Univerzita Pardubice
Fakulta elektrotechniky a informatiky

# Automatizované testování webových aplikací

Bc. Martin Kment

Diplomová práce

2018

**Univerzita Pardubice**
**Fakulta elektrotechniky a informatiky**
Akademický rok: **2017/2018**

# ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

| | |
|---|---|
| Jméno a příjmení: | **Bc. Martin Kment** |
| Osobní číslo: | **I15210** |
| Studijní program: | **N2646 Informační technologie** |
| Studijní obor: | **Informační technologie** |
| Název tématu: | **Automatizované testování webových aplikací** |
| Zadávající katedra: | **Katedra softwarových technologií** |

Zásady pro vypracování:

V úvodní teoretické části práce je nutné provést přehled problematicky testování webových aplikací a porovnat jednotlivé přístupy. V práci je potřeba porozumět a zdokumentovat problematiku toho, co je vhodné testovat (testovací pyramidu). Práci bude obsahovat popis dvou základních návrhových vzorů pro tvorbů testů uživatelské vrstvy webových aplikací - page object pattern a screeflow pattern, a tyto patterny srovná z pohledu jejich výhod a nevýhod. V praktické části bude práce rozdělená do dvou oblastí. První část bude implementace webové aplikace pro sběr dat z internetových diskuzí na news portálech, jako idnes.cz, ihned.cz nebo novinky.cz. Aplikace bude obsahovat jednoduché uživatelské rozhraní pro zadání klíčového slova pro sběr dat a prohlížení nasbíraných dat. Druhá část bude implemetace automatizovaných testů této webové aplikace.

Rozsah grafických prací:

Rozsah pracovní zprávy: **cca 65 stran**

Forma zpracování diplomové práce: **tištěná**

Seznam odborné literatury:

**PATTON, Ron. Testování softwaru. Praha: Computer Press, 2002. Programování. ISBN 9788072266364.**
**FEWSTER, Mark a Dorothy GRAHAM. Software test automation: effective use of test execution tools. Reading, MA: Addison-Wesley, 1999. ISBN 978-0201331400.**
**MOSLEY, Daniel J. a Bruce A. POSEY. Just enough software test automation. Upper Saddle River, NJ: Yourdon Press, 2002. ISBN 978-0130084682.**
**SATYA, Avasarala. Selenium WebDriver Practical Guide:Community experience distilled. Packt Publishing Ltd, 2014. ISBN 1782168869.**
**MARTIN, Robert C. Čistý kód. Computer Press, Albatros Media a.s., 2016. ISBN 8025142752.**
**online 2017. SOURCE: https://www.pluralsight.com/guides/software-engineering-best-practices/getting-started-with-page-object-pattern-for-your-selenium-tests**
**online 2017. SOURCE: https://www.infoq.com/articles/Beyond-Page-Objects-Test-Automation-Serenity-Screenplay**

Vedoucí diplomové práce: **Ing. Pavel Jetenský, Ph.D.**
Katedra informačních technologií

Datum zadání diplomové práce: **30. října 2017**

Termín odevzdání diplomové práce: **18. května 2018**

Ing. Zdeněk Němec, Ph.D.
děkan

L.S.

prof. Ing. Antonín Kavička, Ph.D.
vedoucí katedry

V Pardubicích dne 15. listopadu 2017

Prohlašuji:

Tuto práci jsem vypracoval samostatně. Veškeré literární prameny a informace, které jsem v práci využil, jsou uvedeny v seznamu použité literatury.

Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorský zákon, zejména se skutečností, že Univerzita Pardubice má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

Beru na vědomí, že v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších předpisů, a směrnicí Univerzity Pardubice č. 9/2012, bude práce zveřejněna v Univerzitní knihovně a prostřednictvím Digitální knihovny Univerzity Pardubice.

V Pardubicích dne 14.5.2018

Bc. Martin Kment

# PODĚKOVÁNÍ

Na tomto místě bych chtěl poděkovat vedoucímu mé diplomové práce Ing. Pavlu Jetenskému, Ph.D. za jeho ochotu a cenné rady při tvorbě práce. Poděkování také patří mé rodině za jejich podporu a trpělivost.

**ANOTACE**

Cílem diplomové práce je návrh a implementace webové aplikace pro sběr dat z internetových diskuzí na zpravodajských webech a implementaci automatizovaných testů této aplikace. V teoretické části budou popsány základy testování softwaru a automatizovaného testování webových aplikací. Na závěr bude představena samotná aplikace, postup jejího vývoje a implementace automatizovaných testů.

**KLÍČOVÁ SLOVA**

automatizované testování, Groovy, Jsoup, Spock, Spring

**ANNOTATION**

The aim of this master thesis is design and implement a web application for collecting data from internet discussions on news sites and implementation of automated tests of this application. The theoretical part will describe the basics of software testing and automated testing of web applications. In conclusion, the application will be presented, the process of its development and the implementation of automated tests.

**KEYWORDS**

automated testing, Groovy, Jsoup, Spock, Spring

# OBSAH

# SEZNAM OBRÁZKŮ

# SEZNAM ZDROJOVÝCH KÓDŮ

# SEZNAM ZKRATEK A ZNAČEK

| | |
|---|---|
| **ACID** | Atomicity-Consistency-Isolation-Durability |
| **API** | Application Programming Interface |
| **ATLM** | Automated Testing Lifecycle Methodology |
| **CI** | Continuous Integration |
| **CSS** | Cascading Style Sheets |
| **DDT** | Data Driven Testing |
| **DI** | Dependency Injection |
| **DOM** | Document Object Model |
| **E2E** | End to End |
| **GUI** | Graphical User Interface |
| **HTML** | Hyper Text Markup Language |
| **IDE** | Integrated Development Environment |
| **IoC** | Inversion of Control |
| **J2EE** | Java 2 Enterprise Edition |
| **JPA** | Java Persistence API |
| **JVM** | Java Virtual Machine |
| **ORM** | Object Relation Mapping |
| **POM** | Page Object Model |
| **SQL** | Structured Query Language |
| **TDD** | Test Driven Development |
| **UML** | Unified Modeling Language |
| **URL** | Uniform Resource Locator |
| **WHATWG** | Web Hypertext Application Technology Working Group |
| **XSS** | Cross Site Scripting |
| **YAML** | YAML Ain´t Markup Language |

# TERMINOLOGIE

| | |
|---|---|
| **Abstrakce** | Oproštění se od nepotřebných detailů. |
| **Anotace** | Přidávají metadata ke třídám, metodám nebo proměnným. |
| **Apache 2.0** | Svobodná softwarová licence. |
| **Boilerplate** | Opakovaně psaný stále stejný kód. |
| **Build** | Sestavení zdrojového kódu, který může být spuštěn. |
| **Continuous Integration** | Technika pro průběžnou integraci kódu v týmu. |
| **Cron** | Plánovač úloh. |
| **EasyMock** | Framework umožňující využívání mock objektů. |
| **Element** | Jednotlivý prvek na stránce. |
| **End-to-end testy** | Testování kompletní aplikace pomocí reálných scénářů. |
| **Firefox** | Populární webový prohlížeč. |
| **Geb** | Řešení pro automatické prohlížení stránek pomocí skriptu. |
| **Groovy** | Dynamický jazyk založený na Javě. |
| **IEEE 830-1984** | Norma stanovující požadavky na dokument specifikace požadavků na software. |
| **IEEE-829** | Norma pro softwarovou a systémovou dokumentaci testů. |
| **Java** | Programovací jazyk. |
| **Jbehave** | Framework pro chováním řízený vývoj. |
| **Jenkins** | Jeden ze serveru pro průběžnou integraci. |
| **JUnit** | Framework pro jednotkové testy v Javě. |
| **Mock** | Nahrazení reálného testovaného objektu falešným, který se tváří jako reálný. |
| **Mockito** | Framework pro mockování. |
| **Plugin** | Zásuvný modul pro nějakou aplikaci |
| **Polymorfismus** | Možnost volání jedné metody objektu s různými parametry. |
| **Replikace** | Tvorba identické kopie. |
| **Repositář** | Datové uložiště, které poskytuje detailní pohled do minulosti. |
| **Skript** | Předem daná posloupnost příkazů. |
| **TestNG** | Testovací framework inspirovaný JUnitem, obohacený o některé funkcionality. |
| **XPath** | Jazyk pomocí, kterého lze adresovat určitou část dokumentu a vybírat jednotlivé elementy a pracovat s nimi. |

# ÚVOD

V dnešní době většina populace nečte zprávy v novinách, ale využívá služeb zpravodajských serverů, kde o nich může zároveň i diskutovat. Tyto diskuze mohou být zajímavým zdrojem dat, které mohou mít další využití. Z tohoto důvodu vznikla tato práce, která si klade za cíl vytvořit nástroj, který bude tyto data extrahovat. Zároveň je tato aplikace vhodný kandidát na automatizované testování, díky předpokládanému přidávání nových portálů, nebo regresnímu testování při změně struktury stávajících portálů.

V úvodní kapitole je krátká rešerše již existujících zdrojů zabývající se testováním. Další kapitoly se nejdříve zaobírají testováním softwaru obecně a později přímo automatizací. Důraz je kladen na popsání tzv. dobrých a špatných praktik používaných při vývoji. Práce ani neopomíjí testovací nástroje a zaměřuje se na frameworky Selenium a Spock. V poslední teoretické kapitole je popsána metodologie průběžná integrace (Continuous integration).

Poslední dvě kapitoly popisují vyvinutou aplikaci a metodiky použité pro její testování, včetně ukázek zdrojových kódů.

# 1 REŠERŠE

O testování, manuálním i automatizovaném, existuje velké množství knih, akademických prací i webových stránek věnující se tomuto tématu. Pokud se jedná o testování pomocí Spocku v jazyce Groovy, tak situace není moc ideální. Existuje pouze několik knih a internetových článků v anglickém jazyce.

## 1.1 Odborná literatura

Většina knih o testování se věnuje z větší části manuálnímu testování a pouze z části automatizovanému. Existuje i dost literatury v českém jazyce, ale většina je pouze přeložena z angličtiny.

1. Testování pro programátory, autor Pavel Herout, rok vydaní 2016. [1] Jedná se o jednu z mála knih o testování od českého autora. Kniha je spíše než pro testery určena pro programátory. Pouze první kapitola obsahuje teoretické základy testování. V dalších kapitolách zabývající se jednotkovému testování pomoci JUnit, mockování, logování či statické analýze kódu je velké množství praktických ukázek. Automatizovanému testování webových aplikaci se ovšem věnuje pouze v jedné kapitole. Stejně jako jeho ostatní knihy je psaná čtivým stylem.
2. Testování software, autor Ron Patton, rok vydání 2002. [2] Jedná se starší knihu, ale je to vynikající úvod do problematiky testování softwaru. Testování se věnuje spíše v teoretické rovině.
3. Software Test Automation: Effective use of test execution tools, autoři Mark Fewster a Dorothy Graham, rok vydání 1999. [3] V první části jsou popsány rozdíly mezi automatizovaným a manuálním testováním. V druhé části je několik případových studií na, kterých autoři vysvětlují implementaci a průběh testování.
4. Programování řízené testy, autor Kent Beck, rok vydání 2004. [4] Kniha o TDD od jednoho z jejich autorů je rozdělena na tři části. V první části je ukázáno, jak probíhá automatickými testy řízená úprava stávajícího kódu, druhá část ukazuje programování pomocí automatickými testy řízeného vývoje a třetí část je referenční příručka s vysvětlením základních pojmů a technik.
5. Selenium Testing Tools Cookbook, autor Unmesh Gundecha, rok vydání 2015. [5] Kniha poskytuje kompletní návod, jak pracovat se Selenium WebDriver. Obsahuje

podrobný návod, jak vyhledávat a pracovat s elementy na webové stránce. Stejně jako praktické použití návrhového vzoru Page Object či integraci Selenia s jinými testovacími nástroji.

Pro testování pomocí Spocku je k dispozici pouze anglicky psaná literatura.

6. Java Testing with Spock, autor Konstantinos Kapelonis, rok vydání 2016. [6] Jedná se o jednu z mála knih věnující se testování pomocí Spocku. Autor v knize popisuje základy jazyka Groovy, pokračuje úvodem do testovacího frameworku Spock a věnuje se podrobně všem důležitým technikám ve Spocku. Jedná se o obsáhlou knihu se spoustou praktických příkladů, která je nezbytností pro každého, kdo začíná s testováním pomocí Spocku.

## 1.2 Internetové zdroje

Článků věnující se testování lze na internetu nalézt velké množství, česky i anglicky, ale webů přímo specializující se na testování je méně, a ne vždy jsou kvalitní a aktualizované.

1. testovanisoftwaru.cz [7] – zde jsou ve stručnosti vysvětleny základní pojmy testování. Web není nijak obsáhlý ani často aktualizovaný, ale jedná se o jednu z mála českých stránek věnující se pouze testování.
2. www.softwaretestingmentor.com [8] – jedná se o stránku Manishe Vermy, který se zde věnuje jak manuálnímu, tak automatizovanému testování. Stránka obsahuje velké množství článků, ale bohužel je již déle neaktualizována.

Dále lze nalézt i praktické kurzy specializující se na testování pomocí nějakého určitého nástroje.

1. www.itnetwork.cz/java/testovani [9] – server, který obsahuje velké množství tutoriálů. Jeden z nich se přímo zabývá testování v Javě pomocí JUnit, Selenia, včetně Page Object patternu. Tutoriály jsou dobře zpracovány, bohužel jejich část je zpoplatněna.
2. www.udemy.com/learn-automation-with-geb-and-spock/ [10] – online kurz věnující se Spocku a Gebu, který v 6 hodinách videa stručně představí.

Jako zdroje dobře poslouží oficiální stránky.

1. spockframework.org/spock/docs [11] – oficiální dokumentace k frameworku Spock, kde lze nalézt krátké představení všech funkcionalit tohoto frameworku.

2. github.com/spockframework [12] – repositář Spocku, kde je jednak jeho zdrojový kód, ale hlavně velké množství praktických příkladů.

## 1.3 Akademické práce

Prací na téma testování se dá nalézt velké množství. Ovšem pro testování pomocí Spocku se mi v době psaní této práce nepodařilo nalézt žádnou česky psanou práci.

1. Diplomová práce Automatizované testování webových aplikací, autorka Klára Smatanová, rok 2015. [13] Autorka popisuje automatizované testování, metodiku a uvádí několik nástrojů pro automatické testování. Praktickou částí je návrh a implementace automatizovaných testů na existující aplikaci TestLink.
2. Diplomová práce Testování webových aplikací s využitím nástroje Selenium WebDriver. autorka Lucie Třísková, rok 2015. [14] V práci porovnává manuální a automatizované testování, metodiku automatizace testování a v příloze poskytuje metodickou příručku k Selenium WebDriver.
3. Diplomová práce Automation of regression testing of web applications, autor Dávid Chmurčiak, rok 2013. [15] Anglicky psaná práce věnující se nástrojům pro automatizaci regresního testování webových aplikací a návrh pro testování aplikace.
4. Diplomová práce Testování webových aplikací, autorka Anna Borovcová, rok 2008. [16] Velmi obsáhlá práce, která je zaměřená na problematiku testování zejména pohledem testera a ne programátora. Popisuje nejdůležitější aspekty testování, včetně praktického příkladu.

# 2 TESTOVÁNÍ SOFTWARU

Tato kapitola slouží jako úvod do testování. V krátkosti zde budou představeny nejdůležitější pojmy týkající se obecně testování softwaru.

## 2.1 Definice testování softwaru

Přesná definice pro testování software neexistuje a spousta autorů si tento pojem vykládá jinak. Podle [17] je testování softwaru dynamické ověření toho, že program poskytuje očekáváné chování pro konečnou sérii zkušebních případů.

Dijkstra ve své práci [18] napsal, že testování může být efektivní cestou k ověření přítomnosti chyb, ale nedokáže vyloučit, že software žádné chyby neobsahuje. Zjednodušeně řečeno pojem softwaru bez chyb je oxymóron.

Podle Manishe Verma [19] se testování dá chápat jako proces s těmito rysy:

- Testování není samostatná činnost, ale celá řada činností. Testuje se ve všech fázích životního cyklu softwaru. Od návrhu až po konečný produkt.
- Testování je statické i dynamické. Statické se používají v počátcích vývoje produktu a dynamické v pozdější fázi vývoje.
- Testování vyžaduje testovací plány, které patří mezi nejdůležitější části testování.
- Při testování se vyhodnocuje, aby bylo ověřeno, že software splňuje všechna výstupní kritéria, intuitivní ovládání a jsou zapracovány všechny požadavky koncového uživatele.
- Testování software není jenom o testování kódu, ale i o otestování souvisejících dokumentů, jako referenční příručka, uživatelský manuál nebo průvodce instalací.

Tento proces má tři cíle:

1. Zjištění, zda software splňuje všechny požadavky koncového uživatele.
2. Ujištění se, že software je vhodný pro užívání.
3. Nalezení závad a chyb v software.

Asi nejoficiálnější je definice podle ISTQB [20]: „ Proces, který se skládá ze všech činností životního cyklu, statického i dynamického, se týká plánování, přípravy a hodnocení

softwarových produktů a souvisejících pracovních produktů k určení, zda splňují specifikované požadavky, aby bylo prokázáno, že jsou vhodné pro účely a zjišťují závady.“

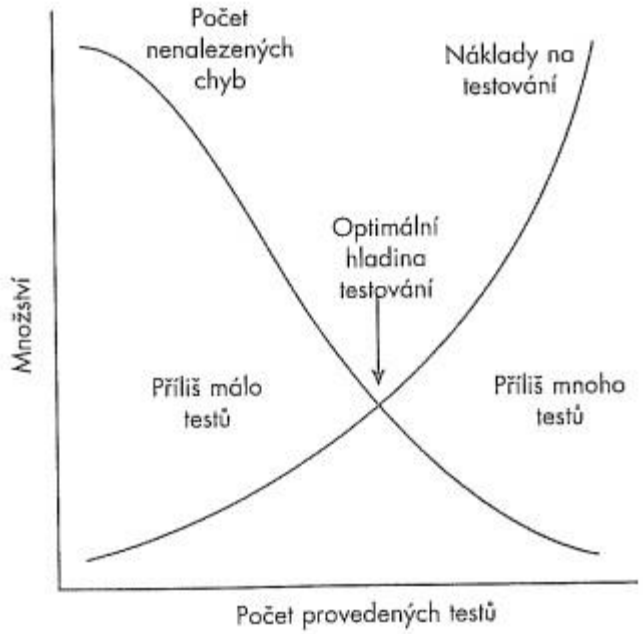
## 2.2 Co je to chyba

V mnoha firmách se vedou debaty o tom, co je to vlastně chyba a jak jí přesně specifikovat. Podle [2] je to chyba, pokud je splněna alespoň jedna z následujících podmínek:

1. Software nedělá něco, co by podle specifikace produktu dělat měl.
2. Software dělá něco, co by podle údajů specifikace produktu dělat neměl.
3. Software děla něco, o čem se produktová specifikace nezmiňuje.
4. Software dělá něco, o čem se produktová specifikace nezmiňuje, ale měla by se zmiňovat.
5. Software je obtížně srozumitelný, těžko se s ním pracuje, je pomalý, nebo – podle názoru testera software – jej koncový uživatel nebude považovat za správný.

## 2.3 Kdy přestat testovat Patton

To je spíše záležitost ekonomická. Na testování bývá vyhrazeno něco mezi 25 % až 50 % celkového rozpočtu [21]. Parametry kvantita, kvalita, termín a rozpočet musí být vzájemně v souladu podle business plánu projektu. Proto existuje optimální hladina testování, jak je vidět na Obrázek 2.1: Efektivní hladina testování [2].




Obrázek 2.1: Efektivní hladina testování [2]

Toto samozřejmě nemusí vždy platit. U projektů týkající se kritické bezpečnosti není možné chyby nechat a musejí se opravit i přes velké náklady. [2]

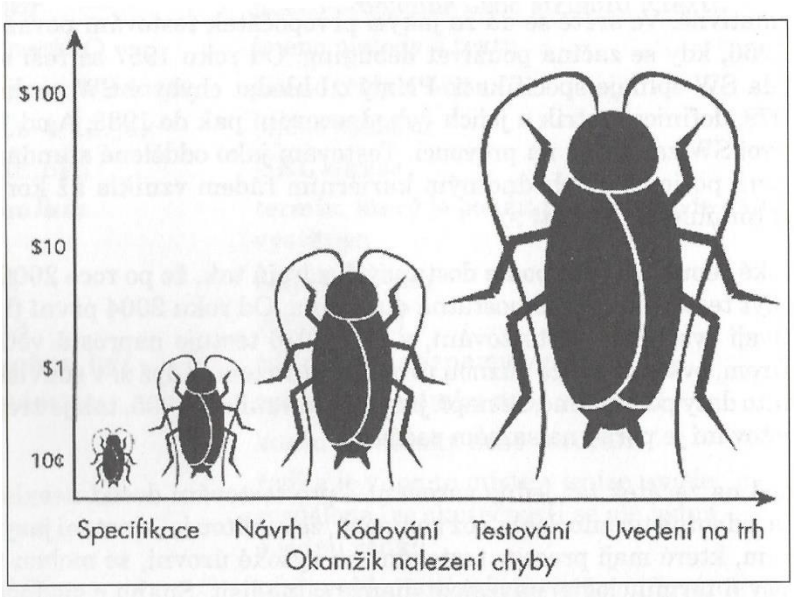
## 2.4 Ne všechny chyby se opraví

Někdy i po sebelepším testování se objeví nějaká chyba. Důvodů, proč ji nechat neopravenou, může být několik [2]:

- Není dostatek času, takže aby se stihl termín, tak se některé nedůležité chyby neopraví.
- Vlastně to není chyba, ale vlastnost produktu, ať už z důvodu nepochopení ze strany testera, nebo z důvodu špatné specifikace od zákazníka.
- Oprava by byla příliš riskantní a způsobila by další chyby. Tomu se ovšem dá docela dobře předcházet regresním testováním.
- Opravovat danou chybu se nevyplatí, protože se vyskytuje velmi ojediněle nebo jenom na nějaké specifické konfiguraci.

## 2.5 Náklady na chybu

Proč se vyplatí řádné testování už od začátku projektu, nejlépe ukazuje obrázek od Rona Pattona. Ukazuje, jak stoupají náklady na odstranění chyby v pozdějších fázích vývoje. Náklady stoupají logaritmicky. Ideální je tedy odhalit a opravit chybu ještě ve fází návrhu.




Obrázek 2.2: Náklady na opravu chyb

Na druhou stranu odstranění chyby může být v pozdní fázi vývoje velmi náročné. Každá změna totiž může způsobit chybu někde jinde (regresní chyba). Pokud je projekt dobře pokryt automatizovanými testy, tak je možné provádět regresní testování po opravě každé chyby a snadno najít chyby, které způsobila tato oprava.

## 2.6 Životní cyklus testování

Označované zkratkou STLC. Pro kvalitní otestování software je nutné mít systematicky naplánovaný celý cyklus testování. Pro vodopádový vývoj se využívá těchto osm kroků [22].

1. Analýza požadavků zákazníka ve spolupráci s vývojáři, aby bylo jasné, co je cílem.
2. Plánování testů, aby bylo jasné, co všechno musí být otestováno a jak. Vytvoří se plán testů.
3. Analýza testů pro rozhodnutí o případné automatizaci některých testů a zjištění ve kterých fázích musí být jaké testy provedeny.

4. Psaní testovacích případů pro manuální testování a případně tvorba scriptů pro automatizované testování.
5. Psaní testů a jejich ověřování. V tomto kroku by měli být napsány všechny testy.
6. Vykonání testu a hlášení chyb. Po jejich opravě, opakované provedení testů pro zjištění, zda byla chyba úspěšně opravena.
7. Nefunkční testování a ověření, že software byl kompletně otestován.
8. Vyčištění testovacího prostředí a zadokumentování všech problému při testování pro budoucí použití.

## 2.7 Pokrytí kódu testy

Je to technika, při které je zjišťováno, jaké procento kódu je pokryto jednotkovými testy. Zjišťuje se při ní, zda tester při psáni testů neopomněl nějakou část kódu. Tato technika patří do testování bílé skříňky, protože je nutný přístup ke zdrojovému kódu. Pokrytí se zjišťuje pomocí automatizovaného procesu, který na základě spuštění jednotkových testů, zjistí, které části kódu jsou neotestované. Existuje několik technik měření pokrytí. Nejjednodušší je pouhé zjištění, jaké řádky kódu jsou volány v rámci testu. To je ovšem nevýhodné, protože nemusí být otestovány všechny rozhodnutí v podmínkách. Nejčastěji používanou metodou je pokrytí cest. Zjišťuje se, jestli při spuštění testů byly projity všechny možné průchody programem. Tím je zajištěné maximální pokrytí kódu. Nevýhodou zde může být to, že množina cest roste exponenciálně s počtem možných průchodů programem. Cílem této techniky není dosažení 100 % pokrytí, ale řídí se pravidlem 80-20, kdy je dosaženo 80 % pokrytí a zvyšování pokrytí nad danou hodnotu je náročné. [23]

### 2.7.1 OpenClover

Patří mezi nejpoužívanější nástroje pro zjišťování pokrytí kódu. Je distribuován zdarma a pro mnoho platforem. V základě podporuje JUnit, TestNG a Spock, ale na základě definování testovacího vzorce dokáže podporovat i ostatní testovací frameworky. Jeho velká výhoda je, že je možné ho pomocí pluginu integrovat do Jenkins a testování pokrytí je v režii CI. Kromě zjišťování pokrytí kódu, také dokáže optimalizovat testy a zjistit který test je nutný znovu

spustit pro modifikovaný kód, díky tomu je možné snížit čas potřebný pro regresní testování. [24]

## 2.8 Dokumentace

Žádné testování by nebylo úplné, kdyby k němu neexistovala nějaká dokumentace. V malém týmu může být dokumentace vedena velmi neformální cestou, ale v případě velkých firem se dokumentace tvoří podle standartu IEEE-829, který se přímo zabývá tvorbou testovací dokumentace. Norma specifikuje formát daných dokumentů, ale nepřikazuje je vyžívat všechny.

Nejdůležitější dokument je produktová specifikace nebo také dokument specifikace požadavků na software. Je to obsáhlý dokument obsahující popis požadavků na software získaných od zákazníka. Je zde nutné přesně specifikovat co by měl požadovaný produkt umět. Podle standartu IEEE 830-1984 by měl obsahovat tyto části [25]:

- Úvod, kde bude specifikován účel aplikace, rozsah systému, reference na dokumenty, na které se odkazuje a přehled co nalezneme ve zbytku produktové specifikace.
- Všeobecný popis obsahující vztahy k ostatním produktům, pokud bude fungovat v rámci většího celku, požadovaný přehled funkcí, popis budoucích uživatelů systému, předpoklady a závislosti.
- Specifikace požadavků obsahující požadavky na funkce, výkonost, vlastnosti a vnější rozhraní.
- Ověřovací kritéria, kde jsou specifikovány, jak bude ověřován výsledný produkt. Pomocí jakých druhů testů nebo výkonnostních charakteristik.
- Přílohy, kde je rejstřík a obsah. Také zde mohou být velké diagramy či zdroje dat pro otestování produktu.

Mezi další důležité dokumenty patří testovací plán, kde je definováno, co se všechno bude testovat, jaké budou potřeba zdroje, jakým způsobem se bude testovat a kolik na to bude času. Neméně důležitý dokument je přehled plánování testovacích případů. Testovací případy (test case), obsahují posloupnost jednotlivých akcí pro daný test. Poslední důležitý dokument je test log, kam jsou zaznamenávány výsledky testů.

# 3 ČLENĚNÍ TESTŮ

Testy se dají dělit podle spousty kritérii. Zde jsou uvedeny nejznámější dělení testů, spolu s jejich krátkým popisem.

## 3.1 Černá, bílá a šedá skříňka

Toto je rozdělení testů na základě toho, jaký máme k produktu přístup. Pokud testujeme bílou skříňku (white box), tak máme plný přístup ke zdrojovému kódu a testujeme na jeho základě. Ať už pouhou kontrolou kódu, jestli odpovídá specifikaci, tak jeho spouštěním a ověřování, jestli funguje, jak má. Tento přístup je problematický v tom, že tester může být ovlivněn kódem a psát tedy testy šité mu na míru.

Černá skříňka (black box) se liší tím, že tester nemá přístup k samotnému programovému kódu, ale pouze k rozhraní. Testuje tedy uživatelské rozhraní, a hlavně provádí testování podle scénářů.

Zvláštním typem je testování šedé skříňky (grey box), kdy jsou kombinovány oba přístupy. Využívá se často při testování webových aplikací, kdy je možné snadno zobrazit kód webové stránky, ale již nemáme přistup k samotnému kódu aplikace.

## 3.2 Statické a dynamické testování

Zde je rozdíl v tom, zdali je potřebné software spustit nebo ne. Statické testování je založeno na revizi zdrojového kódu a není nutné mít spustitelnou verzi. V této fázi se dají relativně jednoduše objevit chyby a rychle je napravit.

Dynamické testování vyžaduje již určitou spustitelnou verzi a testujeme na základě zadávaných vstupů a výstupů programů.

## 3.3 Manuální a automatické

Pokud daný test potřebuje vyhodnocení člověkem, tak se jedná o manuální test. Používá se hlavně v pokročilejších fázích vývoje, kdy se testuje GUI.

Oproti tomu automatické testování se vyznačuje tím, že není potřeba lidského zásahu k tomu, aby proběhl a byl vyhodnocen. Vhodné jsou pro často opakované testy.

## 3.4 Pozitivní a negativní testování

Pokud využíváme pozitivní testování, jinak také testy splněním, tak ověřuje, zda po zadání validních vstupů se chová software podle očekávání. Naopak při negativním testování neboli testy selháním, ověřujeme, jak se software chová po zadání nevalidních údajů. Nikdy nevíme, co zadá uživatel a nikdy nemůžeme předpokládat, to že tohle by přece nikdy uživatel neudělal. [26]

## 3.5 Testování podle scénáře

Velmi rozšířené testování, kdy se manuálně podle specifikace nebo předpokládaného využívání aplikace vytvoří scénáře. Scénář vždy popisuje sled kroků, které tester musí vykonat a jejich očekávaný výstup. Nejde o aktivní vyhledávání chyb, ale ověřování správné funkčnosti podle specifikace.

## 3.6 Průzkumné testování

Zjednodušeně by se dalo říci, že tester se snaží aplikaci rozbít a tím nalézt chyby. V tomto případě si tester musí přesně zaznamenávat co provedl, aby bylo možné chybu replikovat.

## 3.7 Testovací mix

Pro kompletní otestování softwaru je nutné vytvořit správný mix testů. Každé testovaní se totiž zaměřuje na něco jiného, má své slabé a silné stránky. Je tedy nutné, aby byly testovací techniky správně vybalancované.

V souvislosti s testovacím mixem se používá pojem pesticidový paradox, který znamená, že metoda použitá k odhalení chyb po sobě zanechává chyby, které již neodhalí a je tedy nutné použít jinou metodiku testování, aby byla odhalena. Další význam je v tom, že pokud se testovací mix neobměňuje, tak se po čase software stane rezistentním a další chyby již neodhalí.

# 4 VHODNÉ PRINCIPY AUTOMATIZOVANÉHO TESTOVÁNÍ

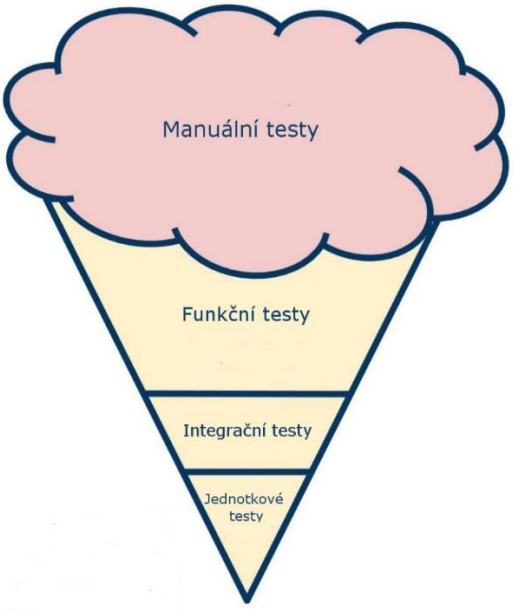
## 4.1 F.I.R.S.T princip

Tento princip shrnuje požadavky na unit testy. [27, 28]

- Fast – testy by měly být rychlé, aby byly možné spouštět co nejčastěji. Pokud by byly příliš pomalé, tak je programátor nebude chtít spouštět často a nebude mít dostatečnou zpětnou vazbu. Musíme si uvědomit, že pokud jsou stovky testů v jednom projektu, tak každá milisekunda je důležitá.
- Isolate/Independent – Testy na sobě nesmějí záviset. Musí být jedno v jakém pořadí se spouští a nesmějí využívat výsledků jiných testů. Všechny proměnné v testu musejí být definovány jako součást testu. Stejně jako by selhání jednoho testu nemělo být přímou příčinou selhání jiného testu.
- Repetable – Test musí vykazovat stále stejné výsledky při každém spuštění. Musí být deterministický (předvídatelný).
- Self-validation – Není třeba manuální kontroly, aby byl určen výsledek testu. Test sám musí vyhodnotit, jestli prošel nebo ne.
- Timely – Testy by měly pokrýt všechny scénáře a nesnažit se o 100 % pokrytí kódu. Všechny testy měly být napsány ještě před samotným kódem programu nebo s ním. V žádném případě by neměly být dopisovány později, to není smyslem unit testování.

## 4.2 Testovací zmrzlina

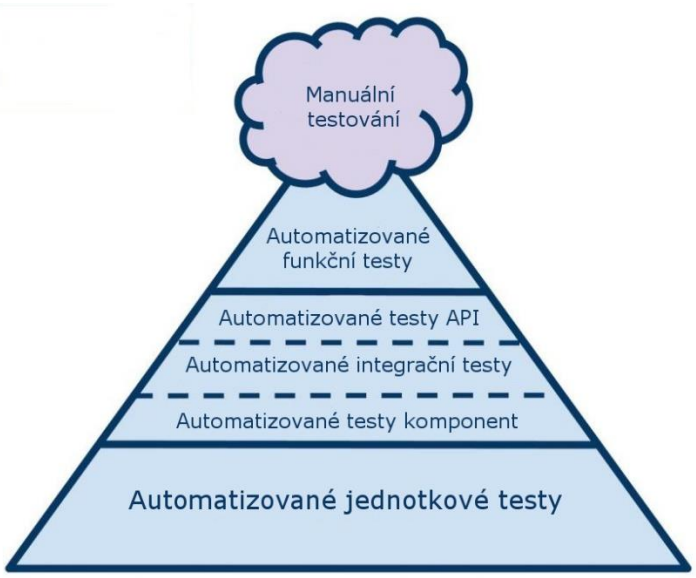
Tento návrhový vzor je založen na předpokladu, že pří vývoji se nejvíce zaměříme na testování manuální a end-to-end testy. Tyto věci jsou nejdůležitější z hlediska businessu. Integračním a unit testům se bude věnovat minimálně. Zde nastává problém, že manuální a e2e testy jsou pomalé a oprava těchto chyb může být velmi drahá a náročná. Hlavní problém, ale tkví v tom, že nebude dostatečně otestován kód samotného programu a při dlouhodobém vývoji to bude způsobovat spoustu problémů. Testovací zmrzlina by se tedy dala považovat za takzvaný antipattern a nemělo by se podle ní postupovat při tvorbě struktury a návrhu testů. [29]




Obrázek 4.1: Testovací zmrzlina [29]

## 4.3 Testovací pyramida

Toto složení testů je opačné proti testovací zmrzlině a dodržuje princip F.I.R.S.T. Základem jsou unit testy, které jsou jednoduché a levné. Naopak e2e a manuální testováni je drahé a pomalé a mělo by mu být věnováno minimum času. Samozřejmě, že v určitých případech není nutné je mít. Pokud se věnujeme pouze vývoji nějakého jádra, tak jsou zbytečné.




Obrázek 4.2: Testovací pyramida [29]

# 5 AUTOMATIZOVANÉ TESTOVÁNÍ WEBOVÝCH APLIKACÍ

Stručné představení automatizovaného testování s jeho výhodami a nevýhodami.

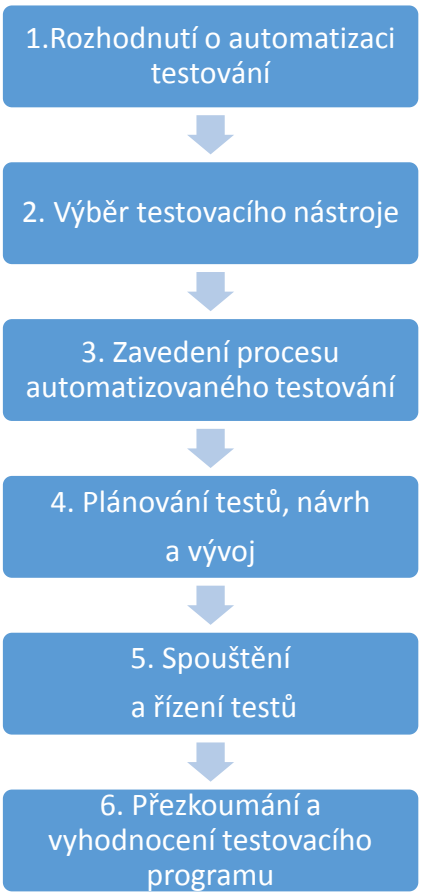
## 5.1 Definice

Aby bylo možné mluvit o automatizovaném testování, tak je nutné, aby alespoň část testovacího procesu probíhala bez zásahu člověka. Většinu často opakovaných manuálních testů lze automatizovat. Cílem tedy je ušetřit čas při testování.

Manish Verma definuje automatizované testování jako: „Použití automatizačních nástrojů pro psaní a provádění testovacích případů je známé jako testování automatizace. Při provádění automatického testovacího balíku není vyžadován žádný ruční zásah. Testeři zapisují zkušební skripty a testovací případy pomocí automatizačního nástroje a pak se seskupují do testovacích sad.“ [30]

## 5.2 Metodika životního cyklu automatizovaného testování

Pokud se rozhodneme automatizovat testování, tak je nejjednodušší využít proces ATLM, který v šesti fázích zajistí, že přechod k automatizaci bude kvalitní a nebude nutné proces automatizace nákladně přepracovávat. Vždy se musí splnit celá fáze, než se může přejít do další a tím je zajištěna úspěšná implementace automatizovaných testů. To se týká již existujících (legacy) softwarových projektů, u nově vznikajícího softwaru je nezbytné, aby automatizované testy vznikaly v průběhu vývoje (implementace). [2]




Obrázek 5.1:Fáze automatizace

## 5.3 Výhody automatizace

Podle Pattona existují čtyři výhody automatizace [2]:

- Rychlost – automatizované testy se provádějí násobně rychle, než by byl schopen provádět člověk.
- Efektivita – testy neprovádí živá entita, ale stroj, takže zbývá více času na jiné aktivity než spouštění testů.
- Správnost a přesnost – každý člověk dělá chyby a po otestování velkého množství scénářů se pravděpodobnost chyby zvyšuje. Stroj ovšem podává stále stejné výsledky.
- Neúnavnost – testy se provádí kdykoliv je potřeba.

Manish Verma na svých stránkách [31] uvádí 6 výhod automatizace testování:

- Rychlost – testy jsou spouštěny rychleji než uživatelem.
- Opakovatelnost – testeři mohou testovat, jak aplikace reaguje po provedení stejné operace.
- Spolehlivost – testy provádějí stejnou operaci pokaždé, když jsou spuštěny a eliminuje se tedy chyba lidského faktoru.
- Komplexnost – tester může vytvořit více sad testů, které pokrývají všechny funkce aplikace.
- Programovatelné – tester může naprogramovat sofistikovanější testy, které mohou přinášet skryté informace.

## 5.4 Nevýhody automatizace

Automatizace nemusí být vhodná pro malé projekty anebo ty kde není plánovaná dlouhá životnost, vždy záleží na konkrétní situaci a rozhodnutí managementu. Pro automatizaci je nutné vykonat více práce a potom není dostatečná návratnost. Testy je také nutné udržovat a provozovat na speciálních serverech, což může být pro malý projekt finančně neúnosné.

Nikdy také není možné úplně nahradit manuální testování. Není třeba možné otestovat, jestli jsou ovládací prvky vhodně umístěny nebo, že zobrazovaná data jsou čitelná pro člověka.

# 6 KATEGORIZACE AUTOMATIZOVANÝCH TESTŮ

Existuje několik různých přístupů, jak využívat výhody automatizovaného testování. V této kapitole budou popsány přístupy pro automatizované testování, které jsou rozděleny podle toho, jak se přistupuje k testování. Na konci kapitoly budou popsány nejznámější metodiky pro psaní automatizovaných testů.

## 6.1 Podle přístupu k aplikaci

### 6.1.1 Nástroje pro testování uživatelského rozhraní (GUI)

To je testování z pohledu koncového uživatele. Přistupuje se zde pouze ke grafickému rozhraní aplikace a z části ke kódu HTML stránky. Jedná se tedy o testování šedé skříňky. Testuje se, zda chování ovládacích prvků odpovídá specifikaci zákazníka.

### 6.1.2 Nástroje pro testování řízené kódem

Založeno na testování z pohledu programátora. Jedná se tedy o testy bílé skříňky. Testují se metody, třídy nebo jednotlivé moduly, zda poskytují očekávané výsledky.

## 6.2 Dle úrovně testování

### 6.2.1 Jednotkové testy

Podle testovací pyramidy se jedná o základní a nejdůležitější testy. Testujeme co nejmenší části programového kódu (třídy, interface, metody). V této fázi je nejjednodušší odhalování a oprava chyb, protože nemusíme řešit dopady na ostatní části programu. [32]

### 6.2.2 Integrační testy

Tyto testy zajišťují otestování jednotlivých modulů software. Software se totiž většinou neskládá pouze z jednoho modulu, ale z více. Je to o stupeň víše než jednotkové testy. Stále se ovšem testuje na úrovni zdrojového kódu.

### 6.2.3 Systémové testy

Zde se již jedná o test celého software, aby bylo zajištěna jeho funkčnost jako celku. Testují se zde specifikace softwaru.

### 6.2.4 Akceptační testy

Tyto testy již neprovádí vývojový tým, ale tester pověřený klientem. Ověřuje se zde zdali software splňuje všechny klientovy požadavky. Jedná se o nejdůležitější testy z hlediska uvedení či neuvedení produktu. Tato úroveň testování obvykle není automatizována. Akceptační testy bývají často součástí smlouvy o dodávku SW, a jejich úspěšné proběhnutí je vázáno na fakturaci.

## 6.3 Podle typu testů

### 6.3.1 Regresní testy

Tyto testy se provádějí, pokud dojde k nějaké změně ve zdrojovém kódu a je nutné ověřit, zda tím není ovlivněna funkcionalita software. U velkých projektů patří mezi nejdůležitější, protože testují, jestli se po opravě chyby nebo přidání nové funkcionality nevyskytla nová chyba. Tyto testy jsou ideální kandidát pro automatizaci, protože jsou zdlouhavé a nákladné.

### 6.3.2 Retestování

Provádí se po opravení chyby. Na rozdíl od regresních testů se netestuje celá aplikace, ale pouze ta část, která byla chybou ovlivněna.

### 6.3.3 Nefunkční testování

Zde se jedná o testování těch funkci, které se netýkají přímo funkcionalit daného software. Patří sem zátěžové testování, kdy na systému simulujeme práci více uživatelů. Testujeme předpokládané zatížení i maximální zatížení a sledujeme co se v systému děje. Identifikujeme tím, kde jsou v softwaru slabá místa anebo jaká je maximální kapacita.

Výkonnostní testování provádíme, abychom zjistily, zda jsou splněny požadavky na výkonost systému. Potřebujeme zjistit, jak se systém chová například v případě zpracovávání transakcí v databázi nebo pří zvýšení latence sítě.

Velmi důležité jsou bezpečnostní testy (někdy též nazývané jako penetrační testy). Zde ověřujeme, zda správně funguje autorizace, zabezpečení dat, ochrana proti SQL Injection, validace a verifikace.

### 6.3.4 Výkonnostní testy

Používají se k ověření, jak funguje daná aplikace pod zátěží, buď simulací přístupu velkého množství uživatelů nebo při velkém množství požadavků.

## 6.4 Metodiky psaní automatizovaných testů

### 6.4.1 Vývoj řízený požadavku na chování (Behavior Driven Development)

Jedná se o specifický přístup k vývoji, kdy se před psaním kódu napíší požadavky na chování aplikace ve formě scénářů. Zakládá se na principu Given-When-Then, což znamená, že v given je nastavení, v when je, co se bude dít a v then je co se stane. Specifikuje se tedy chování aplikace uživatelem a ne programátorem. [33]

### 6.4.2 Test driven development

Patří do extrémního programování. Jedná se o přístup, kdy se nejdříve píší jednotkové testy a potom teprve samotný kód programu. Cílem je nejdříve napsat test, který selže a následně psát kód, dokud test neprojde. Zjednodušeně řečeno nejdříve si vytyčíme cíl, čeho chceme dosáhnout a potom se toho snažíme dosáhnout. Test zároveň říká, co přesně se má udělat a taky co je špatně v kódu. Zároveň se používají i nástroje na pokrytí kódu testy. Pokud nějaký kód není pokryt testy, tak je zbytečný a není nutný k funkci programu. Pokud se tento nástroj používá u projektu, kde jsou testy psány až po napsání kódu, tak to vede k tomu, že se píše test i na části kódu, které nejsou potřebné pro běh a neodhalí se, že je zbytečný. Pokud se testy píšou až po napsání kódu, tak se stává, že jsou psány přímo na míru danému kódu a při jeho změně se bude muset změnit i test, protože není dostatečně obecný. Problém u TDD je, pokud přesně nevíme, co bude výsledkem. V tomto případě by se museli často měnit testy.

Podle Kenta Becka [4] je cyklus vývoje:

1. Napsání testů – pro přidání nové funkcionality se začíná napsání testu nebo sady testů. Případně při změně specifikací aplikace i změnu stávajících testů. Zde je i výhoda pro programátora, že musí porozumět, jak to má fungovat ještě předtím, než začne psát kód.
2. První spuštění testů – ověřuje se, že testy při prvním spuštění selžou. Většinou ani nejdou zkompilovat. Pokud testy projdou, tak je někde chyba v definici testu.
3. Psaní vlastního kódu – nyní se podle testů vytvoří zdrojový kód, tak aby testy prošly V této fázi není nutné dodržovat všechny konvence čistého kódu. Je hlavně důležité mít funkční kód.
4. Opětovné spuštění testů – nyní je možné opět spustit testy, které by již měli projít nebo se odhalí, kde jsou nedostatky. Výsledkem by tedy měl být již plně funkční zdrojový kód.
5. Úpravy kódu (refactoring) – vzhledem k tomu, že v druhém kroku byl kód psán, tak aby byl funkční bez ohledu na jeho čistotu, tak nyní je nutné kód očistit od duplicit a optimalizovat ho.

Jak je vidět z jeho postupu, tak se nejdříve snažíme o to, aby byl vytvořen funkční kód a teprve potom se staráme o to, aby byl kód čistý. Nevytváříme tedy žádnou dokumentaci, protože testy sami o sobě slouží jako dokumentace.

TDD je dobře použitelné, pokud je jasné, jak bude software řešen. V případě, že by se postupovalo metodou pokus, omyl, tak je velmi nevhodný a vytvoří víc škody než užitku, kvůli stálému přepisování testů. [34]

### 6.4.3 Data Driven testing

Zjednodušeně řečeno se jedná o využití jednoho testu, kdy pro různá vstupní data jsou očekávány různé výstupy. Test tedy využívá data z externího souboru, jako vstup pro určitý test. Díky tomu je možné snadno upravovat testovací data, přidávat nová data, aniž by byl nějak ovlivněn kód testu. Také je možné sdílet stejná testovací data pro více testů. [5, 35]

### 6.4.4 Testování doménové třídy

Jednotkové testování je bez problémů, pokud testujeme pouze entitní třídy. To jsou třídy, které nezávisejí na jiných našich testovaných třídách, mohou ovšem záviset na knihovnách třetích

stran. Běžně se ale používají i doménové třídy, kde jsou již závislosti na jiné naše třídy. Pro úspěšný test této třídy je tedy nutné, aby i závislé třídy úspěšně prošly testy. Zde je ale problém v tom, že je porušeno druhé pravidlo F.I.R.S.T. a to, že výsledek testu nesmí záviset na úspěšnosti jiného testu. Zde nastupuje technika mockování, kdy můžeme provádět izolované testy doménových tříd, stejně jako u entitních tříd. Mockování zjednodušeně znamená, že místo závislé třídy vytvoříme pouze objekt, který má její vlastnosti a díky tomu není nutné, aby závislá třída byla funkční. [1]

Pro provádění mockování je nutné využít nějaký framework např. EasyMock, Mockito anebo nějaký komplexnější nastroj jako je Spock. Ukázka mockování je v Zdrojový kód 11.3: Ukázka mockování.

# 7 NÁSTROJE PRO JEDNOTKOVÉ TESTY

Tato kapitola se věnuje nástrojům pro jednotkové testování. Většina kapitoly je věnována Spock frameworku, jeho základním vlastnostem a výhodám proti JUnit.

## 7.1 xUnit

Je to základní testovací knihovna pro jednotkové testování. Existuje pro velké množství jazyků, pro Javu je to konkrétně JUnit. V dnešní době je JUnit považován za průmyslový standart a je podporován ve všech moderních vývojových IDE. Velmi dobře jsou základní principy JUnit popsány v JUnit in Action Paperback [36].

## 7.2 Spock framework

Spock je testovací framework umožňující automatizovat nudný, opakující manuální proces testování. Byl vytvořen Peterem Niederwieserem v roce 2008 a byl inspirován existujícími testovacími nástroji s cílem vytvořit nástroj, který v sobě spojuje to nejlepší z nich. Spock je tedy spojení výhod JUnit, Mockito a Jbehave. Umožňuje tedy psaní jednotkových testů jako JUnit, jak pro Java třídy, tak i pro Groovy třídy. Obsahuje v sobě podporu pro mocking a stubbing, který je běžně nutno řešit přes nějaký externí framework, jako je Mockito. JUnit nemá standardně podporu mockování. Má také podporu pro Behavior Driven development a není tedy nutné využívat JBehave pro tyto účely. Spock je tedy vhodný jako nástroj pro podnikové aplikace a jejich procesy automatizace testování.


Obrázek 7.1: Z čeho se skládá Spock [6]

### 7.2.1 Groovy

Je to objektově orientovaný jazyk pro platformu Java. Je ji velice podobný, a proto je pro Java vývojáře velice snadné se s ní naučit. Pro spouštění je využívána JVM, takže jde spustit všude, kde je nainstalovaná Java, takže můžou vedle sebe existovat v jednom projektu. Skoro veškerý kód napsaný v Javě je validní i v Groovy. Dokáže ušetřit místo tím, že eliminuje zbytečný kód tzv. Boilerplate. V Groovy se dají např. velmi jednoduše zapisovat cykly nebo je velmi jednoduchá práce s datem a časem. Getter a setter jsou automaticky vytvářeny za běhu a nejsou nutné deklarovat ve třídě. Další velmi zajímavou vlastností je možnost inicializovat objekty prostým vyjmenováním jejich parametrů v konstruktoru. Jeho velkou výhodou je to, že je to dynamický jazyk. Není tedy nutné definovat typ proměnné a Groovy to udělá automaticky za běhu. Ovšem to znamená, že je oproti Javě pomalý, takže se nehodí do produkčního prostředí, ale jeho vlastnosti se dají využít pro psaní efektivních testů. Pro jednoduché mockování není zapotřebí žádný framework, Groovy má podporu pro jednoduché mocky. V souvislosti s možností inicializovat objekt pomocí jmenování parametrů v konstruktoru je možné použít pouze nějaké parametry anebo vnutit parametry pro které ani neexistuje konstruktor, což se obzvlášť hodí pro testování. Velká výhoda pro testy je možnost napsat název metody jako String, což velmi zpřehledňuje testy. Groovy lze využít pro psaní testů v JUnit, ale existuje i přímo testovací framework pro Groovy. [37]

### 7.2.2 Výhody Spocku

Podle [6] má Spock následující výhody:

- Podnikový nástroj – Spock je možné snadno integrovat s nejpopulárnějšími buildovacími nástroji Maven a Gradle. Spock běží jako součást buildovacího procesu a automaticky zprostředkovává výsledky testů.
- Obsáhlý – Spock má obrovský záběr v testování. Má zabudovaný nástroj pro mocking a stubbing. Dokáže otestovat jednotlivé třídy, moduly i celou aplikaci.
- Známý/kompatibilní – testy se spouští stejně jako JUnit testy. Je možné používat Spock i JUnit testy v jednom projektu.
- Inspirovaný – Spock je relativní nováček na scéně testování, ale bere si to nejlepší z existujících testovacích knihoven.

- Stručnost – používá jednoduchou syntaxi a spolu s Groovy je možné se vyvarovat zbytečné upovídanosti kódu ve které se ztrácí pravý smysl testu.
- Čtivý – Díky struktuře kódu jsou testy dobře čitelné i pro netechnicky zaměřené lidi (management).
- Pečlivý – pokud test selže, tak Spock poskytuje velmi detailní výpis chyby, ze kterého se dá snadno pochopit kde nastal problém.
- Rozšiřitelnost – Spock umožňuje dopsání vlastních rozšíření podle potřeb. Několik z dnes již implementovaných vlastností, bylo na začátku pouze rozšířením.

### 7.2.3 Testovací bloky

Spock test je přehledně rozdělen na několik bloků. Každý blok může mít své vlastní pojmenování.

`Given` – zde se inicializuje veškerý kód, který bude potřeba v průběhu testu. Tento kód je možné umístit i do when bloku a given přeskočit, ale není to doporučená praktika.

`Setup` – je to alternativa ke given. Rozdíl je pouze sémantický a záleží pouze na preferencích uživatele.

`When` – nejdůležitější blok, který je přítomný téměř ve všech testech. Měl by být, pokud možno krátký a výstižný, protože to bývá první blok, na který se každý zaměří při čtení kódu. V tomto bloku je kód, který něco provádí, spouští.

`Then` – obsahuje tvrzení, které se ověřuje proti výsledku testu.

`And` – umožňuje rozdělit blok na několik částí a tím zlepšit jeho srozumitelnost, protože každý `and` může mít svůj popis.

`Expect` – jeho význam se mění podle toho kde je použit. Nejčastěji se používá jako náhrada bloku `given-when-then`, kdy obstará všechny činnosti těchto bloků. Stejně tak může nahradit pouze bloky `when` a `then`, kdy se provede nějaká akce a zároveň její vyhodnocení.

`Where` – využívá se pouze u parametrizovaných testů, kde se do něho zapisují parametry těchto testů.

`Cleanup` – kód obsažený v tomto bloku je vždy spouštěn až na konci testu, i když test selhal. Slouží tedy pro úklid po testu.

### 7.2.4 Životní cyklus Spock testu

Pokud máme v testovací třídě více testů, tak se může stát, že v některých blocích budeme psát stále stejný kód. Týká se to třeba bloků `setup` a `cleanup`. Tyto bloky je tedy možné nahradit metodami. Metody `Setup` a `Cleanup` se spustí před/po vykonání každého testu v příslušné třídě. Někdy není potřeba provádět kód znovu a znovu před/po každé metodě a k tomu slouží `setupSpec` a `cleanupSpec`, které se vykonají pouze jednou. Pokud potřebujeme nějaký objekt, který bude inicializován pouze jednou pro všechny testy, tak je možné použít anotaci `@Shared`. [6]

### 7.2.5 Parametrizované testy

Pokud potřebujeme testovat stejné metody, ale s jinými parametry, tak je zbytečné psát duplikátní testy, které se liší pouze v zadaných parametrech. Lepší řešení jsou parametrizované testy, kdy testovací kód napíšeme pouze jednou a spouštíme ho pouze s různými parametry. Právě pro zápis parametrů slouží blok where. Ten musí být vždycky uveden až jako poslední jinak se test nespustí. Parametry se zadávají ve formě datové. Data jsou zapsána ve formě tabulky, kdy hlavičky sloupců označují název proměnné a každý řádek obsahuje proměnné pro jednu iteraci testu. Sloupečky jsou od sebe odděleny buď jednou svislou čarou, kterou se oddělují sloupečky a dvě svislé čáry označují, kde končí vstupní parametry a začínají výstupní parametry testu. Where blok musí vždy obsahovat alespoň dva sloupečky. Pokud tento parametrický test spustíme, tak se na jednom řádku vypíše, zda uspěl nebo selhal. Pokud doplníme nad test anotaci `@Unroll`, tak již dostaneme výsledek rozepsaný pro každý řádek zvlášť. Abychom již z výpisu snadno zjistili, který test selhal je možné zajistit, aby se do názvu metody zapsala i hodnota proměnné. To lze zajistit pomocí mřížky (#) a názvu proměnné. Potom se ve výpise již objeví i název proměnné. Jako parametry se nemusí zadávat jenom hodnoty, ale i výrazy. Ukázka parametrického testu je Zdrojový kód 11.1: Ukázka jednotkového *testu* a jeho vyhodnocení na Obrázek 11.1: Ukázka výsledku testu PrepareUrlForArchiveSpec.

### 7.2.6 JUnit vs. Spock

Zde bude uvedeno několik rozdílů mezi JUnit 5.0 a Spock 1.1. Pro člověka znalého JUnit to bude určité seznámení se Spockem.

- Označení testů

JUnit používá anotaci `@Test` označující metodu, která má být spouštěna jako test. Nelze označit celou třídu a metoda nesmí vracet hodnotu.

U Spocku stačí u třídy přidat extends Specification a všechny metody uvnitř třídy budou brány jako testovací.

- Jména testů

JUnit dříve používal klasické jednoslovné názvy. V nové verzi umožňuje pomocí anotace `@DisplayName` přidat popis jako stringový řetězec.

Spock nevyužívá žádnou anotaci, ale v groovy se názvy metod zapisují přímo jako String řetězec.

- Neprovádění testu

Pokud chceme v JUnit neprovádět některé z testů, tak stačí přidat anotaci `@Disabled` nad metodu anebo přímo nad celou třídu.

Spock umožňuje pomocí anotace `@Ignore`, aby test nebyl prováděn. Dále nabízí i anotaci `@NotYetImplemented`, která označuje test, který ještě nebyl plně implementován.

- Očekávání výjimek

Pokud očekáváme v JUnit testu výjimku, tak stačí kód obalit metodou `assertThrows`.

Spock to řeší pomocí anotace `@FailsWith`, kde se do závorky uvede, k jaké výjimce může dojít.

- Opakování testu

JUnit pomocí anotace `@RepeatedTest` umožňuje zvolit kolikrát daný test proběhne.

Spock tuto funkcionalitu nepodporuje, takže je nutné zajistit opakování přímo v těle testu.

- Timeout

JUnit využívá metodu `assertTimeout`, kde stanoví, jak maximálně dlouho může být testovací metoda vykonávaná.

Spock má jednoduchou anotaci `@Timeout`, která může být pro metodu nebo i pro celou třídu.

- Parametrizované testy

V JUnit je nejdříve nutné k metodě přidat anotaci `@ParamerizedTest`, která značí, že test bude prováděn vícekrát. Pomocí anotace `@MethodSource` můžeme odkázat na metodu, která poskytuje testovací data.

Spock to řeší velmi jednoduše, pomocí tabulky, kde jsou přehledně zapsaná veškerá data. Pokud chceme zobrazovat výsledky pro jednotlivé hodnoty, a ne pouze výsledek celého testu, je nutné nad testovací metodu přidat anotaci `@Unroll`. Podrobně byly parametrizované testy popsány v 7.2.5.

- Mocking

JUnit nemá nativní podporu pro mockování, takže je nutné používat externí knihovnu, jako třeba populární Mockito.

Spock má v sobě integrovanou podporu pro mock a stub objekty. Stačí využít metody Mock a předat jí jako parametr objekt pro mockování. [1, 38]. Ukázka Zdrojový kód 11.3: Ukázka mockování.

# 8 NÁSTROJE PRO TESTY UŽIVATELSKÉHO ROZHRANÍ

## 8.1 Selenium

Slouží pro testování GUI webových aplikací. Je ve vývoji již od roku 2004 a za tu dobu se stal jedním z nejpoužívanějších nástrojů pro funkční testování. Je šířen pod svobodnou licencí Apache 2.0. Skládá se ze Selenium IDE a Selenium WebDriver.

### 8.1.1 Selenium IDE

Je dostupný jako plugin pro Firefox do verze 54. Od verze 55 není podporován [39]. Jedná se o velmi jednoduchý nástroj k vytvoření testu. Pouze se zapne nahrávání a test se nakliká, poté je možné ho v daném prohlížeči spouštět. Jedná se tedy o vytváření skriptu, který je pomocí něho možné opakovaně spouštět. Posloupnost seleniových příkazů se nazývá Selenese. Příkazy se dělí do tří základních kategorií [40]:

- Action, která mění stav aplikace. Je to například kliknutí na nějaký odkaz, výběr hodnot z roletky, za kliknutí checkboxu.
- Accessory, které kontrolují stav aplikace a ukládají jí do proměnné. Například kontrolují, jestli se daný prvek nachází na stránce a ukládá hodnotu jako true nebo false.
- Potvrzovací accessory, které ověří, že stav aplikace je v souladu s očekávaným stavem. Například, že je správný titul stránky.

Script lze uložit do vybraného programovacího jazyka a následně upravovat. Tento nástroj je vhodný pouze pro otestování jednoduché aplikace.

### 8.1.2 Selenium WebDriver

Poskytuje API k tomu, aby bylo možné vytvářet automatizované testy v programovacím jazyce. Na výběr jsou Java, Python, Ruby, C# a Javascript. Následně je možné scripty spouštět nejen ve Firefox, ale i dalších prohlížečích včetně HtmlUnit, který umožňuje otestovat webovou aplikaci bez nutnosti jejího zobrazení. Ovšem není možné automaticky testovat pro všechny dostupné prohlížeče. V rámci testu je nutné zvolit pouze jeden driver pro určitý prohlížeč. [1]

WebDriver potřebuje nějaký identifikátor pro každý element se kterým pracuje. Existují tři základní přístupu:

- Pomocí jednoznačného id,
- pomocí CSS selektoru,
- pomocí XPath selektorů.

Pokud test v nějaké své části selže, tak je možné nastavit, aby byl sejmut snímek obrazovky, aby bylo možné přesně identifikovat, kde se stala chyba. Vylučuje se tím tedy ruční spouštění testu a hledání v kterém přesně okamžiku se stala chyba.

## 8.2 Návrhové vzory

Každý software je sice jiný, ale při jeho vývoji se vyskytují velice podobné problémy. Pro řešení těch nejčastějších existují návrhové vzory. Návrhový vzor je postup, jak daný problém vyřešit co nejlépe (tzv. best practise). Návrhový vzor tedy poskytuje jakousi šablonu, jak daný problém nejlépe vyřešit.

### 8.2.1 Page Object pattern

Selenium test lze jednoduše naklikat a potom jenom spouštět. Při testování komplexní aplikace se ale jednotlivé akce začnou opakovat, takže budeme mít zbytečně duplikátní kód. Takže je nutné začít testy upravovat a extrahovat opakující se kód. Přístup různých pomocných metod a tříd se brzo stane značně nepřehledné. Zde nastupuje návrhový vzor Page Object, který v roce 2009 představil Simon Stewart [41].

Tvoříme dva druhy tříd testy a stránky (Page Objects). Stránky obsahují všechny metody, které daná stránka nabízí uživatelům. Martin Fowler v [42] doporučuje vytvářet stránku pro každou významnou komponentu na stránce. Zde se pracuje přímo se Selenium API, neprobíhají zde žádné asserty. Každá metoda vrací objekt typu Page, což je stránka, na kterou přechází nebo vrací sama sebe. V testech se pouze skládá dohromady volání stránek podle testovacího scénáře a obsahuje potřebné asserty. Nepracuje se zde tedy přímo s uživatelským rozhraním stránky, takže při jeho změně stačí provést změnu v Page objektu a není nutné nijak měnit testy [43].

Tento návrhový vzor, ale není dokonalý. Dost častý problém jsou velké třídy [44]. Pokud je na stránce 10 elementů, tak se není problém dostat přes 200 řádků kódu v jedné třídě [45]. V takové třídě je velmi obtížné se orientovat. Jedno řešení nastínil Martin Fowler v [42], kde

uvádí, že objekt Page by se neměl vytvářet pro každou stránku, ale pro každou významnou komponentu. Ale ani při tomto přístupu není zaručeno, že u rozsáhlých komponent nevzniknou dlouhé třídy a není porušen princip SOLID [46]. Page Object třídy typicky porušují první dva.

**Single Responsibility Principle**

Každá třída by měla mít pouze jednu zodpovědnost. Jinak řečeno třída by měla zodpovídat pouze za to co je jasně řečeno jejím názvem. Díky tomu by při návrhu mělo vznikat více malých tříd než malé množství velkých tříd, které dělají několik věcí najednou. Při dodržení tohoto principu je pro vývojáře velice jednoduché provádět úpravy [47]. Page Object porušuje tím, že zajišťuje abstrakci elementů na stránce a zároveň popisuje úkoly, které lze provést pomocí těchto elementů.

**Open Closed Princile**

Podle tohoto principu by třídu mělo být možné rozšířit pouze tím, že budeme přidávat pouze nový kód, ale nebude nutné modifikovat ten stávající. Aby bylo možné dodržet tento princip, tak je nutné využívat abstrakci a polymorfismus [47]. U Page Object je princip porušen, pokud dojde k přesunu nějakého elementu na jinou stránku, tak musíme změnit kód v původním Page objektu, který náleží stránce, ze kterého je element původně, tak i v Page objektu, který náleží stránce, na které je element nově umístěn.

Možnost, jak Page Object pattern upravit, aby vyhovoval SOLID principu existují, ale nikdy je nevyřeší úplně dokonale [41].

Zde přichází na řadu návrhový vzor Screenplay.

### 8.2.2 Screenplay pattern

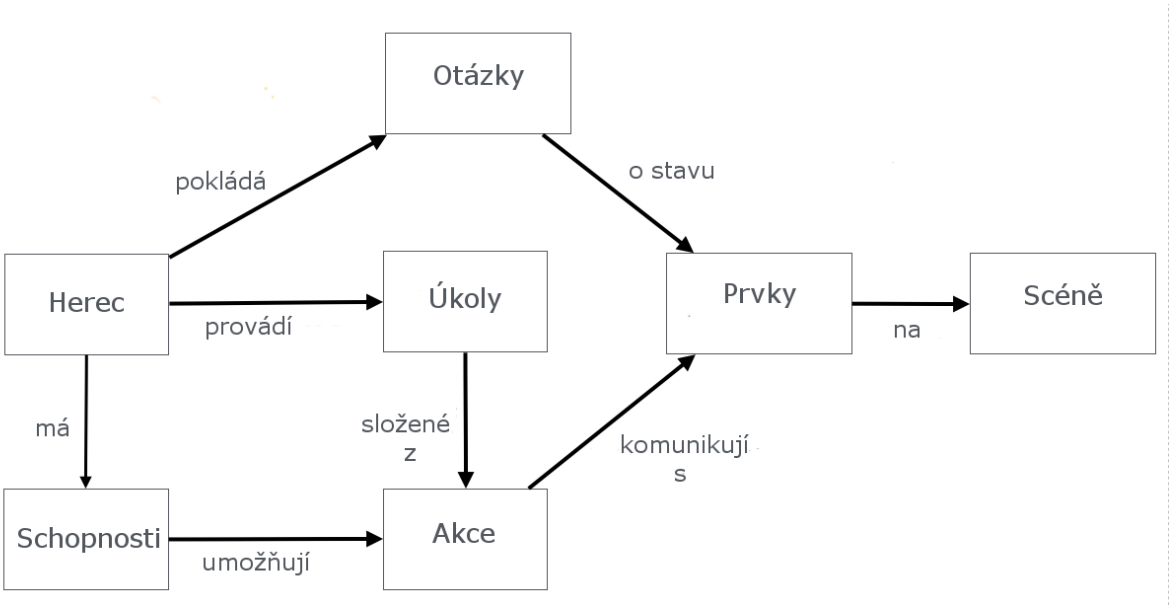
Dříve známý také jako Journey Pattern. Původně představen v roce 2007 Antony Marcanem, takže jeho myšlenka je starší nežli Page Object Pattern a vznikla nezávisle na něm.

Na testy je možné pohlížet jako na seznam pokynů, které se provádějí. Ale existuje i jiná cesta, jak se dívat na testy:

- **Role –** Kdo to je?
- **Cíle –** Proč jsou tady a jaký výsledek od nich očekáváme?
- **Úkoly –** Co musíme udělat pro dosažení cílů?
- **Činnosti –** Jak splnit každý jednotlivý úkol prostřednictví daných kroků?

A to je základ návrhového vzoru Screenplay. Podle toho sestavíme scénář testu, odtud název vzoru Screenplay (scénář). Ze scénáře můžeme jednoduše zjistit, že:

- Role – nějaká osoba, která má specifickou roli se nazývá herec. Herec, který hraje roli, tak vykonává každý úkol v zadaném scénáři. Ve scénáři může být více herců, kdy každý z nich má k dispozici vlastní instanci prohlížeče. Každý herec má k dispozici nějakou schopnost např. prohlížení webu.
- Cíle – Co je cílem celého scénáře.
- Úkoly – K tomu, aby byly dosažený cíle, tak musí být splněny jednotlivé úkoly. Úkoly jsou plněny hercem.
- Činnosti – Aby mohl být splněn nějaký dílčí úkol, tak musí být definován sled událostí, které k tomu vedou.




Obrázek 8.1: Screenplay diagram [41]

Na Obrázek 8.1 je názorně vidět podle jakého schéma je celý scénář vystaven. Z něho vidíme, že se nejvíce času věnuje tomu, jak byl úkol vykonán, a ne tolik jak byl vytvořen, díky tomu jsou hlavní třídy velmi přehledné a je z nich vidět co se děje v testu. [48]

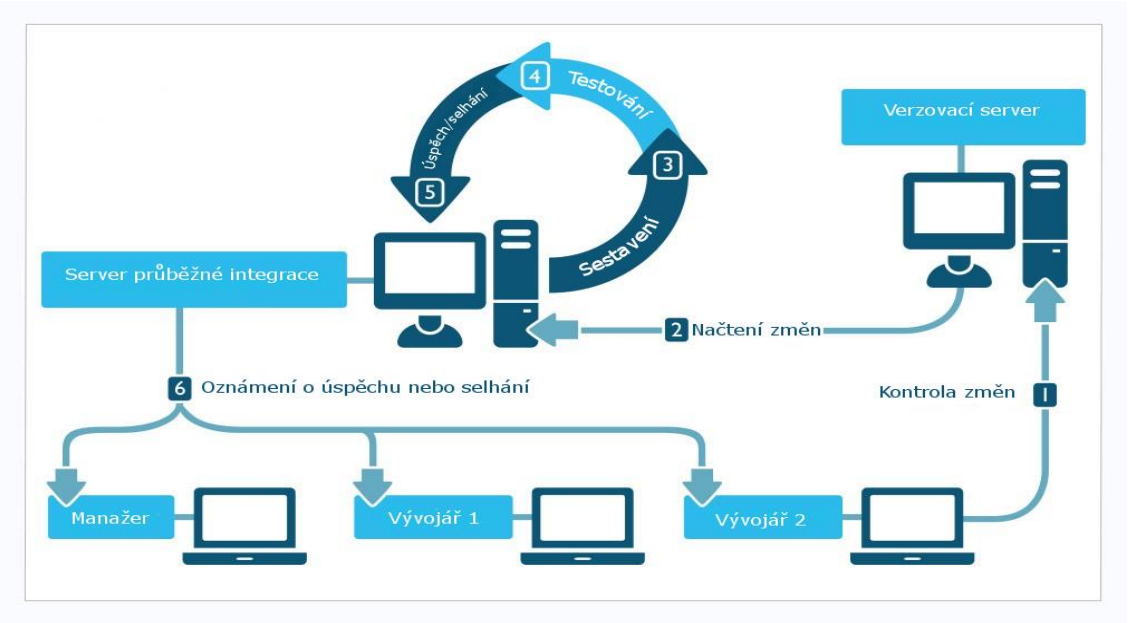
# 9 CONTINUOUS INTEGRATION

## 9.1 Definice

Asi nejlepší definici CI poskytuje jeden z jejich autorů Martin Fowler v [49]: „CI je praxe rozvoje softwaru, kdy členové týmu často integrují svou práci, obvykle každý člověk integruje alespoň jednou denně – což vede k mnoha integracím za den. Každá z těchto integrací je ověřena automatizovanou sestavou (včetně testování), která detekuje integrační chyby co nejrychleji.“

Postup při využívání CI je:

1. Programátor pushne novou verzi na centrální uložiště.
2. CI server detekuje změnu ve zdrojovém kódu a stáhne si nejnovější verzi.
3. Na CI serveru je proveden build.
4. CI server spustí testy.
5. Build a testy uspějí nebo selžou.
6. Odešle se zpráva o úspěchu či neúspěchu.




Obrázek 9.1: Proces průběžné integrace [50]

Hlavním důvodem pro používání CI je tedy zamezení, aby se k ostatním dostala verze, která obsahuje chyby. [51]

## 9.2 Nástroje pro CI

Existuje mnoho nástrojů pro průběžnou integraci. Liší se podporou programovacích jazyků nebo náročností konfigurace.

### 9.2.1 Jenkins

Patří mezi nejznámější a nejrozšířenější CI server. Díky tomu má obrovskou podporu od komunity a je možné do něho doinstalovat obrovské množství pluginů a upravit téměř jakoukoliv část jeho rozhraní. Podporuje všechny nejznámější systémy pro správu verzí kódu. Je zdarma a po stažení ho stačí pouze spustit. V základu, ale podporuje pouze Javu. Další funkcionality je nutné doinstalovat. [52]

### 9.2.2 TeamCity

Pochází od studia Jetbrains, stejně jako IntelliJ IDEA. Díky tomu, že je napsán v jazyce Java, tak je možný provozovat na všech systémech, kde je podpora Javy. Podporuje všechny hlavní systémy pro správu verzí kódu jako je Git a Mercurial. Je možné ho rozšiřovat pomocí pluginů, a tak přizpůsobit pro potřeby každého uživatele. Průběh buildů a testů je dostupný přes přehledné webové rozhraní. Produkt je placený, ale pro open source poskytuje licenci zdarma. [53]

### 9.2.3 Travis CI

Na rozdíl od předchozích se jedná o webovou službu, která je propojená s GitHubem, takže není možné používat jiný systém pro správu verzí kódu, ani mít svou lokální instalaci. Pro open source je zdarma. Pokud používáte privátní úložiště na GitHubu, tak je zpoplatněn.

Služba na základě změny zdrojového kódu v úložišti spouští sestavení a testování aplikace. V případě neúspěchu zasílá vývojáři upozornění na mail. Oproti přechozím neposkytuje takové množství nastavení, ale díky své jednoduchosti je vhodný pro malé nebo individuální projekty.

Pro konfiguraci se používá soubor travis.yml, který je psaný v jazyce YAML, a musí být umístěn v kořenovém adresáři projektu. [54]

# 10 NÁVRH TESTOVANÉ APLIKACE

V praktické části byla vyvinuta aplikace pro sběr komentářů ze zpravodajských portálů. Aplikace byla vyvíjena tak, aby bylo možné snadno přidávat moduly pro další zpravodajské portály. V aplikaci jsou tedy zastoupeny servery iDnes, Novinky a Lidovky.

Aplikace v předem stanovený čas automaticky stáhne všechny články z předchozího dne a k nim patřící komentáře. Zároveň jsou sesbírány nové komentáře u článků za poslední tři dny. U každého portálu je uloženo datum posledního sběru. Všechny tyto údaje jsou uloženy do databáze.

## 10.1 Použité technologie

V této podkapitole budou popsány technologie použité pro vývoj aplikace, s výjimkou Travis, Spock a Groovy, které již byly popsány v předchozích kapitolách.

### 10.1.1 Spring

Jedná se o framework určený pro usnadnění vývoje J2EE aplikací. Jádro je postaveno na návrhovém vzoru Inversion of Control, který zajišťuje odstranění těsných vazeb (tight coupling[1]). Zodpovědnost za provázání objektů je přesunuta z aplikace na framework, který vytváří volné vazby (loose coupling). Toho je docíleno využívání Dependenci Injection(DI), který řeší předávání závislostí mezi objekty. Závislosti předává pomocí tří základních způsobů: constructor injection, setter injection a interface injection. [55]

### 10.1.2 Hibernate

Jedná se o nástroj pro objektově-relační mapování(ORM). Jde o jednu z implementací JPA. Slouží k převodu mezi objektově orientovanými jazyky a relačními databázemi. Zjednodušuje práci s daty, tím, že stačí pouze napsat entitní třídy, které budou představovat tabulky v databázi. Jednotlivé objekty tříd jsou potom reprezentovány jako řádky v tabulce. Pro programátora je to velké zjednodušení, protože se nemusí zaobírat tím jaká je použitá databáze, protože používá stejné API Hibernate u všech databází. [56]

[1] Mezi třídami je zbytečné množství vazeb, které díky provázanosti znemožňují modifikaci. Jedná se o jeden z antipaternů.

### 10.1.3 Jsoup

Jedná se o Java knihovnu pro práci s HTML. Poskytuje rozhraní pro extrahování a manipulaci s daty. Používá DOM, CSS a jQuery metody. Implementuje HTML5 podle specifikace WHATWG a převádí HTML do stejného DOM jako dnešní prohlížeče. Při návrhu této knihovny bylo dbáno, aby bylo možné pracovat se všemi verzemi HTML. Díky tomu, že je šířen pod licencí MIT, tak je možné ho i dále modifikovat a vylepšovat.

Jsoup nabízí:

- Získávání HTML dokumentů z URL, soboru nebo String řetězce.
- Nalezení a extrahování dat pomocí procházení DOM nebo CSS selektorů.
- Manipulaci s HTML elementy, atributy a texty pře uložením na disk.
- Vyčištění uživatelem odesílaným obsahem proti whitelistu, pro předcházení XSS útoku.
- Úhledný výstupní HTML.

Jsoup dokáže získat všechen statický obsah stránek, který je vykreslený na straně serveru. Nedokáže, ale získat dynamickou část webu, která může být napsána např. v JavaScriptu. [57]

## 10.2 Infrastruktura běžící aplikace

### 10.2.1 PostgreSQL

Jedná se o objektově-relační databázový systém. Jedná se open source, který běží na systémech Linux, UNIX a Windows. Svými vlastnostmi se podobá Oracle Database. Stejně jako Oracle využívá pro své operace jazyku SQL. Funkce je možné zapisovat pomocí PL/pgSQL, který je podobný PL/SQL známému procedurálnímu jazyku Oracle DB. Splňuje všechny požadavky ACID na bezpečný transakční systém. [58]

### 10.2.2 Heroku

Je to cloudová platforma, která umožňuje hostování aplikaci napsaných pomocí Node, Ruby, Java, PHP, Python nebo Go. Umožňuje také využívat Heroku Postgres jako databázi. Základní verze hostingu včetně databáze je poskytována zdarma. Pokud potřebujeme větší databázi, další služby, tak je možné kdykoli si připlatit za lepší a vždy platit pouze to co potřebujeme. [59]

## 10.3 Nástroje použité při vývoji

### 10.3.1 IntelliJ IDEA

Jedná se o moderní vývojové prostředí (IDE) pro Java a jazyky na ní založené. Poskytuje velké množství doplňků, které usnadňují práci.

Stejně jako každé moderní IDE poskytuje kontrolu kódu, dokončování klíčových slov, správu importů a mnoho dalších. Umožňuje se připojit a spravovat databáze, takže ulehčuje práci s ní. [60]

### 10.3.2 Maven

Slovo Maven pochází z jidiš[2] a dal by se přeložit jako znalec. Jedná se nástroj pro správu a automatizaci buildů. Je určen pro všechny projekty založené na Javě. Dle [61] poskytuje pět základních vlastností:

- Zjednodušení procesu buildování projektu.
- Poskytování jednotného buildovacího systému.
- Poskytování kvalitních informací o projektu.
- Poskytování průvodce pro nejlepší programátorské praktiky.
- Umožnění jednoduché migrace na nové funkce.

K popsání všech závislostí, externích knihoven, zásuvných modulů, procesu buildování využívá soubor Project Object Model (POM). [62]

### 10.3.3 GitHub

Pro uchovávání kódu, včetně jeho historie se využívá verzovací systém Git. GitHub, slouží jako server pro hostování těchto repositářů. Pro open source projekty je zdarma. Za používání soukromého repositáře se ovšem musí platit. GitHub také jako jediný umožnuje propojení s CI serverem Travis, který umožňuje spuštění a otestování každé verze po jejím nahrání do repositáře. [63]

[2] Jazyk, podobný němčině, který používá část židů.

## 10.4 Samotná implementace

Pro vývoj nebyl použit vývoj řízený návrhem, kdy by se nejdříve navrhly třídy s metodami pomocí UML a následně z nich byly vygenerovány kostry tříd a doplněna implementace. Pro většinu aplikace využita extrémní programovací technika zvaná vývoj řízený testy (TDD).

### 10.4.1 Metodika při vytváření pomocí TDD

Nejdříve byl napsán test. Na obrázku je jednoduchý test pro otestování parsování URL pomocí Jsoup. Test projde, pokud je nějaký dokument načten.

```
def "parse url and return as jsoup document"() {
    given: "prepare url"
    def url = "https://www.idnes.cz/"
    when: "return parse document from url"
    Document document = parseUrl.parse(url)
    then: "document is not null"
    document != null
}
```

Zdrojový kód 10.1: Ukázka testu parsování URL

Teprve poté následuje napsání metody podle testu.

```
public Document parse(String urlString) throws IOException {
    return Jsoup.connect(urlString).get();
}
```

Zdrojový kód 10.2: Implementace metody parse

### 10.4.2 Struktura aplikace

V příloze A této práce je vidět struktura src složky projektu. Je zde jasně vidět rozdělení na Java kód samotné aplikace a testy v Groovy. Do databáze jsou ukládány entity `Article`, `Comment` a `Portal` umístěné ve stejnojmenné složce. Pro práci s databází jsou určeny třídy ve složce `springData` pro každou z entit. Každý z portálů má svoji složku, ve které se nacházejí třídy pro extrakci článků a komentářů, individuální pro každý ze zpravodajských serverů. Složka extractor obsahuje třídy společné pro všechny servery a rozhraní, které musí implementovat. Třída `Init` a `ScheduledTask` slouží pro samotné spouštění aplikace.

Testy jsou rozděleny na databázové, společné pro všechny portály. V individuálních složkách portálů jsou třidy, které se starají o přípravu testovacích dat pro daný portál. Ve složce resources se nalézají zdrojové HTML soubory pro testy.

### 10.4.3 Pravidelná aktualizace dat pomocí automaticky spouštěného cronu

Aby bylo zajištěno automatické stahování článků a komentářů, tak bylo nutné využít plánovač. Ve Springu jde plánovač jednoduše definovat pomocí anotace `@Scheduled`, kde se definuje kdy se má spustit. Zdrojový kód 10.3: Ukázka metody plánovače ukazuje, že plánovač je nastaven na každý den v 1:30 ráno středoevropského času. Spustí se vždy extrahování a uložení článku z předchozího dne a extrahování a uložení nových komentářů z předchozích dnů. Vše se provádí pro každý portál, který implementuje rozhraní `IPortalExtractor`.

```
// set crone to run every day at 1:30 morning CET
@Scheduled(cron = "0 30 1 * * *", zone = "Europe/Prague")
public void scheduledRun() {
    // getting all beans extends IPortalExtractor
    Map<String, IPortalExtractor> extractors =
applicationContext.getBeansOfType(IPortalExtractor.class);
    log.info("Time is ", dateFormat.format(new Date()));
    // iteration through collection
    for (IPortalExtractor portalExtractor : extractors.values()) {
        try {
            // start saving yesterday articles and comments
            run.extractAndSaveYesterday(portalExtractor);
            // start updating comment for 7 days ago
            update.update(NUMBER_OF_DAYS_FOR_UPDATE, portalExtractor);
            Portal portal =
portalSpringDataRepository.findByName(portalExtractor.getPortalName()).get(
0);
            // setting date of last collection for portal
            portal.setLastCollection(new Date());
            portalSpringDataRepository.save(portal);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Zdrojový kód 10.3: Ukázka metody plánovače

Aby byl plánovač aktivní, tak je také nutné přidat anotaci `@EnableScheduling` do hlavní třídy aplikace a tím ho povolit.

# 11 METODOLOGIE TESTOVÁNÍ

Pro testování aplikace jsou využívány jednotkové, integrační a databázové testy. Pro jejich napsání byl zvolen Spock framework, který využívá jazyk Groovy. Některé z testů vyžadují připojení k internetu, protože se přímo připojují k některému ze zpravodajských portálů. Pokud to bylo možné, tak pro účely testů byly stránky staženy a uloženy do resources, aby nebyly ovlivněny změnami na straně serveru.

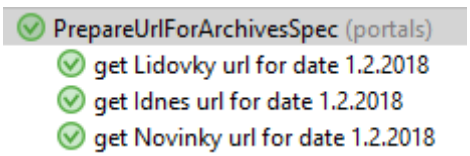
## 11.1 Jednotkový test

Níže je ukázka jednoho z jednotkových testů. Tento test je využíván pro všechny portály a v budoucnu je možné snadno přidat další portál jako řádek do parametrů v bloku `where`. To umožňuje snadnější testování nově přidávaných portálů.

```
@Unroll
def "get #portalName url for date 1.2.2018"() {
    given: "preparing extractor and date"
    def extractor = this."prepareUrlForArchives${portalName}"
    SimpleDateFormat sdf = new SimpleDateFormat("dd.MM.yyyy")
    String dateInString = "01.02.2018"
    Date date = sdf.parse(dateInString)
    when: "getting resulting url"
    String url = extractor.prepareUrl(date)
    then: "check if returning url is the same as we expected"
    url == result
    where: "parameters for test"
    portalName | result
    "Lidovky"  |
"https://www.lidovky.cz/archiv.aspx?datum=01.02.2018&idostrova=ln_lidovky"
    "Idnes"    |
"https://zpravy.idnes.cz/archiv.aspx?datum=01.02.2018&idostrova=zpravodaj"
    "Novinky"  | "https://www.novinky.cz/archiv?id=966&date=01.02.2018"
}
```

Zdrojový kód 11.1: Ukázka jednotkového *testu*

Díky anotaci `@Unroll` jsou výsledky testů rozdělené pro každý portál zvlášť a je tedy dobře vidět, který z testů projde a který ne.




Obrázek 11.1: Ukázka výsledku testu PrepareUrlForArchiveSpec

## 11.2 Databázový test

Všechny databázové testy se nacházejí ve třídě `DatabaseSpec`. Níže je ukázka testu, který ověřuje, zda je funkční vyhledávání portálu podle jeho jména. Je zde využita jedna z velkých výhod Spocku a to, že není nutné vytvářet nový objekt `Portal` pomocí konstruktoru, ale stačí mu nastavit pouze parametry, které potřebujeme k testu.

```
@Rollback
def "find portal by name"() {
    given: "save new portal to db"
    Portal portal = portalSpringDataRepository.save(name: new
Portal("iDNES"))
    when: "finding portal by name"
    List<Portal> portalList =
portalSpringDataRepository.findByName("iDNES")
    then: "check if any portal exist"
    portalList.contains(portal)
}
```

Zdrojový kód 11.2: Ukázka databázového testu

Databázi je také možné testovat pomocí falešných mock objektů, jak je vidět níže. Více v kapitole 6.4.4.

```
def "mock save portal"() {
    given:
    def portal = new Portal(name: "iDnes")
    IPortalSpringDataRepository iPortal = Mock()
    when:
    iPortal.save(portal)
    then:
    1 * iPortal.save(portal)
}
```

Zdrojový kód 11.3: Ukázka mockování

## 11.3 Integrační test

Zde se již jedná o integrační test, který testuje funkčnost celé třídy `ExtractArticle`. Ta má za úkol extrahovat všechny články z archivu pro daný den. Zde se testuje, zda počet článků extrahovaných z archivu odpovídá skutečnému počtu článků. Kvůli přehlednosti jsou testovací data odděleny ve vlastní třídě a zde se pouze volají metody pro získání URL a pro ověření počtu.

```
@Unroll
def "number of articles in #portalName archive for one day"() {
    given: "preparing extractor for portals"
    def extractor = this."extractArticle${portalName}"
    when: "extract number of articles in archive"
```

```
    List<Article> articleList =
extractor.findArticles(preparedData.getUrlForArchive())
    then: "comparing size of list with real number of article in archive"
    articleList.size() == preparedData.getNumberOfArticlesInArchive()
    where: "parameters for test"
    portalName | preparedData
    "Lidovky"  | new ExtractArticlesLidovkyPreparedData()
    "Idnes"    | new ExtractArticlesIdnesPreparedData()
    "Novinky"  | new ExtractArticlesNovinkyPreparedData()
}
```

Zdrojový kód 11.4: Ukázka integračního testu

## 11.4 Abstrakce na úrovni automatizovaných testů pro jednotlivé portály

Pro zajištění Open Closed principu je nutné, aby při přidávání dalších portálů nemusel být měněn zdrojový kód ve spouštěcích třídách `Update`, `Run`, `Init` a `ScheduledTask`. To je zajištěno tím, že každý portál má třídu `PortalExtractor`, která implementuje jednotné rozhraní `IPortalExtractor` a v těchto třídách jsou volány pouze metody z tohoto rozhraní.

### 11.4.1 Možnost přidávání dalších modulů

Přidávání dalších modulů je velice jednoduché. Stačí pouze do projektu přidat třídu, která bude implementovat rozhraní `IPortalExtractor` a je komponentou. Zdrojový kód tohoto rozhraní se nalézá v příloze B této práce. Projekt také obsahuje Javadoc pro toto rozhraní. Aplikace využívá jednu z možností Springu. Třídy `Init` a `ScheduledTask` si načítají všechny beany v aplikačním kontextu, které implementují rozhraní `IPortalExtractor` a v cyklu pro ně volá dané metody deklarované v rozhraní.

Aby bylo po přidání nového portálu zajištěno jeho správné vložení do databáze je určena metoda `initPortals` ve třídě Init. Ta projde všechny portály, které implementují třídu `IPortalExtractor` a zjistí, jestli už je v databázi. Pokud ne, tak ji přidá záznam do tabulky portal a zajistí úvodní naplnění daty za poslední týden.

```
public void initPortals() {
    // get all beans of type IPOrtalExtractor
    Map<String, IPortalExtractor> extractors =
applicationContext.getBeansOfType(IPortalExtractor.class);
    // iteration thought collection
    for (IPortalExtractor portalExtractor : extractors.values()) {
        try {
            // checking if portal exist in db
            if
(portalSpringDataRepository.findByName(portalExtractor.getPortalName()).siz
e() == 0) {
                // save new portal to table
```

```
                portalSpringDataRepository.save(new
Portal(portalExtractor.getPortalName(), portalExtractor.getUrl(), new
Date()));
                // extract and save yesterday article and comment
                run.extractAndSaveYesterday(portalExtractor);
                // extract and save article and comment for the last week

run.extractAndSaveMultipleDaysBefereYesterday(NUMBER_OF_DAYS,
portalExtractor);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Zdrojový kód 11.5: Ukázka inicializační metody pro nový portál

### 11.4.2 Využití automatizace pro testovaní dalších modulů

Většina testů, u kterých to má nějaký smysl je parametrická, takže v případě přidávání nového portálu je možné snadno přidat nový řádek do těchto testů a podle nich udělat implementaci. Jako příklad bude uvedena tvorba metody `getKeywords`, která získává klíčová slova z článku. Situace je z doby, kdy byly v aplikaci pouze dva funkční zpravodajské portály, iDnes a Lidovky, a přidával se server Novinky. Test pro metodu `getKeywords`, již existoval, stejně jako parametrický test pro otestování této metody.

```
@Unroll
def "get keywords from article on #portalName"() {
    given: "prepared extractor"
    def extractor = this."extractMetaFromArticle${portalName}"
    when: "get keywords"
    String keywordsFromArticle =
extractor.getKeywors(preparedData.getArticleAsDocument())
    then: "compare keywords from article with real keywords"
    keywordsFromArticle.equals(preparedData.getKeywords())
    where: "parameters for test"
    portalName | preparedData
    "Lidovky"  | new ExtractMetaFromArticleLidovkyPreparedData()
    "Idnes"    | new ExtractMetaFromArticleIdnesPreparedData()
}
```

Zdrojový kód 11.6: Ukázka testu ověřující získávání klíčových slov z článku, část 1

Pro přidání této metody pro Novinky, stačí pouze připravit testovací data, kterými jsou Jsoup dokument z HTML stránky nějakého článku na Novinkách a Stringový řetězec, který obsahuje klíčová slova z připraveného článku. Pro větší přehlednost testů jsou tyto data v samostatné třídě a v testu se pouze volají.

```
// parse document from HTML file
Document getArticleAsDocument() {
    return
parseFromFile.getDocumentFromFile("html_novinky/article_source/article.html
")
}

// real keywords from article
def getKeywords() {
    return "d6, dálnice, nehoda,, Krimi"
}
```

Zdrojový kód 11.7: Ukázka z přípravy testovacích dat

Do testu tedy stačí pouze přidat jeden řádek, který zajistí, že tato metoda bude testovaná i pro server Novinky.

```
"Novinky"   | new ExtractMetaFromArticleNovinkyPreparedData()
```

Zdrojový kód 11.8: Ukázka tetu ověřující získávání klíčových slov z článku, část 2

Nyní stačí vytvořit kostru metody, aby test proběhl a neskončil výjimkou.

```
String getKeywors(Document document) { return null; }
```

Zdrojový kód 11.9: Kostra metody getKeywords

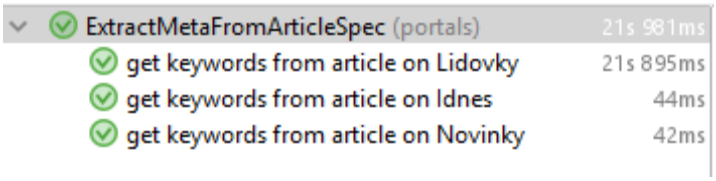
Pokud nyní spustíme test, tak pro všechny servery proběhnou testy úspěšně, pouze pro novinky selžou. Spock zajistí výpis, ze kterého vidíme, že metoda nevrací správná data a test selhal, tak jak se předpokládalo a je to i jedna z fází TDD.

```
Condition not satisfied:

keywordsFromArticle.equals(preparedData.getKeywords())
|                   |     |            |
null                false |            d6, dálnice, nehoda,, Krimi
                          novinky.source.ExtractMetaFromArticleNovinkyPreparedData@1eb6037d
```

Obrázek 11.2: Ukázka selhání testu

Nyní již stačí pouze implementovat tělo metody a znovu spustit test, který pří správné implementaci již projde.




Obrázek 11.3: Ukázka výsledku testu ExtractMetaFromArticleSpec

Takto lze jednoduše pomocí TDD praktiky implementovat velkou část metod nutnou pro nový portál. Výjimkou jsou pouze doplňkové metody specifické pro nějaký server, jako například nutnost použít regulární výraz při parsování jména diskutujícího v komentářích na serveru iDnes. Na ostatních serverech to není nutné implementovat, protože poskytují jméno diskutujícího jako čistý textový řetězec. Naopak někdy není nutné implementovat nějakou metodu, protože není na serveru potřebná.

## 11.5 Continuous integration pomocí Travis CI

Aby aplikace mohla fungovat online je nutné jí hostovat na nějakém prostředí, v tomto případě na Heroku. Po každé úpravě je, ale nutné provést testy, jestli se danou úpravou nenarušila jiná část aplikace. Je tedy nutné zabránit, aby se verze s chybami dostala na produkční prostředí. Testy, ale neprovádí programátor lokálně, ale jsou prováděny automatizovaně pomocí CI serveru, stejně jako deploy upravené aplikace. K tomu je využíván Travis CI, který je propojen s GitHubem. Aby Travis věděl, co má provádět, tak je nutné mít v kořenovém adresáři projektu soubor travis.yml, ve kterém je uvedeno, co se má provádět. Níže je ukázka ze souboru travis.yml, kde je pouze uvedeno, v jakém je projekt jazyce a script pro Maven.

```
language: java
jdk: oraclejdk8
script: "mvn test"
```

Zdrojový kód 11.10: Ukázka ze souboru travis.yml

Po nahrání aktuální verze na GitHub si tedy stáhne projekt, provede sestavení a následně spustí všechny testy. Vzhledem k tomu, že Groovy je založen na jazyce Java, tak ho není nutné explicitně jmenovat. Pokud sestavení nebo testy neprojdou, tak je na to vývojář upozorněn zasláním mailu. Zároveň je možné se z výpisu dozvědět, které testy přesně selhaly a co přesně je tedy nutné opravit.

```
[INFO] Results:
[INFO]
[ERROR] Failures:
[ERROR]   ExtractCommentSpec.first comment for #portalName:50 Condition failed with Exception:

commentList.get(0).getContent() == instance.getFirsComment()
|           |
[]          java.lang.IndexOutOfBoundsException: Index: 0, Size: 0

[ERROR]   ExtractCommentSpec.last comment for #portalName:65 Condition failed with Exception:

commentList.get(commentList.size() - 1).getContent() == instance.getLastComment()
|           |   |           |      |
[]          |   []          0      -1
            java.lang.ArrayIndexOutOfBoundsException: -1

[ERROR]   ExtractCommentSpec.number of comments for #portalName:35 Condition not satisfied:

commentList.size() == instance.getNumberOfComments()
|           |      |  |        |
[]          0      |  |        29
                   |  novinky.source.ExtractCommentNovinkyPreparedData@7b7e4b20
                   false

[ERROR]   NumberOfPagesSpec.number of comment's pages in one article for #portalName:49 Condition not satisfied:

pages == result
|     |  |
0     |  2
      false

[INFO]
[ERROR] Tests run: 48, Failures: 4, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD FAILURE
```

Obrázek 11.4: Travis CI výpis chyb při selhání testů

```
[INFO] Results:
[INFO]
[INFO] Tests run: 45, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 02:23 min
[INFO] Finished at: 2018-05-01T10:08:51Z
[INFO] Final Memory: 39M/765M
[INFO] ------------------------------------------------------------------------
```

*Obrázek 11.5: Travis CI při úspěšném buildu a proběhnutí testů*

Pokud sestavení a testy proběhnou úspěšně, tak je aplikace automaticky nahrána na hosting Heroku, který dokáže provádět automatický deploy z GitHubu a je možné mu nastavit, aby nahrával pouze verze, které úspěšně prošly, přes CI server.

# ZÁVĚR

V rámci diplomové práce byla úspěšně vyvinuta aplikace pro sběr dat ze zpravodajských portálů při využití praktiky TDD spolu s automatizovaným testováním. V rámci této aplikace byly zahrnuty tři zpravodajské servery: iDnes.cz, Lidovky.cz a Novinky.cz, ze kterých je každý den extrahované mezi 10 až 15 tisíci komentáři. Tyto komentáře jsou využívány jako zdroj dat pro jinou diplomovou práci, která má za cíl tyto data vhodně interpretovat.

Vzhledem k podrobnějšímu rozpracování požadavků po zadání práce, bylo vyhodnoceno jako zbytečné, aby v aplikaci bylo webové rozhraní pro vyhledávání podle klíčových slov, vzhledem k tomu, že zpracování těchto dat je cílem jiné práce.

K testování aplikace založené na platformě Java nebyl použit JUnit, jak bývá obvyklé, ale méně běžný testovací framework Spock. Zároveň mu bylo i v teoretické práci věnováno více prostoru. Pro člověka znalého JUnit to může být jakási ukázka, jak lze testy psát jinak.

Teoretická část obsahuje popis některých z nejdůležitějších aspektů manuálního a automatizovaného testovaní, spolu s některými z vybraných nástrojů. Práce tedy může sloužit jako úvod do problematiky testování, spolu s představením některých i méně známých nástrojů.

# POUŽITÁ LITERATURA A ZDROJE


[1] HEROUT, Pavel. *Testování pro programátory*. První vydání. České Budějovice: Kopp, 2016. ISBN 978-80-7232-481-1.

[2] PATTON, Ron. *Testování softwaru*. Vyd. 1. Praha: Computer Press, 2002. Programování. Pro každého uživatele. ISBN 978-80-7226-636-4.

[3] FEWSTER, Mark a Dorothy GRAHAM. *Software Test Automation*. Reading, MA: Addison-Wesley Professional, 1999. ISBN 978-0-201-33140-0.

[4] BECK, Kent. *Programování řízené testy*. 1. vyd. Praha: Grada, 2004. Moderní programování. ISBN 978-80-247-0901-7.

[5] GUNDECHA, Unmesh. *Selenium Testing Tools Cookbook – Second Edition*. 2 edition. B.m.: Packt Publishing, 2015. ISBN 978-1-78439-251-2.

[6] KAPELONIS, Konstantinos. *Java Testing with Spock*. 1 edition. Shelter Island, New York: Manning Publications, 2016. ISBN 978-1-61729-253-8.

[7] HLAVA, Tomáš. *Testování softwaru* [online]. [vid. 2018-04-28]. Dostupné z: http://testovanisoftwaru.cz/

[8] Software Testing Mentor – A comprehensive website for Software Testing Folks! *Software Testing Mentor* [online]. [vid. 2018-04-28]. Dostupné z: http://www.softwaretestingmentor.com/

[9] ČÁPKA, David. Testování v Javě. *ITnetwork* [online]. [vid. 2018-04-28]. Dostupné z: https://www.itnetwork.cz/java-testovani

[10] Learn Automation with Geb and Spock. *Udemy* [online]. [vid. 2018-04-28]. Dostupné z: https://www.udemy.com/learn-automation-with-geb-and-spock/

[11] *Spock Framework Reference Documentation* [online]. [vid. 2018-04-28]. Dostupné z: http://spockframework.org/spock/docs/1.1/index.html

[12] Spock Framework. *GitHub* [online]. [vid. 2018-04-28]. Dostupné z: https://github.com/spockframework

[13] SMATANOVÁ, Klára. *Automatizované testování webových aplikací*. Hradec Králové, 2015. Diplomová práce. Univerzita Hradec Králové, Fakulta informatiky a managementu.

[14] TŘÍSKOVÁ, Lucie. *Testování webových aplikací s využitím nástroje Selenium Webdriver*. Praha, 2013. Diplomová práce. Vysoká škola ekonomická v Praze.

[15] CHMURČIAK, Dávid. *Automatizace regresního testování webové aplikace*. Brno, 2013. Master's thesis. Masarykova univerzita, Fakulta informatiky.

[16] BOROVCOVÁ, Anna. *Testování webových aplikací*. Praha, 2008. Diplomová práce. Univerzita Karlova v Praze.

[17] BOURQUE, Pierre, Richard E. FAIRLEY a IEEE Computer SOCIETY. *Guide to the Software Engineering Body of Knowledge (SWEBOK(R)): Version 3.0*. 3. vyd. B.m.: IEEE Computer Society Press, 2014. ISBN 978-0-7695-5166-1.

[18] DIJKSTRA, Edsger W. The Humble Programmer. *Commun. ACM* [online]. 1972, **15**(10), 859–866. ISSN 0001-0782. Dostupné z: doi:10.1145/355604.361591

[19] What is Software Testing? *Software Testing Mentor* [online]. 26. leden 2013 [vid. 2018-04-28]. Dostupné z: http://www.softwaretestingmentor.com/what-is-software-testing/

[20] *ISTQB Glossary* [online]. [vid. 2018-04-28]. Dostupné z: http://glossary.istqb.org/

[21] LI, Kanglin, Mengqi WU a SYBEX. *Effective Software Test Automation: Developing an Automated Software Testing Tool*. 1 edition. San Francisco, Calif.; London: Sybex, 2004. ISBN 978-0-7821-4320-1.

[22] Software Testing Life Cycle (STLC). *Software Testing Mentor* [online]. 26. leden 2013 [vid. 2018-04-28]. Dostupné z: http://www.softwaretestingmentor.com/software-testing-life-cycle/

[23] *OpenClover 4.2: About Code Coverage* [online]. [vid. 2018-04-28]. Dostupné z: http://openclover.org/doc/manual/4.2.0/general--about-code-coverage.html

[24] *OpenClover – Java, Groovy and AspectJ code coverage tool* [online]. [vid. 2018-04-28]. Dostupné z: http://openclover.org/features

[25] *Specifikace požadavků dle IEEE standardu* [online]. [vid. 2018-04-28]. Dostupné z: http://www.fit.vutbr.cz/study/courses/RPS/public/pro-vytah.html

[26] *Positive Vs Negative testing* [online]. [vid. 2018-04-28]. Dostupné z: https://www.guru99.com/positive-vs-negative-testing.html

[27] KUMAR, Sujit. *SFDC_Best_Practices: SFDC Best Practices* [online]. 2018 [vid. 2018-04-28]. Dostupné z: https://github.com/ghsukumar/SFDC_Best_Practices

[28] MARTIN, Robert C. *Čistý kód: návrhové vzory, refaktorování, testování a další techniky agilního programování*. Vyd. 1. Brno: Computer Press, 2009. ISBN 978-80-251-2285-3.

[29] Testing Pyramids & Ice-Cream Cones. *WatirMelon* [online]. [vid. 2018-04-28]. Dostupné z: https://watirmelon.blog/testing-pyramids/

[30] What is Automation Testing. *Software Testing Mentor* [online]. 26. leden 2013 [vid. 2018-04-28]. Dostupné z: http://www.softwaretestingmentor.com/introduction-to-automation/

[31] Advantages of Automation. *Software Testing Mentor* [online]. 26. leden 2013 [vid. 2018-04-28]. Dostupné z: http://www.softwaretestingmentor.com/advantages-of-automation/

[32] *What is Unit testing?* [online]. [vid. 2018-04-30]. Dostupné z: http://istqbexamcertification.com/what-is-unit-testing/

[33] GÄRTNER, Markus. *ATDD by Example: A Practical Guide to Acceptance Test-Driven Development*. 1 edition. Upper Saddle River, NJ: Addison-Wesley Professional, 2012. ISBN 978-0-321-78415-5.

[34] Proč psát unit test dříve než implementaci. *AspectWorks* [online]. 9. prosinec 2013 [vid. 2018-04-28]. Dostupné z: http://www.aspectworks.com/2013/12/proc-psat-unit-test-drive-nez-implementaci/

[35] *IBM Knowledge Center – Přehled testů řízených daty* [online]. [vid. 2018-04-28]. Dostupné z: https://www.ibm.com/support/knowledgecenter/cs/SSBLQQ_8.5.1/com.ibm.rational.test.ft.doc/topics/dpaboutdatadrivingtscripts.html

[36] TAHCHIEV, Petar, Felipe LEME, Vincent MASSOL a Gary GREGORY. *JUnit in Action, Second Edition*. Second edition. Greenwich: Manning Publications, 2010. ISBN 978-1-935182-02-3.

[37] Testování v Groovy. *AspectWorks* [online]. 18. říjen 2013 [vid. 2018-04-28]. Dostupné z: http://www.aspectworks.com/2013/10/testovani-v-groovy/

[38] *JUnit 5 vs. Spock feature showdown* [online]. [vid. 2018-04-28]. Dostupné z: http://bmuschko.com/blog/junit5-vs-spock-showdown/#mocking

[39] Firefox 55 and Selenium IDE. *Official Selenium Blog* [online]. 9. srpen 2017 [vid. 2018-04-28]. Dostupné z: https://seleniumhq.wordpress.com/2017/08/09/firefox-55-and-selenium-ide/

[40] What is Selenese? *Software Testing Mentor* [online]. 27. leden 2013 [vid. 2018-04-28]. Dostupné z: http://www.softwaretestingmentor.com/what-is-selenese/

[41] RIVERGLIDE. Page Objects Refactored. *RiverGlide Ideas* [online]. 11. únor 2016 [vid. 2018-04-28]. Dostupné z: https://ideas.riverglide.com/page-objects-refactored-12ec3541990

[42] PageObject. *martinfowler.com* [online]. [vid. 2018-04-28]. Dostupné z: https://martinfowler.com/bliki/PageObject.html

[43] Selenium a návrhový vzor Page Objects. *AspectWorks* [online]. 30. červen 2010 [vid. 2018-04-28]. Dostupné z: http://www.aspectworks.com/2010/06/selenium-a-navrhovy-vzor-page-objects/

[44] *Design Patterns and Refactoring* [online]. [vid. 2018-04-28]. Dostupné z: https://sourcemaking.com/

[45] SchedulerPage.java. *Gist* [online]. [vid. 2018-04-28]. Dostupné z: https://gist.github.com/RiverGlide/7718ab70f9d1ee0eddbf8bcf95887555

[46] *ArticleS.UncleBob.PrinciplesOfOod* [online]. [vid. 2018-04-28]. Dostupné z: http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

[47] JONÁŠ, Martin. Návrhové principy: SOLID. *Zdroják* [online]. 9. květen 2012 [vid. 2018-04-28]. Dostupné z: https://www.zdrojak.cz/clanky/navrhove-principy-solid/

[48] *Beyond Page Objects: Next Generation Test Automation with Serenity and the Screenplay Pattern* [online]. [vid. 2018-04-28]. Dostupné z: https://www.infoq.com/articles/Beyond-Page-Objects-Test-Automation-Serenity-Screenplay

[49] Continuous Integration. *martinfowler.com* [online]. [vid. 2018-04-28]. Dostupné z: https://martinfowler.com/articles/continuousIntegration.html

[50] SMITH, Sean. Improving Delivery with Continuous Integration. *Mission Data Journal* [online]. 26. červenec 2016 [vid. 2018-04-30]. Dostupné z: https://journal.missiondata.com/improving-delivery-with-continuous-integration-72a9ffea2117

[51] BERG, Alan Mark. *Jenkins Continuous Integration Cookbook – Second Edition*.. edition. Birmingham, England: Packt Publishing – ebooks Account, 2015. ISBN 978-1-78439-008-2.

[52] Jenkins. *Jenkins* [online]. [vid. 2018-05-01]. Dostupné z: https://jenkins.io/index.html

[53] TeamCity: Hassle-free CI and CD Server by JetBrains. *JetBrains* [online]. [vid. 2018-05-01]. Dostupné z: https://www.jetbrains.com/teamcity/

[54] *Travis CI User Documentation* [online]. [vid. 2018-05-01]. Dostupné z: https://docs.travis-ci.com/

[55] *spring.io* [online]. [vid. 2018-05-01]. Dostupné z: https://spring.io/

[56] *Your relational data. Objectively. - Hibernate ORM* [online]. [vid. 2018-05-01]. Dostupné z: http://hibernate.org/orm/

[57] *jsoup Java HTML Parser, with best of DOM, CSS, and jquery* [online]. [vid. 2018-05-01]. Dostupné z: https://jsoup.org/

[58] *PostgreSQL: The world's most advanced open source database* [online]. [vid. 2018-05-01]. Dostupné z: https://www.postgresql.org/

[59] *What is Heroku | Heroku* [online]. [vid. 2018-04-28]. Dostupné z: https://www.heroku.com/what

[60] IntelliJ IDEA: The Java IDE for Professional Developers by JetBrains. *JetBrains* [online]. [vid. 2018-05-01]. Dostupné z: https://www.jetbrains.com/idea/

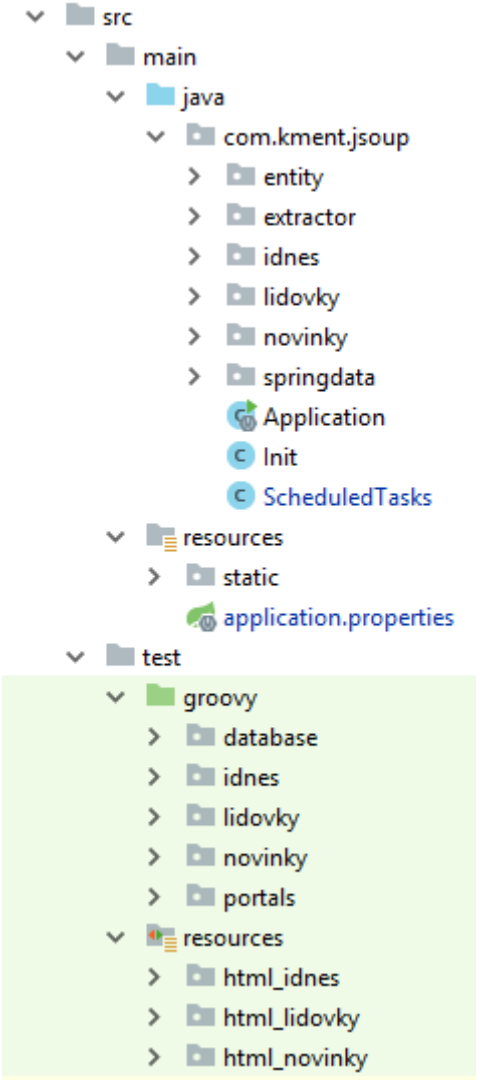
[61] *Maven – Introduction* [online]. [vid. 2018-04-28]. Dostupné z: http://maven.apache.org/what-is-maven.html

[62] *Maven – Welcome to Apache Maven* [online]. [vid. 2018-05-01]. Dostupné z: https://maven.apache.org/

[63] Build software better, together. *GitHub* [online]. [vid. 2018-05-01]. Dostupné z: https://github.com

# PŘÍLOHY

# PŘÍLOHA A – STRUKTURA PROJEKTU



# PŘÍLOHA B – IPORTALEXTRACTOR

```
public interface IPortalExtractor {
    /**
     * Returns name of the portal as a String.
     * Using for saving this name to db.
     *
     * @return portal name
     */
    String getPortalName();

    /**
     * Returns an article list from the news server archive for one spe-
cific day.
     * If the archive does not contain any items, it returns an empty list,
not null.
     *
     * @param url of archive with day in address
     * @return article list from the news server archive for one specific
day
     * @throws IOException
     * @throws ParseException
     */
    List<Article> findArticles(String url) throws IOException, ParseExcep-
tion;

    /**
     * Returns the archive address for yesterday's day
     *
     * @return address of the archive for yesterday's day
     */
    String prepareUrlForYesterday();

    /**
     * Return url for archive in String format for one specific day.
     *
     * @param date for specific day
     * @return address of the archive for specific day
     */
    String prepareUrl(Date date);

    /**
     * Returns a comment list for one specific article.
     *
     * @param urlComment from article
     * @param idArticle  for saving to db
     * @return a list of comments that are found on that address
     * @throws IOException
     * @throws ParseException
     */
    List<Comment> findComments(String urlComment, long idArticle) throws
IOException, ParseException;

    /**
     * Returns creation date of article.
     *
     * @param document created from article parse by Jsoup
     * @return creation date
```

```
     * @throws ParseException
     */
    Date getCreatedDate(Document document) throws ParseException;

    /**
     * Returns keyfords from article.
     *
     * @param document created from article parse by Jsoup
     * @return string with keyword from article
     */
    String getKeywors(Document document);

    /**
     * Returns description for specific article.
     *
     * @param document created from article parse by Jsoup
     * @return string with description
     */
    String getDescription(Document document);

    /**
     * Returns number of comments for one specific article.
     *
     * @param document created from article parse by Jsoup
     * @return number of comment
     */
    int getNumburOfComment(Document document);

    /**
     * Returns the author of the article
     *
     * @param document created from article parse by Jsoup
     * @return string with author
     */
    String getAuthor(Document document);


    /**
     * Returns Jsoup document from specific URL.
     * If document is null, than return null, not empty document.
     *
     * @param urlString
     * @return parsed jsoup document
     * @throws IOException
     */
    Document parse(String urlString) throws IOException;

    /**
     * Returns comments URL from article URL.
     *
     * @param articleUrl
     * @return comment address as a string
     * @throws IOException
     */
    String prepareUrlForCommentPage(String articleUrl) throws IOException;

    /**
     * Returns urtl for specific news portal.
     *
     * @return url of news portal
```

```
     */
    String getUrl();
}
```