TECHNICKÁ UNIVERZITA V LIBERCI
Ekonomická fakulta

Návrh optimalizace E2E testů pro webovou aplikaci EXevido

Bakalářská práce

Studijní program: B6209 Systémové inženýrství a informatika
Studijní obor: Manažerská informatika

Autor práce: **Jakub Jukl**
Vedoucí práce: Ing. Michal Dostál
Katedra informatiky

Liberec 2022

TECHNICKÁ UNIVERZITA V LIBERCI
Ekonomická fakulta

Zadání bakalářské práce

Návrh optimalizace E2E testů pro webovou aplikaci EXevido

Jméno a příjmení: **Jakub Jukl**
Osobní číslo: E19000220
Studijní program: B6209 Systémové inženýrství a informatika
Studijní obor: Manažerská informatika
Zadávající katedra: Katedra informatiky
Akademický rok: **2021/2022**

Zásady pro vypracování:

1. Webové aplikace a jejich testování
2. Agilní metodiky vývoje
3. Návrh optimalizace E2E testů a její implementace
4. Zhodnocení a závěr

Rozsah grafických prací:
Rozsah pracovní zprávy: 30 normostran
Forma zpracování práce: tištěná/elektronická
Jazyk práce: Čeština

Seznam odborné literatury:

- AMODEO, Enrique, 2015. *Learning behavior-driven development with JavaScript: create powerful yet simple-to-code BDD test suites in JavaScript using the most popular tools in the community*. Birmingham: Packt Publ. Community experience distilled. ISBN 978-1-78439-264-2.
- GUNDECHA, Unmesh, 2012. *Selenium testing tools cookbook: over 90 recipes to build, maintain, and improve test automation with Selenium Webdriver*. Birmingham Mumbai: Packt Publ. Quick answers to common problems. ISBN 978-1-84951-574-0.
- ROSE, Seb, Matt WYNNE a Aslak HELLESØY, 2015. *The cucumber for Java book: behaviour-driven development for testers and developers*. Dallas, Texas: The Pragmatic Bookshelf. The pragmatic programmers. ISBN 978-1-941222-29-4.
- SMART, John Ferguson, 2015. *BDD in action: Behavior-Driven Development for the whole software lifecycle*. Shelter Island, NY: Manning Publications. ISBN 978-1-61729-165-4.
- ŠOCHOVÁ, Zuzana a Eduard KUNCE, 2014. *Agilní metody řízení projektů*. Brno: Computer Press. ISBN 978-80-251-4194-6.
- PROQUEST, 2021. Databáze článků ProQuest [online]. Ann Arbor, MI, USA: ProQuest. [cit. 2021-09-26]. Dostupné z: http://knihovna.tul.cz

Konzultant: Bc. Ondřej Jakub – Seniorní programátor pro Actis,s.r.o

Vedoucí práce: Ing. Michal Dostál
Katedra informatiky

Datum zadání práce: 1. listopadu 2021
Předpokládaný termín odevzdání: 31. srpna 2023

L.S.

doc. Ing. Aleš Kocourek, Ph.D.
děkan

Ing. Petr Weinlich, Ph.D.
vedoucí katedry

V Liberci dne 1. listopadu 2021

Prohlášení

Prohlašuji, že svou bakalářskou práci jsem vypracoval samostatně jako původní dílo s použitím uvedené literatury a na základě konzultací s vedoucím mé bakalářské práce a konzultantem.

Jsem si vědom toho, že na mou bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb., o právu autorském, zejména § 60 – školní dílo.

Beru na vědomí, že Technická univerzita v Liberci nezasahuje do mých autorských práv užitím mé bakalářské práce pro vnitřní potřebu Technické univerzity v Liberci.

Užiji-li bakalářskou práci nebo poskytnu-li licenci k jejímu využití, jsem si vědom povinnosti informovat o této skutečnosti Technickou univerzitu v Liberci; v tomto případě má Technická univerzita v Liberci právo ode mne požadovat úhradu nákladů, které vynaložila na vytvoření díla, až do jejich skutečné výše.

Současně čestně prohlašuji, že text elektronické podoby práce vložený do IS/STAG se shoduje s textem tištěné podoby práce.

Beru na vědomí, že má bakalářská práce bude zveřejněna Technickou univerzitou v Liberci v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších předpisů.

Jsem si vědom následků, které podle zákona o vysokých školách mohou vyplývat z porušení tohoto prohlášení.

1. května 2022 Jakub Jukl

Anotace

Tato bakalářská práce Návrh optimalizace E2E testů pro webovou aplikaci EXevido se zabývá metodami testování webových aplikací a optimalizací E2E testů pro testování konkrétní webové aplikace.

V práci jsou popsány druhy webových aplikací a jejich testování. Dále popisuje agilní metodiky a jejich provázanost s testováním díky behavior-driven development a test-driven development.

Praktická část je soustředěna na vyhotovení nového řešení E2E testů pro testování webové aplikace s popsáním klíčových návrhových vzorů a problémů.

Klíčová slova

Testování webových aplikací, webové aplikace, behavior-driven development, test-driven development, end-to-end testování, JavaScript, Protractor, Cucumber.

Annotation

This bachelor’s thesis Design of E2E test optimization for web application EXevido studies methods for testing web applications and optimization of E2E tests for specific web application.

First part of this thesis describes types of web applications and their testing procedures. Furthermore, this part addresses agile methodologies and their connection with testing thanks to behavior-driven development and test-driven development.

Second part of this thesis focuses on designing new E2E tests solution for testing web application while describing key design patterns and problems.

Key words

Web application testing, web applications, behavior-driven development, test-driven development, end-to-end testing, JavaScript, Protractor, Cucumber.

Poděkování

Touto cestou bych chtěl poděkovat vedoucímu práce Ing. Michalu Dostálovi za odborné vedení práce a za praktické rady. Dále bych rád poděkoval Bc. Ondřeji Jakubovi za možnost využití jeho zkušeností v této problematice.

Obsah

Seznam obrázků

Seznam ukázek kódu

Seznam zkratek

AJAX	Asynchronous JavaScript and XML
ATM	Automated teller machine
BDD	Behavior-driven development
E2E	End-to-end (testy)
HTTP	Hypertext Transfer Protocol
JS	JavaScript
JSON	JavaScript Object Notation
PIN	Personal identification number
PWA	Progressive web application
QA	Quality assurance
ROI	Return on investment
SEO	Search engine optimization
SPA	Single page application
SSB	Site-specific browser
SW	Software
TDD	Test-driven development
XML	Extensible Markup Language

Úvod

E2E (end-to-end) testy jsou v dnešním agilním světě nedílnou součástí procesu vývoje webových aplikací. Společnosti chápou důležitost testování svých software produktů a potřebu co nejvíce tohoto procesu automatizovat. Proto jsou nyní, místo ručního testování, s výhodou aplikovány právě E2E testy.

Hlavním cílem této bakalářské práce je navrhnout optimalizované řešení E2E testů pro testování webové aplikace EXevido. Zároveň by autor rád seznámil čtenáře s dalšími možnostmi a praktickou využitelností testování webových aplikací.

V první části se bakalářská práce zabývá problematikou testování webových aplikací. Cílem této části je krátce představit nejoblíbenější druhy webových aplikací a možnosti jejich testování. Další kapitoly se zabývají obecným seznámením čtenáře s agilními metodikami vývoje. V neposlední řadě je pak znázorněno propojení mezi automatizovaným testováním a agilními přístupy.

V druhé, praktické, části je navržena optimalizace dosavadního řešení E2E testů pro webovou aplikaci EXevido. Vývojáři této webové aplikace nebyli s dosavadním řešením spokojeni, a proto autor práce navrhl řešení nové.

1 Webové aplikace

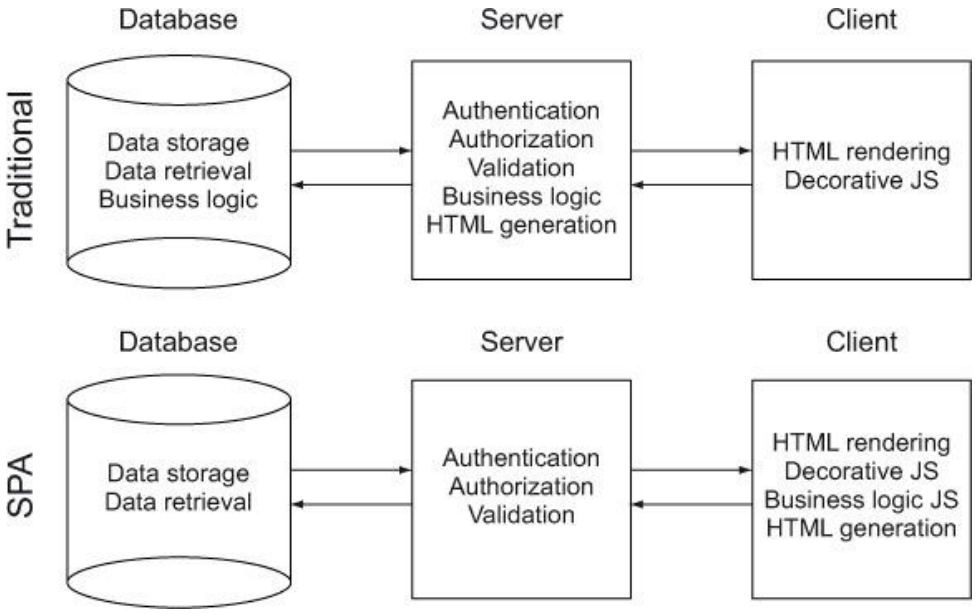
Webové aplikace jsou počítačové programy, které běží na vzdáleném serveru (v cloudu) a přistupuje se k nim skrze internetový prohlížeč (Athmeeya, 2020). Luchaninov (2021) udává, že rozdíl mezi webovou stránkou a aplikací je především v jejich dynamičnosti. Webová stránka zobrazuje stále stejný obsah, ale webová aplikace přizpůsobuje výstup na základě vstupních informací. Lze tedy zjednodušeně říci, že čím více je webová stránka dynamická, tím blíže má k webové aplikaci.

1.1 Architektura webových aplikací

Z hlediska architektury existuje mnoho druhů webových aplikací. V následujících podkapitolách jsou představeny ty nejpodstatnější.

1.1.1 Single page web application

Mikowski (2014) uvádí jako charakteristický rys jednostránkových webových aplikací neboli SPA (Single page application), že se nemusejí znovu načítat během používání. Největšími dnešními zástupci jsou aplikace jako Facebook, Twitter nebo GitHub. Hlavní výhodou je rychlé renderování (= načítání) obsahu, ale za cenu špatné SEO (optimalizace pro vyhledávače).



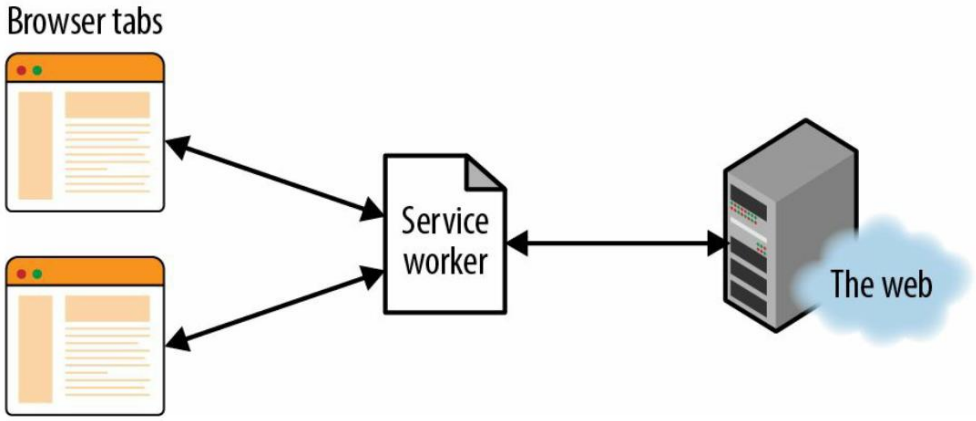
Obrázek 1 – Rozdíl mezi odpovědnostmi u SPA a tradiční webové aplikace
Podle Mikowski (2014)

Obrázek 1 zobrazuje rozdíl odpovědností jednotlivých komponent mezi jednostránkovou aplikací a tradiční webovou aplikací. U tradiční aplikace řeší většinu logiky server, kdežto

u SPA se tato odpovědnost přenáší ke klientovi a server se stará jen o autorizaci, autentizaci a ověřování (Mikowski, 2014).

1.1.2 Progressive web application

Ater (2017) představuje progresivní webové aplikace (PWA) jako podobné jednostránkovým aplikacím, avšak umožňující i práci offline. Díky tomu více stírají rozdíl mezi webovými a nativními aplikacemi (PWA nabízejí funkčnost nainstalované aplikace bez nutnosti instalace). Toto je umožněno přidáním další vrstvy (tzv. service worker) mezi server a klienta. Hlavní nevýhodou je nutnost podpory prohlížečem. Například Firefox v roce 2021 ukončil podporu SSB (site-specific browser), což je funkce umožňující přidat si PWA jako zástupce na plochu a pak k ní přistupovat ze zjednodušeného prohlížeče (Townsend, 2020).

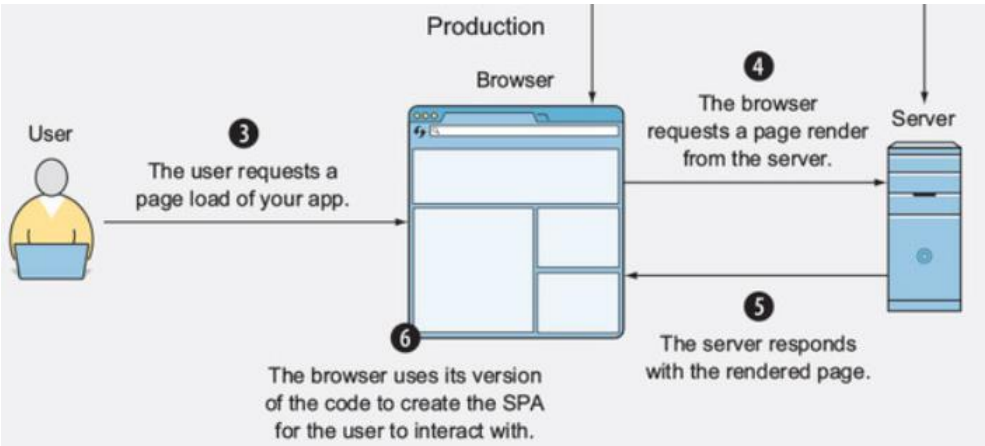


Obrázek 2 – Architektura PWA
Zdroj: (Ater, 2017)

Obrázek 2 ukazuje kam přibyla v architektuře nová vrstva service worker. Tento service worker je schopný komunikovat se serverem i poté, co uživatel zavřel prohlížeč a zároveň je schopný komunikovat s klientem i pokud je server nedostupný (Ater, 2017).

1.1.3 Isomorphic web application

Dle Gordona (2018) je izomorfní (stejnorodá) webová aplikace kombinací jednostránkové webové aplikace a tradiční webové aplikace (renderované na serveru). Při první návštěvě uživatelem je aplikace vyrenderována serverem a obsah poslán klientovi. Když se uživatel pohybuje po aplikaci, tak se aplikace chová stejně jako SPA a každá stránka je vykreslována klientem. Tato architektura řeší, mimo jiné, problémy se SEO u jednostránkových aplikací, kde SEO boti (web crawlers) vidí něco jiného než uživatelé.



Obrázek 3 – Posloupnost vykreslování izomorfní webové aplikace
Zdroj: (Gordon, 2018)

Na obrázku (Obrázek 3) Gordon (2018) znázorňuje, jaká je posloupnost akcí při přístupu k izomorfní webové aplikaci. Body jedna a dva byly vynechány, protože značí proces vývoje. Při pokusu o načtení stránky si klient od serveru vyžádá vykreslenou aplikaci. Po poskytnutí obsahu serverem klient zobrazí uživateli aplikaci jako jednostránkovou.

1.2 Testování webových aplikací

Po představení webových aplikací budou nyní uvedeny způsoby jejich testování. Existuje spousta možností, jak testovat webové aplikace, popřípadě stránky. Všichni weboví vývojáři znají rutinní postup, kdy něco upraví v kódu a pak ručně testují, jestli se změny projevily tak, jak chtěli. Výsledkem tohoto procesu je pouze pachuť z vyvíjení nových vlastností způsobená ohromnou časovou náročností důkladného manuálního testování. Proto je práce zaměřena na automatizované testování webových aplikací.

1.2.1 Fáze testování webových aplikací

Athmeeya (2020) uvádí existenci několika testovacích fází. Každá z nich je neméně důležitá. Vynechání některé fáze může znamenat infiltraci produkčního prostředí bugem, což má za následek ztrátu důvěry zákazníků.

Je logické začít testováním funkčnosti. V této fázi je ověřováno, zda aplikace vyhovuje veškerým zadaným specifikacím (Athmeeya, 2020). Dle Johansena (2011) by mělo být zahrnuto i testování business logiky. Typickým příkladem je ověřování funkčnosti formulářů nebo práce s uloženými daty. Dalším krokem je ověření zachování kompatibility mezi podporovanými prohlížeči.

Athmeeya (2020) uvádí, že následovat by mělo testování výkonu. Zjišťuje se, jak se aplikace chová při kombinaci různé úrovně zatížení a specifických podmínek. Tato fáze se dělí na load testing a stress testing. Při load testingu je na aplikaci vyvíjen přiměřený nápor. Hlavním účelem je zjistit stabilitu aplikace v různých podmínkách, jako je nízká rychlost připojení a různé kombinace hardwaru. Stress testing na druhé straně zkouší, jak velký nápor aplikace vydrží, dokud nepřestane fungovat. Jako poslední dochází k testování bezpečnosti, aby se zajistila funkčnost autentizace (přihlášení), autorizace (správné nastavení práv), šifrování a jiných důležitých aspektů každé webové aplikace.

1.2.2 Unit testing

Pro ověřování funkčnosti jednotlivých komponent se používají unit testy. Johansen (2011) uvádí jako princip fungování unit testů nastavení objektu (aplikace) do známého stavu a zavolání metody. Následným porovnáním výsledného stavu s očekávaným stavem se ověřuje funkčnost dané metody.

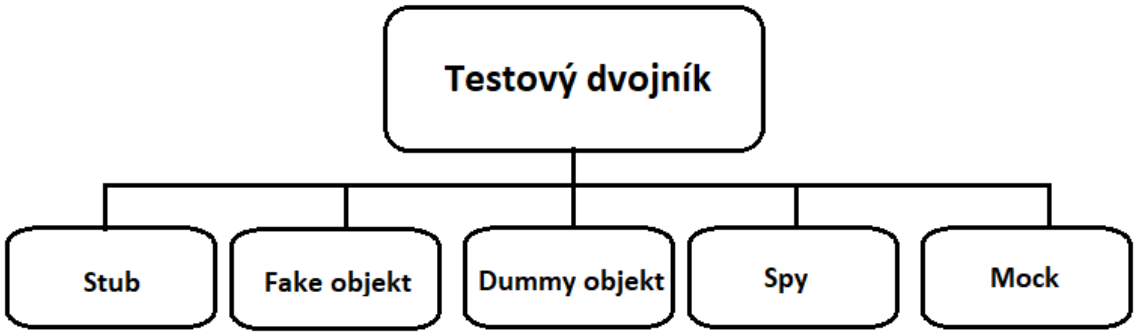
Khorikov (2020) definuje unit testy pomocí tří zásad, které musí splňovat: ověřování malé jednotky, rychle a izolovaně. Johansen (2011) odůvodňuje zásadu rychlosti ochotou vývojářů testy pouštět. Trvalo-li by jejich provedení příliš dlouho, tak by se tato ochota snížila. Khorikov (2020) dále hovoří o existenci dvou přístupů k izolovanosti, které nazývá klasický přístup a Londýnský přístup. Blíže jim jsou věnovány pozdější kapitoly.

1.2.2.1 Testoví dvojníci

Khorikov (2020) představuje testové dvojníky jako objekty, které podporují minimálně část API skutečného objektu, ale nemusí se nutně chovat stejně jako skutečný objekt. Zjednodušeně se dá říct, že pokud něco vypadá jako kachna, kváká to jako kachna, ale neumí to plavat, tak se může jednat o testového dvojníka.

Koskela (2013) doplňuje možnost využít dvojníky k nasimulování podmínek, jenž by jinak bylo složité vytvořit a sledovat jinak neviditelné interakce. Dále dvojníci umožňují oprostit se od náhodných jevů. Krásným příkladem jsou funkce reagující specificky na různý čas. Bez zadaného času, by se konzistentně testovali jen obtížně.

Pro usnadnění testování se využívá několik druhů testových dvojníků, z nichž každý má své unikátní využití a pomáhá dodržovat zásady unit testů definované výše. Na obrázku (Obrázek 4) je vidět pět typů testových dvojníků.



Obrázek 4 – Rozdělení testových dvojníků
Podle Koskela (2013)

Typy dvojníků se dělí na ty, kteří připravují půdu pro testy, což jsou stuby, falešné objekty a dummy objekty (Kaczanowski, 2013).

- Stuby se využívají se pro nahrazování s nejjednodušší možnou implementací, kde typickým příkladem je pouhé určení návratové hodnoty (Koskela, 2013).
- Falešný objekt poskytuje zjednodušenou implementaci původního objektu. Používá se například pro nahrazení databázových tříd (Koskela, 2013).
- Dummy objekt je pouze prázdná schránka bez jakékoliv implementace metod (Kaczanowski, 2013).

Druhý typ dvojníků ověřuje komunikaci mezi objekty, i když mohou být též použiti pro nastavování prostředí pro testy, je to pouze jejich druhotný efekt (Kaczanowski, 2013).

- Koskela (2013) představuje testové „špióny“ jako objekty hlásící všechno, co se s nimi děje. Díky nim lze ověřit, jestli byla zavolána určitá funkce a případně kolikrát. Jejich unikátnost spočívá v ponechání si původní implementace metod.
- Dle Koskely (2013) jsou mocky nejkomplexnější ze všech typů dvojníků, jelikož kombinují jejich funkcionalitu. Mock může vracet různé hodnoty pro různé předané parametry nebo ověřovat kolikrát se funkce zavolala. Na rozdíl od testového „špióna“ si mock neponechává původní implementaci metod.

1.2.2.2 Přístupy k izolovanosti

Když už jsou výše představeni testoví dvojníci a k čemu jsou dobří, pak může být navázáno přístupy k izolovanosti. Nejdříve bude blíže představen Londýnský přístup.

Dle Khorikova (2020) je u Londýnského přístupu k izolovanosti přistupováno tak, že veškeré závislosti jsou nahrazeny jejich dvojníky. Díky tomu se testování soustředí jen na testovanou třídu, tedy malou jednotku kódu.

```
public class Bank() {
    private String name;
    private List<AuthenticatedUser> authUsers;

    public Bank(String name) {
        this.name = name;
    }

    public AuthenticatedUser authenticate(CreditCard card, Integer
     pin) {
        for (AuthenticatedUser authUser : authUsers) {
            if (authUser.creditCard == card && authUser.pin == pin) {
                return authUser;
            }
        }
        return null;
    }
}

public class ATM() {
    private Bank bank;
    private AuthenticatedUser authUser;
    private List<Banknote> banknotes;

    public ATM(Bank bank) {
        this.bank = bank;
    }

    public void authenticate(CreditCard card, Integer pin) throws
            WrongCredentialsException {
        authUser = bank.authenticate(card, pin);
        if (authUser == null) {
            throw WrongCredentialsException;
        }
    }

}
```

Ukázka kódu 1 – Testovaný systém (System under test)
Vlastní tvorba

Všechny ukázky kódu jsou psané v jazyce Java a Groovy s využitím testovacího frameworku Spock. V ukázce kódu (Ukázka kódu 1) jsou vidět části tříd Bank a ATM použitých pro demonstraci rozdílů mezi Londýnským přístupem k testování a klasickým přístupem. Tento kód umožňuje přihlášení uživatelů do bankomatu. Třída ATM (bankomat) má atribut vlastnické banky, přihlášeného uživatele a bankovky, které má k dispozici. Třída Bank (banka) má pro jednoduchost pouze atribut jména a všech uživatelů, kteří mohou využívat bankomaty.

Pokud se chce uživatel přihlásit, tak musí do bankomatu vložit svou kreditní kartu a zadat PIN (Personal Identification Number). Bankomat pak požádá svou banku o ověření, tedy zda existuje uživatel se zadanou kartou a PINem. Banka následně vrátí buď existujícího uživatele, nebo null. Vrátí-li banka null, bankomat vyhodí výjimku, že uživatel s těmito přihlašovacími údaji neexistuje.

```
public void authenticateValidUser() {
    given:
    Bank bankStub = new Stub(Bank)
    AuthenticatedUser user = new AuthenticatedUser()
    bankStub.authenticate(_, _) >> user
    ATM atm = new ATM(bankStub)

    when:
    atm.authenticate(new CreditCard(), 123)

    then:
    atm.authUser == user
    noExceptionThrown()
}

public void authenticateInvalidUser() {
    given:
    Bank bankStub = new Stub(Bank)
    bankStub.authenticate(_, _) >> null
    ATM atm = new ATM(bankStub)

    when:
    atm.authenticate(new CreditCard(), 123)

    then:
    atm.authUser == null
    thrown(WrongCredentialsException)
}
```

Ukázka kódu 2 – Unitové testy dle Londýnského přístupu
Vlastní tvorba

V ukázce kódu (Ukázka kódu 2) je vidět implementace testů na otestování metody authenticate třídy ATM. Testy jsou rozděleny do tří sekcí, kde v první se nastavují objekty pro daný test, v druhé se provádí testovaná metoda, a nakonec se ověřuje výsledek testované metody. V obou testech je vytvořen dvojník objektu Bank. V prvním testu vrací metoda authenticate stubovaného objektu uživatele vytvořeného výše. Podtržítka slouží jako „divoké karty“ pro předávané parametry. Určuje se tím, že při každém zavolání dané metody s libovolnými dvěma argumenty se vrátí stejná hodnota. Po zavolání metody authenticate bankomatu se očekává nastavení daného uživatele jako přihlášeného uživatele a nevyhození žádné výjimky.

Druhý test ukazuje testování chování bankomatu, pokud banka žádného uživatele s danými údaji nezná. Bankomat by pak měl vyhodit výjimku WrongCredentialsException a neměl by být přihlášený žádný uživatel.

Dle výsledku těchto dvou testů je možné usoudit, zda metoda authenticate v třídě ATM funguje správně. Tyto testy však nic nevypovídají o funkčnosti třídy Bank, proto by měly existovat ještě testy na otestování metody authenticate třídy Bank.

Jako druhý představuje Khorikov (2020) klasický přístup. Na rozdíl od Londýnského se klasický přístup neupíná k izolaci tříd a metod, místo toho izoluje testy způsobem, aby bylo možné je spouštět paralelně a v jakémkoliv pořadí. Klade se důraz na eliminaci sdílených stavů, skrz které se testy mohou ovlivňovat. Typickým příkladem sdílených stavů je práce s databází. Zastánci tohoto přístupu využívají mnohem méně dvojníků než zastánci Londýnského přístupu a testují jednotku chování, spíše než jednotku kódu.

```
    public void authenticateValidUser() {
        given:
        Bank bank = new Bank("Kuba")
        AuthenticatedUser user = new AuthenticatedUser()
        CreditCard creditCard = new CreditCard()
        Integer pin = 123
        user.creditCard = creditCard
        user.pin = pin
        bank.authUsers.add(user)
        ATM atm = new ATM(bank)

        when:
        atm.authenticate(creditCard, pin)

        then:
        atm.authUser == user
        noExceptionThrown()
    }

    public void authenticateInvalidUser() {
        given:
        Bank bank = new Bank("Kuba")
        AuthenticatedUser user = new AuthenticatedUser()
        CreditCard creditCard = new CreditCard()
        user.creditCard = creditCard
        user.pin = 124
        bank.authUsers.add(user)
        ATM atm = new ATM(bank)

        when:
        atm.authenticate(creditCard, 123)

        then:
        atm.authUser == null
        thrown(WrongCredentialsException)
    }
```

Ukázka kódu 3 – Unitové testy dle klasického přístupu
Vlastní tvorba.

V ukázce kódu výše (Ukázka kódu 3) je vidět, jak by vypadaly stejné testy na ověření metody authenticate třídy ATM psané v klasickém stylu. Místo vytváření dvojníka objektu Bank je vytvořen skutečný objekt, kterému musí být přiřazen skutečný objekt AuthenticatedUser do listu uživatelů. Testy se tak staly znatelně delší a spadnou i pokud přestane fungovat metoda authenticate třídy Bank. Čas provádění testu se zvýší, jelikož test musí vykonat metodu navíc.

Při porovnání těchto dvou přístupů a praktických ukázek si čtenář může vyvodit závěr, že Londýnský přístup se nese více v duchu unit testů. Dodržuje lépe zásadu testování malé části kódu a testy probíhají rychleji.

Khorikov (2020) dále jako výhodu Londýnského přístupu uvádí ulehčení testování propojených tříd a odhalení přesné lokace problému. Na druhou stranu výhoda klasického

přístupu spočívá v upozornění programátora na důležitost komponenty v celém systému. Pokud po úpravě kódu popadají testy velkého množství tříd, programátor si může uvědomit, že u této komponenty by měl postupovat opatrněji.

1.2.3 Integrační testování

Na rozdíl od unit testů, kde je testována každá komponenta samostatně, je v integračních testech testována funkčnost aplikace jako celku. Johansen (2011) ukazuje rozdíl mezi unitovými a integračními testy na příkladu auta. Unitový test zkontroluje funkčnost volantu tak, že pokud se s ním dá otočit, tak funguje. Oproti tomu integrační test otočí volantem a zkontroluje, jestli jsou kola otočena správným směrem.

Kaczanowski (2013) udává jako hlavní nevýhodu integračních testů jejich pomalost. Většinou běží mnohem pomaleji než unitové testy. Vyžadují, aby byla předem aplikace nějak nastavená a volají objekty, které často odpovídají pomalu (databáze, webové služby).

Londýnský styl považuje jakýkoliv test používající reálné objekty (kromě testované třídy) za integrační (Khorikov, 2020). Na ukázce kódu s unitovými testy dle Londýnského přístupu (Ukázka kódu 2) jsou oba testy podle této definice integrační. Opravit se dají nahrazením (nebo odebráním) veškerých reálných objektů, kromě ATM, za dvojníky. Třída CreditCard může být nahrazena dummy objektem, v popisovaném případě je však použit pro nahrazení null. Testy po úpravě jsou znázorněny v ukázce kódu níže (Ukázka kódu 4).

```
    public void authenticateValidUser() {
        given:
        Bank bankStub = new Stub(Bank)
        AuthenticatedUser userStub = new Stub(AuthenticatedUser)
        bankStub.authenticate() >> userStub
        ATM atm = new ATM(bankStub)

        when:
        atm.authenticate(null, 123)

        then:
        atm.authUser == userStub
        noExceptionThrown()
    }

    public void authenticateInvalidUser() {
        given:
        Bank bankStub = new Stub(Bank)
        bankStub.authenticate() >> null
        ATM atm = new ATM(bankStub)

        when:
        atm.authenticate(null, 123)

        then:
        atm.authUser == null
        thrown(WrongCredentialsException)
    }
```

Ukázka kódu 4 – Přepsané integrační testy na unitové testy
Vlastní tvorba

Khorikov (2020) udává, že pokud test nesplňuje alespoň jedno ze základních kritérií unitových testů, tak jak jim rozumějí klasici, je v klasickém stylu test integrační. Pokud tedy test testuje více než jednu jednotku chování nebo je pomalý anebo neběží v izolaci od ostatních testů.

Jednotkou chování rozumíme něco, co by shledal jako přínosné člověk věnující se business logice bez znalosti kódu (Koskela, 2013). Jako příklad uvádí Khorikov (2020) příkaz psovi. Pokud zavolám: „Ke mně!“ Tak jednotkou chování je, že pes ke mně přijde a ne, že čtyřikrát pohne pravou přední packou dopředu.

1.2.4 End-to-end testování

Posledním zmíněným typem testů jsou E2E testy. Tyto testy ověřují, zda kód funguje z pohledu zákazníka. Testuje se celá aplikace tak, jak by jí používal uživatel. V E2E testech se dvojníci používají jen zřídka a z uváděných tří druhů testů trvá jejich vykonání nejdéle (Kaczanowski, 2013). Khorikov (2020) dodává, že někteří developeři o nich mluví také jako o UI/GUI testech, protože ověřují funkčnost skrze uživatelské prostředí. Kvůli integraci veškerých částí aplikace a závislosti na změnách frontendu, jsou E2E testy ze všech druhů testů nejnáročnější na údržbu.

2 Agilní metodiky vývoje

Stellman a Greene (2014) představují agilní metodiku jako sadu metod, které pomáhají týmům uvažovat a pracovat efektivněji. Každá z metodik se skládá z praktik optimalizovaných pro jednoduchou implementaci a přijetí za své. Dle Mednieksе (2021) agilní přístup k vývoji software dorazil jako odpověď na předchozí styly vedení vývoje software. Nedá se považovat za soupeře anebo vylepšení předchozích stylů, jelikož neobsahuje veškeré jejich vlastnosti.

2.1 Vznik pojmu „agilní“

V únoru 2001 se sešlo 17 lidí praktikujících různé agilní přístupy, aby jim našli společné jméno (Mednieks, 2021). Dohodli se a sepsali Agilní manifest (Beck a kol., 2001). V tomto manifestu jsou vyloženy čtyři hlavní body:

1) „***Jednotlivci a interakce*** před procesy a nástroji“ (Beck a kol., 2001)
2) „***Fungující software*** před vyčerpávající dokumentací“ (Beck a kol., 2001)
3) „***Spolupráce se zákazníkem*** před vyjednáváním o smlouvě“ (Beck a kol., 2001)
4) „***Reagování na změny*** před dodržováním plánu“ (Beck a kol., 2001)

„Jakkoliv jsou body napravo hodnotné, bodů nalevo si ceníme více.“ (Beck a kol., 2001)

Tyto body zobrazují hlavní rozdíly mezi agilním a tradičním přístupem, kde agilní přístup si více cení bodů vlevo a tradiční přístup je znám svým důrazem na body vpravo. Tato událost dala jednotný název skupině metod řízení vývoje, jako jsou Scrum, Extreme Programming, Pragmatic Programming a další (Mednieks, 2021).

2.2 Výhody agilních metodik oproti standardním

Dle Salameh (2014) měly týmy využívající agilní přístup pětkrát vyšší efektivitu oproti týmům využívajícím standardní. Dále měly společnosti využívající agilní metodiky jedenáctkrát vyšší ROI (return on investment).

Salameh (2014) ke svým výsledkům dodává, že ve světě vývoje SW, kde se potřeby zákazníků rychle mění a kolikrát oni sami ani nevědí, co chtějí, jsou agilní metodiky jasnou volbou pro většinu týmů a společností. Častá komunikace se zákazníky a testování konceptů v brzkých fázích vývoje zvyšuje možnost reakce na nálady na trhu, což v důsledku zvyšuje spokojenost zákazníků a všeobecné zisky.

U vodopádového (waterfall) přístupu často dochází k zastarání produktu už v den dodání (Stellman a Greene, 2014). Shore a kol. (2022) udávají jako největší výhodu agilního přístupu jeho jednoduchost. Pokud bychom žili v perfektním světě, tak by dost možná byl standardní přístup lepší. Jenže v reálném světě má většina týmů a zákazníků problém tento přístup zvládnout. Oproti tomu agilní přístup je v principu velmi jednoduchý a přináší benefity i pokud není zvládnut perfektně.

Stellman a Greene (2014) doplňují informace o projektech, pro které je lepší použít standardních metodik. Typickým příkladem je situace, kdy zákazník (například nemocnice) přesně ví, co potřebuje a že se jeho požadavky nebudou měnit. V tom případě dokáže waterfall přístup nabídnout efektivní procesy.

2.3 Agilní trendy

Digital.ai (2021) každoročně provádí State of Agile Report, což je celosvětový průzkum ohledně využívání agilních metodik. Jeho patnáctý ročník probíhal mezi únorem a dubnem 2021 a účastnilo se ho 4 182 respondentů, z nichž 1 382 odeslalo plné odpovědi.

Dle Digital.ai (2021) se v roce 2021 oproti předchozímu roku zvýšilo procento developerských týmů, které aplikují agilní metodiky z 37 % na 86 %. Současně s tím se zdvojnásobil počet podniků nepodnikajících v IT využívajících agilní metodiky. Jako největší výhody pro využívání agilních metodik uvedli respondenti zvýšenou možnost pro změnu priorit a zrychlené dodávání SW.

Digital.ai (2021) uvádí, že jako největší překážky pro přechod na agilní metodiky vnímá 46 % respondentů nekonzistentní procesy a praktiky mezi týmy. Nejpoužívanější agilní metodikou je Scrum, kterou používá 66 % respondentů.

Dolezel a kol. (2019) publikovali výzkum zabývající se agilními trendy vývoje SW v České republice. Výzkumu se účastnilo 120 respondentů. Tento průzkum také potvrzuje dominanci metodiky Scrum s 47% zastoupením. Jako nejpoužívanější praktika je označena product backlog (seřazený seznam funkcí, které tým může dodat). Nejméně se používá TDD (test-driven development) a BDD (behavior-driven development). Pouhých 62,5 % respondentů uvedlo využívání unit testů.

Průzkumy jasně ukazují, že současným trendem je přechod k agilním metodikám, a to zejména na populární Scrum, který je velmi jednoduchý na pochopení.

2.4 Scrum

Podle výsledků výše je nejoblíbenější agilní metodika Scrum, a proto v nadcházejících několika odstavcích budou představeny její principy.

2.4.1 Sprint

Jelikož součástí Scrumu je iterativní vydávání softwaru, tak využívá Sprinty. Šochová a Kunce (2014) vysvětlují Sprint jako jednu iteraci. Zjednodušeně řečeno se jedná o časově vymezený úsek, na jehož konci se vydá další iterace funkčního softwaru. Každý Sprint trvá stejně dlouhou dobu (dle Stellmana a Greene (2014) je to často měsíc, ale může se lišit tým od týmu). Sprint se dá i prodloužit, ale mělo by se k tomu uchylovat jen velmi výjimečně, jelikož tím dojde k opoždění vydání softwaru a ztrátě důvěry zákazníka (Šochová a Kunce, 2014).

2.4.2 Product Backlog

Šochová a Kunce (2014) představují Product Backlog jako seřazený seznam všech funkcí, které tým plánuje někdy udělat. Funkcionalita by měla být popsána z pohledu zákazníka. Kromě samotných funkcionalit Product Backlog obsahuje také odhad náročnosti.

2.4.3 Sprint Backlog

Šochová a Kunce (2014) definují Sprint Backlog jako část Product Backlogu, která se stihne vyhotovit za jeden sprint. Může se stát, že se nestihnou všechny funkcionality zpracovat. V takovém případě se nezpracované funkce přesunou do následujícího Sprintu a na retrospektivním meetingu se probere, proč k tomu došlo.

2.4.4 User Story

Stellman a Green (2014) označují User Story jako oblíbený způsob vedení záznamů o funkcích v Product Backlogu. Jsou oblíbené, jelikož popisují příběh. Říkají, kdo chce, co chce a proč to chce.

„Jako **prodavač**, chci **zobrazit stav produktu na skladě**, abych ho **nehledal, pokud je jen poslední vystavený kus**.“

Výše je možné vidět, jak by mohlo vypadat User Story zadané zákazníkem. Obsahuje informace o tom kdo (prodavač), co (zobrazit stav produktu na skladě) a proč (zbytečná práce s hledáním produktu).

Šochová a Kunce (2014) vysvětlují kritéria INVEST, které musí User Story splňovat:

- I jako Independent (nezávislost). User Story nesmí být na sobě závislé.
- N jako Negotiable (popsatelný). Product Owner musí každé User Story rozumět tak, jako by ji napsal on sám. Pokud jí nerozumí, nemůže jí pak dobře vysvětlit týmu a ten ji nemůže dobře zpracovat.
- V jako Valuable (hodnotný). Každá User Story musí mít hodnotu pro uživatele. Pokud tým nevidí přidanou hodnotu, tak se jim bude těžko programovat.
- E jako Estimable (odhadnutelný). Tým musí být schopen odhadnout u dané User Story náročnost.
- S jako Small (malý). Pokud je User Story příliš velká (komplexní), tak ji tým musí rozdělit na více menších.
- T jako Testable (testovatelný). Musíme umět poznat, že je User Story hotová.

Je na Product Owneru, aby zajistil, že každá User Story splňuje výše zmíněná kritéria (Šochová a Kunce, 2014).

2.4.5 Standup Meeting

Dle Šochové a Kunce (2014) se jedná o nejrozšířenější agilní praktiku. Mají formu rychlých denních setkání celého týmu. Používá se termínu Standup, jelikož meetingy probíhají ve stoje, což podporuje zásadu rychlosti. Do 15 minut musí všichni členové týmu sami odpovědět na tři otázky. Co jsem dělal včera? Co budu dělat dnes? Jaké mám problémy? Díky odpovědím na tyto otázky si tým udělá představu v posunu projektu ke společnému cíli (úspěšné dokončení sprintu). Scrum Master se díky tomu dozví o případných problémech, které může odstranit.

2.4.6 Lidé v Scrumu

Kromě praktik je vhodné představit i různé role, bez nichž by se hladký běh Scrumu neobešel.

Stellman a Greene (2014) představují *Scrum Mastera* jako člena týmu starajícího se o odstranění veškerých překážek během Sprintu. Jeho povinností je sledovat, jestli Scrum funguje a případně udělat změny. Dále se snaží motivovat tým a dovést ho ke splnění cílů. Cílem Scrum Mastera by mělo být vytvoření týmu, který se dokáže sám organizovat (self-organizing team). Dle Šochové a Kunce (2014) je v některých společnostech role Scrum Mastera kombinována s jinými rolemi, ale to většinou vede k nedomyšleným následkům. Jako příklad je uvedena kombinace s rolí vývojáře. Takový Scrum Master často preferuje

práci vývojáře, a proto se pak snadno může stát, že tým je bez Scrum Mastera, když ho potřebuje nejvíce.

Stellman a Greene (2014) dále představují roli *Product Owner*. Product Owner se stará o zájmy zákazníků a manažerů. Stará se o backlog a komunikaci s vývojáři, aby vše fungovalo podle představ zákazníků. Dále komunikuje s managementem, aby bylo jasné, kam produkt směřuje. Tato role funguje jako prostředník mezi agilním týmem a jeho okolím (zákazníky a managementem). Šochová a Kunce (2014) dodávají, že pro správné vykonávání svých povinností musí Product Owner pochopit produkt a pochopit zákazníky. K tomu by měl být velmi komunikativní. Pokud chápe produkt, pak ho dokáže jednoduchým jazykem vysvětlit jak zákazníkům, tak managementu, aby v něm všichni našli přidanou hodnotu.

Jako další je představena role, která se mnohým může zdát pro agilní týmy zbytečná, a to *Projektový Manažer*. Dle Šochové a Kunce (2014) se jeho povinnosti v agilních týmech velmi liší. Většinou se stará o administrativní stránku projektu. Například administrativa v informačních systémech firmy, kde je projekt vyvíjen. Kolik času se na projektu strávilo, rozpočet apod.

Nakonec se nesmí zapomenout na celý tým, který má v Scrumu nejdůležitější roli. Dle Stellman a Greene (2014) musí být správný agilní tým multifunkční. V týmu jsou zahrnuti všichni lidé potřební pro vydání fungujícího SW. Pro jeho multifunkčnost však nestačí, že jsou tito lidé jen součástí týmu na papíře. Musí se cítit jakou součást týmu, být angažováni a využíváni. Šochová a Kunce (2014) dodávají nutnost, aby tým byl tzv. self-organizing. Self-organizing tým se dokáže samostatně organizovat. Zní to jako utopie, ale každý tým je jiný. Neexistuje žádný univerzální postup, jak nejefektivněji organizovat všechny týmy. Proto agilní týmy kladou důraz na své členy a možnost změny zavedených procesů pro zvýšení efektivity.

2.5 Water-Scrum-Fall

I když spousta týmů uvádí, že používá metodiku Scrum, tak ve skutečnosti mohou využívat Water-Scrum-Fall, což je něco mezi Waterfall a Scrum přístupy (Stellman a Greene, 2014).

West (2011), který dle Stellmana a Greenea (2014) poprvé použil tento termín, rozděluje postup při této metodice do tří fází. Každé fázi odpovídá jedno slovo z názvu metodiky. První fáze „Water“ značí mnoho času stráveného s projektem předem. Další fáze „Scrum“

vypovídá o využívání agilních metodik (zejména rozšířeného Scrum) pro vývoj. Poslední „Fall“ označuje bariéry častého vydávání verzí.

Ke všem těmto fázím určuje West (2011) problémy, které brání posunutí se na efektivnější metodiky:

- Velké množství počátečních požadavků a detailní požadavky, jenž si stejně uživatel s první verzí rozmyslí.
- Tým nemá společný cíl, čímž by měla být nejvyšší možná přidaná hodnota společnosti.
- Tým není multifunkční, což omezuje jeho efektivitu spočívající ve spolupráci. (Pokud jsou součástí týmu jen vývojáři bez QA, testerů a systémových administrátorů, tak je to špatně.)
- Opomíjení komunikace s klienty po stanovení počátečních požadavků.
- Vydávací aktivity (datové migrace, testování) neprobíhají během sprintu.

Dále dodává West (2011) tři rady, jak se zbavit Water-Scrum-Fall metodiky:

- Zavrhnout trávení času s projektem předem.
- Vytvořit opravdu multifunkční agilní tým složený ze všech rolí, které jsou potřeba pro vydání aplikace.
- Vydávat verze software častěji.

Stellman a Greene (2014) uvádí, že i když tento přístup vede k lepším výsledkům než klasický Waterfall (autoři ho označují za nejefektivnější Waterfall přístup), přesunem na čistě agilní metodiky se dají výsledky ještě více zlepšit.

2.6 Behavior-driven development

Behavior-driven development umožňuje provázat kapitolu o testování webových aplikací a kapitolu o agilních metodikách. Proto bude blíže představen i přesto, že podle výsledků průzkumu výše je to nejméně používaná agilní praktika.

Smart (2015) představuje BDD (behavior-driven development) jako nadstavbu už zavedených agilních metodik a uvádí, že byl vymyšlen Danem Northem jako pomůcka pro snadnější praktikování TDD (test-driven development).

Jelikož byl BDD vymyšlen jako pomůcka pro TDD, tak mají tyto praktiky mnoho společného. Jako nejpodstatnější rozdíl uvádí Rose a kol. (2015) způsob napsání testu.

U TDD přístupu se začíná s padajícím testem, který ověřuje funkčnost zákazníkova požadavku. U BDD tomu je podobně, s rozdílem, že test je napsaný způsobem, aby mu rozuměl každý člen týmu.

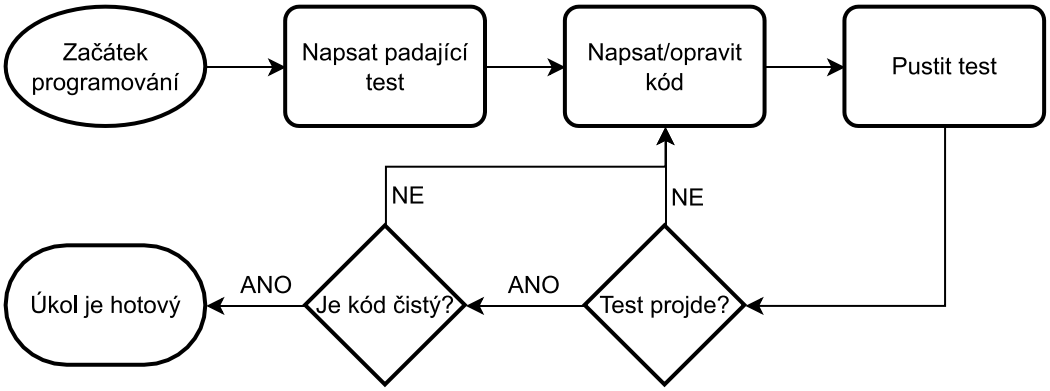
2.6.1 Test-driven development

Rose a kol. (2015) udává metodiku Extreme Programming jako zdroj TDD. Praktika je založena na testu napsaném společně zákazníky a vývojáři. Tento test vystihuje zákazníkovy požadavky a ze začátku padá, jelikož ještě požadavky nejsou splněny. Pozor, nejedná se o unit testy.

Jako příklad si lze představit: Úkolem je postavit dům, ke kterému byla obdržena specifikace. Když je dům postaven, je pozván statik, který zkontroluje, že je dům postaven správně. Po statikovi, který vše schválil, přijde zákazník. Zákazník je rozčílený, protože bylo postaveno úplně něco jiného, než chtěl on.

V tomto příkladu slouží přirovnání statika k unitovým testům. Víme, že dům (aplikace) funguje. Statik nám však neřekne nic o tom, jestli ho zákazník takto chtěl. Že zákazník chtěl místo domu stodolu zjistíme až od něj.

Proto se někdy také říká, že unitové testy ověřují, že je software postaven správně a ne, jestli je postaven správný software (Rose a kol., 2015).



Obrázek 5 – Workflow u test-first přístupu.
Podle Amodeo (2015)

Na obrázku (Obrázek 5) je zobrazen workflow u test-first cyklu, jak ho popisuje Amodeo (2015). Začátkem programování je myšleno obdržení zadání od zákazníka. Takto by se mělo postupovat při dodržování TDD nebo BDD principů. Se zákazníkem napíšeme test, který nejdříve padá, jelikož funkce ještě není implementována. Následně naprogramujeme funkcionalitu a testem ověříme, že funguje podle očekávání zákazníka. Pokud test projde,

tak se můžeme pustit do refactorování kódu, dokud nebudeme spokojeni s jeho technickou stránkou.

2.6.2 Rozdíl mezi BDD a TDD

Hlavní rozdíl spočívá ve způsobu psaní testů. U TDD píšou testy vývojáři, kdežto u BDD jsou testy psány uživateli nebo testery (a vývojáři je pak zprovozňují) (Smart, 2015).

Při praktikování TDD by tedy test vypadal podobně, jako testy ukázané v kapitole o unitových testech. Kdežto při využití BDD bude test vypadat jako v ukázce níže (Ukázka kódu 5) napsané v Gherkinu.

```
Požadavek: Vyhledávání obrázků

Scénář: Vyhledání obrázků s kočkami

Za předpokladu že se dostanu na stránku Google
Když zadám do vyhledávače výraz „kočka“
A dám hledat
Pak mi Google našel články o kočkách
Když přepnu na panel s obrázky
Pak mi Google zobrazil obrázky s kočkami
```

Ukázka kódu 5 – Jednoduchý scénář pro vyhledávání obrázků
Vlastní tvorba

Takto by mohl vypadat jednoduchý požadavek uživatele na funkci. Tuto funkcionalitu může uživatel předat k dalšímu zpracování. Scénář zajišťuje, že Google, jako doposud, bude defaultně vyhledávat články o zadaném výrazu, ale po přepnutí se budou zobrazovat obrázky týkající se vyhledané fráze.

Smart (2015) zmiňuje důležitost příkladů pro praktikování BDD. Na začátku procesu implementace funkce komunikuje tým s uživateli, aby definovali scénáře, jaké má tato funkce zahrnovat. Uživatelé tedy definují sadu příkladů, které definují danou funkcionalitu.

V ukázce níže (Ukázka kódu 6) je znázorněna funkcionalita (požadavek) na posílání zlaťáků (fiktivní měny) mezi hráči v počítačové hře. Uživatelé přišli také se třemi příklady, jak by se posílání zlaťáků mělo podle nich chovat. Pokud hráč posílá zlaťáky existujícímu uživateli a má jich dost, tak by se měly v pořádku odeslat a uživatel chce dostat nějaké vizuální potvrzení o proběhnutí transakce. Dále scénář kontroluje, zda se zlaťáky opravdu odeslaly a nedošlo například k jejich duplikaci.

Druhý scénář určuje chování při pokusu o odeslání většího množství zlaťáků, než má uživatel k dispozici. Z pohledu vývojáře existuje více možností, jak to vyřešit. Uživatelé

navrhují transakci neuskutečnit a dát hráči vizuální zpětnou vazbu, že se transakce nepodařila. Třetí scénář je podobný tomu druhému, jen s rozdílem, že se zlaťáky posílají neexistujícímu hráči.

```
Požadavek: Poslání zlaťáků jinému hráči

Scénář: Převod validního počtu zlaťáků

Za předpokladu že jsem přihlášen do hry
A že „já“ mám 100 zlaťáků
A že „Franta“ má 0 zlaťáků
Když pošlu 50 zlaťáků hráči „Franta“
Pak se objeví hláška „Převod byl úspěšný“
A „já“ mám 50 zlaťáků
A „Franta“ má 50 zlaťáků


Scénář: Převod nevalidního počtu zlaťáků

Za předpokladu že jsem přihlášen do hry
A že „já“ mám 100 zlaťáků
A že „Franta“ má 0 zlaťáků
Když pošlu 150 zlaťáků hráči „Franta“
Pak se objeví hláška „Převod se nepodařil“
A „já“ mám 100 zlaťáků
A „Franta“ má 0 zlaťáků


Scénář: Převod zlaťáků nevalidnímu hráči

Za předpokladu že jsem přihlášen do hry
A že „já“ mám 100 zlaťáků
Když pošlu 50 zlaťáků hráči „Neexistuji“
Pak se objeví hláška „Převod se nepodařil“
A „já“ mám 100 zlaťáků
```

Ukázka kódu 6 – Tři možné scénáře u převodu zlaťáků
Vlastní tvorba

2.6.3 Hlavní výhody BDD

Po představení BDD a vysvětlení rozdílů mezi jím a jeho předchůdcem TDD, mohou být shrnuty hlavní výhody, které plynou z používání této praktiky.

Jelikož agilní metodiky jsou založené na komunikaci, tak BDD přináší možnost, jak informovat všechny osoby podílející se na projektu (zákazníci, vývojáři, testeři, akcionáři…) jednotným jazykem o nových funkcích (Rose a kol., 2015). Dle Smart (2015) slouží takto napsané testy jako živá dokumentace odrážející funkčnost všech těchto funkcí. Dále se dají

testy využívat pro sledování postupu vývoje (u nehotových funkcí neprojdou). Navíc k těmto všem výhodám slouží BDD testy i pro automatické testování regrese (bugy způsobené přidáváním nových funkcí) (Amodeo, 2015).

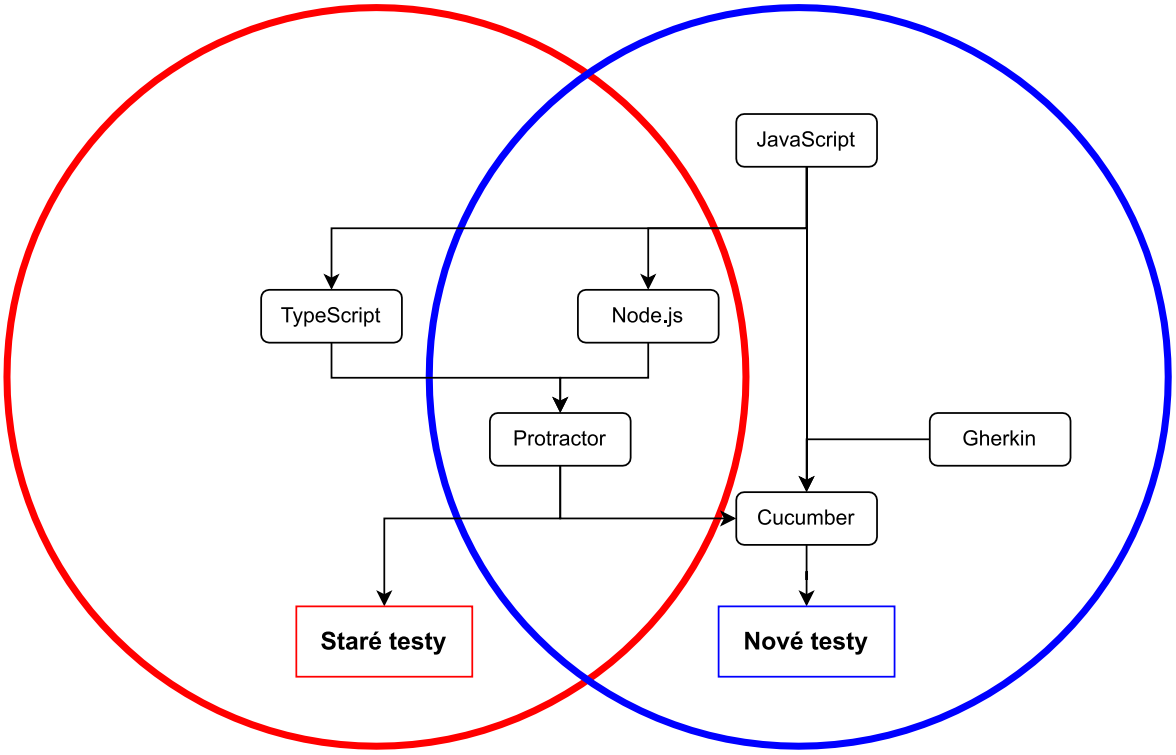
3 Návrh optimalizace E2E testů a její implementace

Autor této bakalářské práce strávil svou roční řízenou praxi jako programátor ve společnosti, kde jsou kromě unit testů využívány i E2E testy pro ověřování funkčnosti aplikace. Původní stav testů byl zásadně nedostačující. Testy procházely pouze občas, a to přesto, že v aplikaci nebyly testované chyby. Výsledkem tedy bylo opětovné pouštění testů, dokud všechny neprošly.

Hlavním úkolem autora této práce bylo E2E testy přepsat. Požadavkem pro nový design E2E testů bylo umožnit využívání BDD a omezení jejich nahodilé nefunkčnosti. Dalším požadavkem bylo rozšíření testů podle napsaných zadání testery.

3.1 Technologie použité k dosažení stanoveného cíle

V následujících odstavcích budou představeny technologie, které byly použity k vypracování praktické části bakalářské práce. Na obrázku (Obrázek 6) je vidět jednoduché grafické znázornění, jak na sobě jednotlivé technologie závisí a jejich využití ve starých nebo nových testech.



Obrázek 6 – Diagram použitých technologií a jejich provázanosti
Vlastní tvorba

3.1.1 JavaScript

Gude (2012) představuje JavaScript jako skriptovací jazyk vyvinutý v roce 1995 Brendanem Eichem pro Netscape. Netscape tímto chtěl umožnit interaktivitu na webu, jelikož nabízel webový prohlížeč Netscape Navigator. V roce 1996 Microsoft použil vlastní verzi JavaScriptu (pojmenovanou JScript) pro jejich prohlížeč Internet Explorer. Existence dvou odlišných verzí stejného jazyka vedla k potřebě vytvoření standardu. Proto v roce 1997 vyšla první edice ECMAScriptu. ECMAScript je standard zajišťující, že weboví vývojáři nemusí podporovat několik verzí JS (JavaScript, JScript).

Wirfs-Brock a Eich (2020) uvádí další osobu, která měla velký přínos pro to, že se JS stal tím, čím je dnes. Douglas Crockford se snažil změnit negativní postoje vývojářů k JavaScriptu (někteří o něm dokonce mluvili jako o nejhorším vynálezu všech dob). Mimo psaní prací o JS napsal také JSLINT, což je linter (nástroj pro analýzu kódu a hledání programátorských, stylistických a dalších chyb) pro ES. Dalším zásadním úspěchem bylo vytvoření JSON objektu. JSON vytvořil pro nahrazení složitého XML něčím jednoduchým a dodnes se používá pro komunikaci mezi frontendem a backendem.

Wirfs-Brock a Eich (2020) dále zmiňují, že zpočátku byl JS využíván pouze pro vykonávání funkcí u klienta (v prohlížeči). Většina webových aplikací v té době byly formuláře, kde JS pouze validoval zadaná data. To se však změnilo s nástupem XMLHTTP, který umožňoval JS kódu asynchronně komunikovat se serverem bez nutnosti stránku znovu načíst. Kolektivně se těmto technologiím říká AJAX (asynchronní JavaScript a XML).

3.1.2 TypeScript

TypeScript je jazyk vyvinutý společností Microsoft, který se kompiluje do ECMAScriptu. Jeho hlavní vlastností je možnost využít statického typování (Wirfs-Brock a Eich, 2020).

3.1.3 Node.js

Young a Harter (2015) představují Node.js jako systém umožňující psaní webových aplikací v JavaScriptu. Webové aplikace se dají (a daly) napsat v JS i bez Node.js, ale neslo to s sebou mnoho obtíží pro vyřešení práce se zapisováním a čtením dat z paměti a sítě. Díky tomu, že tento problém řeší elegantně a rychle, se Node stal velmi populárním.

Young a Harter (2015) dále uvádějí jako hlavní výhody Node.js jeho základní knihovnu, systém modulů a npm. Npm je online softwarový repozitář open-source Node.js projektů a nástroj pro interakci s tímto repozitářem přes příkazový řádek. Pomocí npm se dají stahovat knihovny, spravovat jejich verze a závislosti.

3.1.4 Protractor

Liau (2021) vysvětluje, že Protractor je E2E framework vytvořený Angular týmem pro testování webových aplikací napsaných v Angularu. Pro ovládání webového prohlížeče využívá Selenium webdriver. V roce 2021 rozhodl Angular tým o termínu ukončení vývoje. Od konce roku 2022 nebude Protractor už nikdo oficiálně aktualizovat. Uživatelům bylo doporučeno přejít na modernější řešení, jako je například Cypress. Výhodou využívání Protractoru pro testování Angular webových aplikací byla funkce Control Flow, jenž přetvářela asynchronní požadavky na synchronní a vývojáři tak nemuseli pracovat s asynchronním kódem.

3.1.5 Cucumber

Rose a kol. předtavují Cucumber jako nástroj pro spouštění BDD testovacích scénářů. Scénáře jsou psané v prostém jazyce, ale musí splňovat určitou syntaxi zvanou Gherkin. Gherkin podporuje více než 70 jazyků. Cucumber existuje v provedeních pro spoustu programovacích jazyků. V této práci je použita jeho verze pro JavaScript.

3.2 Výchozí stav

Kód původních E2E testů je napsaný v TypeScriptu. Jako framework byl využíván Protractor. Jak už bylo zmíněno, spolehlivost těchto testů byla nízká. Úkolem autora této práce bylo toto změnit. Dle zadání měl být kód nových E2E testů napsaný pomocí CucumberJS využívajícího Protractor. K dispozici byly i libovolné knihovny dostupné pro Node.js.

Soubor `app.e2e-spec.ts` ve složce `old_e2e` sloužil jako hlavní soubor pro spouštění testů. Na začátku se připravují objekty s uživateli, rolemi a další, které se následně používají v testech. Pod sekcí vytváření objektů je if statement, který určuje pouštěné E2E testy a jejich parametry. Jaké testy se pustí je rozhodnuto na základě proměnné bankCode. Tato proměnná se E2E testům poskytuje jako parametr při spuštění z příkazového řádku. Připraveny jsou testy pro 4 klienty, s tím, že spousta testů je zakomentovaná pro jejich nespolehlivost.

Mezi nezakomentovanými testy je:

- Přihlášení do aplikace.
- Vytvoření nového uživatele a jeho přidání do skupiny uživatelů.
- Změna hesla.
- Vytvoření obecného číselníku.

- Smazání obecného číselníku.
- Vytvoření role.
- Smazání role.
- Otestování filtrů na stránce s uživateli.
- Přidání datové schránky.
- Vytvoření notifikace.
- Smazání notifikace.
- Přidání uživatelské skupiny.
- Upravení uživatelské skupiny.
- Přidání pravidla pro delegování.
- Přidání skupiny zpráv.

Těchto 15 testů autor považoval za základ, na kterém může pracovat a vylepšit ho přepsáním do nového frameworku.

3.3 Architektura starých testů

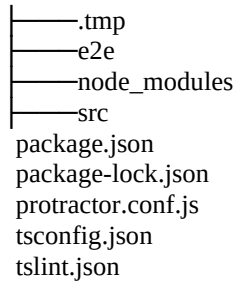
Před psaním nových testů chtěl autor porozumět testům starým a najít v nich inspiraci pro návrhové vzory nových testů a čemu se případně vyhnout. Přítomnost veškerých testů v jednom souboru `uni.app.e2e-spec.ts` není vyhovující. Zajímavý je však systém organizace souborů podle názvů jednotlivých stránek, kde každý takový soubor obsahuje konstanty lokátorů elementů (jako je id) a metody pro získání těchto elementů. Takže pro práci s elementem stačí zavolat metodu příslušné stránky. Tento návrhový vzor usnadňuje práci (je potřeba méně kódu) a omezuje chyby (ví se, z jaké stránky se element volá). Určitým způsobem také usnadňuje orientaci v kódu, jelikož prozrazuje, na jaké stránce se právě prohlížeč má nacházet. Tomuto přístupu se více věnuje Gundecha (2012).

Dalším zajímavým přístupem je složka `utils` se souborem `universal.ts`. Tento soubor obsahuje lokátory elementů, které se vyskytují univerzálně napříč stránkami (hlavička, univerzální tlačítko uložit). Dále obsahuje funkce, které může využít široké spektrum testů, jako například generace náhodného čísla nebo řetězce znaků.

Jako možnost pro zlepšení autor viděl snížení využití chainování promisů pomocí then ve prospěch async/await. Async/await udělá kód mnohem čistší a jednodušší pro čtení a zároveň se i lépe programuje.

3.4 Struktura projektu

Na obrázku (Obrázek 7) je zobrazena jednoduchá adresářová struktura projektu. V dalších odstavcích jsou popsány účely jednotlivých souborů a složek.



Obrázek 7 – Adresářová struktura hlavní složky
Vlastní tvorba

Složka `.tmp` slouží pro uchovávání dočasně vytvářených souborů. Ve složce `src` jsou soubory využívané převážně starými testy. Složka `e2e` obsahuje zdrojový kód ke všem napsaným testům. V kořenovém adresáři jsou ještě podstatné soubory package.json a protractor.conf.js.

```
exports.config = {
    framework: 'custom',
    frameworkPath: require.resolve(
    'protractor-cucumber-framework'),
    specs: [
        './e2e/features/specs/*.feature',
        './e2e/features/specs/*/*.feature',
        './e2e/features/specs/*/*/*.feature'
    ],
    cucumberOpts: {
        require: [
            './e2e/features/steps/*.steps.js',
            './e2e/features/support/hooks.js',
            './e2e/features/steps/*/*.steps.js',
            './e2e/features/steps/*/*/*.steps.js'
        ],
        format: "json:.tmp/results.json",
    },
    capabilities: {
        browserName: "firefox",
        acceptInsecureCerts: true,
        'moz:firefoxOptions': {
            prefs: {
                'browser.download.folderList' : 2,
                'browser.download.dir' :
                 path.join(process.cwd(), "/.tmp"),
'services.sync.prefs.sync.browser.download.useDownloadDir'
            : true,
                'browser.download.useDownloadDir' : true,
             }
        }
    },
    onPrepare() {
        browser.driver.manage().window().setSize(1920, 1080)
        browser.waitForAngularEnabled(false);
        const {Given, Then, When, Before} =
               require('@cucumber/cucumber');
        global.Given = Given;
        global.When = When;
        global.Then = Then;
    },
    onComplete(){
        var reporter = require('cucumber-html-reporter');
        var options = {
            theme: 'bootstrap',
            jsonFile: '.tmp/results.json',
            output: '.tmp/report/index.html',
            reportSuiteAsScenarios: true,
            scenarioTimestamp: true,
            launchReport: false,
  };
        reporter.generate(options);
    }
}
```

Ukázka kódu 7 – Soubor s nastavením Protractoru
Vlastní tvorba

Soubor `protractor.conf.js` (Ukázka kódu 7) obsahuje nastavení Protactoru pro spouštění E2E testů. Definuje se tam používaný framework, v tomto případě custom

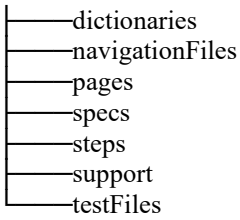
protractor-cucumber-framework. Pod vlastností specs jsou definovány cesty k souborům obsahujícím specifikace testů. V cucumberOpts se nastavují cesty k definicím kroků a k hooks. Dále se specifikuje formát a lokace umístění souboru s reportem ohledně běhu E2E testů. V capabilities se nastavuje využívaný prohlížeč a jeho nastavení. Nastavení pro Firefox umožňují v tomto případě stahování souborů E2E testy. Následuje nastavení rozlišení prohlížeče a definice klíčových slov pro kroky. Níže se pak ještě nastavuje plugin pro vytváření přehlednějších reportů po doběhnutí testů.

```
{
  "scripts": {
    "e2e:dev": "npm run protractor --
                --params.url=http://localhost:4200
                --params.bankCode=5500
                --params.exevidoVersion=DEV
                --cucumberOpts.tags=@dev",
    "e2e:rb": "npm run protractor --
               --params.url=http://localhost:4200
               --params.bankCode=5500
               --params.exevidoVersion=DEV
               --cucumberOpts.tags=@rb",
},
  "private": true,
  "devDependencies": {
    "@cucumber/cucumber": "^7.3.0",
    "chai": "^4.3.0",
    "cucumber-html-reporter": "^5.5.0",
    "geckodriver": "^3.0.1",
    "protractor": "7.0.0",
    "protractor-cucumber-framework": "^8.4.1"
  },
  "engines": {
    "node": ">= 4.2.1",
    "npm": ">= 3"
  },
  "dependencies": {
    "chai-as-promised": "^7.1.1"
  }
}
```

Ukázka kódu 8 – Soubor package.json
Vlastní tvorba

V `package.json` (Ukázka kódu 8) jsou definovány scripty pro rychlé spouštění, jako *e2e:dev:rb*, který spustí v příkazovém řádku příkaz pro spuštění testů pro klienta RB. Dále obsahuje závislosti knihoven. Například knihovna chai umožňuje používání BDD tvrzení a kontrolu jejich správnosti. Geckodriver je ovladač pro spouštění testů přes Firefox. Pak protractor a protractor-cucumber-framework knihovny, které zajišťují běh testů. Ts-node, tslint a typescript jsou knihovny využívané starými testy psanými v typescriptu. V závislostech je definovaná knihovna chai-as-promised umožňující využívání chai tvrzení s JS promises.

3.5 Struktura nových testů



Obrázek 8 – Adresářová struktura složky s novými testy
Vlastní tvorba

Pro nové testy byla vytvořena podsložka `features`. Na obrázku (Obrázek 8) je vidět její adresářová struktura. Složka `dictionaries` obsahuje slovník stránek, kterému bude věnováno více prostoru v samostatné kapitole. Soubory v `navigationFiles` zajišťují funkčnost navigace v menu aplikace. Složka `pages` zaštiťuje soubory s funkcemi pro získávání elementů daných stránek. Ve složce `specs` jsou soubory s definicemi testů a ve složce `steps` soubory s definicemi kroků. Složka support se používá pro soubor s tzv. hooks (funkce pozměňující běh programu). Poslední složka, `testFiles`, obsahuje soubory příloh využívaných novými testy.

3.6 První jednoduchý test

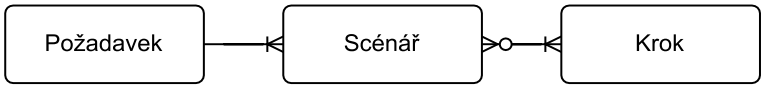
Pro dobré pochopení funkčnosti E2E tesů je tato kapitola věnována popisu jednoduchého testu. V ukázce níže (Ukázka kódu 9) je test, kde se uživatel pokouší přihlásit do aplikace se špatným heslem.

```
# language: cs
Požadavek: Přihlášení do exevida
  @rb @rsts @eqb @penny @moneta @rl @mydsy
  Scénář: Přihlášení uživatele se špatným heslem
    Za předpokladu že jdu na stránku exevida
    Když se přihlásím jako "system" s heslem "admin188"
    Pak přihlášení selže
```

Ukázka kódu 9 – Test na přihlášení napsaný v jazyce Gherkin
Vlastní tvorba

Na prvních řádcích ukázky (Ukázka kódu 9) se definuje jazyk testu a požadavek. Požadavkem je fungování přihlašování do aplikace. Následují jednotlivé scénáře s tagy, pro které klienty se spouští. První scénář se spouští pro všechny klienty.

Na obrázku (Obrázek 9) je zobrazen diagram popisující vztah jednotlivých komponent testu. Každý požadavek má jeden nebo více scénářů. Scénář je použit pouze v jednom požadavku a má jeden nebo více kroků. Krok na druhou stranu může být použit v několika scénářích anebo nemusí být použit vůbec.



Obrázek 9 – ERD vztahu požadavku, scénářů a kroků
Vlastní tvorba

```
let loginPage = require('../pages/login.js');
let universal = require('../pages/universal.js');
var chai = require('chai');
var chaiAsPromised = require('chai-as-promised');
const until = ExpectedConditions;
chai.use(chaiAsPromised);
var expect = chai.expect;
const {element, browser} = require("protractor");

Given("(že )jdu na stránku exevida", {timeout: 90 * 1000},
async function () {
    let currentUrl = await browser.getCurrentUrl();
    if (!currentUrl.includes('auth/login')){
        browser.get(browser.params.url);
    }
    await
expect(browser.getTitle()).to.eventually.equal("Exevido
        Banking " + browser.params.exevidoVersion);
});

When("(se )přihlásím (se )jako {string} s heslem {string}",
{timeout: 70 * 1000}, async function (username, password) {
    await browser.wait(until.elementToBeClickable(
        loginPage.getLoginButton()), 60 * 1000,
        'login - loginBtn');
    await loginPage.getUsername().clear();
    await loginPage.getUsername().sendKeys(username);
    await loginPage.getPassword().clear();
    await loginPage.getPassword().sendKeys(password);
    await loginPage.sendLogin();
});

Then('přihlášení selže', {timeout: 30 * 1000}, async
function () {
    let popup = await element(by.id('divSmallBoxes'))
                    .all(by.tagName('div')).first();
    await browser.wait(until.presenceOf(popup), 20 * 1000,
        'login fail - presence popup');
    await browser.wait(until.textToBePresentInElement(
        popup,'Chyba'), 20 * 1000,
        'login fail - text in popup');
    await expect(popup.getText())
        .to.eventually.contain('Chyba');
});
```

Ukázka kódu 10 – Kroky k testu pro přihlášení
Vlastní tvorba.

První krok „Za předpokladu že jdu na stránku exevida“ je definovaný v ukázce kódu výše (Ukázka kódu 10). Na začátku souboru s kroky jsou definice souborů stránek využívaných pro získávání prvků a proměnných dostupných z knihoven.

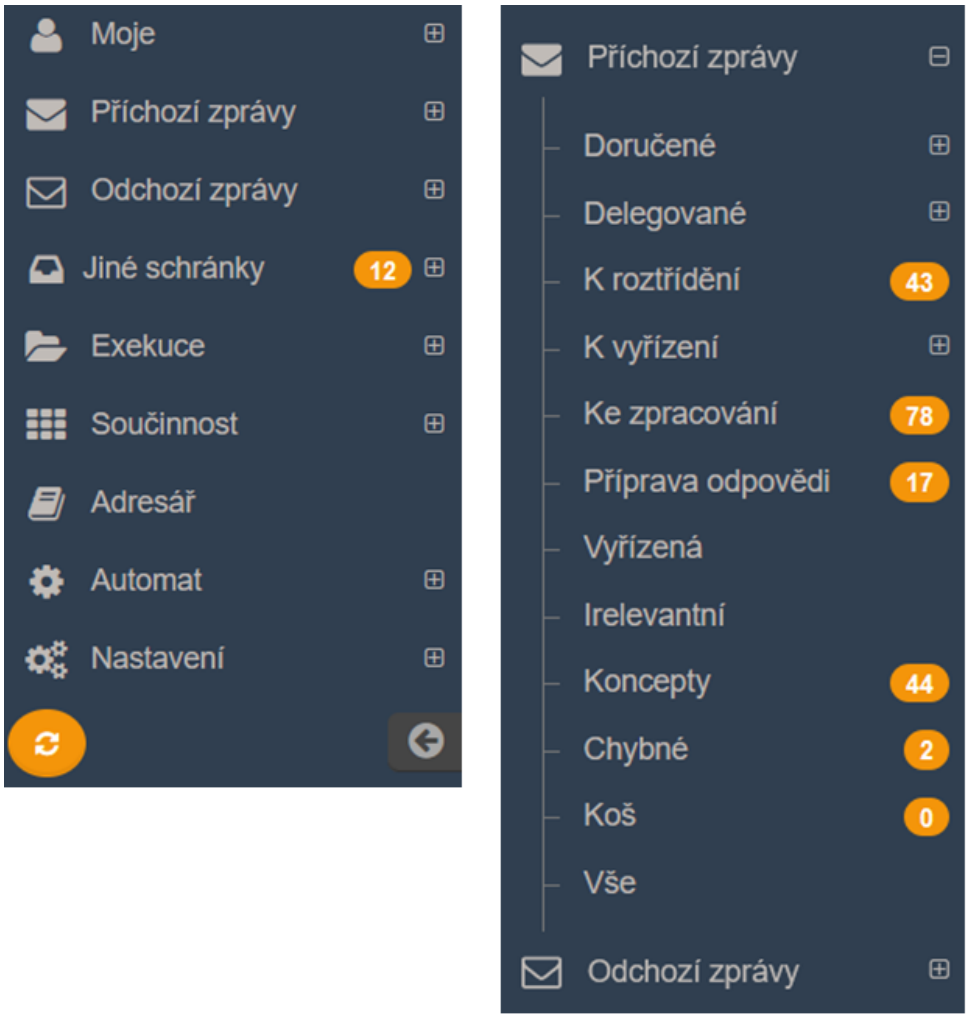
Zápis definice kroku pro přesun na stránku Exevida umožňuje zápis kroku v testu dvěma způsoby. První způsob je „jdu na stránku exevida“ a druhý způsob „že jdu na stránku exevida“. Definice je takto napsaná, aby krok dával smysl při použití jak s frází „Když“, tak „Za předpokladu“. V tomto kroku se kontroluje, zda prohlížeč je přítomný na přihlašovací stránce aplikace. Pokud není, tak na ni přejde. Pak kontroluje, zda sedí titulek stránky s titulkem daným v parametrech při volání E2E testů.

Druhý krok „Když se přihlásím jako "system" s heslem "admin188"“ dostává dva parametry, a to přihlašovací jméno a heslo, se kterými se pokusí uživatele přihlásit. Poslední krok tohoto scénáře „Pak přihlášení selže“ kontroluje, že vyskočilo vyskakovací okno s textem „Chyba“, značící neúspěšné přihlášení.

Na tomto prvním scénáři je vidět, jak v principu fungují Cucumber E2E testy. Framework si spojí text v souboru `login.feature` s definicí kroku v souboru `login.steps.js` a tyto kroky vykonává. Názvy souborů mohou být libovolné, dokud mají správnou koncovku. Kroky mohou být parametrizované pro lepší znovu využitelnost, jako v případě kroku na přihlášení. Pro psaní definic kroků lze využít i regulárních výrazů, ale této možnosti autor nevyužíval.

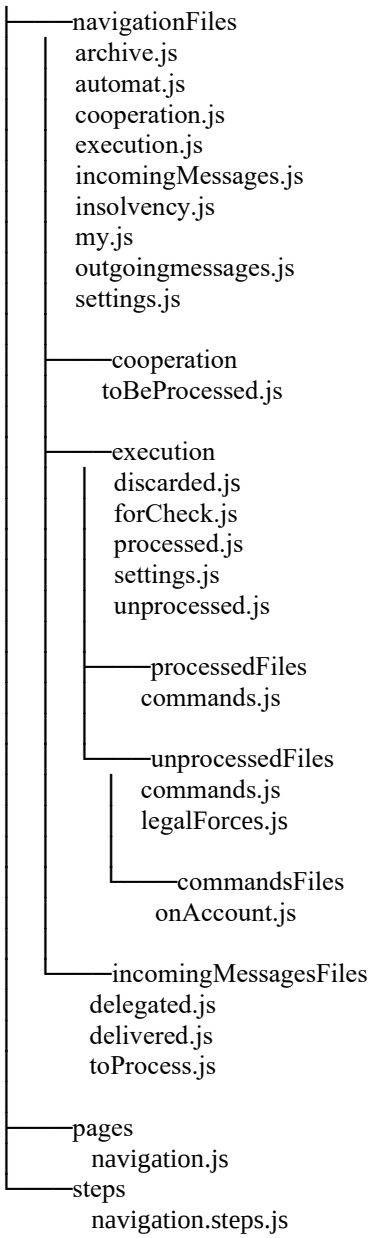
3.7 Design navigace v menu

Pro demonstraci funkčnosti navigování v menu je dobré si představit, že se prohlížeč má dostat na stránku Doručené příchozí zprávy. K této stránce se jde v menu proklikat přes příchozí zprávy a pak doručené, jak je vidět na obrázku níže (Obrázek 10).



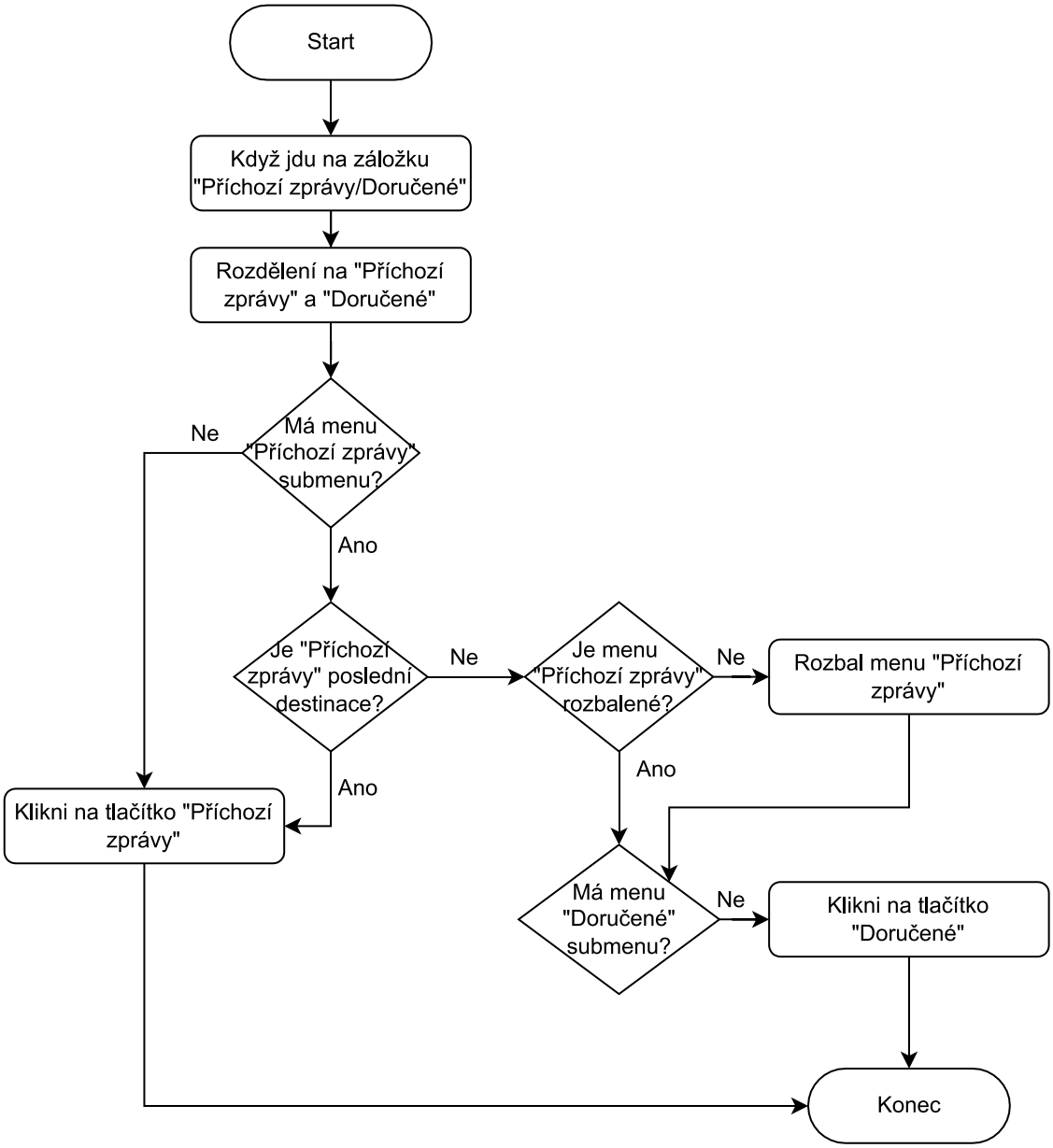
Obrázek 10 – Menu aplikace EXevido
Vlastní tvorba

Na obrázku níže (Obrázek 11) je vyobrazeno, kde se nacházejí všechny soubory potřebné pro fungování navigace v E2E testech.



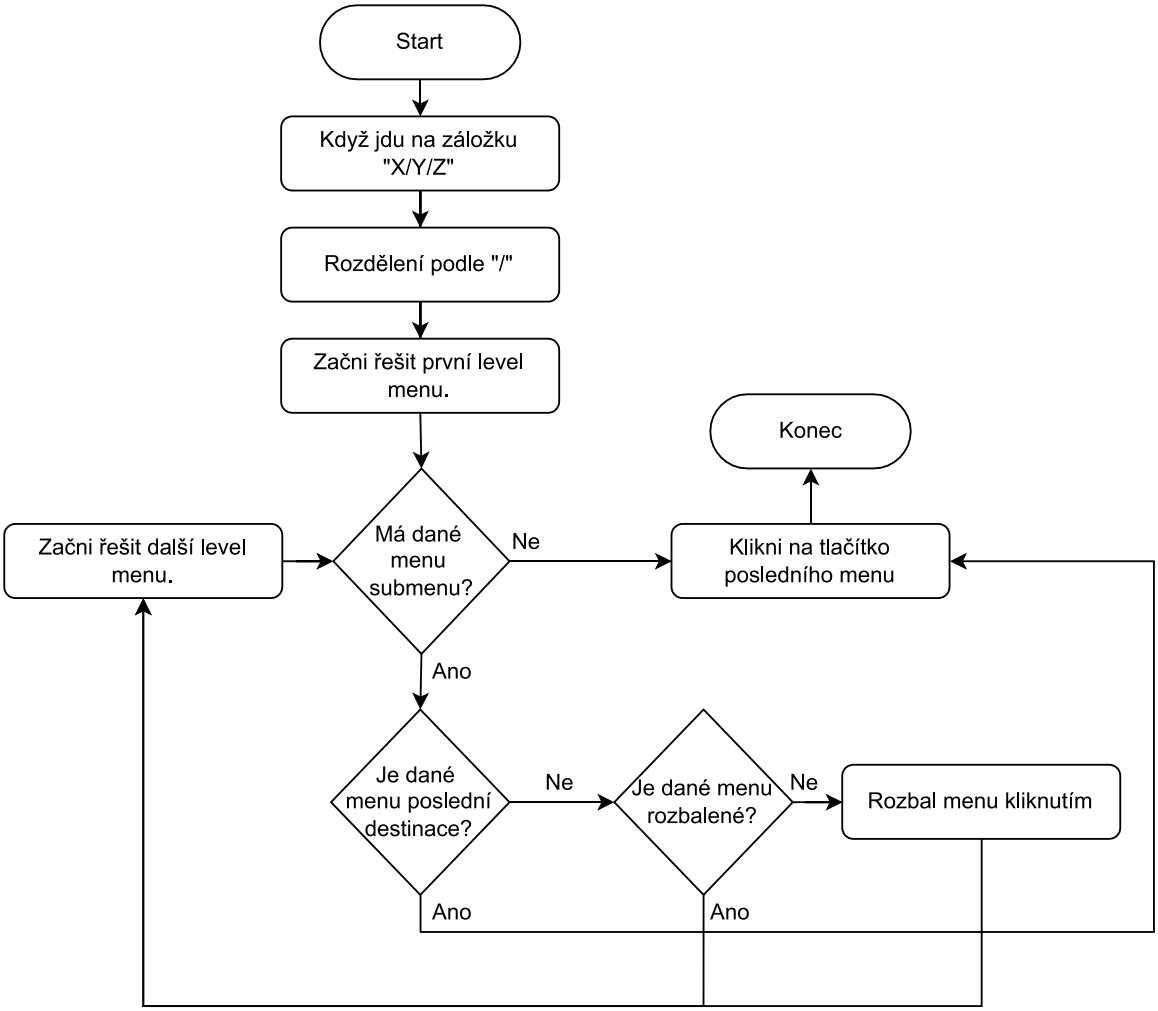
Obrázek 11 – Adresářová struktura se soubory potřebnými k navigaci
Vlastní tvorba

Pro navigaci na stránku Doručené příchozí zprávy se v E2E testech použije krok „Když jdu na záložku "Příchozí zprávy/Doručené"“. Na obrázku níže (Obrázek 12) je vidět, jak postupuje program v případě, že má přejít na Doručené příchozí zprávy. Z diagramu byla záměrně vynechána větev, která nastane v případě, že menu „Doručené“ má submenu.



Obrázek 12 – Vývojový diagram pro navigaci na Doručené příchozí zprávy
Vlastní tvorba

Na dalším vývojovém diagramu (Obrázek 13) je znázorněno, jak program postupuje obecně. Z tohoto diagramu vyplývá, že postup je podobný while cyklu, který pokračuje, dokud se program nedostal k finální destinaci v menu.



Obrázek 13 – Vývojový diagram vysvětlující obecné fungování navigace v testech
Vlastní tvorba

Definice kroku „Když jdu na záložku…“ zkrácená o většinu možností switch statementu je zobrazena v ukázce kódu níže (Ukázka kódu 11).

```
When('jdu na záložku {string}', {timeout: 100 * 1000},
async function (string) {
    await browser.wait(until.stalenessOf(
          universal.getLoadingCheck()
          ), 20 * 1000, 'navigation steps - loadingCheck');
    let arr = string.split("/");
    let buttonToClick;
    let button;
    let page;
    switch (arr[0]) {
        case "Moje":
            button = navigationPage.getMyButton();
            page = stepMy;
            break;

        case "Příchozí zprávy":
            button = navigationPage.getIncomingMessagesButton();
            page = incomingPage;
            break;

        case "Odchozí zprávy":
            button = navigationPage.getOutgoingMessageButton();
            page = stepOutgoingMessages;
            break;

        default:
            assert.fail(`'Položka ${arr[0]}' neexistuje nebo
             není v záznamech.`);
    }
    if (button) {
        buttonToClick = await navigationPage.navigationClick(
                                button, page, arr, 0);
    }
    await navigationPage.clickOnButtonToClick(
          buttonToClick, arr[0]);
    await browser.wait(until.stalenessOf(
          universal.getLoadingCheck()
          ), 40 * 1000, 'navigation - loadingCheck');
})
```

Ukázka kódu 11 – Krok pro navigaci testech
Vlastní tvorba

Tento krok funguje univerzálně pro navigaci celým menu aplikace. Jako parametr dostává cestu k výsledné lokaci, kde znak lomítka odděluje podsložky navigace. V univerzálním kroku se nejdříve rozdělí cesta na řetězce podle lomítek. Pak se snaží najít ve switch statementu první level navigace, na který má přejít.

Princip je takový, že existují tři proměnné *buttonToClick*, *button* a *page*. Do *buttonToClick* se ukládá tlačítko v případě, že pod sebou nemá žádné další menu. Do tlačítka *button* se ukládá tlačítko, pokud pod sebou má další vrstvy menu a zároveň s tím se uloží odkaz na modelový soubor stránky do *page*.

Pod switch statementem je if větev. Pokud má menu nějaká submenu, zavolá se metoda navigationClick. Tato metoda je v ukázce kódu pod tímto odstavcem (Ukázka kódu 12).

```
async navigationClick(ele, page, arr, index) {
    let btn = null;
    if (arr[index + 1] !== undefined) {
        await browser.wait(until.elementToBeClickable(ele),
          30 * 1000, 'navigationClick - navigation is clickable');
        let bool = await page.isClicked();
        await browser.executeScript(
              'arguments[0].scrollIntoView(false)', ele);
        if (!bool){
            await ele.click();
        }
        await page.navigationStep(arr);
    } else {
        btn = ele;
    }
    return btn;
}
```

Ukázka kódu 12 – Metoda navigationClick pro posun na další level navigace
Vlastní tvorba

Metoda navigationClick kontroluje, jestli se program dostal do finální destinace (jestli je další prvek pole undefined, pak chce uživatel skončit zde). Pokud je poslední, tak pouze vrátí tlačítko, které tato funkce dostala, aby se na něj kliklo. V opačném případě zavolá metodu souboru stránky, kterou dostala jako parametr, isClicked. Tuto metodu má každá stránka navigace. Metoda vrací boolean, jestli je submenu dané položky rozbalené.

Další řádek zaručí, že se na tlačítko „nascrolluje“. Protractor řídí scrollování automaticky, jenže občas má problém s hlavičkou, která může některé elementy překrývat. Pokud není submenu rozbalené, rozbalí ho a dojde k zavolání metody stránky navigationStep. V další ukázce kódu (Ukázka kódu 13) jsou metody isClicked a navigationStep. Většina větví switch statementu byla záměrně vynechána.

```
    isClicked(){
        return navigationPage.getIncomingDeliveredButton()
                .isDisplayed();
    },

    async navigationStep(arr) {
        let buttonToClick;
        let button;
        let page;
        switch (arr[menuLevel]) {
            case "Doručené":
                button = navigationPage
                         .getIncomingDeliveredButton();
                page = deliveredPage
                break;

            case "Delegované":
                button = navigationPage.getIncomingDelegated();
                page = delegatedPage
                break;

            case "K roztřídění":
                buttonToClick = navigationPage.getIncomingToSort();
                break;

            default:
                assert.fail(`Položka '${arr[menuLevel]}' neexistuje
                 nebo není v záznamech.`);
        }
        if (button) {
            buttonToClick = await navigationPage.navigationClick(
                                button, page, arr, menuLevel);
        }
        await navigationPage.clickOnButtonToClick(buttonToClick,
              arr[menuLevel]);
    }
```

Ukázka kódu 13 – Metody pro navigaci v submenu
Vlastní tvorba

Jak je vidět na vývojových diagramech (Obrázek 12 a Obrázek 13), tato metoda velmi podobná definici samotného kroku. Poslední nerozebranou metodou je clickOnButtonToClick v následující ukázce (Ukázka kódu 14). Tato metoda pouze kontroluje, zda je na co kliknout a pokud ano, tak na to klikne. Čili probíhá kontrola, zda tato proměnná není null, než se na ní prohlížeč pokusí kliknout.

```
    async clickOnButtonToClick(buttonToClick, lastPath) {
        if (buttonToClick) {
            await browser.wait(until.elementToBeClickable(
                  buttonToClick), 30 * 1000,
                  `button '${lastPath}' is not clickable`);
            await buttonToClick.click();
        }
    }
```

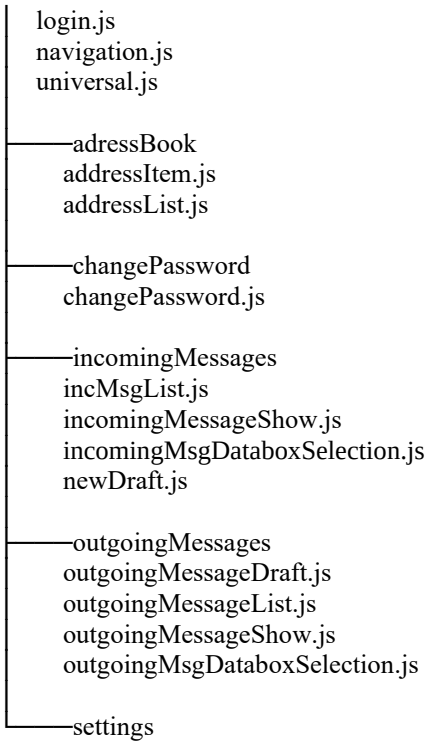
Ukázka kódu 14 – Metoda pro kliknutí na tlačítko navigace
Vlastní tvorba

3.8 Využití slovníku stránek

Ve složce `pages` jsou jednotlivé soubory rozdělené do složek podle jejich polohy v menu aplikace. Tato adresářová struktura je vidět na obrázku níže (Obrázek 14) s vynechanými podsložkami `settings`. Většina souborů této složky je mapovaná slovníkem stránek. Například ve složce `incomingMessages` jsou čtyři soubory:

- `incMsgList.js` obsahuje prvky nacházející se na stránce s listem příchozích zpráv
- `newDraft.js` obsahuje prvky na stránce, kde se zakládá nová příchozí zpráva
- `incomingMessageShow.js` obsahuje prvky na stránce vytvořené příchozí zprávy
- `incomingMsgDataboxSelection.js` obsahuje prvky v modálním oknu, kde se vybírá datová schránka v případě, že uživatel chce založit příchozí zprávu bez vybrané datové schránky

Obdobné vzory pojmenovávání jsou použity i v ostatních složkách.



Obrázek 14 – Adresářová struktura složky s modelovými soubory stránek
Vlastní tvorba

Důležitost slovníku stránek se postupně v projektu rozrůstala a autor věří, že možnosti jeho využití se budou nadále rozšiřovat. Velmi malá část souboru je dostupná v ukázce kódu níže (Ukázka kódu 15).

Ukázka kódu 15 – Soubor slovníku stránek

Vlastní tvorba

Soubor začíná deklaracemi proměnných se soubory ve složce `pages`. Následně je v module.exports proměnná *pagesDictionary*. Proměnná je v module.exports, aby se k ní dalo přistupovat z jiných souborů. Tento slovník uchovává jako klíč pomyslnou cestu ke stránce z menu aplikace a jako hodnotu odkaz na modelový soubor stránky. Například pokud je v testu potřeba přistoupit k prvku na listu příchozích zpráv, může být přes slovník získán odpovídající soubor, toto ovšem není důvod existence tohoto slovníku (a ani tímto způsobem není v kódu používán).

Řešení pomocí slovníku umožňuje vytvořit univerzální kroky, které zabraňují nutnosti vytvářet stejné kroky, lišící se pouze jménem a jednou proměnnou.

```
When('(jsem se )dostal( jsem se) na {string}', {timeout: 100 *
1000},
async function (pageName){
    let page = dictionaries.pagesDictionary[pageName];
    await page.waitForLoad();
    let isOnPage = await page.isOnPage();
    await expect(isOnPage).to.ok;
});

When("kliknu na tlačítko {string} na stránce {string}",
{timeout: 100 * 1000}, async function (buttonName, pageName){
    let page = dictionaries.pagesDictionary[pageName];
    let button = element(by.id(
                  page.buttonDictionary[buttonName]));
    await browser.wait(until.presenceOf(button), 40 * 1000,
          'button present');
    await browser.wait(until.elementToBeClickable(button),
          20 * 1000, 'button clickable');
    await button.click();
});
```

Ukázka kódu 16 – Kroky využívající slovník stránek
Vlastní tvorba

V ukázce kódu výše (Ukázka kódu 16) jsou kroky pro kontrolu, že se prohlížeč dostal na požadovanou stránku a pro kliknutí na specifické tlačítko. Oba kroky využívají slovník stránek.

Krok pro kontrolu, zda se prohlížeč dostal na požadovanou stránku volá metody modelového souboru stránky waitForload a isOnPage, jejich implementace v souboru `incomingMessageShow.js` je vidět v další ukázce kódu (Ukázka kódu 17).

```
    buttonDictionary:  {
        'Přidat pozitivní subjekt k ověření':
         LOCATOR_ID_ADD_COOPERATION_SUBJECT,
        'Zpracovat a zobrazit odpověď':
         LOCATOR_ID_PROCESS_AND_SHOW_ANSWER_BUTTON,
        'Přidat komentář': LOCATOR_ID_LOGGING_HISTORY_COMMENT_BTN,
        'Odpověď': LOCATOR_ID_ANSWER,
        'Delegovat': LOCATOR_ID_DELEGATE_SOLVER_BTN,
        'Změnit typ zprávy': LOCATOR_ID_CHANGE_MESSAGE_TYPE_BTN,
        'Nastavit datum': LOCATOR_ID_SET_DATE_BTN,
        'Zpracovat vybrané přílohy ve vlastní zprávě':
         LOCATOR_ID_ATTACHMENT_PROCESS_BTN
    },

    async isOnPage(){
        return (await this.getAnswerButton().isDisplayed());
    },

    async waitForLoad(){
        await browser.wait(until.elementToBeClickable(
              this.getAnswerButton()), 40 * 1000,
              'incomingMessages/show - answer button');
        await browser.wait(until.stalenessOf(
              universal.getLoadingCheck()), 40 * 1000,
              'incomingMessages/show - loading check');
    }
```

Ukázka kódu 17 – Soubor modelu stránky příchozí zprávy
Vlastní tvorba

Metoda isOnPage vrací boolean, zda je prohlížeč přítomný na správné stránce. Metoda waitForLoad čeká, až se daná stránka načte. V tomto případě může působit metoda isOnPage jako zbytečná, jelikož se kontroluje, zda je zobrazeno tlačítko „Odpověď“, na které se čeká v metodě waitForLoad. Pokud tedy prohlížeč není na správné stránce, pak krok neprojde dříve, než se zavolá metoda isOnPage. V jiných případech, jako na stránce outgoingMessageShow.js v ukázce kódu níže (Ukázka kódu 18), metoda isOnPage opravdu poskytuje podrobnější kontrolu.

```
    async isOnPage() {
        let headerText = await this.getPageHeader().getText();
        return (headerText.includes('Odchozí zpráva') &&
                !headerText.includes('Koncept'));
    },

    async waitForLoad() {
        await browser.wait(until.elementToBeClickable(
              this.getCopyButton()), 20 * 1000,
              'outgoingMessages/show - waitForLoad');
    }
```

Ukázka kódu 18 – Soubor modelu stránky vytvořené odchozí zprávy
Vlastní tvorba

Tlačítko „Vytvořit kopii“ se nachází jak na stránce už vytvořené odchozí zprávy, tak při vytváření odchozí zprávy. Je tedy potřeba druhá kontrola přes text v nadpisu, zda neobsahuje text „Koncept“.

Krok pro kliknutí na tlačítko přijímá jako parametr text tlačítka a cestu ke stránce. V Ukázka kódu 17 je vidět slovník *buttonDictionary*, kde jako klíč slouží text tlačítka a hodnotou je id tlačítka. Přes získané id může prohlížeč následně kliknout na správné tlačítko.

3.9 Problémy při vývoji E2E testů

Vývoj E2E testů se neobešel bez problémů. Nejednalo se pouze o jednoduché problémy, ale hlavně o problémy způsobené interakcí stroje se stránkou, která na to není připravena. V praxi to znamená, že některé věci fungují občas jinak, než by měly, a to pouze v případě E2E testů. Krokem, na němž to lze snadno demonstrovat je „vyplním do filtru ID Zprávy ID vytvořené zprávy“ v ukázce pod tímto odstavcem (Ukázka kódu 19).

```
When("vyplním do filtru ID Zprávy ID vytvořené zprávy",
{timeout: 140 * 1000}, async function () {
    await browser.wait(until.stalenessOf(
          universal.getLoadingCheck()
          ), 20 * 1000, 'fill ID filter - loadingCheck 1');
    await browser.wait(until.elementToBeClickable(
          incomingMsgList.getCreateIncomingMessageButton()),
          20 * 1000, 'create new message to be clickable 1');
    await incomingMsgList.getMessageIdFilter().clear();
    await browser.wait(until.elementToBeClickable(
          incomingMsgList.getCreateIncomingMessageButton()),
          20 * 1000, 'create new message to be clickable 2');
    await incomingMsgList.getMessageIdFilter().sendKeys(
          incomingMsgList.messageId + '\n');
    await browser.wait(until.elementToBeClickable(
          incomingMsgList.getCreateIncomingMessageButton()),
          20 * 1000, 'create new message to be clickable 3');
    // check if first row mesasge has matching ID
    await browser.wait(until.textToBePresentInElement(
        incomingMsgList.getFirstRow(),incomingMsgList.messageId),
        10 * 1000).catch(async function () {
          if (await incomingMsgList.getMessageIdFilter()
              .getAttribute('value') !==
              incomingMsgList.messageId) {
            await incomingMsgList.getMessageIdFilter().clear();
            await incomingMsgList.getMessageIdFilter().sendKeys(
                  incomingMsgList.messageId + '\n');
            await browser.wait(until.elementToBeClickable(
           incomingMsgList.getCreateIncomingMessageButton()),
            20 * 1000, 'create new message to be clickable 4');
            await browser.wait(until.textToBePresentInElement(
                  incomingMsgList.getFirstRow(),
                  incomingMsgList.messageId), 10 * 1000,
                  `first row is not message with id:
                  ${incomingMsgList.messageId}`)
          }else {
            await assert.fail(`first row is not message with id:
                  ${incomingMsgList.messageId}`)
          }
    });
});
```

Ukázka kódu 19 – Krok pro vyplnění filtru ID příchozí zprávy
Vlastní tvorba

Při vytváření příchozí zprávy testem, se uloží ID vytvořené zprávy do proměnné *messageId* v souboru `pages/incomingMessages/incMsgList.js`. V tomto kroku se pomocí uloženého ID vyfiltruje založená zpráva. Nejdříve prohlížeč čeká až zmizí kolečko načítání a půjde kliknout na tlačítko, které vytváří novou příchozí zprávu. Jedna z oprav, kterou autor přidal do aplikace byla, aby se na tlačítko pro vytvoření nové příchozí zprávy nedalo kliknout, dokud frontend nezíská data z backendu, jelikož kolečko načítání toto neodráží. Následně dojde k vyčištění pole, kam se ID zadává pro případ, že by se už někdy filtrovalo a znovu se musí čekat na tlačítko. V původní implementaci kroku se po vyčištění na tlačítko

nečekalo a způsobovalo to nevyfiltrovaní zprávy podle ID, jelikož frontend občas reagoval pouze na první požadavek na filtrování, a to ten s prázdným filtrem.

Pak je do filtru vyplněno ID zprávy a opět se počká na tlačítko. Když došlo k vyfiltrování, tak se čeká až bude první vyfiltrovaná zpráva (a tedy jediná vyfiltrovaná zpráva) mít správné ID. Občas z filtru zmizí vyplněné ID a nedojde k vyfiltrování. Pokud se to stane, krok znovu vyfiltruje tuto zprávu.

Krok i přes již prodělané úpravy někdy neprojde. Je to způsobené chybou, kterou při filtrování vyhodí backend. Několik těchto chyb autor v aplikaci již opravil, ale jelikož se s nimi obyčejný uživatel nesetká (jsou způsobeny příliš rychlými akcemi), nejsou na seznamu prioritních chyb. V E2E testech tento krok kvůli podobné chybě neprojde zřídka a autor si myslí, že je to tak v pořádku. Test prochází dostatečně často na to, aby nebyl považován za vadný, ale zároveň neprojde, pokud backend vyhodí výjimku (i když je způsobená rychlostí E2E testů).

Podobně je to i u dalších kroků. Často je chyba způsobena přílišnou rychlostí vykonávání pokynů E2E testy a takovéto chyby se opravují velmi složitě. V jiných případech jsou problémy způsobeny rozdíly mezi tím, jak využívá aplikaci uživatel a jak stroj. Jako praktický příklad bude uvedeno zadávání textu. Uživatel do pole nejdříve klikne a následně zadá text. Stroj do pole pouze zadá text (jestli ho vypisuje rychlými „stisky“ kláves nebo pouze vloží se autorovi zjistit nepodařilo). Tato drobnost způsobuje problémy například při opakovaném vyplňování totožného pole.

3.10 Hákové funkce

Velmi důležité pro běh E2E testů jsou už zmíněné tzv. hooks (hákové funkce) v souboru `support/hooks.js`. Tyto funkce jsou spouštěny podle jejich nastavení. Možnosti jejich spouštění jsou následující:

- před nebo po všech testech
- před nebo po každém testu
- před nebo po každém kroku

Jejich spouštění se dá nastavit ještě pomocí tagů a dalších proměnných dostupných z testů nebo kroků. To znamená, že háková funkce spouštěná po každém kroku může být nastavena tak, aby se spouštěla po každém kroku pouze v testech s určitým tagem a pouze pokud daný krok neprojde.

```
let universal = require('../pages/universal.js');
const {browser} = require("protractor");
let fs = require('fs');
const TIMESTAMP_FILE_PATH = '.tmp/timestamps.txt';

After({timeout: 40 * 1000}, async function (scenario) {
    //scenario result status for video description
    fs.appendFile(TIMESTAMP_FILE_PATH,
    scenario.result.status + '\n',
    function (err) {
        if (err) throw err;
    });

    //hook to logout current user
    await universal.logoutFully();
});

AfterStep({tags: 'not @screenshot and not @dev'}, async function
({result}) {
    if (result.status === 'FAILED') {
        let screenshot = await browser.takeScreenshot();
        this.attach(screenshot, 'image/png');
    }
});

AfterStep({tags: '@screenshot or @dev'}, async function () {
    let screenshot = await browser.takeScreenshot();
    this.attach(screenshot, 'image/png');
});

Before(function (scenario) {
    fs.appendFile(TIMESTAMP_FILE_PATH, Date.now() + '|' +
    scenario.pickle.name + '|', function (err) {
        if (err) throw err;
    });
});

BeforeAll(function () {
    fs.unlink(TIMESTAMP_FILE_PATH, function (err) {
        if (err != null && err.code !== 'ENOENT') throw err;
    })
});
```

Ukázka kódu 20 – Hákové funkce používané E2E testy
Vlastní tvorba

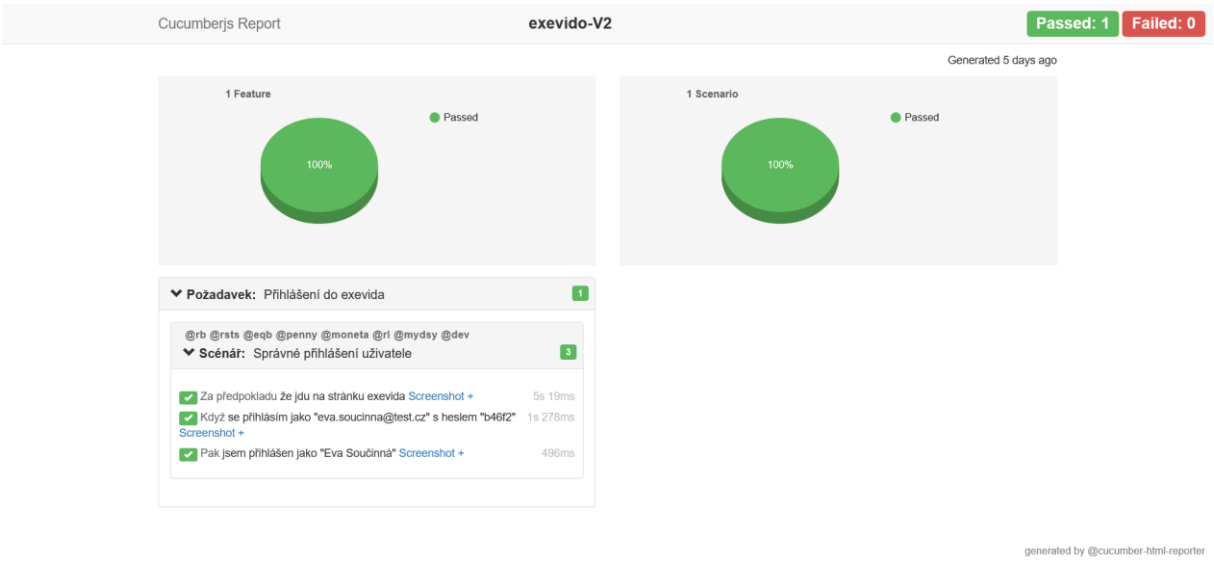
Na předcházející ukázce kódu (Ukázka kódu 20) je vidět soubor hooks.js obsahující veškeré hákové funkce využívané v projektu. První funkce se spouští po každém testu. Její účel byl prvotně pouze odhlášení. Po přidání nahrávání videí z běhu testů na serveru byla tato funkce rozšířena. Momentálně i zapisuje do textového souboru výsledek testu, s tím, že předposlední funkce před každým testem uloží do souboru čas a název testu. Soubor s výsledky se nemůže mazat po doběhnutí testů, jelikož pak by z něj nešel vygenerovat popis s časovými značkami začátku testů. Proto dochází ke smazání souboru v poslední hákové funkci, která se spouští před všemi testy. Aby nedošlo k chybě, když soubor ještě nebyl

vytvořený, tak funkce vyhodí chybu pouze v případě, že se nejedná o chybu s kódem 'ENOENT', značící neexistující soubor.

Hákové funkce AfterStep se provádí po každém kroku a jsou využívány pro pořizování snímků obrazovky. První z těchto funkcí pořizuje snímek pouze v případě, že krok neprošel. Druhá funkce pořizuje snímek obrazovky pouze u testů s tagem @dev nebo @screenshot.

3.11 Ladění E2E testů

Právě hákové funkce zmíněné v kapitole výše umožňují ladění E2E testů psaných v Protractoru. Nejde o nic přepychového, ale snímky obrazovky jsou občas klíčové k rozpoznání, kde nastala chyba. Po každém běhu E2E testů se pomocí knihovny cucumber-html-reporter vygeneruje report obsahující pořízené obrázky. Mimo obrázků také zobrazuje, jak dlouho jednotlivé kroky trvaly (viz Obrázek 15).



Obrázek 15 – Vygenerovaný report z běhu E2E testů
Vlastní tvorba

I když není mnoho s čím lze při ladění E2E testů pracovat, je tento proces velmi nápomocný. Chybové hlášky sice prozradí, proč test spadnul, ale už neprozradí nic o tom, co spadnutí předcházelo. Proto je dobré mít možnost podívat se na snímek obrazovky daného kroku anebo všech kroků předtím.

3.12 Zhodnocení přínosu řešení

Nové E2E testy za dobu fungování odhalily již nespočet chyb, které se dostaly do vývojové větve. Při brzkém odhalení chyby je lehčí ji opravit, než pokud se najde později a jsou na ni navázány další funkce. I přes to však E2E testy slouží hlavně ke kontrole funkčnosti při vydávání nové verze aplikace.

Optimalizace testů přepsáním do jiného frameworku ušetřila dost času, jelikož testy běží mnohem rychleji a nemusí se pouštět vícekrát díky minimalizaci falešných pádů. Odhalení chyb před vytvářením verze a kontrolou testerem zabraňuje zbytečným opožděním v dodávání nových verzí zákazníkům.

3.13 Ekonomické zhodnocení

Z ekonomického hlediska znamená dřívější odhalení chyby její levnější opravení. Na druhou stranu každé spuštění E2E testů znamená nutnost nasadit aplikaci na server a pustit testy tam, což není zdarma. Pro plynulé fungování E2E testů bylo potřeba rozšířit infrastrukturu a zajistit pronájem dalších serverů. V některých případech nefunguje nasazování a běh na serverech tak, jak má a je potřeba vnější zásah systémovým administrátorem. I přes tyto všechny nepříjemnosti, které zvyšují cenu běhu E2E testů, se jejich pouštění vyplatí.

E2E testy přidávají další záchranou vrstvu, na kterou se může vývojář spolehnout při provádění změn v aplikaci a nemusí následně ručně provádět kontrolu, zda aplikaci nerozbil. Mění se tím peníze vynaložené na psaní a údržbu testů za peníze vynaložené na vyvíjení nových funkcí aplikace. Dokud údržba a psaní testů stojí méně než platba vývojářům za čas, který by strávili testováním, podílejí se E2E testy na vytváření zisku.

Dříve odchycené chyby a tím i funkční SW udržují dobré jméno společnosti a loajalitu jejich zákazníků.

Závěr

Cílem této bakalářské práce bylo navrhnout optimalizované řešení E2E testů pro testování webové aplikace. Autor takové řešení navrhl a uvedl do provozu s tím, že se zbavil chyb předchozích testů, testy udělal srozumitelnější a přístupnější všem členům agilního týmu.

Největším přínosem této bakalářské práce je optimalizace testů a tím zefektivnění procesu vývoje webové aplikace. Dále dochází ke shrnutí dvou velmi důležitých aspektů profesionálního programování – agilních přístupů řízení SW projektů a jejich testování.

Jako největší problém pro další optimalizace testů autor shledává nutnost obcházení chyb aplikace, které se projevují pouze při E2E testech. Dalším problémem je složitost odhalování závad projevujících se pouze za specifických podmínek. Testy probíhají na stejném prostředí a zároveň se ovlivňují, takže i pouhá změna pořadí spouštění testů může ovlivnit jejich funkčnost.

Pro nejlepší poměr účinnost/náklady autor doporučuje testovat pomocí E2E testů pouze klíčové aspekty aplikace a zbytek testovat pomocí unitových nebo integračních testů. Díky rozšíření původního souboru 15 testů na 56 funkčních testů, jejichž provedení trvá kolem 15 minut, jsou základy webové aplikace EXevido otestované. Dobu provedení 15 minut považuje autor za přijatelnou a předpokládá, že doba provedení do 20 minut neovlivní ochotu vývojářů testy pouštět.

Na druhou stranu autor chápe možnost využití E2E testů jako prostředku pro zadávání a kontrolu vyhotovení nových funkcí aplikace. Společnosti, které se rozhodnou jít touto cestou se musí připravit na velké množství práce vydané na údržbu E2E testů.

E2E testy jsou v dobrých rukou silným pomocníkem při testování webových aplikací. Díky tomu, že je každý může využívat v míře, která mu vyhovuje, se hodí pro testování všech větších webových aplikací. Při použití správného frameworku a dobrém základu mohou testy psát jak programátoři, tak i lidé bez programovacích znalostí, což otevírá firmám možnosti na tuto práci využít kohokoliv správně motivovaného.

Citace

AMODEO, Enrique, 2015. Learning behavior-driven development with JavaScript: create powerful yet simple-to-code BDD test suites in JavaScript using the most popular tools in the community. Birmingham: Packt Publ. Community experience distilled. ISBN 978-1-78439-264-2.

ATER, Tal, 2017. Building progressive web apps: bringing the power of native to the browser. First edition. Sebastopol, CA: O'Reilly Media. ISBN 978-1-4919-6165-0.

BECK, Kent, Mike BEEDLE, Arie van BENNEKUM, Alistair COCKBURN, Ward CUNNINGHAM, Martin FOWLER, James GRENNING, Jim HIGHSMITH, Andrew HUNT, Ron JEFFRIES, Jon KERN, Brian MARICK, Robert C. MARTIN, Steve MELLOR, Ken SCHWABER, Jeff SUTHERLAND a Dave THOMAS, 2001. Manifest Agilního vývoje software [online] [vid. 2022-01-12]. Dostupné z: https://agilemanifesto.org/iso/cs/manifesto.html

DIGITAL.AI, 2021. *15th Annual State Of Agile Report | Digital.ai* [online] [vid. 2022-01-19]. Dostupné z: https://digital.ai/resource-center/analyst-reports/state-of-agile-report

DOLEZEL, Michal, Alena BUCHALCEVOVA a Michal MENCIK, 2019. The State of Agile Software Development in the Czech Republic: Preliminary Findings Indicate the Dominance of "Abridged" Scrum. In: Petr DOUCEK, Josef BASL, A Min TJOA, Maria RAFFAI, Antonin PAVLICEK a Katrin DETTER, ed. Research and Practical Issues of Enterprise Information Systems [online]. Cham: Springer International Publishing, Lecture Notes in Business Information Processing, s. 43–54 [vid. 2022-01-19]. ISBN 978-3-030-37631-4. Dostupné z: doi:10.1007/978-3-030-37632-1_4

GORDON, Elyse Kolker, 2018. Isomorphic web applications: universal development with React. Shelter Island, New York: Manning. ISBN 978-1-61729-439-6.

GUDE, Sharath Chowdary, 2012. JavaScript: the used parts. In: the 3rd annual conference: Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity - SPLASH '12 [online]. Tucson, Arizona, USA: ACM Press, s. 109 [vid. 2022-02-23]. ISBN 978-1-4503-1563-0. Dostupné z: doi:10.1145/2384716.2384762

GUNDECHA, Unmesh, 2012. Selenium testing tools cookbook: over 90 recipes to build, maintain, and improve test automation with Selenium Webdriver. Birmingham Mumbai: Packt Publ. Quick answers to common problems. ISBN 978-1-84951-574-0.

JOHANSEN, Christian, 2011. Test-driven JavaScript development. Upper Saddle River, NJ: Addison-Wesley. Developer's library series. ISBN 978-0-321-68391-5.

KACZANOWSKI, Tomek, 2013. Practical unit testing with JUnit and Mockito. Leipzig: CreateSpace. ISBN 978-83-934893-9-8.

KHORIKOV, Vladimir, 2020. Unit testing: principles, practices, and patterns. Shelter Island, NY: Manning. ISBN 978-1-61729-627-7.

KOSKELA, Lasse, 2013. Effective unit testing: a guide for Java developers. Shelter Island, NY: Manning. ISBN 978-1-935182-57-3.

LIAU, Keen Yee, 2021. Future of Angular E2E & Plans for Protractor · Issue #5502 · angular/protractor. GitHub [online] [vid. 2022-02-16]. Dostupné z: https://github.com/angular/protractor/issues/5502

LUCHANINOV, Yuriy, 2021. Web Application Architecture in 2021: Moving in the Right Direction. Mobidev [online]. Dostupné z: https://mobidev.biz/blog/web-application-architecture-types

MEDNIEKS, Zigurd, 2021. The Agile Idea. Shelter Island, NY: Manning Publications. ISBN 978-1-61729-931-5.

MIKOWSKI, Michael S. a Josh C. POWELL, 2014. Single page web applications: JavaScript end-to-end. Shelter Island, NY: Manning. ISBN 978-1-61729-075-6.

ROSE, Seb, Matt WYNNE a Aslak HELLESØY, 2015. The cucumber for Java book: behaviour-driven development for testers and developers. Dallas, Texas: The Pragmatic Bookshelf. The pragmatic programmers. ISBN 978-1-941222-29-4.

SALAMEH, Hanadi, 2014. What, When, Why, and How? A Comparison between Agile Project Management and Traditional Project Management Methods.

SHORE, James, Diana LARSEN, Gitte KLITGAARD a Shane WARDEN, 2022. The art of agile development. ISBN 978-1-4920-8069-5.

SMART, John Ferguson, 2015. BDD in action: Behavior-Driven Development for the whole software lifecycle. Shelter Island, NY: Manning Publications. ISBN 978-1-61729-165-4.

STELLMAN, Andrew a Jennifer GREENE, 2014. Learning Agile. First edition. Beijing: O'Reilly. ISBN 978-1-4493-3192-4.

ŠOCHOVÁ, Zuzana a Eduard KUNCE, 2014. Agilní metody řízení projektů. Brno: Computer Press. ISBN 978-80-251-4194-6.

TOWNSEND, Dave, 2020. Remove the SSB feature. In: bugzilla.mozzila.org [online]. Dostupné z: https://bugzilla.mozilla.org/show_bug.cgi?id=1682593

WEST, Dave, 2011. Water-Scrum-Fall Is The Reality Of Agile For Most Organizations Today. 17.

WIRFS-BROCK, Allen a Brendan EICH, 2020. JavaScript: the first 20 years. Proceedings of the ACM on Programming Languages [online]. 4(HOPL), 1–189. ISSN 2475-1421. Dostupné z: doi:10.1145/3386327

YOUNG, Alex a Marc HARTER, 2015. Node.js in practice: includes 115 techniques. Shelter Island: Manning. ISBN 978-1-61729-093-0.