Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačního inženýrství

Diplomová práce

Návrh a implementace automatizovaných testů pro webovou aplikaci

František Křivanec

© 2022 ČZU v Praze

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ DIPLOMOVÉ PRÁCE

Bc. František Křivanec

Systémové inženýrství a informatika
Informatika

Název práce

Návrh a implementace automatizovaných testů pro webovou aplikaci

Název anglicky

Design and implementation of automated tests for a web application

---

Cíle práce

Hlavním cílem diplomové práce je návrh a implementace sady automatizovaných testů napsaných v jazyce Java pro kontrolu požadovaného chování webové aplikace. Testy jsou spouštěny za pomoci nástroje Selenium.
Vedlejším cílem práce je objasnění teoretických principů návrhu testovacího scénáře a analýza současného stavu testování softwaru.
Součástí práce je otestování funkčnosti vytvořené sady testů a prodiskutování jejího přínosu.

Metodika

Práce začíná literární rešerší zaměřenou na danou problematiku. Teoretické znalosti jsou získávány studiem odborné literatury zaměřené na téma testování softwaru z knih i internetových zdrojů. Informace nabrané z těchto zdrojů jsou zachyceny v teoretické části práce.

V praktické části je nejprve vytvořen a ozkoušen testovací scénář pro manuální testování, a poté je na stejnou problematiku vytvořen automatizovaný test. Sada automatizovaných testů je poté spuštěna na webové aplikaci a výsledky testů jsou ověřovány.

Testy jsou napsány v jazyce Java a je využito nástroje Selenium, díky němuž je možné nasimulovat průchod aplikací.

Doporučený rozsah práce

60-80 stran

Klíčová slova

Automatizované testování, Java, webová aplikace, testovací scénář, testování webové aplikace, selenium

---

Doporučené zdroje informací

BUREŠ, Miroslav, Miroslav RENDA, Michal DOLEŽEL, Peter SVOBODA, Zdeněk GRÖSSL, Martin KOMÁREK, Ondřej MACEK a Radoslav MLYNÁŘ. Efektivní testování softwaru: klíčové otázky pro efektivitu testovacího procesu. Praha: Grada, 2016. Profesionál. ISBN 978-80-247-5594-6.

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

HAVELKA, Arnošt a Rudolf PECINOVSKÝ. JUnit 5: jednotkové testování na platformě Java. Praha: Grada Publishing, 2018. Knihovna programátora (Grada). ISBN 978-80-271-0733-9.

ROUDENSKÝ, Petr. Kvalita softwaru: teorie a praxe. Prostějov: Computer Media, 2016. ISBN 978-80-7402-294-4.

Předběžný termín obhajoby

2022/23 ZS – PEF

Vedoucí práce

Ing. Marek Pícka, Ph.D.

Garantující pracoviště

Katedra informačního inženýrství

Elektronicky schváleno dne 4. 11. 2022	Elektronicky schváleno dne 28. 11. 2022
Ing. Martin Pelikán, Ph.D.	**doc. Ing. Tomáš Šubrt, Ph.D.**
Vedoucí katedry	Děkan

V Praze dne 29. 11. 2022

Čestné prohlášení

Prohlašuji, že svou diplomovou práci "Návrh a implementace automatizovaných testů pro webovou aplikaci" jsem vypracoval(a) samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu použitých zdrojů na konci práce. Jako autor(ka) uvedené diplomové práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne 30. 10. 2022 ______________________________

Poděkování

Rád bych touto cestou poděkoval Ing. Marku Píckovi, Ph.D., za odborné vedení při tvorbě diplomové práce. Bez jeho rad a připomínek by dokončení diplomové práce bylo mnohem složitější.

Návrh a implementace automatizovaných testů pro webovou aplikaci

Abstrakt

Tato práce se zabývá návrhem a následné implementace sady automatických testů k otestování firemní webové aplikace. Testy jsou psány v jazyku Java a spouštěny za pomoci nástroje Selenium webdriver. V teoretické části jsou vysvětleny termíny z oblasti testování softwaru, popsány životní cykli vývoje, metody a principy testování a obeznámení s nástroji používanými pro automatizované testování.

V praktické části je přiblížen současný stav testování ve firmě. Hlavní téma praktické části je vytvoření automatického testu. Je zde tedy vytvořena analýza, co se má přesně otestovat, na základě čehož je udělán testovací scénář, dle které je napsán kód automatického testu. Nejdůležitější části kódu jsou v práci popsány. V závěru praktické části je nastíněno monitorování proběhlých testů, a celá práce je zakončena shrnutím.

Klíčová slova: automatizované testování, Java, webová aplikace, testovací scénář, testování webové aplikace, Selenium, IPO, software

Design and implementation of automated tests for a web application

Abstract

This thesis deals with the design and subsequent implementation of a set of automatic tests to test a company's web application. Tests are written in Java and run using Selenium webdriver. In the theoretical part, terms from the field of software testing are explained, development life cycles, methods and principles of testing and familiarization with tools used for automated testing are described.

The practical part describes the current state of testing in the company. The main topic of the partical part is the creation of an automatic test. An analysis of what exactly needs to be teste dis created here, based on which a test scenario is created, according to which the automatic test code is written. The most important parts od the code are described in the thesis. At the end of the practical part, the moniroting of the completed test sis outlined, and the entire work is concluded with a summary.

Keywords: automated tests, Java, web application, test case, web application testing, Selenium, IPO, software

Obsah

1 Úvod ... 12

2 Cíl práce a metodika ... 14

2.1 Cíl práce ... 14

2.2 Metodika ... 14

3 Teoretická východiska ... 15

3.1 Kvalita softwaru ... 16

3.2 Pojmy a jejich definice ... 18

3.2.1 Přesnost a správnost ... 18

3.2.2 Verifikace a validace ... 20

3.2.3 Kvalita a spolehlivost ... 20

3.2.4 Testovací scénář ... 20

3.2.5 Chyba ... 20

3.2.6 Závažnost ... 21

3.2.7 Priorita ... 21

3.2.8 Incident ... 22

3.3 Životní cyklus systému ... 22

3.3.1 Fáze životního cyklu ... 22

3.3.1.1 Analýza ... 22

3.3.1.2 Návrh ... 23

3.3.1.3 Implementace ... 23

3.3.1.4 Testování ... 23

3.3.1.5 Zahájení provozu ... 23

3.3.1.6 Údržba a provoz ... 24

3.3.2 Tradiční vývojové modely ... 24

3.3.2.1 Vodopádový model ... 24

3.3.2.2 Spirálový model ... 25

3.3.2.3 RUP ... 27

3.3.3 Agilní vývoj ... 28

3.3.3.1 Scrum ... 29

3.4 Metody a principy testování ... 30

3.4.1 Testování programátorem ... 30

3.4.2 Testování jednotek ... 30

3.4.3 Funkční a nefunkční testy ... 31

3.4.4 Integrační testy 31
3.4.5 Systémové testování 31
3.4.6 Akceptační testování 32
3.4.7 Testování černé a bílé skříňky 32
3.4.8 Statické a dynamické testování 32
3.4.9 Testování specifikace 32
3.4.10 Testování softwaru bez znalosti kódu 34
3.4.11 Zkoumání programového kódu 35
3.4.12 Dynamické testování bíle skříňky 37
3.4.13 Manuální testování 37
3.4.14 Automatizované testování 37
3.5 JAVA 40
3.5.1 specifické rysy jazyka 40
3.5.2 Objektově orientované programování 41
3.6 Selenium 41
3.6.1 Selenium IDE 42
3.6.2 Selenium RC 43
3.6.3 Selenium WebDriver 43
3.6.4 Selenium Grid 44
3.7 Lokalizování prvků webu pomocí XPath 44
3.8 Nástroje usnadňující práci s kódem 46
3.8.1 IDE 46
3.8.1.1 NetBeans 46
3.8.1.2 IntelliJ IDEA 47
3.8.1.3 Eclipse 47
3.8.2 Git 48
3.9 Současný stav testování softwaru 49
4 Vlastní práce 50
4.1 Situace automatizovaného testování softwaru v Antee 50
4.2 Představení IPO 50
4.2.1 IPOAdmin 51
4.2.2 IPO 52
4.2.3 Role v IPO 52
4.3 Identifikace testu 53
4.4 Analýza testu 54
4.5 Testovací scénář 58
4.6 Vytvoření testu 60

4.6.1 Vytvoření nové větve projektu 60
4.6.2 Struktura projektu 61
4.6.2.1 Třída NewsIpoDetail 63
4.6.2.2 Třída PostNewsIpoTest 65
4.6.2.3 Test přihlášení 68
4.6.2.4 Test vytvoření novinky 68
4.6.2.5 Test změny konceptu novinky 72
4.6.2.6 Test publikování novinky 72
4.6.2.7 Test kontroly publikované novinky 73
4.6.2.8 Test změny publikované novinky 74
4.6.2.9 Test smazání novinky 74
4.6.2.10 Test smazání novinky z administrace 75
4.7 Monitorování testu 75
4.7.1 Chyba v buildu 57 77
4.7.2 Chyba v buildu 68 77
5 Výsledky a diskuse 79
6 Závěr 81
7 Seznam použitých zdrojů 82
8 Seznam obrázků 85
Přílohy 86

1 Úvod

V dnešní době je většině světa běžné každodenní setkání se s elektronickými zařízeními a jejich používání. Tato zařízení využívají pro svůj běh jednodušší a, nebo složitější software. V návaznosti na to, k čemu je zařízení využíváno, rostou úměrně i nároky na bezchybnou funkčnost tohoto softwaru. Z toho důvodu je potřeba software před uvedením mezi lidi důkladně otestovat, aby se předešlo případným nehodám.

Čím déle jsou chyby v softwaru odhaleny, tím složitější a zároveň i finančně náročnější je jejich odstranění. Chyba, která se vyskytuje v softwaru po jeho uvedení na trh může mít nedozírné následky. Kromě zhoršení reputace firmy a obrovských nákladech spojených se stažením produktu z trhu a přepracováním, může u některých produktů dojít i k ohrožení zdraví lidí.

Testování softwaru může být v celkové hodně projektu nákladné a především zdlouhavé. Ať už se jedná o manuální testování, nebo přípravu automatických testů. Nicméně s každou včas odhalenou chybou se investice vyplácí a šetří peníze, které by bylo potřeba v budoucnu na opravu stejně vynaložit.

Manuální testy jsou časově náročné a pro testera mohou být po čase únavné čímž se může snížit jeho pozornost. Na rozdíl od automatizovaných testů může tester na případnou chybu přímo reagovat. Zautomatizováním opakujících se testů je ušetřeno spoustu času pro testera, který se může věnovat dalším úkolům, a odstranění možné chybovosti v testech z důvodů nepozornosti.

V praktické části práce je popsán test vytvoření „Novinky IPO“ v aplikaci IPOAdmin. Jedná se o nově přidanou funkcionalitu do této aplikace. Test je ve firmě spouštěn každý den spolu s dalšími automatickými testy pro tuto aplikaci.

U automatického testu je potřeba, aby fungoval spolehlivě. Pokud by pravidelně vykazoval chyby, které nemají racionální vysvětlení, byl by tento test zbytečný. Proto je potřeba testy monitorovat, a pokud začnou vykazovat určitou chybovost i přesto, že aplikace funguje, jak má je potřeba je předělat.

Teoretická část práce se zaměřuje na seznámení čtenáře s testováním softwaru. Jsou zde popsaný důležité termíny z této problematiky, vysvětlené nejběžnější vývojové postupy a představeny softwary a nástroje používané pro usnadnění vytváření automatizovaných testů.

Praktická část se v krátkosti zabývá současným stavem testování softwaru. Poté je představena aplikace IPOAdmin jejíž otestování je tématem práce. Dále je popsán prováděný test a vysvětlení jednotlivých funkcí kódu. Na závěr praktické části je ukázáno, jak několik posledních testů aplikace probíhalo, a jsou vysvětleny případné chyby v proběhlých testech.

2 Cíl práce a metodika

2.1 Cíl práce

Hlavním cílem diplomové práce je návrh a následná realizace sady automatizovaných testů pro kontrolu správného chování webové aplikace. K napsání testovacího skriptu je využit objektově orientovaný programovací jazyk Java. Ke spouštění testů je použito nástroje Selenium webdriver.

Vedlejším cílem práce je seznámení s teoretickými principy pro vytváření testovacích scénářů, seznámení se současnou situací a otestování funkčnosti vytvořené testovací sady.

V závěru práce je vyhodnocen přínos a funkčnost testovací sady a navržení možného zlepšení.

2.2 Metodika

První část práce je zaměřena na literární rešerši pro danou problematiku. Teoretické znalosti jsou vyňaty z odborné literatury, která se zabývá dohled na kvalitu vytvářeného softwaru z knih a internetových zdrojů. Informace nabrané během studia uvedených pramenů jsou shrnuty do teoretické části práce.

V praktické části je v práci stručně shrnut současný stav celkového testování a automatizovaného testování softwaru. Poté je představena aplikace, pro kterou je test vytvářen, a konkrétní funkce aplikace které se bude test týkat. Na základě požadovaného chování je sestaven testovací scénář v několika krocích, a následně napsán test v objektově orientovaném jazyce Java, který poběží za pomoci nástroje Selenium webdriver. V práci jsou vysvětleny nejdůležitější metody kódu. Dále je ukázáno monitorování několika posledních buildů testu a vysvětlení případných chyb v buildech.

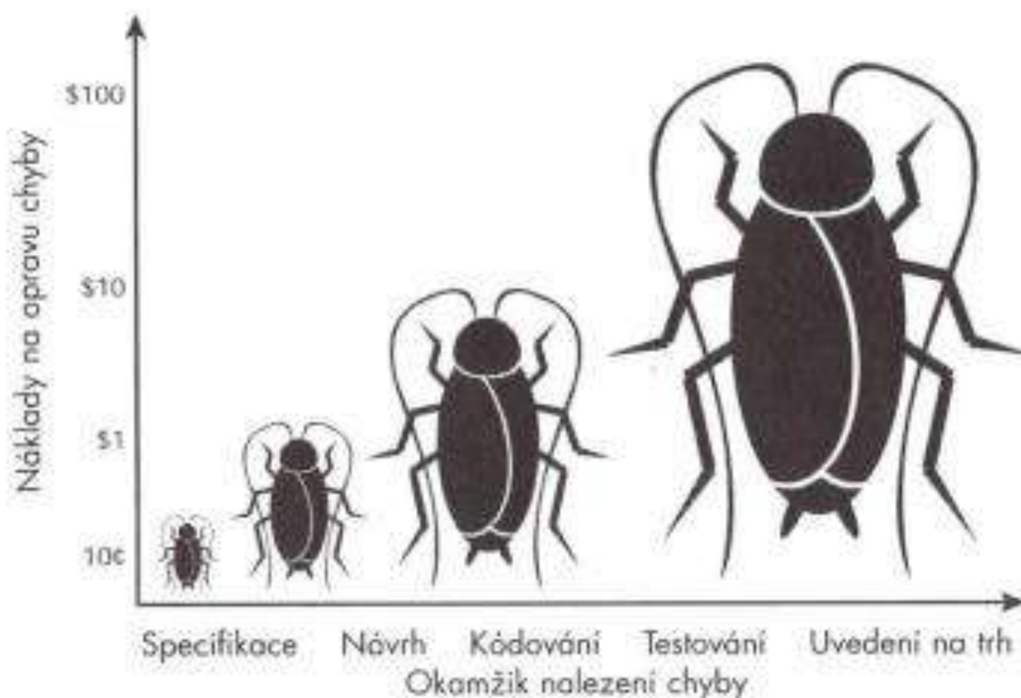
3 Teoretická východiska

Lidé se v dnešní době se softwarem v jeho nejrůznějších podobách setkávají v podstatě každý den. Může se jednat o jednoduchý software v digitálních hodinkách, přes software v telefonech a počítačích, až po software, který je zodpovědný za řízení letecké dopravy nebo správné fungování lékařských přístrojů. V závislosti na přístrojích, na kterých se mohou chyby vyskytnout se může jednat někdy i chybu s nedozírnými následky. Pokud by například svítil displej na hodinkách trošku jinou barvou, než měl. Někteří lidé by to například nemuseli ani považovat za chybu, jako třeba kdyby se trojka zobrazovala obráceně. Pokud by se ale objevila závažnější chyba v leteckých systémech nebo softwaru lékařských přístrojů, již by taková chyba s sebou přinesla velké finanční ztráty v lepším případě, v tom horším by ale mohla stát i lidské životy. [1]

Průběh testování není jen hledání chyb v daném softwaru, ale jde i o to, zanalyzovat si průběh testování, vytvořit si potřebná data pro testování, a především správně prezentovat výsledky testů. [1]

Firmy, které jsou ochotny testovat kvalitně i za cenu vyšších nákladů na takové investici nakonec výrazně ušetří. Platí totiž pravidlo, že čím dříve je ve fázi vývoje chyba odhalena, tím levněji vychází její oprava. S každou další fází vývoje, do které se chyba dostane z předchozí bude cena na její odstranění několikanásobně vyšší. Pokud je například špatně napsaná specifikace a takováto chyba je odhalena před tím, než začne programátor daný software psát podle chybné specifikace, k jejímu opravení nám postačí pouze čas vynaložený na přepsání dané definice. Pokud se ale chyba objeví až v okamžiku, kdy je software napsaný, bude potřeba aby i programátor vynaložil čas k nalezení chybné části kódu a jeho úpravě dle nové specifikace. Pokud je software uveden na trh i s chybou, která se poté projeví její oprava již bude několikanásobně dražší, a může i poškodit dobré jméno firmy. Na obrázku 1 lze vidět, jak cena chyby postupně roste. [1]

Obrázek 1 Cena chyby



Zdroj [1]

3.1 Kvalita softwaru

Se stále se zvyšující složitostí a nákladností systémů, se logicky zvyšují i požadavky na jejich kvalitu. Kvalitu softwarového produktu je však možné chápat více způsoby. Analyzujme si tedy kvalitu z pěti různých pohledů [2]:

- Cenového - Kvalita je zde taková, kolik je zákazník ochoten za ni zaplatit.
- Produktu - Z tohoto pohledu se kvalita váže na vnitřní kvality produktu, které určují jeho vnější kvality.
- Výroby - Výrobek je tím kvalitnější, čím více splňuje specifikace.
- Uživatele - Z tohoto pohledu vyplývá, že produkt je vhodný pro zamýšlené využití. Splňuje tedy požadavky a potřeby uživatele.
- Transcendentálního - Kvalita je v tomto pohledu něco, co lze rozpoznat, ale je složité ji definovat. Dá se tedy říct, že kvalitu produktu poznáme, nedokážeme ale zdůvodnit proč nám to tak přijde.

Jelikož bylo potřeba jednoznačně vymezit kvalitu softwaru a zvolit k tomu vhodné ukazatele vzniklo hned několik nadnárodních norem, zejména ISO/IEC 9126, ISO/IEC 14598 a ISO/IEC 12119. Jedná se však o nekonzistentní a zastaralé normy, a z toho důvodu je postupně vytlačují normy ISO/IEC 25000-25099, které spadají pod projekt SQuaRe. Zkratku můžeme do češtiny volně přeložit jako Požadavky na kvalitu softwaru a její hodnocení. [2]

Dle normy ISO/IEC 25010 se kvalitou softwaru rozumí míra, do jaké softwarový produkt splňuje určené a implicitní potřeby, je-li používán za daných podmínek. Dle normy je vystavěn model kvality užití a produktu. Tato kvalita je složena nově namísto z dřívějších šesti charakteristik i z dvou nových [2]:

- Funkčnost - Do jaké míry výrobek zabezpečuje funkce, které splňují stanovené a předpokládané potřeby, je-li přípravek používán za stanovených podmíněk.
- Účinnost - Výkonnost ve vztahu k výši prostředků používaných za stanovených podmínek.
- Kompatibilita (nová) - Do jaké míry dva systémy nebo jejich součásti mohou vyměňovat informace, a nebo vykonávat své požadované funkce při sdílení stejného hardwaru nebo softwaru.
- Použitelnost - Do jaké míry výrobek může být používán určenými uživateli k dosažení stanovených cílů účinně, efektivně a uspokojivě ve stanoveném kontextu používání.
- Bezporuchovost - Do jaké míry systém nebo jeho součást plní stanovené úkoly v rámci stanovených podmínek na určitou dobu.
- Bezpečnost (nová) - Do jaké míry jsou informace a data chráněna tak, aby nedošlo k neoprávněné modifikaci nebo přístupu neautorizovanou osobou nebo systémem, a oprávněným osobám nebo systému není odepřen přístup k nim.
- Udržovatelnost - Stupeň účinnosti a efektivity, s nimiž může být produkt modifikován.

- Přenositelnost - Do jaké míry systém nebo jeho součást může být účinně a efektivně převedena z jednoho hardware, software nebo jiného prostředí do druhého.

Tyto charakteristiky se se dále dělí ještě na podcharakteristiky, které jsou předmětem měření, a jednou z těchto měr je testování. Díky testování jsme tedy schopni získat informace o kvalitě produktu. [2]

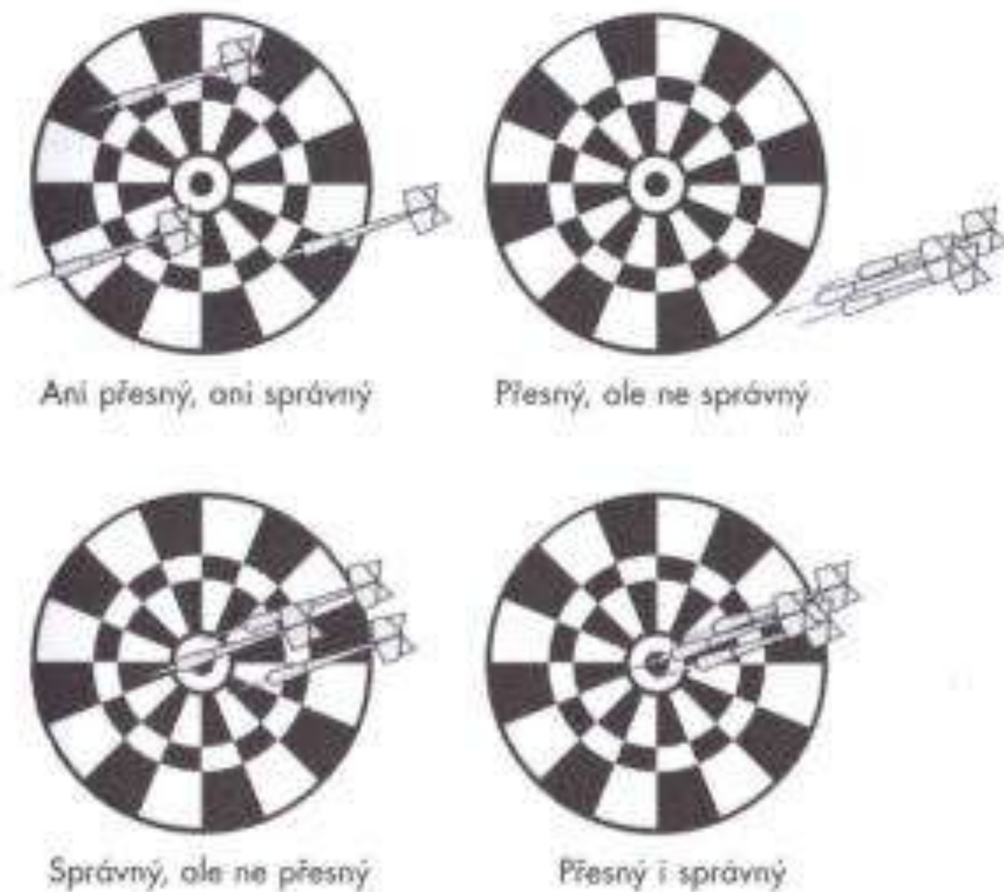
3.2 Pojmy a jejich definice

Pojmy uvedené v této kapitole popisují základní principy, vztahující se k vývoji a testování softwaru. Některé pojmy se totiž snadno chápou jako synonyma, případně se za sebe zaměňují. Zde tedy jsou vysvětleny jejich rozdíly.[1]

3.2.1 Přesnost a správnost

Je velmi důležité být si vědom rozdílu mezi pojmem přesnost a správnost. V určitých situacích, kdy se termín projektu blíží je třeba rozhodnout, které testování bude prioritní. Potom záleží na konkrétním projektu, jaké kritérium bude prioritní. Rozdíl mezi přesností a správností si lze zobrazit na obrázku.

Obrázek 2Rozdíl mezi přesností a správností



Zdroj [1]

Cílem hráče šipek je trefovat střed terče. Obrázek vlevo nahoře není ani přesný ani správný. Šipky jsou každá někde jinde a ani netrefily střed.

Obrázek vpravo nahoře má trefy sice přesné, neboť šipky padali do stejného místa, ale již není správný, jelikož netrefily terč.

Vlevo dole jsou hráčovi pokusy správné, neboť trefují terč blízko k středu, již ale nejsou přesné, neboť mají velký rozptyl.

Vpravo dole je hráč již přesný i správný, neboť všechny šipky zasáhly střed a jsou blízko sebe.[1]

3.2.2 Verifikace a validace

Ačkoliv se dají pojmy verifikace a validace do češtiny přeložit stejným slovem (ověřování nebo kontrola) jejich význam je ve skutečnosti odlišný a pro testování velmi důležitý.

Během verifikace jde o to, aby v našem případě software odpovídal zadané specifikaci.

U validace se však kontroluje, zda software vyhovuje požadavkům uživatele.[1]

3.2.3 Kvalita a spolehlivost

Kvalitou se rozumí že výsledný produkt dokáže uspokojit potřeby zákazníka v několika různých směrech z nichž jedním směrem je i spolehlivost. Spolehlivostí se rozumí, že produkt bude stabilní nebude havarovat a nebude v něm docházet ke ztrátě dat.[1]

3.2.4 Testovací scénář

Pod testovacím scénářem si lze představit dokument, který popisuje postup, jak by se mělo projít programem, a kontroluje se zda software splňuje požadované funkce. Jeden testovací scénář může obsahovat několik testovacích případů a je potřeba, aby testy pokryli chování aplikace z pohledu koncového uživatele. Test je úspěšný, pokud se program chová tak, jak je dle testovacího scénáře očekáváno. [3]

3.2.5 Chyba

Chyba vzniká vždy lidskou chybou. Nejčastěji je to již během specifikace softwaru. Dále chyby vznikají především během návrhu a programování zdrojového kódu. Pro přesnou definici chyby lze tedy použít vztah chování programu vzhledem k jeho specifikaci. Ve specifikaci by mělo být popsáno co má a co nemá daný program dělat. Za chybu tedy lze považovat vše co odpovídá alespoň jednomu z následujících pravidel.[1]

- Software nedělá něco, co by podle specifikace produktu dělat měl.
- Software dělá něco, co by podle údajů specifikace produktu dělat neměl.

- Software dělá něco, o čem se produktová specifikace nezmiňuje.
- Software nedělá něco, o čem se produktová specifikace nezmiňuje, ale měla by se zmiňovat.
- Software je obtížně srozumitelný, těžko se s ním pracuje, je pomalý, nebo podle názoru testera softwaru jej koncový uživatel nebude považovat za správný.

3.2.6 Závažnost

Závažnost neboli severita je pojem, kterým se rozumý jak moc velký dopad bude mít chyba na fungování systému. Čím větší vliv taková chyba má tím bude vyšší i její závažnost. Úroveň závažnosti chyby by měl většinou určit technik kvality. Jak se bude závažnost určovat by mělo být dáno již před začátkem projektu. Každý projekt může mít jiná kritéria pro hodnocení závažnosti ve směs jsou ale čtyři hlavní.[4]

1. Kritická - program s takovouto chybou přestane úplně pracovat. S takovouto chybou není možné pokračovat ve vývoji.
2. Vysoká - takováto chyba je velmi závažná. Způsobuje kolabování systému, některé části však stále mohou dále fungovat.
3. Střední - systém je stále funkční i když se místy objevuje nežádoucí chování.
4. Nízká - systém se nehroutí a nadále funguje s menšími nepřesnostmi.

3.2.7 Priorita

Prioritou se rozumí pořadí, ve kterém by se chyby měli opravovat. S vyšší prioritou je potřeba závady dříve odstraňovat. Pokud se v systému objeví chyba, která zapříčiní nepoužitelnost softwaru bude mít potřeba jejího odstranění vyšší prioritu než závada, která pouze ovlivní část softwaru. [4]

1. Vysoká - závadu je potřeba odstranit v nejkratší možné době, na funkci systému se významně projevuje a nebude možné ho použít dokud nebude odstraněna.
2. Střední - chyba by se měla stihnout odstranit během normálního vývoje softwaru. Lze počkat až do vytvoření nové verze
3. Nízká - jedná se o takové vady systému, s jejichž opravou se může počkat až do okamžiku, kdy budou závažnější vady odstraněny.

3.2.8 Incident

Tímto termínem je označena situace, pokud se program nechová tak, jak by se od něj očekávalo. V takové situaci je třeba problémem nadále zabývat a zjistit, zda se jedná o chybu, nebo jen špatnému porozumění specifikaci. Příčiny incidentu mohou být.[5]

- Nesprávná specifikace a v důsledku toho nesprávné očekávání ze strany testera.
- Chyba v testovacím prostředí.
- Konfigurační chyba.
- Chyba testera.
- Opravdová chyba v testovaném softwaru.

3.3 Životní cyklus systému

U složitých softwarových projektů je nevyhnutelné členění na dílčí jasné a přehledné fáze. Ty se poté dají ilustrovat v praktických příkladech. V začátku je vymodelován uživatelský proces a následný reengineering. Děje se tomu z důvodů, že nový software zasahuje do mechanismů řídících struktur. Dalším krokem je vytvoření konceptuálních a softwarových modelů což vyústí v softwarovou implementaci. [6]

3.3.1 Fáze životního cyklu

Životní cyklus systému je posloupnost etap, která u daného systému vede od chvíle kdy byl zadán požadavek přes vývoj, včlenění do provozu, údržbu a končí až s ukončením provozu. Během zadání je dobré vycházet ze stávajícího stavu a postupně definovat požadované chování systému. [6]

3.3.1.1 Analýza

Analýza je problém z podnikatelského prostředí. Během této etapy se vymezuje problém a zjišťují procesy, které se v systému budou odehrávat. Je třeba rozebrat samotný problém a seznámit se s procesy v systému. Postupně se dostáváme z AS-IS (fáze, jak to je) k TO-BE (jak to má být). To někdy může vyvolat i značné změny procesu a organizace prostředí, ve kterém se má celý systém implementovat. [6]

3.3.1.2 Návrh

Návrh se dá označit za první fázi, během které je snaha o samotné sestavení systému, k tomu, aby bylo možné implementovat software. Na rozdíl od analýzy už je v tuto chvíli vše ze zadání hotovo a rozpoznáno. Dále se rozhoduje, který programovací jazyk, databáze apod. má pro implementaci nejlepší vlastnosti. Též se zjišťuje problematika okolo opakované použitelnosti softwarových artefaktů a napojení systému na komponenty celku. [6]

3.3.1.3 Implementace

Během implementace se programuje samotný program. Tím se rozumí, že vše, co bylo do této doby navrhnuto, naplánováno a odsouhlaseno převedu na požadovaný software. Je třeba, aby byla vedena aktuální dokumentace, ta je nezbytná do budoucna, pokud bude software následně rozšiřován a, nebo udržován. [6]

3.3.1.4 Testování

Během testování softwaru jde o nalezení všech chyb, které se při implementaci v programu vyskytli. V podstatě se při něm kontroluje, zda výsledný program odpovídá tomu, jak byl navrhnut a nepostrádá žádné funkce, které měly být jeho součástí. [7]

Pokud je během testování nelezen nějaký nedostatek, vrací se daný program zpět do fáze implementace, aby se objevený problém opravil. Poté co je oprava hotová je nutné provádět testování znovu, aby se zjistilo, zda nedošlo k dalším chybám během opravy.

3.3.1.5 Zahájení provozu

Pokud je aplikace po otestování shledána bezchybnou, je možné uvést ji do provozu. To znamená, že je nahrána na nějaký server, odkud již bude dostupná pomocí internetu či intranetu. Ačkoliv je aplikace otestována a shledána funkční. Měla by být ještě po nějakou dobu sledována, neboť všechny chyby nemusely být nalezeny a opraveny.

3.3.1.6 Údržba a provoz

Poslední etapou života aplikace před ukončením jejího provozu je provoz a s tím související údržba. S touto etapou je spojena velká finanční a časová náročnost. Odpovídá 100% - 400% náročnosti od počáteční fáze po samotnou implementaci aplikace. [6]

3.3.2 Tradiční vývojové modely

Existuje celá řada modelů životního cyklu softwaru, které popisují, jak na sebe jednotlivé fáze životního cyklu navazují a jaké mají mezi sebou vztahy. Modely používají vlastní metodiky pro to, aby zaručili kvalitu produktu, kterými se od sebe liší. Je však důležité si v tomto kontextu uvědomit rozdíl mezi termínem model a metodika. Pro to, aby byl náš projekt úspěšný je klíčové zvolit vhodný model. Většina modelů je odvozena ze starších modelů vodopádového a spirálového. Do popředí se u firem nyní dostává také model RUP (Rational Unidied Process). [8]

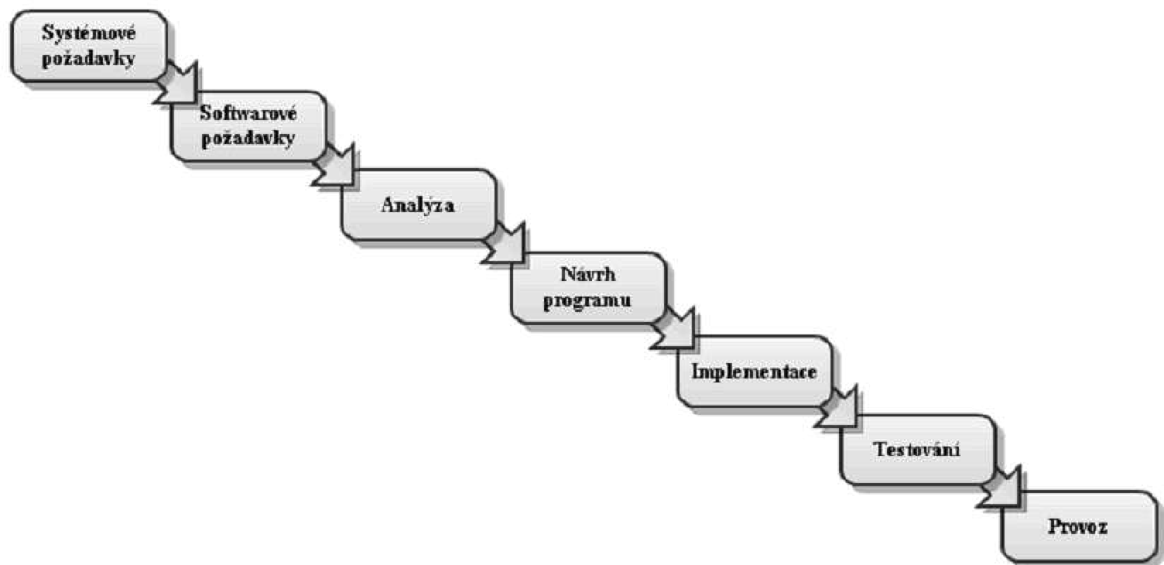
3.3.2.1 Vodopádový model

Tento model je tím nejstarším ze všech. Z názvu je patrný postup, jak se prochází jednotlivými fázemi. Lze si představit jako několik nádob, kdy z nejvýše položené nádoby začne postupně přetékat voda do té nejnižší jako vodopád. Z toho plyne postupnost jednotlivých fází. Důležité ale je, že každá fáze může začít až poté, co je předchozí fáze dokončena. U modelu je velmi důležité věnovat se počátečním fázím, díky čemuž lze počítat s úsporami v pozdějších fázích. Jde to dohromady s tím, že odstranění chyby, která se nalezne již během analýzy zabere mnohem méně času a vyjde mnohem levněji levněji, než kdyby se taková chyba odhalila až během testování produktu. Při vodopádovém modelu je stěžejní, aby před přechodem do další fáze byla ta současná kompletní a validní. [9]

Nevýhodou tohoto modelu je skutečnost, že v praxi především u rozsáhlejších projektů není možné dokončit jednu fázi před zahájením té následující. S největší pravděpodobností bude potřeba se k některým fázím v pozdější době vracet. Může to být způsobeno například tím, že klient trošku poupraví své požadavky. Poté je potřeba tyto změny uvést i ve specifikaci a dokumentaci produktu. Finální verzi produktu klient obdrží ve stavu, kdy již není možné provádět úpravy. Díky tomu může být i celý projekt

neúspěšný. Testování zde probíhá ve chvíli, kdy jsme hotovy se samotnou implementací produktu. Produkt již je tedy v podstatě připraven na předání klientovi. V tuto chvíli je již oprava velmi složitá a drahá. Pokud bude chyba již v analýze, může se stát, že bude potřeba přepracovat úplně celý projekt. Na obrázku 3 je vidět průběh vodopádového modelu. [9]

Obrázek 3 Vodopádový model



Zdroj [9]

Tento model je ideální na pochopení fungování životního cyklu softwaru. Je při něm důležité postupovat přesně podle návrhu, který je perfektně prověřený. Integrace takového systému je jednoduchá. Používá se však jen u projektů, u kterých se nepředpokládají další změny požadavků na chování výsledného programu. [9]

3.3.2.2 Spirálový model

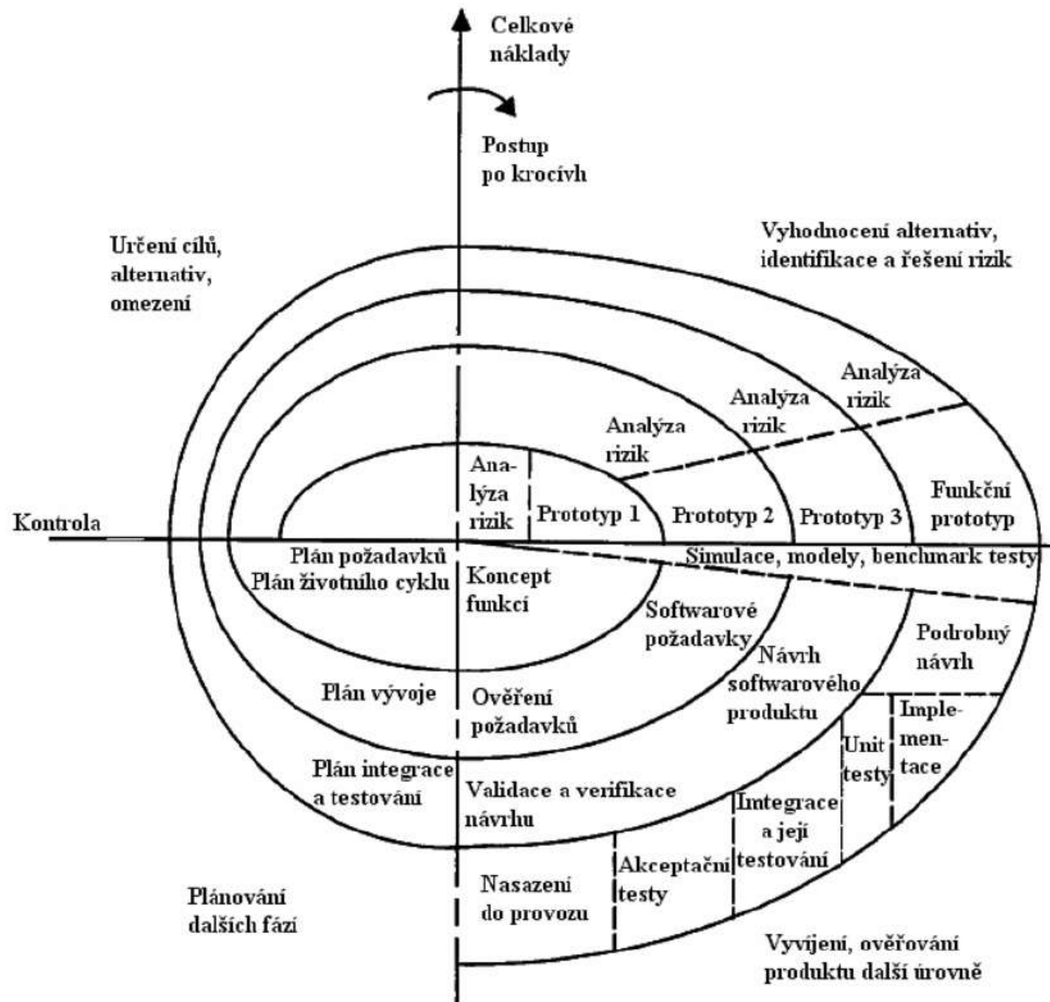
Jedná se o model, který kvalitně vyplňuje nedostatky vodopádového modelu. Před vstupem do další fáze se zde provádí precizní analýza rizik a možných problémů. Tato analýza je prováděna několikrát v rámci projektu díky čemuž je možné provádět i úpravy během projektu. Model má několik základních kroků, které se neustále opakují doku není projekt dokončen. Nejprve se pouze nahrubo udělá základ projektu, a poté je s každým

opakováním základních kroků specifikuje a upravuje více do hloubky. Model má čtyři hlavní kroky [10]:

1. Určení cílů, alternativ, omezení
2. Vyhodnocení alternativ, identifikace a řešení rizik
3. Vývoj a verifikace další úrovně produktu
4. Plánování následujících fází

Všechny fáze jsou zakončeny testováním a zhodnocením průběžných výsledků. Testování zde tedy probíhá pravidelně po celou dobu vývoje. Především proto, že se testy opakují je vhodné využití automatizovaných testů. Je třeba jen poupravit testovací scénáře na základě aktuálních aktualizací systému. Jelikož je systém pravidelně testován je většina především hrubých chyb odhalena velmi brzy a oprava je proto snadná. Na obrázku 10 lze vidět průběh spirálového modelu. [10]

Obrázek 4 Spirálový model

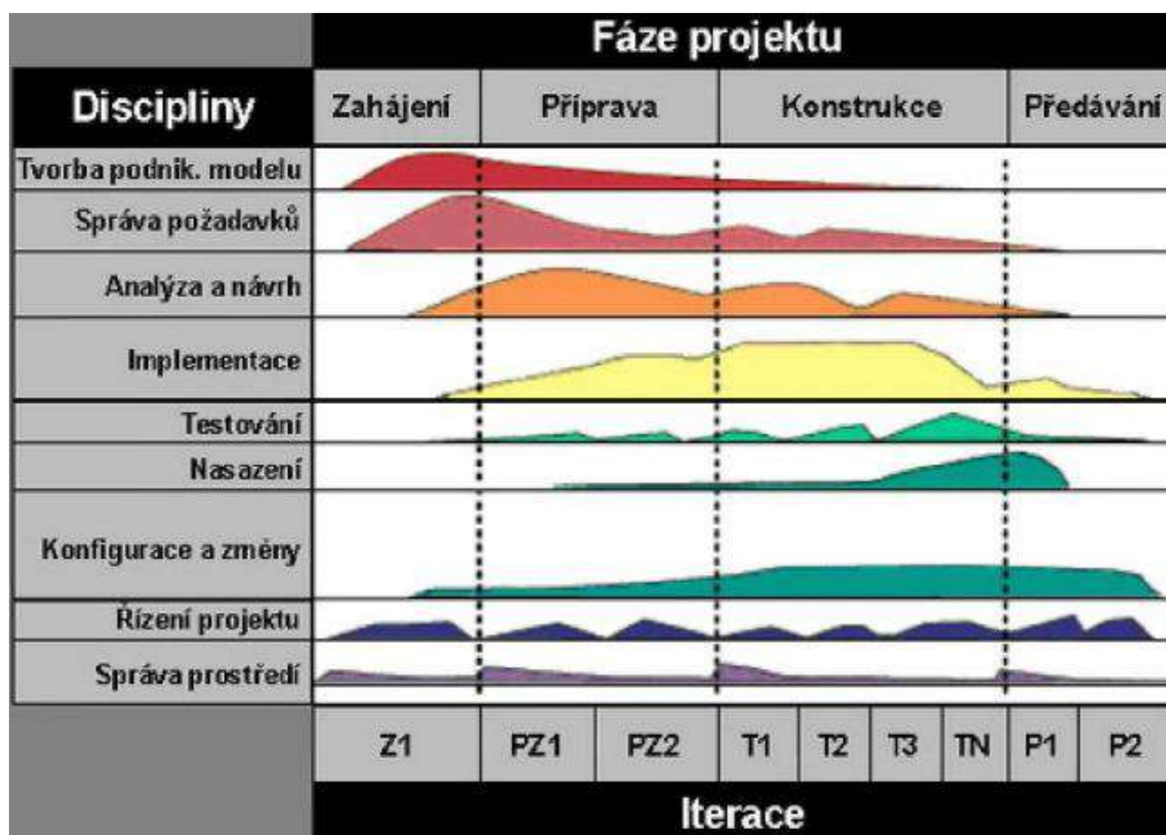


Zdroj [10]

3.3.2.3 RUP

Zkratka RUP znamená Rational Unified Process a jedná se o objektově orientovaný přístup k životnímu cyklu softwaru. V tomto modelu je testování velmi rozsáhlá oblast. Z obrázku 5 je patrné, že testování prochází všemi etapami projektu. Největší důraz je kladen na testování v závěrečné fázi vývoje před předáváním klientovi. Jsou zde definovány role a aktivity, které testováním musejí projít. Jakmile je testování hotové, sestaví se zpráva, ze které lze vyčíst které části systému je třeba pozměnit. [11]

Obrázek 5 Fáze RUP modelu



Zdroj [11]

3.3.3 Agilní vývoj

Hlavním znakem a výhodou agilního vývoje před tradičními modely je vysoká přizpůsobivost změnám, které se objevují v průběhu vývoje. V roce 2001 byl odborníky v IT formulován Agilní manifest, který obsahuje hodnoty a principy agilního vývoje. Hodnoty agilního vývoje [2]:

- Lidé a jejich vzájemná interakce jsou důležitější než procesy a nástroje.
- Fungující software je důležitější než vyčerpávající dokumentace.
- Spolupráce se zákazníkem má přednost před vyjednáváním smluv.
- Je důležitější reagovat na změny než dodržovat plán.

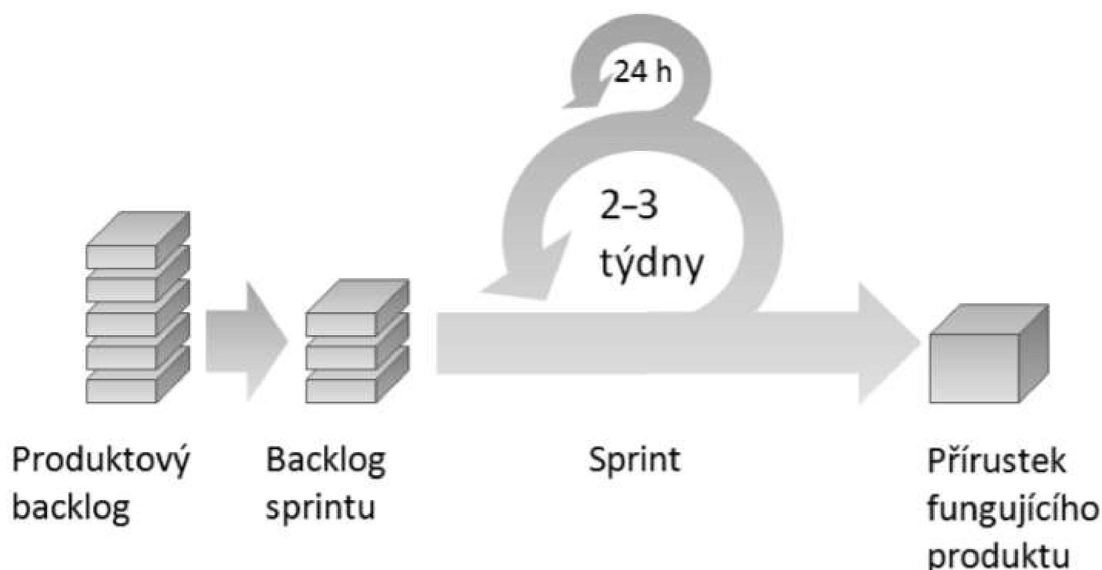
Agilní přístup se na rozdíl od tradičního zaměřuje na samotné členy týmu, společně s jejich dovednostmi a komunikací. Striktní dodržování procesů již není nutností a

dokumentaci je třeba vytvářet jen pokud je nezbytně nutná. Důraz je kladen především na funkčnost softwaru, podle čehož se dá odvodit k jakému posunu při tvorbě systému došlo. V intervalu dvou až čtyř týdnů jsou dodávány jednotlivé fungující části systému. Těmto intervalům se říká sprinty. Implementace celků je řízena zákazníkovo prioritou. Velkou roly zde hraje předpokládaná cena a čas vymezený k vytvoření konkrétního dílu softwaru. [2]

3.3.3.1 Scrum

Jedná se o jednu z nejznámějších a nejpopulárnějších metodik agilního vývoje.

Obrázek 6 Scrum model



Zdroj [2]

Na obrázku 6 je schéma znázorňující průběh Scrumu. Pod produktovým backlogem si lze představit seznam funkcionalit, které by měl systém splňovat řazený podle priorit daných zákazníkem s ohledem na předpokládanou složitost zavedení této úpravy. Do konkrétních sprintů jsou zařazeny úkoly, které by tým měl během konkrétního sprintu stihnout dokončit. Výsledkem sprintu by měl být fungující kód, který rozšíří, nebo upraví současný systém. Denně se pořádají schůzky celého týmu, na kterých každý popíše, čeho od poslední schůzky dosáhl, a čeho má v plánu do další schůzky dosáhnout. Rozebírají se zde i situace, které mohou bránit dalšímu vývoji. [2]

3.4 Metody a principy testování

Testování systému si lze rozdělit do čtyřech základních směrů, lišících se od sebe vzájemnou kombinací toho, zda k testování potřebujeme software spustit, a zda vidíme do zdrojového kódu softwaru nebo pouze pozorujeme jeho chování. [1]

Před tím, než se budeme zabývat podrobnostmi těchto konkrétních směrů si uvedeme úrovně neboli kategorie testů, které se od sebe liší podrobností zkoumání softwaru a dalšími ukazateli.

3.4.1 Testování programátorem

Jedná se o jedno z prvních testování, které lze při vývoji softwaru provádět a úplně první testování, které lze provádět na již funkčním programu. Během tohoto testování si programátor znovu projde kód, a zkontroluje, zda funkce opravdu dělají to, co bylo uvedeno ve specifikaci. Je doporučováno provádět tzv. „test čtyř očí“. Jde o to, že si program po sobě nekontroluje programátor, který jej napsal, ale předá ho k překontrolování některému ze svých kolegů. Pokud po sobě kód kontroluje programátor, který jej psal, je zde mnohem větší šance že některé chyby zůstanou přehlédnuty, než když se na kód podívá někdo jiný. V praxi se však toto testování hojně nevyužívá, přitom odhalení chyby v tomto stádiu ušetří spoustu času a prostředků v rámci celého projektu. [12]

3.4.2 Testování jednotek

Tyto testy přicházejí na řadu po překontrolování kódu programátory. Jako testovaná jednotka jsou zde myšleny části softwaru, která jdou samostatně otestovat. Např. u programů psaných v objektově orientovaném jazyce se tím myslí testování konkrétních tříd a metod. Těmto testům je třeba věnovat pozornost již před započetím vývoje, například již během návrhu aplikace. Je totiž potřeba, aby program byl sestaven tak, aby na něj šly tyto testy aplikovat. Pokud již je program vytvořen, a není dostatečně rozdělen do jednotek které by šli testovat musela by se provést např. refaktoring kódu nebo i daleko podrobnější úpravy. U menších projektů se však nevyplatí obětovat tolik času a u rozsáhlejších by zásah také nemusel dopadnout dobře, a proto již většinou k úpravě nedojde. [12]

3.4.3 Funkční a nefunkční testy

Během funkčního testování se ověřuje, zda aplikace dokáže bez chyb plnit všechny úkoly pro které byla vytvořena. Jde o to otestovat veškeré implementované funkce aplikace a že jejich fungování odpovídá tomu, jak je zákazník specifikoval. [13]

Nefunkční testy naopak testují takové vlastnosti aplikace, které ačkoliv nemají vliv na funkci, jsou podstatné pro to, aby fungovala tak, jak by se očekávalo. Testuje se zde třeba to, že aplikace bude fungovat v dostatečné rychlosti i při zátěži, čímž může být více aktivních uživatelů v jeden čas, nebo velký objem proudících dat. Zároveň se k těmto testům může řadit i testování, jak moc program zatíží systém, na kterém běží. [13]

3.4.4 Integrační testy

Integrační testy jsou prováděny poté, co je programátor hotov se svými testy. Nyní již testy provádí testovací tým. Během testů je ověřováno, zda komponenty aplikace správně komunikují mezi sebou. Dále se může testovat i komunikace mezi aplikací a systémem, hardwarem nebo rozhraním. Integrační testování je možné zcela vynechat. Pokud se správně provedou následující úrovně testování měli by být chyby, které jsou odhaleny zde být odhaleny později. Hlavní přínos integračních testů je v tom, že pokud je chyba odhalena nyní její oprava nás vyjde lépe než při odhalení v pozdějších úrovních. [12]

3.4.5 Systémové testování

V tomto kroku je již funkčnost aplikace testována jako celek. Testy probíhají v konečných fázích projektu a ověřují software pohledem, jaký bude mít na aplikaci zákazník. Jsou připraveny testovací scénáře kroků, jakými lze s aplikací pracovat a podle nich se následně aplikací prochází. Tyto testy mají několik kol. Pokud se v systému objeví chyba měla by být opravena a v dalším kole opět otestována. Systémové testy jsou poslední úrovní, která probíhá před tím, než si produkt převezme klient. Jedná se v podstatě o výstupní kontrolu vytvořeného programu. Jde o nejvýznamnější úroveň celého testování. Tato úroveň testování má největší podíl na tom, aby byl produkt bezporuchový. [12]

3.4.6 Akceptační testování

Tyto testy jsou prováděny poté, co všechny předchozí úrovně skončily úspěšně a aplikace byla předána zákazníkovi. Testy si již provádí zákazník sám podle scénářů, které v ideálním případě připravil společně s dodavatelem. Pokud se objeví odchylka od specifikace je tato skutečnost reportována zpět vývojářům, kteří by měli v co nejkratším době chybu opravit. Pokud je během akceptačních testů nalezeno příliš mnoho chyb, nebo jsou chyby pomalu opravovány dochází ke zpožďování nasazení do provozu, což může mít velké finanční následky. [12]

3.4.7 Testování černé a bílé skříňky

Rozdíl mezi těmito dvěma druhy testování je v tom, zda má tester přístup ke zdrojovému kódu programu či nikoliv.

Během testování pomocí černé skříňky tedy tester pouze vidí, jaký vstup podal programu a jaký od něj dostal výstup. Neví však jakým způsobem k tomu výsledku program došel. Zda je výstup k zadanému vstupu očekávaný je potřeba odvodit ze specifikace případně ověřit pomocí jiných způsobů.

U testování pomocí bílé skříňky může tester nahlédnout do zdrojového kódu a podívat se jakým způsobem program došel ze zadaného vstupu ke konkrétnímu výstupu. Lépe tak může rozumět fungování programu a odhadnout proč k chybě dochází. [1]

3.4.8 Statické a dynamické testování

Statické a dynamické testování se liší tím, zda je v průběhu testů program v běhu či nikoliv. U statického testování tedy program neběží pouze na pohled testujeme, zda splňuje to, co má. U dynamického testování již ale program funguje a mi pozorujeme jeho chování, zda je odpovídající či nikoliv. [1]

3.4.9 Testování specifikace

Jedné se o statické testování metodou černé skříňky. K testování tedy nepotřebujeme běh systému a ani nevidíme do kódu samotného programu. Samotné

testování může probíhat již před započetím vývoje softwaru. Chyby jsou díky tomuto testování odhaleny ještě předtím, než se do programu vůbec dostanou.

Od specifikace se očekává, že dopodrobna popíše, jak se má aplikace zachovat, pokud ji budeme používat uvedeným způsobem. Programátor by měl tedy při vývoji postupovat tak, aby se aplikace podle specifikace chovala. Zároveň i tester potom přesně ví jaké chování má u aplikace očekávat a jednotlivé odchylky reportovat. Pokud by specifikace nebyla dostatečně podrobná je zde velká šance, že programátorovo pohled na chování aplikace se bude rozcházet s pohledem uživatele na fungování aplikace.

V prvním kroku kontroly specifikace není hned nutné hledat v ní chyby. Je nutné najít spíše vynechané funkčnosti, omyly nebo přehlédnutí. Důležité je se na specifikaci podívat pohledem zákazníka. Pokud bude program určený pro interní fungování firmy je vhodné promluvit si o něm s lidmi, kteří jej budou používat abychom lépe pochopili jaké chování od něj očekávají. Kvalitní souftware jak bylo uvedeno dříve se pozná tím, že je v souladu s potřebami zákazníka.

Pro testera je důležité, aby dané specifikaci porozuměl. V konečné fázi se budou na základě specifikace vytvářet testy, a proto bude stejně nutné ji rozumět. Čím dříve tedy bude specifikaci rozumět tím dříve v ní bude moci objevit případné chyby a snadněji je opravit.

Vhodnou metodou, jak lépe pochopit požadované fungování produktu, je vyzkoušení si a pochopení fungování podobného softwaru. I přesto, že se prozkoumávaný software nemusí příliš shodovat s námi testovaným produktem, jeho prozkoumání nám pomůže nastínit s jakými problémy bychom se mohly setkat. Při prozkoumávání je vhodné se zaměřit na pár bodů:

- Velikost: zda bude náš software rozsáhlejší, případně zda bude mít velikost vliv na testování.
- Složitost: zda bude testovaný software složitější, a bude to mít vliv na testování.
- Testovatelnost: bude dostatek prostředků, času a zkušeností k otestování takového softwaru.
- Kvalita: odpovídá prozkoumávaný software kvalitou testovanému softwaru.

Aby bylo možné specifikaci považovat za správnou je potřeba aby splňovala osm základních atributů:

1. Úplná: je potřeba aby ve specifikaci bylo opravdu všechno, a byla samostatná.
2. Správná: je potřeba aby řešení a cíl projektu byly popsány správně bez jakékoliv chyby.
3. Jednoznačná: specifikace musí být snadno srozumitelná a její interpretace musí mít jediný výsledek.
4. Konzistentní: popis funkcí nesmí být v konfliktu se sebou samým nebo jinými částmi specifikace.
5. Relevantní: jsou dané popisy opravdu nezbytné? Dá se ze specifikace vyčíst původní potřeba zákazníka?
6. Proveditelná: je možná danou funkci vytvořit s dostupným personálem prostředky a v dostačujícím časovém horizontu?
7. Bez kódu: specifikace by se měla věnovat pouze definici produktu, a ne návrhu softwaru nebo kódu.
8. Testovatelná: dá se funkce otestovat? Specifikace musí obsahovat dostatek informací k tomu, aby byl tester schopný vytvořit podle ní adekvátní testy.

Pokud při některé části testování specifikace narazíte na to, že neodpovídá těmto vlastnostem, jedná se o chybu a je potřeba ji začít řešit. [1]

3.4.10 Testování softwaru bez znalosti kódu

Během takzvaného dynamického testování černé skříňky se testuje software za běhu bez toho, abychom znali jeho zdrojový kód. Testování probíhá stylem, jako kdyby s programem pracoval běžný uživatel. Do programu jsou zadávány různé vstupy, a jsou kontrolovány výstupy, které po konkrétním vstupu vychází. K tomu, aby bylo možné zkontrolovat správnost výstupů, je potřeba znát definice činnosti softwaru tzn. jeho specifikaci.

Ne pokaždé je ale při vývoji softwaru k němu dostupná i specifikace daného systému. Jedná se především o menší projekty. V takovém případě je potřeba přistoupit na badatelské testování. V takovém případě je potřeba si software nejdříve projí a udělat si obrázek o tom, jak má fungovat. Takový software sice nelze prozkoumat tak důkladně, ale i přesto lze v testech udělat jistou systematičnost.

Pokud je známo, jaký výstup má program po zadaní vstupu vrátit, je na řadě vytvoření testových případů. V testových případech je definována množina vstupů, dále postup, jakým se mají vstupy zadávat, a nakonec očekávané chování softwaru.

Testování si lze rozdělit do dvou základních tříd na testy splněním a testy selháním. Nejprve se začíná testovat pomocí testů splněním. Během těchto testů se program testuje normálními vstupy a očekává se, že bude fungovat tak jak bylo zamýšleno. Teprve až pokud tyto testy prochází bez problému lze přistoupit i na testy selháním. Zde se již testů hraniční situace a případy, ve kterých jsou chyby předpokládány, že by se mohly vyskytnout. Při těchto testech lze záměrně zadávat do programu vstupy, které nejsou ve specifikaci uvedeny, a sleduje, zda se program chová, jak má nebo zda např. nepřestane fungovat.

Množina všech testů na zkontrolování chování programu je takřka nekonečná, a proto je potřeba rozdělit si testy do tříd ekvivalentních případů. Do třídy ekvivalentních případu spadají úkony, které mají stejné chování přesto, že se vstupy mohou trochu lišit. Lze si představit třeba kalkulačku, kde namísto vyzkoušení sčítání všech čísel je vyzkoušeno pouze sčítání na několika číslech. Chování bude v takovém případě stejné pro celou množinu, která by do této třídy zapadala.

Při testování je potřeba si dát pozor zejména na zkontrolování hraničních podmínek. Jedná se o hodnoty, které jsou ve vzdálenosti 1-2 od okrajů množin ekvivalence. V hraničních hodnotách je největší pravděpodobnost výskytu chyb. [1]

3.4.11 Zkoumání programového kódu

Zkoumání programového kódu spadá pod statické testování bílé skříňky. Jedná se tedy o testování, při kterém není program spuštění, a tester má možnost nahlédnout do jeho zdrojového kódu.

Ačkoliv se jedná o velmi účinnou možnost, jak odhalit chyby již v počátcích projektu, kdy jejich oprava bude jednoduchá a levná, je tento postup často opomíjený. Testuje se zde návrh výsledného softwaru a jeho zdrojový kód. Pro kvalitní otestování je potřeba, aby kód otestovalo co nejvíce nezávislých osob od programátora, který kód napsal. Může se jednat o ostatní programátory, nebo testovací tým.

Ze zkoumání kódu programu mohou vzniknout testové případy, které se poté dají využít během dynamického testování černé skříňky. Testeři totiž mohou z komentářů,

které vytvoří programátoři během revize kódu zjistit, které funkce softwaru jsou problémové, nebo náchylné k chybám.

Nejjednodušším testováním je, pokud kód po napsaní zkontroluje po programátorovi nějaký jeho kolega nebo tester. Tím, že kód kontroluje někdo jiný, se zamezí tomu, aby programátor své chyby přehlédl. Zároveň je zde zohledněn i pohled na problém jiného člověka, a tak spolu mohou diskutovat, jak by se dal problém řešit jinak.

Další možností otestování kódu je, že programátor svůj kód prezentuje před skupinou ostatních programátorů, kteří se s kódem již dopředu seznámili a během průchodu kódu vznášejí na programátora své dotazy. Programátor tedy postupně vysvětluje jednotlivé funkce a následně ke každé funkci proběhne diskuze o tom, kde by mohly být případné chyby, nebo zda by se to dalo udělat lépe.

Kód programu se nekontroluje jen kvůli odhalování chyb, dalším důležitým faktorem je i zkontrolování dodržování jistých zásad a standardů programování. Tyto chyby se tedy neprojevují špatnou funkčností programu ale tím, jak je kód napsaný a čitelný. Takové chyby tedy může obsahovat i kód, který perfektně plní vše, co od něj bylo požadováno.

Standardy se rozumí pevně daná pravidla, které přesně říkají, co se smí a nesmí dělat. S výjimkami se zde prakticky nedá setkat. Naproti tomu zásady jsou volnější. Jedná se o jistá doporučení a navrhování nejlepších postupů pro psaní kódu.

Řídit se standardy a zásadami je důležité z třech základních důvodů:

1. Spolehlivost: Pokud je kód napsán se zavedenými standardy a zásadami je prokazatelně spolehlivější a k chybám méně náchylný, než kdyby tak nebylo.
2. Čitelnost a udržitelnost: Takto napsaný kód je snadno čitelný a srozumitelný, především pokud s ním pracuje více programátorů a jeho udržitelnost je proto snazší.
3. Přenositelnost: Pokud bude potřeba program spustit na jiném hardwaru nebo s různými komponenty, bude přechod na další platformu mnohem jednodušší, pokud budeme postupovat v souladu se standardy a zásadami. Takovýto přechod může být i naprosto bezproblémoví. [1]

3.4.12 Dynamické testování bíle skříňky

Jedná se o testování, při kterém je potřeba aby byl program spuštěný a zároveň tester ví, jak vypadá samotný kód programu. Co se budete testovat se odvíjí od toho, co tester vyčte ze zdrojového kódu.

Díky tomu, že má tester přístup ke zdrojovému kódu, se jeho práce může výrazně zkrátit, jelikož může přesně určit co je potřeba otestovat, a kde by byly testy naopak zbytečné. Po důkladném pročtení kódu lze vyloučit redundantní testové případy, a naopak se zaměřit na takové, které v začátcích testování vůbec nebyly v plánu. Efektivita testování se díky tomu výrazně navyšuje. [1]

3.4.13 Manuální testování

Za manuální testování se považuje takové, které provádí samotný tester celé sám a nijak mu v tom nepomáhá žádný další software. Celý test probíhá krok po kroku podle testovacího scénáře, který byl ke konkrétnímu testu připraven. Takovéto testování je vhodné v okamžiku, kdy se předpokládá, že se test nebude opakovat, pokud je k vyhodnocení testu potřeba lidský pohled na věc, nebo je nemožné test zautomatizovat.

Výhodou manuálního testování je skoro nulová časová náročnost na přípravu testu. Provádění samotných testů je již ale v porovnání s automatizovanými testy mnohem delší. Největším nevýhodou manuálních testů je že celý postup i pozorování výsledků provádí člověk, a proto se může stát, že test neproběhne správně, případně se přehlédne nějaké nečekané chování. [14]

3.4.14 Automatizované testování

Automatizované testování funguje principem, že software není testován přímo testerem ale programem, který je nastaven tak, aby správnou funkčnost softwaru zkontroloval. Automatické testy se vyplatí užívat, pokud se testy neustále opakují. Dochází k tomu například pokud se přidává nová část programu, aby se zkontrolovalo, zda chod starých částí zůstal správný. Tím, že testy probíhají automaticky ušetří to spoustu času testerům, kteří se mohou věnovat jiným činnostem. [2]

I přes výhody automatizovaných testů nelze všechny testy provádět tímto způsobem. U některých testů je potřeba zohlednit lidský faktor, a proto jsou potřeba i nadále manuální testy. Hlavními vlastnostmi automatizovaných testů jsou:

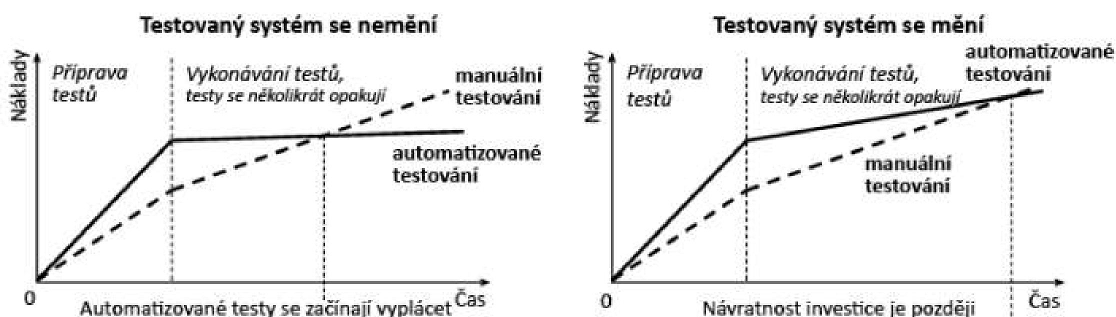
- Rychlost: pokud tester prochází nějakou aplikaci třeba jen přihlášení mu může zabrat několik vteřin. Oproti tomu software takovýto úkon dokáže provést klidně 10x nebo i 1000x rychleji.
- Efektivita: pokud tester provádí manuální testy musí se plně soustředit na průběh testu. Pokud pouze spustí automatický test může během času, kdy čeká na výsledky přemýšlet nad dalšími možnými testovacími scénáři.
- Správnost a přesnost: člověk začne po stálém opakování stejné práce ztrácet postupně pozornost a v testech se začnou objevovat chyby. Stroj udělá stejný test při každém opakování naprosto stejně.
- Neúnavnost: testové nástroje se nemohou unavit. Mohou testy opakovat neustále dokola bez přestávky, dokud bude testování programu vyžadované. [1]

Během automatizovaného testování se můžeme setkat se dvěma základními typy nástrojů. Těmi jednoduššími jsou nástroje neinvazivní. Takové nástroje dokáží software sledovat, neprovádí však žádnou modifikaci ani změnu parametrů. Tester tedy nevnímá, co se děje uvnitř nástroje a pouze sleduje co nástroj dělá. Jedná se tedy o nástroj, který testuje principem černé skříňky.

Složitější skupina nástrojů je označována jako invazivní. Tyto nástroje testerovi umožňují měnit parametry a provádět modifikace. Je spousta nástrojů lišících se od sebe množstvím invazivnosti. Upřednostňovat by se měli takové nástroje, jejichž invazivnost je co nejnižší. Tester musí při užívání těchto nástrojů znát procesy programu, a proto využití invazivních nástrojů spadá pod testování bílé skříňky. [2]

Z finančního hlediska by se automatizování testů mělo vyplatit, po uplynutí jistého počtu manuálního opakování konkrétního testu. Je to zapříčiněno téměř nulovými náklady na opakování testu kdy opakovaná manuální opakování je stále stejně drahé. Proto po jisté době opakování manuálních testů přesáhne cenu na vytvoření automatizovaného testu. Na obrázku vlevo jsou znázorněny předpokládané náklady na testování a na obrázku vpravo je znázorněn reálný pohled na náklady testování.

Obrázek 7 Porovnání ceny automatizovaného a manualního testování



Zdroj [15]

Důvodem, proč cena automatizovaných testů s časem stále stoupá je jistá změna systému a s tím související výdaje na údržbu testů. Z těchto důvodů je vhodné si před vytvářením automatizovaných testů udělat analýzu, zda se opravdu vyplatí a zda náklady na následnou údržbu nebudou tak vysoké, že by se úspora nevyplatila. [15]

Automatizované testy lze vytvářet pomocí několika odlišných technik. V praxi je možné tyto techniky různě spojovat, díky čemuž lze dosáhnout nejlepšího výsledku. Techniky se dělí na čtyři základní odvětví [2]:

1. Zachycení a přehrávání aktivity uživatele – jedná se o nejjednodušší a nejznámější formu automatizovaného testu. Podstata testu je založena na tom, že si program zapamatuje aktivitu testera, a poté stejnou aktivitu vykonává automaticky. Úspešnost testu je vyhodnocena podle očekávané hodnoty některé z proměnných na konci testu.
2. Modifikace vygenerovaných skriptů – jedná se o úpravu skriptů z předchozí techniky. Díky znalosti skriptovacího jazyka je tester schopný takovéto skripty poupravovat a dosáhnout tak náročnější logiky případně rozšířit záběr konkrétního testu. Testy jsou díky tomu i lépe udržitelné.
3. Testování řízené daty – tento postup je využit v okamžiku kde probíhá stejný test několikrát za sebou ale pokaždé s jinými vstupními daty a s tím i souvisejícími výstupy. Vstupní data jsou proto do skriptu postupně načítána z jiného souboru, ideálně tabulky a po provedení testu je výstup opět zkontrolován s výstupem uvedeným v souboru se vstupy.
4. Testování řízené klíčovými slovy – jedná se o příbuzný postup k předchozímu. V tabulce s daty jsou však umístěny i jednotlivé příkazy,

které následně mohou poskládat skript. V průběhu testu jsou tyto příkazy načítány čímž se zajistí dynamičnost testu.

3.5 JAVA

Java je čistě objektově orientovaný jazyk a interpretovaný jazyk. Její vývoj začal v roce 1990 ve společnosti Sun Microsystems v týmu pod vedením Jamese Goslinga. Původním cílem bylo vytvořit systém, který bude fungovat v domácích spotřebičích. Nejprve se systém pokoušeli vytvořit pomocí jazyka C++, avšak on ani žádné z jeho modifikací tomuto účelu nevyhovovaly. Pokoušeli se použít i jazyk Pascal, výsledek byl ale opět stejný. Nakonec tedy přistoupili na nový jazyk, který navrhl Gosling s názvem Oak. Jazyk přímo vycházel z jazyka C++.

V roce 1995 po zjištění, že již jazyk s názvem Oak existuje byl jazyk přejmenován na Javu. Tentýž rok byla Java poprvé představena na konferenci SunWorld. Programy psané Javou jsou interpretovány pomocí virtuálního počítače JVM, proto může fungovat nezávisle na konkrétním hardwaru nebo systému. Dnes již jazyk není čistě interpretovaný, ale je používána Just-In-Time kompilace, což programy zrychluje. Podle společnosti Sun je jazyk Java objektově orientovaný, interpretovaný, vysoce výkonný, více vláknový, nezávislý na architektuře, robustní, bezpečný, distribuovaný a jednoduchý. [16]

3.5.1 specifické rysy jazyka

Od Javy bylo očekáváno, že bude snadná k naučení a používání. Dosáhlo se toho odstraněním konstrukcí moc nepoužívaných, nebo takových, které v mnoha případech vedli k velkému množství skrytých chyb. Znaky Javy je silná typová kontrola, nelze užívat ukazatele a nepřítomnost vícenásobné dědičnosti.

Správu paměti má v Javě na starost JVM. Aplikace jej žádají o přidělení paměti a její uvolnění probíhá automaticky za pomoci garbage collectoru. Díky tomuto přístupu nemohou nastat problémy se špatně uvolněnou alokovanou pamětí čímž je zamezeno vzniku memory leaks. Garbage collector nyní běží stále na pozadí jako samostatné vlákno o nízké prioritě. [16]

3.5.2 Objektově orientované programování

V průběhu let programátorům došlo, že kvalita programu se zvětšuje, s mírou abstrakce, jaká je při jejich výrobě použita. Lze si představit například při programování pohybu ruky robota. Lépe se bude pracovat s jazykem, kde je možné zadat příkaz „Zvedni ruku” místo přesných informací, kterým by porozuměl stroj jako je „Dej do registru A jedničku a poté pošli obsah registru na port pět”. Díky zvýšení množství abstrakce, které lze v kódu použít je nyní možné rychle a spolehlivě vytvářet větší a komplikovanější programy.

Myšlenka, která ovlivňuje objektově orientované programování je, že vytvářené programy jsou simulací vymyšleného nebo i skutečného světa. Čím výstižněji dokáže programátor simulaci vystihnout, tím lepší bude výsledný program.

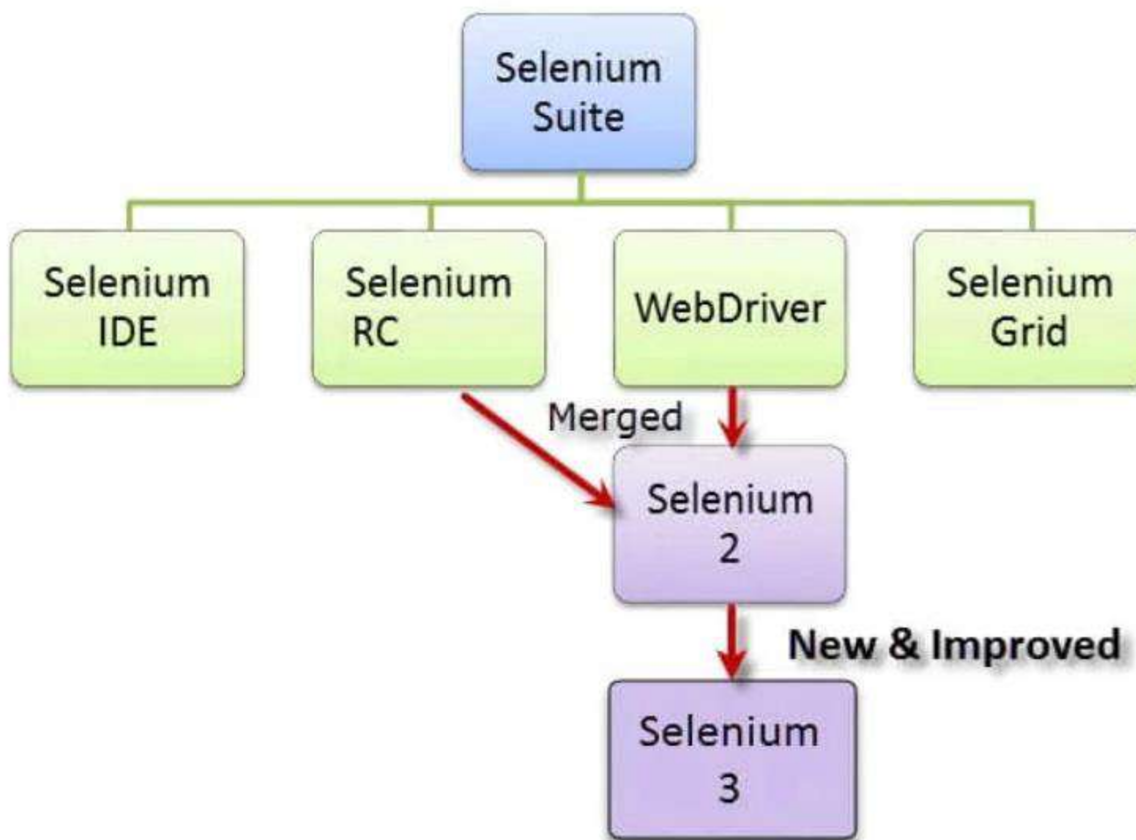
Jednotlivé světy jsou považovány za objekty se specifickými vlastnostmi. Tyto objekty mezi sebou komunikují posíláním zpráv. Pokud je požadováno rychle psát kvalitní program, je nezbytné mít k tomu vhodný programovací jazyk, s jehož konstrukcemi lze co nejpřesněji vystihovat co má program provádět. Java je přesně takový jazyk, díky kterému je možné s vysokou mírou abstrakce popisovat chování programu. [17]

3.6 Selenium

Testovací nástroj Selenium vznikl v roce 2004 ve firmě ThoughtWorks v Chicagu. Původně se jednalo o nástroj „JavaScriptTestRunner“, která testovala interní aplikaci firmy pro sledování času a výdajů spojených s projekty. Hlavním vývojářem byl Jason Huggins společně s Paulem Grossem a Jie Tina Wangem. Ze Selenia se později stal program pomáhající s funkčním testováním webových aplikací. [18]

Dnes je Selenium open-source framework pro automatizované testování webových aplikací napříč webovými prohlížeči a operačními systémy. Testovací skripty je v Seleniu možná psát v Java, C#, Pythonu a spoustě dalších programovacích jazycích. Selenium je celá sada softwarů, které lze užít při testování. Každá část je schopna poskytnou odlišné možnosti testování. Na obrázku 8 lze vidět rozdělení nástroje Selenium. [19]

Obrázek 8 Rozdělení nástroje Selenium



Zdroj [19]

3.6.1 Selenium IDE

Jedná se o jednoduchý framework, který zaznamenává chování uživatele a může ho později napodobit. Díky tomu je jednoduchý k naučení a testy se s ním dají dělat poměrně vysokou rychlostí. Je to plugin do Firefoxu, který se dá jednoduše nainstalovat. Právě ale díky jeho jednoduchosti by se mělo Selenium IDE užívat pouze jako prototyp. Pokud chtějí testeři vytvářet složitější testy měli by vždy použít buď Selenium RC nebo WebDriver.

Vytvořeno bylo japonským programátorem Shinyou Kasatani, kterému se líbil princip programu JacaScript Test Runner a chtěl vytváření automatických testů ještě více usnadnit, proto napsal program, s kterým mohou snadno a rychle pracovat i začátečníci především díky jeho přívětivému rozhraní a ovládání.

Jednoduchost programu je ale i jeho největší slabinou. Cenou za jednoduchost je nevhodnost pro vytváření složitějších testů, především kvůli absenci možností, které

poskytuje psaní testů za pomoci programovacích jazyků. Hlavními nedostatky jsou například:

- Nepřítomnost polí
- Absence seznamů
- Nepraktické vytváření cyklů
- Nelze kód opětovně použít

Starat se o kód s takovými nedostatky je poté trnem v patě, jelikož pokud se tester rozhodne provést nějakou změnu bude ji muset provádět jednotlivě v každém zakomponovaném testu. [19]

3.6.2 Selenium RC

Selenium Remote Control byl hlavní testovací framework celé sady Selenia. Jednalo se o první možnost automatického testování webu, kde mohly uživatelé požít jazyk který preferovali. Podporované jazyky jsou: [19]

- Java
- C#
- PHP
- Python
- Perl
- Ruby

3.6.3 Selenium WebDriver

V mnoha ohledech se jedná o lepší framework, než je IDE nebo RC. Jsou zde použity modernější a stabilnější přístupy k automatizaci prohlížeče. Na rozdíl od RC se WebDriver nespoléhá při testování automatizace na JavaScript a prohlížeč ovládá přímou komunikací. Podporované jazyky jsou stejné, jako byli u RC.

WebDriver je považován především za nástroj, se kterým lze jednoduše automaticky testovat webové aplikace přes jejich uživatelské rozhraní. To není ale jedinou věcí, kterou lze provádět. WebDriver je schopen provádět jakoukoliv práci na webových stránkách, u které lze předpokládat že se bude opakovat, a proto je potřeba ji

automatizovat. Pro napodobení akcí uživatele ve webovém prohlížeči používá WebDriver své originální programové rozhraní, díky kterému lze nativně pracovat s prohlížečem. [19]

3.6.4 Selenium Grid

Grid je využíván společně se Seleniem RC aby mohly testy běžet paralelně napříč různými přístroji a prohlížeči ve stejný čas. Díky tomuto přístupu lze při testování ušetřit obrovské množství času. Tvůrcem této části byl Patrick Lightbody, který byl natolik odhodlán co nejvíce zrychlit exekuci automatických testů, že opustil i svou původní práci, aby se mohl projektu více věnovat. Pro používání Selenia Grid jsou dva podstatné důvody:

1. Testy je nutné provádět na více prohlížečích, ať už druzích, verzích anebo prohlížečích běžících na rozdílných platformách.
2. Potřeba provést sady testů v co nejkratším čase. [19]

3.7 Lokalizování prvků webu pomocí XPath

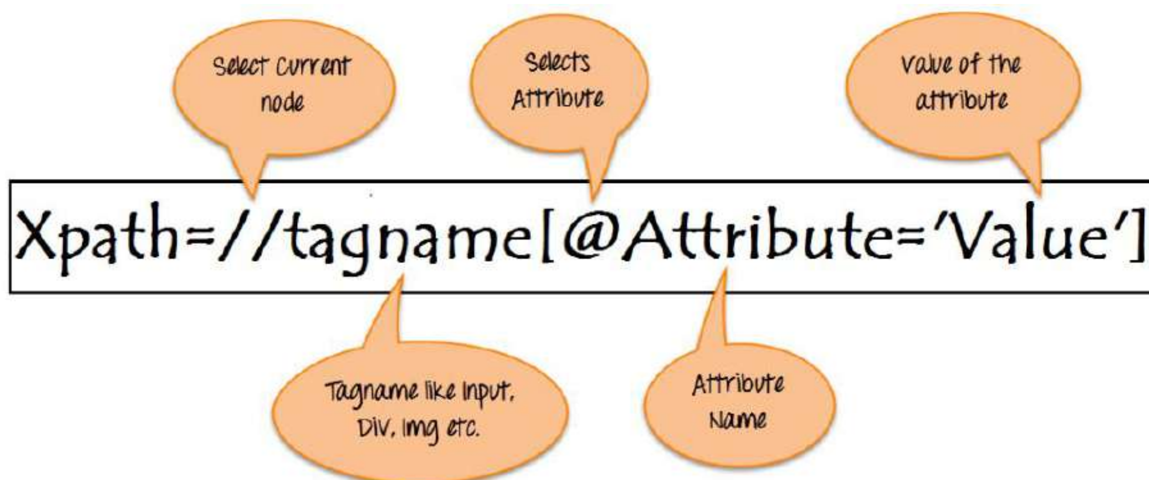
Pro správné automatizované ovládání akcí na webu je vhodné vědět, jak jednotlivé elementy na stránce přesně vyhledat. Takovými elementy se rozumí například různá tlačítka, vstupy nebo notifikace. K takovému účelu slouží Document Object Model neboli DOM. Díky tomu lze XML stránky načíst do paměti v jakési objektově prezentované stromové struktuře. Pro lokalizování prvků v HTML stránky lze využít několika možností. Těmi nejvhodnějšími je lokalizace podle CSS selektoru, a především podle výrazu XPath. [20]

XPath lze požít jako dotazovací jazyk pro vyhledávání elementů webové stránky. Jednotlivými dotazy lze vyhledávat na stránce množinu uzlů, nebo i konkrétní uzly a dále s nimi pracovat. XPath má dvě možnosti, jak zapsat cestu:

1. Absolutní: Při tomto způsobu se vypisuje kompletně celá struktura stromu, za pomoci hierarchie konkrétních elementů od absolutního začátku HTML až po vyžadovaný element. Na začátku absolutní cesty je vždy použito lomítko (/), které znamená, že se jedné o kořen HTML. Problémem tohoto typu cesty je, že pokud dojde k úpravě cesty, která původně vedla k hledanému prvku, je absolutní cesta nefunkční.

2. Relativní: Od absolutní cesty se liší tím, že se v ní nacházejí pouze elementy, beze kterých by nebylo možná nalézt potřebný prvek. Na začátku zápisu se nachází lomítka dvě (//) a hledá prvky všude po stránce. Používání relativní cesty by mělo být preferované. [21]

Obrázek 9 Relativní cesta XPath



Zdroj [22]

Na obrázku je ukázka relativní cesty XPath. Zápis celého výrazu začne dvojitým lomítkem, kterým se označí konkrétní uzel. Tagname se rozumí o jaký element ve struktuře se bude jednat, na webových stránkách jsou to elementy jako je třeba div, span, input, button nebo li. Možností je užít i symbol hvězdičky (*), kterým se dá najevo že je možné použít všechny elementy které budou odpovídat zbylým kritériím. Pro vybírání atributu je potřeba užít symbolu zavináče (@) poté napsat název atributu (class, id, title, placeholder…) a do uvozovek požadovanou hodnotu atributu.

Ke klasickému způsobu zapisování XPath za pomoci hierarchických vztahů, které mají elementy mezi sebou je možné používat i funkce. Díky funkcím je možné používat dynamické zápisy XPath. Takové zápisy se hodí, pokud lokalizace klasickou metodou nejde provést. Lze si uvést nejběžnější funkce, které k vyhledávání využít: [22]

- contains() se používá tehdy, pokud se hodnota některého atributu průběžně mění. Funkce dokáže vyhledat kde se v DOM vyskytuje konkrétní text. Lze tedy najít například element jehož atribut obsahuje požadovaný text nebo i konkrétní text na stránce.

- starts-with() je o něco specifičtější metoda než contains(). Jak název napovídá, bude hledat text, který bude vždy na začátku. Vhodná je především u takových elementů kde se může název atributu v čase měnit, ale začátek zůstane stejný.
- OR & AND jsou běžné logické operátory, které se uvádějí běžně v závorkách společně s predikátem. Pomocí operátoru OR, který je logickým součtem lze najít elementy, který obsahují alespoň některou z uvedených kritérií. AND, jež je logickým součinem naopak lze vyhledat takové elementy, které splňují všechny kritéria.

3.8 Nástroje usnadňující práci s kódem

Existuje celá řada programů a služeb, které ulehčují psaní kódu a následnou práci s ním. Mezi nezákladnější nástroje pro vývojáře patří IDE, ve kterých svůj kód píše a společně s ním nějaký program určený pro verzování kódu.

3.8.1 IDE

Většina programátorů dnes používá tzv. integrovaná vývojová prostředí jejich zkratka z anglického překladu je IDE. S nimi je programování mnohem pohodlnější, neboť za programátora vyřeší mnoho konfiguračních detailu.

U vyspělejších IDE se může stát, že jejich plné ovládnutí bude zezačátku mnohem složitější než naučení se konkrétního programovacího jazyka. Pokud ale programátor práci s IDE porozumí, jeho práce se značně ulehčí, neboť kvalitní IDE spoustu věcí vyřeší za programátora a, nebo mu bude dopodrobna napovídat kde udělal chybu.

Vývojových prostředí existuje nepřeberné množství, z nichž některá jsou určena začátečníkům a další jsou spíše připravená pro zkušené programátory. Práce s profesionálními prostředími je sice mnohem složitější, ale výsledná pomoc, kterou prostředí poskytnou je daleko větší. [23]

3.8.1.1 NetBeans

Prostředí, které vzniklo jako závěrečná práce na Matematicko-fyzikální fakultě Univerzity Karlovy v roce 1996. V jeho historii se několikrát změnil vlastník, nyní je ale

vyvíjen nadací Apache Software Founfation. Jedná se o nejstarší IDE v rámci třech velkých vývojových prostředí. Mezi vývojáři se těší stabilní popularitě mezi 10-15%. Dle jednotlivých recenzí je to nejsnáze zvladatelné z profesionálních prostředí, bez toho, aby to ovlivnilo jeho funkce. Je vhodné pro budování velkých projektů v případě, že za prostředí nechce programátor utrácet. [23]

3.8.1.2 IntelliJ IDEA

IDE za jejímž zrodem stojí pražská společnost JetBrains vyšlo v lednu 2001. I když bylo nejprve pouze v placené verzi, mezi kodéry se setkalo s velkou oblibou. V srpnu 2013 vyšla i edice, kterou šlo stáhnout a zcela bezplatně, a tím se popularita začala nepřetržitě zvyšovat. Dnes toto prostředí používá přibližně 60% vývojářů, kde neuvěřitelných 50% představuje placenou verzi a zbylých 10% jsou hlavně studenti a začínající vývojáři.

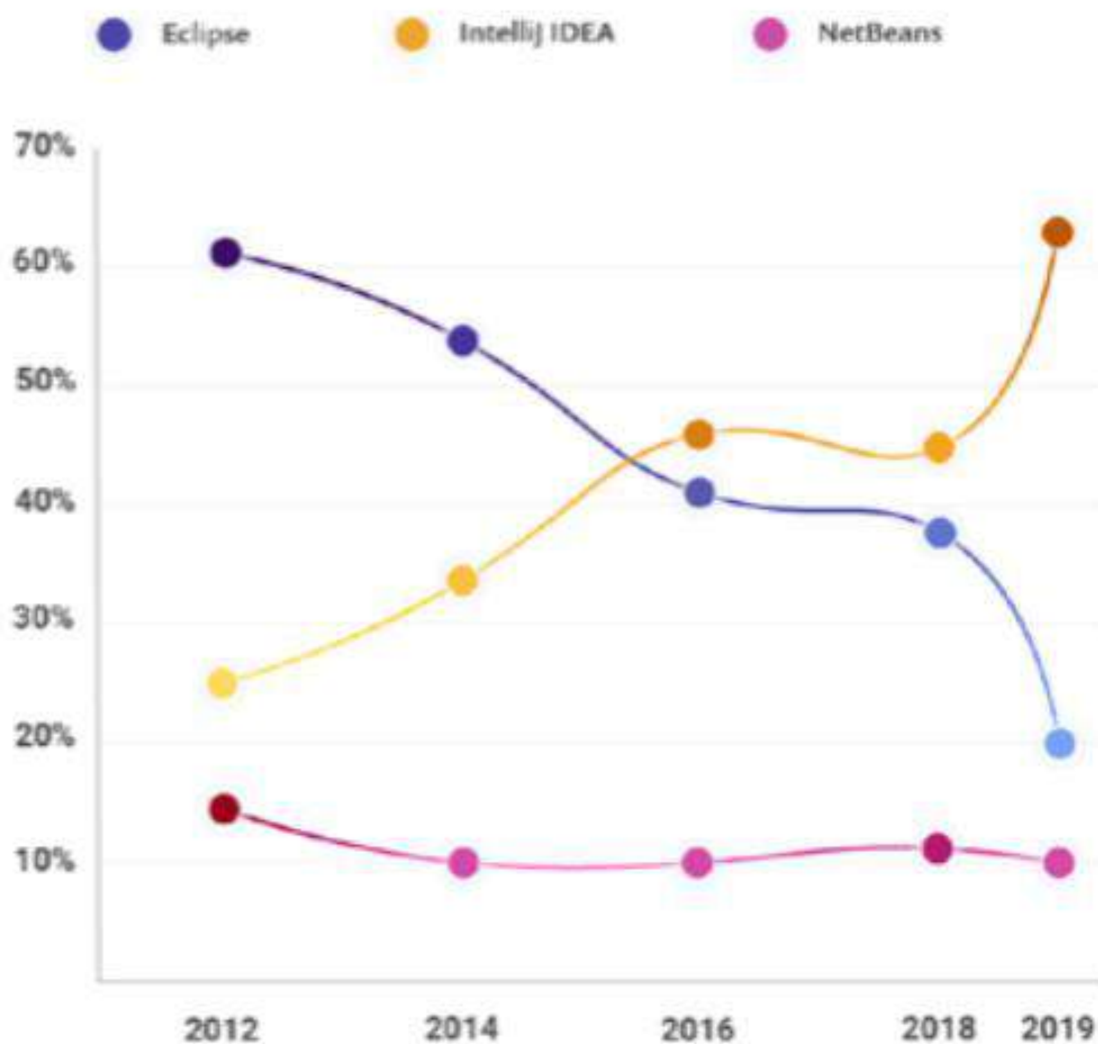
Slabinou prostředí je trošku těžší porozumění všem jeho funkcím. Začátečníkům se může stát, že z některých jeho nápověd budou poněkud zmatení a budou jim překážet. Ale jak bylo uvedeno výše, pokud se vývojář naučí všechny vymoženosti vývojového prostředí jeho vynaložená práce se mu vrátí.

S bezplatnou verzí je bohužel možné vytvářet pouze středně velké projekty, pokud je potřeba se ponořit do velkých programů je potřeba přejít na placenou verzi. Dle komunity se ale investice do verze Ultimate bohatě vrací. [23]

3.8.1.3 Eclipse

Jedná se o poslední z největších vývojových prostředí, které vzniklo ve firmě IBM v srpnu 2001. Díky tomu že IBM dokázalo seskupit koalici firem, pro vylepšování prostředí a zároveň bylo zdarma, bylo během chvíle využíváno přes 60% programátory v Javě. V posledních letech ale jeho popularita výrazně klesá a ní je na přibližně 20% které stále klesají. Na popularitu jednotlivých prostředí se lze podívat v následujícím obrázku. [23]

Obrázek 10 Vývoj popularity jednotlivých IDE



Zdroj [23]

3.8.2 Git

Tento systém slouží ke správě verzí kódu programu. Prvního vydání se dočkal 7. dubna 2005. Jeho stvořitelem byl Linus Torvalds, který patří k hlavním programátorům linuxu. Dříve totiž k vývoji linuxu používali Bitkeeper, ale po jeho zpoplatnění se vývojáři rozhodli, že vyvinou vlastní program pro správu verzí. Git byl navržen tak, aby v něm bylo možné uchovávat velké projekty a zároveň bylo pro vývojáře pohodlné s ním pracovat.

Git v sobě tedy uchovává jednotlivé verze, díky čemuž mají Programátoři přístup k celé historii projektu. Historií projektu se mohou vývojáři volně pohybovat, prohlížet si

kdo jakou část vytvořil a přesně vidět které řádky se mezi jednotlivými verzemi změnili. Velkou výhodou je možnost větvení, kdy může vývojář na své větvi provádět jakékoliv změny, aniž by to ovlivnilo projekt. Jestliže se poté rozhodne, že změny, které provedl jsou nevhodné, může větev jednoduše opustit a vrátit se zpět do bodu před začátkem úprav. Projekt jako takový ale zůstane nedotčen. [24]

3.9 Současný stav testování softwaru

Jak bylo uvedeno v teoretické části práce, tak testování softwaru má v současnosti velký podíl na úspešnosti projektu. Díky vhodnému testování je možné ušetřit velké náklady za případné opravy, nebo předejít selhání projektu, pokud by byl vytvořen s chybami neslučitelnými se správným fungováním aplikace.

V současných agilních přístupech výroby softwaru je celosvětově na dohled na kvalitu softwaru vynaloženo 35% rozpočtu projektu. Díky těmto výdajům je možné produkt co nejrychleji zveřejnit i s odpovídající kvalitou.

Nyní je v průměru automatickými testy pokryto 50% testovaného softwaru. Pro DevOps je ale doporučeno, aby organizace dosahovali pokrytí 75-80%. Přesto i dosažení 50% pokrytí je považováno za správný krok pro zvýšení úspěšnosti projektu. [25]

4 Vlastní práce

V praktické části je ve stručnosti probrána současná situace automatizovaného testování softwaru ve firmě Antee.

Dále je představena aplikace IPOAdmin, o jejíž otestováním se praktická část zabývá. Je zanalyzováno, jak bude test probíhat a poté popsány hlavní části automatiziovaného testu.

Na závěr praktické části je rozebrán výstup z Jenkins, a vysvětlení, proč některé buildy skončily neúspešně.

4.1 Situace automatizovaného testování softwaru v Antee

Ve firmě Antee probíhá automatizované testování za pomoci testovacího nástroje Selenium WebDriver. Testy jsou psány v jazyce Java, používají se buď na webu seleniumtests.antee.cz a, nebo v interní aplikaci na webu test.antee.cz.

Testy jsou součástí Maven projektu, který je spouštěn každý den kolem 8 hodiny servisou Jenkins. V Jenkins je možné zkontrolovat výsledky proběhlých testů.

Testy jsou rozděleny do dvou kategorií pro IPOAdmin, což je interní firemní aplikace, a pro IPO, což je redakční systém na webových stránkách, se kterým mají možnost pracovat i klienti firmy.

IPO a IPOAdmin má dle odhadu kolem 800 funkcí, které by bylo možné pokrýt automatizovanými testy. V současnosti jsou testy vytvořeny na necelých 200 funkcí.

Pomocí Selenia jsou zpravidla prováděny pouze testy s tzv. Happy Path. Statisticky není možné automatickými testy pokrýt kompletní funkčnost systému, s ohledem na všechny možné postupy, vstupy a výjimky.

4.2 Představení IPO

IPO je firemní aplikace, která je rozdělena na dvě hlavní části. Základní částí, se kterou pracují pouze zaměstnanci firmy je IPOAdmin. Druhou částí, do které mají přístup i klienti firmy je redakční systém IPO na kterém je postaveno 99% webů od společnosti Antee.

4.2.1 IPOAdmin

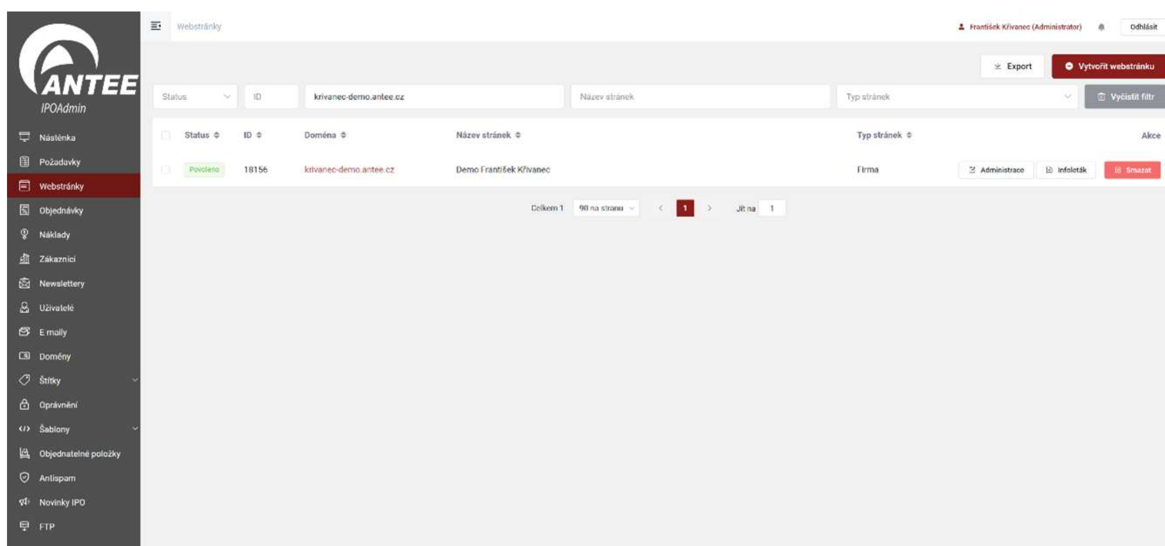
Jedná se o vnitřní firemní aplikaci, ve které probíhá většina firemních agend. Rozdělena je na několik základních sekcí:

- Nástěnka: úvodní stránka uživatele. Je zde vypsaný přehled přidělených požadavků, a přehled nákladů.
- Požadavky: jsou zde všechny aktuální i historické požadavky. Lze vytvářet nové požadavky.
- Webstránky: přehled webstránek, které firma spravuje nebo spravovala. Lze vytvořit nové webstránky, nebo upravovat detaily existujících webstránek.
- Objednávky: přehled všech objednávek. Lze vytvářet nové objednávky a spravovat existující objednávky.
- Náklady: přehled jednotlivých nákladů. Lze vytvářet nové náklady a pracovat s existujícími náklady.
- Zákazníci: přehled zákazníků. Lze vytvářet nové zákazníky, zobrazovat detaily zákazníků a upravovat je.
- Newslettery: přehled newsletterů a vytváření nových.
- Uživatelé: přehled uživatelů IPO. Vytváření nových uživatelů a upravování existujících uživatelů.
- E-maily: přehled existujících emailů pod správou firmy, možnost vytvářet nové a pracovat s existujícími.
- Domény: přehled domén registrovaných pod firmou, možnost registrace nových domén a upravování současných.
- Štítky: přehled existujících štítků k různým sekcím aplikace. Vytváření nových.
- Oprávnění: přehled oprávnění pro uživatele. Možnost přidělení oprávnění různým uživatelským skupinám.
- Šablony: nacházejí se zde různé šablony pro emaily, PDF šablony, šablony stránek a požadavků. Možnost vytváření nových šablon.
- Objednatelné položky: přehled položek, které lze u firmy objednat. Možnost vytvoření nových položek.

- Antispam: přehled zablokovaných IP adres. Možnost přidat nové adresy na seznam, případně odstranění adresy ze seznamu.
- Novinky IPO: přehled novinek v IPO možnost založení nových novinek a upravování existujících novinek.
- FTP: přehled FTP pod správou firmy. Možnost vytvoření nové FTP.

Na obrázku 11 lze vidět vzhled aplikace IPOAdmin.

Obrázek 11 Vzhled aplikace IPOAdmin



Zdroj: vlastní

4.2.2 IPO

Součástí IPOAdmin je i aplikace IPO, která je používána jako redakční systém webových stránek klientů firmy Antee. Zákazníci se tedy běžně setkají pouze s částí IPO, kde mohou provádět úpravy svých stránek přijatelnější formou, než je pouze HTML kód.

4.2.3 Role v IPO

Pro uživatele IPO je v aplikace připraveno několik rolí, které dávají uživatelům oprávnění, ke kterým činnostem v aplikaci mají přístup. Díky tomu se nemusí pro každého uživatele nastavovat oprávnění zvlášť. Oprávnění pro konkrétního uživatele se dají případně i přidat nebo odebrat individuálně.

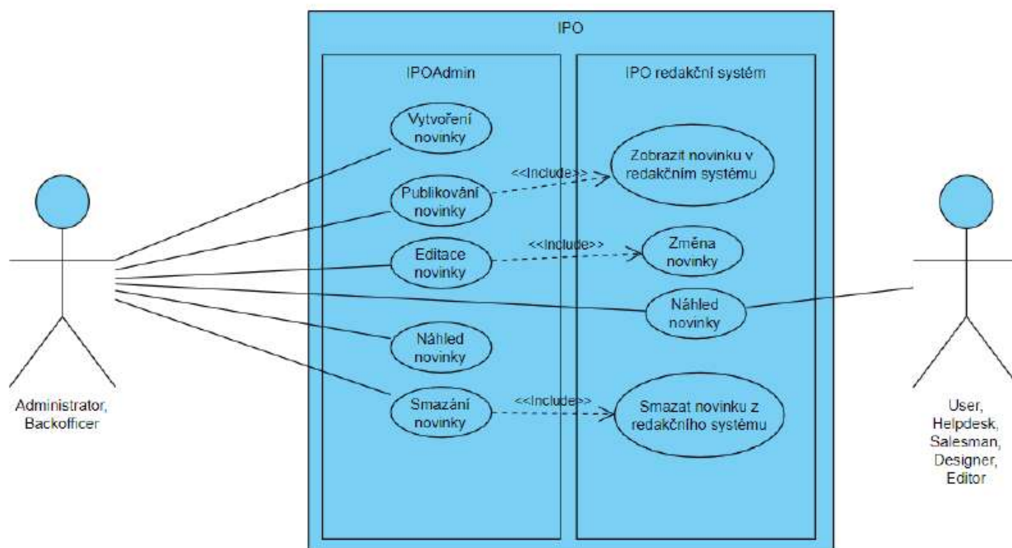
1. Administrator: jediná role která má všechna oprávnění.

2. Helpdesk: tuto roly dostávají zaměstnanci technické podpory. Má přístup k základním sekcím IPOAdmin, a ke všem webstránkám.
3. Backoffice: role s oprávněním pro všechna pokročilá nastavení IPO. Přístup k pokročilým nastavením a sekcím v IPOAdmin. Přístup ke všem webstránkám.
4. Salesman: role pro obchodní zástupce. U webstránek má přístup pouze k takovým, které patří jeho zákazníkům, nebo mu jsou v nastavení přiděleny.
5. Designer: role pro grafiky. Má oprávnění pro nastavování grafiky webů. Má přístup pouze k webům, ke kterým mu byl přístup udělen.
6. User: role pro zákazníky firmy. Lze se přihlásit pouze do části IPO, nikoliv IPOAdmin. Má přístup ke všem základním oprávněním pro ovládání modulů a jejich nastavení.
7. Editor: role pro redaktory stránek. Tyto uživatele může vytvářet a rušit i zákazník s rolí User. Má přístup pouze do IPO a měnit může pouze stránky, ke kterým má přidělený přístup.
8. Guest: role, která v základu nemá žádné oprávnění. Oprávnění se přidají konkrétnímu účtu podle potřeb.

4.3 Identifikace testu

V této práci je řešeno vytvoření testu pro novou funkci publikování novinky v aplikaci IPOAdmin. Jedná se o možnost napsání novinky, která se poté bude zobrazovat v administraci stránek firemním zákazníkům. Oprávnění pro práci s novinkami mají v základu pouze role Administrator a Backofficer. Na obrázku 12 lze vidět use case diagram vytváření novinky. Testem je potřeba zjistit, zda se novinka dá založit dle očekávání, a zda se zobrazí na stránkách kde má. S novinkami se s největší pravděpodobností setkají všichni zákazníci firmy.

Obrázek 12 UseCase diagram



Zdroj: vlastní

Test lze nejprve provést manuálně, nicméně díky jeho automatizaci se dá jednoduše otestovat, zda funkce aplikace bude fungovat správně i po nasazení dalších rozšířeních, které mohou s funkcionalitou přímo i nepřímo souviset.

4.4 Analýza testu

Přehled novinek a jejich vytváření probíhá na záložce „Novinky IPO“, která je vidět na obrázku 13. Novinky je možné si zde vyfiltrovat (dle ID, statusu, názvu, typu stránek, na kterých je zobrazena a modulech u kterých je zobrazena), upravovat, smazat, anebo vytvářet.

Obrázek 13 Stránka Novinky IPO

Novinky IPO / Přehled novinek

František Křivanec (Administrator) Odhlásit

Vytvořit novinku

ID | Status | Název | Typ stránek | Moduly | Vyčistit filtr

ID	Status	Název	Vytvořil	Typ stránek	Moduly	Akce
13	Publikováno	Testovací novinka přímo z api 1660575568551	selenium admin	Všechny typy	Všechny moduly	Upravit Smazat
12	Publikováno	Testovací novinka přímo z api 1660489170767	selenium admin	Všechny typy	Všechny moduly	Upravit Smazat
11	Publikováno	Testovací novinka přímo z api 1660402761251	selenium admin	Všechny typy	Všechny moduly	Upravit Smazat
10	Publikováno	Testovací novinka přímo z api 1660316371224	selenium admin	Všechny typy	Všechny moduly	Upravit Smazat
9	Publikováno	Máme pro Vás nový článek na našem blogu!	Jakub Jonáš	Firma, Hotel/Penzion	Všechny moduly	Upravit Smazat
8	Publikováno	Je váš obecní web připraven (nejen) na nejnovější vyhlášky?	Jakub Jonáš	Obec	Všechny moduly	Upravit Smazat
7	Publikováno	Analýza marketingového potenciálu	Jakub Jonáš	Firma, Hotel/Penzio...	Všechny moduly	Upravit Smazat
6	Koncept	Novinky v modulu Úřední deska	Eduard Beneš	Všechny typy	Úřední deska, Přehl...	Upravit Smazat
5	Koncept	Nový poskytovatel SMS brány	Eduard Beneš	Všechny typy	Rozesílání sms	Upravit Smazat
4	Koncept	Lednové novinky	Eduard Beneš	Všechny typy	Všechny moduly	Upravit Smazat

Celkem 63 10 na stranu < 1 2 3 4 5 6 7 > Jít na 6

© 2022 ANTEE s.r.o. · Nápověda

Zdroj: vlastní

Po kliknutí na tlačítko „Vytvořit novinku“ se zobrazí nová záložka viditelná na obrázku 14, kde se vyplňují podrobnosti o novince a její obsah. Stejná ale již předvyplněná záložka se objeví při kliknutí na možnost upravit.

Obrázek 14 Podstránka pro vytvoření nové novinky

Zdroj: vlastní

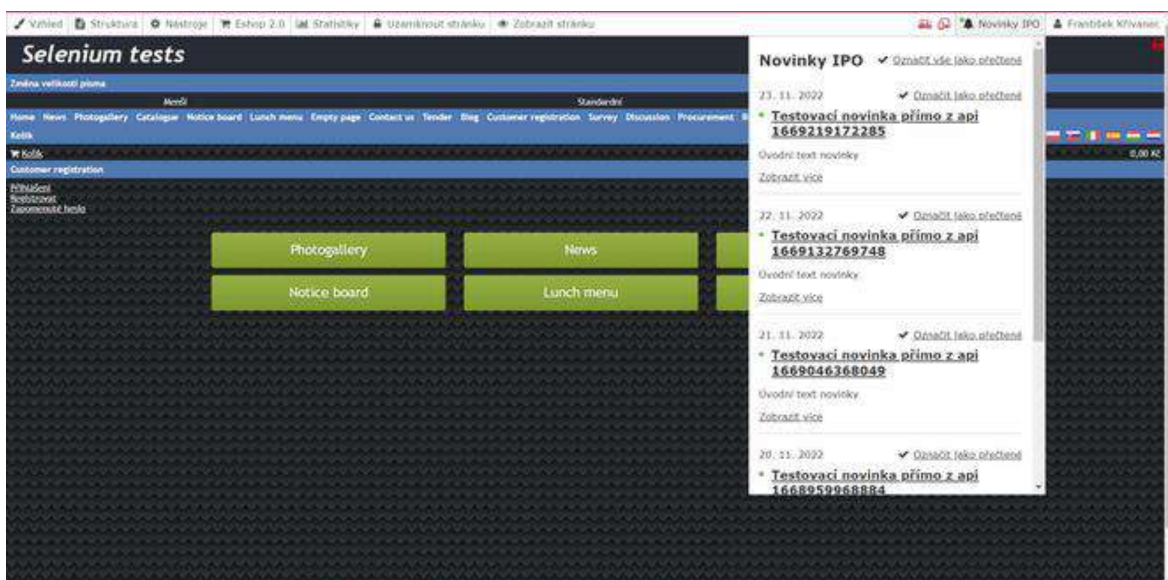
Pro uložení vyplněné novinky je potřeba vyplnit všechna pole.

- Název: jméno novinky, které se zobrazuje v přehledu.
- Úvodní text: stručný popis, který se zobrazuje na stránce před přechodem do detailu novinky
- Typ stránek: šablona stránek na kterých se má novinka zobrazovat (npř. obec, škola, firma…)
- Moduly: novinka se zobrazuje na webových stránkách, které mají konkrétní modul povolený na stránkách (npř. aktuality, úřední deska, fulltextové vyhledávání…)
- Popis: jedná se o vlastní text aktuality, který se zobrazí v detailu aktuality, pokud se ji zákazník rozhodne přečíst.

Novinku je možná buďto rovnou publikovat, nebo uložit jako koncept a publikovat ji až později, případně ji před publikováním ještě pozměnit.

Po publikování by se měla novinka ihned objevit v administraci webových stránek, které splňují zadaná kritéria (obrázek 15).

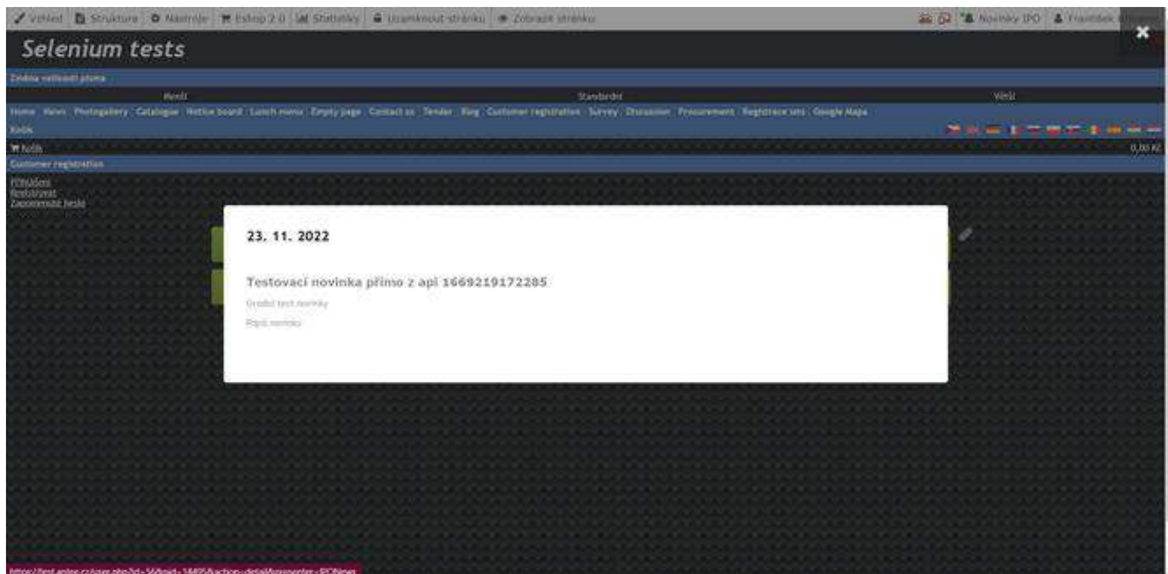
Obrázek 15 Umístění novinek v administraci stránek



Zdroj: vlastní

Novinky se v administraci webových stránek nachází v horní liště pod tlačítkem „Novinky IPO“. Po rozkliknutí se novinka zobrazí v pop up okně (obrázek 16).

Obrázek 16 Pop up okno novinky



Zdroj: vlastní

Novinky, které jsou přečtené se na stránkách již nezobrazují. Novinku lze označit jako přečtenou, aniž by ji zákazník ve skutečnosti četl.

4.5 Testovací scénář

Po zanalyzování průběhu vytváření novinky a kontrolování, zda byla novinka vytvořena je možné sestavit konkrétní testovací scénář s pokrytím všech náležitostí. V kroku musí být popsány kroky, kterými lze testem projít. Potřebná vstupní data a na to i navazující očekáváné výstupy.

1. Přihlášení backofficera do IPOAdmin na testovací stránce.
   Výsledek: backofficer se přihlásí.
2. Uživatel přejde na stránku novinky IPO a dá vytvořit novinku.
   Výsledek: otevře se okno pro vytváření nové novinky.
3. Uživatel vyplní novinku a dá uložit jako koncept.
   Výsledek: koncept se vytvoří, je zobrazena notifikace, že koncept byl uložen.
4. Uživatel vyplní filtr, podle detailů konceptu.
   Výsledek: je vyfiltrován správný koncept.
5. Uživatel otevře koncept a zkontroluje, zda data souhlasí.
   Výsledek: data odpovídají
6. Uživatel změní data v konceptu a opět koncept uloží.
   Výsledek: koncept se uloží, a zobrazí se notifikace, že koncept byl uložen.
7. Uživatel vyfiltruje změněný koncept.
   Výsledek: je vyfiltrován správný koncept.
8. Uživatel otevře koncept, a zkontroluje správnost dat.
   Výsledek: data jsou správně vyplněna.
9. Uživatel klikne na možnost publikovat.
   Výsledek: koncept se publikuje, a je zobrazena správná notifikace.
10. Uživatel vyplní filtr podle detailu publikované novinky.
    Výsledek: vyfiltruje se požadovaná novinka.
11. Uživatel zkontroluje správnost dat v novince.
    Výsledek: data odpovídají poslední změně.
12. Uživatel se odhlásí, přejde na stránky administrace a přihlásí se jako zákazník.
    Výsledek: přihlášení proběhlo úspěšně.

13. Uživatel zkontroluje, že v administraci webu je novinka zobrazena s požadovaným textem.
    Výsledek: novinka se v administraci zobrazuje s požadovaným textem.
14. Uživatel se odhlásí z administrace a přihlásí se jako backofficer v IPOAdmin
    Výsledek: uživatel je příhlášem v IPOAdmin
15. Uživatel přejde na stránku Novinky IPO a vyfiltruje si publikovanou novinku.
    Výsledek: je vyfiltrovaná správná novinka.
16. Uživatel otevře detail publikované novinky, změní ho a uloží.
    Výsledek: novinka se uloží, je zobrazena správná notifikace.
17. Uživatel se odhlásí z IPOAdmin a přejde do administrace, kde se přihlásí jako uživatel.
    Výsledek: uživatel je přihlášen v administraci.
18. Uživatel zkontroluje, zda se novinka změnila.
    Výsledek: novinka je změněna.
19. Uživatel se odhlásí z administrace a přihlásí se jako backofficer v IPOAdmin.
    Výsledek: uživatel je přihlášen v IPOAdmin.
20. Uživatel přejde na stránku Novinky IPO a vyfiltruje si správnou novinku.
    Výsledek: novinka je vyfiltrována
21. Uživatel otevře detail novinky, a novinku smaže.
    Výsledek: novinka se smaže, je zobrazena správná notifikace.
22. Uživatel se odhlásí z IPOAdmin, přejde na stránky a administrace a přihlásí se jako uživatel.
    Výsledek: přihlášení proběhlo úspěšně.
23. Uživatel zkontroluje, že smazaná novinka se již nezobrazuje.
    Výsledek: novinka již není v administraci k dohledání.

4.6 Vytvoření testu

Automatizovaný test je vytvořen v jazyce JAVA, jako součást celého projektu automatizovaných testů pro IPOAdmin. Díky podobnosti s jinými úkony testování (kliknout myší na konkrétní místo, zkontrolovat text na stránce…) a základním principem objektově orientovaného programování je možné mnoho již existujících funkcí z projektu znovu použít, tak aby fungovali přesně pro tento test. Testy probíhají za pomoci nástroje Selenium, konkrétně Selenium WebDriver. Psaní testu probíhalo v IntelliJ IDEA.

V příloze práce se nachází pouze očištěný projekt obsahující třídy a funkce, které jsou přímo využity pro funkčnost vytvořeného testu.

4.6.1 Vytvoření nové větve projektu

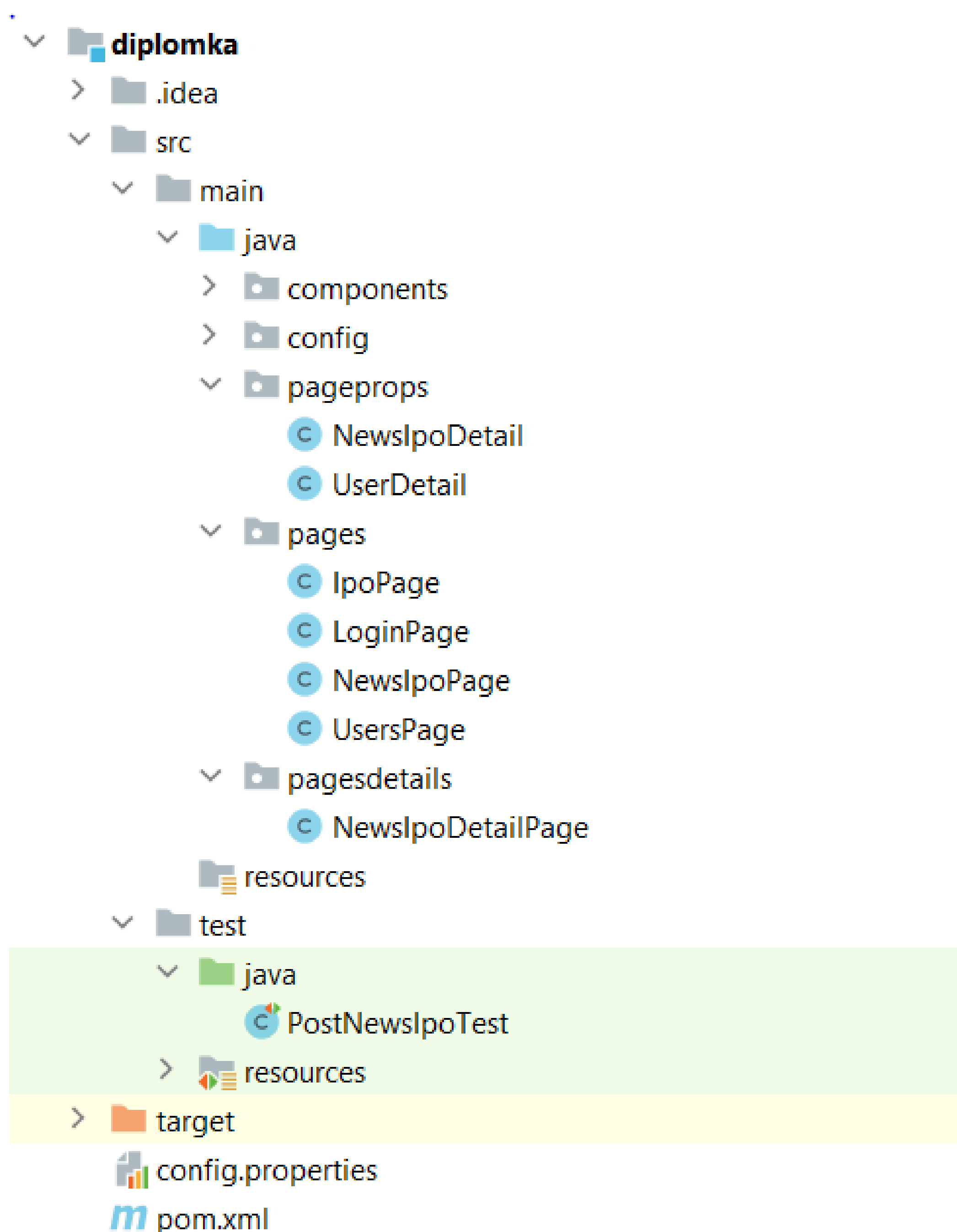
Před započetím práce na novém projektu je potřeba stáhnout aktuální verzi projektu a vytvořit novou lokální větev v GITu, na které budou probíhat změny.

Pomocí příkazu „git fetch" lze ověřit, zda je v lokálním pracovišti otevřena stejná verze projektu, která se nachází ve vzdáleném repozitáři. Pokud se verze liší, je potřeba pomocí „git pull" stáhnout aktuální verzi repozitáře do lokálního pracoviště.

Pomocí příkazu „git branch nazev_vetve" lze vytvořit novou větev v projektu, na které se bude psát nový test. Pokud by projekt dopadl špatně, nebo dokonce rozbil původní funkčnost, je možné celou větev zahodit a vrátit se opět do stavu před započetím práce na projektu.

4.6.2 Struktura projektu

Obrázek 17 Struktura projektu



Zdroj: vlastní

Struktura projektu (obrázek 17) se skládá z několika nepostradatelných souborů pro správnou funkci projektu a souborů samotného projektu.

- .idea: jedná se o soubory, které souvisí s nastavením IDE k danému projektu.
- chrome a chromedriver: soubory, díky kterým lze testy spouštět. Chromedriver ovládá google chrome.
- config.properties: soubor obsahující konfigurační detaily npř. přístupová hesla do aplikace…
- pom.xml: v tomto souboru jsou uloženi informace a konfigurace projektu pro Maven.
- src: zdrojová složka obsahující všechny zdrojové kódy projektu. Ve složce test se nachází třída s testem, ze které se test pouští. Ve složce main se nacházejí všechny třídy, se kterými testy pracují.
    - V package components se nacházejí třídy se základními metodami, jako je kliknutí v prohlížeči. Vyplnění textu do pole. Vybrání možnosti. Vyčkání na odpověď aplikace atp…
    - V package config se nacházejí podrobnosti o nastavení prohlížeče.
    - V package pageprops se nacházejí třídy instancí v tomto projektu konkrétně instance uživatelů, a instance novinky. Instance uživatelů obsahuje npř. jméno a přihlašovací údaje, instance novinky zase text novinky, její název atp…
    - V package pages jsou třídy, obsahující metody, které se provádějí při testech na jednotlivých podstránkách aplikace. Metody uložené ve třídě NewsIpoPage budou tedy při testu používány pouze tehdy, probíhá-li test v aplikaci na podstránce Novinky IPO.
    - V package pagesdetails je třída NewsIpoDetailPage. Package funguje podobně jako předchozí, jen třídy v této package obsahují metody, které jsou používány až na stránce detailu konkrétní novinky.

4.6.2.1 Třída NewsIpoDetail

Jedná se o třídu, podle které se tvoří jednotlivé instance novinek IPO. Třída se tedy skládá z jednotlivých proměnných, které instance novinky využívá, a z metod, které s těmito proměnnými pracují. Jedná se o konstruktor, gettery a settery.

Proměnné použité ve třídě jsou následovné:

```
private final String nazev;

private final String uvodniText;

private final String popis;

private final List<String> typStranek;

private final List<String> moduly;

private final NewsIpoPage.Status status;

private final String notifikace;
```

Proměnné typu String v sobě uchovávají pouze textové informace. Dle názvu lze odvodit o které informace se jedná.

Proměnná typu List<String> v sobě může obsahovat více textových řetězců. Je to z důvodu, protože lze mít více typů stránek i modulů.

Proměnná typu NewIpoPage.Status v sobě uchovává instance ze třídy NewIpoPage. Jedná se o proměnnou enum.

```
public enum Status {
    KONCEPT("Koncept","info"), PUBLIKOVANO("Publikováno","primary"),
SMAZANO("Smazáno","");
    private final String name;
    private final String createBtnClass;

    Status(String name, String createBtnClass) {
        this.name = name;
        this.createBtnClass = createBtnClass;
    }

    public String getName() {
        return name;
    }

    public String getCreateBtnClass() {
        return createBtnClass;
    }
}
```

}

Z kódu je patrné, že proměnná má předdefinované 4 instance. KONCEPT, PUBLIKOVANO a SMAZANO. Každá z instancí v sobě obsahuje další 2 proměnné typu String, ve kterých je uloženo jméno statusu potřebné pro filtrování, a třída tlačítka v aplikaci, na které se bude v průběhu testu klikat.

Konstruktor třídy vypadá následovně:

```
private NewsIpoDetail(String nazev, String uvodniText, String popis,
List<String> typStranek, List<String> moduly, NewsIpoPage.Status status,
String notifikace) {
    this.nazev = nazev;
    this.uvodniText = uvodniText;
    this.popis = popis;
    this.typStranek = typStranek;
    this.moduly = moduly;
    this.status = status;
    this.notifikace = notifikace;
}
```

Lze vidět že konstruktoru jdou všechny jeho proměnné jako parametry, a díky přiřazení this.proměnná = proměnná se přiřadí konkrétní parametr proměnné do správné proměnné dané instance. Pokud by nebylo použito this, měnila by se proměnná celé třídy, nikoliv však instance.

Ke každé proměnné je vytvořen getter, pomocí kterého lze získat hodnotu proměnné v instanci. Takto vypadá getter pro vrácení názvu novinky:

```
public String getNazev() {
    return nazev;
}
```

Dále je ve třídě vytvořena metoda builder, jejíž voláním se dají vytvářet instance, případně pokud je jí instance dána jako parametr, lze dalšími metodami pouze měnit některé proměnné. Hlavní části buildru jsou:

```
public Builder() {
}

public Builder(NewsIpoDetail newsIpoDetail) {
    this.nazev = newsIpoDetail.getNazev();
    this.uvodniText = newsIpoDetail.getUvodniText();
    this.popis = newsIpoDetail.getPopis();
    this.typStranek = newsIpoDetail.getTypStranek();
    this.moduly = newsIpoDetail.getModuly();
    this.status = newsIpoDetail.getStatus();
    this.notifikace = newsIpoDetail.getNotifikace();
}

public NewsIpoDetail build() {
    return new NewsIpoDetail(nazev, uvodniText, popis, typStranek,
moduly, status, notifikace);
}
```

```
public NewsIpoDetail.Builder withNazev(String nazev) {
    this.nazev = nazev;
    return this;
}
```

Metoda Builder() se volá pokud se vytváří první instance. Poté se na metodu přidávají další metody stylem Builder().withNazev(„String“). Metody withProměnná() jsou vytvořeny pro všechny proměnné třídy, znázorněna je jen jedna. Aby se instance vytvořila je potřeba zakončit metodou build() výsledné volání pro vytvoření instance kde název bude „Název novinky“ by vypadalo takto:

```
NewsIpoDetail novinka = new NewsIpoDetail().Builder().withNazev("Název
novinky").build();
```

Vznikla by tedy nová instance třídy NewsIpoDetail, která by obsahovala pouze název této novinky.

Dále lze volat metodu Builder(NewsIpoDetail newsIpoDetail) {…}, která si vezme za parametr již existující instanci. Proměnné v nové instanci budou mít stejnou hodnotu jako instance předaná v parametru, a pokud je potřeba některé proměnné změnit přidají se v metodě withProměnná(parametr). Pokud by se tedy vytvářela instance se stejnými proměnnými, které měla původní instance novinka, ale lišila by se v názvu lze to udělat voláním.

```
NewsIpoDetail novinka2 = new NewsIpoDetail(novinka).Builder().withNazev("
Jiný název").build();
```

4.6.2.2 Třída PostNewsIpoTest

Jedná se o třídu, ve které se nachází konkrétní testy. Dědí atributy a metody z třídy BaseTest, která se nachází v package components. Proměnné ve třídě jsou:

```
private LoginPage loginPage;

private NewsIpoPage newsIpoPage;

private NewsIpoDetail newsIpoDetailConcept;

private NewsIpoDetail newsIpoDetailConceptChanged;

private  NewsIpoDetail newsIpoDetailPublished;

private  NewsIpoDetail newsIpoDetailPublishedChanged;

private  NewsIpoDetail newsIpoDetailDeleted;
```

Většina proměnných je vyčleněna pro instance třídy NewsIpoDetail a jsou v nich tedy uloženy podrobnosti o novinkách. Dále jsou tu proměnné představující třídu NewsIpoPage a LoginPage pomocí kterých se dá přistoupit k metodám, které tyto třídy obsahují.

Další částí třídy je inicializace proměných:

```
@Parameters({"nazev", "uvodniText", "typStranek", "moduly", "popis",
"notifikace"})
@BeforeClass
public void init(@Optional("Nové vylepšení přidávaní fotek do galerie ")
String nazev,
                 @Optional("Připravili jsme pro Vás vkládání obrazků jako
z příštího století") String uvodniText,
                 @Optional("Všechny typy stránek") String typStranek,
                 @Optional("Všechny moduly") String moduly,
                 @Optional("Fotky můžete nyní vkládat mnohem rychleji a
efektivněji stačí postupovat podle přiloženého návodu.") String popis,
                 @Optional("vytvořena") String notifikace
) {
    loginPage = new LoginPage(driver);
    String timeStamp = TimeStamp.getTimeStamp();
    newsIpoPage = new NewsIpoPage(driver);
    newsIpoDetailConcept = new NewsIpoDetail.Builder()
            .withNazev(nazev + timeStamp)
            .withUvodniText(uvodniText)
            .withTypStranek(typStranek)
            .withModuly(moduly)
            .withPopis(popis)
            .withPublished(NewsIpoPage.Status.KONCEPT)
            .withNotifikace(notifikace)
            .build();
    newsIpoDetailConceptChanged = new
NewsIpoDetail.Builder(newsIpoDetailConcept)
            .withNazev("Aktualizovaný " + nazev + timeStamp)
            .withUvodniText("Aktualizovaný " + uvodniText)
            .withPopis("Aktualizovaný " + popis)
            .withNotifikace("uložena")
            .build();
    newsIpoDetailPublished = new
NewsIpoDetail.Builder(newsIpoDetailConceptChanged)
            .withPublished(NewsIpoPage.Status.PUBLIKOVANO)
            .build();
    newsIpoDetailPublishedChanged = new
NewsIpoDetail.Builder(newsIpoDetailPublished)
            .withNazev("Po publikování pozměněný " + nazev + timeStamp)
            .withUvodniText("Po publikaci pozměněný " + uvodniText)
            .withPopis("Po publikaci pozměněný " + popis)
            .build();
    newsIpoDetailDeleted = new
NewsIpoDetail.Builder(newsIpoDetailPublishedChanged)
            .withPublished(NewsIpoPage.Status.SMAZANO)
            .build();
}
```

Dosazování do instancí třídy NewsIpoDetail probíhá pomocí builderů z minulé kapitoly. Je zde názorně vidět, jak byla vytvořena první instance dosazením všech hodnot, a následná instance byla tvořena z té původní a pouze se měnili požadované proměnné.

Poslední částí třídy je volání samotných testů:

```
@Test
public void login() {
    Assert.assertTrue(loginPage.loginSeleniumAdmin());
}

@Test(priority = 3)
public void createNewsIpoConcept() {
    Assert.assertTrue(newsIpoPage.createNewsIpo(newsIpoDetailConcept));
    Assert.assertTrue(newsIpoPage.isNewsOK(newsIpoDetailConcept));
}

@Test(priority = 4)
public void changeNewsIpoConcept() {
    Assert.assertTrue(newsIpoPage.changeNewsConcept(newsIpoDetailConcept,
newsIpoDetailConceptChanged));
    Assert.assertTrue(newsIpoPage.isNewsOK(newsIpoDetailConceptChanged));
}

@Test(priority = 5)
public void publishNewsIpo() {

Assert.assertTrue(newsIpoPage.publishNewsIpo(newsIpoDetailConceptChanged)
);
    Assert.assertTrue(newsIpoPage.isNewsOK(newsIpoDetailPublished));
}

@Test(priority = 7)
public void checkNewsIpoPublished() {
    Assert.assertTrue(newsIpoPage.checkNewsIpo(newsIpoDetailPublished));
}

@Test(priority = 8)
public void changeNewsIpoPublished() {

Assert.assertTrue(newsIpoPage.changeNewsConcept(newsIpoDetailPublished,
newsIpoDetailPublishedChanged));

Assert.assertTrue(newsIpoPage.isNewsOK(newsIpoDetailPublishedChanged));
}

@Test(priority = 9)
public void checkChangedNewsIpoPublished() {

Assert.assertTrue(newsIpoPage.checkNewsIpo(newsIpoDetailPublishedChanged)
);
}

@Test(priority = 10)
public void deleteNewsIpo() {

Assert.assertTrue(newsIpoPage.deleteNewsIpo(newsIpoDetailPublishedChanged
```

```
));

Assert.assertTrue(newsIpoPage.isNewsDeleted(newsIpoDetailPublishedChanged
));
}

@Test(priority = 11)
public void checkNewsIpoDeleted() {
    Assert.assertTrue(newsIpoPage.checkNewsIpo(newsIpoDetailDeleted));
}
```

Z volaných metod je vidět, že některé testy probíhají víckrát, pouze s jiným parametrem. Je to proto, že je potřeba některé kontroly provést víckrát, nebo na základě parametrů chtít pouze malý rozdíl ve funkci metody, než měla původně. Díky možnosti zavolat metodu znovu namísto jejího opětovného vytváření je ušetřeno spoustu času a možnosti udělat chybu. Především při pozdější úpravě kódu, kdy by se mohlo stát, že nebudou opraveny všechny funkce shodně.

4.6.2.3 Test přihlášení

První test, který probíhá je, zda se povede správně přihlásit do aplikace IPOAdmin. Poté, co se spustí prohlížeč, je metodou navigator.goToBasicIpoadminTestUrl() přesměrován na stránku test.antee.cz, kde funguje aplikace napojená na testovanou databázi. Pokud by se na stránku nedostal napoprvé, je zde nastaveno, aby se o přístup pokusil ještě párkrát, a poté až případně test vyhodnotil jako neúspěšný.

V okamžiku, kdy se již test dostane na požadovanou stránku, je spuštěna metoda, která si získá z přihlašovací údaje z konfiguračního souboru, a doplní je do příslušných polí. Jako poslední metoda nalezne tlačítko pro přihlášení, klikne na něj a dle úspěchu této akce vyhodnotí celý test.

4.6.2.4 Test vytvoření novinky

Jedná se o druhý test, který navazuje na přihlášení. Skládá se z dvou metod, které jako svůj parametr mají instanci třídy NewsIpoDetail:

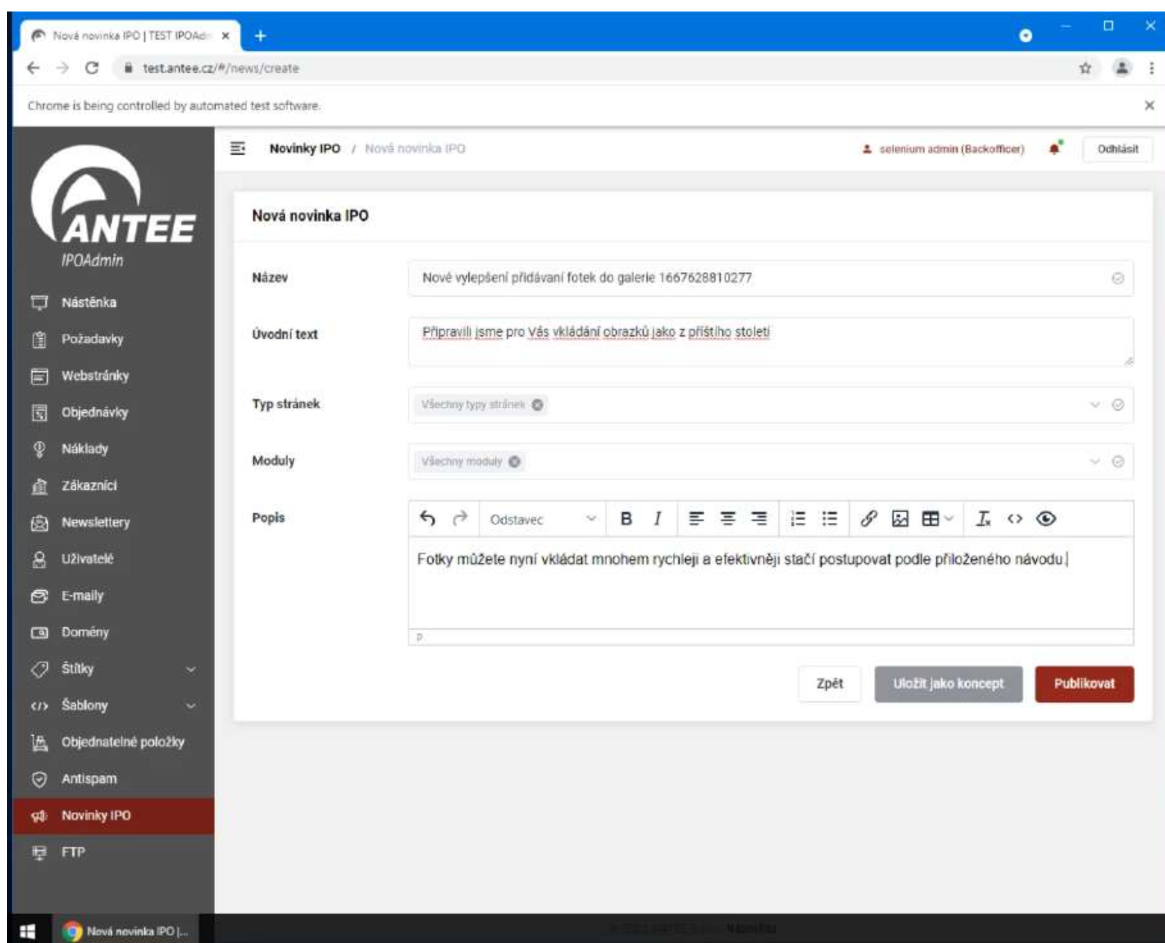
1. newsIpoPage.createNewsIpo(NewsIpoDetail newsIpoDetail) pro vytvoření nové novinky.
2. newsIpoPage.IsNewsOK(NewsIpoDetail newsIpoDetail) pro zkontrolování, zda se novinka uložila tak jak měla.

První metoda začíná ověřením, zda se dá dostat na stránku novinek. Pokud se na stránku novinek dostane klikne na tlačítko vytvořit novinku, a jako návratovou hodnotu vrací spuštění metody, která má vyplnit novou novinku fillOrChangeNews(NewsIpoDetail newsIpoDetail).

```
public boolean fillOrChangeNews(NewsIpoDetail newsIpoDetail) {
    String nazev = newsIpoDetail.getNazev();
    String uvodniText = newsIpoDetail.getUvodniText();
    String popis = newsIpoDetail.getPopis();
    List<String> typStranek = newsIpoDetail.getTypStranek();
    List<String> moduly = newsIpoDetail.getModuly();
    String notifikace = newsIpoDetail.getNotifikace();
    NewsIpoPage.Status status = newsIpoDetail.getStatus();
    if (nazev != null) {
        fillNthInput(1,nazev);
    }
    if (uvodniText != null) {
        fillNthInputTextarea(2, uvodniText);
    }
    if (typStranek != null && !"Publikováno".equals(status.getName())) {
        if
(wait.isElementInDOM(By.xpath("//div[label[contains(text(),'Typ
stránek')]]//div[contains(@class,'el-
select__tags')]//span[contains(text(),'" + typStranek.get(0) + "')]"), 1)
&& wait.isElementInDOM(By.xpath("//div[label[contains(text(),'Typ
stránek')]]//div[contains(@class,'el-
select__tags')]//span//i[contains(@class,'el-icon-close')]"), 1)){
            clicker.click(By.xpath("//div[label[contains(text(),'Typ
stránek')]]//div[contains(@class,'el-
select__tags')]//span//i[contains(@class,'el-icon-close')]"));

optionSelector.clickMultipleSelect(typStranek,"//div[label[contains(text(
),'Typ stránek')]]");
        } else {

optionSelector.clickMultipleSelect(typStranek,"//div[label[contains(text(
),'Typ stránek')]]");
        }
    }
    if (moduly != null && !"Publikováno".equals(status.getName())) {

optionSelector.clickMultipleSelect(moduly,"//div[label[contains(text(),'M
oduly')]]");
    }
    if (popis != null) {
        fillInputPopis(popis);
    }
    clicker.click(By.cssSelector("div.el-card__body > div:nth-child(2) >
button.el-button.el-button--" +
newsIpoDetail.getStatus().getCreateBtnClass() + " >
span"),By.cssSelector("div.el-notification"));
    return notificationRecognizer.notification("Novinka IPO byla úspěšně
" + notifikace + ".");
}
```

Metoda na vyplnění novinky si nejprve inicializuje jednotlivé proměnné z předaného parametru např. pro název String nazev = newsIpoDetail.getNazev();, a poté, pokud mají proměnné nějakou hodnotu, spustí metody, které dokáží hodnoty vepsat do jednotlivých inputů např. pro název if (nazev != null) {fillNthInput(1,nazev);} (v if je podmínka, zda opravdu proměnná nazev obsahuje nějaký text, a poté metoda fillNthInput(1,nazev) doplní text z proměnné do prvního vstupu.) (obrázek 18).

Obrázek 18 Náhled vytvoření novinky



Zdroj: vlastní

Poté, co jsou jednotlivé inputy vyplněny, si metoda nalezne tlačítko pro uložení konceptu novinky a klikne na něj clicker.click(By.*cssSelector*("div.el-card__body > div:nth-child(2) > button.el-button.el-button--" + newsIpoDetail.getStatus().getCreateBtnClass() + " > span"),By.*cssSelector*("div.el-notification"));. Test je vyhodnocen jako úspěšný, pokud se na obrazovce zobrazí

požadovaná notifikace return notificationRecognizer.notification("Novinka IPO byla úspěšně " + notifikace + ".");.

Druhá metoda tohoto testu začíná na stránce Novinky IPO. Výsledek testu závisí na výsledcích dvou metod. Nejprve je filtr vyresetován a poté vyplněn pomocí proměnných, které se vyplňovali do novinky, a zkontrolováno zda se vyfiltrovali novinky se správnými parametry, pro které byly filtrovány if(nazev!=null) { LogSelenium.*debug* ("Filtering nazev:" + nazev); filterHandler.doFilter (3, nazev); if (controlResults && !tableBasic.isCorrectCellContentsDisplayedOnPage (3, nazev)) return false;}. Díky použití TimeStamp.getTimeStamp v začátku testu, při inicializování proměnných v instanci nové novinky by měl být při každém testu vytvořen unikátní název, a proto by měla být vždy vyfiltrována právě jedna novinka.

```
public boolean filter(NewsIpoDetail newsIpoDetail, boolean resetFilter,
boolean controlResults) {
    String nazev = newsIpoDetail.getNazev();
    List<String> typStranek = newsIpoDetail.getTypStranek();
    List<String> moduly = newsIpoDetail.getModuly();
    NewsIpoPage.Status status = newsIpoDetail.getStatus();
    if (!navigator.goToMenuPage(Navigator.Page.NOVINKY_IPO)) {
        return false;
    }
    navigator.refreshIpoadminPage(Navigator.Page.NOVINKY_IPO);
    if (resetFilter)
        filterHandler.resetFilter();
    if (status != null) {
        LogSelenium.debug("Filtering status:" + status);
        filterHandler.doSelectClickFilter("Status", status.getName());
        clicker.clickNeutralPlace();
        if(controlResults &&
!tableBasic.isCorrectCellContentsDisplayedOnPage(2, status.getName()))
            return false;
    }
    if (nazev != null) {
        LogSelenium.debug("Filtering nazev:" + nazev);
        filterHandler.doFilter(3, nazev);
        if(controlResults &&
!tableBasic.isCorrectCellContentsDisplayedOnPage(3, nazev))
            return false;
    }
    if (typStranek != null) {
        LogSelenium.debug("Filtering typStranek:" + typStranek);
        filterHandler.clickMultipleSelect(4,typStranek);
    }
    if (moduly != null) {
        LogSelenium.debug("Filtering moduly:" + moduly);
        filterHandler.clickMultipleSelect(5,moduly);
        return !controlResults ||
tableBasic.isCorrectCellContentsDisplayedOnPage(6, moduly);
    }
```

```
    return true;
}
```

Po vyfiltrování, a zkontrolování výsledků filtrování je otevřen detail novinky, který je nyní potřeba ověřit. To probíhá zkontrolováním textů v polích název, úvodní text a popis.

```
public boolean isDetailOk(NewsIpoDetail newsIpoDetail) {
    String nazev = newsIpoDetail.getNazev();
    String uvodniText = newsIpoDetail.getUvodniText();
    String popis = newsIpoDetail.getPopis();

wait.isElementInDOM(By.xpath("//div[contains(@class,'breadcrumb')]//span[
contains(text(),'Detail novinky IPO')]"));
    if (nazev != null)
        if (!isInputContains("Název", "input", nazev))
            return false;
    if (uvodniText != null)
        if (!isInputContains("Úvodní text", "textarea", uvodniText))
            return false;
    if (popis != null)
        return isIframeContains(popis);
    return true;
}
```

4.6.2.5 Test změny konceptu novinky

Test je složen z dvou metod kde:

1. newsIpoPage.changeNewsConcept(NewsIpoDetail newsIpoDetailConcept, NewsIpoDetail newIpoDetailChanged) slouží ke změnění textů v konceptu.
2. newsIpoPage.IsNewsOK(NewsIpoDetail newsIpoDetail) slouží ke zkontrolování, zda se koncept opravdu změnil. Stejná metoda byla i v předchozím testu, kde je popsaná.

První metoda je složena ze dvou metod, které byly již použity v předchozím testu. Vrací návratovou hodnotu nejprve metody, kde se vyfiltruje metoda podle detailů prvního parametru, a poté se provede metoda, která byla použita pro vyplnění novinky, nicméně nyní původní hodnoty přepíše na hodnoty, které byly uloženy v druhém parametru.

Druhá metoda testu probíhá stejně, jako druhá metoda předchozího testu pouze má kontrolní parametry změněné na ty, které by měli být v konceptu nově uloženy.

4.6.2.6 Test publikování novinky

Test je složen ze dvou metod:

1. newsIpoPage.publishNewsIpo(NewsIpoDetail newsIpoDetail) slouží k publikování konceptu.
2. newsIpoPage.isNewsOK(NewsIpoDetail newsIpoDetail) slouží ke zkontrolování, zda je novinka publikováná v aplikace IPOAdmin se správnými parametry.

První metoda je složena z dvou podmetod, kde první z nich slouží k vyfiltrování a otevření detailu novinky. Druhá podmetoda slouží ke stisknutí tlačítka publikovat, a následně zkontroluje, zda se na obrazovce objeví správná notifikace.

Druhá metoda je stejná jako druhá metoda předchozích dvou testů, pouze využívá jiných parametrů při filtrování a kontrole.

4.6.2.7 Test kontroly publikované novinky

V tomto testu se vyhodnocuje jediná metoda, která má za úkol zjistit, zda je novinka správně publikována v administraci webu. Metoda začíná odhlášením aktuálního uživatele, a poté se přihlásí na webu pro klientskou administraci webů pod účtem uživatele.

```
public boolean checkNewsIpo(NewsIpoDetail newsIpoDetail) {

usersPage.loginDifferentUser(UserDetail.User.SELENIUM_USER.getUserDetail(
));

wait.isElementInDOM(By.xpath("//ul[contains(@class,'ipo')]//span[contains
(text(),'Novinky IPO')]"));
    return ipoPage.checkNewsIpo(newsIpoDetail) &&
usersPage.loginDifferentUser(UserDetail.User.SELENIUM_USER.getUserDetail(
), UserDetail.User.SELENIUM_BACKOFFICER.getUserDetail());
}
```

V uživatelské administraci nejprve zkontroluje, zda se nachází v horní liště pole „Novinky IPO“, a poté spouští metodu, která novinky otevře a skontruje, zda se zde nachází testem vytvořená novinka podle názvu a popisu novinky.

```
public boolean checkNewsIpo(NewsIpoDetail newsIpoDetail) {
    By ipoNewsNazev =
By.xpath("//div[contains(@id,'ipopage')]//div[contains(@class,'ipo-news-
content')]//div[contains(@class,'ipo-news-item')]//h2");
    By ipoNewsPopis =
By.xpath("//div[contains(@id,'ipopage')]//div[contains(@class,'ipo-news-
content')]//div[contains(@class,'ipo-news-item')]//p");

clicker.click(By.xpath("//ul[contains(@class,'ipo')]//span[contains(text(
),'Novinky IPO')]"));
    clicker.click(By.xpath("//div[contains(@class,'ipo-news-
footer')]//a[contains(text(),'Zobrazit starší novinky')]"));
```

```
    if (newsIpoDetail.getStatus().equals(NewsIpoPage.Status.PUBLIKOVANO))
{
        return wait.isAnyElementContaining(ipoNewsNazev,
newsIpoDetail.getNazev()) &&
                wait.isAnyElementContaining(ipoNewsPopis,
newsIpoDetail.getPopis());
    } else {
        return wait.areNoElementsContaining(ipoNewsNazev,
newsIpoDetail.getNazev()) &&
                wait.areNoElementsContaining(ipoNewsPopis,
newsIpoDetail.getPopis());
    }
}
```

Pokud vše proběhne v pořádku tak je test ještě zakončen odhlášením, a přihlášením se zpět na účet administrátora.

4.6.2.8 Test změny publikované novinky

Test skládající se ze dvou metod:

1. newsIpoPage.changeNewsConcept(NewsIpoDetail newsIpoDetail, NewsIpoDetailChanged newsIpoDetailChanged) metoda která změní obsah publikované novinky.
2. newsIpoPage.isNewsOK(NewsIpoDetail newsIpoDetail) metoda kontrolující, zda se novinka změnila do správného tvaru.

Tento test probíhá naprosto stejně jako test změny konceptu. Jediným rozdílem v těchto testech jsou hodnoty, se kterými metody pracují.

4.6.2.9 Test smazání novinky

Test obsahující dvě metody:

1. newsIpoPage.deleteNewsIpo(NewsIpoDetail newsIpoDetail) která slouží k vyfiltrování a smazání novinky.
2. newsIpoPage.isNewsDeleted(newsIpoDetail newsIpoDetail) která zkontroluje, zda se novinka opravdu v aplikaci IPOAdmin smazala.

První metoda má za úkol vyfiltrovat vytvořenou novinku, poté otevře detail novinky, klikne na tlačítko smazat odsouhlasí upozornění, zda si opravdu přejeme novinku smazat, a poté zkontroluje, zda se objevila správná notifikace.

```
public boolean deleteNewsIpo(NewsIpoDetail newsIpoDetail) {
    filter(newsIpoDetail, true);
    return clicker.click(By.xpath("//tr[contains(@class,'el-
```

```
table__row')]//span[contains(text(),'Smazat')]"),
By.xpath("//div[contains(@class,'el-message-
box__message')]//p[contains(text(),'Opravdu chcete odstranit
novinku?')]")) &&
            clicker.click(By.xpath("//div[contains(@class,'el-message-
box')]//span[contains(text(),'OK')]")) &&
            notificationRecognizer.notification("Novinka byla smazána.");
}
```

Druhá metoda se pokusí novinku znovu vyfiltrovat, a jako kontrolu bere, že se ve výsledném výpisu novinek nic nezobrazí.

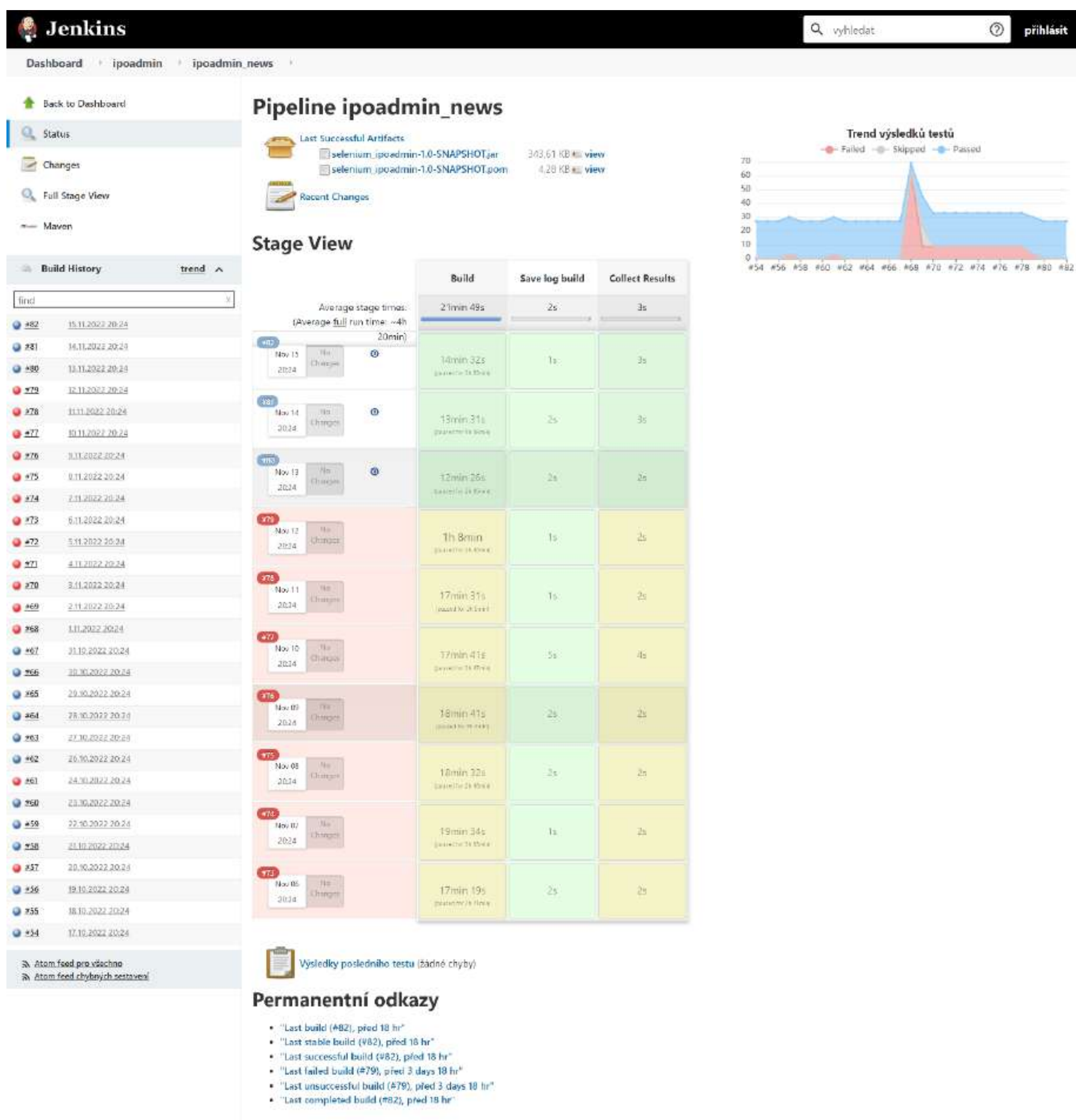
4.6.2.10 Test smazání novinky z administrace

Tento test probíhá stejně, jako kontrola, zda se novinka v administraci zobrazuje. Rozdílem je, že namísto toho, aby hledala že se dané informace z novinky v administraci vyskytují, tak kouká, jestli se náhodou dané informace na webu nevyskytují. Pokud novinka v administraci není, metoda se přihlásí zpět na administrátora a test vyhodnotí jako úspěšný.

4.7 Monitorování testu

K monitorování automatických testů je používán Jenkins. Jenkins se stará o jejich automatické spouštění, a uchovává v sobě historii průběhu testů a případné další detaily o průběhu testu. V následujícím obrázku je vidět shrnutí testů za poslední měsíc.

Obrázek 19 Jenkins přehled posledních buildů



Zdroj: vlastní

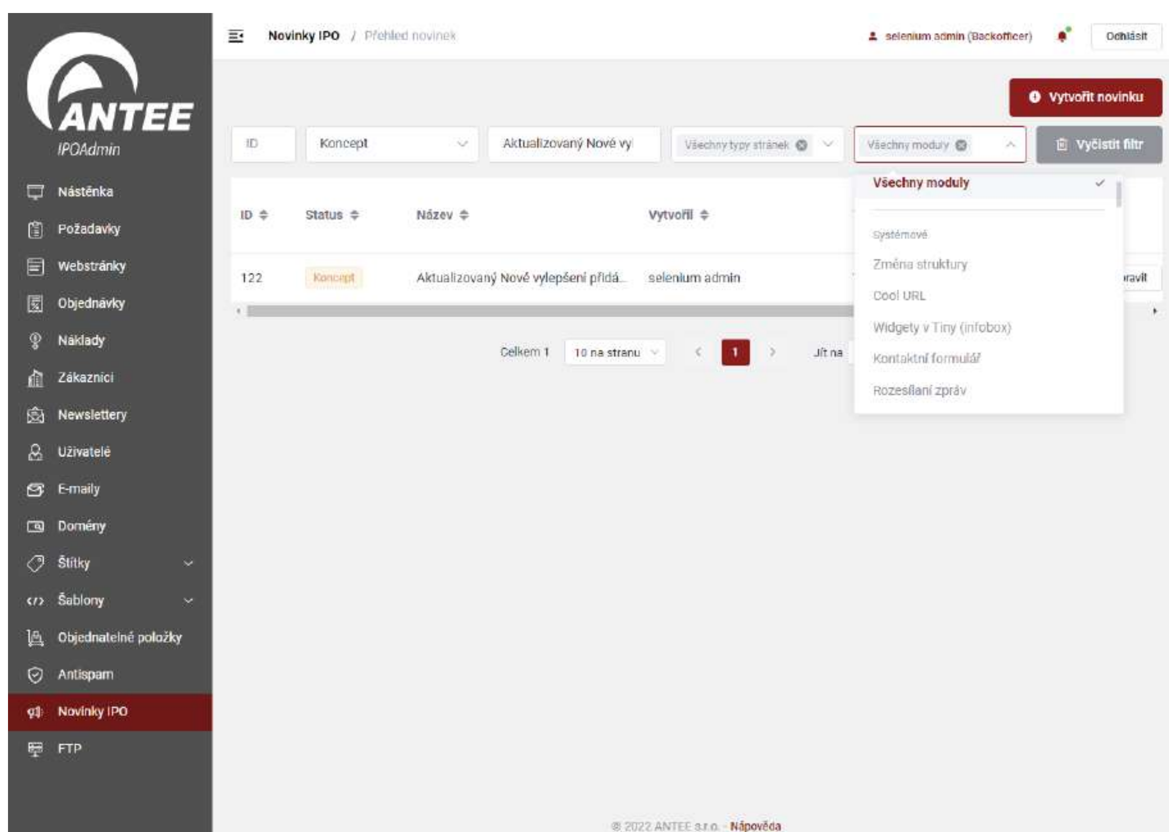
V levé straně obrázku je vidět historie buildů. Červeně označené buildy jsou takové, které skončili s výstupovou hodnotou failed. V dlouhé řadě selhaných testů bylo deset posledních buildů sice úspěšně proběhlých, nicméně na základě chyby z předchozího dne zůstal na stránce fragment, kvůli kterému byly testy vyhodnoceny špatně.

V historii buildů lze také vyčíst, že testy se spouští každý den ve 20:24. Průměrně test trvá 4 hodiny. Samotný build však pouze 21 minut. Nicméně výpočet se počítá z posledních deseti buildů, kde převládají chybné buildy, které trvali kolem 18 minut jeden dokonce přes hodinu. Úspěšné buildy trvají kolem 14 minut.

4.7.1 Chyba v buildu 57

Po kontrole logů chyby byl vyhodnocen závěr testu, že se buďto špatně vygeneroval objekt v HTML aplikace, nebo ho Selenium nemohlo nalézt. Dle logů Selenium nenašlo objekt s touto XPath cestou: „//span[contains(@class,'el-select__tags-text') and contains(text(),'Všechny moduly')]“. Poté, co ho objekt nemohl minutu najít v DOM stránky ukončil test jako neúspěšný. Nicméně, podle screenshotu okna (obrázek 20), který se dělá, pokud dojde k chybě se objekt na stránce nacházel. Zřejmě se jen něco v cestě špatně vygenerovalo.

Obrázek 20 Screenshot chyby v buildu 57



Zdroj: vlastní

4.7.2 Chyba v buildu 68

Chyba nastala z důvodů špatné role testovacího uživatele. Zřejmě se prováděli nějaké změny v databázi testovacího serveru, a testovací administrátorský účet měl

namísto role backofficer roli helpdesk. Tato role neumožňuje novinky vytvářet, a ani nemá přístup na podstránku novinek v aplikaci. V následujícím obrázku lze v horním rohu vidět špatnou roli uživatele.

Obrázek 21 Screenshot chyby v buildu 68

Zdroj: vlastní

5 Výsledky a diskuse

V rámci diplomové práce byla řešena problematika řešení kvality softwaru jeho testování, především za použití automatizovaných testů.

Teoretická část práce byla zaměřena na seznámení se s testováním po teoretické stránce. Byla zde uvedeny definice vysvětlující kvalitu softwaru. Dále byly uvedeny nejdůležitější termíny a pojmy, a uvedeny rozdíly mezi nimi. Byly zde uvedeny fáze životního cyklu vývoje softwaru jak z pohledu tradičních způsobů, tak i modernějších a v dnešní době více využívaných agilních postupech. Poté se práce více do hloubky zabývala samotnými principy a metodami testování. Závěr teoretické části byl věnován nástrojům, které jsou buďto přímo potřeba, nebo usnadňují vytvoření automatizovaných testů. Konkrétně se jednalo o seznámení s programovacím jazykem Java, který sloužil k napsaní automatizovaného testu. Seznámení se s nástroji Selenium, díky kterým je možná automatizované testy spouštět. Dále to byly nástroje, které usnadňují práci s programovým kódem. Jednalo se o představení několika IDE, díky kterým je možné psát kód rychleji, přehledněji a v mnoha případech od něj můžeme dostat radu kde v kódu se nachází chyba nebo jak ho usnadnit. Druhým nástrojem byl program pro verzování GIT, který slouží k uchovávaní jednotlivých verzí kódu. Přehledně tedy uloží kód po každé změně, díky čemuž je možné se ke všem verzím jednoduše vracet, nebo zjistit v čem se verze lišší.

Praktická část práce se zabývala samotným návrhem a implementací automatizovaného testu pro aplikaci IPOAdmin. Nejprve byl proveden krátký rozbor současné situace testování softwaru. Poté byla představena aplikace IPOAdmin spolu s konkrétní funkcí, jejíž otestování bylo hlavním cílem diplomové práce. Na základě požadovaného chování testované funkce aplikace byl vytvořen v několika krocích testovací scénář.

Podle testovacího scénáře byl poté v jazyce Java napsán test, který je spouštěn pomocí nástroje Selenium webdriver, a jeho výsledkem je to, že aplikace bude po zadávání stanovených vstupů produkovat očekávané výstupy.

Další část práce obsahuje popisy nejdůležitějších částí kódu testu a na závěr je provedeno monitorování, kde bylo v několika posledních buildech odhaleno neočekávané

chování aplikace. K těmto neočekávaným buildům byl proveden rozbor, kde se zjišťovala příčina chyby, která byla v práci popsána.

Tento test je v současnosti spouštěn každý den, spolu se sadou všech testů k aplikaci IPOAdmin. U chyb, které byly popsány v rámci práce bylo patrné, že chování konkrétní funkce probíhá stále správně, nicméně ve dvou případech se aplikace zasekla během načítání a bylo potřeba ji restartovat, s čímž test nepočítá. V jednom případě se špatně vygeneroval cesta k elementu, jehož nalezení na stránce bylo potřebné pro pokračování testu, a v jednom případě byla špatně nastavená práva testovacího uživatele, který poté neměl přístup k testované funkci.

Při jednom z těchto nedokončených buildů zůstala v aplikaci jedna novinka publikována. Při probíhání následných 10 testů bylo kvůli této novince vyhodnocení testů vždy jako selhání, i když test vlastně proběhl správně. Na vinně je, že test kontroluje v názvu a textu aktuality pouze kořen, který je při každým testu stejný, a proto si nevšiml že novinka vytvořená během testu byla smazána. Nejrychlejším řešením pro zajištění správných výsledků by bylo manuální smazání pozůstalé novinky, nicméně takové řešení nelze považovat nejefektivnější. V ideálním případě je třeba do testu přidat, aby při kontrole názvu a textu publikované novinky zohlednil i unikátní část, která je generována při začátku testu za pomoci systémového času.

To že byl test vytvořen však neznamená, že už se budou pozorovat pouze jeho výsledky. Test je potřeba stále udržovat, aby fungoval efektivně i po změnách provedených v aplikaci. S každou změnou, která bude v rámci aplikace vytvořena se může stát, že skript program pro své efektivní fungování bude potřebovat drobné úpravy jinak bude do výsledků posílat nepřesné údaje.

6 Závěr

Testování je v rámci celého projektu výroby softwaru velmi důležitá, avšak občas trochu opomíjená část. Čím rozsáhlejší je projekt, tím větší bývají i náklady a časová náročnost na jeho otestování. V současnosti je celosvětově na testování softwaru a celkový dohled na kvalitu projektu vynaloženo okolo třetiny finančních zdrojů. To, jakým způsobem bude testování probíhat je odvozeno od zvoleného vývojového modelu. Nyní se ve firmách nejvíce uplatňují agilní způsoby vývoje softwaru, kde testování probíhá neustále v průběhu psaní samotného programu. U menších projektů je ale běžné, že aplikace je nejprve celá vytvořena a až poté otestována.

Stejně jako se v průmyslu osvědčilo automatizovat výrobu, tak i správná automatizace testování softwaru dokáže výrazně zefektivnit, zlevnit a urychlit samotné testování. Většina světových firem proto v jisté míře automatizované testování softwaru využívá. Ačkoliv se může zdát, že automatizované testování má oproti manuálnímu pouze výhody a mělo by se používat vždy opak je pravdou. Vytváření automatizovaných testů má vysokou počáteční investici, a proto by se měli automatizovat pouze testy, které se budou několikrát opakovat. Zároveň se můžeme setkat i se situací, pro jejíž správné otestování je potřeba zohlednit lidský faktor, který automatické testy postrádají.

Součástí této práce bylo vytvoření automatizovaného testu, který kontroluje správnou funkčnost vytvoření novinky v aplikaci IPOAdmin. Výsledný test kontroluje správné chování při vytváření konceptu, publikování a mazání novinky. Jedná se o tzv. happy path test, kdy se očekávají standartní vstupy a nejsou zde žádné výjimečné nebo chybové podmínky.

Na konec praktické části bylo zmonitorováno několik buildů samotného testu, a objasněno kde nastali chyby, které vedli k chybovému hlášení. V diskuzi jsou uvedeny možnosti případného opravení nalezeného nedostatku v testu.

7 Seznam použitých zdrojů

1. PATTON, Ron. Testování softwaru. Praha: Computer Press, 2002. ISBN 80-7226-635- 5.
2. ROUDENSKÝ, Petr a Anna HAVLÍČKOVÁ. *Řízení kvality softwaru: průvodce testováním*. Brno: Computer Press, 2013. ISBN 978-80-251-3816-8.
3. Co je testovací scénář?. *Grafika, Design, Výpočty, Teorie A Praxe Programování, Osobního A Profesního Růstu - Na Stránkách Našich Webových Stránkách* [online]. [cit. 2022-07-08]. Dostupné z: https://cs.education-wiki.com/2263266-what-is-test-scenario
4. Závažnost a priorita v testování: Rozdíly a příklad - SoftGeek. *SoftGeek.net je blog specializující se na počítačový a mobilní software - SoftGeek* [online]. [cit. 2022-07-08]. Dostupné z: https://softgeek.org/cs/zavaznost-a-priorita-v-testovani-rozdily-a-priklad#Zavaznost_chyby
5. Co je to Incident při testování software?. *IT Slovník - počítačový slovník* [online]. [cit. 2022-07-08]. Dostupné z: https://it-slovnik.cz/pojem/incident
6. POLÁK, Jiří, Antonín CARDA a Vojtěch MERUNKA. *Umění systémového návrhu: objektově orientovaná tvorba informačních systémů pomocí původní metody BORM*. 4., aktualiz. vyd. Praha: Grada, 2003. Management v informační společnosti. ISBN 80-247-0424-2.
7. Co je testování softwaru? *Kitner* [online]. [cit. 2022-07-14]. Dostupné z: https://kitner.cz/testovani_softwaru/co-je-testovani-softwaru/
8. Modely životního cyklu softwaru. *Testování softwaru* [online]. [cit. 2022-07-14]. Dostupné z: http://testovanisoftwaru.cz/manualni-testovani/modely-zivotniho-cyklu-softwaru/
9. Vodopádový model. *Testování softwaru* [online]. [cit. 2022-07-14]. Dostupné z: http://testovanisoftwaru.cz/manualni-testovani/modely-zivotniho-cyklu-softwaru/vodopadovy-model/
10. Spirálový model. *Testování softwaru* [online]. [cit. 2022-07-15]. Dostupné z: http://testovanisoftwaru.cz/manualni-testovani/modely-zivotniho-cyklu-softwaru/spiralovy-model/

11. RUP - Rational Unified Process. *Testování softwaru* [online]. [cit. 2022-07-15]. Dostupné z: http://testovanisoftwaru.cz/manualni-testovani/modely-zivotniho-cyklu-softwaru/spiralovy-model/
12. Úrovně provádění testů. *Testování softwaru* [online]. [cit. 2022-07-28]. Dostupné z: http://testovanisoftwaru.cz/metodika-testovani/druhy-typy-a-kategorie-testu/faze-testu/
13. Funkční a nefunkční testy. *Testování softwaru* [online]. [cit. 2022-07-28]. Dostupné z: http://testovanisoftwaru.cz/metodika-testovani/druhy-typy-a-kategorie-testu/funkcni-a-nefunkcni-testy/
14. Manuální testování softwaru. *Testování softwaru* [online]. [cit. 2022-08-09]. Dostupné z: http://testovanisoftwaru.cz/manualni-testovani/
15. BUREŠ, Miroslav, Miroslav RENDA, Michal DOLEŽEL, Peter SVOBODA, Zdeněk GRÖSSL, Martin KOMÁREK, Ondřej MACEK a Radoslav MLYNÁŘ. *Efektivní testování softwaru: klíčové otázky pro efektivitu testovacího procesu*. Praha: Grada, 2016. Profesionál. ISBN 978-80-247-5594-6.
16. Historie jazyka Java. *FI MU* [online]. [cit. 2022-08-16]. Dostupné z: https://www.fi.muni.cz/usr/jkucera/pv109/2003p/xnovotn8.htm
17. PECINOVSKÝ, Rudolf. Myslíme objektově v jazyku Java: kompletní učebnice pro začátečníky. 2., aktualiz. a rozš. vyd. Praha: Grada, 2009. Myslíme v.. ISBN 978-80-247-2653-3.
18. History. *Selenium* [online]. [cit. 2022-08-16]. Dostupné z: https://www.selenium.dev/history/
19. What is Selenium? Introduction to Selenium Automation Testing. *Meet Guru99* [online]. [cit. 2022-08-16]. Dostupné z: https://www.guru99.com/introduction-to-selenium.html
20. Avasarala, Satya. Selenium WebDriver Practical Guide. Birmingham : Pack Publishing Ltd., 2014. 978-1-78216-885-0.
21. Herout, Pavel. XSLT 2.0 a SVG prakticky. České Budějovice : Kopp nakladatelství, 2010. 978-80-7232-406-4.
22. XPath in Selenium. *Meet Guru99* [online]. [cit. 2022-08-18]. Dostupné z: https://www.guru99.com/xpath-selenium.html

23. PECINOVSKÝ, Rudolf a Jarmila PAVLÍČKOVÁ. *Začínáme programovat v jazyku Java*. Praha: Grada Publishing, 2021. Začínáme s.. ISBN 978-80-271-3062-7.
24. Git - Historie a principy. *Itnetwork.cz - Učíme národ IT* [online]. [cit. 2022-08-19]. Dostupné z: https://www.itnetwork.cz/programovani/git/git-tutorial-historie-a-principy/
25. How to Get the Most Value Out of Your Software Testing Budget. *Perfecto by Perforce* [online]. [cit. 2022-11-18]. Dostupné z: https://www.perfecto.io/blog/how-get-most-value-out-your-software-testing-budget

8 Seznam obrázků

Obrázek 1 Cena chyby ... 16
Obrázek 2Rozdíl mezi přesností a správností ... 19
Obrázek 3 Vodopádový model ... 25
Obrázek 4 Spirálový model ... 27
Obrázek 5 Fáze RUP modelu ... 28
Obrázek 6 Scrum model ... 29
Obrázek 7 Porovnání ceny automatizovaného a manualního testování ... 39
Obrázek 8 Rozdělení nástroje Selenium ... 42
Obrázek 9 Relativní cesta XPath ... 45
Obrázek 10 Vývoj popularity jednotlivých IDE ... 48
Obrázek 11 Vzhled aplikace IPOAdmin ... 52
Obrázek 12 UseCase diagram ... 54
Obrázek 13 Stránka Novinky IPO ... 55
Obrázek 14 Podstránka pro vytvoření nové novinky ... 56
Obrázek 15 Umístění novinek v administraci stránek ... 57
Obrázek 16 Pop up okno novinky ... 57
Obrázek 17 Struktura projektu ... 61
Obrázek 18 Náhled vytvoření novinky ... 70
Obrázek 19 Jenkins přehled posledních buildů ... 76
Obrázek 20 Screenshot chyby v buildu 57 ... 77
Obrázek 22 Screenshot chyby v buildu 68 ... 78

Přílohy

CD se zdrojovými kódy a videem průběhu testu.