrh a chyba v kódu až její důsledek, chyby způsobené čistě kódem jsou z hlediska četnosti kořenových

příčin na třetím místě. Pokrývají zhruba jednu osminu celkového počtu nalezených chyb. Jedná se většinou o chyby provázané s lidským faktorem, jako je nedostatek zkušeností nebo přehlédnutím důležitého prvku. Na tento typ chyb má také velký vliv časová tíseň, ke které na konci mnohých projektů v jejich závěru dochází.

Kategorie „Jiné“ pokrývá nesprávné předpoklad; nalezené chyby, které se nakonec ukázaly jako korektní chování softwaru; více nalezených chyb, které způsobila jedna příčina a další.

Obrázek 3 Graf zobrazující příčiny vzniku softwarových chyb



Zdroj: (1)

3.5.2 Klasifikace chyb

Chyby mají na celkové fungování systému různě vážný dopad. Pro jejich efektivní řešení je důležité vhodné třídění do kategorií dle priority. V knize Řízení kvality softwaru (2) se setkáme s rozlišováním pojmů závažnost a priorita. Závažnost je zde hodnocena jako míra dopadu chyby na fungování systému a priorita jako naléhavost řešení dané chyby. V prioritě je zároveň reflektována pravděpodobnost, s jakou k aktivaci chyby dojde. Ačkoliv má tento dvoustupňový způsob klasifikace své výhody, v praxi je s ohledem na zachování jednoduchosti nejčastěji používané hodnocení, ve kterém závažnost a priorita splývají.

Pro klasifikaci závažnosti chyb neexistuje žádná obecně platná škála. Toto hodnocení se proto může mezi jednotlivými společnostmi a týmy výrazně lišit.

Často používaná škála závažnosti, kterou mimo jiné zmiňuje i (2) je čtyřstupňová:

- **Kritická**: znemožňuje fungování celkového systému, jeho důležité části nebo představuje velké bezpečnostní riziko. Často se jedná o chyby vedoucí k havarování systému.
- **Vysoká**: znemožňuje fungování dílčí funkcionality, ale systém jako takový lze v omezené míře používat. Např. nefunkční přidání zboží do košíku na e-shopu.
- **Střední**: chyba neomezuje žádnou důležitou funkcionalitu. Např. nemožnost prohlédnout si detail zboží na e-shopu.
- **Nízká**: chyba neovlivňuje funkčnost systému. Jedná se povětšinou o drobné chyby v textaci nebo uživatelském rozhraní (barvy, umístění prvků, zarovnání, velikost atd.)

Od závažnosti chyby se zpravidla odvíjí priorita, s jakou je přistupováno k jejich řešení. Pokud se jedná chyby odhalené během vývoje, řeší se postupně od kritických ke středním. K řešení chyb s nejnižší závažností se mnohdy přistupuje až na samotném konci vývoje.

Priorita řešení se může v průběhu vývoje mírně měnit v závislosti na čase, rozhodnutí zákazníka, zdrojích atd. V dřívějších fázích často slouží priorita pro seřazení chyb za účelem zajištění plynulého pokračování vývoje a testování. S blížícím se datem vydání softwaru bývá priorita upravována dle výše zmíněných faktorů. V praxi bývá například na konci vývoje přiřazena vyšší priorita chybám uživatelského rozhraní, které byly v dřívějších fázích upozaděny.

3.6 Testovací dokumentace

Testovací dokumentace je soubor dokumentů, které popisují proces testování softwaru a výsledky těchto testů. Slouží jako příručka pro vývoj a testování softwaru. Obsahuje informace o testovacích plánech, scénářích, případech, datech, postupech a metodách testování, výsledcích testů a závěrečné zprávě. Testovací dokumentace je důležitá pro kvalitu softwaru, protože pomáhá identifikovat a opravit chyby a nedostatky v softwaru před jeho uvedením do provozu. Slouží také jako důkaz o tom, že byl software důkladně testován a že splňuje požadavky a specifikace (3).

Testovací dokumentace by měla být připravena již v prvních fázích vývoje softwaru, aby bylo zajištěno, že testování probíhá efektivně a že jsou identifikovány všechny chyby a

nedostatky včas. Měla by být aktualizována a revidována po každém testovacím kroku, aby byly zaznamenány všechny změny a aby bylo zajištěno, že je dokumentace aktuální a relevantní (3).

3.6.1 Testování dokumentace

Testování softwarové dokumentace je proces, který se zaměřuje na ověření správnosti, přesnosti a srozumitelnosti dokumentace týkající se softwarového projektu. Je to důležitý krok v procesu vývoje softwaru, který pomáhá zajistit, že dokumentace odpovídá požadavkům a specifikacím softwaru a že je pro uživatele snadno použitelná.

Existuje několik druhů testování softwarové dokumentace, těmi základními jsou (26):

- **Technická revize:** Ověřuje se správnost jazyka, gramatiky, formátu a použití standardních označení. Tato revize se zaměřuje na to, aby byla dokumentace stylisticky a gramaticky správná, aby byla snadno čitelná a aby odpovídala předepsaným standardům.
- **Funkční revize:** Ověřuje se, zda dokumentace odpovídá požadavkům a specifikacím softwaru. Tato revize se zaměřuje na to, aby dokumentace popisovala správně funkce a vlastnosti softwaru a aby byla v souladu s plánem projektu.
- **Usability test**: Ověřuje se, zda dokumentace je pro uživatele srozumitelná a snadno použitelná. Tato revize se zaměřuje na to, aby dokumentace byla jasná a srozumitelná pro uživatele, aby jim umožnila snadno se seznámit s funkcemi a vlastnostmi softwaru.
- **Revize konzistence:** Ověřuje se, zda dokumentace je konzistentní v rámci celého projektu. Tato revize se zaměřuje na to, aby byla dokumentace konzistentní ve stylu, terminologii a formátu po celou dobu projektu.

3.6.2 Testovací plán

Testovací plán je stěžejní dokument pro celý proces testování. Tvoří se v prvních fázích vývoje, kdy je hotová specifikace a návrh softwaru, ale samotné testování ještě nezačalo. Jeho účelem je specifikovat průběh procesu, kterým bude kvalita softwaru ověřena (16).

Testovací plán by dle (17) měl obsahovat:

- **Rozsah testování**: ne vždy je potřeba testovat celý systém, někdy se jedná pouze o testování nové funkcionality/části systému
- **Seznam testovacích případů**: seznam testovacích případů s jejich identifikátory
- **Harmonogram testování**: plán, podle kterého se budou funkcionality testovat, datum začátku testování a datumy do kterých mají být dodány dílčí výsledky testování
- **Alokace lidských zdrojů:** definování, kdo bude na daném testování pracovat, případně s odhadem, kolik testování zabere času
- **Prostředí:** definice testovacího prostředí, konfigurace a dat, která budou k testování potřeba
- **Nástroje:** které nástroje se budou používat pro testování, reportování chyb a další aktivity
- **Defekt management:** jak budou nalezené chyby reportovány, komu budou předávány, co vše je potřeba v reportu dodávat (screenshoty, logy, videa, kód,..)
- **Risk management:** seznam rizik, jaká mohou nastat během testování, jaká rizika hrozí pokud bude software vydán bez otestování/nedostatečně otestovaný
- **Výstupní parametry**: informace, kdy musí být testování ukončeno, očekávané výsledky, které mohou být na konci srovnány s reálnými výsledky

3.6.3 Případ užití (use case)

Případy užití jsou narozdíl od testovacích případů a testovacích scénářů spjaté nejenom s testováním, ale především se samotným procesem vývoje softwaru. Obvykle bývají součástí softwarové specifikace. Slouží k popisu funkcionalit skrze příklady jejich využití. Zpravidla jsou formulované jako interakce mezi uživatelem a systémem. Pro zjednodušení je lze zobrazit pomocí diagramů (16).

V procesu testování softwaru slouží případy užití jako základ pro tvorbu testovacích scénářů. Pokud jsou funkcionality softwaru dostatečně pokryté příklady užití, výrazně se snižuje šance, že bude při tvorbě testovacích scénářů některá z funkcionalit opomenuta. Na jeden případ užití zpravidla připadá více testovacích scénářů.

3.6.4 Testovací případ (test case)

Dle (3) je testovací scénář definován jako: *„Sada vstupních podmínek, vstupů, očekávaných výsledků, výstupních podmínek a případně akcí, která je vypracována na základě testovacích podmínek.“* V praxi jsou testovací případy soubory kroků a jejich očekávaných výsledků aplikovaných na předem definovaném stavu aplikace a datech. K tvorbě testovacích případů přistupujeme po vytvoření testovacího plánu a definici případů užití.

3.7 Kategorie testů

Je důležité rozdělit si a charakterizovat jednotlivé kategorie testů. Existuje mnoho způsobů, jak testy dělit. Pro účely obecného porozumění problematice jsem zvolila rozdělení dle toho, v jaké fázi vývoje se provádí a rozdělení na bílou a černou skříňku. Zároveň považuji za důležité pro účely této práce přiblížit pojem regresní testování, které je realizováno v praktické části.

3.7.1 Testování dle fáze vývoje

Dle fáze, v jaké k realizaci testování přistupujeme lze testy dělit na (2):

1. **Testování jednotek (unit testing)**: testování jednotek provádí zpravidla programátor, jedná se o testování na nejnižší úrovni prováděné během vývoje. Slouží k základnímu ověření korektního fungování vyvíjené funkcionality. V tomto případě se jednotky testují izolovaně, bez interakce s ostatními jednotkami.
2. **Integrační testování:** k integračnímu testování přistupujeme ve chvíli, kdy máme dokončené alespoň 2 moduly softwaru otestované jednotkovým testováním a můžeme tedy testovat jejich fungování dohromady. Fáze integračního testování je z celého procesu testování nejdelší.
3. **Systémové testování:** systémové testování je realizováno ve fázi, kdy je dokončený stabilní systém. Účelem systémového testování je ověření, zda systém splňuje veškeré požadavky stanovené zákazníkem a jestli funguje jako celek. Pod systémové testování spadají zároveň specifické podkategorie testů, jako jsou například:

   3.1. Bezpečnostní testy: slouží k ověření, zda nedochází k úniku dat, zda jsou data dostatečně chráněná a šifrovaná.

3.2. **Testy robustnosti:** ověřují, zda se systém chová korektně i pokud pracuje s chybnými vstupy a neočekávanými situacemi spojenými s prostředím, ve kterém systém pracuje.

3.3. **Zátěžové testy:** slouží k ověření, zda systém zůstává stabilní i pokud je vystavený zátěži. Tou může být například velký počet uživatelů, kteří ho používají současně, mnoho požadavků zaslaných na API a podobně.

3.4. **Testy hraniční zátěže:** zkoumají chování systému během extrémní zátěže, vyšší než u zátěžových testů. Cílem je ověření hranic výkonosti softwaru.

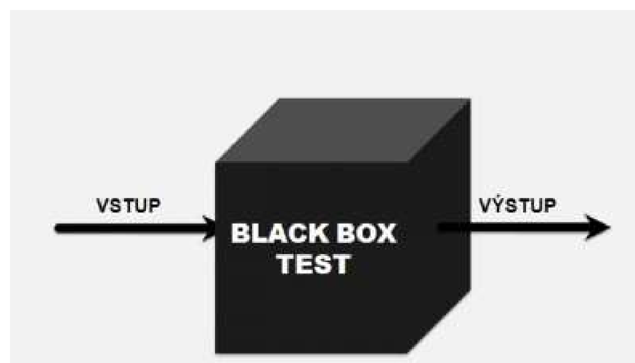
3.5. **Testy spolehlivosti:** vyhodnocují, jak dlouho a za jakých podmínek je systém schopný pracovat bez selhání.

4. **Akceptační testování:** poslední fáze procesu testování během vývoje softwaru. Je obvykle realizováno zákazníkem nebo jím vybranou třetí stranou. Slouží zákazníkovi k ověření, zda výsledný produkt splňuje všechny požadavky definované na začátku a během vývoje.

3.7.2 Black box vs white box

Black box testování je metoda testování softwaru, která se zaměřuje na testování funkčnosti aplikace bez ohledu na její interní implementaci. Při black box testování se tester soustředí na to, zda aplikace generuje pro dané vstupy korektní výstupy, ale nezkoumá její interní stav. Při testování se vychází primárně ze specifikací uvedených v dokumentaci. Tento přístup se využívá především u funkčního, integračního, regresního, bezpečnostního a zátěžového testování. Výhodou této metody je její snadnost (tester zpravidla nepotřebuje hluboké znalosti z oblasti vývoje softwaru), rychlost a transparentnost (12).

Obrázek 4 Schématické zobrazení Black box testování



Zdroj: (12)

White box testing, také nazývaný jako glass box nebo clear box testing, se naopak zaměřuje na testování v závislosti na interních částech aplikace. Soustředí se na testování jednotlivých částí kódu, jako jsou funkce, procedury a třídy. Tester má kompletní přehled o tom, jak se vstupy převádějí na výstupy a může tak lépe testovat jednotlivé části kódu. White box testing také umožňuje testovat kód z hlediska jeho kvality. Využíváme ho především při unit testování, kontrole kvality kódu, testování pokrytí kódu a debuggingu. Výhodou je, že chyby se zpravidla objeví dříve než při black box testování. Nevýhodou je ovšem náročnost jak z hlediska požadavků na schopnosti testera, tak časová náročnost. Na tu se zároveň váží vyšší náklady (13).

V některé literatuře se také můžeme setkat s pojmem „gray box testing“, který kombinuje charakteristiky obou výše uvedených přístupů (14).

Tabulka 1 Porovnání black box a white box testování

Black box	White box
Zaměřuje se na vstupy a výstupy	Zaměřuje se na interní implementaci
Nevyžaduje znalost kódu	Vyžaduje znalost kódu
Provádí se v pozdějších fázích vývoje nebo po dokončení	Lze provádět již během vývoje
Používá se k testování celkové funkčnosti aplikace	Používá se k testování jednotlivých funkcí a metod
Výhody: snadnost, rychlost, transparentnost, testovací dokumentaci lze vytvořit s předstihem dle specifikace	Výhody: Včasné odhalení chyb, zvýšení kvality kódu
Nevýhody: neodhalí nedostatky kódu, nemusí odhalit nespecifikované chování aplikace	Nevýhody: náročnost, náklady

Zdroj: Autor

3.7.3 Regresní testování

K regresnímu testování přistupujeme ve chvíli, kdy do hotového systému přidáváme novou funkcionalitu a ověřujeme, zda i po této změně dříve vyvinuté moduly fungují korektně. Regresní testování je velmi podobné integračnímu. Rozdíl mezi nimi je ve fázi vývoje, ve které se testování realizuje. Integrační testování probíhá během vývoje softwaru jako celku, k regresnímu testování přistupujeme, když máme hotový software, který dále rozšiřujeme (2). Regresní testy bývají často automatizovány, protože se mnohdy opakují na denní bázi.

3.8 Automatizované testování softwaru

Automatizované testování softwaru je proces, při kterém se pomocí automatizovaných nástrojů a skriptů provádějí testy na software. Cílem je snížit manuální práci a urychlit proces testování. Automatizované testy jsou napsány v programovacím jazyce a jsou spouštěny pomocí speciálního automatizačního nástroje. Tyto testy se opakují snadno a rychle a jsou schopné detekovat chyby, které by mohly uniknout manuálnímu testování. Automatizace testů umožňuje také snadné srovnání výsledků testů v různých verzích softwaru a snadné sledování a reporting výsledků. Automatizace testů se často využívá pro rutinní a opakující se testy, jako jsou testy pokrytí kódu, testy regresní, testy performance a testy integrace.

3.8.1 Význam automatizace

Automatizace testování softwaru se stala nezbytnou součástí vývoje software, jelikož přináší mnoho výhod, které pomáhají zlepšovat kvalitu a efektivitu celého procesu. Její největší přínos je ve zvýšení rychlosti a produktivity.

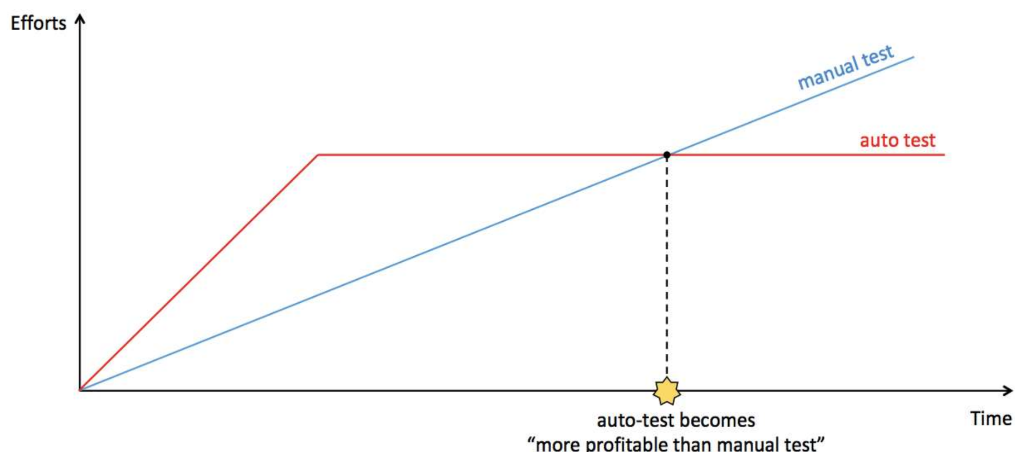
Testování softwaru může být v případě manuálního testování velmi časově náročný proces. Člověk – tester je schopen vyvinout pouze omezenou rychlost práce, která je v čase proměnlivá. Zároveň do procesu zasahují externí vlivy jako nemoci, dovolené, které pracovníka dočasně z procesu vyřadí. Automatizované testy jsou na těchto vlivech nezávislé – může je pouštět takřka kdokoliv nebo lze jejich spuštění nastavit předem. Rychlost jejich běhu je při optimálně napsaném kódu několikanásobně vyšší než v případě manuálního testování. Díky tomu lze celkový čas potřebný k ověření funkčnosti aplikace výrazně zkrátit.

Dále automatizace testování zvyšuje kvalitu. Automatizované testy jsou přesnější a méně náchylné k lidské chybě než manuální ruční testy. Tyto testy jsou napsány přesně podle

specifikací a předpisů a jsou opakovaně spouštěny, což znamená, že se snižuje riziko opomenutí kritických částí aplikace nebo přehlédnutí chyby. Dochází zde ale k takzvanému paradoxu pesticidů (39), kdy kvůli dokola opakovanému, zdánlivě funkčnímu procesu můžeme přehlédnout jeho v čase vzniklé nedostatky, pročež jim testování nepřizpůsobíme.

Automatizace testování také snižuje náklady. Díky automatizaci se snižuje potřeba lidských zdrojů na provádění manuálního testování. Tento faktor se může stát klíčovým při hodnocení nákladů v projektech s omezeným rozpočtem. Protiváhou k těmto úsporám je ale větší náročnost na kvalifikaci daného pracovníka, s čímž jsou spojené jeho vyšší nároky na finanční ohodnocení.

Graf 1 Porovnání profitability manuálního a automatizovaného testování v čase



Zdroj: (11)

Graf 1 znázorňuje profitabilitu automatizovaného testování. Ačkoliv jsou v počátečních fázích náklady na automatizované testování vyšší než na manuální, v průběhu času se ustálí a nezvyšují se. Oproti tomu náklady na manuální testování se s dalším růstem aplikace a tím i delším procesem testování, zvyšují. Považuji ovšem za důležité zmínit, že k bodu, když se automatizované testování stává výhodnějším dochází v různém čase, v závislosti na konkrétním projektu. Tvrzení, že je automatizované testování vždy výhodnější je tedy pravdivé jen z části. Může se stát, že automatizace se stane výhodnější v tak dlouhém časovém horizontu, že pro daný projekt nebude výhodná.

Dalším přínosem automatizace je schopnost průběžně testovat. Automatizované testy mohou být prováděny v různých fázích vývoje, což umožňuje průběžné ověřování

funkčnosti aplikace a zajišťuje, že se chyby podaří nalézt brzy. Kromě toho je lze provádět takřka kdykoliv. Na rozdíl od pracovníků tedy nejsou limitovány aktuálními časovými možnostmi.

3.8.2 Jak automatizovat

Best practices automatizovaného testování softwaru zahrnují řadu principů a postupů, které pomáhají vytvářet efektivní a kvalitní automatizované testy. V této podkapitole vycházím především z knihy (24) a článku (25) zaměřeného na Cypress.

Best practices zahrnují následující:

- **Definování cílů a účelů automatizace:** Je důležité mít jasně stanovené cíle a účely automatizace, aby bylo možné vybrat správný přístup k testování a vytvořit testy, které budou odpovídat vašim požadavkům.
- **Plánování a design testů:** Automatizované testy by měly být pečlivě navrženy a plánovány tak, aby bylo zajištěno, že budou testovat správné funkce a části aplikace. Tento proces by měl být prováděn s pečlivou pozorností k detailům, aby bylo zajištěno, že testy budou dostatečně komplexní a že budou testovat správné funkce.
- **Výběr vhodného nástroje pro automatizaci**: Je důležité vybrat vhodný nástroj pro automatizaci, který odpovídá vašim požadavkům na testování a který má funkce, které jsou pro vás důležité.
- **Výběr vhodných testovacích scénářů**: Je důležité vybrat správné testovací scénáře, které budou testovat různé funkce a části aplikace a které budou zahrnovat různé typy testů, jako jsou například funkční, integrační a zátěžové testy.
- **Maintainability:** Automatizované testy by měly být navrženy tak, aby byly udržitelné a aby bylo možné je opakovaně používat a upravovat. Tuto udržitelnost lze dosáhnout například pomocí dobrého kódování, dobré dokumentace a správného řízení verzí.

3.8.3 Nástroje pro automatizaci testování softwaru

Výběr vhodného nástroje pro automatizované testování je klíčový pro optimální provedení automatizace. Existuje mnoho různých nástrojů pro automatizované testování. Největší rozdíly najdeme ve využívaných programovacích jazycích, dostupných typech testů a prohlížečích ve kterých se testy spouští. Dalšími, méně významnými rozhodovacími kritérii může být cena (ačkoliv je většina testovacích nástrojů open source), integrace na

další nástroje, možnosti reportingu, individuální zkušenosti a jiné. Mezi nejznámější nástroje pro automatizované testování patří Cypress, Selenium, Appium, SoapUI, WebDriver IO a mnoho dalších. Každý z těchto nástrojů má své výhody a nevýhody, a proto je důležité důkladně zvážit, jaký nástroj vyhovuje potřebám konkrétního projektu.

Cílem této podkapitoly je poskytnout základní přehled o dvou nejpoužívanějších nástrojích pro automatizované testování. Budou popsány hlavní funkce a vlastnosti těchto nástrojů, jejich výhody a nevýhod.

3.8.4 Selenium

Selenium je open-source nástroj pro automatizované testování webových aplikací. Byl vyvinut v roce 2004 a od té doby se stal jedním z nejpopulárnějších nástrojů pro testování webových aplikací (35).

Selenium umožňuje vytváření skriptů pro automatizované testování webových aplikací v mnoha programovacích jazycích jako je Java, Python, Ruby nebo C#. Jednou z hlavních výhod Selenium je jeho schopnost testovat aplikace v reálném prohlížeči, což umožňuje testovat aplikace tak, jak by je používali skuteční uživatelé. Aplikace je možné testovat na různých platformách a prohlížečích. Tím se zvyšuje flexibilita a komplexnost testování (34). Díky velké popularitě a široké uživatelské základně je kromě oficiální dokumentace k dispozici mnoho dalších zdrojů informací (35).

Mezi nevýhody Selenium patří jeho složitost pro začátečníky, kteří se s nástrojem teprve seznamují. Kromě toho vyžaduje Selenium také určitou úroveň technických znalostí a schopností pro jeho používání.

Vzhledem k jeho schopnostem a popularitě je Selenium významným nástrojem pro automatizované testování webových aplikací a stále se používá v mnoha softwarových projektech po celém světě (35).

Tabulka 2 Výhody a nevýhody Selenia

Výhody	Nevýhody
Flexibilita	Náročnost
Široká podpora prohlížečů	Nízká rychlost
Vhodný pro projekty všech velikostí	Pouze pro webové aplikace
Integrace na CI/CD	

Zdroj: (35)

3.8.5 Cypress

Cypress je open-source nástroj pro testování webových aplikací, který se zaměřuje na automatizaci testování uživatelského rozhraní. Je možné v něm nasimulovat komplexní průchod aplikací a tím ji testovat jako celek. Zároveň ale lze testovat i jednotlivé funkce/komponenty. Cypress tak umožňuje programátorům a testerům snadno vytvářet, spouštět a ladit testy pro webové aplikace (21).
Jedním z hlavních důvodů, proč se Cypress stal populárním nástrojem pro testování webových aplikací, je jeho jednoduchost - je navržen tak, aby byl snadno použitelný a přístupný i pro uživatele bez hlubších znalostí programování.

Cypress běží přímo v prohlížeči, což umožňuje přímý přístup k DOM (Document Object Model) a všem jeho prvkům. Tento přístup umožňuje snadno testovat dynamické webové stránky. Nabízí také vestavěnou asynchronní architekturu, která umožňuje snadno ovládat asynchronní kód v testovacím prostředí. V neposlední řadě také umožňuje integrace s nástroji pro CI/CD. Pro reporting je možné využít další z nabízených integrací (Allure, Mochawesome) nebo Cypress Dashboard. Testy lze psát v JavaScriptu nebo TypeScriptu (21).

Pro Cypress existuje mnoho informačních zdrojů. Mezi nejlepší zdroje patří oficiální dokumentace na webových stránkách nástroje (https://docs.cypress.io/), kde lze nalézt podrobné informace o všech funkcích a obecném použití Cypressu. Dalším zdrojem je blog Cypress (https://www.cypress.io/blog/). Existuje také mnoho nezávislých blogů a videí, které nabízejí tipy pro používání tohoto nástroje. Díky intuitivnímu a jednoduchému rozhraní, obsáhlé přehledné dokumentaci a velké uživatelské základně se snižují počáteční nároky na užití této technologie (22).

Tabulka 3 Výhody a nevýhody Cypressu

Výhody	Nevýhody
Snadná instalace a použití	Omezená podpora pro cross-browser testování
Mnoho funkcí a příkazů	Omezené pokrytí funkčních testů
Přesnost, rychlost	Pouze pro webové aplikace
Přehledná, obsáhlá dokumentace	Není vhodný pro velké projekty

Zdroj: (21)

4 Praktická část práce

Bakalářská práce se věnuje automatizaci testování webové aplikace vyvíjené ve společnosti Quanti s.r.o. Z výsledků vícekriteriální analýzy vyplynulo, že pro automatizaci bude nejvhodnější využití Cypressu. V následujících kapitolách budou rozebrány všechny praktické aspekty tvorby automatizace, její samotná realizace a implementace řešení. Jeho přínosnost bude následně vyhodnocena porovnáním s aktuálně používaným manuálním testováním.

4.1 Představení aplikace a důvod automatizace

Aplikace, ve které je automatizované testování realizováno se nazývá Finanční informační manažerský systém, zkráceně FIMS3. Jedná se o interní aplikaci vyvíjenou společností Quanti s.r.o., konkrétně jde o její třetí verzi. Aplikace je používaná v anglickém jazyce. Ukázky jejího uživatelského rozhraní lze nalézt v příloze na obrázcích 9 a 10.

FIMS3 slouží ke správě firemních financí, projektů a pracovníků. Aktuální verze rozlišuje 3 uživatelské role – admin, manager a user. Admin má přístup ke všem datům a má právo data vytvářet, editovat a mazat. Manager má přístup pouze ke správě projektů a má povoleno nahlédnou do faktur, které jsou na projekty napárované. Správu financí nemá povolenu. Role managera má taktéž omezený přístup k informacím o zaměstnancích. Může vidět záznamy o sobě a zaměstnancích, kteří jsou v jeho týmu. User má přístup pouze k datům o sobě tzn. zaměstnanci samotném. Podrobněji se v této práci uživatelskými rolemi zabývat nebudeme, protože pro automatizované scénáře nejsou relevantní.

Aplikace je rozdělena do několika dílčích sekcí::

1. **Invoices** – správa příchozích (income) a odchozích (outcome) faktur a dobropisů (credit notes), sledování jejich stavu;
2. **Projects** – obsahuje čtyři typy projektů – budget, tariff, internal budget a internal long-term, správu aktuálně vyvíjených a ukončených projektů, eviduje aktuální stav vývoje, prostředky na vývoj, vypočítává budoucí finanční náročnost, profitabilitu a výdaje spojené s vývojem, k projektům lze připojit fakturu, k projektu se

synchronizují odpracované hodiny z Jiry[1] a to přes separátní komponentu Jira Connector;

3. **Workers** – správa personálních záležitostí, eviduje hodinovou mzdu, odpracované hodiny celkově i na jednotlivých projektech, vyplacené mzdy, zaměstnance lze přiřazovat k jejích týmovým vedoucím.

4.2 Požadavky na automatizaci

K požadavku na automatizaci aplikace došlo z důvodu důležitosti jejího korektního fungování. Firemní finance jsou kritická oblast, chyby zde mohou mít velký dopad na fungování firmy. Proto je důležité časté regresní testování. Pokud by ale bylo na denní bázi prováděno manuálně, stalo by se velmi nákladným a náchylným na chyby způsobené lidským faktorem.

Provedená automatizace bude pokrývat 5 případů užití z oblasti projektů a jejich vazby s fakturami. Na tyto testovací případy se váže 40 testovacích scénářů. Každému scénáři odpovídá jeden test. Automatizované testy budou sloužit pro regresní testování a budou spouštěny každý den. Data pro testování se budou generovat v samotných testech a po jejich skončení se smažou.

Technologie vybraná pro automatizaci musí umožňovat psaní rychlých a stabilních testů, neměla by příliš zvýšit personální náročnosti testování a musí umožňovat integraci s GitLabem.

4.3 Výběr testovacího nástroje

Existuje mnoho nástrojů pro automatizaci testování softwaru. Na základě uživatelských recenzí a žebříčků popularity byly do užšího výběru zvoleny 4 testovací nástroje: Cypress, Selenium, Playwright a WebDriver IO.

Cypress je moderní testovací nástroj, který umožňuje psát rychlé a stabilní testy pomocí JavaScriptu a TypeScriptu. Nabízí komplexní rozhraní pro psaní a spouštění testů, integraci s dalšími nástroji včetně CI/CD nástrojů a podporu pro paralelní a distribuované testování. Přestože se jedná o relativně nový testovací nástroj, poskytuje rozsáhlou, přehlednou dokumentaci, čímž výrazně ulehčuje začátky juniorním testerům. Jeho nevýhodou je, že se při použití na velkých projektech stává nepřehledným. Kromě toho

[1] Jira je software pro řízení projektů užívaný především v agilním vývoji. Obsahuje zpravidla správu úkolů a evidenci chyb a problémů (36).

funguje pouze s prohlížeči Chrome a Electron prohlížeči, což může být pro některé projekty omezením. Tyto nevýhody by nicméně na vybraný projekt neměly příliš velký dopad – jedná se o malý projekt využívaný především v prohlížeči Google Chrome (20).

Selenium je starší a široce používaný testovací nástroj, který umožňuje psát testy v mnoha programovacích jazycích a spouštět je na různých prohlížečích a platformách. V tomto ohledu žádný z testovacích nástrojů neposkytuje více možností. Jeho výhodou oproti konkurenčním nástrojům je umožnění cross-browser testování. Obecně Selenium poskytuje vysokou flexibilitu a rozšiřitelnost, což ale znamená vyšší složitost a nižší stabilitu. Nevýhodou je, že se obecně jedná o poměrně komplikovaný nástroj, který do začátku vyžaduje určitou úroveň znalostí programování. Oproti Cypressu má mírné nevýhody i v samotném užívání – na rozdíl od Cypressu neumožňuje automatické scrollování a čekání na elementy. Vzhledem k požadavkům na automatizaci nebylo Selenium pro využití na projektu vyhodnoceno jako vhodné. Příliš by zvýšilo personální náročnost a složitost testovacího procesu.

Playwright je nový testovací nástroj, který umožňuje psát testy v JavaScriptu, Pythonu a C# a spouštět je na Chrome, Firefox, Safari nebo WebKit prohlížeči. Playwright využívá stejnou architekturu jako testovací nástroj Puppeteer. Nabízí podporu pro moderní webové funkce jako např. interakce s klávesnicí nebo myší, simulaci geolokace nebo mobilních zařízení. Stejně jako Cypress podporuje automatické čekání na element. Jeho velkou nevýhodou ovšem je, že na rozdíl od Cypressu a Selenia neposkytuje nativní integraci s CI/CD nástroji, čímž nesplňuje jeden z požadavků na automatizaci. Také je často méně stabilní než Cypress (42).

WebDriver IO je JavaScriptová knihovna založená na Selenium WebDriveru, která usnadňuje psaní synchronních nebo asynchronních test. Stejně jako Cypress a Playwright poskytuje automatické čekání na elementy. Má také mnoho funkcí na zachycení síťových událostí, jejich trasování a umožňuje cross-browser testing. Kromě toho WebDriver IO nabízí integraci s různými službami jako např. Sauce Labs nebo BrowserStack, bohužel stejně jako Playwright nemá nativní integraci s CI/CD nástroji. Také podobně jako Selenium vyžaduje výchozí znalosti programování (42).

Na základě uvedeného porovnání byl jako nejvhodnější testovací nástroj vybrán Cypress - nabízí nejlepší kombinaci rychlosti, stability, rozhraní, integrace a distribuce testů a nejlépe splňuje požadavky na automatizaci. Jeho nevýhody budou mít na vybraný projekt takřka minimální vliv a jeho výhody budou maximálně zužitkovány.

4.4 Případy užití

Pro tuto práci bylo vybráno pět případů užití. V této kapitole budou všechny uvedeny a v krátkosti vysvětleny. Tři z nich budou poté rozepsány podrobněji.

- **Vytvoření projektu**: uživatel přejde do části aplikace s názvem Projects určené pro správu projektů, vyplní formulář k vytvoření nového projektu a projekt uloží. Po uložení je přesměrován na detail nově vytvořeného projektu.
- **Úprava projektu:** uživatel může v detailu projektu přejít k jeho editaci. Systém zobrazí editační formulář. Uživatel změní hodnoty v polích a potvrdí je. Systém zobrazí detail projektu s upravenými hodnotami.
- **Připojení faktury k projektu**: v části aplikace s názvem Invoices určené pro správu faktur, lze v detailu faktury připojit fakturu k projektu.. Pokud je k projektu připojená faktura, pole s hodnotami, které připojená faktura ovlivňuje (náklady, profitabilita) se automaticky přepočítají. Systém zobrazí připojenou fakturu v detailu projektu.
- **Úprava faktury připojené k projektu:** z detailu faktury lze přejít do formuláře pro její editaci. Po uložení úprav je uživatel přesměrován zpět na detail faktury. Systém po uložení úprav automaticky přepočítá hodnoty v projektu.
- **Odpojení faktury z projektu**: v detailu faktury může uživatel smazat spojení mezi projektem a fakturou. Po odpojení projekt zmizí z detailu faktury a systém automaticky přepočítá hodnoty v projektu. Faktura zmizí i z detailu projektu.

4.4.1 Případ užití: Vytvoření projektu

Název případu užití: Vytvoření projektu

Popis: Případ užití popisuje scénář vytvoření nového projektu v aplikaci FIMS3. Popsaný případ pro zjednodušení pracuje se sekcí Budget projects, ale analogicky lze aplikovat na všechny typy projektů.

Aktéři: Uživatel s rolí administrátor

Kroky:

1. Uživatel klikne na záložku „Budget projects“ v hlavní nabídce v sekci Projects
2. Systém zobrazí přehled vytvořených projektů
3. Uživatel klikne na tlačítko „New budget project“
4. Systém přesměruje uživatele na formulář pro vytvoření nového projektu

5. Uživatel vyplní všechna povinné pole formuláře
6. Uživatel klikne na tlačítko „Create“
7. Systém ověří, zda jsou všechny údaje správně vyplněné
8. Pokud jsou údaje správné, systém vytvoří nový projekt a přesměruje uživatele na jeho detail
9. Pokud některé údaje chybí, uživatel je na tuto skutečnost upozorněn chybovou hláškou a projekt se neuloží

Výstup: Projekt byl vytvořen

4.4.2 Případ užití: Úprava projektu

Název případu užití: Úprava projektu

Popis: Případ užití popisuje scénář úpravy projektu v aplikaci FIMS3. Popsaný případ pro zjednodušení pracuje se sekcí Budget projects, ale analogicky lze aplikovat na všechny typy projektů.

Aktéři: Uživatel s rolí administrátor

Kroky:

1. Uživatel klikne na záložku „Budget projects“ v hlavní nabídce v sekci Projects
2. Systém zobrazí přehled vytvořených projektů
3. Uživatel klikne na jeden z projektů v seznamu (sloupec Name, nebo ikona s lupou na konci řádku)
4. Systém přesměruje uživatele na detail vybraného projektu
5. Uživatel klikne na tlačítko „Edit“
6. Systém přesměruje uživatele na formulář pro úpravu projektu
7. Uživatel změní hodnoty v některých polích
8. Uživatel klikne na tlačítko „Update“
9. Systém ověří, zda jsou všechny údaje správně vyplněné
10. Pokud jsou údaje správné, systém změní hodnoty v projektu a přesměruje uživatele na jeho detail
11. Pokud jsou údaje nesprávné, uživatel je na tuto skutečnost upozorněn chybovou hláškou a editační formulář se neuloží, dokud nebude správně vyplněn

Výstup: Projekt byl upraven

4.4.3 Případ užití: Připojení faktury k projektu

Název případu užití: Připojení faktury k projektu

Popis: Případ užití popisuje scénář připojení faktury k projektu v aplikaci FIMS3. Předpokládá se, že je v systému vytvořená alespoň jedna faktura a alespoň jeden projekt, ke kterému je možné připojit fakturu.

Aktéři: Uživatel s rolí administrátor

Kroky:

1. Uživatel klikne na záložku „Invoices“ v hlavní nabídce v sekci Invoices
2. Systém zobrazí přehled vytvořených faktur
3. Uživatel klikne na jednu z faktur v seznamu (sloupec s ID, nebo ikona s lupou na konci řádku)
4. Systém přesměruje uživatele na detail vybrané faktury
5. Uživatel klikne na tlačítko „Assign projects“
6. Systém přesměruje uživatele na formulář pro připojení projektu
7. Uživatel klikne na pole pro vyhledání projektu
8. Systém vyhledá projekty, které lze připojit k faktuře a zobrazí je v list boxu pod vyhledávacím polem
9. Uživatel vyhledá projekt a klikne na něj
10. Ve vedlejším poli zadá hodnotu faktury bez DPH
11. Uživatel klikne na tlačítko se symbolem „+” a uzavře formulář přes tlačítko „Close”
12. Systém přesměruje uživatele na detail faktury. V sekci „Assigned projects“ je zobrazený připojený projekt se sumou bez DPH.

Výstup: Faktura byla připojena k projektu

4.5 Testovací scénáře

V této kapitole bude uveden kompletní seznam testovacích scénářů. Několik vybraných scénářů bude následně detailně rozvedeno. Zvolená struktura scénářů vychází z (15) a je doplněná o typ testu a prioritu. Scénáře, které ve svém řešení zpracovávám vychází z případů užití rozepsaných v předchozí kapitole. U všech scénářů se předpokládá, že je uživatel do aplikace přihlášený coby administrátor nebo manager. Bez přihlášení nemá uživatel do aplikace přístup. U některých scénářů se očekává, že v aplikaci existuje alespoň jeden projekt nebo faktura uvedeného typu. Takový požadavek je u scénáře uvedený

v kolonce „předpoklady“. ID scénářů byly pro zachování přehlednosti této práce přečíslovány.

1. **Vytvoření projektu –** scénáře jsou zaměřené na vytvoření projektu. Očekává se, že v aplikaci existuje alespoň jeden libovolný projekt.
   1.1. Vytvoření budget main projektu
   1.2. Vytvoření budget analytic projektu
   1.3. Vytvoření tariff main projektu
   1.4. Vytvoření tariff analytic projektu
   1.5. Vytvoření internal budget main projektu
   1.6. Vytvoření internal budget analytic projektu
   1.7. Vytvoření internal long-term main projektu
   1.8. Vytvoření internal long-term analytic projektu
2. **Úprava projektu**
   2.1. Úprava budget projektu
   2.2. Úprava tariff projektu
   2.3. Úprava internal budget projektu
   2.4. Úprava internal long-term projektu
3. **Připojení faktury k projektu**
   3.1. Připojení income faktury k budget main projektu
   3.2. Připojení income faktury k budget analytic projektu
   3.3. Připojení outcome faktury k budget main projektu
   3.4. Připojení outcome faktury k budget analytic projektu
   3.5. Připojení income faktury k tariff main projektu
   3.6. Připojení income faktury k tariff analytic projektu
   3.7. Připojení outcome faktury k tariff main projektu
   3.8. Připojení outcome faktury k tariff analytic projektu
   3.9. Připojení income faktury k internal budget main projektu
   3.10. Připojení income faktury k internal budget analytic projektu
   3.11. Připojení outcome faktury k internal budget main projektu
   3.12. Připojení outcome faktury k internal budget analytic projektu
   3.13. Připojení income faktury k internal long-term main projektu
   3.14. Připojení income faktury k internal long-term analytic projektu

3.15. Připojení outcome faktury k internal long-term main projektu

3.16. Připojení outcome faktury k internal long-term analytic projektu

4. **Úprava faktury připojené k projektu**
    4.1. Úprava faktury připojené k budget projektu
    4.2. Úprava faktury připojené k tariff projektu
    4.3. Úprava faktury připojené k internal budget projektu
    4.4. Úprava faktury připojené k internal long-term projektu
5. **Smazání faktury z projektu**
    5.1. Odpojení income faktury z budget projektu
    5.2. Odpojení outcome faktury z budget projektu
    5.3. Odpojení income faktury z tariff projektu
    5.4. Odpojení outcome faktury z tariff projektu
    5.5. Odpojení income faktury z internal budget projektu
    5.6. Odpojení outcome faktury z internal budget projektu
    5.7. Odpojení income faktury z internal long-term projektu
    5.8. Odpojení outcome faktury z internal long-term projektu

4.5.1 Testovací scénář: Vytvoření budget main projektu

ID testovacího případu: 1.1

Typ testu: Regresní, automatizovaný

Priorita: Critical

Účel: Ověření korektního vytvoření budget main projektu

Předpoklady: Uživatel je přihlášený jako administrátor, v aplikaci existuje alespoň jeden libovolný analytic projekt, do aplikace je synchronizovaný alespoň jeden projekt z Jiry, uživatel má k dispozici přehled vzorců pro výpočet hodnot v projektu

Testovací data: Libovolný analytic projekt, libovolný projekt synchronizovaný z Jiry

Prohlížeč: Electron

Tabulka 4 Testovací scénář: Vytvoření budget main projektu

Kroky	Očekávaný výsledek
V bočním menu klikni na záložku „Budget projects“	Uživatel je přesměrován na stránku „Budget projects list“
Klikni na tlačítko „New budget project“	Uživatel je přesměrován na formulář „New budget project“
Do pole Type zadej „Main project“, do pole Linked analytic project zadej připravený projekt, do pole Jira Project Codes vyplň projekt synchronizovaný z Jiry (viz. testovací data). Ostatní pole vyplň libovolnými korektními hodnotami	Pole jsou vyplněná
Klikni na tlačítko „Create“	Uživatel je přesměrován na detail nově vytvořeného projektu. V pravém horním rohu se zobrazí zelené pole se zprávou „Budget project was successfully created.“
Zkontroluj, jestli zobrazené informace odpovídají tebou zadaným. Dle přehledu vzorců zkontroluj, zda byly ostatní hodnoty správně vypočítány.	Všechny zobrazené informace jsou korektní

Zdroj: Autor

4.5.2 Testovací scénář: Úprava budget projektu

ID testovacího případu: 2.1

Typ testu: Regresní, automatizovaný

Priorita: Critical

Účel: Ověření možnosti upravit projekt

Předpoklady: Uživatel je přihlášený jako administrátor, do aplikace je synchronizovaný alespoň jeden projekt z Jiry, uživatel má k dispozici přehled vzorců pro výpočet hodnot v projektu

Testovací data: Libovolný analytic projekt synchronizovaný z Jiry

Prohlížeč: Electron

Tabulka 5 Testovací scénář: Úprava budget projektu

Kroky	Očekávaný výsledek
Prostřednictvím API requestu vytvoř nový main budget projekt	Projekt byl úspěšně vytvořen
V bočním menu klikni na záložku „Budget projects“	Uživatel je přesměrován na stránku „Budget projects list“
Vyhledej vytvořený projekt a otevři ho	Uživatel je přesměrován na detail vytvořeného projektu
Zkontroluj, jestli všechny hodnoty odpovídají hodnotám odeslaným v requestu	Všechny hodnoty odpovídají
Klikni na tlačítko „Edit“	Uživatel je přesměrován na formulář „Edit budget project“
Změň všechny informace včetně pole „Type“	Všechny informace ve formuláři jsou změněny
Klikni na tlačítko „Update“	Uživatel je přesměrován na detail nově vytvořeného projektu. V pravém horním rohu se zobrazí zelené pole se zprávou „Budget project was successfully created.“
Zkontroluj, jestli zobrazené informace odpovídají tebou zadaným. Dle přehledu vzorců zkontroluj, zda byly ostatní hodnoty správně vypočítány.	Všechny zobrazené informace jsou korektní

Zdroj: Autor

4.5.3 Testovací scénář: Připojení income faktury k budget main projektu

ID testovacího případu: 3.1

Typ testu: Regresní, automatizovaný

Priorita: Critical

Účel: Ověření možnosti připojit income fakturu k budget main projektu

Předpoklady: Uživatel je přihlášený jako administrátor, v aplikaci existuje alespoň jeden libovolný main a jeden analytic projekt, do aplikace je synchronizovaný alespoň jeden

projekt z Jiry, v aplikaci existuje alespoň jedna income faktura, uživatel má k dispozici přehled vzorců pro výpočet hodnot v projektu

Testovací data: Budget projekt, libovolný projekt synchronizovaný z Jiry

Prohlížeč: Electron

Tabulka 6 Připojení income faktury k budget main projektu

Kroky	Očekávaný výsledek
Prostřednictvím API requestu vytvoř nový main budget projekt	Projekt byl úspěšně vytvořen
Prostřednictvím API requestu vytvoř novou income fakturu	Faktura byla úspěšně vytvořena
Vyhledej vytvořenou fakturu a otevři ji	Uživatel je přesměrován na detail vytvořené faktury
Kliknu na tlačítko „Assign projects“	Uživatel je přesměrován na formulář pro připojení projektu
V poli „Select“ vyhledej vytvořený projekt, vyber ho a klikni na zelené tlačítko s „+“	Vybraný projekt se zobrazí v seznamu „Project“
Klikni na tlačítko „Close“	Uživatel je přesměrován zpět na detail faktury
V sekci „Assigned projects“ vyhledej nově připojený projekt a otevři jej kliknutím na tlačítko s ikonou lupy	Uživatel je přesměrován na detail připojeného projektu
Zkontroluj, zda se v bočním menu v sekci „Linked income invoices“ nachází připojená faktura a zda uvedené údaje odpovídají	Připojená faktura se nachází v sekci „Linked income invoices“, všechna zobrazená data jsou korektní
Zkontroluj, jestli jsou zobrazené údaje o projektu korektní. Dle přehledu vzorců zkontroluj, zda byly ostatní hodnoty správně vypočítány.	Všechny zobrazené informace jsou korektní

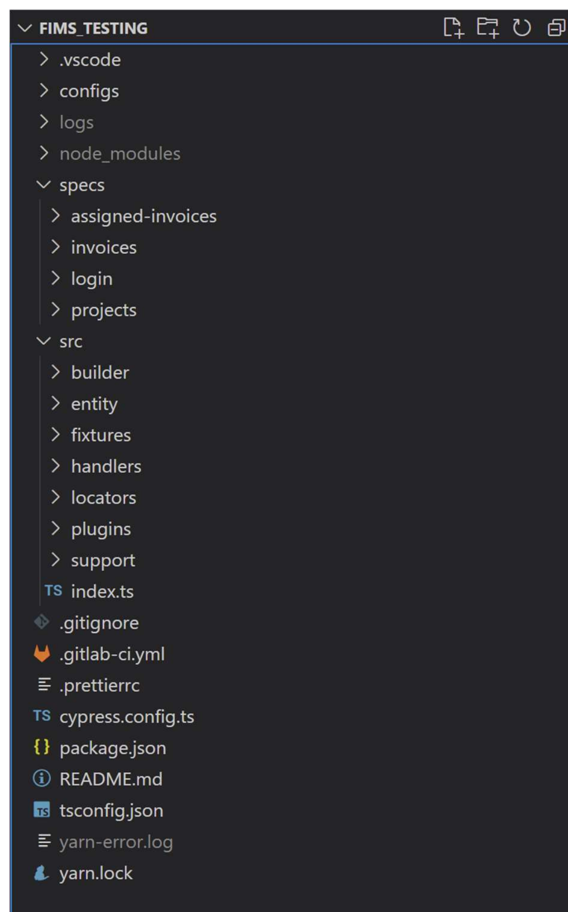
Zdroj: Autor

4.6 Vývoj automatizovaných testů

Následující podkapitola je zaměřená na realizaci řešení automatizovaného testování. Je zde ve stručnosti popsán způsob, jakým byla implementace automatizovaného testování provedena – především jak byl organizován a vytvořen kód. Popis je doplněn o ilustrativní ukázky kódu.

4.6.1 Organizace repozitáře

Obrázek 5 ScreenShot struktury repozitáře otevřeného v IDE Visual Studio



Zdroj: Autor

Základní struktura repozitáře je daná zvolenou technologií, kterou je Cypress verze 8.19.2. Soubory s testy jsou ve složkách umístěných v adresáři „specs" – pro každou kategorii testů je zde samostatná složka s příslušným souborem. Každý soubor pokrývá

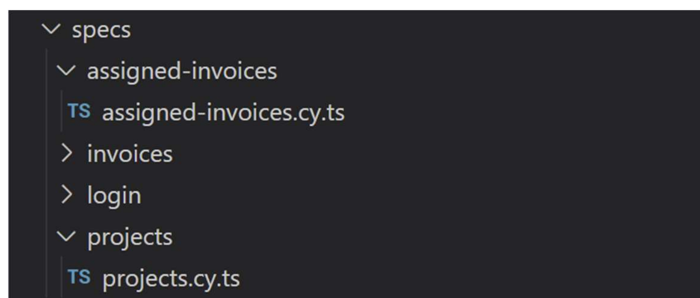
jednu testovací suitu[2]. Repozitář obsahuje mimo jiné složku „invoices" a „login". Tyto testovací suity ale nejsou součástí této práce.

Ve složce „src" se nachází soubory s jednotlivými třídami a metodami. Blíže budou vysvětleny v následující části. Kromě těchto dvou hlavních složek jsou v repozitáři další složky a soubory vztahující se především ke konfiguraci repozitáře, integracím a připojeným knihovnám.

4.6.2 Testy

Testovací suita pro projekty se nachází v souboru projects.cy.ts, suita pro testování vztahu mezi projektem a fakturou je v souboru assigned-invoices.cy.ts.

Obrázek 6 ScreenShot souborů s testovacími suitami v IDE Visual Studio Code



Zdroj: Autor

Všechny testy začínají hookem[3] BeforeEach, který obsahuje přihlášení do aplikace a pomocí API requestu GET získá data o Workerovi, která uloží do globální proměnné.

[2] Testovací suita je kolekce testů, které jsou navrženy tak, aby pokryly různé funkcionality, vlastnosti a scénáře použití testovaného softwarového produktu (29).
[3] Hooky v kontextu Cypressu označují funkcionalitu, která umožňuje spouštět blok kódu před nebo po každém testu ve vybrané testovací suitě (31).

Zdrojový kód 1 Hook BeforeEach

```
beforeEach('Visit FIMS3 and login', () => {
  cy.visit(data.base_url)
  CommonHandler.loginWithToken(
    data.user_roles.admin.username,
    data.user_roles.admin.password
  ).as('token').then((token) => {
    ApiRequestHandler.GetBusinessWorkerId(String(token)).then((vendor) => {
      ApiRequestHandler.GetWorkerAttributes(String(token),
String(vendor)).then(vendor => {
        vendorName = vendor.name
      })
    })
    ApiRequestHandler.GetBusinessWorkerId(String(token), 2).then((vendor2)
=> {
      ApiRequestHandler.GetWorkerAttributes(String(token),
String(vendor2)).then(vendor2 => {
        vendorName2 = vendor2.name
      })
    })
  })
})
```

Zdroj: Autor

Jednotlivé testy se skládají z mnoha funkcí ověřujících, zda je v aplikaci očekávaná hodnota. Tyto funkce jsou vždy v souladu s Best practices pro Cypress uzavřené v assertation[4] it a dohromady tvoří kompletní test odpovídající jednomu testovacímu scénáři.

Jako příklad je zde uveden test pro vytvoření budget main projektu. Test dle Cypress standardů začíná id a názvem, aby byla zachována přehlednost výsledných logů. Samotné metody s funkcemi pro ověření se v tomto případě volají ze souborů CommonHandler a ProjectHandler.

[4] Pojem Assertation v kontextu Cypressu označuje soubor validačních kroků k ověření, jestli testovací scénář uspěl nebo neuspěl (33).

Zdrojový kód 2 Test na vytvoření Budget main projektu

```
it('C1.1 - Create Budget project - main', () => {
  CommonHandler.menu('Budget projects')
  ProjectHandler.NewProject(Category.Budget)
  ProjectHandler.ProjectForm(vendorName, budget_project, Action.Create)
  cy.get('@token').then(token => {
    ProjectHandler.DeleteProjectFromDetail(Category.Budget, Type.TypeMain,
String(token))
  })
})
```

Zdroj: Autor

Obrázek 7 Výsledek spuštění uvedeného testu v Cypress Dashboardu

Zdroj: Autor

4.6.3 Handlers

Složky handlers obsahují soubory s největší částí celkového kódu. Jsou v nich soustředěny všechny metody potřebné k práci s konkrétní oblastí aplikace.

Pro ilustraci je zde uvedena jednoduchá metoda sloužící k otevření detailu faktury připojené k projektu. Metoda byla jako ukázka zvolena právě pro svou jednoduchost na které lze názorně ukázat práci se Cypressem.

Každý řetězec začíná předponou cy., za kterou následuje příkaz, v tomto případě „get“. Cy.get vybere na základě selektoru uloženého v ProjectDetailLocators konkrétní element na html stránce, ověří, zda obsahuje požadovaný textový řetězec a poté na něj klikne. Postup ve druhém příkazu je takřka totožný. Metoda je uzavřená funkcí která načte url adresu a ověří, zda byl uživatel přesměrován na správnou stránku.

Zdrojový kód 3 Metoda pro otevření faktury

```
/**
 * Opens the selected invoice from the side menu of the project detail
 */
static OpenInvoice(type: TransactionType, id: string) {
    cy.get(ProjectDetailLocators.side_content.side_menu).contains('Linked '
+ type.toLowerCase() + ' invoices').click()
    cy.get(ProjectDetailLocators.side_content.invoice_id).should('contain',
id).click()
    cy.url().should('contain', 'invoices')
}
```

Zdroj: Autor

4.6.4 Fixtures

Soubory s příponou fixtures obsahují datové typy enum[5] s konstantami. Protože byly některé z konstant příliš dlouhé nebo využívané na více místech v kódu, byly pro zachování přehlednosti přesunuty do samostatného souboru.

Zdrojový kód 4 Enum s částí URL

```
export enum ProjectUrl {
    Budget = 'projects/budgetProject/',
    Tariff = 'projects/tariffProject/',
    InternalBudget = 'projects/internalBudgetProject/',
    InternalLongterm = 'projects/internalLongTermProject/',
}
```

Zdroj: Autor

Jako příklad je zde uveden enum s částí url adresy vztahující se k projektům. K tomuto řešení bylo přistoupeno z důvodu parametrizování metod s requesty na API.

4.6.5 Builder a entity

Při vyplňování formuláře pro vytvoření projektu bylo potřeba předávat mezi metodami větší objem hodnot. Při zdánlivě nejjednodušším způsobu předání parametru - skrze argumenty, docházelo k prodlužování kódu až k desítky řádků. Proto bylo přistoupeno k implementaci builderu a entit. Builder je v kontextu TypeScriptu návrhový vzor umožňující pomocí jednoduchých metod sestavovat datové objekty (43). Builder byl

[5] V TypeScriptu, enum (výčet) označuje datový typ, který slouží k definování sady konstant s explicitně přiřazenými hodnotami. Výčet umožňuje definovat konstanty, které mají významově související hodnoty a umožňuje snadněji pracovat s nimi v kódu (38).

implementován pomocí třídy (class), která má metody pro přidávání jednotlivých částí objektu a metodu pro vytvoření finálního objektu. Entita je zde použita k definování datové struktury, do které builder následně vkládá obsah. Při každém vytvoření nového objektu je možné nastavit vlastní hodnoty nebo použít výchozí. Díky tomu lze v souboru se samotnými testy vytvořit v jednom řádku konstantu obsahující komplexní datovou strukturu s daty, která se následně vyplní ve formuláři.

Zdrojový kód 5 Úprava budget projektu

```
it('C2.1 - Edit budget project', () => {
  let budget_project = ProjectBuilder.BudgetMainProject().setName('Budget
project for edit ' + CommonHandler.NewId()).build()
  cy.get('@token').then((token) => {
    ApiRequestHandler.CreateProject(String(token), budget_project)
    let edited_project = ProjectBuilder.BudgetAnalyticProject()
      .setName('Edited budget project ' + CommonHandler.NewId())
      .setDescription('Updated description')
      .setJiraProjectCodes(JiraProjectCodes.ProjectCode2)
      .setJiraEpics(JiraEpics.JiraEpics2)
      .setBudget('10000')
      .setEstOtherCosts('6000')
      .setInitHourEst('100')
      .build()
    CommonHandler.menu('Budget projects')
    ProjectHandler.OpenProjectFromList(budget_project.name)
    cy.get(ProjectDetailLocators.buttons.edit).click()
    ProjectHandler.ProjectForm(String(vendorName2), edited_project,
Action.Edit)
    ProjectHandler.DeleteProjectFromDetail(edited_project.category,
edited_project.type, String(token))
  })
})
```

Zdroj: Autor

4.7 Měření manuálního testování

Manuální testování bylo z důvodu snahy o objektivitu provedeno a měřeno třikrát. Z výsledků byla spočítána průměrná délka trvání průchodu testovacím scénářem. Pro ilustraci jsou zde uvedena měření testování vytvoření projektu a tabulka souhrnných výsledků. Kompletní tabulka výsledků ze všech měření se nachází v příloze.

Z přiložené tabulky 7 lze vidět, že mezi jednotlivými měřeními v rámci stejného druhu projektu nejsou vidět výrazné rozdíly. Velmi se ale liší délka měření mezi druhy

projektů – průchod testovacím scénářem na vytvoření budget projektu trval necelých 6 minut. Oproti tomu vytvoření internal long-term projektu včetně kontroly trvalo pouze necelou minutu. Tento rozdíl je zapříčiněn množstvím parametrů, které se dopočítávají na základě zadaných hodnot a pro testera je jejich manuální kontrola časově náročnější.

Tabulka 7 Ukázka části výsledků měření manuálního testování

	Měření v minutách			Průměr
	1. měření	2. měření	3. měření	
1.1. Vytvoření budget main projektu	5,93	5,81	5,50	5,75
1.2. Vytvoření budget analytic projektu	5,85	5,67	6,25	5,92
1.3. Vytvoření tariff main projektu	3,20	3,33	3,10	3,21
1.4. Vytvoření tariff analytic projektu	3,15	3,37	3,38	3,30
1.5. Vytvoření internal budget main projektu	5,82	5,88	5,65	5,78
1.6. Vytvoření internal budget analytic projektu	5,73	5,98	5,80	5,84
1.7. Vytvoření internal long-term main projektu	0,87	0,98	0,90	0,92
1.8. Vytvoření internal long-term analytic projektu	0,90	0,85	0,97	0,91

Zdroj: Autor

Průchod všemi scénáři zabral průměrně zhruba 121 minut. Kompletní tabulku měření lze nalézt v příloze. Mezi výsledkem 1. a 2. měření takřka není rozdíl, zajímavé ale je, že 3. měření bylo o 4 minuty kratší než předchozí 2 měření. Lze předpokládat, že ke zrychlení procesu došlo díky zkušenostem získaným během předchozích měření.

Tabulka 8 Celkové výsledky manuálního testování

	Měření v minutách			Průměr
	1. měření	2. měření	3. měření	
Průchod všemi scénáři	122,82	122,83	118,23	121,29
Doplňkové činnosti	23,67	20,18	19,08	20,98
	Průměrná délka testování v minutách			142,27
	Průměrná délka testování v hodinách			2,37

Zdroj: Autor

Kromě průchodů scénářem byl měřen čas strávený činnostmi spojenými s testováním, které jsou nezbytné, ale nedají se do průchodu scénářem započíst. Jedná se například o přihlašování do aplikace, hledání scénářů, manipulace s daty, a jiné. Tyto doplňkové činnosti zabraly v průměru 20,98 minut.

Celková průměrná časová náročnost jednoho kompletního manuálního průchodu testovacími scénáři včetně všech doplňkových činností je 142,27 minut, tedy 2,37 hodiny. Během průchodu všemi 40 scénáři došlo průměrně k 6,33 chyb, zpravidla se jednalo o chyby ve výpočtu některé z hodnot. Chyby neovlivnily celkovou úspěšnost testování, pouze mírně zpomalily průchod scénářem, protože bylo potřeba se vrátit a spočítat hodnotu znovu.

4.8 Měření automatizovaného testování

Stejně jako v případě manuálního testování bylo i automatizované testování provedeno třikrát. Ačkoliv automatizované testování není náchylné na lidskou chybu, stále lze předpokládat drobné výkyvy ve výkonu Cypress cloudu případně serveru, který hostuje FIMS3. Není vyloučen ani výskyt flaků [6](viz. podkapitola Cypress). Testy byly pro účely měření spuštěny v konzoli WSL[7] Ubuntu v operačním systému Windows 11 Pro. Kromě Cypress Dashboardu (obrázek 11 a 12 v příloze) byl průběh testů v reálném čase sledován i v konzoli (obrázek 8).

[6] Pojem „flaky test" označuje situaci, kdy test náhodně selže, ale při opětovném spuštění projde, aniž by mezitím došlo ke změně kódu (40).

[7] WSL (Windows Subsystem for Linux) je funkce v operačním systému Microsoft Windows, která umožňuje uživatelům spouštět příkazy a aplikace z Linuxu přímo v rámci Windows bez nutnosti instalace samostatného operačního systému (30).

Obrázek 8 Výsledek prvního běhu testovací suity

	Spec		Tests	Passing	Failing	Pending	Skipped
✓	assigned-invoices/assigned-invoices .cy.ts	06:46	28	28	-	-	-
✓	projects/projects.cy.ts	04:02	12	12	-	-	-
✓	All specs passed!	10:48	40	40	-	-	-

Zdroj: Autor

Tabulka 9 Celkové výsledky automatizovaného testování

	Měření v minutách			Průměr
	1. měření	2. měření	3. měření	
Průchod všemi scénáři	10,80	11,13	12,73	11,55
Doplňkové činnosti	1,23	1,22	1,28	1,24
	Průměrná délka testování v minutách			12,80
	Průměrná délka testování v hodinách			0,21

Zdroj: Autor

Při automatizovaném testování zabral průchod testovacími scénáři 11,55 minuty, což odpovídá přibližně 0,5 minutě na 1 testovací scénář. Doplňkové činnosti, ke kterým v případě automatizovaného testování patří například připojení k Cypress Cloudu zabraly v průměru 1,24 minuty.

Celková průměrná délka testování byla při automatizovaném testování 12,8 minuty, tedy 0,21 hodiny. Během testování nebyly nalezeny žádné chyby. Všechny testy proběhly korektně, bez flaků a jiných komplikací.

4.9 Porovnání měření manuálního a manuálního testování

Jak bylo zmíněno v předchozích kapitolách, vhodnost implementace automatizovaného testování bude posouzena z hlediska časové úspory, finanční návratnosti, personální náročnosti a chybovosti. Porovnání vychází z metrik získaných měřením časové náročnosti implementace automatizace, manuálního testování a automatizovaného testování.

4.9.1 Časová úspora

Jeden průchod všemi testovacími scénáři včetně započítání doplňkových činností jako je příprava dat, hledání scénářů a jiné trval v průměru 2,37 hodiny. Kontrola běhu

automatizovaných testů trvala v průměru 0,25 hodiny. Denně tedy došlo k úspoře 2,12 hodiny. Do porovnání měsíční úspory času byla zahrnuta i údržba testů, která činila po zaokrouhlení v průměru 11 hodin měsíčně. Měsíčně tedy došlo k úspoře 25,69 hodiny, což se rovná zhruba 3 pracovním dnům jednoho zaměstnance.

Tabulka 10 Porovnání celkové časové náročnosti testování

	Manuální	Automatizované	Ušetřeno
Testování denně v hodinách	2,37	0,25	2,12
Testování týdně v hodinách	9,48	1,25	8,23
Testování měsíčně v hodinách	37,94	12,25	25,69
Testování ročně v hodinách	455,26	147,00	308,26

Zdroj: Autor

4.9.2 Finanční návratnost

Pro vyhodnocení finanční návratnosti bylo nutno nalézt věrohodná statistická data s průměrnou hodinovou mzdou manuálního a automation testera. Obecné statistiky zaměřené na průměrnou mzdu bohužel většinou neobsahují průměrné mzdy takto specifických pracovních pozic, ale spíše průměr napříč oborem. Pro srovnání proto byla použita statistická data z portálu Jooble (44), vycházející z inzerátů uveřejňovaných na této stránce.

K 12.3.2023 byla uvedena průměrná hodinová mzda junior testera 353,68 Kč. Pozice junior testera byla pro srovnání zvolena z důvodu, že kvalifikací nejvíce odpovídala požadavkům na manuálního testera. Průměrná mzda automation testera byla ke stejnému dni 423,88 Kč. Uvedená data byla zaokrouhlena na celé číslo a pro ověření porovnána s daty z portálu Platy.cz (45). Zde byly průměrné mzdy přibližně o 20 korun nižší, ale poměrem se takřka shodovaly.

Tabulka 11 Celkové náklady na implementovanou automatizaci

	Čas v hodinách	Náklady v Kč
Vytvoření projektu	130	55104
Úprava projektu	29	12293
Připojení faktury k projektu	97	41116
Úprava faktury připojené k projektu	30	12716
Smazání faktury z projektu	3	1272
Refactoring, nutné úpravy testů v průběhu 6 měsíců	66	27976
Refactoring a úpravy v přepočtu na měsíc	11	4663
Náklady celkem v Kč	122501	

Zdroj: Autor

Tabulka uvádí čas vynaložený na automatizování jednotlivých testovacích suit a náklady vypočítané na základě průměrné mzdy na odpovídající pozici. Do času stráveného automatizací suit byly kromě samotné implementace zahrnuty i doplňkové činnosti jako konzultace se členy týmu a code review. Celkové náklady na implementování automatizace byly 122501 Kč (zaokrouhleno na celé číslo). Do této hodnoty není započítána údržba.

Je zde potřeba vzít v úvahu, že průměrná mzda bude odpovídat z hlediska kvalifikace testerovi s průměrným množstvím zkušeností – tedy ne juniorovi. Uvedenou implementaci ale prováděl junior po získání základních znalostí použitých technologií. Lze tedy předpokládat, že návratnost investice bude ve skutečnosti trochu dřívější než vyplývá z uvedených dat.

Tabulka 12 Zhodnocení návratnosti investice do automatizovaného testování

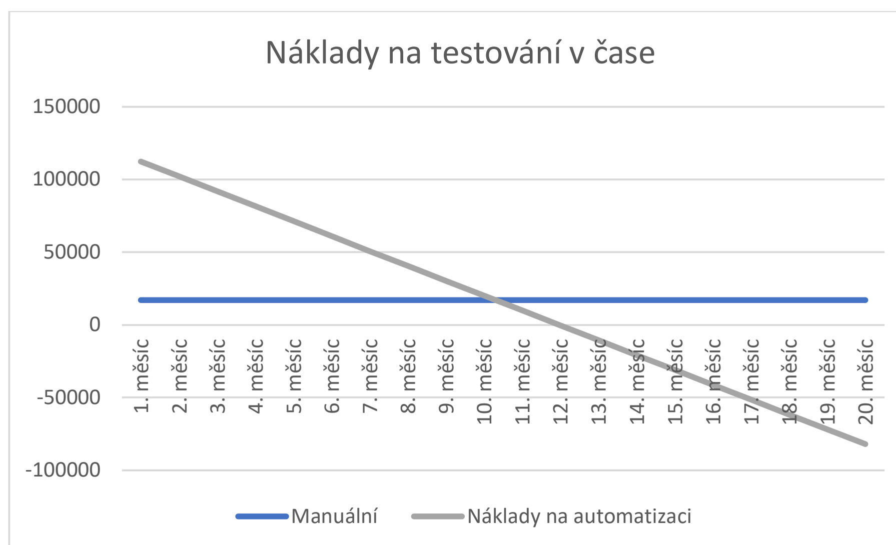
	Manuální	Automatizované	Ušetřeno	Ušetřeno se započítáním nákladů
Testování denně	850	105	745	-121757
Testování týdně	4252	530	3723	-118779
Testování měsíčně	17010	6782	10228	-112274
Testování ročně	204118	81385	122733	231

Zdroj: Autor

Na základě průměrných mezd a naměřeného času stráveného manuálním testováním a kontrolou automatizovaných testů byly vypočítány denní, týdenní, měsíční a roční náklady. Do měsíčních nákladů byly započítány průměrné měsíční náklady na refactoring a údržbu testů z tabulky č. 10. Sloupec „Ušetřeno" uvádí, kolik korun je v uvedeném časovém úseku ušetřeno, pokud místo manuálního testování budeme testovat automatizovaně. Sloupec

„Ušetřeno se započítáním nákladů“ uvádí výslednou situaci, pokud od ušetřené částky odečteme celkové náklady na automatizaci uvedené v tabulce č. 10. Z těchto dat vyplývá, že investice do automatizace se při započítání všech nákladů včetně měsíční údržby vrátí po 1 roce od implementace.

Graf 2 Náklady na testování v čase



Zdroj: Autor

4.9.3 Personální náročnost

Lze s jistotou předpokládat, že implementací automatizace dojde ke zvýšení nároků na znalosti a schopnosti na testera, neboť pro provedení automatizace je nezbytné znát vybranou technologii a programovací jazyk. Je rovněž vhodné mít alespoň základní znalost nástroje Git, který slouží k verzování kódu.

Důležité je vzít v úvahu i kvalitu kódu, která má přímý vliv na stabilitu a rychlost testů. Pro její zvýšení je potřeba mít k dispozici alespoň jednu osobu s minimálně stejnými, ideálně však většími znalostmi vybrané technologie. Ačkoliv kontrolu průběhu testů může provádět takřka kdokoliv, pro porozumění logům s výsledky je především v případě selhání testů stále potřeba člověk se znalostí použité technologie.

Ačkoliv došlo k nespornému zvýšení personální náročnosti vybraného projektu, toto zvýšení bylo dle požadavků na automatizaci částečně zmírněno výběrem uživatelsky přívětivé technologie s dostatečně obsáhlou dokumentací.

4.9.4 Chybovost

Tabulka 13 Porovnání chybovosti manuálního a automatizovaného testování

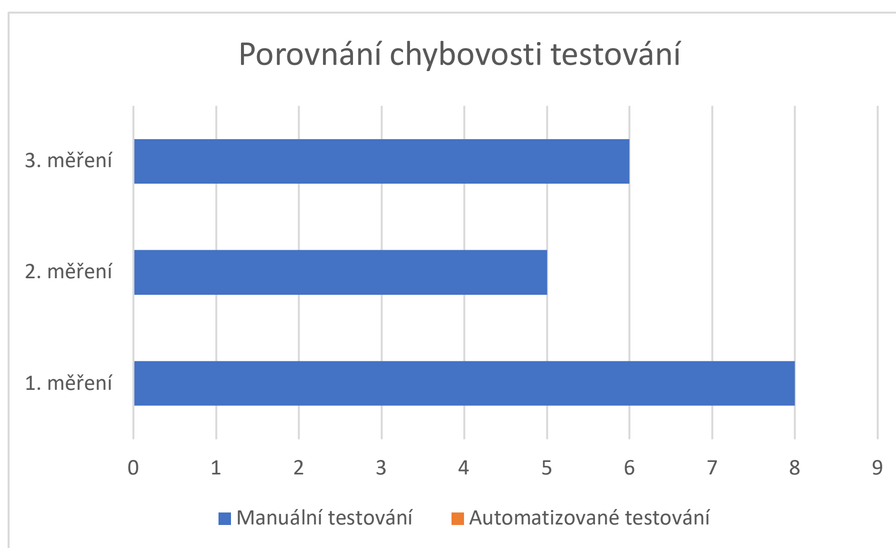
	Měření			Průměrně
	1. měření	2. měření	3. měření	
Manuální testování	8	5	6	6,33
Automatizované testování	0	0	0	0

Zdroj: Autor

Během manuálního testování došlo během průchodu 40 testovacími scénáři v průměru k 6,33. Ve všech případech se jednalo o chybu ve výpočtu některé z hodnot a chyba výrazně neovlivnila celkový výsledek testování, pouze mírně prodloužila čas průchodu daným testem.

Automatizované testování proběhlo během všech tří testovacích běhů bez chyb. Potvrdil se tedy předpoklad, že automatizované testování vykazuje nižší chybovost. Porovnání výsledků testování je znázorněno v grafu 3.

Graf 3 Porovnání chybovosti testování



Zdroj: Autor

5 Výsledky a diskuse

Výstupem této práce je 5 případů užití a 40 zautomatizovaných testovacích scénářů. Automatizací se denně ušetří 2,37 hodiny práce. Implementace automatizace včetně doplňkových činností jako jsou konzultace a code review trvala 289 hodin. Průměrná doba potřebná pro aktualizace a opravy testů byla spočítána na 11 hodin měsíčně. Tato hodnota vychází z údajů získaných ze 6 po sobě jdoucích měsíců. Návratnost finanční investice do implementace se očekává během jednoho roku. Při výpočtu návratnosti byla započítána i doba potřebná pro denní kontrolu testů a průběžné aktualizace a opravy testovacího repozitáře.

Doba návratnosti může být zkrácena použitím testovací sady při dalších typech testování – například při integračním nebo smoke testování. Nicméně, primárním účelem byla automatizace regresního testování, které je na konkrétním projektu prováděno denně, tudíž lze předpokládat, že by další využití testů zkrátilo dobu finanční návratnosti pouze mírně.

Z hlediska časové a na ní navazující finanční náročnosti považuji rovněž za důležité poukázat na možnost mírné nekonzistence získaných dat. Doba strávená implementováním automatizovaných testů odpovídá testerovi s juniorní úrovní zkušeností, ale lze předpokládat, že průměrný plat této úrovni zkušeností neodpovídá. Navzdory této skutečnosti však můžeme s jistotou zkonstatovat, že na uvedeném projektu došlo implementací automatizovaného testování k výraznému ušetření času, snížení chybovosti a v horizontu jednoho roku bude dosaženo finanční úspory, která se bude v průběhu času zvyšovat.

Vzhledem ke skutečnosti, že implementace proběhla bez potíží a žádné se neobjevily ani po nasazení testů, můžeme zkonstatovat, že nebyla prokázána nevhodnost vybraného testovacího nástroje. Lze tedy předpokládat, že testovací nástroj byl vybraný vhodně.

Za nevýhodu implementace automatizovaného testování může být považována skutečnost, že byly na vybraném projektu zvýšeny nároky na schopnosti testera. Nicméně, tato nevýhoda je částečně snížena vybranou technologií, která se prokázala jako vhodná i v případě juniorního zaměstnance se základní znalostí programování.

Získané poznatky mohou být využity na podobných projektech při rozhodovacím procesu, zda přistoupit k automatizaci či nikoliv. Snížení chybovosti je nesporné, nicméně jak ukazují získaná data, při hodnocení časové úspory a finanční návratnosti je důležité vzít

v úvahu především frekvenci testování. Pokud by pro automatizaci byly zvoleny testy spouštěné méně často, mohla by se automatizace stát výhodnou až ve velmi dlouhém časovém intervalu (v řádku let), čímž by mohla být považována za ekonomicky nevýhodnou.

6 Závěr

Předmětem této práce bylo navržení a implementace automatizace testování softwaru na vybraném projektu a následné zhodnocení jeho přínosnosti. Vybraným projektem byla aplikace zaměřená na správu firemních financí, projektů a zaměstnanců. Oblastí vybranou pro implementaci automatizace byla správa projektů a jejich vazeb s fakturami.

Nejdříve byla provedena rešerše odborných literárních a webových zdrojů. Na základě teoretických znalostí získaných z těchto zdrojů byly definovány požadavky na automatizaci, vybrán vhodný testovací nástroj, sepsány případy užití a testovací scénáře. Dle nich byla posléze provedena implementace automatizovaného testování. Implementace spočívala v automatizaci 5 případů užití a z nich vycházejících 40 testovacích scénářů. Pro tyto účely byl zvolen testovací nástroj Cypress a programovací jazyk TypeScript.

Vhodnost implementovaného řešení byla vyhodnocena pomocí porovnání metrik manuálního a automatizovaného testování z hlediska finanční návratnosti, časové úspory, personální náročnosti a chybovosti. Data pro hodnocení byla získána změřením časové náročnosti implementace automatizace, časové náročnosti a chybovosti manuálního testování a implementovaného automatizovaného testování aplikace. Na základě těchto dat byla vypočítána finanční náročnost.

Porovnáním získaných dat jsme došli k závěru, že na vybraném projektu:

- Došlo ke snížení časové náročnosti
- V průběhu jednoho roku dojde k návratnosti investice, poté díky automatizaci dojde ke snížená nákladů
- Došlo k mírnému zvýšení personální náročnosti
- Došlo ke snížení chybovosti

Získané poznatky lze aplikovat na projekty obdobné velikosti s podobným testovacím procesem, viz. kapitola Zhodnocení výsledků.

7 Seznam použitých zdrojů

1. PATTON, Ron. *Software testing.* 2nd ed. Indianapolis: Sams Publishing, 2006. ISBN 0672327988.
2. ROUDENSKÝ, Petr. *Kvalita softwaru: teorie a praxe*. Upravené a rozšířené 2. vydání. Prostějov: Computer Media, 2018. ISBN 978-80-7402-322-
3. CaSTB, Czech and Slovak Testing Board. *Učební osnovy – Certifikovaný tester základní úrovně* [online].(PDF). [cit. 14.03.2023]. Dostupné z: https://castb.org/wp-content/uploads/2020/05/ISTQB_CTFL_CZ_3_1_1-6.pdf
4. Co je agilní vývoj? - Azure DevOps | Microsoft Learn. [online]. Copyright © Microsoft 2023 [cit. 14.03.2023]. Dostupné z: https://learn.microsoft.com/cs-cz/devops/plan/what-is-agile-development
5. What is Agile? | Agile 101 | Agile Alliance. *Agile Alliance* [online]. Copyright ©2023 Agile Alliance [cit. 14.03.2023]. Dostupné z: https://www.agilealliance.org/agile101/
6. [online]. Copyright © 2001, výše zmínění autoři [cit. 14.03.2023]. Dostupné z: https://agilemanifesto.org/iso/cs/manifesto.html
7. *Moodle UK pro výuku 1* [online]. Copyright © [cit. 14.03.2023]. Dostupné z: https://dl1.cuni.cz/pluginfile.php/864918/mod_resource/content/1/Metodiky-v%C3%BDvoje-software-studijn%C3%AD-text.pdf
8. KITNER, Radek. Co dělá tester - Radek Kitner. *Radek Kitner - konzultant, lektor testování softwaru* [online]. [cit. 14.03.2023]. Dostupné z: https://kitner.cz/testovani_softwaru/co_dela_tester/
9. Why Cypress? | Cypress Documentation. *Cypress Documentation* [online]. Copyright © 2023 Cypress.io. All rights reserved. [cit. 14.03.2023]. Dostupné z: https://docs.cypress.io/guides/overview/why-cypress
10. Use Case Modeling - Kurt Bittner, Ian Spence - Knihy Google. *Knihy Google* [online]. Dostupné z: https://books.google.cz/books?id=zvxfXvEcQjUC&lpg=PR15&ots=zSmqRmcZ56&dq=use%20case&lr&hl=cs&pg=PR11#v=onepage&q=use%20case&f=false
11. Manual Test Cases vs. Automated Test Cases — who wins? Round 2. | by Alexey Himself | Practical Software Testing | Medium. *Medium – Where good ideas find you.* [online]. [cit. 14.03.2023]. Dostupné z: https://medium.com/practical-

software-testing/manual-test-cases-vs-automated-test-cases-who-wins-round-2-a6251e959fa5

12. Black box test - CleverAndSmart Management Consulting. *CleverAndSmart Management Consulting* [online]. Copyright © 2008 [cit. 14.03.2023]. Dostupné z: https://www.cleverandsmart.cz/black-box-test/
13. White box test - CleverAndSmart Management Consulting. *CleverAndSmart Management Consulting* [online]. Copyright © 2008 [cit. 14.03.2023]. Dostupné z: https://www.cleverandsmart.cz/white-box-test/
14. Grey box test - CleverAndSmart Management Consulting. *CleverAndSmart Management Consulting* [online]. Copyright © 2008 [cit. 14.03.2023]. Dostupné z: https://www.cleverandsmart.cz/grey-box-test/
15. How to write Test Cases (with Format & Example) | BrowserStack. *Most Reliable App & Cross Browser Testing Platform | BrowserStack* [online]. Copyright © 2023 BrowserStack. All rights reserved. [cit. 14.03.2023]. Dostupné z: https://www.browserstack.com/guide/how-to-write-test-cases
16. FOURNIER, Greg. *Essential Software Testing: A Use-Case Approach*. New York: Auerbach Publications, 2008. ISBN 9781420089813.
17. Test Planning: A Detailed Guide | BrowserStack. *Most Reliable App & Cross Browser Testing Platform | BrowserStack* [online]. Copyright © 2023 BrowserStack. All rights reserved. [cit. 14.03.2023]. Dostupné z: https://www.browserstack.com/guide/test-planning
18. 7 Phases of the System Development Life Cycle Guide. *Stay Secure as You Build | CloudDefense.AI* [online]. [cit. 14.03.2023]. Dostupné z: https://www.clouddefense.ai/blog/system-development-life-cycle
19. ISO 25000. *ISO/IEC 25010* [online]. [cit. 14.03.2023]. Dostupné z: https://iso25000.com/index.php/en/iso-25000-standards/iso-25010
20. ISO - ISO/IEC/IEEE 12207:2017 - Systems and software engineering — Software life cycle processes. [online]. Copyright © All Rights Reserved [cit. 14.03.2023]. Dostupné z: https://www.iso.org/standard/63712.html
21. Why Cypress? | Cypress Documentation. *Cypress Documentation* [online]. Copyright © 2023 Cypress.io. All rights reserved. [cit. 14.03.2023]. Dostupné z: https://docs.cypress.io/guides/overview/why-cypress

22. Cypress.io, *Cypress blog* [online]. Copyright © 2023 Cypress.io. All rights reserved. [cit. 14.03.2023]. Dostupné z: https://www.cypress.io/blog/
23. Best practices for test automation | 2023 tester's checklist. *Katalon Software Quality Management Platform* [online]. Copyright © 2023 Katalon, Inc. All rights reserved. [cit. 14.03.2023]. Dostupné z: https://katalon.com/resources-center/blog/test-automation-best-practices
24. CRISPIN, Lisa a Janet GREGORY. *Agile testing: a practical guide for testers and agile teams.* New Jersey: Addison-Wesley, c2009. The Addison-Wesley signature series. ISBN 9780321534460.
25. Best Practices | Cypress Documentation. *Cypress Documentation* [online]. Copyright © 2023 Cypress.io. All rights reserved. [cit. 14.03.2023]. Dostupné z: https://docs.cypress.io/guides/references/best-practices
26. P26513, Mar 2016 - ISO/IEC/IEEE International Draft Standard - Systems and Software Engineering -- Requirements for Testers and Reviewers of User Documentation | IEEE Standard | IEEE Xplore. *301 Moved Permanently* [online]. Copyright © Copyright 2023 IEEE [cit. 14.03.2023]. Dostupné z: https://ieeexplore.ieee.org/document/7469978
27. Guru99 | *Quality Assurance vs Quality Control – Difference Between Them*. [online]. [cit. 14.03.2023]. Dostupné z: https://www.guru99.com/quality-assurance-vs-quality-control.html
28. Builder in TypeScript / Design Patterns. *Refactoring and Design Patterns* [online]. [cit. 14.03.2023]. Dostupné z: https://refactoring.guru/design-patterns/builder/typescript/example
29. Test Suites and Their Test Cases: The Hierarchy Explained - Testim. *Automated UI and Functional Testing - AI-Powered Stability - Testim.io* [online]. Copyright © testim 2023 [cit. 14.03.2023]. Dostupné z: https://www.testim.io/blog/test-suite/
30. What is Windows Subsystem for Linux | Microsoft Learn. [online]. Copyright © Microsoft 2023 [cit. 14.03.2023]. Dostupné z: https://learn.microsoft.com/en-us/windows/wsl/about
31. What are before and beforeEach hooks in Cypress?. *Educative: Interactive Courses for Software Developers* [online]. Copyright ©2023 Educative, Inc. All rights reserved [cit. 14.03.2023]. Dostupné z: https://www.educative.io/answers/what-are-before-and-beforeeach-hooks-in-cypress

32. Best Practices | Cypress Documentation. *Cypress Documentation* [online]. Copyright © 2023 Cypress.io. All rights reserved. [cit. 14.03.2023]. Dostupné z: https://docs.cypress.io/guides/references/best-practices
33. What are Cypress Assertions and How to use Assertions in Cypress?. *Tools QA* [online]. Copyright © 2013 [cit. 14.03.2023]. Dostupné z: https://www.toolsqa.com/cypress/cypress-assertions/
34. The Selenium Browser Automation Project | Selenium. *Selenium* [online]. Copyright © 2023 Software Freedom Conservancy All Rights Reserved [cit. 14.03.2023]. Dostupné z: https://www.selenium.dev/documentation/
35. UNADKAT, Jash. *Getting Started with Selenium IDE* [online] 2023 [cit. 14.03.2023]. Dostupné z: https://www.browserstack.com/guide/what-is-selenium-ide
36. Welcome to Jira Software | Atlassian . *Collaboration software for software, IT and business teams* [online]. Copyright © 2023 Atlassian [cit. 14.03.2023]. Dostupné z: https://www.atlassian.com/software/jira/guides/getting-started/introduction#what-is-jira-software
37. Tester mzda - Zjistěte průměrnou tester výši mzdy na Jooble. *Práce v České republice - 152.000+ aktuálních nabídek práce - Jooble* [online]. Copyright © [cit. 14.03.2023]. Dostupné z: https://cz.jooble.org/salary/tester#hourly
38. TypeScript: Handbook - Enums. *TypeScript: JavaScript With Syntax For Types.* [online]. Copyright © 2012 [cit. 14.03.2023]. Dostupné z: https://www.typescriptlang.org/docs/handbook/enums.html
39. *7 Principles Of Software Testing: Defect Clustering And Pareto Principle* [online]. 2023 [cit. 14.03.2023]. Dostupné z: https://www.softwaretestinghelp.com/7-principles-of-software-testing/#6_Pesticide_Paradox
40. Flaky Test Management | Cypress Documentation. *Cypress Documentation* [online]. Copyright © 2023 Cypress.io. All rights reserved. [cit. 14.03.2023]. Dostupné z: https://docs.cypress.io/guides/cloud/flaky-test-management
41. WebDriver | Selenium. *Selenium* [online]. Copyright © 2023 Software Freedom Conservancy All Rights Reserved [cit. 14.03.2023]. Dostupné z: https://www.selenium.dev/documentation/webdriver/
42. Puppeteer, Selenium, Playwright, Cypress - how to choose? - AI-driven E2E automation with code-like flexibility for your most resilient tests. *Automated UI*

and Functional Testing - AI-Powered Stability - Testim.io [online]. Copyright © testim 2023 [cit. 14.03.2023]. Dostupné z: https://www.testim.io/blog/puppeteer-selenium-playwright-cypress-how-to-choose/

43. Mastering the Builder Pattern in TypeScript - Upmostly. *Upmostly - Learn React and JavaScript the Right Way* [online]. Copyright © [cit. 14.03.2023]. Dostupné z: https://upmostly.com/typescript/mastering-the-builder-pattern-in-typescript
44. Práce v České republice - 152.000+ aktuálních nabídek práce - Jooble. *Práce v České republice - 152.000+ aktuálních nabídek práce - Jooble* [online]. Copyright © [cit. 14.03.2023]. Dostupné z: https://cz.jooble.org/
45. Přehled platů | průměrná mzda | Česká republika - Platy.cz. *Přehled platů | průměrná mzda | Česká republika - Platy.cz* [online]. Copyright © 1997 [cit. 14.03.2023]. Dostupné z: https://www.platy.cz/

8 Seznam obrázků, tabulek, grafů a zkratek

8.1 Seznam obrázků

Obrázek 1 Schématické zobrazení Scrumu ... 15
Obrázek 2 Graf zobrazující stoupající náklady na opravu chyby v průběhu času ... 21
Obrázek 3 Graf zobrazující příčiny vzniku softwarových chyb ... 22
Obrázek 4 Schématické zobrazení Black box testování ... 27
Obrázek 5 ScreenShot struktury repozitáře otevřeného v IDE Visual Studio Code ... 45
Obrázek 6 ScreenShot souborů s testovacími suitami v IDE Visual Studio Code ... 46
Obrázek 7 Výsledek spuštění uvedeného testu v Cypress Dashboardu ... 48
Obrázek 8 Výsledek prvního běhu testovací suity ... 53

8.2 Seznam tabulek

Tabulka 1 Porovnání black box a white box testování ... 28
Tabulka 2 Výhody a nevýhody Selenia ... 32
Tabulka 3 Výhody a nevýhody Cypressu ... 33
Tabulka 4 Testovací scénář: Vytvoření budget main projektu ... 42
Tabulka 5 Testovací scénář: Úprava budget projektu ... 43
Tabulka 6 Připojení income faktury k budget main projektu ... 44
Tabulka 7 Ukázka části výsledků měření manuálního testování ... 51
Tabulka 8 Celkové výsledky manuálního testování ... 52
Tabulka 9 Celkové výsledky automatizovaného testování ... 53
Tabulka 10 Porovnání celkové časové náročnosti testování ... 54
Tabulka 11 Celkové náklady na implementovanou automatizaci ... 55
Tabulka 12 Zhodnocení návratnosti investice do automatizovaného testování ... 55
Tabulka 13 Porovnání chybovosti manuálního a automatizovaného testování ... 57

8.3 Seznam grafů

Graf 1 Porovnání profitability manuálního a automatizovaného testování v čase ... 30
Graf 2 Náklady na testování v čase ... 56
Graf 3 Porovnání chybovosti testování ... 57

8.4 Seznam zdrojových kódů

Zdrojový kód 1 Hook BeforeEach ... 47
Zdrojový kód 2 Test na vytvoření Budget main projektu ... 48
Zdrojový kód 3 Metoda pro otevření faktury ... 49
Zdrojový kód 4 Enum s částí URL ... 49
Zdrojový kód 5 Úprava budget projektu ... 50

Přílohy

Obrázek 9 Ukázka formuláře pro vytvoření projektu z aplikace FIMS3

New budget project

Name*
Test projekt
Type*
Main project
Linked analytic projects
Manager*
Maryann Torphy
Supervisor*
Maryann Torphy

JIRA project codes*
× JIRA_TEST_KEY - JIRA_TEST_KEY
JIRA Epics
× JIRA_TEST_KEY - Ducimus odit soluta.

Budget*
500000
Estimated other costs
100
Initial hour estimate*
50
Description
This is description

Close Create

Zdroj: Autor

Obrázek 10 Ukázka vytvořeného projektu z aplikace FIMS3

Test projekt
This is description

Details

Type	Main Budget project
Status	In progress
Sync status	Up to Date
Last successful sync	2023-03-15 12:59:31
Last updated on	2023-03-15 12:59:18
Last updated by	fimsaf fimsaf
Progress	42 %
Manager	Maryann Torphy
Supervisor	Maryann Torphy
JIRA project codes	JIRA_TEST_KEY - JIRA_TEST_KEY
JIRA Epics	JIRA_TEST_KEY - Ducimus odit soluta.
Budget	500 000
Estimated other costs	100
Other costs	1 104 691
Max. hours to be profitable	833,17
Initial hour estimate	50
Spent hours total	21
Spent hours previous month	0
Budget spent	210 000
Wage costs	5 250
Estimated hourly rate	-12 093,82
Expected future income	500 000
Estimated current profit	-899 941
Estimated total profit	-617 191
Estimated profitability	-429 %

Archive Edit Delete

Linked income invoices | Linked outcome invoices | Linked analytic projects | Logged work | Costs

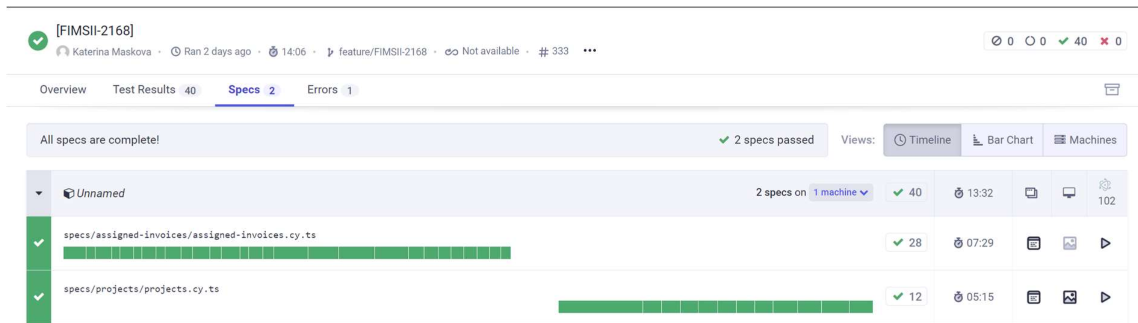
#	Number	Amount without VAT	Currency	Tax date	Issue date	Due date	Payment date	Vendor	Note
71124	Outcome invoice 4497	1 104 691,00	CZK	2023-03-14	2023-03-14	2023-03-14	2023-03-14	Ambrose Maggio	Note automatic test outcome invoice
Total:		1 104 691,00	CZK						

Showing 1 to 1 of 1 entries

Previous 1 Next

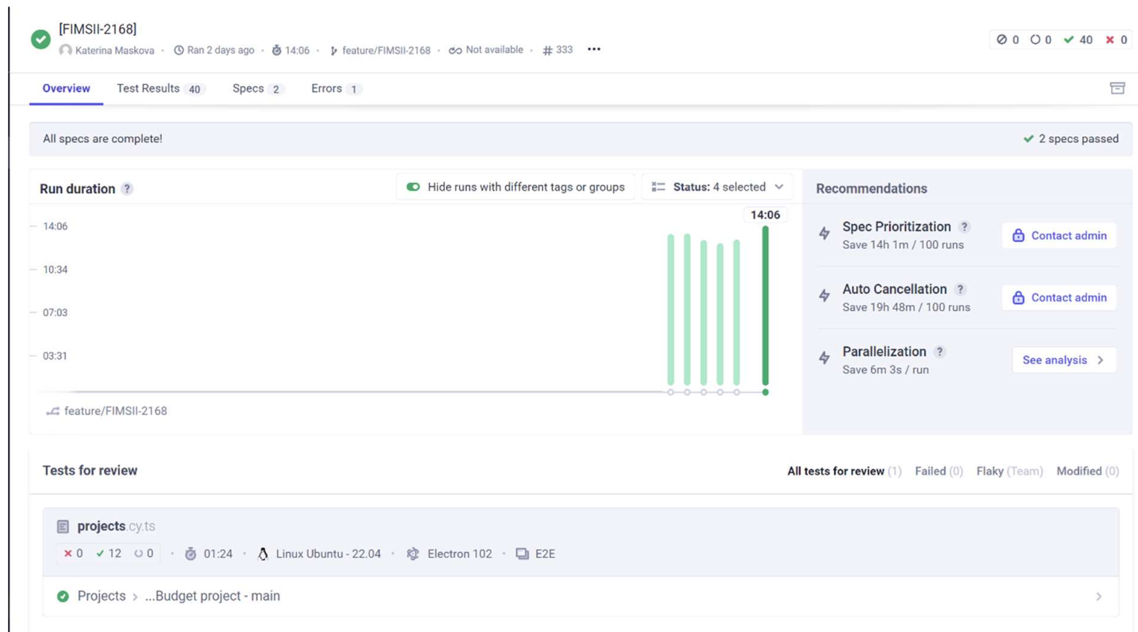
Zdroj: Autor

Obrázek 11 Ukázka testovacího běhu v Cypress Dashboardu



Zdroj: Autor

Obrázek 12 Ukázka testovacího běhu v Cypress Dashboardu



Zdroj: Autor

Tabulka 14 Měření délky manuálního testování

	Měření			Průměr
	1. měření	2. měření	3. měření	
1.1. Vytvoření budget main projektu	5,93	5,81	5,50	5,75
1.2. Vytvoření budget analytic projektu	5,85	5,67	6,25	5,92
1.3. Vytvoření tariff main projektu	3,2	3,33	3,1	3,21
1.4. Vytvoření tariff analytic projektu	3,15	3,37	3,38	3,30
1.5. Vytvoření internal budget main projektu	5,82	5,88	5,65	5,78
1.6. Vytvoření internal budget analytic projektu	5,73	5,98	5,8	5,84
1.7. Vytvoření internal long-term main projektu	0,87	0,98	0,9	0,92

1.8. Vytvoření internal long-term analytic projektu	0,9	0,85	0,97	0,91
2.1. Úprava budget projektu	7,85	7,31	7,13	7,43
2.2. Úprava tariff projektu	1,65	1,3	1,53	1,49
2.3. Úprava internal budget projektu	7,6	8	7,67	7,76
2.4. Úprava internal long-term projektu	2,03	2,25	2,15	2,14
3.1. Připojení income faktury k budget main projektu	2,88	3,01	0,67	2,19
3.2. Připojení income faktury k budget analytic projektu	0,92	0,87	1	0,93
3.3. Připojení outcome faktury k budget main projektu	2,67	3	2,92	2,86
3.4. Připojení outcome faktury k budget analytic projektu	0,83	0,92	0,78	0,84
3.5. Připojení income faktury k tariff main projektu	2,72	2,92	2.95	2,82
3.6. Připojení income faktury k tariff analytic projektu	0,87	0,9	0,95	0,91
3.7. Připojení outcome faktury k tariff main projektu	3,03	2,9	2,82	2,92
3.8. Připojení outcome faktury k tariff analytic projektu	0,72	0,8	0,82	0,78
3.9. Připojení income faktury k internal budget main projektu	2,87	2,98	3,17	3,01
3.10. Připojení income faktury k internal budget analytic projektu	0,72	0,82	0,78	0,77
3.11. Připojení outcome faktury k internal budget main projektu	2,98	2,78	3,05	2,94
3.12. Připojení outcome faktury k internal budget analytic projektu	0,72	0,87	0,8	0,80
3.13. Připojení income faktury k internal long-term main projektu	2,93	2,77	2,83	2,84
3.14. Připojení income faktury k internal long-term analytic projektu	0,83	0,93	1,03	0,93
3.15. Připojení outcome faktury k internal long-term main projektu	3,17	2,88	2,93	2,99
3.16. Připojení outcome faktury k internal long-term analytic projektu	0,93	0,9	0,83	0,89
4.1. Úprava faktury připojené k budget projektu	4,17	3,95	4,43	4,18
4.2. Úprava faktury připojené k tariff projektu	3,53	3,68	3,9	3,70
4.3. Úprava faktury připojené k internal budget projektu	4,25	4,03	3,82	4,03
4.4. Úprava faktury připojené k internal long-term projektu	4,15	3,75	3,83	3,91
5.1. Smazání income faktury z budget projektu	3,33	3,05	3,43	3,27
5.2. Smazání outcome faktury z budget projektu	3,2	3,6	3,47	3,42
5.3. Smazání income faktury z tariff projektu	3,42	3	3,1	3,17
5.4. Smazání outcome faktury z tariff projektu	3,48	3,57	3,68	3,58
5.5. Smazání income faktury z internal budget projektu	3,18	3,42	3,12	3,24
5.6. Smazání outcome faktury z internal budget projektu	2,97	3,27	3,3	3,18
5.7. Smazání income faktury z internal long-term projektu	3,35	3,5	3,52	3,46
5.8. Smazání outcome faktury z internal long-term projektu	3,42	3,03	3,22	3,22

Zdroj: Autor

Tabulka 15 Výpočet návratnosti nákladů po měsících

	Manuální	Automatizované	Ušetřeno celkem	Náklady na automatizaci
1. měsíc	17010	6782	10228	112274
2. měsíc	17010	6782	10228	102046
3. měsíc	17010	6782	10228	91818
4. měsíc	17010	6782	10228	81590
5. měsíc	17010	6782	10228	71363
6. měsíc	17010	6782	10228	61135
7. měsíc	17010	6782	10228	50907
8. měsíc	17010	6782	10228	40680
9. měsíc	17010	6782	10228	30452
10. měsíc	17010	6782	10228	20224
11. měsíc	17010	6782	10228	9996
12. měsíc	17010	6782	10228	-231
13. měsíc	17010	6782	10228	-10459
14. měsíc	17010	6782	10228	-20687
15. měsíc	17010	6782	10228	-30914
16. měsíc	17010	6782	10228	-41142
17. měsíc	17010	6782	10228	-51370
18. měsíc	17010	6782	10228	-61598
19. měsíc	17010	6782	10228	-71825
20. měsíc	17010	6782	10228	-82053

Zdroj: Autor