Bakalářská práce

České vysoké učení technické v Praze

F3

Fakulta elektrotechnická
Katedra počítačů

# Příprava DevOps prostředí a automatického testování na projektu Hodnocení pracovníků

**Jan Vanke**

Vedoucí: Ing. Jan Zídek
Obor: Softwarové inženýrství a technologie
Květen 2020

# ZADÁNÍ BAKALÁŘSKÉ PRÁCE

## I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Vanke** Jméno: **Jan** Osobní číslo: **478080**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Softwarové inženýrství a technologie**

## II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

**Příprava DevOps prostředí a automatického testování na projektu Hodnocení pracovníků**

Název bakalářské práce anglicky:

**DevOps environment and test automatization for project Employees evaluation**

Pokyny pro vypracování:

1. Proveďte rešerši druhů testování, úrovní testů a procesů v testování.
2. Popište základní principy DevOps, jaké výhody a možná rizika přináší.
3. Dále proveďte rešerši nástrojů používaných v DevOps prostředí pro Java EE aplikace, zvláště pak nástroje využívané pro automatizaci testování. Vymezte pojmy jako jsou continuous integration, delivery a deployment.
4. Navrhněte testovací strategii a procesy správy testů v automatickém testování pro projekt Hodnocení pracovníků.
5. Implementujte testy dle testovací strategie za použití vhodných nástrojů.
6. Navrhněte workflow v DevOps prostředí pro potřeby týmu pracujícím na projektu Hodnocení pracovníků.
7. Implementujte na projektu Hodnocení pracovníků CI/CD pipeline.
8. Zhodnoťte testovací strategii na projektu Hodnocení pracovníků na základě odhalených chyb a jejich závažností.

Seznam doporučené literatury:

[1] ISTQB's Syllabi [online] [cit. 2019-11-03]. Dostupné z: https://www.istqb.org/downloads/syllabi.html
[2] BLACK, Rex; MITCHELL, Jamie L. Advanced software testing. 1st Edition. Santa Barbara, CA: Rocky Nook, 2011. ISBN 978-1-933952-39-0.
[3] PATTON, Ron. Testování softwaru. 1. vydání. Praha: Computer Press, 2002. ISBN 80-722-6636-5.
[4] KNEUPER, Ralf. Sixty Years of Software Development Life Cycle Models. IEEE Annals of the History of Computing [online]. 2017, roč. 39, č. 3,s. 41–54 [cit. 2019-11-10]. ISSN 1058-6180. Dostupné z DOI: 10.1109/MAHC.2017.3481346
[5] HAMBLING, B.; MORGAN, P.; SAMAROO, A.; THOMPSON, G.;WILLIAMS, P. Software testing : An istqb-bcs certified tester foundation guide - 4th edition. 4th edition. BCS Learning & Development Limited,2019. ISBN 9781780174921.

Jméno a pracoviště vedoucí(ho) bakalářské práce:

**Ing. Jan Zídek, Centrum znalostního managementu**

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **14.02.2020** Termín odevzdání bakalářské práce: ______________

Platnost zadání bakalářské práce: **30.09.2021**

| Ing. Jan Zídek<br>podpis vedoucí(ho) práce | podpis vedoucí(ho) ústavu/katedry | prof. Mgr. Petr Páta, Ph.D.<br>podpis děkana(ky) |
|---|---|---|

 © ČVUT v Praze, Design: ČVUT v Praze, VIC

## III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

.

Datum převzetí zadání

Podpis studenta


© ČVUT v Praze, Design: ČVUT v Praze, VIC

# Poděkování

Rád bych zde především poděkoval vedoucímu práce, Ing. Janu Zídkovi, za jeho ochotu, pomoc a cenné rady při psaní této práce.

Dále bych chtěl poděkoval Ing. Lukáši Zoubkovi za pomoc a konzultace k této práci. Velké díky také patří Centru Znalostního Managementu za umožnění vzniku této práce a za podporu, které se mi od jejích zaměstnanců a stážistů dostalo.

V neposlední řadě chci poděkovat své rodině, přítelkyni a její rodině za trpělivost a obrovskou podporu, nejen při psaní této práce.

# Prohlášení

Prohlašuji, že jsem bakalářskou práci vypracoval samostatně a v souladu s Metodickým pokynem o dodržování etických principů pro vypracování závěrečných prací, a že jsem uvedl všechny použité informační zdroje.

V Praze, 22. května 2020

Jan Vanke

# Abstrakt

Tato bakalářská práce se zabývá konceptem DevOps a testováním softwarových projektů v agilním prostředí, konkrétně aplikací těchto principů na softwarový projekt Hodnocení pracovníků.

Hlavní část práce sestává z provedení rešerše týkající se testování software a principů DevOps, vytvoření obecné testovací strategie a konkrétního testovacího plánu pro projekt Hodnocení pracovníků a následné implementace testů dle tohoto plánu.

Dle rešerše nástrojů pro projekt v jazyce Java EE jsou pro implementaci testů a prostředí vybrány vhodné nástroje, jako je JUnit5 pro jednotkové testování a Arquillian pro integrační testování.

Pro správné začlenění testování do životního cyklu software je v rámci práce vytvořen návrh změny workflow, který využívá moderních principů, jako jsou Continuous Integration a Delivery. Za pomocí nástroje Gitlab CI/CD je pro toto nové workflow vytvořena Continuous Delivery Pipeline.

Nově navržené workflow je demonstrováno na typických příkladech, se kterými se lze v rámci procesu dodávání software setkat, jako je například oprava závažné produkční chyby nebo vydání nové verze software. Provedené testování je poté vyhodnoceno na základě množství a závažnosti odhalených chyb.



**Klíčová slova:** Java EE, Automatizované testování, DevOps, testování, testovací strategie, Continuous Integration, Continuous Delivery, agilní testování



**Vedoucí:** Ing. Jan Zídek



# Abstract

This bachelor thesis deals with the DevOps concept and software project testing in an agile environment. More specifically, the application of those principles to software project - Employee Evaluation.

The main focus of this work is to briefly introduce software testing and DevOps principles and to demonstrate the design of the testing strategy for project Employee Evaluation. The common testing strategy design is followed by a specific test plan design. Tests are then implemented according to the specific test plan, including the environment needed for automatization of the tests. Tool comparison is also included in the theoretical part, and appropriate tools are then selected for the implementation based on this comparison. The two main tools used for testing are JUnit5 test framework for unit tests and Arquillian for integration tests.

New workflow processes are designed to integrate those modern principles, such as Continuous Integration and Delivery, to the software development lifecycle. With the use of Gitlab CI/CD tool, a Continuous Delivery Pipeline is created for the new workflow.

The new workflow is demonstrated in the form of several examples common within software delivery processes, such as critical production bug repair or release of a new version. Based on the detected defects and their severity, evaluation for the finished testing is conducted.



**Keywords:** Java EE, Automated testing, DevOps, testing, testing strategy, Continuous Integration, Continuous Delivery, agile testing


**Title translation:** DevOps environment and test automatization for project Employees evaluation

# Obsah


**1 Úvod** 1
1.1 Předmluva 1
1.2 Motivace a cíl 1
1.3 Struktura práce 1
1.3.1 Teoretická část 1
1.3.2 Praktická část 2
1.3.3 Závěr a přílohy 2

**Část I**
**Teoretická část**

**2 Testování software** 4
2.1 Základní pojmy 4
2.1.1 Životní cyklus software 4
2.1.2 Verification a validation 5
2.1.3 Test case a test condition 5
2.1.4 Happy a sad path 5
2.1.5 User stories a epics 5
2.1.6 Entry a Exit Criteria 6
2.2 Cíle testování 6
2.3 Současné modely životního cyklu software 7
2.4 Testování v agilních modelech životního cyklu software 7
2.4.1 Role a odpovědnost testerů 8
2.4.2 Testeři a struktura týmu 9
2.4.3 Výstupy a dokumentace 9
2.5 Shrnutí kapitoly 10

**3 Druhy testování software** 11
3.1 Definice pojmu druh testování software 11
3.2 Functional a non-functional testing 11
3.3 White-box a black-box testing 12
3.3.1 Black-box techniky 12
3.3.2 White-box techniky 13
3.4 Statické a dynamické testování 14
3.5 Testování spojené se změnami -confirmation a regression testing 14
3.6 Exploratory testing 14
3.7 Shrnutí kapitoly 15

**4 Úrovně testování software** 16
4.1 Definice pojmu úroveň testování 16
4.2 Testování komponent 17
4.3 Integrační testování 17
4.4 Systémové testování 18
4.5 Akceptační testování 18
4.5.1 Uživatelské akceptační testování - UAT 18
4.5.2 Operační akceptační testování - OAT 18
4.5.3 Regulační testování a testování podle smluv 19
4.5.4 Alfa a beta testování 19
4.6 Testovací kvadranty 19
4.7 Shrnutí kapitoly 20

**5 Procesy v testování software** 21
5.1 Definice procesu v testování software 21
5.2 Plánování monitorování a kontrola testů 22
5.2.1 Testovací strategie a testovací plán 22
5.2.2 Sledovatelnost a matice sledovatelnosti 22
5.2.3 Monitorování, kontrola testů a metriky 22
5.3 Analýza a design testů 23
5.3.1 Analýza 23
5.3.2 Design 24
5.4 Implementace a spouštění testů 24
5.5 Vyhodnocení a reporting 24
5.6 Shrnutí kapitoly 25

**6 Základní principy DevOps** 26
6.1 Definice pojmů 26
6.1.1 DevOps 26
6.1.2 Key performance indicators 26
6.1.3 Business drivers 27
6.1.4 Version control system 27
6.2 Původní myšlenka 27
6.3 Continuous Delivery Pipeline 28
6.4 Continuous Integration 28
6.4.1 Prerekvizity a postupy 28
6.5 Continuous Delivery 29
6.6 Návrh a struktura delivery pipeline 29
6.7 Automatické testování 31
6.7.1 Testy směřující k byznysu, které podporují vývojový tým 32
6.7.2 Testy směřující k technologiím, které podporují vývojový tým 32

6.7.3 Testy směřující k byznysu, které kritizují produkt . . . 32
6.7.4 Testy směřující k technologiím, které kritizují produkt . . . 32
6.8 Continuous Monitoring a další podpůrné principy . . . 33
6.9 Shrnutí kapitoly . . . 33

**7 Adopce DevOps principů 34**
7.1 Stávající stav a cíle . . . 34
7.1.1 Mapování stávajícího stavu . . . 34
7.1.2 Nastavení cílů a business drivers . . . 36
7.2 Implementace DevOps a Continuous Delivery Pipeline . . . 36
7.2.1 Časté výzvy při adopci DevOps . . . 37
7.3 Shrnutí kapitoly . . . 37

**8 Rešerše nástrojů 38**
8.1 Nástroje pro testování . . . 38
8.1.1 Testování komponent . . . 38
8.1.2 Integrační testování . . . 40
8.1.3 Akceptační testování . . . 41
8.2 Nástroje pro DevOps . . . 41
8.2.1 Version Control System . . . 41
8.2.2 Build . . . 42
8.2.3 CI/CD . . . 43
8.3 Shrnutí kapitoly . . . 45

**Část II**
**Praktická část**

**9 Testovací strategie a testovací plán 47**
9.1 Popis testované aplikace Hodnocení pracovníků . . . 47
9.2 Návrh testovací strategie . . . 48
9.2.1 Technologie a infrastruktura . . . 48
9.2.2 Prostředí a další omezení . . . 48
9.3 Testovací strategie . . . 49
9.3.1 Prostředí . . . 49
9.3.2 Postupy testování . . . 49
9.3.3 Nástroje, automatizace a úrovně testů . . . 52
9.3.4 Nahlašování chyb . . . 54
9.3.5 Struktura testovacího plánu . . . 55
9.3.6 Výsledky testování . . . 57
9.4 Testovací plán pro projekt Hodnocení pracovníků . . . 57
9.4.1 Testovací plán pro projekt Hodnocení pracovníků . . . 57
9.4.2 Postup při návrhu testovacího plánu pro projekt Hodnocení pracovníků . . . 59
9.5 Shrnutí kapitoly . . . 60

**10 Implementace testů a delivery pipeline 61**
10.1 Příprava prostředí . . . 61
10.1.1 Gitlab CI/CD . . . 61
10.1.2 Jednotkové a integrační testy 65
10.1.3 Explorační testování . . . 67
10.1.4 Reporting chyb . . . 68
10.2 Návrh testů . . . 68
10.2.1 Jednotkové testy . . . 68
10.2.2 Integrační testy . . . 69
10.2.3 Testovací data . . . 70
10.2.4 Explorační testování . . . 70
10.3 Implementace testů . . . 70
10.3.1 Jednotkové testy . . . 71
10.3.2 Integrační testy persistentní vrsty . . . 73
10.3.3 Integrační testy přes více vrstev aplikace . . . 74
10.3.4 Explorační testování . . . 75
10.4 Shrnutí kapitoly . . . 75

**11 Vyhodnocení testování 76**
11.1 Výsledky testování pro projekt Hodnocení pracovníků . . . 76
11.1.1 Vyhodnocení na základě odhalených chyb . . . 76
11.1.2 Sledované metriky . . . 76
11.1.3 Shrnutí . . . 77

**12 Návrh workflow 79**
12.1 Stávající workflow na projektu Hodnocení pracovníků . . . 79
12.1.1 VSM . . . 79
12.1.2 Vyhodnocení neefektivit a plýtvání . . . 80
12.1.3 Nastavení metrik . . . 81
12.2 Nové workflow na projektu Hodnocení pracovníků . . . 81
12.2.1 Prerekvizity . . . 81

12.2.2 Obecný popis . . . . . . . . . . . . . 82
12.2.3 Konkrétní příklady . . . . . . . . 84
12.3 Shrnutí kapitoly . . . . . . . . . . . . . 85

**13 Závěr 86**

**Přílohy**

**A Literatura a zdroje 88**

**B Seznam zkratek 92**

**C Testovací matice 93**

**D Příprava prostředí 97**

**E Test charters 99**

**F Analýza stavů a přechodů 104**

**G Odhalené chyby při testování 105**

# Obrázky


2.1 Ukázka user story ve formě kartičky, zdroj: [3] . . . . . . . . . . . . . . . 6
2.2 Agilní model SDLC, zdroj: [3] . . . 8

3.1 Ukázka testované části aplikace, zdroj: [11] . . . . . . . . . . . . . . . . . . . . 13

4.1 Testovací pyramida podle ISTQB - úrovně testování, zdroj: Autor . . . 16
4.2 V model - úrovně testování odpovídající aktivitám SDLC, zdroj: Autor . . . . . . . . . . . . . . . . . . . . . . . . 17
4.3 Testovací kvadranty, zdroj: [13] 20

5.1 Procesy testování v tradičním modelu SDLC, zdroj: [4] . . . . . . . . . 21

6.1 Obecná delivery pipeline, zdroj: [20] . . . . . . . . . . . . . . . . . . . . . . . . . . 30
6.2 Delivery pipeline jako diagram aktivit, zdroj: [23] . . . . . . . . . . . . . . 31

7.1 Příklad modelu Value Stream Mapping, zdroj: [18] . . . . . . . . . . . . 35
7.2 Konkrétní příklad VSM s hodnotami, zdroj:[26] . . . . . . . . . . . 35

9.1 Proces zavádění testování na projektu, zdroj: Autor . . . . . . . . . . 50

10.1 Ukázka rozhraní Gitlab CI/CD, zdroj: Autor . . . . . . . . . . . . . . . . . . . 63

12.1 VSM pro proces - jednoduchý změnový požadavek, zdroj: Autor 80
12.2 VSM pro proces - nová funkcionalita, zdroj: Autor . . . . . . 80

C.1 Tabulka testovací matice, zdroj: Autor . . . . . . . . . . . . . . . . . . . . . . . . 93
C.2 Tabulka testovací matice, zdroj: Autor . . . . . . . . . . . . . . . . . . . . . . . . 94
C.3 Tabulka testovací matice, zdroj: Autor . . . . . . . . . . . . . . . . . . . . . . . . 95
C.4 Tabulka testovací matice, zdroj: Autor . . . . . . . . . . . . . . . . . . . . . . . . 96

F.1 Analýza přechodu stavů pro evaluace, zdroj: Autor . . . . . . . . . 104


# Tabulky


11.1 Počet chyb podle priority, zdroj: Autor . . . . . . . . . . . . . . . . . . . . . . . . 77

E.1 Test charters, zdroj: Autor . . . 103

G.1 Objevené chyby - export z Gitlab Issues, zdroj: Autor . . . . . . . . . . . . 106

# Kapitola 1
# Úvod

V následující kapitole představuji motivaci, hlavní cíle a strukturu své práce.

## 1.1 Předmluva

Tradiční role testování softwarových projektů se s postupných rozšiřováním agilních principů mění. V rámci této práce se zabývám právě těmito moderními přístupy k vývoji software a tomu, jaké místo v nich má samotné testování. Využívám pro to principů, jako jsou Continuous Integration a Delivery. Popisuji také moderní metody testování, jako je explorační testování nebo model testovacích kvadrantů, které umožňují integrovat testování do stávajícího životního cyklu software. Tyto principy a postupy demonstruji na projektu Hodnocení pracovníků, vytvořením obecné testovací strategie, nového workflow a samotnou implementací pomocí moderních nástrojů pro testování a DevOps.

## 1.2 Motivace a cíl

Vzhledem k zvyšujícím se nárokům na kvalitu software a na rychlost vývoje došlo v posledních letech k rozšíření moderních principů a nástrojů využívaný v životním cyklu software. Hlavním cílem této práce tedy je aplikovat tyto principy na projekt Hodnocení pracovníků a vytvořit tak pro tým, který na projektu pracuje, prostředí a pracovní postupy, které jim umožní dodávat výstupy ve vyšší kvalitě a v kratších intervalech.

## 1.3 Struktura práce

Práce je rozdělena na praktickou, teoretickou část a závěr obsahující shrnutí a přílohy. V této sekci popisuji, co bude obsahem jednotlivých částí práce.

### 1.3.1 Teoretická část

V rámci teoretické části stručně uvedu čtenáře do problematiky testování software, současných metodik vývoje a toho, jak se do jednotlivých druhů

modelů životního cyklu začleňuje testování.

Dále se v teoretické části věnuji druhům testování, úrovním testování a souvisejícím procesům v testování software. Zaměřuji se na to, jaké místo má testování v agilím prostředí a jakým způsobem se využívá automatizace testování.

Další z kapitol je věnována konceptu nazvanému DevOps, vymezuji v ní pojmy, jako jsou Continuous Integration a Continuous Delivery a tomu jak souvisí s automatickým testováním.

Závěr teoretické části sestává z rešerše nástrojů určených pro testování, automatizaci testování a podporu DevOps principů. Rešerše je zaměřena na projekt v programovacím jazyku Java EE.

### 1.3.2 Praktická část

V praktické části nejprve vytvářím návrh testovací strategie a konkrétní testovací plán pro projekt Hodnocení pracovníků.

Dále podle vypracovaného testovacího plánu navrhuji a implementuji konkrétní testy pro projekt Hodnocení pracovníků. Zároveň připravuji prostředí pro automatizaci testů a další postupy vycházející z principů DevOps.

Následně vyhodnocuji provedené testování na projektu Hodnocení pracovníků za pomocí metrik stanovených v testovacím plánu.

V závěrečné kapitole praktické části se věnuji mapování stávajícího stavu workflow na projektu Hodnocení pracovníků. Podle analýzy stávajícího workflow navrhuji nové, které umožní týmu pracovat efektivněji a zároveň integrovat testování do procesu dodávání software.

### 1.3.3 Závěr a přílohy

Závěrečná část práce obsahuje shrnutí výsledků práce a vyhodnocení toho, zdali byly naplněny stanovené cíle. Mimo jiné také obsahuje přílohy k praktické části práce.

# Část I

## Teoretická část

# Kapitola 2

# Testování software

V následující části práce vymezuji základní pojmy z oblasti testování software a stručně uvádím čtenáře do dané problematiky. Zaměřuji se také na hlavní rozdíly v současných metodikách vývoje software a jakým způsobem se do nich testování začleňuje.

Testování software je vzhledem k rozsahu a složitosti dnešních softwarových systémů nedílnou součástí jejich životního cyklu. Jedná se o skupinu procesů a technik, které využíváme k zaručení co nejvyšší možné kvality software a ke snížení rizika vzniku chyby v produkci. V oboru vývoje software a jeho testování se vyskytuje několik specifických pojmů, které uvádím a popisuji v následující sekci 2.1.

## 2.1 Základní pojmy

V této sekci vymezuji základní pojmy, které souvisí s vývojem a testování software.

### 2.1.1 Životní cyklus software

Pojem životní cyklus software je časové období, které začíná prvotním záměrem a končí až když není dostupný k používání. V anglické literatuře se pro pojem životní cyklus software využívá zkratka SDLC - Software Development Life Cycle. Pojem by mohl být chybně zaměňován se specifičtějším - životní cyklus vývoje software, který značí časové období od prvotního záměru až do uvedení produktu do produkce.[1][2]

Pro oba je však typické, že sestávají z více fází, které se mohou vzájemně překrývat, nebo se iterativně opakovat. Záleží na použitém modelu životního cyklu. Fáze vyskytující se ve většině modelů SDLC jsou následující:[2]

- Sběr požadavků a analýza
- Design
- Implementace
- Testování
- Údržba

### 2.1.2 Verification a validation

Pojmy verification a validation se dají do češtiny přeložit jako jedno slovo - ověřování či kontrola. Právě proto může často docházet k jejich zaměňování nebo případné nejednoznačnosti. Verifikací se označuje ověřování testovaného subjektu vůči požadavkům nebo zadané specifikaci. Validace oproti tomu ověřuje, jestli testovaný subjekt splňuje potřeby uživatelů a stakeholderů. Provedením validace nad testovaným subjektem mimo jiné můžeme zjistit zda specifikace odpovídá potřebám uživatelů a stakeholderů. Validace tedy pomáhá odhalit chyby ve specifikaci.[1]

### 2.1.3 Test case a test condition

Test case je množina vstupních hodnot, očekávaných výsledků a podmínek spuštění, vyvinutá na základě test condition. Test case tedy testuje jeden cíl, například určitou cestu programem nebo splnění konkrétního požadavku. Test condition oproti tomu popisuje jednu konkrétní vlastnost testovaného systému. Ta může být verifikována jedním nebo více test cases, jako je například jedna funkce, transakce nebo nějaký atribut kvality.[2]

### 2.1.4 Happy a sad path

V případě happy path se jedná o průchod nějakým testovacím scénářém (test case) s pozitivním výsledkem. Například pro test case - úspěšné vytvoření objednávky - by se jednalo o průchod, kdy dojde k vytvoření objednávky a odeslání potvrzení.

Oproti tomu sad path, někdy také označována jako edge case nebo unhappy path, značí průchod scénářem s negativním výsledkem. Příkladem může být přihlašování uživatele s nesprávnou kombinací přihlašovacích údajů.

### 2.1.5 User stories a epics

Pojmem user story se v agilním prostředí označuje neformálním jazykem popsaný požadavek na funkcionalitu nebo vlastnost systému. User story může mít například formu kartičky, jak můžete vidět na obrázku 2.1.

Priority: Size:

**User story:**

As a: [role]

I want to: [what]

So that: [why]

**Acceptance criteria:**

I know when I'm done when:

**Obrázek 2.1:** Ukázka user story ve formě kartičky, zdroj: [3]

Jako epic se označuje rozsáhlejší user story, která je většinou moc velká na to, aby ji bylo možné implementovat v jedné iteraci.

V některých implementacích agilních modelů, zvláště pak u komplexních projektů, dochází k formalizaci těchto výstupů do formálních požadavků.[4]

### 2.1.6 Entry a Exit Criteria

Entry a exit criteria značí, kdy daná testovací aktivita může začít a kdy ji můžeme považovat za hotovou. Typickým příkladem entry criteria může být dostupnost testovatelných požadavků, user stories nebo dostupnost testovacích nástrojů, dat nebo testovacího prostředí. Jako příklad exit criteria je možné definovat úrovně pokrytí testy, přijatelný odhadovaný počet neodhalených chyb, vyhodnocené úrovně určitých parametrů kvality - například dostupnost, zabezpečení.[4]

## 2.2 Cíle testování

Hlavní cíle testování software jsou podle ISTQB Foundation Syllabus následující:[4]

- Zhodnotit požadavky, user stories, design a kvalitu kódu.
- Verifikovat zda byly všechny požadavky naplněny.
- Validovat, jestli testovaný subjekt funguje způsobem, jakým uživatelé a stakeholdeři očekávají.
- Zajistit určitou úroveň kvality testovaného subjektu.
- Zabránit vznikům vad.

- Nalézt vady a chyby.
- Poskytnout dostatečné informace o kvalitě subjektu stakeholderům, aby na jejich základě mohli rozhodovat, zvláště o kvalitě testovaného subjektu.
- Snížit riziko nedostatečné kvality testovaného subjektu (např. vznik dříve neodhalených chyb v produkci).

## 2.3 Současné modely životního cyklu software

V následující sekci uvádím základní charakteristiku a rozdělení modelů životního cyklu software, které by mělo postačit jako stručný teoretický úvod do další sekce 2.4, ve které uvádím, čím se vyznačuje testování v agilních metodikách.

SDLC modely se dají v základu rozdělit na tři typy a to na tradiční, iterativní, často také označovaný jako agilní, a kombinaci předchozích dvou.

Nejvyužívanějšími zástupci tradičních modelů jsou Waterfall, V a W model. Je pro ně typické, že jednotlivé fáze následují sekvenčně za sebou. Moderními zástupci iterativního modelu jsou například Scrum, Dynamic Systems Development Method nebo Feature Driven Development. Typické pro agilní metodiky je opakování fází v krátkých iteracích. Velký důraz je při nich také kladen na komunikaci, rychlou reakci na změny a funkční software i za cenu méně obsáhlé dokumentace.[5][6][7]

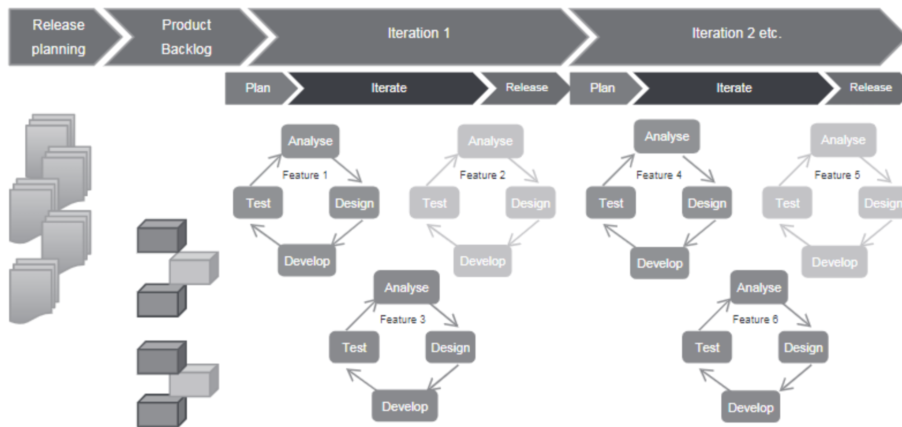
Zbývající modely jsou převážně kombinací iterativních a tradičních modelů. Je to například spirálový model nebo se jedná o modely, které nelze přesně začlenit. Příkladem může být takzvaný Big Bang Model jež se nedrží žádného přesného postupu, a je proto aplikovatelný jen na velmi malé systémy.

V období od 70. let minulého století po současnost prošly modely životního cyklu software vývojem, který se postupně snaží brát si to nejlepší z obou druhů modelů. Soustředí se tedy na takzvaný „scaling agile". U opravdu komplexních systémů, jako jsou například systémy pro letectví nebo zdravotnictví, však stále převažují tradiční - waterfall metody.[4][6]

## 2.4 Testování v agilních modelech životního cyklu software

V této sekci stručně popisuji, jakým způsobem probíhá testování software v agilních modelech SDLC, a jak se liší od testování v tradičních modelech. Vzhledem k prostředí, pro které práce vzniká, a kde se využívá právě agilní metodiky, jsem se proto zaměřil na testování v agilním prostředí.

Jak již bylo zmíněno v předešlé sekci 2.3, je v rámci agilních modelů software dodáván průběžně ve velmi krátkých iteracích, během niž je stakeholderovi dodána část funkcionality. Na začátku každé iterace probíhá plánování, kde se definuje její rozsah. Následně je dle rozsahu funkcionalita implementována. Na obrázku 2.2 můžete vidět obecný agilní model SDLC.




**Obrázek 2.2:** Agilní model SDLC, zdroj: [3]

Testování v agilním prostředí by se mělo aplikovat již v průběhu iterace, nikoliv jen jako konečná fáze, jako je tomu u některých tradičních metodik. Na testování se nepodílí jen tester, ale často se využívá i součinnost vývojářů a stakeholderů. Pomoc s testováním může mít různou formu, u vývojářů je to nejčastěji tvorba testů komponent. Testy komponent jsou blíže popsány v sekci 4.2. U stakeholderů se pak jedná především o experimentování s novou funkcionalitou a podávání zpětné vazby týmu.[4]

Hlavními rozdíly oproti tradičním modelům je zapojování testerů v průběhu celého SDLC, vzájemné vzdělávání v rámci týmů, spolupráce na testování a párování. Zásadní je také schopnost reagovat na změny.[3]

### 2.4.1 Role a odpovědnost testerů

Zapojení testerů by mělo začít již v plánovací fázi iterace. Testeři se mohou například podílet na tvorbě detailní analýzy rizik pro konkrétní user stories, rozhodovat jestli je daná user story správně formulována a testovatelná. Dále musí určit kolik času a jaké zdroje budou potřeba pro testování v dané fázi. Neméně důležitým úkolem je pomoc a zapojování se do tvorby a správy automatických testů na různých úrovních testování, zvláště pak na úrovni integračních, systémových a systémově integračních testů. Jednotlivé úrovně testů jsou popsány blíže v kapitole 4. Díky rozsáhlé automatizaci testů je možné věnovat na projektech více času manuálnímu testování, například pomocí experience-based a defect-based technik, které vzhledem k rozsahu své práce dále nepopisuji.[3][4]

Během plánování iterace je také možné rozhodnout o provedení takzvaného nefunkčního testování, které je detailněji pospané v sekci 3.2. Jedná se o testování výkonu, zabezpečení, spolehlivosti nebo použitelnosti.[3]

Jedním ze základních principů agilních přístupů je, že v rámci projektu mohou nastat změny. Způsob jakým řídit rizika spojená se změnami v průběhu projektu jsou automatické testy. Je důležité, aby změny nepřesáhly schopnosti týmu řídit rizika spojená s těmito změnami.[4]

Dalším častým rizikem je, že testeři ztratí objektivní pohled na testování, začnou ignorovat nedostatky a tolerovat nízkou kvalitu výstupů týmu. Jedním z důvodů pro tento nežádoucí stav může být velký tlak na rychlý vývoj, nebo to že testeři moc dlouho spolupracují s vývojáři a dochází tak k jejich ovlivňování a ztrátě objektivního pohledu.[3][4]

### 2.4.2 Testeři a struktura týmu

Obecně lze rozlišit tři způsoby, jakými se můžou testeři začleňovat do týmu podle úrovně nezávislosti na vývojovém týmu. Každá z možností je vhodná pro jiný druh vytvářených projektů a každá s sebou nese určitá rizika, které uvádím v následující části.

#### Testeři součástí vývojového týmu

Hlavní výhodou tohoto přístupu je, že testeři mají dobrou znalost projektu a mohou poskytovat rychlejší zpětnou vazbu vývojovému týmu. Na druhou stranu je možné, že dlouhodobější spoluprácí s vývojovým týmem ztratí tester nezávislost a může docházet k ignorování některých nedostatků vývoje a celkově projektu.[3][4][8]

#### Oddělený testerský tým - krátkodobě přiřazovaní testeři

V rámci tohoto přístupu je tester přiřazován na projekty ze separátního testerského týmu dle potřeby a jeho schopností. Hlavní výhodou je nezávislost testerů. Ti mohou poté hodnotit kvalitu software objektivně, bez ovlivnění členy vývojového týmu. Nevýhodou může být špatná znalost změn a nových vlastností projektu nebo problémy v komunikaci s vývojovým týmem a případný tlak z hlediska času. Vývojovým týmem pak mohou být testeři vnímáni jen jako zpomalující faktor.[3][4][8]

#### Oddělený testerský tým - dlouhodobě přiřazovaní testeři

Z hlediska struktury funguje jako předešlý přístup, jen jsou testeři přiřazováni na projekty dlouhodobě na jeho začátku. Zachovávají si přitom nezávislost na vývojovém týmu a jedná se o vyčleněný celek. Kombinuje výhody předešlých přístupů, tedy testeři si udržují nezávislost na vývojovém týmu, ale zároveň mají dobrou znalost projektu. Je také možné mít v rámci testerského týmu specialisty přes různé oblasti a dosazovat je na projekty dle potřeby. Tento přístup není příliš vhodný pro menší společnosti, zvláště pak z hlediska lidských zdrojů.[3][4][8]

### 2.4.3 Výstupy a dokumentace

Jednou ze základních myšlenek agilních metodik je snaha omezit dokumentaci a místo toho se zaměřit na funkční software a automatické testy, které verifikují požadavky na systém. V úspěšném agilním projektu je nutné nalézt

balanc mezi vyšší efektivitou a dostatečnou úrovní dokumentace pro klienta, testování, vývoj a údržbu projektu. V rámci fáze plánování nového releasu musí tým rozhodnout o vhodné úrovni dokumentace výstupů.[3][4]

Výstupy projektu, v anglické literatuře označované jako work products, se dají rozdělit do tří skupin:[4]

- business-oriented work products
- development work products
- test work products

První z nich - business-oriented work products - označuje výstupy, které popisují co je od systému požadováno. Například specifikaci nebo požadavky a jak se má systém používat - uživatelská dokumentace.[3][4][8]

Development work products jsou výstupy, které zahrnují převážně kód, ale také to jak systém funguje, což může být například popis architektury nebo databázová schémata.[3][4][8]

Poslední skupina - test work products - zahrnuje samotné implementované testy a automatické testy, ale i dokumenty popisující jak je systém otestován - testovací strategie, plány a výstupy z testování, tedy výsledky testování, například ve formě test dashboard.[3][4][8]

## 2.5 Shrnutí kapitoly

V této kapitole jsem definoval pojem testování software a uvedl jsem jeho hlavní cíle. Dále jsem vymezil základní pojmy z daného oboru. Zbytek kapitoly jsem věnoval tomu, jak se testování software začleňuje do jeho životního cyklu, zvláště pak do agilních modelů. Také jsem uvedl, jaké jsou typické výstupy softwarových projektů a jak souvisí s testováním. Mimo jiné jsem se také zabýval tím, jak se mohou začleňovat testeři do struktury týmů.

# Kapitola 3

# Druhy testování software

V následující kapitole uvádím, co jsou to druhy testování software a jaké hlavní skupiny známe. Dále uvádím jejich hlavní charakteristiky, příklady a některé konkrétní pokročilejší testovací techniky.

## 3.1 Definice pojmu druh testování software

Skupiny testovacích aktivit, které mají společný cíl otestovat některou z částí systému nebo některou z jeho vlastností, se označují jako druhy testování software. Dělí se jak podle znalosti vnitřní struktury testovaného systému, tak například podle druhu požadavků vůči kterým daný subjekt testuji.[4]

## 3.2 Functional a non-functional testing

Jako funkční testování, anglicky functional testing, se označuje testování konkrétní funkcionality testovaného subjektu - funkční požadavky. Funkční testování má za cíl ověřit úplnost, korektnost a správnost systému, v angličtině system completeness, correctness, and appropriateness.[4][9]

Funkční požadavky určují, co konkrétně by měl systém vykonávat. Mohou být specifikované ve formě business požadavků, user stories, use cases nebo funkční specifikace.[4]

Nefunkční testování, anglicky non-functional testing, značí testování vůči nefunkčním požadavkům, tedy jak dobře se systém chová. Například standart ISO/IEC 25010[1] definuje takzvaný quality model a specifikuje jednotlivé druhy charakteristik kvality. Testování nefunkčních požadavků by se mělo provádět na všech úrovních testování. Pokud dojde k pozdnímu odhalení chyb v nefunkčních požadavcích, může to způsobit v pozdějších fázích projektu značné problémy. Jednotlivé testovací úrovně jsou popsány v kapitole 4.[4]

[1] standart ISO/IEC 25010: `https://iso25000.com/index.php/en/iso-25000-standards/iso-25010`

## 3.3 White-box a black-box testing

„Jedná se o dělení podle znalosti vnitřní struktury testovaného software, tedy jeho zdrojového kódu, toků dat a architektury." (překlad autora)[9]

V případě white-box testingu má tester přístup k vnitřní struktuře software, které testuje. Typicky se white-box testing provádí na úrovni komponent a integračních testů. Testování na úrovni komponent je popsáno v sekci 4.2 a úroveň integračních testů v sekci 4.3.[1][9]

White-box testing je někdy také označován jako Clear Box testing nebo Glass Box testing. Příkladem white-box testingu je například funkční testování založené na případech užití - use cases.

Black-box testing naopak značí, že není známa vnitřní struktura testovaného software a je nutné test-cases vytvářet na základě specifikace a dokumentace obsahující požadavky. Příkladem black-box testingu je například statická analýza kódu testovaného subjektu.[1]

Někdy se také lze setkat s pojmem grey-box testing, který je kombinací dvou výše uvedených.

### 3.3.1 Black-box techniky

Pro black-box testing jsou specifické některé pokročilejší testovací techniky, které jsou uvedeny v této podsekci. Jak již bylo zmíňěno tyto techniky jsou většinou založeny na specifikaci, bez znalosti vnitřní struktury testovaného subjektu.[10]

#### Analýza tříd ekvivalence

Tato analýza spočívá v určení stejných skupin, nebo-li tříd vstupů nebo výstupů, pro které je podle specifikace možné očekávat stejný výsledek nějaké operace v systému.

#### Analýza hraničních hodnot

Používá se často v kombinaci s analýzou tříd ekvivalence. Hraniční hodnoty se volí těsně pod a nad hranicí dané třídy ekvivalence. Tester poté ověřuje, zda-li se systém chová pro hraniční hodnoty podle specifikace.

#### Analýza přechodů stavů

Tuto analýzu je vhodné použít pro část systému, která má více různých vzájemně propojených stavů. Pomocí jednoduché tabulky lze určit jednotlivé akce v systému, a jak ovlivňují přechody mezi stavy testované části.

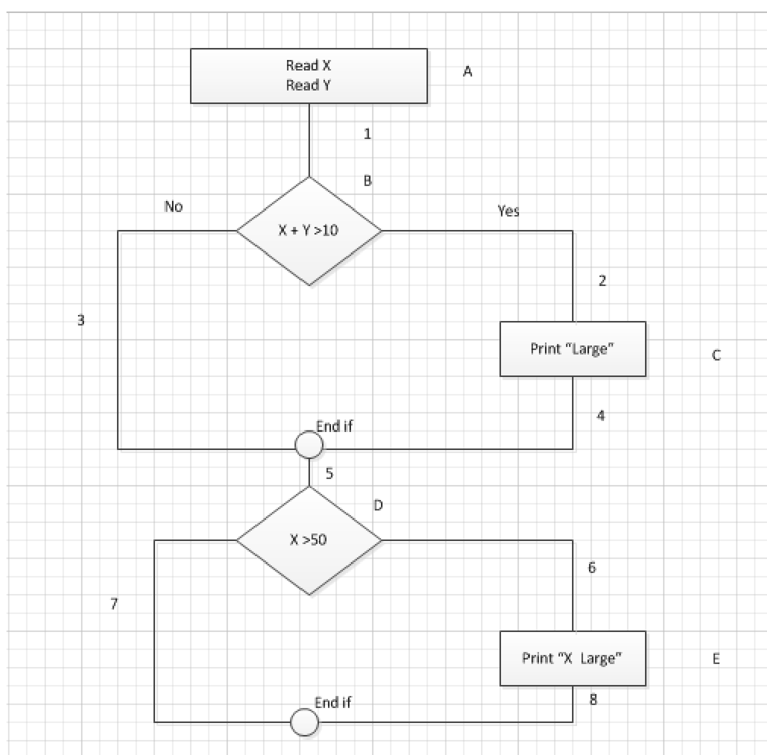
#### Testování rozhodovacích tabulek

Technika určená pro návrh testů, kde se využívá tabulky s jednotlivými kombinacemi vstupů a očekávanými výsledky.

### 3.3.2 White-box techniky

V případě white-box technik má tester znalost vnitřní struktury testovaného subjektu, tedy typicky zdrojového kódu. U těchto technik se určuje pokrytí části kódu podle daného kritéria. Následující jsou různé druhy pokrytí, podle toho jakou část kódu test vykonal:[11]

- Příkazy - Na diagramu 3.1 jsou příkazy označeny písmeny A,B,C,D,E.
- Větve - K větvení dochází například při vyhodnocování podmínek. Na diagramu 3.1 dochází k větvení například v bodě označené písmenem B, a to na větve B2 a B3.
- Cesty - Cesty jsou kompletní průchody diagramem. V případě diagramu 3.1 by se jednalo například o průchod A1-B3-5-D7.
- Podmínky - Jednotlivé podmínky, které se v průběhu vyhodnocují. Na diagramu 3.1 jsou označeny písmeny B a D.




**Obrázek 3.1:** Ukázka testované části aplikace, zdroj: [11]

Pokrytí se následně vypočítá podle vzorce:[11]

```
Pokrytí = počet vykonaných položek/celkový počet položek * 100%
```

## 3.4 Statické a dynamické testování

Při statickém testování se reviduje neběžící subjekt, tedy je prováděna revize a kontrola. Při dynamickém naopak kód spouštíme a provádíme nad ním testy.[1]

Statické testování je možné aplikovat pomocí nástrojů k tomu určeným nebo se velmi často využívá manuálního testování ve formě revizí - reviews. Revize se dá provést nad jakýmkoliv výstupem projektu, nejčastěji se jedná o kód, architekturu, design, specifikaci nebo požadavky. Využívat nástroje pro statické testování lze nad jakýmkoliv výstupem, který má formální strukturu. Nejčastěji se jedná o statickou analýzu kódu, případně lze testovat i požadavky.[4]

Největším rozdílem je, že statické testování hledá defekty přímo, naproti tomu dynamické testování hledá chyby způsobené těmito defekty. Pokud je tedy defekt objeven pomocí statického testování co nejdříve, je oprava defektu levnější, než kdyby byl objeven později pomocí dynamického testování.[1][4][8]

## 3.5 Testování spojené se změnami - confirmation a regression testing

Regresivní testování je druh testování spojený se změnami v daném systému, kdy dochází ke kontrole částí systému, ve kterých nedošlo k přímé změně, ale mohly by být ovlivněny změnami v jiných částech systému.[2]

Jako confirmation testing se označuje testování, zda-li byla objevená chyba opravdu v nové verzi software opravena.[4]

## 3.6 Exploratory testing

Jedná se o druh testování, který je zvlášť důležitý pro agilní prostředí. Využívá se při něm především kritických schopností testera, jeho znalost testovaného subjektu a jeho zkušeností z podobných projektů. Základem pro explorační testování jsou takzvané test charters, které určují jaké aspekty systému se budou testovat. Tyto test charter mohou mít různou úroveň detailu, ale je nutné aby nebyly příliš detailní a tím tester nepřišel o volnost a kreativitu při testování daného subjektu. Explorační testování není založené na skriptech, ale jedná se o manuální testování spadající do úrovně akceptačního testování, která je blíže popsána v sekci 4.5.[4][12]

Struktura test charters může být například následující:

```
Prozkoumej . . . <cíl>
Se . . . <zdrojem>
Aby jsi odhalil . . . <informaci>
```

Tato struktura však není striktně dána a některé test charters mohou být ve zjednodušené podobě, jako je například:

```
Otestuj ... <komponentu>
v nové verzi prohlížeče.
```

## 3.7 Shrnutí kapitoly

V této kapitole jsem uvedl, co jsou to druhy testování a jak se dělí. Detailněji jsem některé z hlavních druhů popsal. Uvedl jsem také nad jakými výstupy a v jaké fázi projektu je vhodné dané druhy testování aplikovat. U některých druhů testovaní jsem také uvedl popis konkrétních testovacích technik.

# Kapitola 4

# Úrovně testování software

V rámci následující kapitoly vymezuji pojem úroveň testování, anglicky test level. Dále uvádím a detailněji popisuji jednotlivé úrovně testování software tak, jak jsou rozděleny podle ISTQB Foundation Syllabus. V poslední sekci kapitoly vysvětluji na testovacích kvadrantech, jakým způsobem spolu souvisejí druhy a úrovně testování.

## 4.1 Definice pojmu úroveň testování

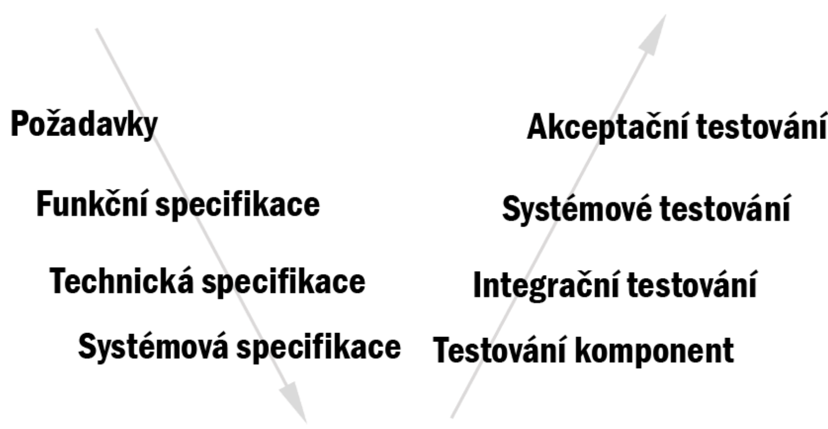
Jako úroveň testování se označuje skupina testovacích aktivit, které jsou společně spravovány a provozovány. ISTQB slovník popisuje úroveň testování následovně: „Testovací úroveň je specifická instance testovacího procesu." (překlad autora)[2]

Na obrázku 4.1 můžete vidět testovací pyramidu, jak je definována podle ISTQB. Typické pro testovací úrovně je, že čím výše jsou na pyramidě, tím větší je cena za opravu odhalené chyby. Směrem dolů stoupá i množství testů, které by mělo být implementováno v dané úrovni.

**Obrázek 4.1:** Testovací pyramida podle ISTQB - úrovně testování, zdroj: Autor

Každá úroveň testování odpovídá nějakým aktivitám v rámci SDLC a je pro ni potřeba specifické prostředí. Příkladem může být například testovací prostředí pro akceptační testování, které je co nejvíce podobné produkčnímu prostředí.

Obrázek 4.2 ukazuje jak u V modelu odpovídají testovací aktivity jednotlivým aktivitám SDLC.




**Obrázek 4.2:** V model - úrovně testování odpovídající aktivitám SDLC, zdroj: Autor

## 4.2 Testování komponent

Testování komponent, nebo také testování modulů či jednotek, anglicky unit tests, představuje testování malých, samostatně testovatelných celků. Testy se vykonávají nad kódem, detailním designem, datovým modelem či specifikací a obvykle se jedná o testování velmi malých funkčních celků jako je například jedna funkce či malá třída.[1][4]

Základním předpokladem je, že jednotlivé testy jsou na sobě nezávislé a nezáleží na pořadí, ve kterém jsou spouštěny. Testování komponent je ze všech úrovní testů nejméně nákladné na implementaci a je ještě levné odstranění defektů, které odhalí. Obvykle je prováděno vývojářem a snažíme se o co největší pokrytí. Zvláště u agilních modelů se využívá automatického testování komponent k regresnímu testování - zda změny v jedné části aplikace neovlivnili část jinou, ve které se žádné změny neprováděly.[1][2][4]

## 4.3 Integrační testování

Integrační testování se zaměřuje na testování komunikace - integrace - mezi komponentami nebo systémy. Integrační testování se provádí po testování komponent a obvykle testuje komunikační protokoly, rozhraní, workflows nebo popis externích rozhraní, a ne zda samotné systémy nebo komponenty fungují správně.[1][2][4]

Stejně jako testování komponent je integrační testování, hlavně v agilním prostředí, využíváno v rámci automatického testování a slouží k regresnímu testování. Integrační testování komponent bývá odpovědností vývojářů a integrační testování systému naopak odpovědností testerů.[4]

Objekty testování jsou obvykle subsystémy, databáze, API's, rozhraní nebo mikroservisy.[4]

## 4.4 Systémové testování

Systémové testování se nejčastěji vykonává na základě specifikace a snaží se testovat, zda subjekt odpovídá jak funkčním, tak nefunkčním požadavkům. Jedná se o testování celých činností, které může systém vykonávat, a toho jak se při vykonávání těchto aktivit testovaný subjekt chová.[1][4][2]

Jako předešlé dvě úrovně může i systémové testování být automatizováno a využíváno pro regresní testování.[4]

Nejčastěji jsou objekty testování aplikace, hardwarové a softwarové produkty nebo systémová konfigurace a data.[4]

## 4.5 Akceptační testování

Stejně jako systémové testování se akceptační testování snaží testovat, zda-li objekt odpovídá jak funkčním tak nefuknčním požadavkům, a zaměřuje se na subjekt jako na celek. Obvykle je u akceptačního testování vyžadována součinnost stakeholderů a osob, které budou v budoucnu se systémem pracovat.[1][2][4]

Akceptační testování se dá dále dělit podle cílů a toho kdo testování vykonává na typy popsané v následujících podsekcích.

### 4.5.1 Uživatelské akceptační testování - UAT

UAT se snaží validovat, zda systém vyhovuje požadavkům kientů a konkrétním budoucím uživatelům systému. Je obvykle prováděn v prostředí, které je co nejbližší produkčnímu, a vykonávajího ho nejčastěji testeři za součinnosti reálných uživatelů.[2][4]

### 4.5.2 Operační akceptační testování - OAT

OAT zjišťuje, zda budou moci správci systému udržovat systém funkční i v případě výjímečných situací. Provádí se obvykle osobami, které se budou v budoucnu o systém starat v produkčním režimu, nebo o odborníky přes danou oblast - například ze zabezpečení. Jedná se o testování záloh, obnovy záloh, aktualizace, migrační úkony, testování zabezpečení nebo výkonu.[1][2][4]

### 4.5.3 Regulační testování a testování podle smluv

Regulační testování a testování podle smluv má za cíl otestovat, zda subjekt splňuje akceptační kritéria stanovené ve smlouvách a regulacích (například zákony nebo bezpečnostní předpisy). Je možné, že výsledky tohoto testování budou v některých společnostech podléhat auditům.[2][4]

### 4.5.4 Alfa a beta testování

Pokud je složité napodobit produkční prostředí, například zátěž nebo způsob jakým bude subjekt využíván, může být použito alfa nebo beta testování. To provádí přímo potencionální zákazníci nebo externí testeři a má za cíl zjistit, zda systém bude moci fungovat v běžném provozu.[2][4]

## 4.6 Testovací kvadranty

Pro agilní prostředí definoval Brian Marick testovací kvadranty, které přiřazují druhy testů úrovním testů.[1] Druhy testů jsou popsané v kapitole 3. Model slouží k určení toho, jaké testy jsou pro testovaný subjekt potřebné, ale také jako srozumitelný přehled pro stakeholdery, vývojáře a testery.[4][13][12]

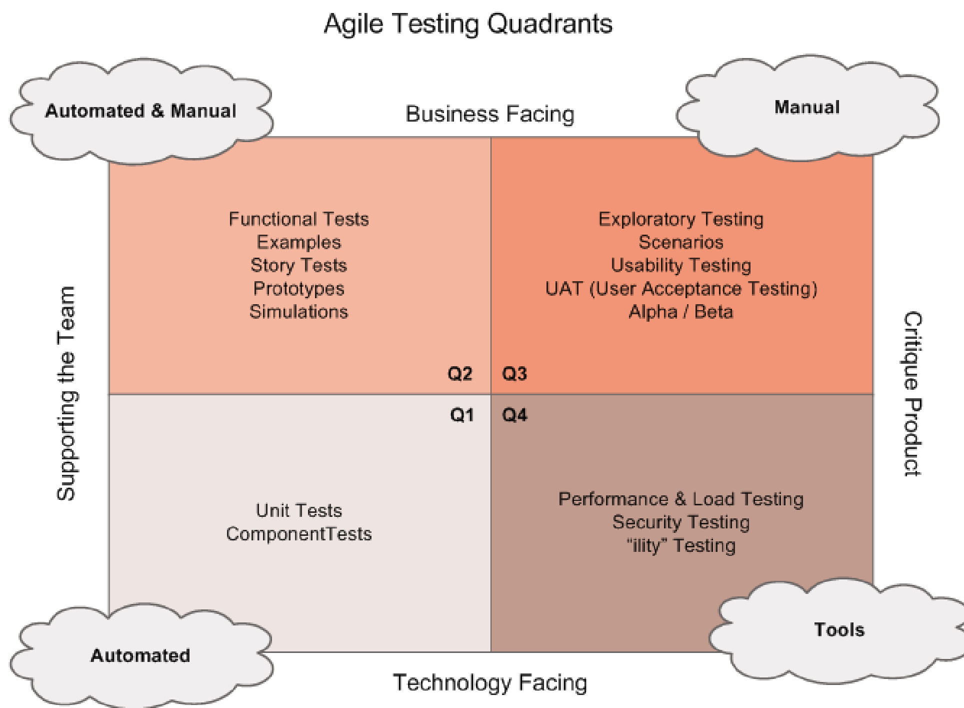
Obrázek 4.3 představuje model testovacích kvadrantů, jak ho definoval Brian Marick, ve zbytku sekce ho dále popisuji.

Levá část značí testy, které podporují vývojový tým a „potvrzují" chování systému. Pravá část diagramu značí testy, které „kritizují" - verifikují - daný systém.[14]

Vrchní část diagramu směřuje k byznysu, tedy ke klientovi, a spodní směřuje k technické stránce. Business facing testy jsou snadno uchopitelné i pro netechnikého odborníka z daného odvětví. Technology facing testy jsou oproti tomu lépe pochopitelné pro vývojáře.[14]

Testy mohou být plně manuální, automatické nebo manuální podpořené nástroji. Během jakékoliv iterace mohou být potřeba testy ze všech kvadrantů.[4][14]

[1] Bližší informace jsou k nalezení v původním článku: `http://www.exampler.com/old-blog/2003/08/22/#agile-testing-project-2`




**Obrázek 4.3:** Testovací kvadranty, zdroj: [13]

1. Kvadrant Q1 je na úrovni komponent. Měl by být automatizován a zahrnut v continuous integration procesu.
2. Kvadrant Q2 je na úrovni systémových testů a potvrzuje chování systému jako celku. Mohou být jak automatizovány tak manuální. Automatizovány mohou být využívány pro regresní testování.
3. Kvadrant Q3 je na úrovni systémých testů a UAT. Jsou často manuální.
4. Kvadrant Q4 je na úrovni systémových a OAT. Většinou se jedná o automatizované testy.

## 4.7 Shrnutí kapitoly

V rámci této kapitoly jsem uvedl definici testovací úrovně a detailněji popsal jednotlivé úrovně. Věnoval jsem se blíže tomu, jaké jsou objekty testování jednotlivých úrovní a jaké je pro ně potřeba testovací prostředí. Na konci kapitoly jsem uvedl, jak spolu souvisí druhy a úrovně testování, popsané pomocí testovacích kvadrantů.

# Kapitola 5

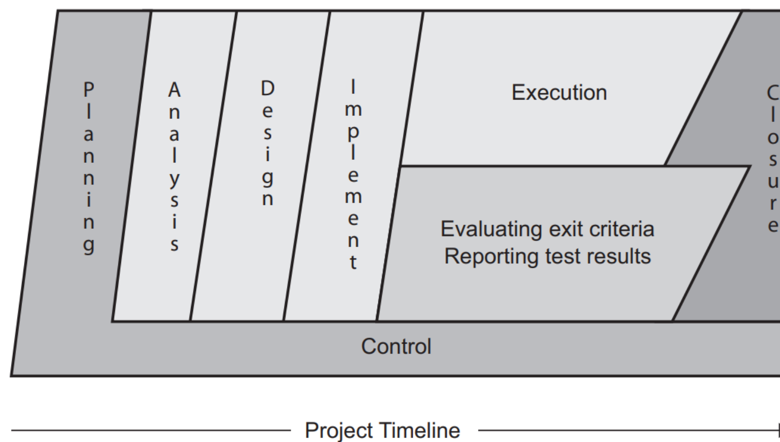
# Procesy v testování software

V tradičních modelech SDLC se vyskytuje několik procesů, s tím že některé z nich nemusí být snadné převést do agilních metodik. V následující kapitole popisuji tyto procesy, jakým způsobem fungují v tradičních modelech a u některých z nich uvádím jejich možné alternativy pro agilní prostředí.

## 5.1 Definice procesu v testování software

Na obrázku 5.1 můžete vidět typické procesy pro testování v tradičním modelu SDLC.

Je důležité úvést, že neexistují žádné přesně dané testovací procesy. Spíše se jedná o skupiny souvisejících aktivit, které sestávají z detailnějších úkolů, které se mohou lišit projekt od projektu, ale i jeden release od dalšího.[4]




**Obrázek 5.1:** Procesy testování v tradičním modelu SDLC, zdroj: [4]

I přes to, že se mohou jednotlivé skupiny aktivit zdát jako sekvenční, často se aplikují iterativně a dochází k jejich překryvu. Je proto nutné tyto skupiny aktivit přizpůsobit konkrétním potřebám daného projektu.[4]

## 5.2 Plánování monitorování a kontrola testů

Plánování sestává z aktivit, které začínají již na začátku projektu a pokračují v rámci jeho celého životního cyklu.

### 5.2.1 Testovací strategie a testovací plán

Testovací strategií se označuje dlouhodobý plán nebo obecný přístup k testování projektů. Měl by obsahovat informace, které nejsou specifické pro projekt, a měl by sloužit jako reference pro zaměstnance k získání high-level pohledu na fungování testování.[12]

Testovací strategie může obsahovat například informace o testovacích postupech, testování user stories, UAT, automatizaci testování, výsledky testování, nástroje pro testování nebo testovací prostředí.[12]

Dle zvolené testovací strategie vyplyne, jaké aktivity se musí naplánovat během plánovací fáze. Pokud například zvolíme risk-based strategii, musí se plánovací fáze odvíjet od analýzy rizik.[4]

Testovací plán je pro projekt specifický dokument, který definuje jaké aktivity a úkoly se musí v rámci testování na projektu provést, jaké úrovně a druhy testování se budou vykonávat, jaké jsou cíle jednotlivých úrovní a jaké techniky se použijí. Testovací plán také může obsahovat seznam vlastností projektu, které budou součástí testování a které nikoliv. Dále může obsahovat popis infrasktrutury vyžadované pro testování a zvláště pak pro automatické testy.[4]

Není vždy nutné, aby testovací plán byl ve formě dokumentu, avšak testovací tým by se měl vždy v plánovací fázi zamyslet nad otázkami, které jsou uvedeny výše, identifikovat možná rizika a závislosti. Zvláště pak u agilních projektů se využívá méně obsáhlých dokumentů popisujících testovací plán.[12]

### 5.2.2 Sledovatelnost a matice sledovatelnosti

U některých projektů vyvstává otázka, jak poznáme, které požadavky jsou a které zatím nejsou otestovány. Pro tento případ se využívá takzvaná traceability matrix - matice sledovatelnosti, která určuje vztah mezi jednotlivými požadavky a stavem testování. V agilních metodikách lze využít výhod, které sebou toto prostředí přináší. Zvláště pak úzké spolupráce mezi testery a vývojáři, kdy každé user story lze otestovat ihned po dokončení vývoje a případné změny komunikovat s vývojáři, takže je k dispozici přehled o změnách a lze podle něho upravit testovací use cases.[4][12]

### 5.2.3 Monitorování, kontrola testů a metriky

Monitorování a kontrola testů je skupina aktivit, která má za cíl sledovat stav testování a naplnění cílů stanovených v testovacím plánu, podle daných metrik. Metriky jsou typicky uvedeny v testovacím plánu.

Aby bylo možné sledovat stav testování efektivně a takovým způsobem, který je srozumitelný nejen týmu, ale také managementu a stakeholderům, je nutné dobře nastavit metriky, které pomáhají sledovat jakým způsobem tým naplňuje stanovené cíle. Zapojení stakeholderů už v raných fázích projektu může pomoci nastavit tyto cíle a metriky takovým způsobem, který umožní lepší sledovatelnost a kontrolu nad testováním.[4][12]

Metriky mohou být velice zrádné a pokud jsou nastaveny špatně, mohou se negativně odrazit na fungování týmu. Využívání metrik, které se zaměřují na fungování týmu jako celku jsou lepší než metriky zaměřené na konkrétní role. Jedním z příkladů takové metriky může být takzvaný cycle time, který značí jak dlouho trvá týmu určitá aktivita s daným výstupem, ať už se jedná o dodání nové vlastnosti projektu nebo o opravu defektu. Tyto metriky mají za cíl tým motivovat ke spolupráci na jejich zlepšení.[1][15]

Další konkrétní ukázky metrik uvádím v praktické části, při sestavování obecné testovací strategie v sekci 9.3.5.

## 5.3 Analýza a design testů

Následující sekce je rozdělena do dvou částí, první z nich popisuje analýzu testů, tedy zjišťování toho, co je předmětem testování. A ve druhé části je popsán design testů - jakým způsobem se bude subjekt testovat.

### 5.3.1 Analýza

V rámci testovací analýzy se rozhoduje, co bude předmětem testování na projektu. Tento proces sestává, podle ISTQB Foundation Syllabus z několika hlavních aktivit:[4]

- Analyzovat výstupy projektu vhodné pro danou úroveň testování.
  - Specifikace požadavků nebo všechny výstupy, které specifikují funkční i nefunkční požadavky na projekt, například user stories, byznys požadavky.
  - Informace o designu a implementaci, tj. všechny výstupy popisující strukturu systému, například diagramy a popis architektury.
  - Samotnou implementaci, nejčastěji kód.
  - Analýzu rizik.
- Kontrola výše uvedených výstupů a hledání nekonzistencí, nejasností, rozporů a dalších problémů, které mohou později vyvolat nejasnosti v návrhu a specifikacích.
- Identifikace vlastností a funkcionality, které se budou testovat.
- Definování a prioritizování test conditions pro vlastnosti a funkcionality systému.

Správně provedená analýza pomáhá zajistit konzistenci a jednoznačnost specifikace, což později v projektu může ušetřit nemalé zdroje, ať už kvůli změnám, tak opravám, které by byli potřeba z důvodu nekonzistentní specifikace.[4][3]

### 5.3.2 Design

V rámci designu testů se určuje, jakým způsobem se bude testovat. Vytváří se a prioritizují se konkrétní test cases z test conditions určených v testovací analýze. Definuje se, jaké prostředí a data budou pro dané testy potřeba. Jedná se tak i například o nástroje a infrastrukturu. Design testů by se měl držet testovací strategie a testovacího plánu.[4][3]

## 5.4 Implementace a spouštění testů

Implementace testů sestává z aktivit, během kterých jsou implementovány a prioritizovány testovací procedury a vytvářeny skripty pro automatizaci testů, provádí se kontrola, zda-li je vše připraveno pro další skupinu aktivit, tedy pro spouštění testů a dochází k přípravě infrastruktury, nástrojů a dat.[3][4][8]

Mezi aktivitami designu testů a implementace dochází velmi často k překrývání. Před provedením obsáhlé implementace testů je dobré vědět, jak funguje SDLC model a jak k velkým změnám může v projektu docházet.[3]

Aby bylo možné spustit testování, je nutné splnit určité předpoklady, většinou definované jako entry criteria v testovací strategii nebo plánu. Typicky se jedná o tyto podmínky:

- Je nakonfigurováno testovací prostředí a testovací tým k němu má přístup.
- Bug tracking systém je nastaven.
- Všechny komponenty jsou pod release management systémem.

V rámci procesu provádění testů jsou dle testovacího harmonogramu (anglicky testing schedule) spuštěny testy, ať už manuálně nebo pomocí nástroju ke spouštění testů. Dochází k porovnávání výsledků testů s očekávanými výsledky, analyzují se anomálie a reportují se případné chyby a zaznamenávají výsledky spouštění testů.

## 5.5 Vyhodnocení a reporting

Během vyhodnocení a reportingu dochází ke sběru dat z dokončených testů. Následně se z těchto dat získávají informace, které se prezentují managementu a stakeholderům a vyhodnocuje se, zda došlo k naplnění exit kritérií. Tyto aktivity fungují jako ukončovací fáze, ať už se jedná o nový release, konec iterace nebo konec samotného (testovacího) projektu.[4][3]

Vyhodnocení a reporting nejčastěji sestává z těchto aktivit:[4]

- Kontrola zda byli veškeré nahlášené defekty vyřešeny nebo přesunuty do backlogu projektu.
- Vytváření reportů a souhrnů z testování pro stakeholdery.
- Archivování testů a testovacího prostředí, například pro regresní testování.
- Předávání testovacího prostředí týmu, který bude projekt dále spravovat.

## 5.6 Shrnutí kapitoly

V této kapitole jsem popisoval procesy, které jsou typické pro testování ve většině SDLC modelů. U procesů, které vychází z tradičních modelů, jsem uvedl jejich ekvivalenty a specifika pro agilní prostředí. U jednotlivých procesů jsem popsal, jakým způsobem se do modelů SDLC zařazují, jaká jsou typická entry kritéria a co je jejich hlavním cílem.

# Kapitola 6

# Základní principy DevOps

V této kapitole se zaměřuji na základní principy DevOps přístupu. Nejprve uvádím základní pojmy, které zatím nebyli definovány v kapitole 2.1 a které jsou převážně používány v kapitolách týkajících se DevOps přístupu. Následně popisuji z čeho vznikla původní myšlenka DevOps, jaká byla motivace pro jeho vznik a na jakých principech je DevOps přístup postaven.

Podrobněji se poté zabývám definicí takzvané Continuous Delivery Pipeline, někdy také označované jako Deployment Production Line. Dále se v kapitole věnuji jejím jednotlivým složkám a principům, kterých využívá, jako jsou Continuous Integration popsané v sekci 6.4 a Continuous Delivery popsané v sekci 6.5. Uvádím jaké znaky by měl mít správný návrh této Delivery Pipeline a jaké místo v ní má automatické testování. Mimo jiné v této kapitole také uvádím příklady podpůrných postupů, jako je například Continuous Monitoring popsané v sekci 6.8.

## 6.1 Definice pojmů

V následující sekci vymezuji základní pojmy potřebné pro následující kapitoly.

### 6.1.1 DevOps

Jako DevOps se označuje moderní paradigma vývoje využívají automatizaci postupů, moderní nástroje a změnu pracovní kultury směřující především k rychlejšímu dodávání nových softwarových produktům ke klientům.

### 6.1.2 Key performance indicators

Správně nastavené klíčové indikátory výkonnosti, také někdy označované jen jako výkonnostní indikátory, určují jakým způsobem se daří společnosti, některému z celků společnosti, nebo některému z procesů dosahovat jeho předem nastavených cílů.

### 6.1.3 Business drivers

Jako business drivers se označují klíčové faktory, zdroje, procesy či aktivity, které ovlivňují finanční a operativní výsledky společnosti. Budou se lišit podle oboru, ve kterém společnost působí a podle cílů, které má společnost nastavené. Může se jednat například o legislativní, technologické nebo politické faktory.[16]

### 6.1.4 Version control system

Zkratkou VCS - version control system se označuje určitý druh nástrojů používaný pro verzování, uchovávání a zaznamenávání změn provedených na určitém produktu, nejčastěji se jedná o softwarový projekt. Příkladem může být například GitHub, GitLab nebo SVN.

## 6.2 Původní myšlenka

Původní myšlenkou DevOps, jak ji představili John Allspaw a Paul Hammond ve své prezentaci v roce 2009, bylo integrovat a redukovat problémy při komunikaci mezi odděleními vývojářů, operations a QA. Další důležité body na kterých je podle nich postaven DevOps přístup jsou následující:[17]

- Automatizovaná infrastruktura
- Sdílený VCS
- Build a nasazení v jednom kroku
- Sdílené metriky
- Změna pracovní kultury (komunikace, vzájemný respekt, důvěra, zdravý přístup k chybám)

K rozšíření DevOps došlo až v roce 2012, kdy tyto principy začaly využívat nadnárodní společnosti, jako je IBM nebo Hewlett-Packard.[18]

DevOps je silně založen na principech, které se v průmyslu, zvláště pak v automobilovém průmyslu, začali objevovat již od počátku 20. století. Jedná se především o přístupy jako jsou Lean, Six Sigma, 5S a PDCA. Další popis těchto principů není vzhledem k rozsahu této práce zahrnut. Jejich základní myšlenkou je však důraz na zefektivnění procesů ve výrobě redukcí plýtvání, průběžným zkalitňováním procesů a stavěním zákazníka v procesu na první místo.[18][19]

DevOps přístup je také velmi výrazně založen na agilních principech, hlavní prvky agilního přístupu jsou definovány v kapitole 2.3.

## 6.3 Continuous Delivery Pipeline

Continuous Delivery Pipeline označuje soubor principů a postupů, které umožňují velice rychle vytvářet, konfigurovat a nasazovat nové prostředí. Dále snadno umožňuje pomocí automatizovaného workflow, rozděleného do více fází, nasazovat otestovaný kód do produkčních nebo testovacích prostředí. Jedná se o základní soubor principů, které jsou využívány v DevOps přístupu.[20]

## 6.4 Continuous Integration

Princip Continuous Integration je postaven na myšlence průběžné integrace kódu mezi jednotlivými členy týmu nebo celými týmy. Na softwarových projektech bylo běžnou praxí, že vývojáři nebo týmy vývojářů integrovali změny, které implementovali na projektu až když byli kompletně hotové, což představovalo velké riziko. Velmi často se jednalo o složitý proces a zároveň vedlo k pozdnímu odhalení chyb v integraci. Princip průběžné integrace se snaží tyto rizika snížit za pomocí workflow využívajících sdílené VCS, automatických testů a CI serveru.[21][22]

### 6.4.1 Prerekvizity a postupy

Pro úspěšnou implementaci by měl softwarový projekt a tým nebo týmy, které na něm pracují, pochopit a splňovat následujcí prerekvizity:[7][23]

1. Jediný sdílený VCS a code base - umožňuje členům týmu nebo týmů pracovat nad stejnou code base a soubory, včetně konfigurace.
2. Automatický build - je nutné mít připravený build, který bude automatizovaný a půjde spustit z příkazové řádky. Skript nebo konfiguraci buildu je nutné vždy udržovat aktuální.
3. Dostatečně rozsáhlé pokrytí automatickými testy - tento bod je více popsán v sekci 6.7.
4. Pravidelný commit změn na hlavní větev - nejdůležitější princip průběžné integrace. Umožňuje co nejdříve odhalit případné chyby a snižuje riziko, že změny od jednotlivých vývojářů se budou výrazně ovlivňovat navzájem.
5. Dostatečně rychlé testy a build projektu - v případě že testy a build trvají příliš dlouho, tak může nastat situace, že členové týmu budou méně často provádět commit na hlavní větev a nebudou spouštět testy před commitnutím změn.
6. Transparentnost celého procesu - každý v týmu, včetně managementu projektu by měl mít okamžitý přístup ke stavu projektu. Jedná se především o stav testování a jednotlivých buildů.

7. Automatické nasazení do testovacího prostředí - testovací prostředí se snaží co nejvíce přiblížit tomu produkčnímu. Umožňuje nám otestovat projekt ve skoro produkčním prostředí.

## 6.5 Continuous Delivery

Continuous Delivery volně rozšiřuje princip průběžné integrace o předání funkčního a otestovaného projektu QA týmu a následně operations týmu. Dále o (možnost) automatické nasazovaní do produkčních prostředí. Cílem Continuous Delivery je umožnit průběžně dodávat v co nejkratších cyklech nové změny v projektu klientům.[20][24]

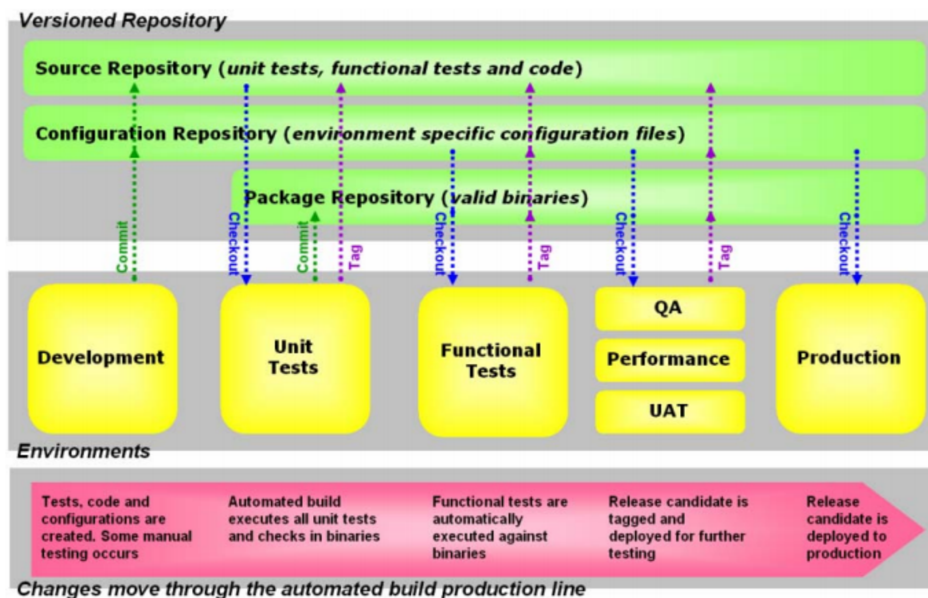
Je důležité zmínit rozdíl mezi pojmy Continuous Delivery a Continuous Deployment, které bývají často chybně zaměňovány. Continuous Delivery neznamená, že každá změna bude okamžitě nasazena do produkčního prostředí, ale že je pro každou změnu na projektu zaručeno, že je potencionálně „nasaditelná" do produkce a že tým je schopný tuto změnu do produkce nasadit velice rychle.[23]

## 6.6 Návrh a struktura delivery pipeline

Hlavní principy, které by se měli dodržovat při návrhu delivery pipeline jsou následujcící:[20]

- Každá fáze buildu by měla dodávat funkční software - nemít separátní fáze buildu pro meziprvky, jako jsou například frameworky.
- Na každé prostředí nasazujte stejné artifakty - konfigurace by měla být oddělená.
- Automatizujte testování a nasazení - rozdělte testování do více fází.
- Delivery pipeline by se měla rozvíjet společně s projektem.

K návrhu delivery pipeline je potřeba přistupovat pragmaticky a vždy si určit, které prvky mají přidanou hodnotu. Jak se postupem času rozvíjí projekt, bude se vyvíjet i delivery pipeline. Teoretický model kompletní delivery pipeline je ukázán na diagramu 6.1.[20]

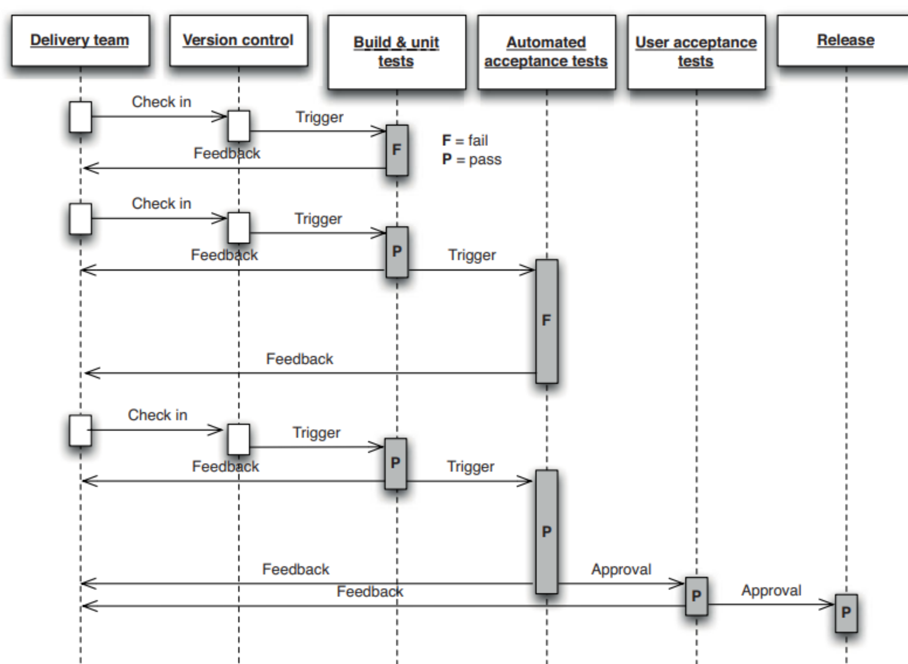



**Obrázek 6.1:** Obecná delivery pipeline, zdroj: [20]

Obecně by se dala delivery pipeline rozdělit do 4 následujících fází:[23]

1. Commit fáze - ověřuje na nejnižší úrovni, že projekt funguje po technické stránce správně. Jedná se především o spuštění buildu, unit testů a statické analýzy kódu.
2. Fáze automatických akceptačních testů - automatické testy na vyšší úrovni, toto téma je více popsáno v sekci 6.7.
3. Fáze manuálního testování - UAT a další vyšší úrovně testování. Ověřuje použitelnost systému.
4. Release fáze - nasazení systému ke klientovi, do produkčního prostředí.

Další způsob, jakým si představit delivery pipeline je diagram aktivit, který ukazuje jakým způsobem se na projektu pohybují změny. Diagram 6.2 je takovým příkladem.




**Obrázek 6.2:** Delivery pipeline jako diagram aktivit, zdroj: [23]

Pokud změna úspěšně projde všemi fázemi, včetně fáze uživatelských akceptačních testů můžeme mít jistotu, že systém splňuje následující:[23]

- Kód lze zkompilovat.
- Po technické stránce funguje kód podle očekávání, prošel unit a integračními testy.
- Kód dělá to, co analytici a uživatelé očekávají, prošel akceptačním testováním.
- Konfigurace a infrastrukturu je nastavena správně, kód byl nasazen v testovacím prostředí co nejvíce podobném tomu produkčnímu.
- Systém pro nasazení funguje.
- Všechno potřebné k nasazení je obsaženo ve sdíleném VCS.

## 6.7 Automatické testování

Testování v agilních prostředích by mělo být odpovědností celého týmu, jak již bylo zmíněno v sekci 2.4 o testování v agilních modelech. Správně navržený a implementovaný test suite zaručuje určitou úroveň kvality a zároveň slouží jako nejaktuálnější (spustitelná) dokumentace kódu.

V následující podsekcích je uvedeno, jakým způsobem mohou být automatizovány jednotlivé úrovně testů. Rozdělení úrovní vychází ze sekce 4.6

o testovacích kvadrantech, proto zde není uvedena detailní charakterstika jednotlivých kvadrantů a druhů testů, ale spíše stručný popis a best practices týkající se jejich automatizace.

### 6.7.1 Testy směřující k byznysu, které podporují vývojový tým

U kvadrantu Q2 se jedná převážně o funkční a akceptační testy. Tyto testy by měly ideálně být navrženy ještě než začne vývoj dané user story. Verifikují, že vyvíjená user story je splněna. Aby testy používaly systém stejně jako ho bude využívat konečný uživatel, je nutné je spouštět v prostředí nejvíce podobném tomu produkčnímu. Velkou výhodou je, že dostatečné pokrytí těmito testy nám zaručuje dobrou úroveň regresního testování. Zároveň některé nástroje umožňují generovat dokumentaci přímo z těchto testů, představují tedy aktuální dokumentaci projektu. Pro vývojáře představují možnost, jak si sami mohou ověřit, zda je daná user story splněna. Velkou nevýhodou akceptačních testů je to, že jsou velice náročné na údržbu. Prvním bodem automatizace by se měly stát akceptační testy takzvaných happy paths, které představují šťastný scénář pro nějaký funkční požadavek na systém. Akceptační testy by měly pokrývat převážnou většinu těchto happy paths, a až následně se soustředit na takzvané sad paths - alternativní průchody scénáři.[13][12][23]

### 6.7.2 Testy směřující k technologiím, které podporují vývojový tým

V modelu kvadrantů se jedná o kvadrant Q1, tedy o testy, které by měly být převážně odpovědností vývojářů. Druhy testů zahrnutých v tomto kvadrantu jsou hlavně jednotkové a integrační testy, které jsou detailněji popsány v sekcích 4.2 a 4.3. U těchto testů se předpokládá co největší možné pokrytí s tím, že spuštění těchto testů netrvá dlouho a měly by tedy být automatizovány a spouštěny častěji.[13][12]

### 6.7.3 Testy směřující k byznysu, které kritizují produkt

Kvadrant Q3 obsahuje manuální uživatelské akceptační testování, které kromě testování toho, zda je splněna specifikace, ještě validuje, zda je specifikace správná. Jedná se především o druhy testů popsané v sekci 4.5.

### 6.7.4 Testy směřující k technologiím, které kritizují produkt

Q4 je posledním kvadrantem, zahrnuje akceptační testování nefunkčních požadavků, které zahrnuje především testování zabezpečení, výkonu, dostupnosti a dalších. Jedná se o testování, které je velmi často opomíjeno. Tento druh testování vyžaduje specifické nástroje a znalosti. Tyto testy, ačkoliv mohou být i plně automatizované, trvají dlouho a jsou spouštěny méně často. Akceptační testy nefunkčních požadavků mohou být zahrnuty alespoň v nejjednodušší formě.[13][12][23]

## 6.8 Continuous Monitoring a další podpůrné principy

Kromě dvou základních principů, kterými jsou Continuous Integration a Delivery vznikly i další podpůrné principy. Jedním z nich je Continuous Monitoring, který umožňuje operations a vývojářskému týmu sledovat, zda se produkční prostředí chová, tak jak se od něj očekává. Umožňuje tedy sledovat a vyhodnocovat metriky v následujcích čtyřech oblastech:[18]

1. Výkon aplikace.
2. Výkon systému.
3. Chování uživatelů aplikace.
4. Uživatelské vnímání produktu.

Dalším moderním principem, který se rozšířil společně s Continuous Integration je Infrastructure as Code (IaC). IaC nám umožňuje zachytit infrastrukturu jako kód, který je poté možné replikovat a verzovat. Využívají se pro to moderní nástroje jako je Chef, Puppet, Ansible nebo Salt. Detailnější popis těchto principů není vzhledem k rozsahu této práce jejím obsahem.[18][25]

## 6.9 Shrnutí kapitoly

V této kapitole jsem uvedl definice základních pojmů, popsal základní principy DevOps a jakým způsobem vznikaly.

Dále jsem se v rámci této kapitoly věnoval základním principům využívaných pro implementaci takzvané Continuous Delivery Pipeline. Mezi tyto principy patří hlavně Continuous Integration a Continuous Delivery a podpůrné principy jako je Continuous Monitoring nebo Infrastructure as Code. Dále jsem definoval základní postupy a prerekvizity pro úspěšnou implementaci těchto principů. Mimo jiné jsem se také detailněji věnoval automatizaci testování v DevOps a tomu jaké místo má v Continuous Delivery Pipeline.

# Kapitola 7

# Adopce DevOps principů

V kapitole 6 jsem definoval hlavní principy a prvky DevOps přístupu, v této kapitole se budu věnovat přímo jejich adopci. Tedy přebírání těchto principů a postupů. Uvedu, jak je možné přistoupit k mapování stávajícího stavu, konkrétně identifikování neefektivity a plýtvání v procesech. Dále popíšu hlavní výzvy, kterým musí čelit většina projektů při adopci DevOps. Následně uvedu různé možnosti přístupů k implementaci DevOps principů a hlavní výzvy se kterými se společnosti setkávají při implementaci DevOps principů.

## 7.1 Stávající stav a cíle

Prvním předpokladem úspěšné implementace DevOps je správné pochopení stávajícího stavu procesu dodávání software a vhodně nastavené cíle. V této sekci jsou uvedeny základní postupy, pomocí kterých je možné modelovat stávající stav procesu dodávání software a identifikovat jeho slabá místa (bottlenecks) a případné plýtvání (waste). Dále je v této sekci popsáno, jak správně nastavit cíle, tak aby bylo možné je správně vyhodnotit a sledovat.

### 7.1.1 Mapování stávajícího stavu

Je mnoho způsobů jak se dívat na vyspělost organizace, týmu nebo samotného procesu dodávání software. Obecně by mělo mapování stávajícího stavu sestávat z následujících tří kroků:[18]

1. Identifikovat neefektivitu a plýtvání.
2. Zjistit příčinu neefektivity a plýtvání.
3. Vyvinout plán, jak se zaměřit na danou příčinu.

Za plýtvání nebo neefektivitu v procesu se považuje vše, co nepřináší žádnou konečnou hodnotu zákazníkovi. Může se jednat například o následující:[15][18]

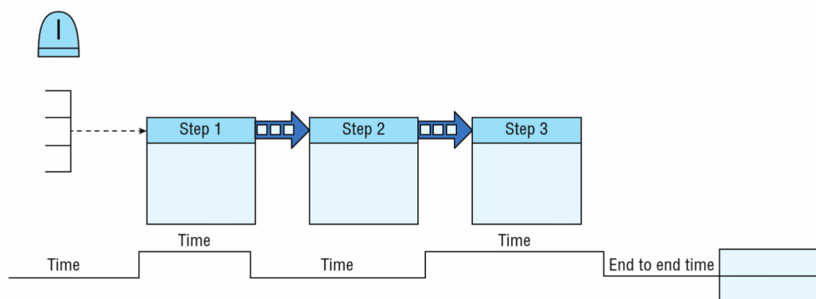
- Nepotřebné aktivity v procesu.
- Čekání na dokončení aktivity někoho jiného.

- Čekání na schválení.
- Čekání na přípravu prostředí.

Jednou z často využívaných metod, jak odhalit plýtvání v procesu vývoje software je takzvaný Value Stream Mapping (VSM). V kontextu vývoje software představuje mapování hodnotového toku požadavky, které na jedné straně vstupují do Delivery Pipeline a na druhé straně běžící aplikace. Delivery pipeline je podrobněji popsána v kapitole 6. Mezi požadavky patří všechno, co způsobí změnu aplikace běžící v produkci. Příkladem mohou být nové požadavky na funkcionalitu, změnové požadavky nebo opravy chyb. Pomocí modelu VSM lze proces zobrazit dvěma následujícími způsoby:[23][18]

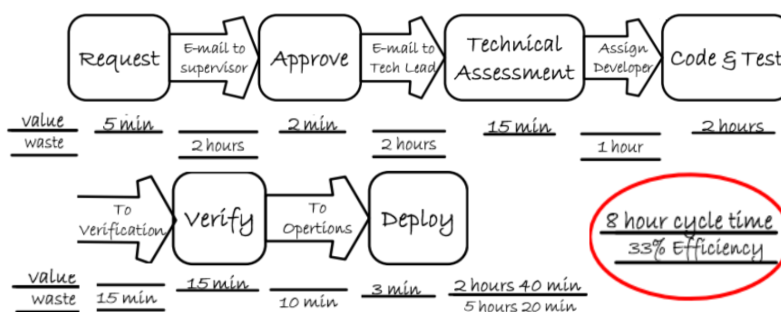
- Activity-centric - Proces je typicky popsán za pomocí vývojového diagramu nebo IDEF diagramu a popisuje jednotlivé aktivity, které vedou k vytvoření nějakého výstupu.
- Artifact-centric - Popisuje proces pomocí přechodů stavů artifaktů. Typicky je popsán pomocí stavového automatu.

Diagram 7.1 představuje příklad modelu procesu pomocí Value Stream Mapping.




**Obrázek 7.1:** Příklad modelu Value Stream Mapping, zdroj: [18]

Výpočet efektivity pomocí VSM představuje poměr mezi časem, který přidává hodnotu a celkovým časem daného cyklu. Konkrétní příklad s časovými údaji a vypočítanou efektivitou představuje diagram 7.2.




**Obrázek 7.2:** Konkrétní příklad VSM s hodnotami, zdroj:[26]

### 7.1.2 Nastavení cílů a business drivers

Hlavní prerekvizitou samotné adopce DevOps principů jsou vhodně identifikované business drivers společnosti, pro kterou dané řešení vyvíjíme. Tyto business drivers následně použijeme k nastavení vhodných metrik a KPI's, které nám budou sloužit ke sledování a vyhodnocení toho, zda jsme provedli adopci DevOps správně. Příkladem business drivers může být rychlost dodání na trh a podíl na trhu. V tomto případě by některé vhodné metriky mohli vypadat následovně:[18]

- Podíl mezi časem stráveným na inovacích / údržbě.
- Čas potřebný k nasazení.
- Čas mezi jednotlivými vydanými verzemi.

## 7.2 Implementace DevOps a Continuous Delivery Pipeline

Prerekvizitou implementace Delivery Pipeline je přijetí principů a postupů Continuous Integration, které jsou blíže popsány v podsekci 6.4.1. Dále je nutné vytvořit dostatečné pokrytí automatickými testy, které zabrání zanesení regrese. Jak určit vhodnou úroveň testů a jakým způsobem se dají testy automatizovat je popsáno v sekci 6.7.[24]

Timothy Fitz [27] ve svém článku o implementaci Continuous Delivery Pipeline uvádí čtyři základní možnosti, jak s implementací začít:

- Začít přímo s automatickým nasazením projektu do produkce. Dále zvyšovat úroveň pokrytí testy, podle míry regrese. Jedná se o možnost vhodnou pro začínající projekty, kde regrese ještě není tak nákladná na opravu. Lze očekávat zvýšenou míru chyb vzniklých regresí, dokud není vytvořené dostatečné pokrytí automatickými testy.
- Nejprve začít s automatickým nasazením projektu do testovacího prostředí. Postupná implementace a rozšiřování automatických testů v testovacím prostředí, dokud není vytvořena dostatečná úroveň vhodná pro produkci. Ve srovnání s první možností přináší menší riziko zanešení regrese do produkce a je tedy vhodnější pro experimentování s DevOps principy, nebo demonstraci principů konzevrativním stakeholderům. Velkou nevýhodou je to, že se tým připravuje o hlavní přidanou hodnotu Continuous Delivery a tou je rychlá zpětná vazba z produkčního prostředí, včetně skutečné zátěže a uživatelů.
- Začít s automatickým nasazování projektu s nejnižším rizikem. Jakmile je vytvořena infrastruktura a snižuje se regrese, lze postupně rozšiřovat na další oblasti/projekty. Výhodou je využití automatického nasazení v reálném prostředí a získání zkušeností bez větších rizik. Nevýhodou je,

že projektům s nižším rizikem je věnováno méně pozornosti a tedy může vést ke zpomalení cyklu zpětné vazby.

- Začít s automatickým nasazování projektu s nejvyšším rizikem. Vytvořit plnou Continuous Delivery Pipeline s vysokým pokrytím automatickými testy. Jedná se o přístup s velkou vstupní investicí. U nejkritičtějších projektů dochází k plnému využití výhod, které přináší DevOps principy, tedy hlavně rychlá zpětná vazba od klientů. Hlavním rizikem jsou nákladné opravy chyb spojené s regresí. Tento přístup je vhodný převážně pro velké společnosti, které se pokouší implementovat DevOps principy co nejrychleji je to možné a které mají zdroje na to, aby byli schopní vytvořit stabilní Continuous Delivery Pipeline s dostatečným pokrytím automatickými testy.

### 7.2.1 Časté výzvy při adopci DevOps

Hlavní výzvu při adopci DevOps principů představuje změna pracovní kultury, které se však výrazně liší podle pracovního prostředí, projektu nebo schopností členů týmu. V DevOps přístupu musí vývojáři být schopní mimo své obvyklé povinnosti také provádět testování na projektu a spravovat infrastrukturu a databáze. Všechny týmy, které se snaží o adopci DevOps principů se setkávají s následujícími výzvami:[19]

- Odstranění bariér v komunikaci v týmu nebo mezi týmy. Typicky mezi vývojáři a operations.
- Rozdělení komplikovaných systémů a architektur na menší celky, které mohou být spravovány a nasazovány nezávisle na sobě.
- Udržování konzistentního nastavení a prostředí, tak aby byly vždy jasné případné závislosti, verze a co přesně je nasazeno u klienta.

## 7.3 Shrnutí kapitoly

V této kapitole jsem uvedl, jakým způsobem začít s adopcí DevOps principů. Jednalo se především o vhodné identifikování stávajícího a budoucího stavu a odhalování neefektivit a plýtvání v procesu dodávání software. Dále jsem uvedl jak vhodně nastavit cíle a v neposlední řadě jakými způsoby lze přistoupit k samotné implementaci Continuous Delivery Pipeline. Nakonec jsem uvedl některé z častých výzev, se kterými se setká většina společností pri adopci DevOps.

# Kapitola 8

# Rešerše nástrojů

V této kapitole stručně popisuji nástroje využívané pro testování software a nástroje využívané v DevOps prostředí. Pokud existuje k nástroji alternativa, uvádím jejich srovnání. Vzhledem k projektu, pro který práce vzniká, jsou nástroje zaměřeny na projekty vyvíjené za pomocí Java Development Kit 8. Prostředí je dále popsáno v podsekci 9.2.1.

## 8.1 Nástroje pro testování

V této sekci je uveden základní popis, obvyklé případy užití a možné alternativy nástrojů využívaných pro testování. Ačkoliv jsou v této sekci nástroje rozděleny podle úrovní testů, nejedná se o jejich striktní rozdělení. Některé nástroje mohou být použity i pro více různých úrovní testů.

### 8.1.1 Testování komponent

Tato podsekce obsahuje nástroje využívané pro testování komponent. Detailní popis jednotkových testů je uveden v sekci 4.2.

#### JUnit a TestNG

JUnit je komplexní framework pro vytváření automatizovaných testů. Byl navržen primárně k testování na úrovni jednotkových testů. V kombinaci s jinými frameworky je ho však možné použít i na jiných úrovních.[28][29]

Základní funkcionalitou, obou frameworků (TestNG, JUnit4) je:[28][30]

- Podpora anotací.
- Exception tests - testy, které očekávají vyhození výjimky.
- Parametrizované testy - testy, kterému jsou předány data pomocí parametrů.
- Timeout - použití k ukončení testů, které trvají déle než stanovenou dobu.
- Spouštění testů pomocí build systému, z konzole nebo z IDE.

TestNG je framework inspirovaný frameworkem JUnit. Snaží se doplnit chybějící funkcionalitu JUnit4. Jedná se především o chybějící vestavěnou podporu pro reportování výsledků testů, seskupování testů a podporu paralelního běhu testů.[30]

JUnit5 je aktuálně nejnovější verzí testovacího frameworku JUnit. Oproti minulým verzím se skládá ze tří modulů JUnit Platform, JUnit Jupiter, JUnit Vintage a nabízí oproti JUnit4 a TestNG některé nové funkce, jako je například snadné opakované spouštění testů pomocí anotace, zobrazování volitelných jmen testů, také pomocí anotace a již zmíněné seskupování testů.[28]

Oba frameworky lze použít pro stejný účel, oba mají dobrou podporu pro vývojová prostředí a v obou případech se jedná o open-source frameworky.

### Mockito a EasyMock

Pokud testovaná část systému závisí na některé komponentě, lze využít jeden z frameworků Mockito nebo EasyMock k vytvoření takzvaných test doubles. Je několik druhů těchto test doubles:[29][31][32]

- Dummy object - Vyhovuje api reálného objektu, ale není nikdy použit. Typicky se využívá k předávání pomocí parametrů.
- Fake object - Nahrazuje reálný object jednodušší implementací, například in-memory databází.
- Stub object - Nahradí reálný objekt pevnými návratovými hodnotami.
- Mock object - Také nahrazuje reálný objekt, ale vlastní naprogramovanou očekávanou odpovědí.
- Spy object - Jen částečný mock object, který využívá část reálné implementace.

EasyMock a Mockito jsou frameworky, které poskytují velice podobnou funkcionalitu. Výhodou frameworku Mockito je rozsáhlejší komunita a přehlednější API.[31][32]

### PowerMock

Někdy je testovaný systém navržen takovým způsobem, který není dobře testovatelný. Příkladem může být třída označená jako final, nebo statické metody. Pro tyto případy nemá Mockito ani EasyMock podporu, a proto je nutné použít framework PowerMock, který rozšiřuje o tyto případy jejich funkcionalitu.[33]

### AssertJ

AssertJ je knihovna napsaná v jazyce Java, určená pro psaní takzvaných assertions, které ověřují zda-li je některá z podmínek pravdivá. Často je využíván právě v kombinaci s některým z frameworků, jako je například

JUnit nebo TestNG. AssertJ zpřehledňuje testovací kód a umožňuje lepší debugování testů díky dobře formulovaným chybovým hláškám. Především v kombinaci s novými funkcemi jazyka Java 8 umožňuje psát přehlednější testovací metody.[34]

Následující příklad je demonstrací testů, jaké umožňuje AssertJ psát.[34]

```
// basic assertion
assertThat(frodo.getName()).isEqualTo("Frodo");

// filtering a collection before asserting
assertThat(fellowshipOfTheRing)
  .filteredOn(character -> character.getName().contains("o"))
  .containsOnly(aragorn, frodo, legolas, boromir);
```

### 8.1.2 Integrační testování

V této podsekci se nachází nástroje využívané pro integrační testování. Integrační testování je detailněji popsáno v sekci 4.3.

#### Arquillian

Arquillian je framework pro vytváření automatizovaných integračních, systémových a akceptačních testů. Na rozdíl od frameworků ze sekce 8.1.1 nevytváří Arquillian takzvané test doubles, ale používá instance reálných objektů.[35]

Hlavní funkcionalita, kterou Arquillian poskytuje je:[35]

- Spravuje životní cyklus kontejneru nebo kontejnerů.
- Zabaluje testovací třídy společně se třídami a zdroji, na kterých test závisí do ShrinkWrap archivu, a nasadí archiv do kontejnerů, které spravuje.
- Podporuje Dependency Injection.
- Spouští testy uvnitř nebo proti nasazenému kontejneru.
- Umožňuje spouštět testy pomocí frameworků JUnit nebo TestNG popsaných v sekci 8.1.1

Kontejnery mohou být takzvaně embedded, managed (kde Arquillian spouští a vypíná danou Java Virtual Machine) nebo remote (tento aplikační server může být lokální nebo například v cloudu).

Arquillian momentálně podporuje několik implementací kontejnerů[1], z nichž například GlassFish má podporu všech tří typů kontejnerů.[36]

Arquillian také nabízí řadu rozšíření, které umožňují obohatit testy například o testování perzistentní vrstvy pomocí rozšíření persistence extension. Dále se dá například rozšířit o automatické uživatelské akceptační testy pomocí rozšíření Arquillian Dron a Selenium WebDriver, který umožňuje

[1] Implementace kontejnerů podporovaných Arquillianem: `https://docs.jboss.org/arquillian/reference/snapshot/en-US/html/containers.html`

spouštět prohlížeče a využívat webové aplikace jako reálný uživatel. Arquillian Drone sám spravuje životní cyklus prohlížeče.[35][36]

Arquillian je komplexní framework ke kterému momentálně neexistují alternativy, které by nabízely alespoň podobnou funkcionalitu pro Java Virtual Machine.

### 8.1.3 Akceptační testování

Následující podsekce obsahuje nástroje používané převážně pro akceptační testování, které je více popsané v sekci 4.5.

#### Selenium

Selenium je nástroj umožňující automatizovat webový prohlížeč a tím provádět automatické uživatelské akceptační testy. Selenium poskytuje tři nejdůležitější nástroje a to Selenium WebDriver, IDE a Server.[37]

Selenium Webdriver přijímá příkazy pomocí JSON-Wire protokolu a odesílá je prohlížeči spuštěnému pomocí třídy konkrétního ovladače některého z webových prohlížečů (ChromeDriver, FirefoxDriver, nebo IEDriver), což je implementováno pomocí ovladače konkrétního prohlížeče. Selenium WebDriver poskytuje velké množství klientských knihoven, které umožňují komunikovat s prohlížečem.[37][38]

Selenium server umožňuje spouštět testy na vzdálených serverech. Případně i pomocí Selenium Grid vytvářet skupiny vzdálených Selenium serverů a spuštět testy paralelně.[37][38]

Selenium IDE je add-on (rozšíření) pro prohlížeče Firefox a Chrome, které umožňuje spouštět a zároveň sledovat a nahrávat testovací skripty. Tímto způsobem lze vytvářet automatizované exploratory testing skripty a velmi rychle reprodukovat chyby.[37][38]

## 8.2 Nástroje pro DevOps

V této sekci se nachází nástroje využívané v DevOps prostředích. Více o DevOps prostředí a principech, které se v něm používají se nachází v kapitole 6 o DevOps principech a v kapitole 7 o adopci DevOps principů.

### 8.2.1 Version Control System

Version Control System (VCS) je nástroj, který se typicky využívá při souběžné práci více vývojářů na společném projektu. Udržuje změny provedené nad soubory a umožňuje je sdílet mezi jednotlivými vývojáři. Příkladem VCS je Git a Subversion popsané v podsekcích níže.

#### Git

Git je nainstalován na pracovní stanici a chová se zároveň jako klient, tak i server. S tím, že typicky existuje ještě centrální, sdílený repozitář na vzdáleném

serveru. Každý z vývojářů má lokální kopii projektu.[39]

Základním konceptem Gitu je takzvaný commit, který je ukazatel na snapshot všech souborů. Commit tedy zaznamenává historii změn provedených nad projektem. Commit také obsahuje odkaz na rodičovský commit, což umožňuješ vytvářet větvení commitů. Pokud se soubor mezi jednotlivými commity nezmění, Git soubor znovu neukládá, ale jen odkazuje na původní soubor.[39][40]

Branching model v Gitu je implementován jen jako odkaz na daný commit, jedná se tedy o nenáročný koncept, který však umožňuje vytvářet a mazat jednotlivé větve bez velkých zásahů do struktury repozitáře.[39]

Mezi nejpopulárnější poskytovatele patří GitLab a GitHub. Oba poskytovatelé také umožňují vytvořit vlastní instanci centrálního repozitáře na serveru třetích stran. S rozšiřující se popularitou Continuous Integration a Delivery, více popsaných v kapitole 6.4.1, se postupně z těchto nástrojů stávají kompletní DevOps platformy. Kromě podpory pro Continuous Integration a Delivery umožňují také nastavení komplexního workflow, například pomocí vytváření a správy úkolů, vytváření revizí kódů a kompletní dokumentace projektu. [39][41]

#### SVN - Subversion

SVN, neboli Subversion, funguje na principu odděleného klienta a serveru. A lokálně jsou udržovány jen soubory na kterých vývojář aktuálně pracuje. Vývojáři potom odesílají změny zpět na server. Kvůli tomu také SVN vyžaduje, aby byl vývojář stále online.[40]

Největší nevýhodou SVN je její branching model, který vytváří na serveru pro každou novou větev novou složku. Změny v této struktuře mohou způsobovat takzvané tree conflicts, které ve výsledku způsobují to, že je branching model oproti Gitu hodně komplexní a náročný na údržbu. Naopak výhodou je že SVN je oproti Gitu rychlejší pro velké projekty.[40]

### 8.2.2 Build

Obecně je hlavní odpovědností build systému usnadnit správu a kompilaci softwarového projektu. Další důležitou vlastností je správa závislostí, takzvaných dependencies. Většina softwarových projektů také nutně vyžaduje, aby nástroj pro build umožňoval rozlišovat různé profily pro různá prostředí. Tyto prostředí jsou typicky například lokální vývojové, testovací a produkční. Dále je také často potřeba, aby build systém podporoval projekty složené z více modulů.[42]

Build systémy se mohou lišit v mnoha bodech. Jedním z hlavních je způsob, jakým se zapisuje konfigurace pro jednotlivé build systémy. V současné době existují dva nejvyužívanější typy pro projekty v jazyce Java a to build systémy založené na konfiguraci ve formátu XML a systémy založené na konfiguraci v některém z DSL (domain specific language). Příkladem DSL může být jazyk Groovy, který má syntax podobnou jazyku Java. Vytváření konfigurace v DSL se podobá psaní skriptů. Build systémy založené na konfiguraci v DSL

umožňují vyšší úroveň přizpůsobení a kratší syntax. XML oproti tomu má pevnou strukturu a je na první pohled lépe čitelná.[43][44]

#### Maven

Maven je open-source zástupcem build systémů s konfigurací ve formátu XML. Důvodem je, že hlavním zaměřením Mavenu je správa závislostí a komplikované skripty pro build by bylo v Mavenu složité vytvořit.[44][45]

Maven je založen na principu spouštění pluginů, které vykonávají všechny aktivity během buildu. Mezi základní se řadí například kompilace (compiler), úklid po buildu (clean) nebo spouštění testů na různých úrovních (surefire, failsafe). Další skupinou jsou pluginy pro vytváření artifaktů různých typů, jako je například JAR, EAR a WAR.[2] Dodatečné pluginy zastávají mnoho funkcí například reportování, generování dokumentace a další.[44][46]

#### Gradle

Gradle je open-source zástupcem build systémů s konfigurací ve formátu DSL, konkrétně se jedná o jazyk Groovy nebo Kotlin. Oproti Mavenu poskytuje Gradle výrazně vyšší úroveň přizpůsobení. Dále se Gradle vyznačuje vyšší rychlostí buildu. Je to hlavně díky jeho schopnosti lepší pararelizace a tomu, že umí znovu používat soubory, které se od posledního buildu nezměnili.[47]

Gradle ke konfiguraci vyžaduje znalost jazyku Groovy nebo Kotlin, oproti Mavenu je tedy těžší se v konfiguraci zorientovat. S vyšší úrovní možné přizpůsobení se také skripty pro build mohou stát zbytečně komplikovanými.[45][47]

### 8.2.3 CI/CD

V kapitole 6 jsou popsány základní principy DevOps, zvláště pak v sekci 6.4 jsou uvedeny prerekvizity, které by měl softwarový projekt ke správné implementaci DevOps splňovat. Vlastnosti, které tedy vyžadujeme od nástroje nebo nástrojů pro Continuous Integration a Delivery jsou možnost automatizovat build, testy a nasazení do testovacího nebo produkčního prostředí. Jednou z dalších důležitých vlastností by mělo být napojení na sdílený Version Control System, tyto systémy jsou blíže popsány v sekci 8.2.1. Dle sekce 6.6 by se také delivery pipeline měla rozrůstat s projektem, očekáváme tedy od nástroje i určitou úroveň škálovatelnosti. Možnost rozdělit delivery pipeline na jednotlivé fáze je také velice důležitým faktorem ovlivňujícím výběr nástroje pro CI/CD. Dále by také bylo možné po nástroji požadovat podporu pro některý z dodatečných principů jako je například Continuous Monitoring, tyto podpůrné principy jsou blíže popsány v sekci 6.8.

[2] Rozdíl mezi jednotlivými typy artifaktů: `https://www.theserverside.com/feature/What-are-the-differences-between-EAR-JAR-and-WAR-files`

## Gitlab CI/CD

Prvním zástupce nástrojů podporujících jednotlivé fáze CI/CD je nástroj Gitlab CI/CD integrovaný do nástroje Gitlab, který je jedním z hlavních poskytovatelů sdíleného VCS. Gitlab je blíže popsán v podsekci 8.2.1.

Hlavní výhodou Gitlab CI/CD je, že veškerá funkcionalita je již vestavěná do daného systému a není tedy potřeba žádných dalších systémů třetích stran. Gitlab CI/CD nabízí podporu pro většinu hlavních postupů v DevOps. Prvním z nich je Continuous Integration, kdy Gitlab umožňuje nad již zmíněným sdíleným VCS spouštět automatický build a testy. Dále samotný Gitlab umožňuje poskytovat rychlou zpětnou vazbu ke změnám pomocí takzvaných merge requestů. Dalšími z postupů jsou Continuous Delivery a Deployment. Pro tyto postupy Gitlab CI/CD umožňuje z rozhraní Gitlabu automaticky, nebo po manuálním schválení, nasazovat do testovacího i produkčního prostředí.[41]

Mimo jiné je pomocí Gitlabu možné vytvářet a udržovat dokumentaci k projektu v Gitlab Wiki, vytvářet podporu pro workflow pomocí issues, které slouží jako karty úkolů. Issues je možné označovat takzvanými tagy, organizovat je pomocí nástěnek a propojovat s merge requesty a samotnými commity. Tyto vlastnosti umožňují pomocí Gitlabu vytvářet komplexní workflow.[41]

Dále je také možné do Gitlabu integrovat nástroje třetích stran, jako je například Prometheus pro monitorování produkčních a testovacích prostředí. Tyto nástroje umožňují podporu pro podrůrné principy jako je Continuous Monitoring.[41]

Základní verze Gitlabu je zdarma a poskytuje všechny z výše uvedených vlastností. Některé z pokročilých funkcionalit Gitlabu jsou však zpoplatněny.

Konfigurace Gitlab CI/CD se provádí pomocí nastavení v rozhraní Gitlabu a souboru .gitlab-ci.yml, který se umístí do sdíleného repozitáře. Tento soubor nám umožňuje konfigurovat komplexní delivery pipeline pomocí skriptů, které jsou poté spouštěny pomocí Gitlab Runneru.[41]

## Jenkins

Dalším zástupce nástrojů podporujících principy DevOps je Jenkins. Jedná se o open-source automatizační server, jehož jádro je možné rozšiřovat pomocí velkého množství pluginů. Pomocí pluginů je možné Jenkins integrovat s různými sdílenými VCS, jako je Git nebo SVN, které jsou blíže popsány v sekci 8.2.1. Hlavní výhodou nástroje Jenkins je jeho rozšiřitelnost, což sebou zároveň přináší určitá rizika, kterýmu jsou hlavně problémy s integrací pluginů, bezpečnostní problémy s pluginy a jejich podpora.[48]

Díky velkému množství pluginů nabízí Jenkins některé funkcionality, pro které v současnosti Gitlab podporu nemá nebo jsou v Gitlabu zpoplatněny. Na druhou stranu je díky tomu konfigurace nástroje Jenkins náročná a pro nové uživatele je těžší se v něm zorientovat.[49][48]

Konfigurace delivery pipeline je v nástroji Jenkins založena na Groovy jazyku, který spadá do kategorie DSL.

## 8.3 Shrnutí kapitoly

V rámci této kapitoly jsem provedl rešerši nástrojů využívaných pro testování software, rozdělených podle úrovní testů. Dále jsem provedl rešerši nástrojů využívaných v DevOps prostředí.

# Praktická část

# Kapitola 9

# Testovací strategie a testovací plán

V této kapitole stručně popisuji testovaný projekt Hodnocení pracovníků, a jakým způsobem jsem postupoval při návrhu jeho testovací strategie. Dále v této kapitole uvádím, jaká omezení bylo potřeba zvážit při vytváření testovací strategie a následně konkrétního testovacího plánu, který je uveden v sekci 9.4.

V případě Hodnocení pracovníků jsem zvolil přístup jedné obecné testovací strategie, použitelné i na jiných projektech s podobnými vlastnostmi a několika menších testovacích plánů, které popisují testování pro jednotlivé nové změny na projektu, typicky vytvořené pro každou iteraci zvlášť.

## 9.1 Popis testované aplikace Hodnocení pracovníků

Webová aplikace Hodnocení pracovníků umožňuje vedení pracovišť Fakulty elektrotechnické ČVUT komunikovat se svými zaměstnanci a sledovat jejich meziroční práci formou hodnocení. Toto hodnocení se skládá ze zaměstnancova sebehodnocení, hodnocení jeho nadřízeného a hodnocení vedoucího katedry.

Sebehodnocení se skládá z hodnocení zaměstnancových výsledků na základě publikovaných prací, výsledků na poli vědy, pedagogiky, ekonomiky a dalších oblastí. Dále zaměstnanec uvádí případné zahraniční pobyty, například ve formě odborných stáží. Zaměstnanec také vyplňuje plány na další rok a pět let a případné požadavky na své pracoviště.

K tomuto sebehodnocení následně přidávají hodnocení jeho nadřízení. Ti hodnotí jeho aktivity a to jakým způsobem se mu daří plnit stanovené plány z předchozích let. Zaměstnanec má následně možnost se k hodnocení nadřízených vyjádřit.

Mimo této základní funkcionality zahrnuje aplikace ještě přístup pro atestační komise, jejíž členové mohou kontrolovat hodnocení pro jednotlivá pracoviště.

## 9.2 Návrh testovací strategie

V této sekci popisuji jakým způsobem jsem postupoval při návrhu testovací strategie a jaké hlavní faktory bylo nutné zvážit při jejím návrhu.

Podle sekce 5.2.1 testovací strategie obsahuje, narozdíl od testovacího plánu, převážně statické informace. Jeho hlavním cílem je stanovit, jakým způsobem se má při testování jednotlivých projektů postupovat a pomáhá členům týmu zorientovat se v procesech testování. Další důležitou úlohou je stanovit, jaké budou procesy správy testů do budoucna, v případě změn na projektu.

Pro správný návrh testovací strategie bylo nutné zvážit především faktory prostředí, ve kterém bude testování vznikat. Zahrnuje to především technologie a nástroje využívané k implementaci projektů, skladbu a omezené kapacity týmu a omezení spojené s chybějící nebo neaktuální dokumentací projektu.

### 9.2.1 Technologie a infrastruktura

Velice důležitým faktorem ovlivňujícím testovací strategii je druh projektů, na které se bude testování aplikovat. Jedná se o webové aplikace s vrstvenou architekturou, vyvinuté za pomocí technologie Java EE 7 nebo 8 a jejích specifikací.

Vzhledem k rozsahu této práce nebude další popis těchto technologií jejím obsahem. Další informace o těchto technologiích lze nalézt například v oficiální specifikaci Java EE 8 technologií.[1]

Všechny projekty se také nacházejí ve sdíleném Gitlab repozitáři. Při implemetaci automatický testů je tedy nutné počítat i s podmínkou, že pro implementaci automatických testů a delivery pipeline se bude využívat jen technologií kompatibilních s tímto nástrojem.

### 9.2.2 Prostředí a další omezení

Hlavním požadavkem na testovací strategii, vzhledem k agilnímu prostředí, pro které je navrhována, je stručnost.

Vzhledem k omezené dokumentaci funkcionalit projektu bude nutné vytvořit jejich high-level popis podle jiných výstupů práce na projektu. Záznam funkcionalit bude dále sloužit jako aktuální dokumentace projektu, jeho stavu a stavu samotného testování.

Skladba týmů na tomto typu projektů, v daném prostředí je následující:

- Projektový manažer - Udává směr a podobu projektu a snaží se udržovat high-level pohled na jeho stav. Obvykle má přiřazených více dalších projektů, které řídí.
- Hlavní (seniorní) vývojář - Má nejlepší znalost projektu. Koordinuje vývoj, a řídí projekt po technické stránce.

[1] Specifikace Java EE 8 technologií: `https://www.oracle.com/cz/java/technologies/java-ee-glance.html#javaee8`

- Vývojáři (juniorní) - Je jich přiřazen různý počet podle momentální potřeby projektu, obvykle na jednom projektu pracuje jeden až čtyři juniorní vývojáři. Jejich hlavní odpovědností je vývoj nových funkcionalit a oprava chyb v projektu.
- Tester - Je přiřazen na projekt dle potřeby, vzhledem k malé kapacitě testovacího týmu není možné, aby byl projektu pevně přiřazen tester. Dochází k rozšiřování testování na jednotlivé projekty, jeho hlavní odpovědností je tedy formální mapování stávajícího stavu projektu, příprava testovacího prostředí, správa a implementace testů a správa automatických testů a prostředí pro automatické testy.
- Operations (správa aplikačního serveru) - Komunikuje s týmem na dálku. Monitoruje stav serveru. Komunikace s týmem probíhá hlavně v případě přípravy nového prostředí, upgradu infrastruktury nebo v případě problémů s testovacím nebo produkčním serverem.

Při návrhu testovací strategie je tedy nutné vzít v potaz překryv odpovědností v týmu a omezenou kapacitu testovacího týmu, který má na starost více projektů. U některých projektů také bude tester začínat s testováním a přípravou prostředí u projektu, který je již nasazen v produkčním prostředí.

## 9.3 Testovací strategie

Následující sekce obsahuje obecnou testovací strategii pro projekt Hodnocení pracovníků. Tuto strategii je možné aplikovat na podobný typ projektů.

### 9.3.1 Prostředí

Je nutné se ujistit, že prostředí a projekt splňují následující podmínky:

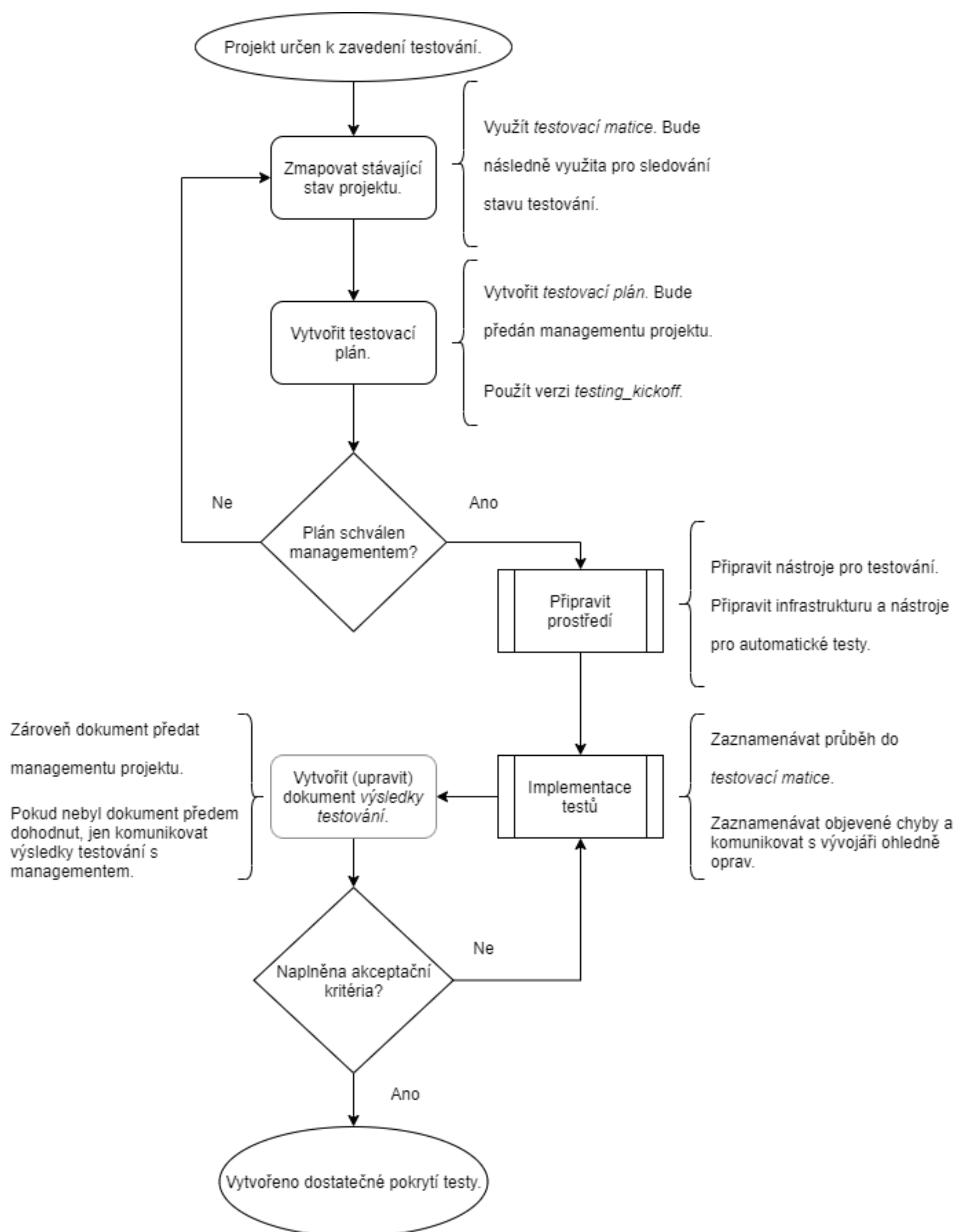
- Projekt je verzován ve sdíleném GitLab repozitáři.
- Projekt je vyvinut za použití technologie Java EE 8.
- Projekt využívá JDK8.
- Projekt využívá build systém - Maven. Případně Gradle.
- Projekt má připravené produkční a testovací prostředí.
- Testovací prostředí má nastavení co nejbližší tomu produkčnímu.

### 9.3.2 Postupy testování

Postupy jsou rozděleny podle fáze projektu z hlediska testování a samotného vývoje. Popisují zapojení testerů v celém životním cyklu software.

### Zavádění testování na projektu - testing kick off

Jedná se o speciální případ, kdy projekt již běží v produkčním prostředí. Zároveň u tohoto projektu bylo rozhodnuto, že je potřeba vytvořit pokrytí testy (stále se objevuje množství chyb v produkci, očekávají se změny v projektu - je potřeba zajistit, že nové změny neovlivní fungující projekt). Postup zavádění testování na projektu je dále popsán pomocí diagramu 9.1.




**Obrázek 9.1:** Proces zavádění testování na projektu, zdroj: Autor

Tento proces vyžaduje, aby měl projekt funkční testovací prostředí, co nejvíce podobné tomu produkčnímu.

Pro mapování stávajícího stavu aplikace lze využít následující výstupy práce na projektu:

- Kód - samotná implementace, dokumentace přímo v kódu (javadoc).
- Dokumentace požadavků - je možné, že nebude aktuální nebo dostatečně obsáhlá.
- Znalost členů týmů.
- Návrh - například návrh jednotlivých funkcionalit. Může se jednat jen o popsaný grafický návrh.
- Testy - pokud existují, mohou testy sloužit jako nejaktuálnější dokumentace projektu.

### Začátek/plánování iterace

Tento postup lze aplikovat na projekt, pro který máme již vytvořené určité pokrytí testy a očekávají se v začínající iteraci změny na projektu (implementace nové funkcionality, větší refaktoring a další).

Je potřeba, aby se tester zapojoval už při návrhu nové funkcionality, jak u byznys, tak technického návrhu. Pohled testera pomůže odhalit možné chyby ve specifikaci nebo návrhu. Pokud je daná změna navržena, aktualizuje tester testovací matici. Poté vytvoří nový testovací plán, který před začátkem implementace bude konzultovat s vývojáři a managementem projektu. Testovací matice je dokument vycházející z matice sledovatelnosti popsané blíže v podsekci 5.2.2. Struktura a význam testovacího plánu je blíže popsán v podsekci 9.3.5. Tester připraví prostředí potřebné pro testování, jedná se hlavně o následující tři body:

- Nástroje pro jednotkové a integrační testy.
- GitLab CI - Automatické jednotkové, integrační (případně akceptační) testy.
- GitLab CI - Automatické nasazení do testovacího prostředí.

### Průběh iterace

Tento postup se aplikuje v průběhu iterace, kdy se dané změny na projektu implementují.

Tester co nejvíce komunikuje s vývojáři ohledně postupu v implementaci. Pokud možno, tester implementuje testy jakmile to postup vývojářů umožní a konzultuje s nimi případné změny. Vývojáři se snaží dodávat co nejdříve funkční celky na testovací prostředí pomocí GitLab CI. Tester reportuje chyby a komunikuje je s vývojáři. Více o reportování chyb v podsekci 9.3.4.

#### Konec iterace - release

Tento postup se aplikuje na konci iterace, kdy dochází k dokončování práce na změnách v dané iteraci.

Nejintenzivnější zapojení testerů je právě v této fázi. Dochází k dokončování práce na změnách a je potřeba odhalit chyby, které se nepodařilo zachytit v průběhu iterace. Pokud je tak určeno v testovacím plánu, tak v této fázi dochází k testování na nejvyšší úrovni, tedy akceptačnímu testování. Tester reportuje chyby a komunikuje je s vývojáři. Více o reportování chyb v podsekci 9.3.4. Změny jsou hotovy, pokud jsou podle testera naplněny akceptační kritéria stanovená v testovacím plánu, případně pokud nejsou pevně stanovena, rozhoduje tester o konečnosti řešení po konzultaci s projektovým manažerem.

### 9.3.3 Nástroje, automatizace a úrovně testů

V této podsekci jsou uvedeny nástroje, které by na projektu měli být využity pro build, testování a continuous integration/delivery. Samotné nástroje a jejich možné alternativy jsou blíže popsány v kapitole 8.

#### Version Control System (VCS)

Projekty v daném prostředí již využívají jako VCS nástroj Git, s vlastní instancí od poskytovatele Gitlab. Je to proto vhodné v jeho používání pokračovat, v kombinaci s pokročilými funkcemi popsanými v podsekci 8.2.3 lze tento systém také rozšířit o podporu DevOps principů. Dalším důvodem je to, že tým pracující na projekt má již se systémem Gitlab zkušenosti.

#### Build

Vzhledem k častému střídání vývojářů na projektech a tomu, že projekty jsou rozsahově spíše menší a střední a zároveň nevyžadují speciální skripty pro build byl zvolen jako vhodný nástroj pro build Maven.

#### Testování

Pro testování je vhodnou kombinací JUnit5 framework v kombinaci s frameworkem Arquillian. JUnit5 umožňuje psát jednotkové testy s moderní syntaxí a v kombinaci s modulem JUnit Vintage a frameworkem Arquillian ho lze použít pro psaní integračních, systémových a akceptačních testů.

Arquillian je vhodné v případě potřeby rozšířit pomocí arquillian persistence extension. Toto rozšíření je určeno pro testování persistentní vrstvy.

Pro vytváření test doubles jsem zvolil framework Mockito vzhledem k rozšířenější komunitě a lepší dokumentaci než má framework EasyMock. Pokud bude potřeba je možné ho kombinovat s frameworkem PowerMock, který umožňuje vytvářet test doubles i v případech, kdy Mockito framework nestačí (třídy označené jako final, statické metody a další).

Pro assertions je vhodné využívat AssertJ, dojde tak k zpřehlednění testů a zlepšení jejich čitelnosti.

V případě, že projekt vyžaduje akceptační testování je vhodné použít rozšíření frameworku Arquillian - Arquillian Drone v kombinaci s nástrojem Selenium WebDriver.

Pro testy dodržovat následující konvence:

- Jednotkové testy
  - Dodržovat jmennou konvenci pro třídy: {NázevTestovanéTřídy}Test.java
  - Dodržovat jmennou konvenci pro metody: {názevTestovanéMetody}_{testovanáPodmínkaOčekávanýVýsledek}
  - Umístění souborů ve složce: /src/test/java
  - Umíštění zdrojů: /src/test/resources
- Integrační testy
  - Dodržovat jmenou konvenci: {NázevTestu}IT.java
  - Umístění souborů ve složce: /src/test/java
  - Umíštění zdrojů: /src/test/resources
  - Umíštění skriptů: /src/test/resources/scripts/{package}
  - Dodržovat jmenou konvenci skriptů spouštěných před/po třídě: {before/after}-{NázevTestu}IT.sql
  - Dodržovat jmenou konvenci skriptů spouštěných před/po metodě: {before/after}-{NázevTestovanéMetody}.sql
  - Umíštění společných/pomocných testovacích tříd: /src/test/java/{package}/testbase

## CI/CD

Pro konfiguraci delivery pipeline, konkrétně automatických testů je vhodné použít nástroje Gitlab CI/CD. Tento nástroj umožňuje nastavit automatické spouštění testů a nasazení do testovacího a případně produkčního prostředí. Gitlab také poskytuje všechnu požadovanou funkcionalitu a nástroje pro DevOps prostředí a zároveň tým již využívá Gitlab jako sdílený repozitář. Nastavení Gitlab CI/CD je také oproti alternativním nástrojům výrazně snazší. Pro tyto důvody je Gitlab CI/CD vhodnou volbou.

Delivery pipeline by se měla skládat z následujících 3 hlavních fází:

- Build
  - Compile - kompilace projektu.
- Test
  - Unit tests - jednotkové testy.
  - Integration tests - integrační testy.
  - Acceptance test - případné akceptační testy.

- Deploy
  - Package - vytvoření artifaktů.
  - Deploy (to test server) - nasazení artifaktů do testovacího prostředí.
  - Deploy (to production server) - nasazení artifaktů do produkčního prostředí.

Nastavení Gitlab CI by se mělo držet následujících postupů:

- Build a automatizované testy (fáze Build a Test)
  - Build a jednotkové testy spouštět na všech větvích při každém commitu.
  - Na test a master větvi spouštět build a všechny testy (unit, integrační, akceptační) při každém commitu.
  - U merge requestu spouštět build a všechny testy (unit, integrační, akceptační).
  - Na pracovních (feature) větvích spouštět při každém commitu i integrační a akceptační, jen v případě, že jejich spuštění netrvá nepřiměřeně dlouho. Jinak spouštět jen jednotkové testy a integrační s akceptačními spouštět s 2-3 hodinovým intervalem.
- Automatické nasazení do testovacího prostředí (fáze Deploy)
  - Aplikovat jen na test větev.
  - Před nasazením spustit všechny testy (unit, integrační, akceptační).
  - Vyžadovat manuální odsouhlasení kroku nasazení po tom co proběhnou všechny předchozí fáze.
  - Pokud selže některá z přechozích fází (build nebo testy), tak nedojde k nasazení.
- Automatické nasazení do produkčního prostředí (fáze Deploy)
  - Aplikovat jen po ověření funkčnosti kompletní delivery pipeline s využitím testovacího prostředí.
  - Master větev musí být nastavena jako protected.
  - Automaticky nasazovat z master větve, postupovat jako při automatickém nasazení do testovacího prostředí.

### 9.3.4 Nahlašování chyb

Jde o nástroj pro komunikaci mezi testery a vývojáři. Jedná se o rychlý přehled o stavu projektu nebo vyvíjené funkcionality. K nahlašování chyb se bude využívat systém Gitlab Issues.

Defekty, které je vždy nutné reportovat:

- Defekty v produkci.

- Defekty objevené po konci iterace.
- Defekty, které není možné opravit hned.

Pokud je možné zajistit, aby došlo k opravě chyby ihned, je doporučené upřednostnit komunikaci s vývojářem a opravit chybu ihned bez zaznamenávání, zvláště pokud se jedná o chybu během iterace.

#### Struktura issues

Issue by mělo obsahovat alespoň následující informace:

- Jak chyba vznikla?
- Popis chyby.
- Jaké bylo očekávané chování?
- Popis prostředí, ve kterém chyba vznikla.
- Případné přílohy (screenshot chyby, záznam z logu..).

#### Priorita issues

Dále je nutné rozlišovat závažnost objeveného defektu. Priority defektů jsou následující:

- Kritická - Chyba s největším dopadem. Omezujeme výrazných způsobem funkčnost aplikace pro větší množství uživatelů. Nutné ji ihned odstranit. S touto chybou nemohou být změny považovány za hotové.
- Vysoká - Aplikace je s touto chybou použitelná, ale chyba výrazně ovlivňuje její funkčnost.
- Střední - Na první pohled viditelná chyba, která však výrazným způsobem neovlivňuje funkčnost aplikace.
- Nízká - Chyba, která není na první pohled viditelná a neovlivňuje výrazným způsobem funkčnost aplikace.

Vývojáři by měli odkazovat na issue, které vyřešili pomocí „#ref {číslo issue}” ve zprávě commitu vždy, pokud je to možné. Dojde tak k propojení issue a její opravy.

### 9.3.5 Struktura testovacího plánu

Testovací plán se skládá ze dvou hlavních dokumentů. Je nutné dodržovat maximální stručnost a udržovat dokumenty aktuální. Zvláště pak dokument testovací matice.

### Testovací matice

Jedná se o aktuální high-level dokumentaci projektu. Je určena převážně testerům. Pomáhá udržet přehled týmu o stavu testování. Musí obsahovat následující:

- High-level požadavky.
- Testovanou oblast - komponentu nebo skupinu funkcionalit, pod kterou high-level požadavky spadají.
- Prioritu požadavku podle analýzy rizik (vysoká, střední, nízká).
- Úrovně testování
  - Jednotkové testy
  - Integrační testy
  - Akceptační testy (exploratory testing)
- Stav testování pro dvojici: požadavek - úroveň testování. (stavy testování jsou násedující tři: hotovo, není hotovo, nebude se provádět)

Testovací matice může také obsahovat jednoduchou analýzu rizik, která může sloužit k určení priority jednotlivých testovaných funkcionalit.

### Testovací plán

Jedná se o stručný souhrn plánu testování. Je určen převážně pro management projektu. Jeho struktura je následující:

- Verze.
- Rozsah testování.
- Přístup k testování.
- Hlavní identifikovaná rizika.
- Jaké funkcionality se budou testovat a na jakých úrovních. (podle situace je možné odkázat na testovací matici, která tuto informaci obsahuje)
- Jaké funkcionality se nebudou testovat.
- Sledované metriky. Konkrétní příklady metrik jsou popsány v podsekci níže.
- Stručný souhrn k testování daných změn.

#### Metriky

Sledované metriky na projektu mohou být následující:

- Počet odhalených defektů podle priority.
- Průměrná doba reakce týmu na report chyby.
- Pokrytí testy podle dané komponenty/skupiny funkcionality.
- Počet automatických testů nebo pokrytí automatickými testy.
- Stručné shrnutí ke každé testované funkcionalitě.

### 9.3.6 Výsledky testování

Tento dokument slouží jako stručný souhrn výstupů testování. Je určen převážně pro management. Týmu slouží jako ohlédnutí za iterací. Tento dokument není vždy vyžadován. Záleží na domluvě testera a managementu projektu. Někdy stačí jako výstup testování aktualizovaná matice testování. U fáze zavádění testování na projektu by však měl být požadován vždy. Možná struktura dokumentu je následující:

- Verze.
- Metriky, které byly testerem sledovány v průběhu testování. Konkrétní příklady metrik jsou popsány v podsekci 9.3.5.
- Odkaz na testovací matici.
- Stručný souhrn k provedenému testování.
- Srovnání odhadovaného času potřebného času k testování a reálně využitého času k testování.

## 9.4 Testovací plán pro projekt Hodnocení pracovníků

Tato sekce obsahuje konkrétní testovací plán pro projekt Hodnocení Pracovníků, který je uveden v podsekci 9.4.1. Testovací plán vychází z obecné testovací strategie uvedené v kapitole 9.3. Zároveň tato sekce obsahuje stručné shrnutí o postupu při vytváření testovacího plánu pro projekt Hodnocení pracovníků. Shrnutí postupu je popsáno v podsekci 9.4.2.

### 9.4.1 Testovací plán pro projekt Hodnocení pracovníků

Tento dokument byl vytvořen jako testovací plán pro projekt Hodnocení pracovníků. Slouží jako shrnutí plánovaného testování.

### Rozsah testování

Jedná se o fázi zavádění testování na projekt. Snaha je tedy pokrýt nejkritičtější části aplikace a zajistit tak správné fungování aplikace v blížícím se období konce akademického roku, kdy dochází k vyplňování hodnocení zaměstnanci a vedoucími kateder. Dále mají být na projektu představeny nové změny, je tedy potřeba zajistit, aby neovlivnily funkční celky aplikace, k čemuž se využije regresního testování.

### Přístup k testování

Vzhledem k tomu, že projekt Hodnocení pracovníků již funguje v produkčním prostředí, je hlavním cílem za pomocí strukturovaného exploračního testování odhalit chyby, které by se mohly nacházet v produkčním kódu. Dále bude potřeba vytvořit dostatečné pokrytí regresním testováním za pomoci automatizovaných unit a integračních testů tak, aby plánované změny neovlivnily funkční části projektu. Jednotkovými testy bude potřeba dosáhnout maximálního možného pokrytí. Za pomocí integračních testů bude potřeba pokrýt minimálně happy paths nejkritičtějších částí aplikace, kterým je dle analýzy rizik jednoznačně celý proces vytváření evaluace, včetně všech jeho stavů. Pro testování formuláře evaluace bude použito pokročilejších testovacích technik, jako je například analýza hraničních hodnot a tříd ekvivalence.

### Identifikovaná rizika

Hlavním rizikem je neaktuálnost a neúplnost dokumentace. Aby došlo ke snížení možného dopadu tohoto rizika, byl vytvořen dokument testovací matice, zahrnující high-level popis funkcionalit. Tento dokument vychází ze stávající dokumentace a byl konzultován s členy týmu a managementem projektu. Bude dále sloužit jako aktuální dokumentace projektu.

### Testovaná funkcionalita

Následuje soupis testované funkcionality:

- Celý proces vytváření evaluace, včetně všech jejích stavů. Pokrytí happy paths unit a integračními testy a exploračním testováním. Použít analýzu přechodů stavů.
- Formulář evaluace. Použití analýzu hraničních hodnot a tříd ekvivalence. Větší úroveň pokrytí integračními testy.
- Administrační část aplikace určená pro správu komisí a období pro hodnocení. Pokrytí happy paths pomocí exploračního testování.
- Lokalizace a další obecné scénáře (odhlášení, udržení preference jazyka, vypnutí emailových notifikací). Pomocí exploračního testování.

#### Netestovaná funkcionalita

Následuje soupis funkcionality, která se nebude testovat:

- Testování frontend oproti grafickému návrhu.
- Okrajové případy v procesu evaluace.
- Administrační část - přehled hodnocení.
- Historická funkcionalita import hodnocení.

#### Sledované metriky

Následující metriky se budou v průběhu testování sledovat:

- Počet odhalených chyb podle závažnosti.
- Stručný souhrn k testované funkcionalitě (skupině funkcionalit).

#### Obecné informace o testování

Součástí testování bude také automatizace unit a integračních testů pomocí Gitlab CI/CD. Detailní dokumentaci, analýzu rizik a postup testování lze nalézt v dokumentu testovací matice, který je součástí přílohy C.

### 9.4.2 Postup při návrhu testovacího plánu pro projekt Hodnocení pracovníků

Vzniku testovacího plánu předcházelo mapování stávajícího stavu projektu a soupis funkcionality do testovací matice. Vzhledem k neaktuálnosti a neúplnosti dokumentace bylo potřeba stav aplikace konzultovat s členy týmu a managementem projektu. Funkcionalitu jsem zaznamenal do testovací matice v různé úrovni detailu podle její priority. Například celý proces evaluace, který jsem identifikoval jako hlavní a nejkritičtější část aplikace, jsem zaznamenal jako jednotlivé user stories. Naopak některé části z administrační strany aplikace jsem zaznamenal jen ve zkrácené podobě.

Dále jsem vytvořil jednoduchou analýzu rizik k určení úrovně testování a priority jednotlivých funkcionalit. Analýzu jsem konzultoval s členy týmu. Tato analýza zároveň potvrdila, že nejrizikovější je proces evaluace. Ten tedy bylo potřeba pokrýt pomocí více úrovní testování. Proces evaluace také sestává z komplexního formuláře. Vzhledem k tomu, že se jedná o důležitou část aplikace, určil jsem, že na ni bude potřeba aplikovat některou z pokročilejších technik testování pro určení přechodů stavů a hraničních hodnot, tyto techniky jsou blíže popsány v sekci 3.3.

Dalším důležitým bodem bylo explorační testování, které vzhledem k rozsahu aplikace pomůže odhalit chyby hlavně v již zmíněném procesu evaluace. Pro explorační testování budou použity takzvané test charters, které pomáhájí snadno zaznamenávat testovací scénáře, a které znázorňují jaký byl cíl testování. Test charters jsou nástroje pro explorační testování, jejich bližší popis a přibližná struktura jsou popsány v podsekci 3.6.

## 9.5 Shrnutí kapitoly

V rámci této kapitoly jsem stručně popsal testovaný subjekt a vytvořil obecnou testovací strategii, která výchází z popisu prostředí, pro které vzniká, a s ním spojených omezení. Testovací strategie obsahuje obecný popis postupů, podle různých fází životního cyklu software a také strukturu a doporučený obsah dokumentů, který by měly v rámci testování vzniknout. Dále jsem vytvořil konkrétní testovací plán pro projekt Hodnocení pracovníků, který vychází z obecné testovací strategie a popsal jsem, jak jsem postupoval při jeho návrhu.

# Kapitola 10

# Implementace testů a delivery pipeline

V následující kapitole popisuji, jakým způsobem jsem postupoval při přípravě prostředí, návrhu a implementaci testů.

## 10.1 Příprava prostředí

V této sekci popisuji přípravu prostředí na projektu Hodnocení pracovníků. To zahrnuje především přípravu DevOps prostředí za využití Gitlab CI/CD popsané v podsekci 10.1.1. Dále přípravu testovacího prostředí pro jednotkové a integrační testování. Tento postup je detailněji popsán v podsekci 10.1.2.

### 10.1.1 Gitlab CI/CD

Gitlab CI/CD je systém podporující jednotlivé fáze DevOps a je blíže popsán v podsekci 8.2.3 o nástrojích pro Continuous Integration a Delivery.

Nastavení Gitlab CI/CD vyžaduje funkční instanci verzovacího systému Gitlab, alespoň verze 8.0. Projekt Hodnocení pracovníků tento systém již využívá jako repozitář pro sdílení zdrojového kódu mezi vývojáři pracujícími na projektu.

Samotné nastavení Gitlab CI/CD jsem prováděl dvojím způsobem. Prvním je nastavení takzvaných Runners přes grafické rozhraní systému. Tento Runner poté spouští konfigurační skript definovaný v souboru `.gitlab-ci.yml`, jehož vytvoření je druhou částí konfigurace. Runner může být virtuální stroj, virtuální privátní server nebo například Docker kontejner. Jsou dva druhy Runnerů a to Specific a Shared Runner. Specific Runner vyžaduje oproti Shared Runner instalaci na infrastrukturu. K této infrastruktuře nebylo, vzhledem k prostředí ve kterém tato práce vzniká, možné zajistit přístup. Zvolil jsem proto Shared Runner. Ten využívá pro spouštění skriptů speciální virtuální stroj na instrastruktuře systému Gitlab.

Nastavení Shared Runner se provádí v grafické rozhraní následovně:

```
Settings => CI/CD => Enable shared runners
```

Druhou částí nastavení bylo vytvořit soubor `.gitlab-ci.yml` obsahující nastavení GitLab CI/CD. Soubor je součástí příloh obsahujících zdrojové kódy a konfiguraci testů, jelikož je také součástí repozitáře.

Soubor se skládá z několika hlavních částí. První z nich je definování takzvaných stages, tedy fáze delivery pipeline. Typicky jsou tři a to následující:

```
stages:
    - Build
    - Test
    - Deploy
```

Hlavní složkou `.gitlab-ci.yml` jsou takzvané Jobs, které definují jednotlivé úkoly, které má Runner vykonat. Těchto úkolů může být definováno neomezené množství a každý z nich musí obsahovat skript, který má Runner vykonat. Dále obsahuje podmínku, při které se má vykonat. Každý Job musí mít také unikátní název.

Následuje ukázka úkolu pro kompilaci projektu Hodnocení pracovníků, fungujícím pod build systémem Maven:

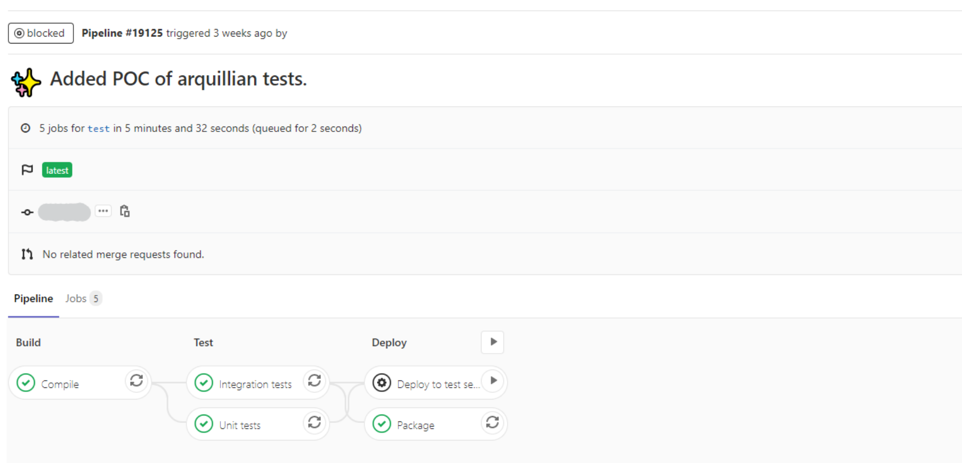
```
Compile:
  allow_failure: false
  stage: Build
  when: always
  script:
    - mvn $MAVEN_CLI_OPTS compile
```

Parametr `allow_failure` značí že pokud daný úkol selže, nemá pipeline pokračovat ve vykonávání dalších úkolů. Parametr `stage: Build` značí, že se má spouštět v rámci fáze Build. Jednotlivé fáze mohou obsahovat více úkolů. Parametr `when: always` obsahuje podmínku, která určuje kdy má dojít ke spuštění daného úkolu. Tento parametr může obsahovat následující hodnoty:

1. `on_success` - Výchozí hodnota. Spustí úkol, pokud předcházející úkoly úspěšně skončí.
2. `on_failure` - Spustí se jen pokud jeden nebo více předcházejících úkolů selže.
3. `always` - Nezáleží na statusu předchozích úkolů.
4. `manual` - Je možná spustit jen manuálně přes grafické rozhraní Gitlabu.
5. `delayed` - Od verze GitLab 11.14. Je možné spustit úkol s časovým zpožděním.

Toto umožňuje vytvářet komplexní pipeline, které je možná spravovat přes grafické rozhraní v systému Gitlab. Tento systém dovoluje sledovat průběh a stav jednotlivých úkolů, případně je ručně spouštět nebo restartovat.

Ukázka grafického rozhraní Gitlab CI na projektu Hodnocení pracovníků je znázorněna na obrázku 10.1.




**Obrázek 10.1:** Ukázka rozhraní Gitlab CI/CD, zdroj: Autor

Dalším důležitým bodem konfigurace je nastavení parametru `image:`, který určuje to, že se má pro spouštění úkolů používat Docker kontejner. Hodnota parametru také určuje, jaký Docker Image se má pro spouštění Gitlab CI úkolů použít. Pokud není uvedeno jinak, Gitlab stahuje Docker Image z Docker Hub úložiště[1]. Pro projekt Hodnocení pracovníků jsem uvedl následující konfiguraci pro předem definovaný Docker Image, který podporuje projekty v JDK8 a Maven3:

```
1 image: maven:3.3.9-jdk-8
```

Dále ještě konfigurační soubor `gitlab-ci.yml` obsahuje nastavení proměnných, které je možné používat v rámci konfigurace a nastavení cache:

```
1  variables:
2    GIT_SUBMODULE_STRATEGY: "recursive"
3    GET_SOURCES_ATTEMPTS: 3
4    MAVEN_CLI_OPTS: "--batch-mode --errors ..."
5    MAVEN_PACKAGE_OPTS: "-DskipTests"
6    MAVEN_OPTS: "-Dhttps.protocols=TLSv1.2 -Dmaven.r ..."
7    MAVEN_PROJECT: "eval"
8  cache:
9    paths:
10     - ${MAVEN_PROJECT}-ear/target/
11     - ${MAVEN_PROJECT}-ejb/target/
12     - ${MAVEN_PROJECT}-web/target/
13     - .m2/repository
```

Dále už jen nastavení samotných úkolů, z nichž uvedu jen úkol pro nasazení do testovacího prostředí a úkol pro jednotkové testy:

[1] Docker hub - úložiště obsahujcíí Docker Image: `https://hub.docker.com/`

```
...
Unit tests:
allow_failure: false
stage: Test
when: on_success
script:
  - mvn -v
  - export M2_HOME=/usr/share/maven
  - mvn $MAVEN_CLI_OPTS test
artifacts:
  reports:
    junit:
     - ${MAVEN_PROJECT}-ejb/target/surefire-reports/TEST-*.xml
     - ${MAVEN_PROJECT}-web/target/surefire-reports/TEST-*.xml

Deploy to test server:
  allow_failure: false
  stage: Deploy
  when: manual
  only:
    - test
  environment:
    name: Evaluation test server
    url: #URL k testovacímu serveru
  script:
    - mvn -B -Ptest,deploy clean install
    -Dapplication.server.password=${SECRET_KEY}
    -DAS_ADMIN_READTIMEOUT=1800000
...
```

Proměnnou `${SECRET_KEY}` je nutné definovat přes grafické rozhraní a obsahuje SSH klíč k testovacímu serveru testovacího prostředí. Dále parametr `only: -test` určuje, na které větvi se má daný job spouštět. V tomto případě jsem úkol nastavil tak, aby se spouštěl jen na větvi test. Úkol pro spouštění jednotkových testů ještě obsahuje cesty, do kterých se mají ukládat výsledky testování.

Nastavení proměnných se provádí přes grafické rozhraní Gitlab následovně:

```
Settings => CI/CD => Variables
    +
Potvrdit možnosti Protected a Masked
```

Výsledek je takový, že se při každém commitu se nad projektem provede build a spustí se jednotkové a integrační testy. V případě větve test se ještě čeká na manuální spuštění úkolu, který nasadí projekt do testovacího prostředí.

### 10.1.2 Jednotkové a integrační testy

Pro implementaci jednotkových testů jsem se rozhodl primárně použít nástroj JUnit5, který je blíže popsán v podsekci 8.1.1 o nástrojích pro jednotkové testy. Pro integrační testování se potom jedná primárně o nástroj Arquillian popsaný blíže v sekci 8.1.2 o nástrojích pro integrační testy. Vzhledem ke špatné kompatibilitě JUnit5 a frameworku Arquillian bylo potřeba využít ještě modulu JUnit Vintage, který umožňuje spouštět testy napsané pomocí frameworku JUnit4, který jsem ve své práci využil pro integrační testy.

Dále jsem využil více podpůrných nástrojů a rozšíření, které usnadňují psaní testů nebo umožňují testování částí aplikace, které není možné otestovat jen pomocí základních frameworků. Jmenovitě se jedná o framework AssertJ, který umožňuje přehlednější ověřování podmínek. Dále framework Mockito, který umožňuje vytvořit takzvané test doubles, tedy náhražky reálných objektů. Pro integrační testování persistentní vrstvy jsem využil rozšíření frameworku Arquillian - Arquillian Persistence Extension. Toto rozšíření za pomocí embedded h2 databáze umožňuje testovat i persistentní vrstvu aplikace pracující nad embedded databází. Tyto nástroje a rozšíření jsou blíže popsány v sekci 8.1 o nástrojích pro testování.

Pro správu závislostí a build projektu jsem využil již nastavený build systém Maven, pod kterým projekt Hodnocení pracovníku pracuje. Build systémy včetně Mavenu jsou popsány v podsekci 8.2.2.

#### Nastavení

Pro využití nástrojů bylo v první řadě nutné nastavit v projektu pomocí souboru `pom.xml` Maven Dependencies - závislosti. Tyto závislosti poté Maven sám spravuje. Projekt Hodnocení pracovníků je složen z více modulů a ty jsou následující:

- eval-ear - Vytváří EAR archiv a závisí na ostatních dvou modulech.
- eval-ejb - Obsahuje back-end aplikace, včetně integrace s dalšími službami jako je například e-mail. Dále obsahuje samotnou persistentní vrstvu. Vytváří JAR archiv.
- eval-web - Obsahuje front-end aplikace. Závisí na modulu eval-ejb. Vytváří WAR archiv.

Projekt obsahuje jeden rodičovský - parent `pom.xml`, který odkazuje na submoduly. Tyto submoduly samy také obsahují soubor `pom.xml`. Rodičovský soubor obsahuje nastavení společné pro všechny submoduly. Dále narozdíl od jednotlivých submodulů nevytváří žádný archiv.

Konkrétní závislosti, které jsem přidal do jednotlivých pom.xml souborů, jsou uvedeny v příloze D.

Dále bylo nutné vytvořit strukturu pro třídy s testy. Při vytváření struktury jsem se držel zásad stanovených v testovací strategii, v sekci 9.3. Jedná se o

standartní Maven strukturu, tedy testy jsou pro každý z modulů umístěny v adresáři {název modulu}/test/java.

Při vytváření testovacích tříd, metod a skriptů jsem se držel jmenných konvencí taktéž definovaných v testovací strategii, v sekci 9.3.

Pro spouštění jednotkových testů pomocí Mavenu jsem použil Maven plugin - Surefire. Tento plugin umožňuje spouštět jednotkové testy během testovací fáze buildu. Pro nastavení tohoto pluginu bylo potřeba přidat následující plugin do `pom.xml` souboru:

```
<plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-surefire-plugin</artifactId>
        <version>${surefire.version}</version>
        <configuration>
                <argLine>
                    -Duser.timezone=Europe/Prague
            </argLine>
        </configuration>
</plugin>
```

Tím byla příprava jednotkových testů hotová.

Pro spouštění integračních testů jsem využil druhý Maven plugin - Failsafe. Tento plugin je přímo určený pro spouštění integračních testů během verify fáze buildu. Pro nastavení tohoto pluginu jsem přidal následující do `pom.xml` souboru:

```
<plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-failsafe-plugin</artifactId>
        <version>${failsafe.version}</version>
        <configuration>
                <argLine>-Duser.timezone=Europe/Prague</argLine>
                <!-- Force new JVM for each test class-->
                <reuseForks>false</reuseForks>
                 ...
        </configuration>
         ...
</plugin>
```

Pro spuštění testů pomocí Arquillianu bylo potřeba provést dodatečné nastavení. Za prvé bylo potřeba do rodičovského `pom.xml` souboru přidat nový profil pro glassfish embedded a nastavit ho tak, aby byl ve výchozím stavu aktivní:

```
<profile>
        <id>glassfish-embedded</id>
        <activation>
                <activeByDefault>true</activeByDefault>
        </activation>
</profile>
```

Dále bylo potřeba přidat do testovacích zdrojů soubor `arquillian.xml`, který obsahuje nastavení embedded kontejneru ve kterém se budou spouštět integrační testy.

Pro zprovoznění integračního testování persistentní vrstvy bylo také potřeba nastavit embedded H2 databázi. Pro toto nastavení se využívají dva soubory a to `glassfish-resources.xml` a `test-persistence.xml`. V souboru `arquillian.xml` jsem poté přidal odkaz na nový datasource a to následovně:

```
<extension qualifier="persistence">
    <property name="defaultDataSource">
        jdbc/arquillian
    </property>
</extension>
```

V tomto případě soubor `glassfish-resources.xml` obsahuje nastavení potřebné pro připojení k H2 databázi, a to ve formě zdrojů společných pro celou aplikaci (takzvaně application-scoped resource). Nastavení v souboru je tedy následující:

```
<resources>
    <jdbc-resource pool-name="ArquillianEmbeddedH2Pool"
                   jndi-name="jdbc/arquillian"/>
    <jdbc-connection-pool name="ArquillianEmbeddedH2Pool"
                          res-type="javax.sql.DataSource"
            datasource-classname="org.h2.jdbcx.JdbcDataSource">
        <property name="user" value="sa"/>
        <property name="password" value=""/>
        <property name="url"
value="jdbc:h2:~/test;MODE=PostgreSQL;DATABASE_TO_LOWER=TRUE"/>
    </jdbc-connection-pool>
</resources>
```

Soubor `test-persistence.xml` je konfigurační soubor specifický pro testy. Tento soubor nahrazuje standartní konfigurační soubor `persistence.xml` pro JPA, ve kterém hlavně dochází k nastavení takzvaných persistence units. Tyto persistence units definují metadata potřebná pro JPA, včetně definovaných entit, nastavení transakcí a datových zdrojů.

Pro integrační testování přes více vrstev, jsem se rozhodl nasazovat do embedded kontejneru celé moduly eval-ejb i eval-web. Proto jsem vytvořil abstraktní třídu `AbstractIntegrationTest`, od které následně dědí všechny testovací třídy obsahující integrační testy. Tato abstraktní třída obsahuje metodu `createArchive`, která má anotaci `@Deployment`, a ve které pomocí ShrinkWrap systému vytvářím WAR archiv z modulu eval-web. Tento WAR archiv dále obsahuje submodul eval-ejb a nastavení H2 databáze pomocí souboru test-persistence.xml.

Tímto byla příprava pro integrační testování hotova.

### 10.1.3 Explorační testování

Pro explorační testování jsem zvolil nástroj Google Docs, ve kterém jsem vytvořil jednoduchou tabulku obsahující test charters, které jsou blíže po-

psané v sekci 3.6. Google Docs je jednoduchý nástroj, který umožňuje sdílet dokumenty mezi členy týmu. Tabulka test charters je součástí přílohy E.

### 10.1.4 Reporting chyb

Dále bylo potřeba připravit prostředí pro reporting chyb. Jako nástroj jsem dle testovací strategie zvolil Gitlab Issues a vytvořil jsem šablony s přibližnou strukturou reportů chyb.

Šablony se vytvářejí ve formátu markdown. Aby je bylo možné použít přes grafické prostředí Gitlab Issues, bylo potřeba v repozitáři vytvořit adresář `.gitlab/issue_templates`. Do tohoto adresáře jsem poté šablony uložil v souborech s koncovkou `.md`.

Šablony jsou součástí repozitáře, proto jsou i součástí přílohy obsahující zdrojové kódy.

## 10.2 Návrh testů

Při návrhu testů jsem vycházel primárně z dokumentu testovací matice, který je součástí přílohy C. Tento dokument vznikl v rámci testovacího plánu, který je uveden v podsekci 9.4.1. Dokument také obsahuje jednoduchou analýzu rizik pro konkrétní funkcionalitu projektu Hodnocení pracovníků. Z této analýzy rizik jsem vycházel při identifikování částí aplikace, na které je potřeba se při testování zaměřit a které mají naopak nižší prioritu.

Při návrhu testů jsem aplikoval přístup bottom-up, kdy jsem začínal s testováním komponent, které jsou nejméně závislé na ostatních částech aplikace a postupoval jsem směrem ke složitějším komponentám.

Při návrhu a samotné implementaci jsem zvolil následující pořadí testů:

- Jednotkové testy.
- Integrační testy persistentní vrstvy.
- Integrační testy využívající více vrstev.
- Explorační testování.

### 10.2.1 Jednotkové testy

Jednotkové testy jsou popsány v sekci 4.2. Při návrhu jednotkových testů jsem se snažil o co možná největší pokrytí testovaných komponent. Výstupem návrhu jednotkových testů byly určené třídy a metody, které bude potřeba pokrýt alespoň testovacími scénáři zahrnujícími happy paths, tedy validní průchody aplikací. Pojem happy path je vysvětlen ve slovníku pojmů v podsekci 2.1.4.

Následující třídy byly určeny pro jednotkové testování:

- CommissionBean - Třída obsahující část logiky správy atestačních komisí.

- EvaluationBean - Nejvíce obsáhlá třída. Zajišťuje primárně logiku spojenou se správou evaluací.
- FrontpageBean - Třída obsahujcí část logiky obsluhující výpis evaluací a požadavků na hodnocení podle rolí.
- EvaluationRequestBean - Třída zajišťující správu požadavků na hodnocení.
- EvaluationYearEditBean - Třída využívaná pro správu atestačních období.
- UsersBean - Tato třída obsahuje část logiky pro delegování požadavku na hodnocení. Zároveň bude však vyžadovat refaktoring. Určil jsem tedy, že bude otestována jen z části.

Ostatní třídy jsem určil jako nevhodné pro jednotkové testy. Ať už vzhledem k omezením spojeným s využíváním statických tříd a metod, nebo se jednalo o třídy, které zajišťují zastaralou nebo nevyužívající se funkcionalitu. Tuto funkcionalitu jsem v rámci testování identifikoval a zahrnul do výsledků testování.

### 10.2.2 Integrační testy

Integrační testy jsem se rozhodl rozdělit na dva druhy. Prvním z nich jsou integrační testy persistentní vrstvy, které testují, jestli se správně vykonávají operace nad databází. Jedná se o třídy v modulu `eval-ejb`. Při návrhu jsem se soustředil na co největší možné pokrytí persistentní vrstvy. Pro tyto třídy nebylo potřeba provádět kompletní nasazení všech modulů do embedded kontejneru. V rámci návrhu jsem proto vytvořil třídu `ShrinkWrapperDefaultConfig`, která obsahuje metody pro vytvoření archivu, který obsahuje jen třídy a závislosti pro testované třídy. Jedná se o následující třídy:

- CommissionFacade - Persistentní vrstva pro atestační komise.
- EmployeeFacade - Persistentní vrstva pro jednotlivé zaměstnance a vazby mezi zaměstnanci a jejich nadřízenými.
- EvaluationRequestFacade - Persistentní vrstva pro požadavky na hodnocení pro zaměstnance.
- EvaluationYearFacade - Persistentní vrstva pro atestační období.
- EvaluationFacade - Persistentní vrstva pro hodnocení zaměstnanců.

Druhý typ integračních testů, který jsem pro testování projektu Hodnocení pracovníků rozhodl použít jsou integrační testy, které prochází přes více vrstev aplikace. Testy přímo využívají třídy obsluhující komponenty na front-endu aplikace. Jednalo se především o následující třídy:

- CommissionBean
- EvaluationBean
- FrontPageBean
- EvaluationRequestBean

Tento druhý typ testů je náročný na údržbu a také na implementaci, rozhodl jsem se proto pro menší pokrytí a zaměření jen na happy paths testovacích scénářů.

### 10.2.3 Testovací data

V rámci návrhu testování bylo také potřeba připravit testovací data pro integrační testy, pro to jsem využil nástroj Mockaroo.[2] Ten umožňuje generovat testovací data pro schéma vytvořené podle SQL CREATE skriptu pro tabulku z databáze. Reálná data nebylo možné vzhledem k jejich citlivost použít. Testovací data jsou obsažena ve formě INSERT sql skriptů přímo v repozitáři, ve složce obsahující zdroje pro testy.

### 10.2.4 Explorační testování

Explorační testování umožňuje testerovi určitou volnost a kreativitu při vytváření testovacích scénářů, jak je blíže popsáno v sekci 3.6. Pro projekt Hodnocení pracovníků jsem se rozhodl pro strukturovanou formu exploračního testování a vytvořil jsem v nástroji Google Docs takzvané test charters také popsané v sekci 3.6 o exploračním testování. Tyto test charters vycházejí z testovací matice, která je součástí přílohy C a samotná tabulka test charters je také součástí přílohy E.

Při návrhu test charters jsem se soustředil na dostatečnou obecnost, aby umožnila testerovi volnost při testování. Vzhledem k omezením, které jsou spojena s front-end technologií použitou na projektu Hodnocení pracovníků, jsem se rozhodl, že pokročilejší black-box testovací techniky budou použity na úrovni exploračního testování. Popis těchto technik je uveden v sekci 3.3. Konkrétní použité pokročilejší techniky jsou:

- Analýza hraničních hodnot a tříd ekvivalence - Použité pro vstupní pole ve formuláři evaluace.
- Analýza přechodů stavů - Technika použitá pro jednotlivé stavy hodnocení - evaluace a přechody mezi nimi.

## 10.3 Implementace testů

V této sekci popisuji postup při samotné implementaci testů a při provádění strukturovaného exploračního testování.

[2]Nástroj pro generování testovacích dat: `https://www.mockaroo.com/`

### 10.3.1 Jednotkové testy

V této podsekci popisuji postup při implementaci jednotkových testů. Pro vytvoření testovací třídy jsem použil anotaci `@ExtendWith(MockitoExt...)`, která umožňuje spouštět JUnit5 testy společně s test doubles vytvořenými pomocí nástroje Mockito. Další použitá anotace `@MockitoSettings(...)` zabraňuje Mockitu vyhazovat `UnneccessaryStubbing` výjimky, pokud některá z testovacích metod nevyužívá některý z mocků. Připravená testovací třída poté vypadá následovně:

```
@ExtendWith(MockitoExtension.class)
@MockitoSettings(strictness = Strictness.LENIENT)
public class FrontPageBeanTest {
...
```

Pro vytvoření testovací metody se využívá anotace `@Test`, při implementaci ještě využívám anotace `@DisplayName("...")` a `@Tag("..")`. Obě anotace jsou určeny k třídění a lepší organizaci testů. Testovací metoda poté vypadá následovně:

```
@Test
@Tag("HappyPath")
@DisplayName("Test create new Evaluation creates correct new
    Evaluation object.")
public void createNewEvaluation_createsNewEvaluation() {
...
```

Pokud bylo potřeba vytvořit společná data nebo objekty pro více testů využil jsem metody s anotací `@BeforeEach`, která se spustí před každou testovací metodou. Pro spuštění společné metody pro testovací třídou se využívá anotace `@BeforeAll`. To samé platí i pro JUnit4, jen jsou anotace označeny jako `@Before` a `@BeforeClass`.

Pro vytvoření mocku objektu se využívá anotace `@Mock` a anotace `@InjectMocks` pro objekt do kterého chceme mocky vložit. Použití poté vypadá následovně:

```
public class FrontPageBeanTest {
...
@Mock
private EmployeeFacade employeeFacade;

@InjectMocks
private FrontPageBean frontPageBean;
...
```

Dále bylo ve většině jednotkových testů potřeba vytvořit mock třídy `FacesContext`, která obsahuje velké množství statických členů a metod. Aby jsem předešel použití další závislosti pro framework PowerMock, který umožňuje vytvářet mocky i pro statické členy a metody, tak jsem vytvořil třídu `ContextMocker`, která umožňuje vytvářet mock i bez potřeby PowerMocku. Použití třídy v testech je následující:

```
...
private static FacesContext facesContext;

@BeforeAll
static void initAll() {
    // mock faces context
    facesContext = ContextMocker.mockFacesContext();
    Mockito.doReturn(false).when(facesContext).isPostback();
}
...
```

Pro větší testovací třídy jsem využil nové funkcionality JUnit5, kterou jsou vnořené testovací třídy. Ty je možné vytvořit následovně:

```
...
@Nested
@DisplayName("Initialization methods for Evaluation")
class EvaluationInitialization {
...
```

Tímto způsobem je možné rozdělit velké testovací třídy na menší celky a určit pro ně vlastní proměnné, jméno a vlastní `@BeforeEach` a `@AfterEach` metody.

U některých testů bylo potřeba ověřit stejnou logiku, jen pro jiná data. Například se jednalo o jednotlivé stavy evaluace. Pro tento případ jsem použil další z nových funkcionalit knihovny JUnit5 a tím jsou parametrizované testy. Parametrizovaný test místo anotace `@Test` využívá anotace `@ParameterizedTest(..)` a některé z anotací, které určují jaký zdroj dat se má využít. Já jsem v testech například využil anotaci `@EnumSource(..)`, která umožňuje spustit test několikrát pro vybrané hodnoty výčtového typu. V definici metody je ještě potřeba uvést parametr daného typu a test se následně spouští s definovanými daty. Definice parametrické testovací metody vypadá například následovně:

```
@Tag("...")
@ParameterizedTest(name = "Test with {index} with param {0}")
@EnumSource(value = EvaluationState.class, names = {"EVAL", "
    EVAL_MGMT", "FEEDBACK"})
@DisplayName("...")
public void test_someTestMethodName(EvaluationState
    evaluationState) {
...
```

Pro ověřování podmínek v testovacích metodách jsem využil framework AssertJ, ukázka takových assertů je následující:

```
assertThat(evaluation.getYear()).isEqualTo(evaluationYear.
    getId());
assertThat(employee.getEvaluations().contains(evaluation)).
    isTrue();
```

Dalším velice častým prvkem testů bylo vytváření mocků a ověřování, že některá z metod mock objektu byla zavolána nebo zavolána se správnými parametry. Příklad takového využití frameworku Mockito je následující:

```
...

// return object when findLastYear() gets called
Mockito.doReturn(evaluationYear).when(evaluationYearFacade).
    findLastYear();

// verify that updateFromTemplate got called with 2 parameters
// first is any instance of Evaluation class
// second is equal to evaluationTemplate object
Mockito.verify(evaluationFacade).updateFromTemplate(Mockito.
    any(Evaluation.class), Mockito.eq(evaluationTemplate));

...
```

Všechny implementované jednotkové testy jsou součástí přílohy obsahující zdrojové kódy testů.

### 10.3.2 Integrační testy persistentní vrstvy

Pro testovací třídu obsahující integrační testy persistentní vrstvy jsem použit anotaci `@RunWith(Arquillian.class)`, která říká frameworku JUnit aby použil Arquillian jako testovací kontroler.

Dále jsem použil anotaci `@Cleanup(phase = TestExecutionPhase.NONE)`, která nastavuje Arquillian tak, že neřeší čištění databáze po každém testu, ale musí být vyčištěna ručně pomocí skriptů. Dále anotace `@ApplyScriptBefore` a `@ApplyScriptAfter` určují jaké skripty obsahující testovací data se mají pro daný test spustit. Skript se jménem začínajícím na after- vrací databázi do stavu před spuštěním testů. Tyto skripty mohou být uvedeny na úrovni třídy, to se pak skript spouští pro každou testovací metodu, nebo na úrovni metody, skript se poté spouští jen pro danou metodu. Testovací třída vypadá následovně:

```
...
@RunWith(Arquillian.class)
@Cleanup(phase = TestExecutionPhase.NONE)
@ApplyScriptBefore("scripts/session/employee/before-
    EmployeeFacadeIT.sql")
@ApplyScriptAfter("scripts/session/employee/after-
    EmployeeFacadeIT.sql")
public class EmployeeFacadeIT {
...
```

Dále musí každá testovací třída obsahovat statickou metodu s anotací `@Deployment`, která vrací typ `JavaArchive` nebo některou z jeho implementací. Jak je zmíněno v podsekci 10.1.2, vytvořil jsem během přípravy testovacího prostředí třídu `ShrinkWrapperDefaultConfig`, která obsahuje metody pomocí nichž je možné vytvořit archiv obsahující základní závislosti

potřebné pro spuštění integračních testů nad persistentní vrstvou. Použití této třídy v metodě s anotací `@Deployment` je následující:

```
@Deployment
public static JavaArchive createArchive() {
    return ShrinkWrapperDefaultConfig
            .createTestPersistenceSetting()
            .addPackages(false, ShrinkWrapperDefaultConfig.
                getCorePackages())
            .addPackage(UDBConnector.class.getPackage())
            .addPackage(AbstractFacade.class.getPackage());
}
```

Pomocí metod `.addPackage(..` a `.addClass(..` je možné do archivu přidávat další třídy potřebné pro spuštění testů. Metoda `.addAsResource(..` umožňuje přidávat nebo nahrazovat zdroje specifické pro testy. V mém případě jsem ji využil pro přidání konfigurace připojení k databázi pomocí souboru test-persistence.xml.

Samotnou implementaci testovacích metod jsem následně prováděl stejně, jako u jednotkových testů. Testovací metody jsem označil anotací @Test.

Pro ověřování podmínek jsem použil framework AssertJ. Hlavní rozdílem však je použití frameworku JUnit4 a to, že se v integračních testech závislosti vkládají přímo pomocí anotace @Inject nebo @EJB a nevytvářejí se test doubles.

Všechny implementované integrační testy pro persistentní vrstvu jsou součástí přílohy obsahující zdrojové kódy testů.

### 10.3.3 Integrační testy přes více vrstev aplikace

V této podsekci popisuji jakým způsobem jsem postupoval při implementaci integračních testů, které využívají více vrstev aplikace.

Konfigurace je obdobná jako v předchozí podsekci 10.3.1. Jen archiv, který se nasazuje do embedded kontejneru je WAR, který obsahuje oba moduly eval-ejb a eval-web podle závislostí z jejich pom.xml souborů. Jelikož se archiv neliší pro jednotlivé testovací třídy, mohou dědit od abstraktní třídy AbstractIntegrationTest, která obsahuje statickou metodu s anotací @Deployment. Testovací třída poté vypadá následovně:

```
@RunWith(Arquillian.class)
@ApplyScriptBefore("scripts/evaluation/before-CommissionBeanIT.
    sql")
@ApplyScriptAfter("scripts/evaluation/after-CommissionBeanIT.sql"
    )
@Cleanup(phase = TestExecutionPhase.NONE)
public class CommissionBeanIT extends AbstractIntegrationTest {
...
```

Stejně jako u předchozích integračních testů jsem i v tomto případě využíval pro ověřování podmínek framework AssertJ. V několika případech bylo potřeba

využít i frameworku Mockito pro mockování tříd, které byly příliš vázané na technologii JSF používanou pro front-end aplikace.

Všechny implementované integrační testy přes více vrstev jsou součástí přílohy obsahující zdrojové kódy testů.

### 10.3.4 Explorační testování

Explorační testování jsem prováděl podle test charters, které jsou uvedeny v příloze E nad testovacím prostředím, které je nastaveno tak, aby co nejvíce připomínalo prostředí produkční. V případě odhalení chyb jsem je zaznamenával do Gitlab Issues podle připravené šablony, které je popsaná v testovací strategii, v sekci 9.3.

Pro některé testovací scénáře bylo potřeba použít pokročilejší testovací techniky, jako je analýza hraničních hodnot a tříd ekvivalence popsané v podsekci 10.2.4. Jelikož se jednalo vždy o velice jednoduchá vstupní pole bez složitých omezení stačilo na určení hraničních hodnot vytvořit za pomocí tužky a papíru jednoduchou tabulku.

Typickým příklad je vstupní textové pole omezené shora na tisíc znaků. Takovéto vstupní pole by mělo pouze 2 třídy ekvivalence pro vstupy s délkou menší než tisíc znaků a s délkou větší než tisíc znaků. Zároveň by testované hraniční hodnoty byly určeny následovně:

- Maximum - 1 (999 znaků) - validní vstup.
- Maximum (1000 znaků) - validní vstup.
- Maximum + 1 (1001 znaků) - nevalidní vstup.

Další pokročilejší technikou, kterou jsem použil pro testování byla analýza přechodu stavů pro jednotlivé stavy hodnocení. Analýzu jsem provedl jako jednoduchou tabulku, která je součástí přílohy F.

## 10.4 Shrnutí kapitoly

V rámci této kapitoly jsem popsal způsob, jakým jsem postupoval při přípravě prostředí pro Continuous Integration a Delivery. Dále jsem popsal postup při přípravě prostředí pro testování, návrhu testů a samotné implementaci testů. Mimo jiné jsem také popsal, jakým způsobem jsem prováděl explorační testování a uvedl příklady jednotlivých druhů testů a konfigurace.

# Kapitola 11

# Vyhodnocení testování

V rámci této kapitoly uvádím konkrétní dokument výsledky testování tak, jak je definován v testovací strategii v podsekci 9.3.6. Dále v rámci tohoto dokumentu provádím vyhodnocení testování na základě sledovaných metrik stanovených v testovacím plánu, v podsekci 9.4.1.

## 11.1 Výsledky testování pro projekt Hodnocení pracovníků

Tento dokument slouží jako stručný popis a zhodnocení testování na projektu Hodnocení pracovníků.

### 11.1.1 Vyhodnocení na základě odhalených chyb

V rámci testování bylo odhaleno celkem 22 chyb. Jejich rozdělení podle priority lze nalézt v tabulce 11.1.

Názvy konkrétních odhalených chyb, které byly zaznamenány do Gitlab Issues jsou součástí přílohy G. Dvě z těchto chyb byly ihned opraveny.

Dále byly během testování odhaleny části aplikace, které obsahují nevyužívaný nebo zastaralý kód. Jmenovitě se jedná o následující funkcionalitu:

- Import hodnocení
- Odesílání emailů

V rámci white-box testování byly identifikovány potencionálně problematické části aplikace, jedná se hlavně o kontrolu správného uzavírání streamů, práce s pamětí a chybějící logování. Tyto problémy byly v rámci testování průběžně konzultovány s vývojáři a budou opraveny v rámci plánovaného refaktoringu.

### 11.1.2 Sledované metriky

První sledovanou metrikou je množství chyb podle priority. Tyto hodnoty jsou uvedeny v tabulce 11.1.

| **Priorita** | Kritická | Vysoká | Střední | Nízká | **Celkem** |
|---|---|---|---|---|---|
| **Množství** | 0 | 4 | 9 | 11 | **22** |

**Tabulka 11.1:** Počet chyb podle priority, zdroj: Autor

Druhou metrikou je stručné shrnutí k testované funkcionalitě. V rámci testování jsem zaznamenával stav pro kritické části aplikace:

- Proces evaluace - Na tuto část aplikace byl během testování kladen největší důraz. V rámci jednotkového a integračního testování pro ni bylo vytvořeno dostatečné pokrytí, které zaručuje, že by v případě změn neměla být tato základní funkcionalita ovlivněna. Potencionálně problematické části kódu byly konzultovány přímo s vývojáři. Chyby, které byly odhaleny přímo v procesu evaluace jsou spíše střední a nižší závažnosti a přímo neovlivňují tuto funkcionalitu.
- Atestační komise - Pro tuto část bylo vytvořeno pokrytí jednotkovými a integračními testy, které zahrnuje převážně happy paths. Pro tuto funkcionalitu nebyla odhalena žádná závažnější chyba, která by přímo ovlivňovala její funkcionalitu. Spíše se jedná o chyby, které nejsou dobré z hlediska uživatelské přívětivost, jako například špatné přesměrování.
- Roční plány - Pro tuto část aplikace bylo vytvořeno pokrytí hlavně jednotkovými testy. Dále byla otestována v rámci exploračního testování. V této části aplikace nebyla odhalena žádná přímo související chyba.
- Administrační strana aplikace - Tato část aplikace byla otestována jednotkovými a integračními testy jen okrajově pro svou nižší prioritu. V rámci exploračního testování byli otestovány happy paths pro tuto část aplikace a nebyla odhalena žádná závažnější chyba.
- Nefunkční testování - V rámci exploračního testování byly odhaleny problémy s lokalizací převážně u méně dostupných textů a dialogů. Dále byla odhalena chyba s chybějící stránkou pro chyby při nenalezení stránky a chyby na serveru.

Konkrétní pokrytí testy podle user stories je uvedeno v testovací matici, která je součástí přílohy C.

### 11.1.3 Shrnutí

V rámci prováděného testování bylo připraveno prostředí pro jednotkové a integrační testy. Zároveň bylo nastaveno Gitlab CI/CD, které umožňuje spouštět automatické testy a provést nasazení do testovacího prostředí z grafického prostředí Gitlab.

Dále bylo vytvořeno pokrytí jednotkovými a integračními testy, které zaručují to, že v případě změn na projektu, nedojde k zanesení chyb do již funkčních celků aplikace.

V rámci exploračního testování bylo odhaleno několik chyb spíše střední a nižší priority, které byli nahlášeny do systému Gitlab Issues a které si mohou převzít vývojáři k opravě.

Mimo jiné také v rámci testování vznikla aktuální dokumentace a popis funkcionality ve formě testovací matice.

# Kapitola 12

# Návrh workflow

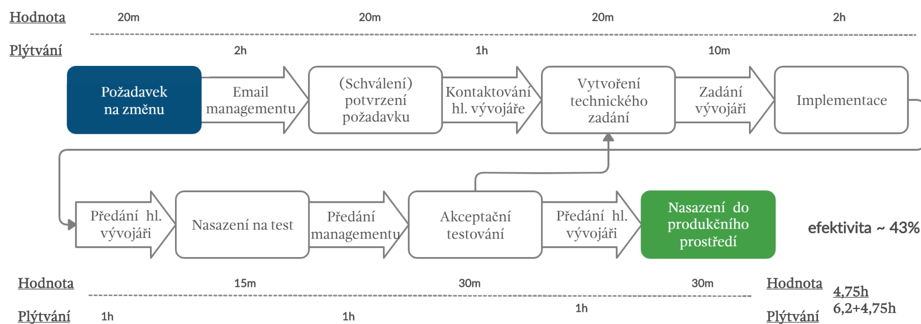
V této kapitole mapuji stávající workflow na projektu Hodnocení pracovníků a snažím se o nalezení případné neefektivity, nebo plýtvání v procesu dodávání software. Dále popisuji, jaké by mělo být worfklow po zavedení DevOps nástrojů a automatického testování na projektu. V sekci 12.2 popisuji navržené workflow a demonstruji ho na konkrétních příkladech.

## 12.1 Stávající workflow na projektu Hodnocení pracovníků

V této sekci mapuji stávající stav workflow na projektu Hodnocení pracovníků a snažím se o nalezení případných neefektivit nebo plýtvání. K detailnímu popisu procesu využívám Activity-centric Value Stream Mapping popsaný v sekci 7.1 o mapování stávajícího stavu, při adopci DevOps principů. Následně v podsekci 12.1.2 pomocí tohoto postupu konkrétní neefektivity a plýtvání vyhodnocuji.

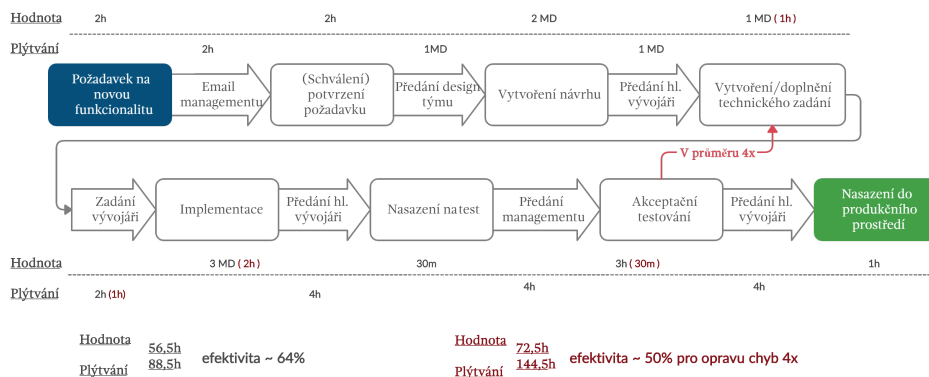
### 12.1.1 VSM

V této podsekci popisuji proces dodávání software na projektu Hodnocení pracovníků pomocí zjednodušeného vývojového diagramu, kde jsou znázorněny jednotlivé aktivity v procesu, které vedou od změnového požadavku až k dodání nějaké změny na projektu ke klientovi. První diagram 12.1 znázorňuje VSM pro jednoduchý změnový požadavek. Oprava složitější produkční chyby by se prováděla obdobně. Struktura a časové hodnoty uvedené v diagramech byly konzultovány s projektovým manažerem.




**Obrázek 12.1:** VSM pro proces - jednoduchý změnový požadavek, zdroj: Autor

Druhý diagram 12.2 znázorňuje jednotlivé aktivity při vytváření nové funkcionality podle požadavků od klienta. Červeně jsou v diagramu označeny hodnoty pro opakující se aktivity (například v případě objevené chyby při akceptačním testování).




**Obrázek 12.2:** VSM pro proces - nová funkcionalita, zdroj: Autor

### 12.1.2 Vyhodnocení neefektivit a plýtvání

Při vyhodnocení neefektivity a plýtvání bylo nutné zvážit specifika prostředí, ve kterém vzniká. Těmito omezeními jsou hlavně specifické časové nároky jednotlivých členů týmu, který je složen převážně ze stážistů. Proto bylo nutné brát v potaz rozdílné trvání jednotlivých aktivit a přechodů, které se liší podle aktuálního období. Například v období letních prázdnin tým pracuje v podstatě na plný úvazek a přes rok se časové nároky liší podle vytíženosti jinými povinnostmi. Dále bylo nutné brát v potaz to, že někteří členové týmu často pracují vzdáleně.

Efektivita procesů se pohybuje přibližně od 40-65%. U druhého diagramu 12.2 efektivita výrazně klesá, pokud dojde k odhalení chyby při akceptačním

testování a celá část procesu se musí opakovat. Proto jsem se právě na tuto část zaměřil. Aktivitu pro nasazení do testovacího prostředí, kdy nejprve vývojář kontaktuje hlavního vývojáře jsem označil za přebytečný krok. Dlouhé čekání na zpětnou vazbu od managementu projektu je také velmi neefektivní.

### 12.1.3 Nastavení metrik

V případě prostředí, ve kterém vzniká projekt Hodnocení pracovníků je velice důležitá vysoká spolehlivost vyvíjených systémů a rychlost reakce na změny. Jako vhodné metriky pro sledování jsem tedy zvolil čas potřebný k dodání funkční změny nebo opravy ke klientovi. Vzhledem k vyšším nárokům na stabilitu řešení jsem jako další vhodnou metriku zvolil celkové pokrytí regresním testováním. Tyto metriky budou po aplikaci nového workflow sledovány managementem projektu.

## 12.2 Nové workflow na projektu Hodnocení pracovníků

V této sekci uvádím prerekvizity, které musejí být splněny ke správné adopci nového workflow. Při určení prerekvizit jsem vycházel z principu Continuous Integration, který je popsán v sekci 6.4. Dále v této sekci uvádím obecný návrh nového workflow, které vychází z dalších principů DevOps, které jsou popsány v kapitole 6 a následně v kapitole 7 o samotné adopci DevOps principů. Při návrhu nového workflow využívám delivery pipeline, která je blíže popsána v sekci 6.6. Na závěr kapitoly uvádím konkrétní příklady nového workflow.

### 12.2.1 Prerekvizity

Následující prerekvizity jsem převzal ze sekce 6.4 o principech Continuous Integration:

1. Sdílený VCS - Tento bod je splněn. Projekt byl verzován v systému Gitlab již před začátkem implementace této práce.
2. Automatický build - Tento bod byl také splněn před začátkem implementace této práce. Projekt využívá build systém Maven, který je blíže popsán v podsekci 8.2.2 o build systémech.
3. Pokrytí automatickými testy - Tento bod byl splněn. Implementace automatických testů je blíže popsána v sekci 10.3.
4. Pravidelný commit změn na hlavní větev - Tento bod bude představen týmu pracujícímu na projektu Hodnocení pracovníků v rámci nového workflow.
5. Dostatečně rychlé testy a build projektu - Tento bod je také splněn. Provedení buildu a spuštění integračních a jednotkových testů v prostředí Gitlab CI/CD trvá vždy mezi deseti a patnácti minutami.

6. Transparentnost procesu - Vzhledem k využívání systému Gitlab pro všechny fáze procesu je umožněn snadný přístup ke stavu projektu.
7. Automatické nasazení do testovacího prostředí - Tento bod je splněn. Automatické nasazení jsem nastavil v rámci přípravy prostředí, blíže popsané v podsekci 10.1.1.

Splněním těchto prerekvizit je možné začít s návrhem a implementací nového workflow na projektu Hodnocení pracovníků.

### 12.2.2 Obecný popis

K začlenění testování, principů CI/CD a redukci identifikaného plýtvání a neefektivity v podsekci 12.1.2 jsem určil obecné úpravy workflow. V podsekci 12.2.3 jsou následně uvedeny konkrétní příklady.

#### Branching model

Pro zefektivnění nasazení do testovacího prostředí a zrychlení zpětné vazby při akceptačním testování projektu jsem se rozhodl využít změny branching modelu ve sdíleném Git repozitáři, principu Continuous Delivery a připraveného automatického nasazení do testovacího prostředí, které je blíže popsané v podsekci 10.1.1.

V novém branching modelu budou jednotlivé větve rozděleny následovně:

- `master` - Hlavní větev, která obsahuje verzi projektu nasazenou v produkčním prostředí. Obsahuje tagy verzí.
- `test` - Vývojová větev. Členové týmu do ní mergují změny z `feature` větví. Umožňuje nasadit pomocí manuálního úkolu v pipeline do testovacího prostředí.
- `feature_{název vyvíjené funkcionality}` - Pracovní větev. Vývojáři na nich zpracovávají vyvíjené změny na projektu. Průběžně na ně commitují svou práci. Testeři na nich provádějí jednotkové a integrační testování změn. V případě opravy chyby může být označena i jako `fix_{stručný popis chyby}`.
- `hotfix_{hotfix verze}` - Tato větev slouží pro opravu chyby v produkčním kódu. Odvíjí se vždy od větve `master`.

Tímto způsobem se umožní testerů a vývojářům průběžně nasazovat na testovací prostředí, bez nutnosti kontaktovat vedoucího vývojáře. Dále se průběžných mergováním změn na vývojovou větev zrychlý zpětná vazba od testerů, případně od managementu projektu, který bude provádět akceptační testování změn.

### Continuous Delivery

Vzhledem k vyšší úrovni regrese jsem zvolil přístup popsaný v sekci 7.2, kdy se nejprve začne s automatickým nasazením do testovacího prostředí, dokud není vytvořena dostatečná úroveň automatického testování vhodná pro produkční prostředí. Tento přístup je vhodný pro demonstraci DevOps principů týmu pracujícímu na projektu a stakeholderům. Po vytvoření obsáhlejšího testovacího pokrytí bude však vhodné rozšířit projekt i o automatické nasazení do produkčního prostředí, aby se tým nepřipravoval o hlavní výhodu Continuous Delivery, kterou je rychlá zpětná vazba.

### Code reviews

K zajištění vyšší kvality výstupů a omezení zanešených chyb do produkčního kódu jsem se rozhodl do workflow začlenit code reviews formou merge requestů v systému Gitlab.

Pro každou větší změnu vytvoří vývojář merge request pracovní větve, na které danou funkcionalitu vyvíjí (do větve `test`). K merge requestu přidá krátký popis provedených změn a přiřadí k němu některého z vedoucích vývojářů. Pokud chce vyvíjenou funkcionalitu konzultovat v průběhu práce, přidá do názvu „WIP: ", což je označení pro merge request, který je rozpracovaný a není připravený na merge do vývojové větve. U tohoto merge requestu může poté s ostatními členy týmu diskutovat nad prováděnými změnami. Určení toho, pro jaké změny je potřebné vytvářet merge request, záleží na domluvě členů týmu. Může se lišit podle schopností daného vývojáře nebo složitosti prováděných změn.

### Testování

Do workflow se testování bude začleňovat podle obecné testovací strategie uvedené v sekci 9.3 a podle testovacího plánu, který je specifický pro daný projekt. Základní obecné zásady týkající se workflow a testování na projektu Hodnocení pracovníků, které jsem pro nové workflow stanovil jsou následující:

- Vývojáři sami píšou jednotkové testy. Snaží se o co největší možné pokrytí.
- Testeři implementují integrační testy a vykonávají explorační testování.
- Pokud selžou automatické testy na vývojové větvi nebo na větvi `master`, stává se jejich opravení, případně opravení chyby která způsobila selhání testů, hlavní prioritou celého týmu.
- Testeři se zapojují už při návrhu změn a snaží se o odhalení chyb ve specifikaci.
- Testeři průběžně reportují stav testování managementu projektu, například pomocí dokumentů popsaných v testovací strategii 9.3.

- Testeři reportují chyby do Gitlab Issues. V případě chyb na pracovních větvích je možné chyby komunikovat s vývojáři přímo, bez reportování do Gitlab Issues.
- Vývojáři odkazují na opravy chyb pomocí commit message přidáním „ref #{číslo issue}" na konec commit message.

### 12.2.3 Konkrétní příklady

V této podsekci uvádím příklady nového workflow pro projekt Hodnocení pracovníků. Jedná se o vybrané příklady, se kterými se lze v procesu dodávání software běžně setkat.

#### Oprava chyby

V případě opravy chyby na pracovní větvi se oprava vypracovává přímo na ní a není nutné vytvářet novou větev. V případě opravy chyby na vývojové větvi je nutné vytvořit novou pracovní větev s názvem začínajícím `fix_` a končícím vhodným označením chyby.

Po vyhotovení opravy chyby na pracovní větvi je potřeba provést merge zpět do větve vývojové. Pokud vývojář, který provádí opravu, nebyl domluvený s vedoucím vývojářem jinak, tak nejprve vytvoří merge request s popisem provedených změn a přiřadí k němu vedoucího vývojáře. Až po schválení vedoucím vývojářem je možné provést merge do vývojové větve.

#### Oprava závažné produkční chyby

V případě opravy závažné produkční chyby se z větve `master` vytvoří nová větev začínající názvem `hotfix_` a končící verzí opravy. Pokud opravu chyby provádí vývojář, vždy je nutné opravu přispívat na větev `master` pomocí merge requestu a přiřadit k němu vedoucího vývojáře.

#### Selhávající automatické testy

Selhávající testy na větvi `master` nebo `test` se automaticky stávají prioritou celého týmu, který musí zajisti co nejrychlejší opravu. V případě větve `master` se oprava provádí stejně jako při opravě závažné produkční chyby. V případě větve `test` se oprava po důkladném prozkoumání příčiny selhávajících testů provede na nové pracovní větvi a postupuje se stejně jako u opravy běžné chyby.

#### Vývoj nové funkcionality

Vývoj nové funkcionality se provádí na nové pracovní větvi s názvem začínajícím na `feature_` a končící vhodný názvem vyvíjené funkcionality. Pokud není stanoveno vedoucím vývojářem jinak, je potřeba vytvořit při dokončení práce merge request. U větších celků je vhodné vytvořit merge request už v průběhu práce a konzultovat provedenou práci s členy týmu. Tento merge

request, který ještě není připravený pro merge do vývojové větve musí mít název začínající na „WIP: ".

### Vydání nové verze

Vydání nové verze se provádí průběžně pro každou novou funkcionalitu tak, jak je předem domluveno s managementem projektu. Pro vydání nové verze se provede merge vývojové větve do větve `master`, a to po naplnění akceptačních kritérií stanovených managementem projektu. Po provedení merge do `master` větve je potřeba vytvořit na `master` větvi nový tag s názvem začínajícím na `stable_` a končícím na číslo verze.

### Reportování chyby

Reportování objevené chyby se provádí podle testovací strategie 9.3 do systému Gitlab Issues.

## 12.3 Shrnutí kapitoly

V rámci této kapitoly jsem nejprve provedl mapování stávajícího stavu workflow pomocí metody Value Stream Mapping. Následně jsem provedl vyhodnocení výstupů této metody. Podle identifikovaných neefektivit a plýtvání v procesu dodávání software, jsem vytvořil návrh nového workflow. Toto nové workflow jsem obecně popsal a následně jsem uvedl konkrétní příklady.

# Kapitola 13

# Závěr

V rámci této práce vznikla rešerše zabývající se testování software, konkrétně tím, jak se testování začleňuje do jednotlivých typů životního cyklu software, druhy a úrovněmi testů a procesy v testování software. Dále byla vypracována rešerše konceptu DevOps a jeho principů, jako jsou například Continuous Integration, Delivery nebo další pomocné postupy. Obsahem rešerše byl také popis rizik, které s sebou tyto moderní principy přináší, a jaké výhody naopak vyvažují tato rizika. V poslední kapitole teoretické části vznikla také rešerše nástrojů pro testování a implementaci DevOps principů pro projekty v jazyce Java EE.

V praktické části byla vytvořena testovací strategie, kterou je možné aplikovat obecně na projekty, které vznikají ve stejném prostředí jako projekt Hodnocení pracovníků. Dále byl navržen testovací plán, který je specifický pro projekt Hodnocení pracovníků a je odvozen od obecné testovací strategie. V rámci testovacího plánu vznikly dokumenty, jako je testovací matice. Tyto dokumenty mohou sloužit jako aktuální dokumentace projektu. Dle testovacího plánu byly navrženy a implementovány testy na více úrovních a za pomocí principů Continuous Integration byly některé z těchto testů automatizovány. V rámci implementace testů, vzniklo pokrytí základní funkcionality zabraňující možné regresi. Testování na projektu bylo následně vyhodnoceno podle metrik stanovených v testovacím plánu.

Aby bylo možné využívat přínosů testování a DevOps principů, bylo analyzováno aktuální workflow na projektu a dle odhalených neefektivit a plýtvání v procesu dodávání software, bylo navrženo nové workflow. Toto nové workflow, společně s testovací strategií, v sobě zahrnuje procesy správy testů a způsob využívaní principů DevOps, například pomocí delivery pipeline, která byla také v rámci praktické části implementována pomocí nástroje Gitlab CI/CD.

Výhodou řešení workflow a obecné testovací strategie je to, že je možné tyto postupy aplikovat na podobné projekty v daném prostředí. Hlavní výhodou řešení však je, že umožňuje týmu pracujícím na projektu, aby mohl dodávat výstupy v kratších intervalech a ve vyšší kvalitě.

Na práci je možné navázat, jak rozšířením testování na projektu podle postupů stanovených v testovací strategii, tak i rozšiřováním workflow, například o testování bezpečnosti daného řešení pomocí principů takzvaného DevSecOps, který rozšiřuje koncept DevOps o prvky bezpečnosti.

Přílohy

# Příloha A

## Literatura a zdroje


1. PATTON, Ron. *Testování softwaru*. 1. vydání. Praha: Computer Press, 2002. ISBN 80-722-6636-5.
2. *ISTQB Glossary* [online] [cit. 2019-11-03]. Dostupné z: https://glossary.istqb.org/.
3. BLACK, Rex; COLEMAN, Gerry. *Agile Testing Foundations : An ISTQB Foundation Level Agile Tester Guide* [online]. 1st Edition. Swindon, UK: BCS, The Chartered Institute for IT, 2017 [cit. 2019-12-16]. ISBN 978-1-78017-33-75. Dostupné z: http://search.ebscohost.com.ezproxy.techlib.cz/login.aspx?direct=true&db=e000xww&AN=1497465&lang=cs&site=ehost-live.
4. *ISTQB's® Syllabi* [online] [cit. 2019-11-03]. Dostupné z: https://www.istqb.org/downloads/syllabi.html.
5. LEAU, Yu Beng; LOO, Wooi Khong; THAM, Wai Yip; TAN, Soo Fun. International Conference on Information and Network Technology. 2012, roč. 37, č. 1, s. 162–167.
6. KNEUPER, Ralf. Sixty Years of Software Development Life Cycle Models. *IEEE Annals of the History of Computing* [online]. 2017, roč. 39, č. 3, s. 41–54 [cit. 2019-11-10]. ISSN 1058-6180. Dostupné z DOI: 10.1109/MAHC.2017.3481346.
7. FOWLER, Martin; HIGHSMITH, Jim. The Agile Manifesto. 2000, roč. 9.
8. BLACK, Rex; MITCHELL, Jamie L. *Advanced software testing*. 1st Edition. Santa Barbara, CA: Rocky Nook, 2011. ISBN 978-1-933952-39-0.
9. HAMBLING, B.; MORGAN, P.; SAMAROO, A.; THOMPSON, G.; WILLIAMS, P. *Software testing : An istqb-bcs certified tester foundation guide - 4th edition*. 4th edition. BCS Learning & Development Limited, 2019. ISBN 9781780174921.
10. *4. Test Design Techniques* [online] [cit. 2020-05-14]. Dostupné z: http://istqbfoundation.wikidot.com/4.

11. TALWAR, Richa. *Structure Based or Whitebox Testing Techniques* [online]. 2020 [cit. 2020-05-14]. Dostupné z: `https://www.hcltech.com/blogs/structure-based-or-whitebox-testing-techniques`.
12. CRISPIN, Lisa; GREGORY, Janet. *Agile testing: a practical guide for testers and agile teams*. 1st Edition. Upper Saddle River, NJ: Addison-Wesley, c2009. ISBN 978-0-321-53446-0.
13. CRISPIN, Lisa. *Using the Agile Testing Quadrants* [online] [cit. 2019-12-28]. Dostupné z: `https://lisacrispin.com/2011/11/08/using-the-agile-testing-quadrants/`.
14. MARICK, Brian. My Agile testing project [online]. 2003, s. 2 [cit. 2019-12-29]. Dostupné z: `http://www.exampler.com/old-blog/2003/08/21.1.html%5C#agile-testing-project-1`.
15. POPPENDIECK, Mary; POPPENDIECK, Thomas David. *Implementing lean software development: from concept to cash*. Upper Saddle River, NJ: Addison-Wesley, c2007. ISBN 978-0321437389.
16. *What are Business Drivers?* [online]. © CFI Education Inc., 2015-2020 [cit. 2020-03-01]. Dostupné z: `https://corporatefinanceinstitute.com/resources/knowledge/modeling/business-drivers/`.
17. ALLSPAW, John; HAMMOND, Paul. *10+ Deploys Per Day: Dev and Ops Cooperation* [online]. 2009 [cit. 2020-02-21]. Dostupné z: `https://www.slideshare.net/jallspaw/10-deploys-per-day-dev-and-ops-cooperation-at-flickr`.
18. SHARMA, Sanjeev. *The DevOps Adoption Playbook : A Guide to Adopting DevOps in a Multi-Speed IT Enterprise*. 1st. John Wiley & Sons, Incorporated, Somerset, 2017. ISBN 9781119310525.
19. EBERT, C.; GALLARDO, G.; HERNANTES, J.; SERRANO, N. *"DevOps,"in IEEE Software* [online]. 2016 [cit. 2020-02-21]. Č. vol.33. ISSN 1937-4194.
20. JEZ, Humble; CHRIS, Read; DAN, North. *Proceedings of the Conference on AGILE 2006: The Deployment Production Line* [online]. IEEE Computer Society, 2006 [cit. 2020-02-22]. Č. AGILE '06. Dostupné z: `https://doi.org/10.1109/AGILE.2006.53`.
21. BECK, Kent; ANDRES, Cynthia. *Extreme programming explained: embrace change*. 2nd ed. Boston: Addison-Wesley, 2005. ISBN 03-212-7865-8.
22. DUVALL, Paul M.; MATYAS, Steve; GLOVER, Andrew. *Continuous integration: improving software quality and reducing risk*. Upper Saddle River: Addison-Wesley, c2007. ISBN 978-032-1336-385.
23. HUMBLE, Jez; FARLEY, David. *Continuous delivery : reliable software releases through build, test, and deployment automation*. 1st. United States, Crawfordsville, Indiana.: Pearson Education, Inc, 2011. ISBN 978-0-321-60191-9.

24. FITZ, Timothy. *Timothyfitz.com* [online]. 2009-2012 [cit. 2020-02-21]. Dostupné z: `http://timothyfitz.com/blog`.

25. *DevOps in practice: A multiple case study of five companies* [online]. 2019 [cit. 2020-02-21]. Č. 114. ISSN 0950-5849. Dostupné z: `https://researchportal.helsinki.fi/en/publications/devops-in-practice-a-multiple-case-study-of-five-companies`.

26. POPPENDIECK, Mary. *Value Stream Mapping: Finding the Constraint* [online]. Poppendieck.LCC, 2008 [cit. 2020-05-14]. Dostupné z: `http://agiles2008.agiles.org/common/pdfs/Poppendieck%5C%20-%5C%20Value%5C%20Stream%5C%20Mapping.pdf`.

27. FITZ, Timothy. *Getting Started with Continuous Deployment* [online]. 2009-2012 [cit. 2020-03-16]. Dostupné z: `http://timothyfitz.com/2012/11/25/paths-to-continuous-deployment/`.

28. *Junit.org* [online]. 2020 [cit. 2020-04-20]. Dostupné z: `junit.org`.

29. GARCÍA, Boni. *Mastering Software Testing with JUnit 5*. © Packt Publishing Limited, October 2017. ISBN 9781787285736.

30. *TestNG docs* [online]. 2019 [cit. 2020-04-20]. Dostupné z: `https://testng.org/doc/`.

31. *Site.mockito.org/* [online]. 2020 [cit. 2020-04-20]. Dostupné z: `https://site.mockito.org/`.

32. *Easymock.org/* [online]. 2020 [cit. 2020-04-20]. Dostupné z: `https://easymock.org/`.

33. *Github.com/powermock/powermock* [online]. 2020 [cit. 2020-04-20]. Dostupné z: `https://github.com/powermock/powermock`.

34. *Assertj.github.io* [online]. 2020 [cit. 2020-04-20]. Dostupné z: `https://assertj.github.io/doc/`.

35. *Arquillian.org* [online]. Red Hat, Inc., 2009-2018 [cit. 2020-04-20]. Dostupné z: `arquillian.org`.

36. AMENT, John D. *Arquillian Testing Guide*. Paperback. Packt Publishing, April 17, 2013. ISBN 9781782160700.

37. GUNDECHA, Unmesh; COCCHIARO, Carl. *Learn Selenium: Build data-driven test frameworks for mobile and web applications with Selenium Web Driver 3*. Paperback. Packt Publishing, July 18, 2019. ISBN 978-1838983048.

38. *Selenium.dev/* [online]. 2020 [cit. 2020-04-20]. Dostupné z: `https://www.selenium.dev/`.

39. GEISSHIRT, Kenneth; ZATTIN, Emanuele; VOSS, Rasmus. *Git Version Control Cookbook: Leverage version control to transform your development workflow and boost productivity, 2nd Edition*. 2nd edition. Packt Publishing, 2018. ISBN 9781789137545.

40. *Git vs. SVN – What Is The Difference?: GIT AT SCALE* [online]. Perforce Software, Inc., 2018 [cit. 2020-04-20]. Dostupné z: https://www.perforce.com/blog/vcs/git-vs-svn-what-difference.
41. *Docs.gitlab.com/* [online]. 2020 [cit. 2020-04-20]. Dostupné z: https://docs.gitlab.com/.
42. *What's in a Build Tool?* [online]. b. r. [Cit. 2020-04-21]. Dostupné z: https://www.lihaoyi.com/post/WhatsinaBuildTool.html.
43. *Getting Started With Java Build Tools* [online]. 2020 [cit. 2020-04-21]. Dostupné z: https://www.jrebel.com/blog/getting-started-with-java-build-tools.
44. *Few points on Java Build Tools: Ant vs Maven vs Gradle* [online]. 2017 [cit. 2020-04-21]. Dostupné z: https://medium.com/@kapil.sharma91812/few-points-on-java-build-tools-ant-vs-maven-vs-gradle-e149a43325b8.
45. FARCIC, Viktor. *Java Build Tools: Ant vs Maven vs Gradle* [online]. 2014 [cit. 2020-04-21]. Dostupné z: https://technologyconversations.com/2014/06/18/build-tools/.
46. *Maven.apache.org* [online]. 2020 [cit. 2020-04-21]. Dostupné z: https://maven.apache.org/what-is-maven.html.
47. *Gradle.org* [online]. 2020 [cit. 2020-04-21]. Dostupné z: https://gradle.org/.
48. *Jenkins.io* [online]. 2020 [cit. 2020-04-21]. Dostupné z: https://jenkins.io/.
49. *Jenkins vs. GitLab* [online]. 2020 [cit. 2020-04-21]. Dostupné z: https://about.gitlab.com/devops-tools/jenkins-vs-gitlab.html%5C#overview.

# Příloha B

# Seznam zkratek

CD Continuous Delivery

CDI Contexts and Dependency Injection for Java

CI Continuous Integration

DSL Domain Specific Language

EJB Enterprise JavaBeans

IaC Infrastructure as code

ISTQB International Software Testing Qualifications Board

JDK Java Development Kit

JPA Java Persistence API

JSF JavaServer Faces

JTA Java Transaction API

KPI Key performance indicator

OAT Operations Acceptance Testing

QA Quality assurance

SDLC Software development lifecycle

UAT User Acceptance Testing

VCS Version Control System

VSM Value Stream Mapping

# Příloha C

# Testovací matice

| Funkcionalita | | | | Analýza rizik | | | Druhy a úrovně testování | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | *Obecný název komponenty, může se jednat o celek jako je jeden proces nebo některá ze sekcí aplikace.* | *Podskupina/část může popisovat konkrétní funkcionalitu nebo může komponentu dělit na další podskupiny, které budou detailněji popsány pomocí user stories.* | *U funkcionality, která je identifikována jako kritická, uvést detailní user story.* | *Určete pravděpodobný dopad a pravděpodobnost vzniku chyby. Na škále od 1 do 5ti, 1 je nejmenší a 5 nejvyšší.* | | | *Q1* | *Q2* | *Q3* | | |
| **Verze** | **Feature / Komponenta** | **Podskupina / část** | **User story** | **Dopad** | **Pravděpodobnost** | **Riziko** | Unit | IT | exploratory | Stav testování | |
| *Testing kickoff* | Evaluace | Požadavek na osobní hodnocení | Jako vedoucí katedry chci vytvořit požadavek na osobní hodnocení pro konkrétního zaměstnance mojí katedry pro dané období hodnocení, tak aby mohl vyplnit osobní hodnocení. Ze sekce moje katedra. | 4 | 2 | 8 | 1 | 1 | 1 | Není hotovo | |
| | | | Jako vedoucí katedry chci odstranit požadavek na osobní hodnocení, včetně všech již existujících hodnocení, která jsou s ním spojená. Ze sekce moje katedra. | 4 | 2 | 8 | 1 | 1 | 1 | Je hotovo | 1 |
| | | Osobní hodnocení | Jako uživatel chci zobrazit přehled osobních hodnocení aktuálně přihlášeného uživatel. Osobní hodnocení za aktuální a všechna předchozí období. | 3 | 3 | 9 | 1 | 1 | 1 | Nelze / nebude se dělat | 0 |
| | | | Jako uživatel chci zobrazit stav osobního hodnocení v sekci osobní hodnocení. | 1 | 4 | 4 | 1 | 0 | 1 | | |
| | | | Jako zaměstnanec chci vytvářet osobní hodnocení, aby se k němu nadřízený a vedoucí katedry mohli vyjádřit. | 5 | 4 | 20 | 1 | 1 | 1 | | |
| | | | Jako zaměstnanec chci vytvářet osobní hodnocení, s předvyplněnými daty z minulých let (podle šablony). | 3 | 4 | 12 | 1 | 1 | 1 | | |
| | | | Jako zaměstnanec chci uložit nedokončené osobní hodnocení, aby jsem se k němu mohl později vrátit. | 3 | 4 | 12 | 1 | 1 | 1 | | |
| | | | Jako zaměstnanec chci aby se evaluace automaticky ukládala po daném intervalu, aby jsem nepřišel o rozpracovanou práci. | 3 | 2 | 6 | 1 | 1 | 1 | | |
| | | | Jako vedoucí katedry chci upravit pracovní pozici a úvazek dokončeného osobního hodnocení zaměstnance. | 2 | 2 | 4 | 0 | 0 | 1 | | |
| | | | Jako vedoucí katedry chci vrátit dokončené osobní hodnocení zpět zaměstnanci k úpravám, aby zaměstnanec mohl opravit chyby nebo doplnit informace. | 4 | 3 | 12 | 1 | 1 | 1 | | |
| | | | Jako nadřízený chci vrátit dokončené osobní hodnocení zpět zaměstnanci k úpravám, aby zaměstnanec mohl opravit chyby nebo doplnit informace. | 4 | 3 | 12 | 1 | 1 | 1 | | |
| | | Hodnocení nadřízeného | Jako nadřízený chci přidat hodnocení nadřízeného k dokončenému osobnímu hodnocení zaměstnance. | 5 | 4 | 20 | 1 | 1 | 1 | | |

**Obrázek C.1:** Tabulka testovací matice, zdroj: Autor

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | Jako nadřízený chci uložit nedokončené hodnocení nadřízeného, aby jsem se k němu mohl později vrátit. | 3 | 3 | 9 | 1 | 0 | 1 |
| | | Jako vedoucí katedry chci delegovat hodnocení nadřízeného na některého ze zaměstnanců katedry, tak aby mohl vyplnit hodnocení nadřízeného k dokončenému osobnímu hodnocení. | 5 | 3 | 15 | 1 | 1 | 1 |
| | Hodnocení vedoucího katedry | Jako vedoucí katedry chci vyplnit hodnocení vedoucího katedry k dokončenému osobnímu hodnocení. | 5 | 4 | 20 | 1 | 1 | 1 |
| | | Jako vedoucí katedry chci uložit nedokončené hodnocení vedoucího katedry, aby jsem se k němu mohl později vrátit. | 3 | 3 | 9 | 1 | 0 | 1 |
| | Vyjádření zaměstnance k hodnocení | Jako zaměstnanec chci přidat vyjádření zaměstnance k dokončenému hodnocení vedoucího a hodnocení nadřízeného. | 5 | 4 | 20 | 1 | 1 | 1 |
| | | Jako zaměstnanec chci uložit nedokončené vyjádření zaměstnance k dokončenému hodnocení vedoucího a hodnocení nadřízeného. | 3 | 3 | 9 | 1 | 0 | 1 |
| | Obecné | Jako uživatel chci zobrazit detail hodnocení, které se mě týká (vytvořil jsem, byl jsem k němu delegován - jsem nadřízený, jsem vedoucí katedry, jsem člen atestační komise pro danou katedru, jsem administrátor) - jedná se vždy o hodnocení se všemi vyplněnými částmi. | 5 | 4 | 20 | 1 | 1 | 1 |
| | | Jako uživatel chci zobrazit detail hodnocení z přechozích let, přechod z sekce osobní hodnocení nebo z detailu hodnocení, které se mě týká (vytvořil jsem, byl jsem k němu delegován - jsem nadřízený, jsem vedoucí katedry, jsem člen atestační komise pro danou katedru, jsem administrátor) - jedná se vždy o hodnocení se všemi vyplněnými částmi. | 2 | 3 | 6 | 1 | 0 | 1 |
| | | Jako uživatel chci uložit hodnocení do pdf, které se mě týká (vytvořil jsem, byl jsem k němu delegován - jsem nadřízený, jsem vedoucí katedry, jsem člen atestační komise pro danou katedru, jsem administrátor) - jedná se vždy o hodnocení se všemi vyplněnými částmi. | 4 | 4 | 16 | 0 | 0 | 1 |
| Administrační strana aplikace (jen Administrátor) | Správa atestačních komisí | Jako administrátor chci vytvářet atestační komise sestávající z atestované katedry a členů pro dané období. | 3 | 3 | 9 | 1 | 1 | 1 |
| | | Jako administrátor chci upravovat (přidávat a odebírat katedry a členy) atestační komise sestávající z atestované katedry a členů pro dané období. | 3 | 2 | 6 | 1 | 1 | 1 |
| | | Jako administrátor chci smazat atestační komisi. | 4 | 3 | 12 | 1 | 1 | 1 |
| | Správa ročních plánů | Jako administrátor chci vytvářet roční plány hodnocení. Obsahující termín notifikace. | 3 | 3 | 9 | 1 | 0 | 1 |
| | | Jako administrátor chci povolit/zakázat zápis pro roční plán. | 3 | 5 | 15 | 1 | 0 | 1 |
| | | Jako administrátor chci změnit termín pro notifikace pro roční plán. | 2 | 2 | 4 | 1 | 0 | 1 |
| | | Jako administrátor chci synchronizovat roční plán s udb. | 4 | 5 | 20 | 0 | 0 | 0 |
| | | Jako administrátor chci smazat roční plán hodnocení. | 4 | 3 | 12 | 1 | 0 | 1 |

**Obrázek C.2:** Tabulka testovací matice, zdroj: Autor

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | Přehled hodnocení (pozor jedná se o sekci přehled hodnocení ne o sekci moje katedra nebo atestace) | Jako administrátor chci zobrazit přehled hodnocení pro celou fakultu, rozdělené podle jednotlivých fakult s poměrem dokončené/ "jiný stav" pro každou katedru. | 1 | 3 | 3 | 0 | 0 | 0 |
| | | Jako administrátor chci zobrazit přehled hodnocení pro jednotlivé katedry. | 1 | 3 | 3 | 0 | 0 | 0 |
| | | Jako administrátor chci odstranit hodnocení z přehledu hodnocení pro individuální katedru. | 1 | 3 | 3 | 1 | 1 | 0 |
| | | Jako administrátor chci vytvořit nový požadavek na hodnocení z přehledu hodnocení pro jednotlivé katedry. | 1 | 3 | 3 | 1 | 1 | 0 |
| | | Jako administrátor chci změnit požadavek na osobní hodnocení z přehledu hodnocení pro jednotlivé katedry. | 1 | 3 | 3 | 1 | 1 | 0 |
| | Odesílání emailů | | 1 | 1 | 1 | 0 | 0 | 0 |
| | Impersonace | | 1 | 1 | 1 | 0 | 0 | 1 |
| | Import hodnocení - historická funkce, nemá už smysl | | 0 | 0 | 0 | 0 | 0 | 0 |
| sekce Moje katedra | | Jako vedoucí katedry (a administrátor) chci zobrazit seznam zaměstnanců katedry a stav jejich evaluace. | 4 | 3 | 12 | 1 | 1 | 1 |
| | | Jako vedoucí katedry (a administrátor) chci změnit období pro zobrazované evaluace. | 2 | 2 | 4 | 0 | 1 | 1 |
| | | Jako vedoucí katedry (a administrátor) chci vidět statistiku stavů evaluací. | 1 | 1 | 1 | 0 | 0 | 0 |
| | | Jako vedoucí katedry (a administrátor) chci filtrovat evaluace. | 2 | 3 | 6 | 1 | 1 | 1 |
| | | Jako vedoucí katedry (a administrátor) chci vidět deadline pro dané období. | 4 | 2 | 8 | 0 | 0 | 1 |
| | | Jako nadřízený chci změnit období pro zobrazované evaluace. | 2 | 2 | 4 | 0 | 1 | 1 |
| | | Jako nadřízený chci vidět statistiku stavů evaluací. | 1 | 1 | 1 | 0 | 0 | 0 |
| | | Jako nadřízený chci filtrovat evaluace. | 1 | 1 | 1 | 1 | 0 | 1 |
| | | Jako nadřízený chci vidět evaluace pro která mám vytvořit (nebo jsem vytvořil) hodnocení nadřízeného. | 4 | 3 | 12 | 1 | 1 | 1 |
| | | Jako nadřízený chci vidět deadline pro dané období. | 4 | 2 | 8 | 0 | 0 | 1 |
| | | Jako administrátor chci z této sekce přejít do přehledu hodnocení. | 1 | 2 | 2 | 0 | 0 | 0 |
| sekce Atestace | | Jako člen atestační komise dané katedry chci zobrazit seznam zaměstnanců katedry a stav jejich evaluace. | 5 | 4 | 20 | 1 | 1 | 1 |
| | | Jako člen atestační komise katedry chci vidět statistiku stavů evaluací. | 2 | 2 | 4 | 0 | 0 | 0 |
| | | Jako člen atestační komise chci změnit období pro zobrazované evaluace. | 1 | 2 | 2 | 0 | 1 | 1 |
| | | Jako člen atestační komise (více atestačních komis), chci zvolit seznam hodnocení podle komise. | 4 | 3 | 12 | 0 | 0 | 1 |

**Obrázek C.3:** Tabulka testovací matice, zdroj: Autor

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | Jako člen atestační komise chci vidět deadline pro dané období. | 4 | 3 | 12 | 0 | 0 | 1 | | |
| | Obecné | Uživatelské akce | Jako uživatel se chci odhlásit. | 4 | 1 | 4 | 0 | 0 | 1 | | |
| | | | Jako uživatel chci zobrazit detail svého profilu. | 3 | 2 | 6 | 0 | 0 | 1 | | |
| | | | Jako uživatel chci vypnout emailové notifikace. | 3 | 2 | 6 | 0 | 0 | 0 | | |
| | | Lokalizace | Jako uživatel chci změnit jazyk a chci aby jazyková preference vydržela i do dalších přihlášení. | 4 | 2 | 8 | 0 | 0 | 1 | | |
| | | Návod | Jako uživatel chci být přesměrován na dokument návod po přechodu do sekce návod. | 4 | 2 | 8 | 0 | 0 | 1 | | |
| | | | | | | | **34** | **27** | **45** | | |

**Obrázek C.4:** Tabulka testovací matice, zdroj: Autor

# Příloha D

# Příprava prostředí

Pro přípravu prostředí bylo potřeba nastavit následující závislosti (Maven Dependencies), rozdělené podle jednotlivých modulů:

- společné
  - org.jboss.shrinkwrap.resolver:shrinkwrap-resolver-bom:3.1.4
  - org.jboss.arquillian:arquillian-bom:1.6.0.Final
  - org.junit.jupiter:junit-jupiter:5.4.2
  - org.junit.jupiter:junit-jupiter-params:5.4.2
  - org.junit.platform:junit-platform-launcher:1.4.2
  - org.junit.vintage:junit-vintage-engine:5.4.2
  - org.assertj:assertj-core:3.15.0
  - potřebné pro arquillian
    - org.jboss.shrinkwrap.resolver:shrinkwrap-resolver-impl-maven:3.1.4
    - org.jboss.shrinkwrap.resolver:shrinkwrap-resolver-depchain:3.1.4
    - org.jboss.arquillian.junit:arquillian-junit-container:1.6.0.Final
    - fish.payara.extras:payara-embedded-all:4.1.153
    - org.slf4j:slf4j-simple:1.7.29
    - org.jboss.arquillian.extension:arquillian-persistence-dbunit:1.0.0.Alpha7
    - com.h2database:h2:1.4.200
- eval-ejb
  - potřebné pro arquillian
    - org.jboss.arquillian.container:arquillian-glassfish-embedded-3.1:1.0.2
  - pluginy
    - org.apache.maven.plugins:maven-surefire-plugin:2.22.2
    - org.apache.maven.plugins:maven-failsafe-plugin:2.22.2
- eval-web
  - org.mockito:mockito-core:3.3.3

- org.mockito:mockito-junit-jupiter:3.3.3
- potřebné pro arquillian
  - org.jboss.arquillian.protocol:arquillian-protocol-servlet
  - org.glassfish.main.extras:glassfish-embedded-all:4.1.2

# Příloha E

## Test charters

| Přibližná struktura testu | | |
|---|---|---|
| Prozkoumej . . . <cíl> | Se . . . <zdrojem> | Aby jsi odhalil . . . <informaci> |
| že jako vedoucí katedry mohu zobrazit všechny zaměstnance katedry, včetně jejich hodnocení | s různými ročnímy plány, jak aktivními, tak neaktivními | že se zaměstnanci a stavy jejich hodnocení zobrazují správně |
| že jako vedoucí katedry mohu zobrazit všechny zaměstnance katedry, včetně jejich hodnocení | s různými ročnímy plány, jak aktivními, tak neaktivními | že je možné provádět jen povolené akce pro dané obodobí, pro požadavky hodnocení (vytvářet, mazat..) |
| že jako vedoucí katedry mohu filtrovat hodnocení zaměstnanců | různými jmény, uživatelskými jmény a podle stavu jejich hodnocení | potencionální problémy s filtrací |
| že jako vedoucí katedry vidím správný deadline pro roční plán | s různými ročnímy plány | potencionální problémy se zobrazování deadlinu, například že se neaktulizuje při změně. |
| že jako vedoucí katedry mohu zobrazit detail hodnocení zaměstnance mojí katedry | pro všechny stavy hodnocení *nové (sebehodnocení), hodnocení nadřízeného, hodnocení vedoucího katedry, vyjádření zaměstnance, dokončeno* | potencionální problémy se zobrazováním detailu (nezobrazí se správné údaje, nezobrazí se správné části hodnocení) |
| že jako vedoucí katedry mohu vytvořit hodnocení vedoucího katedry pro zaměstnance, který má hodnocení ve stavu *hodnocení vedoucího katedry* | s různými stavy hodnocení zaměstnanců | že není možné vytvořit hodnocení vedoucího katedry pro hodnocení v jiných stavech než je stav *hodnocení vedoucího katedry* |

| | | |
|---|---|---|
| že jako vedoucí katedry mohu vytvořit hodnocení vedoucího katedry pro zaměstnance, který má hodnocení ve stavu *hodnocení vedoucího katedry* | s různými kombinacemi vstupů | potencionální problémy s formulářem vytváření hodnocení, včetně validace |
| že jako vedoucí katedry mohu vytvořit nový požadavek na hodnocení zaměstnance | pro zaměstnance, který ještě nemá vytvořené hodnocení a který již má vytvořené hodnocení | že nemohu vytvořit duplicitní požadavek na hodnocení a že je možné vytvořit nový validní požadavek na hodnocení |
| Že jako vedoucí katedry mohu odstanit požadavek na hodnocení | s různými stavy hodnocení zaměstnanců | potencionální problémy s odstraněním |
| že jako vedoucí katedry mohu vrátit k přepracování hodnocení nadřízeného | s různými stavy hodnocení zaměstnanců | že je možné vrátit jen hodnocení ve stavu *hodnocení nadřízeného*, a že po navrácení může nadřízený znovu upravit hodnocení a nedošlo ke ztrátě dat a je možné ho znovu uložit. |
| že jako vedoucí katedry mohu provést export hodnocení zaměstnance do PDF | pro všechny stavy hodnocení *nové (sebehodnocení), hodnocení nadřízeného, hodnocení vedoucího katedry, vyjádření zaměstnance, dokončeno* | potencionální problémy s exportem (nezobrazí se správné údaje, nezobrazí se správné části hodnocení, export trvá příliš dlouho) |
| že jako vedoucí katedry mohu delegovat hodnocení zaměstnance na jeho nadřízeného | s různými stavy hodnocení zaměstnanců | že je možné delegovat na nadřízeného a že je možné změnit delegování dokud není hodnocení ve stavu hodnocení nadřízeného. |
| průchody aplikací jako vedoucí katedry | různými kombinacemi průchodů ze stránek (detail hodnocení - s různými stavy hodnocení, moje katedra, osobní hodnocení) | potencionální chyby při přesměrování. |
| Lokalizace jako vedoucí katedry | všechny stránky | potencionální opomenutá místa k překladu |

| | | |
|---|---|---|
| Zvolená lokalizace vydrží do dalšího přihlášení | | |
| jako zaměstnanec mohu vytvářet nové osobní hodnocení | různými kombinacemi vstupů | potencionální problémy s formulářem evaluace, včetně validace |
| jako zaměstnanec mohu vytvářet nové osobní hodnocení podle šablony z přechozích let | více různými šablonami | potencionální problémy s formulářem evaluace, včetně validace, některá data se nenačtou |
| jako zaměstnanec mohu vytvářet odpověď na hodnocení vedoucího katedry | různými stavy hodnocení | že lze odpověď na hodnocení přidat jen pokud je hodnocení ve stavu *dokončeno* |
| jako zaměstnanec mohu exportovat hodnocení do pdf | různými stavy hodnocení | potencionální problémy s exportem, chybějící prvky, špatná pole |
| jako zaměstnanec mohu zobrazit detail hodnocení z přechozích let | více hodnocení | |
| jako zaměstnanec mohu zobrazit detail hodnocení | různými stavy hodnocení | případná chybějící pole, nevyplněné části... |
| jako zaměstnanec mohu uložit rozpracované hodnocení | s kompletně vyplněným formulářem | případná chybějící pole, nevyplněné části po znovunačtení... |
| jako zaměstnanec mohu využít funkce autosave a znovunačíst rozpracované hodnocení | s kompletně vyplněným formulářem | případná chybějící pole, nevyplněné části po znovunačtení.. špatně fungující interval automatického ukládání |
| jako zaměstnanec vidím v osobním hodnocení stav svého hodnocení a hodnocení z přechozích let, včetně možnosti vytvořit nové pro aktuální rok, otevřít detail a exportovat do pdf | s různými stavy hodnocení a více hodnoceními z předchozích let | potencionální problémy se zobrazováním stavu hodnocení, chybějící hodnocení z přechozích let nebo problém s exportem/otevřením detailu z dané sekce |
| Lokalizace jako zaměstnanec | | |
| Průchody aplikací jako zaměstnanec | stránky osobní hodnocení a moje katedra | problémy s přechody/přesměrováváním v aplikaci |

| | | |
|---|---|---|
| Jako člen komise mohu v sekci atestace zobrazovat stavy evaluací zaměstnanců katedry, pro kterou je moje atestační komise přidělena | různá období | problémy se zobrazováním zaměstnanců katedry, nepovolené akce, špatný stav |
| Jako člen komise mohu ze sekce atestace zobrazit detail hodnocení některého ze zaměstnanců | všechny stavy hodnocení | problémy se zobrazením detailu hodnocení |
| Jako člen komise mohu ze sekce atestace změnit jaká hodnocení chci zobrazit, podle komise, které jsem členem | zaměstnancem, který je členem více komisí s různými přiřazenými katedrami | potencionální problémy se zobrazování hodnocení nebo jejich stavů |
| průchody aplikací jako člen komise | | |
| jako nadřízený (zaměstnanec na kterého je delegováno hodnocení jiného zaměstnance) mohu v sekci moje katedra vidět seznam hodnocení, které mám vytvořit nebo jsem vytvořil. | různé stavy hodnocení, více delegovaných hodnocení | potencionální problémy se zobrazováním hodnocení |
| jako nadřízený mohu vytvořit hodnocení nadřízeného pro hodnocení zaměstnance jenž mi byl vedoucím katedry delegován, pro hodnocení ve stavu *hodnocení nadřízeného* | různé kombinace vstupů | potencionální problémy s formulářem vytváření hodnocení, včetně validace |
| jako nadřízený mohu ze sekce moje katedra přejít do detailu hodnocení některé ze zaměstnanců, ke kterému jsem vytvářel hodnocení nadřízeného nebo ho mám vytvořit | různé stavy hodnocení | potencionální problémy se zobrazováním hodnocení, nepovolené akce |

| | | |
|---|---|---|
| jako nadřízený mohu vrátit k opravě hodnocení zaměstnance, jen pokud je ve stavu *hodnocení nadřízeného* | různé stavy hodnocení | že je možné vrátit hodnocení jen ve stavu *hodnocení nadřízeného*, a že zaměstnanec může hodnocení opravit a znovu odeslat. |
| průchody aplikací jako nadřízený | stránky moje katedra, osobní hodnocení, detail hodnocení | potencionální problémy s přesměrováním |
| jako nadřízený mohu provést export hodnocení do pdf. | různé stavy hodnocení, vyplněné kompletní hodnocení | žádná pole nechybí, všechny hodnoty jsou validní, u chybějícíh polí je uveden správný text: „nevyplněno" |
| jako nadřízený mohu uložit nedokončené hodnocení nadřízeného | vyplněné všechny údaje v hodnocení nadřízeného | žádná pole nechybí, všechny hodnoty jsou validní |
| jako nadřízený mohu filtrovat hodnocení podle jména, stavu hodnocení a uživatelského jména ze sekce moje katedra | více hodnocení, lišící se stavem | potencionální problémy s filtrací hodnocení |
| jako zaměstnanec mohu provést export evaluace. | s hotovou evaluací ve stavu dokončeno a všemi vyplněnými poli | export evaluace obsahuje všechny vyplněné informace |
| jako adminsitrátor mohu vytvářet atestační komise | | |
| jako adminsitrátor mohu upravovat atestační komise | více uživatel a kateder (přidání/odebrání) | |
| jako administrátor mohu odstranit atestační komisi | | |
| jako administrátor mohu uzavřít/povolit roční plán | | potencionální problémy s uzvařením/povolením ročního plánu - nedojde k uzavření a uživatelé mohou stále vytvářet hodnocení a požadavky hodnocení a naopak |

**Tabulka E.1:** Test charters, zdroj: Autor

# Příloha F

# Analýza stavů a přechodů

| Stav evaluace / akce | | Vytvoření žádosti hodnocení administrátorem nebo vedoucím katedry | Odstranění žádosti administrátorem nebo vedoucím katedry | Vyplnění sebehodnocení zaměstnancem (je delegován nadřízený) | Vyplnění sebehodnocení zaměstnancem (není delegován nadřízený) | Vyplnění hodnocení nadřízeného | Vyplnění hodnocení vedoucího katedry (je vyplněno hodnocení nadřízeného) | Vyplnění hodnocení vedoucího katedry (není vyplněno hodnocení nadřízeného) | Vrácení k opravě nadřízeným | Vrácení k opravě vedoucím katedry | Vyplnění vyjádření zaměstnance |
|---|---|---|---|---|---|---|---|---|---|---|---|
| NEVYTVOŘENO | S1 | S2 | | | | | | | | | |
| NOVÉ (SEBEHODNOCENÍ) | S2 | | S1 | S3 | S4 | | | | | | |
| HODNOCENÍ NADŘÍZENÉHO | S3 | | S1 | | | S4 | | | S2 | S2 | |
| HODNOCENÍ VEDOUCÍHO KATEDRY | S4 | | S1 | | | | S5 | S3 | | S3 | |
| VYJÁDŘENÍ ZAMĚSTNANCE | S5 | | S1 | | | | | | | | S6 |
| DOKONČENO | S6 | | S1 | | | | | | | | |
| IMPORTOVÁNO | | | | | | | | | | | |

**Obrázek F.1:** Analýza přechodu stavů pro evaluace, zdroj: Autor

# Příloha G

## Odhalené chyby při testování

| Název | Priorita |
|---|---|
| Přehled hodnocení - nelze otevřít detail hodnocení z předchozích období | Střední |
| Přehled hodnocení - lze vytvářet nové požadavky hodnocení pro uzavřené období | Střední |
| Filtrování podle stavu je za určité podmínky nefunkční | Střední |
| Detail hodnocení - zobrazuje se Hodnocení nadřízeného v levém sloupci | Nízká |
| Detail hodnocení - formát datum pohovoru | Nízká |
| Hodnocení vedoucího katedry/nadřízeného - nelze zrušit výběr | Střední |
| Vrácení hodnocení k opravě vedoucím katedry funguje zvláštně v případě že není delegován nadřízený | Střední |
| Export do PDF - místo nevyplněný datum se zobrazí dvě tečky | Nízká |
| Zobrazují se chybové hlášky serveru při chybě (404, 500 ..) | Vysoká |
| Lokalizace - opomenuté překlady | Nízká |
| Výběr možnosti z nápovědy způsobí refresh formuláře evaluace | Nízká |
| Formulář evaluace - chybějící validace | Vysoká |
| Formulář evaluace - neformátované chybové hlášky | Nízká |
| Formulář evaluace - umožňuje zadat text delší než 1000 znaků do pole Úspěchy | Střední |
| Formulář evaluace - neimportuje se kvartil u publikace | Nízká |
| Detail hodnocení - chybí tlačítko pro změnu jazyka | Nízká |
| Sekce Moje katedra a Osobní hodnocení se zobrazují stále | Střední |

| | |
|---|---|
| Atestační komise - návrat z detailu hodnocení | Vysoká |
| Hodnocení nadřízeného/vedoucího katedry - jméno je editovatelné | Nízká |
| Lokalizace exportu do PDF - k dořešení | Vysoká |
| Evaluation_requestBean vytváření požadavku selže pokud není zavolána metoda getEvaluationRequests | Střední |
| Evaluation_requestBean vyhazuje UnsupportedOperationException | Nízká |

**Tabulka G.1:** Objevené chyby - export z Gitlab Issues, zdroj: Autor