Optimalizace, udržování a vývoj zaběhnutých projektů

Mgr. Bc. Šimon Zouvala

Diplomová práce
2022

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
Ústav informatiky a umělé inteligence

Akademický rok: 2021/2022

ZADÁNÍ DIPLOMOVÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Bc. Šimon Zouvala**
Osobní číslo:	**A20198**
Studijní program:	**N0613A140022 Informační technologie**
Specializace:	**Softwarové inženýrství**
Forma studia:	**Kombinovaná**
Téma práce:	**Optimalizace, udržování a vývoj zaběhnutých projektů**
Téma práce anglicky:	**Optimization, Maintenance and Development of Established Projects**

Zásady pro vypracování

1. Vypracujte literární rešerši na dané téma.
2. Popište groovy/java ekosystém.
3. Vypracujte dokumentaci převzatého enginu.
4. Proveďte analýzu dosavadního procesu vývoje.
5. Navrhněte optimalizaci dosavadního vývojového procesu.
6. Proveďte závěr.

Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam doporučené literatury:

1. BARCLAY, Kenneth; SAVAGE, John. *Groovy programming: an introduction for Java developers*. Elsevier, 2010.
2. GUCKENHEIMER, Sam a Juan J. PEREZ. *Efektivní softwarové projekty*. Brno: Zoner Press, 2007, xxx, 255 s. Encyklopedie Zoner Press. ISBN 9788086815626.
3. PRESSMAN, Roger S. a Bruce R. MAXIM. *Software engineering: a practitionerʹs approach*. Ninth edition. New York, NY: McGraw-Hill, [2020], xxx, 671 s. ISBN 978-1-260-54800-6.
4. SOMMERVILLE, Ian. *Software engineering*. Tenth edition. Boston: Pearson, [2016], 810 s. Always learning. ISBN 9781292096131.
5. TSUI, Frank F., Orlando KARAM a Barbara BERNAL. *Essentials of software engineering*. Fourth edition. Burlington, Massachusetts: Jones & Bartlett Learning, [2017], 1 online zdroj. ISBN 9781284106077. Dostupné také z: http://search.ebscohost.com/login.aspx?direct=true&scope=site&db=nlebk&AN=1466575&authtype=ip,shib&custid=s39367
6. SUBRAMANIAM, Venkat. *Programming groovy 2: Dynamic productivity for the Java developer*. Pragmatic Bookshelf, 2013.

Vedoucí diplomové práce: **doc. Ing. Roman Šenkeřík, Ph.D.**
Ústav informatiky a umělé inteligence

Konzultant diplomové práce: **Ing. Peter Janků, Ph.D.**
Ústav informatiky a umělé inteligence

Datum zadání diplomové práce: **3. prosince 2021**
Termín odevzdání diplomové práce: **23. května 2022**

UNIVERZITA TOMÁŠE BATI VE ZLÍNĚ
Fakulta aplikované informatiky
L.S.

doc. Mgr. Milan Adámek, Ph.D. v.r.
děkan

prof. Mgr. Roman Jašek, Ph.D., DBA v.r.
ředitel ústavu

Ve Zlíně dne 24. ledna 2022

Prohlašuji, že

- beru na vědomí, že odevzdáním diplomové práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že diplomové práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk diplomové práce bude uložen v příruční knihovně Fakulty aplikované informatiky. Univerzity Tomáše Bati ve Zlíně;
- byl/a jsem seznámen/a s tím, že na moji diplomovou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má Univerzita Tomáše Bati ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – diplomovou práci nebo poskytnout licenci k jejímu využití jen připouští-li tak licenční smlouva uzavřená mezi mnou a Univerzitou Tomáše Bati ve Zlíně s tím, že vyrovnání případného přiměřeného příspěvku na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše) bude rovněž předmětem této licenční smlouvy;
- beru na vědomí, že pokud bylo k vypracování diplomové práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky diplomové práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem diplomové práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji,

- že jsem na diplomové práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně, dne 19.5.2022 Šimon Zouvala, v. r.

podpis studenta

ABSTRAKT

Diplomová práce se zabývá vývojem a dokumentací převzatého projektu, ke kterému nebyla předána dokumentace k enginu, jenž spravuje důležité části programu za využití programovacích jazyků Java a Groovy. Dále se zabývá vhodností dosavadního procesu vývoje v prostředí, kde se tradičně využívá převážně vodopádový způsob. Nabízí také polemiku na jeho vylepšení pro implementaci v jiných projektech. Celkově práce poskytuje vodítko k vytváření nebo udržování softwaru nejenom po technické stránce, ale také po stránce dokumentační pro případné budoucí použití.

Klíčová slova: Scrum, údržba softwaru, vývoj, dokumentace, Groovy, Java

ABSTRACT

The diploma thesis deals with the development and documentation of the project, to which the documentation for the engine, which manages important parts of the program using the Java and Groovy programming languages, was not submitted. It also deals with the suitability of the current development process in an environment where the mainly waterfall method is traditionally used.It also offers a discussion about its improvements for implementation in other projects. Overall, the work provides guidance for creating or maintaining software not only from a technical point of view, but also from a documentation point of view for possible future use.

Keywords: Scrum, software maintenance, development, documentation, Groovy, Java

Na tomto místě bych chtěl poděkovat vedoucímu mé diplomové práce doc. Ing. Romanu Šenkeříkovi, Ph.D. za veškerou pomoc, trpělivost a cenné rady při tvorbě a sepisování práce. Dále bych chtěl poděkovat konzultantovi panu Ing. Peteru Janků, Ph.D. za technické konzultace a praktické rady. Chtěl bych poděkovat i lidem z firmy inQool a.s., kteří byly zainteresovaní během vytváření práce. Nakonec bych chtěl poděkovat své rodině za veškerou podporu a trpělivost při mém studiu.

OBSAH

ÚVOD ... **9**

I TEORETICKÁ ČÁST ... **10**

1 VÝVOJOVÉ METODY ... **11**

1.1 Vodopádový model ... 11

1.2 Agilní metody ... 13

1.3 Srovnání metodik ... 14

2 SCRUM ... **16**

2.1 Definice ... 16

2.2 Teorie ... 17

2.3 Hodnoty ... 18

2.3.1 Role ... 18

2.4 Scrum tým ... 19

2.4.1 Scrum Master ... 19

2.4.2 Product Owner ... 20

2.4.3 Development team ... 20

2.5 Artefakty ... 20

2.5.1 Product Backlog ... 20

2.5.2 Sprint Backlog ... 21

2.5.3 Inkrement ... 21

2.6 Události ... 22

2.6.1 Sprint ... 22

2.6.2 Sprint Planning ... 22

2.6.3 Daily Scrum ... 23

2.6.4 Sprint Review ... 23

2.6.5 Sprint Retrospective ... 23

2.7 Varianty Scrumu ... 23

2.8 Kombinace s Extrémním programováním ... 24

2.9 Kombinace s Kanbanem ... 25

3 OBECNÉ PRINCIPY ÚDRŽBY A VÝVOJE SOFTWARU ... **26**

3.1 Nestrukturovaný kód ... 27

3.2 Nedostatečná znalost systému nebo aplikační oblasti ... 28

3.3 Nedostatečná dokumentace ... 28

3.4 Špatná pověst softwarové údržby ... 29

II PRAKTICKÁ ČÁST ... **30**
4 EKOSYSTEM GROOVY/JAVA ... **31**
4.1 Odlišnosti jazyků Groovy a Java ... 31
4.2 Closure ... 32
4.3 Binding ... 34
5 ANALÝZA A OPTIMALIZACE VÝVOJOVÉHO PROCESU ... **36**
5.1 Popis vývojového procesu ... 36
5.1.1 Složení členů vývojového týmu ... 36
5.1.2 Používané nástroje ... 37
5.1.3 Nastavení Scrumu ... 39
5.2 Návrh dotazníku ... 42
5.2.1 Nastavení Scrumu ... 42
5.2.2 Komunikace ... 43
5.2.3 Možné zlepšení vývoje ... 43
5.2.4 Otevřené otázky ... 43
5.3 Výsledky dotazníkového šetření ... 43
5.4 Závěr a možné vylepšení procesu vývoje ... 48
6 DOKUMENTACE PŘEVZATÉHO ENGINU ... **50**
6.1 Popis struktury enginu ... 50
6.1.1 Modul `grools` ... 50
6.1.2 Modul `cml` ... 54
6.2 Ukázka funkcionality ... 54
6.3 Shrnutí ... 59
ZÁVĚR ... **60**
SEZNAM OBRÁZKŮ ... **65**
SEZNAM TABULEK ... **66**
SEZNAM PŘÍLOH ... **67**

ÚVOD

Problematika udržování a vývoje zaběhnutých projektů je široká. Velkým problémem může být projekt, kde se vývoj provádí způsobem, který není pro firmu obvyklý. Další překážku může způsobit převzatý projekt, kde chybí dokumentace důležitého enginu, jenž kontroluje zobrazení, dostupnost a vlastnosti jednotlivých komponent v aplikaci. Z tohoto důvodu se diplomová práce zabývá vhodností daného vývojového procesu pro firmu a jeho využití, s případnou optimalizací pro další projekty, které by tento způsob vývoje umožňovaly.

První část práce představuje teoretické ohlédnutí na rozdíly mezi vodopádovým modelem a agilním způsobem vývoje. Dále popisuje podrobněji Scrum jako agilní metodiku řízení projektu, která se používá ve zmíněném převzatém projektu. Nakonec se zabývá obecným principem jak udržovat nebo vyvíjet software, a jak teoreticky zabránit hlavním příčinám, které mohou způsobovat problematickou údržbu softwaru.

Praktická část se zabývá odlišností mezi programovacími jazyky Java a Groovy. Dále představuje dvě groovy třídy, které se ve větší míře používají v převzatém enginu a jsou důležité pro pochopení jeho fungování. Následují část nabízí popis již zmíněného vývojového procesu a zhodnocuje, jestli je jeho dosavadní podoba vhodná pro další možné využití v jiných projektech, které by tuto variantu vývoje umožňovaly. Dále jsou předkládána vylepšení tohoto procesu tak, aby se během vývoje mohla dostávat lepší zpětná vazba na základě číselných hodnot. Nakonec je vytvořena dokumentace již zmíněného enginu, která by mohla pomoci pro využití enginu i na jiných místech či zefektivnit jeho údržbu.

Práce má za úkol zkoumat, zda je možné používat agilní způsob vývoje ve firmě, kde se zatím používal především vodopádový model, a tím pádem pak po implementaci možných navržených vylepšení takovýto způsob vývoje přenést do jiných a nových projektů v blízké budoucnosti. Dále si bere za cíl vytvořit dokumentaci převzatého enginu. Nově vytvořená dokumentace by mohla pomoci pochopit fungování a význam enginu v projektu a případně umožnit její využití v jiných projektech.

I. TEORETICKÁ ČÁST

1 VÝVOJOVÉ METODY

V této kapitole jsou popsány dvě základní metody vývoje. Jako první je popsána vodopádový model jako nejznámější reprezentant tzv. tradičních metodik a poté následuje obecný popis agilních metodik. Dále je v této kapitole podrobně popsána metodika Scrum, která je z velké části používána v dosavadním vývojovém procesu, který bude blíže popsán v kapitole 5.

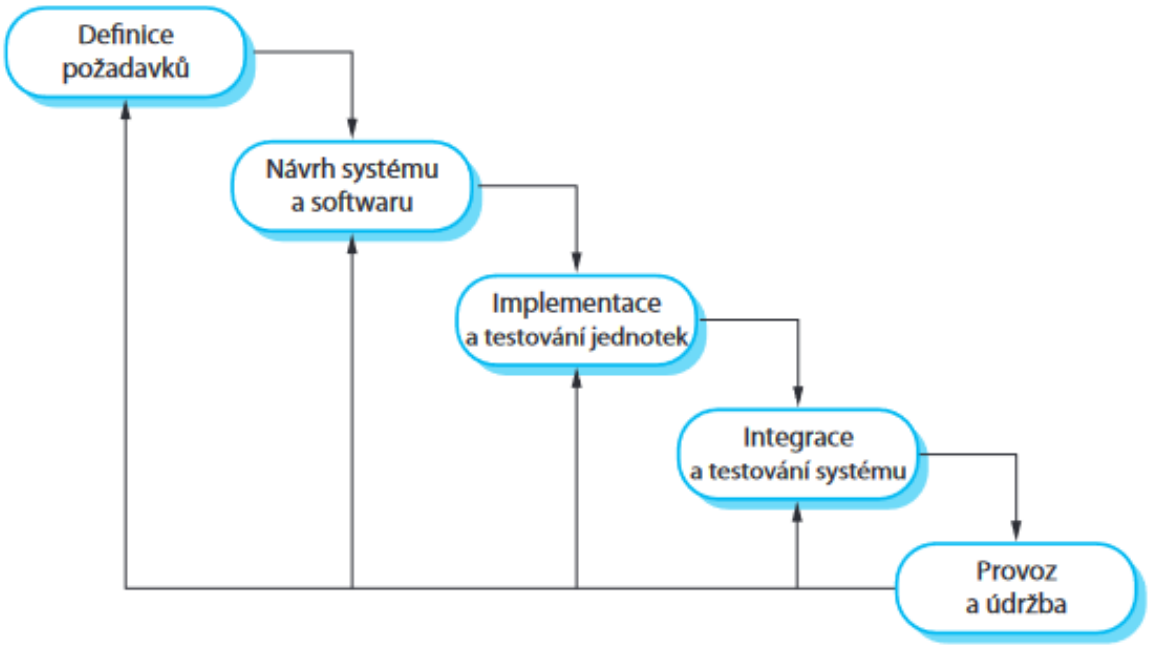
1.1 Vodopádový model

Jedná se o nejstarší publikovaný model procesu vývoje softwaru, který byl definován Winstonem W. Roycem v roce 1970.

Název je odvozen podle přirovnání posloupnosti jednotlivých fází procesu k protékání vody vodopádem. Toto přirovnání zároveň ukazuje na základní myšlenku vodopádového modelu a to na sekvenční přístup k jednotlivých fází s tím, že vstup do další fáze je možný až tehdy, pokud je předchozí fáze dokončena a uzavřena. [1]

Vodopádový model v podstatě představuje příklad plánovaného proces, ve kterém je principem nutnost plánovat všechny aktivity procesu předtím, než se na nich začne pracovat. [2]

Stejně jako vodopádový model tak i ostatní tradiční metodiky jsou velmi podrobné, formální a direktivní. [3]



Obrázek 1.1 Posloupnost jednotlivých fází ve vodopádovém modelu. [2]

Hlavní fáze vodopádového modelu jsou znázorněny na obrázku 1.1 a přímo odrážejí základní vývojové aktivity:

- Definice požadavků:

  Jedná se o fázi, kde se na základě konzultací s uživateli systému a analýzy určí systémové služby, omezení a cíle. Následně se definují detailněji a slouží jako specifikace systému.

- Návrh systému a softwaru:

  Proces návrhu systému zahrnuje přidělování požadavků na hardwarové či softwarové systémy tak, že definuje celkovou systémovou architekturu. Návrh softwaru zahrnuje identifikaci a popis elementárních abstrakcí softwarového systému a jejich vztahů.

- Implementace a testování jednotek:

  Při této fázi se realizuje návrh systému jako sada programů nebo programových jednotek. Testování jednotek slouží k ověření, zda každá jednotka splňuje danou specifikaci.

- Integrace a testování systému:

  Zde se jednotlivé programové jednotky nebo programy integrují a testují jako kompletní systém, tak aby bylo zajištěno, že vyhovují požadavkům na software. Po testování se softwarový systém předává zákazníkovi.

- Provoz a údržba:

  Většinou se jedná o nejdelší fázi, kde se systém instaluje a předává k praktickému používání. Údržba zahrnuje opravu chyb, které nebyly zjištěny v dřívějších fázích. Dále se zlepšuje implementace systémových jednotek a zdokonaluje systémových služeb s ohledem na nové požadavky.

Každá fáze je většinou zakončena jedním nebo více schválených dokumentů. I když by následující fáze neměla začít dříve než skončí předchozí, tak v praxi se často stává, že se jednotlivé stupně překrývají a dochází mezi nimi přenosu informací. Příkladem může být, že při kódování se naleznou problémy v návrhu atd. Z tohoto důvodu je možné, že dokumenty, které vzniknou na konci každé fáze, bude nutné upravit na základě provedených změň. [2]

Úspěšnost vodopádového modelu je závislá na dostatku času v počáteční fázi. Jenom tak je možné mít co největší úspory v pozdějších fázích, jelikož odhalení a odstranění

chyby, jenž je objevena na počátku (například při analýze), je mnohem levnější než kdyby se ta samá chyba opravovala později (například při testování).

Další nevýhoda vodopádového modelu je de facto nemožnost reagovat v průběhu vývoje na změny požadavků od zákazníka. [1]

1.2 Agilní metody

Pod pojmem agilní vývoj softwaru si lze představit soubor postupů, kde hlavní princip spočívá v úzké spolupráci mezi vývojářem a zákazníkem. Zákazník může přímo ovlivnit celý průběh vytváření webové nebo mobilní aplikace, jelikož vždy po dokončení určitého cyklu předkládá své připomínky k vývoji. Agilní metody vývoje stanovují, jak by měla vypadat komunikace v týmu a vztah s klientem/zákazníkem. V hlavní roli tedy již nestojí pouze znalosti a dovednosti programátora. Právě průběžnou spoluprací se zákazníkem se odlišuje od původního rigorózního vývoje, který zákazníkovi předkládá až finální verzi aplikace. Agilní způsob vývoje tak může přispět k větší spokojenosti klienta i postupnému zavádění aplikace do ostrého provozu, rovněž vede ke zvýšení produktivity a efektivity práce. Kupříkladu známá hudební streamovací služba Spotify tento způsob využila pro adaptaci svého prostředí posluchačům.

Pro agilní vývoj softwaru vznikl v roce 2001 i oficiální dokument s názvem „Manifest Agilního vývoje software“ (The Agile Manifesto), který blíže specifikuje jakési hodnoty a principy důležité při práci na projektu řízeném agilním přístupem [4].

Autoři manifestu dospěli k těmto hodnotám:

- „jednotlivci a interakce před procesy a nástroji,
- fungující software před vyčerpávající dokumentací,
- spolupráce se zákazníkem před vyjednáváním o smlouvě,
- reagování na změny před dodržováním plánu,

Jakkoliv jsou body napravo hodnotné, bodů nalevo si ceníme více.“ [5]

Autoři rovněž uvádí celkem 12 principů nebo tezí, na kterých stojí agilní vývoj softwaru. Mezi něž patří například jednoduchost, zdokonalování, udržitelný vývoj, motivace, fungující software známkou úspěchu, vzájemná komunikace a další [6].

Pokud je vytvářen projekt, který je řízen agilním přístupem, lze ho členit do několika fází:

1. nultá iterace – předložené požadavky se podrobí krátké analýze a vytvoří se základní kostra,

2. analýza změny – nastaví se plán pro jeden cyklus, jednotliví programátoři si rozdělí úkoly, které je nutno zpracovat do určitého času,
3. implementace požadované změny – fáze programování a realizace činností dle stanoveného plánu,
4. prezentace zákazníkovi – klientovi je předvedena první verze projektu, ke kterému může vznést připomínky.

Druhý cyklus projektu začíná v okamžiku vyjádření klienta, cyklus musí opět projít všemi uvedenými fázemi. Tímto způsobem se neustále opakuje až do úplné spokojenosti klienta. Klient má tedy na projektem průběžnou kontrolu. [4]

Pod agilní programování spadá hned několik přístupů/metod, které se vyznačují dalšími principy a vlastnostmi, přičemž každý je něčím specifický. V této části bude uveden pouze jejich stručný výčet:

- Extrémní programování (XP),
- Scrum,
- Crystal metodiky,
- Vlastnostmi řízený vývoj (FDD),
- Dynamic System Development Method (DSDM),
- Adaptivní vývoj software (ASD),
- Lean Development,
- Testy řízený vývoj (TDD). [7]

1.3 Srovnání metodik

Protože pro každý typ projektu se hodí využít jiný způsob vývojového procesu, je vhodné shrnout jejich odlišnosti. K porovnání metodik slouží tabulka číslo 1.1.

Hledisko	**Tradiční metodika**	**Agilní metodika**
Podrobnost metodiky	procesy a činnosti jsou popsány velmi podrobně	zaměřuje se na činnosti, které vytvářejí hodnotu a eliminuje činnosti, které hodnotu nepřinášejí
Kvalita	zaměření na kvalitu procesů a předpoklad, že kvalitní procesy povedou ke kvalitnímu výsledku	zaměření na hodnotu pro zákazníka a vysokou kvalitu produktu
Předvídatelnost	předpokládá předvídatelnost budoucnosti, důraz na anticipaci (sběr požadavků předem, plánování předem)	předpokládá nepředvídatelnost budoucnosti, důraz na adaptaci na změny (přírůstkové shromažďování požadavků, plánování pro iteraci)
Změny	změny podléhají řízení změn a je snaha změny minimalizovat	snaha změny umožnit a využít je, umožňují zákazníkům přehodnotit své požadavky s ohledem na nové znalosti
Participace zákazníka na projektu	jen v počátečních a koncových fázích, po podpisu dokumentu specifikace požadavků řízení přebírá tým technologických pracovníků	přesun nositele řízení z týmu na zákazníka, zákazník je řídícím subjektem během celého projektu, při každé iteraci zákazník může měnit priority funkcí
Vztah zákazníka s vývojářem	zajištěn smluvně, nedůvěra	důvěra a spolupráce
Lidský faktor	sekundární, dokumentačně zaměřené procesy se snaží vykázat lidi do role zaměnitelné součástky	primární, využívá individualit a silných stránek lidí
Kvalifikace lidí	stačí standardní jedinci	důraz na schopnosti, znalosti a dovednosti lidí
Forma komunikace	převážně písemná	důraz na komunikaci tváří v tvář
Dokumentace	rozsáhlá dokumentace	podstatná není dokumentace, ale pochopení

Tabulka 1.1 Porovnání metodik podle [8].

2 SCRUM

Scrum je jednou z nejpoužívanějších metodik agilního řízení projektu. Byl vytvořen v roce 1990 Kenem Schwaberem a Jeffem Sutherlandem.

Slovo Scrum v sobě obsahuje metaforu s hrou ragby. [9] „Scrum“ ,neboli česky „Mlýn“ je v ragby herní situace, kdy se shromáždí celý tým a společně se snaží získat držení míče. V této situaci musí celý tým tlačit stejným směrem a právě toto je příhodná metafora pro scrum tým, který pracuje společně na dosažení společného záměru. [9, 10]

Scrum dává vhodný rámec (framework) jak pro řízení malých, tak i větších a složitých projektů za využití produktových požadavků, každodenních schůzek a retrospektiv. [11]

Slovo „framework“, neboli česky „rámec“, znamená, že mnoho není specifikováno a proto je potřeba, aby byl navržen, tím kdo jej používá. Schwaber přirovnává Scrum k šachům. Šachy mají svá oficiální pravidla, ve kterých jsou specifikovány veškeré tahy, hráči, sekvence, bodování a tak dále. Pokud se je člověk naučí, může hrát šachy. Z počátku se může stát, že šachová partie nebude příliš dobrá, ale naučením různých strategií, taktik a pravidelným cvičením se lze neustále zdokonalovat. Nelze však měnit pravidla, nejednalo by se už více o šachy. Dostatečnou výzvou je naučit se hrát šachy s dokonalostí.

I autoři Scrumu několik desetiletí tvrdě pracovali na tom, aby měl podobu jakou má dnes. Při jeho použití je zapotřebí zdokonalovat techniky, týmovou práci, zapojovat zákazníky a mnoho dalšího. Nicméně rámec Scrum dobře poslouží jako zpětná vazba o pokroku a úspěchu toho, kdo jej používá. [12]

2.1 Definice

Jedná se o jednoduchý framework, který pomáhá lidem, týmům a organizacím vytvářet hodnotu skrze adaptivní řešení komplexních problémů.

V jednoduchosti Scrum potřebuje Scrum Mastera (2.4.1), aby vytvářel prostředí, kde:

1. Product Owner (2.4.2) řadí práci na složitém problému do Backlogu Produktu (2.5.1).
2. Scrum tým (2.4) přepracuje v rámci Sprintu (2.6.1) vybrané položky z Backlogu v hodnotný Inkrement (2.5.3).
3. Scrum tým a další zainteresované strany revidují výsledek a uzpůsobí patřičně své budoucí plány pro další Sprint

4. Opakuje

Framework je záměrně neúplný, definuje pouze části nutné k implementaci jeho teorie. Důvodem je to, že Scrum je postaven na kolektivní inteligenci lidí, jenž jej aplikují. Pravidla Scrumu spíše definují vztahy a interakce, než aby dávala lidem podrobné pokyny.

V rámci Scrumu je možné využívat různé procesy, techniky a metody. Framework zahrnuje dosavadní postupy nebo je činí zbytečnými. Dále zviditelňuje relativní účinnost současného řízení, prostředí a pracovních technik tak, aby se mohlo zrealizovat vylepšení.

2.2 Teorie

Scrum staví na empirismu a štíhlém myšlení (lean thinking). V empirismu se znalosti zakládají na zkušenostech a rozhodování na základě pozorování. Štíhlé myšlení snižuje plýtvání a zaměřuje se na to co je podstatné.

Scrum používá iterativní a inkrementální přístup k optimalizaci předvídatelnosti a k řízení rizika. Ve Scrumu jsou zapojeny skupiny lidí, kde všichni mají potřebné znalosti a dovednosti, aby zvládli určitou práci a sdílejí nebo nabývají dané dovednosti dle nutnosti.

Framework využívá událost nazvanou Sprint, ve které kombinuje čtyři formální události pro prozkoumání a přizpůsobení. Tyto události fungují, protože Scrum implementuje tři empirické pilíře. [11] Mezi tyto pilíře patří:

- Transparentnost (transparency):

  Všichni zúčastnění by měli mít jasný přehled o tom, co a proč se dělá a v jakém stavu se to nachází. Transparentnost umožňuje kontrolu.

- Kontrola (inspection):

  V krátkých periodách je nutná kontrola, aby se zjistilo, zda nemá dosavadní pokrok potencionálně nežádoucí odchylku nebo problémy. Na základě kontroly je možná adaptace. [9]

- Adaptace (adaptation):

  Na základě kontroly je možné adaptovat dosavadní proces, pokud daný aspekt nebo aspekty překračují hranici přijatelnosti. Tato adaptace by se měla provést co nejdříve tak, aby se minimalizovaly budoucí nejasnosti. Adaptace tedy pomáhá k vylepšení produktu. [13]

2.3 Hodnoty

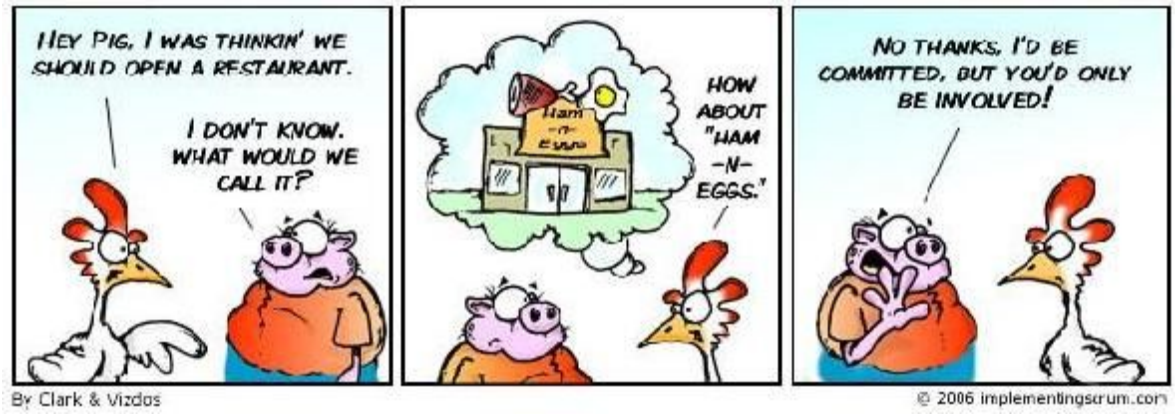
Členové Scrum týmu mohou ovlivnit, zda je použití Scrumu úspěšné. Závisí to na tom, jak zdárně dodržují hodnoty Scrumu. Mezi tyto hodnoty patří:

- Odhodlanost
- Soustředění
- Otevřenost
- Respekt
- Odvaha.

Scrum tým by tedy měl být odhodlaný dosáhnout cíle, podporovat a respektovat se navzájem. Dále by se měl tým primárně soustředit na to, aby ve Sprintu dosáhli co nejlepšího pokroku. Členové týmu by k sobě měli být otevřeni, pokud jde o práci, a měli by mít odvahu dělat správnou věc. [11]

2.3.1 Role

Ve Scrumu rozdělujeme role podle toho jak jsou zainteresovány v projektu. První skupinu tvoří Pigs (prasata) a druhou Chicken (kuřata). Toto rozdělení neznamená hanlivé označení, ale pouze vychází z krátké povídky o kuřeti a praseti, která je znázorněná na obrázku 2.1. [14]



Obrázek 2.1 Povídka o praseti a kuřeti. [15]

Příběh prasete a kuřete popisuje druhy lidí, kteří se účastní každodenních Scrum schůzek. Prasata jsou ti, kteří jdou na porážku – oddaní lidé, kteří mají na projektu podíl a jsou nezbytní pro jeho úspěch či neúspěch. Kuřata jsou ti, kteří se účastní

schůzky, ale nemají přímý vztah k aktualizaci, průběhu schůzky nebo projektu. Kuřata jsou tedy lidé, kteří mají co říct, ale obvykle nemají čím přispět během procesu vytváření. [16]

Mezi prasata tedy řadíme role Product Owner, Scrum Master a další členy Scrum týmu, jako jsou Developeři. Na druhou stranu mezi kuřata lze považovat koncového uživatele, konzultanta a další, kteří zasahují do schůzek, ale nemají přímou roli. [14]

2.4 Scrum tým

Scrum se zakládá na sebe-organizačním týmu, transparentní komunikaci, otevřené kultuře, sdílení informací a spolupráci. Pro správné fungování systému jsou ve Scrum týmu specifické role, které neexistují v klasických metodách managementu. [17] Mezi tyto role patří Product Owner (vlastník produktu), Scrum Master (nepřekládá se) a Development Team (vývojový tým). [9]

Scrum tým by měl být dostatečné malý, aby zůstal flexibilní a zároveň dostatečně velký, aby zvládl velkou část práce během Sprintu. Většinou se tým skládá z 10 nebo méně lidí. Bylo zjištěno, že menší týmy mají lepší komunikaci a produktivitu. Většinou se v týmu objevuje jeden Product Owner, jeden Scrum Master a zbytek tvoří Developeři.

Tým by měl být strukturovaný a zplnomocněný k řízení vlastní práce. Proto v týmu nebývají žádné dílčí týmy ani hierarchie. Zároveň je Scrum tým zodpovědný za vytvoření hodnotného a užitečného přírůstku, tzv. inkrementu, každého Sprintu. [11]

2.4.1 Scrum Master

Role Scrum Master není zcela standardní. Není to manažer ani klasický teamleader, je to člen týmu, stejně jako ostatní. Jeho úkolem je chránit tým a vytvářet prostředí, kde by nikdo nerušil členy týmu. Dále odstraňuje problémy, podporuje tým a pomáhá mu efektivně pracovat. [17] Scrum Master je tedy zodpovědný za úspěch procesu. Přesněji za to, že pomáhá Product Ownerovi a týmu Developerů používat správný proces k vytvoření úspěšného produktu a za usnadnění organizačních změn a zavedení agilního způsobu práce. V důsledku toho Scrum Master spolupracuje s Product Ownerem a vývojovým týmem, stejně jako s vrcholovým managementem, lidskými zdroji (HR) a obchodními skupinami, kterých se Scrum týká. [18]

Dobrý Scrum Master by tedy měl být:

- zodpovědný za efektivitu týmu,
- skromný, aby stavěl zájmy týmu nad své vlastní,

- schopný spolupráce, aby vytvářel prostředí, kde se členové týmu nebojí klást otázky a kolektivně řešit problémy,
- zavázaný k úspěšnému dokončení Sprintu,
- schopný ovlivnit ostatní, protože ve Scrumu není nadřízeným týmu a může tak spoléhat jenom na přirozenou autoritu a dovednost moderovat tok diskuzí,
- erudovaný, jelikož vedoucí osobnost orientující se v dané problematice navyšuje šanci týmu na úspěch. [19]

2.4.2 Product Owner

Vlastník produktu odpovídá za maximalizaci hodnoty produktu vyplývající z práce Scrum týmu. [11] Představuje zájmy lidí, kteří financují projekt nebo mají v plánu nějak využívat výsledný produkt, popřípadě budou jiným způsobem projektem ovlivněni. Product Owner odpovídá za tvorbu požadavků, návratnost investic a plán vydání. Jeho náplní práce je také určovat priority jednotlivých položek v Product backlogu, kde se zachytávají požadavky. [20]

2.4.3 Development team

Developeři jsou lidé ve Scrum týmu, kteří se snaží vytvářet použitelný přírůstek, neboli Inkrement, každého Sprintu. [11] Snaží se tedy vyvíjet funkcionalitu a přitom jim nikdo nezadává práci ani příkazy. Tým se tedy musí cítit zavázaný splnit cíl Sprintu, tak aby bylo dosaženo úspěchu v každé iteraci. [20]

Ideální počet vývojářů je 5-9 osob, i když je doporučováno kolem 7 lidí pro zachování co nejvyšší efektivity. Existuje i taková poučka, že když se nasytí tým dvěma pizzami, tak má tým správnou velikost. [19]

2.5 Artefakty

Artefakty reprezentují práci a její hodnoty různými způsoby, které jsou prospěšné pro poskytování transparentnosti. Dále jsou artefakty možností pro kontrolu a adaptaci.

Každý z artefaktů obsahuje tvz. závazek poskytující informace pro zvýšení transparetnosti. Závazky existují za cílem posílení empirismu a hodnot Scrumu. [11]

2.5.1 Product Backlog

Produktový Backlog je seznam s prioritami všech požadavků na funkcionalitu produktu. Zároveň se jedná o jediný zdroj práce pro Scrum tým. Zodpovědnost za něj nese Product Owner, který zadává položky a upravuje jejich prioritu. [11]

Product Backlog by měl dodržovat zásady DEEP:

- Detailed appropriately:

  Množství detailu položky roste s jeho rostoucí prioritou. Ze začátku jsou položky popsány pomocí User Story (uživatelské příběhy), což znamená z pohledu zákazníka a ne z pohledu technologie nebo systému. Následně se zvyšující se prioritou se User Story zpřesňují, případně rozdělují na několik dílčích.

- Estimated:

  K položkám se připojují odhady pracnosti.

- Emergent:

  Product Backlog se pořád mění.

- Prioritized:

  Položky slibující poskytnutí vyšší hodnoty produktu, mají vyšší prioritu. [19]

Product Goal (Produktový cíl) je závazek Product Backlogu a popisující budoucí stav produktu. Produktový cíl je pro Scrum tým dlouhodobým cílem a bez jeho splnění, případně zahození, nemůže přejít na další produktový cíl.

2.5.2 Sprint Backlog

Tento Backlog je tvořen z množiny vybraných z Produktového Backlogu, plánu dodání inkrementu a splnění cíle Sprintu. Jedná se o plán Developerů a je pro ně určen. V podstatě se jedná o obraz práce v reálném čase, který se plánuje udělat v rámci Sprintu. Umožňuje tedy jednoduše vidět kolik práce je hotovo a kolik zbývá během Sprintu. Stejně jako Produckt Backlog, tak i tento se mění v průběhu na základě více detailů. Změny ve Sprint Backlogu se provádí na tvz. Daily Scrumu.

Závazkem Sprint Backlogu je Sprint Goal neboli Cíl Sprintu. Tento Cíl je vytvořen během události Sprint Planning a následně se dodá do Sprint Backlogu. Slouží také ke kontrole pokroku, k řešení rizik, pomáhá stanovit priority, podporuje týmovou práci a umožňuje efektivní rozhodování.

2.5.3 Inkrement

Neboli přírůstek je konkrétní odrazový můstek k cílovému stavu produktu. Jednotlivé inkrementy jsou aditivní ke všem předchozím inkrementům. Proto aby měl přírůstek hodnotu, tak musí být použitelný.

Definice Hotovo je závazek přírůstku a jedná se v podstatě o formální popis jeho stavu, pokud splňuje kvalitativní rysy pokroku, jež je požadován u produktu. Definice

tvoří transparentnost tím, že umožňuje všem sdílené pochopení toho, která práce byla dotvořena během Inkrementu. Pokud položka nemá tuto definici, nemůže být ukončená nebo prezentována na Sprint Review. Každý produkt nebo systém by měl mít popsanou svou vlastní Definici Hotovo jako standard pro všechnu svou práci. [11]

2.6 Události

Události ve Scrumu fungují jako formální příležitost kontrolovat a přizpůsobovat artefakty Scrumu. Tyto události jsou navrženy tak, aby umožňovaly transparentnost. Proto neprovedení nějaké události znamená ztratit příležitost kontrolovat a přizpůsobit proces vývoje. Zároveň události ve Scrumu nabízí vytvořit pravidelnost a minimalizovat potřeby schůzek, jenž nejsou ve Scrumu definované. Z tohoto důvodu je nejoptimálnější konat události ve stejnou dobu a na stejném místě, aby se snížila složitost.

2.6.1 Sprint

Jedná o hlavní událost ve Scrumu, kde se nápady proměňují v hodnoty. [11] Sprint má pevnou délku má pevnou délku. Většinou je doporučována délka jeden měsíc nebo i méně. Scrum je flexibilní natolik, že se lze setkat i s délkou méně než týden. [20] Důvodem pevné délky je vytvoření konzistence. Další Sprint začíná až poté, co je předchozí ukončen. Jinými slovy, všechna práce nutná dosažení Produktového cíle se vykonává ve Sprintech.

2.6.2 Sprint Planning

Plánování Sprintu zahajuje Sprint a to tak, že stanoví práci, která má být hotová během daného Sprintu. Tento plán je vytvořen za spolupráce celého Scrum týmu. [11]

Na začátku plánování Product Owner sdělí Scrum týmu Cíl Sprintu a co je potřeba udělat. Tento výběr je vytvořen na základě prioritizace položek z Product Backlogu. Developeři po úvaze sdělují kolik z požadovaných položek lze stihnout v nadcházejícím Sprintu. Tento dialog moderuje Scrum Master, pokud je potřeba. Po této části přichází plánování průběhu daného Sprintu. [20]

Pro nenarušování kontextu a práce týmu víkendem je doporučováno, aby se Sprint Planning konal začátkem týdne. [21]

Do Sprint Planningu lze zařadit i Backlog Grooming, kde Product Owner s vývojovým týmem prochází jednotlivé položky z Product Backlogu pro lepší pochopení jejich významu. [17]

2.6.3 Daily Scrum

Cílem Denního Scrumu je kontrolovat progres směrem k Cíli Sprintu a dle potřeby poupravit Sprint Backlog a uspořádat plánovanou nadcházející práci.

Jedná se o událost, která se koná pravidelně každý pracovní den, trvá 15 minut a účastní se jí jen Developeři.[11] Většinou se koná ve stoje, aby byla podpořena tendence ke stručnosti vyjadřování. Každý z účastníků odpovídá na tři otázky: Co jsi na projektu udělal od posledního Daily Scrumu? Co plánuješ udělat na projektu do příštího Daily Scrumu? Jsou nějaké překážky, které stojí v cestě splnění závazků na aktuálním Sprintu a projektu? [20]

2.6.4 Sprint Review

Cílem Sprint Review je zkontrolovat výsledek Sprintu a definovat další úpravy. Developeři na této události představují výsledky vykonané práce Scrum Masterovi, Product Ownerovi a dalším, například zákazníkovi nebo jiným vývojářů z jiného projektu. Jedná se o předposlední událost ve Sprintu. [11] Dalším důvodem konání události je povzbudit přítomné k další práci. [20].

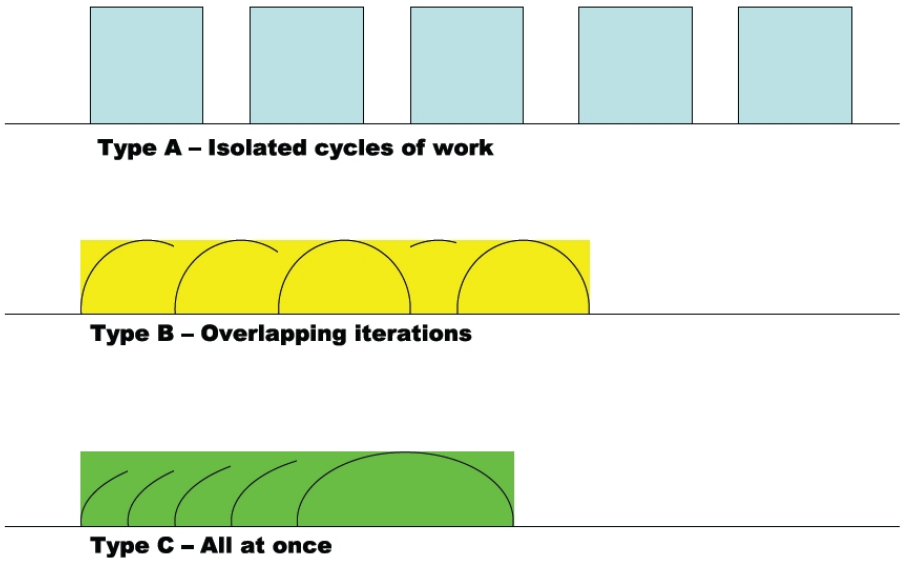
2.6.5 Sprint Retrospective

Účelem Sprint Retrospektivy je vymyslet způsoby, jak zvýšit kvalitu a efektivitu. To je učiněno na základě diskuze o průběhu posledního Sprintu, co fungovalo, jaké byly problémy a případně jakým způsobem byly či nebyly řešeny. Jedná se o poslední událost, která uzavírá daný Sprint. [11]

2.7 Varianty Scrumu

V komunitě softwarových vývojářů se mluví o tzv. variantách Scrumu. V původní variantě se počítá s vydáním produktu přibližně jednou za půl roku a to nemusí vždy stačit. Proto za pomoci uživatelské základny se tedy evolučně vyvinuly varianty Scrumu, které nabízejí o něco vyšší produktivitu a škálovatelnost. Lze tedy rozlišovat Scrum typu A, B nebo C. Varianta A značí původní podobu. Typ B už spočívá v prolínání a typ C dokonce přímo ve sloučení Sprintů. Pro vizuální porovnání jsou zmíněné verze Srumu znázorněny na obrázku 2.2. Doporučuje se nejprve dobře zvládnout původní variantu a až poté přejít na typ B. Přitom je nutné pochopit i princip tzv. Lean developmentu. Pro variantu C je zase vhodné zvládnout dobře typ B.

Původní podoba, neboli varianta A, má jednotlivé Sprinty odděleny od sebe. Z tohoto důvodu mezi jednotlivými iteracemi nastává zdržení v podobě reorganizace na další Sprint. Navíc v počátečních fázích Sprintu nějakou dobu trvá, než vývojáři dobře



Obrázek 2.2 Varianty Scrumu. [22]

pochopí požadavky od uživatele a začnou s programováním.

Typ B používá včlenění přípravných prací na příští Sprint už do aktuálně běžícího. Tím se zbaví prodlevy, která nastává u varianty A. Sprinty tedy na sebe navazují kontinuálně a Produckt Backlog je neustále plný a tým je celkově produktivnější. Z tohoto důvodu je možné variantu B označit jako Continous Flow.

Při variantě C probíhá několik Sprintů současně, proto se označuje jako All-At-Once. U tohoto typu se vyskytují nové artefakty a to Scrum of Scrums, kde se na denní bázi setkávají vedoucí jednotlivých týmů. Jednou týdně se také objevuje MetaScrum, určený pro všechny stakeholdry, jehož záměrem je kontrolovat všechny Sprinty a upravovat jejich směřování. Výsledkem jsou minimální režijní náklady, nicméně předpokladem je pokročilá automatizace řídících procesů. [22]

2.8 Kombinace s Extrémním programováním

Začlenění postupů Extrémního programování (XP) a technik Scrumu je považováno za docela efektivní, když se XP zaměřuje na technické aspekty a Scrum organizuje proces. [23] Zároveň se jedná o nejčastěji spojované metodiky. [24] Dokonce i některé literatury zabývající se Scrumem automaticky propojují některé techniky XP. Příkladem může být párové programování, 40hodinový týden a důraz na kód a testy. [19, 25]

- Párové programování:

  Zdrojový kód je psán v páru, kdy dva vývojáři sedí u jednoho počítače. Jeden programátor má roli „pisatele“ a uvažuje o nevhodnější implementaci části systému, druhý programátor plní roli „kontrolora“ a musí brát v potaz celou strategii

projektu. Páry se většinou mění v průběhu dne.

- 40hodinový týden:

  V XP je kladen důraz na 40hodinový pracovní týden, jelikož neodpočinutí programátoři jsou demotivovaní a neodvádějí tak kvalitní práci. Přesčasy jsou zde brány jako známka problému s projektem. Přesčasy v prvním týdnu jsou ještě v pořádku, ale v druhém týdnu jsou již špatné a znamenají velké problémy s projektem, které mnohdy nevyřeší pouze přesčasy. Někdy je pak nutné přeplánovat celý projekt.

- Standardy pro psaní zdrojového kódu:

  Důležitost nastavení standardů spočívá ve společném vlastnictví kódu. Standard by se měl zaměřovat především na komunikaci. Tato pravidla by měl znát každý programátor v týmu a přijmout je za své.

- Testování:

  Pomocí jednotkových testů (unit tests) je ověřován zdrojový kód a programátoři si budují důvěru v provozuschopnost již dříve vyvinutých částí systému. Zákazník za pomoci testů funkcionality rozpozná, která část již úspěšně funguje a kolik ještě chybí. [26]

2.9 Kombinace s Kanbanem

Se Scrumem lze zkombinovat metodiku Kanban, tak aby se využily jejich nejlepší funkce. Spojení Scrumu s Kanbanem může mít desítek variací. Jedna z nejpopulárnějších zdokumentovaných možností se nazývá Scrumban. Tato varianta využívá Scrum jako svůj základ s několika rozdíly podle Kanbanu:

- Neexistují žádné předdefinované role, tým si ponechává role, které již má, nebo si vytváří jiné, které potřebuje.
- Iterace jsou většinou velmi krátké.
- Všechny úkoly jsou umístěny na tabuli.
- Používá se limit pro množství práce.

Nicméně je možné vytvořit vlastní kombinaci Scrumu a Kanbanu, který může vypadat více jako Kanban nebo více jako Scrum. [13]

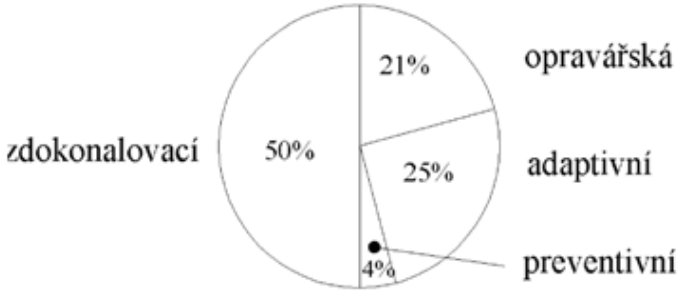
3 OBECNÉ PRINCIPY ÚDRŽBY A VÝVOJE SOFTWARU

Údržba programového díla bývá často podceňována, i když představuje téměř 80 % času v životě programu. Přitom udržet nestrukturované, upravované a špatně dokumentované programy v chodu spotřebovává velké množství materiální a lidské kapacity.

Protože chyby je nutné opravovat a systémy přizpůsobovat měnícím se podmínkám, tak se softwarová údržba neomezuje jenom na opravu latentních chyb, ale rozlišují se 4 druhy údržbových prací:

1. Opravářská údržba - odstranění nalezené chyby
2. Adaptivní údržba - přizpůsobení softwaru změnám prostředí například novému hardwaru nebo nové verzi operačního systému. Tato údržba nemění funkce systému
3. Zdokonalovací údržba - zahrnutí nových nebo změněných požadavků uživatele. Vede k funkčním vylepšení systému (dle Murpyho zákona každá změna je k horšímu). Občas se tento typ údržby věnuje zvýšení výkonnosti systému a vylepšení uživatelského rozhraní.
4. Preventivní údržba - práce a aktivity zaměřené na zvýšení udržovatelnosti systému například aktualizování a doplnění dokumentace nebo komentářů.

Na obrázku 3.1 je znázorněn graf podílu jednotlivých aktivit založených na datech ze 70. let. Z grafu je patrné, že změny okolního systému nebo uživatelských požadavků jsou nevyhnutelné, neboli software se musí vyvíjet.



Obrázek 3.1 Podíly údržbových prací. [27]

Profesionální přístup k údržbě softwaru respektuje následující aspekty:

- znalost hlavních zdrojů problémů řešených při údržbě
- cílené postupy pro zvonu nalezení faktů a metodicky prováděnou restrukturalizaci softwaru zaměřenou na zlepšení údržby
- organizační a manažerská opatření vedoucí ke zlepšení údržby

Za hlavní příčiny problematické údržby lze považovat nestrukturovaný kód, nedostatečná znalost systémové či aplikační oblasti, nedostatečná dokumentace a špatná pověst softwarové údržby.

3.1 Nestrukturovaný kód

Pojem nestrukturovaný kód v sobě obsahuje všechny špatné jevy, které se můžou vyskytovat při vytváření softwaru. [27]

Zde je výčet chyb a jak se jim vyhnout při psaní kódu:

- Příliš mnoho argumentů:

  Funkce by měly mít co nejmenší počet argumentů. Nejlepší je žádný argument, následovaný jedním, dvěma a třemi. Použít víc jak tři argumenty je velmi sporné a je třeba se tomu vyhýbat.

- Booleanské argumenty:

  Booleovské argumenty říkají, že funkce dělá více než jednu věc. Z tohoto důvodu jsou matoucí a měly by se používat jako argument.

- Nepoužívané funkce:

  Metody, které nejsou nikdy volány, je možné odstranit. Uchovávat tzv. mrtvý kód je zbytečné. Není třeba se bát ji smazat, protože řídicí systém zdrojového kódu si stále pamatuje předchozí podobu kódu.

- Duplikace kódu:

  Duplikace kódu představuje promarněnou příležitost k abstrakci. Duplikace by se pravděpodobně mohla stát podprogramem nebo možná rovnou jinou třídou. Složením duplikace do takové abstrakce se rozšíří slovní zásoba jazyka návrhu. Ostatní programátoři mohou používat abstraktní prostředky, které tímto způsobem vytvoří. Díky tomu je kódování rychlejší a méně náchylné k chybám, protože se zvýšila úroveň abstrakce.

3.2 Nedostatečná znalost systému nebo aplikační oblasti

Nedostatečná znalost aplikační oblasti je možné chápat jako všeobecný jev, se kterým se setkává vývoj softwaru. Následně během údržby se tento nedostatek navyšuje.

Většinou je pouze vlastní zdrojový text programu spolehlivý zdroj poznatků o systému či aplikační oblasti. Bez tohoto zdroje může nastat situace, že dobře zamýšlející zásah do nedostatečně pochopené části programu může mít nepředvídatelné následky a dále se kvůli tomu můžou zvyšovat nároky na údržbu.

3.3 Nedostatečná dokumentace

Chybějící nebo neúplná dokumentace přidává při údržbě programu problémy. Nejhorší je hlavně neaktuální dokumentace, která už není odpovídající dle skutečnosti. S blížícím se termínem se kvalita dokumentace zhoršuje. [27]

Při vytváření komentářů je vhodné držet se pravidly čistého kódu:

- Nevhodné informace:

  Komentář by neměl obsahoval informace, které jsou lépe uchovány v jiném typu systému jako například verzovací systém. Historie změn například jen zahlcuje zdrojové soubory množstvím historického a nezajímavého textu. Obecně platí, že metadata, jako jsou autoři nebo datum poslední změny, by se v komentářích neměla objevovat. Poznámky by měly být vyhrazeny pro technické poznámky ke kódu a designu.

- Zastaralý komentář:

  Zastaralý komentář je ten, který je irelevantní a nesprávný. Nejlepší je nepsat komentář, který se stane zastaralým. Pokud se najde zastaralý komentář, je nejlepší jej aktualizovat nebo se ho co nejrychleji zbavit. Zastaralé komentáře mají tendenci migrovat pryč z kódu, který kdysi popsali. V kódu se stávají plovoucími ostrovy irelevantnosti a nesprávného nasměrování.

- Nadbytečný komentář:

  Komentář je nadbytečný, pokud popisuje něco, co adekvátně popisuje sám kód. Komentáře by měly říkat věci, které kód sám o sobě říct nemůže.

- Špatně napsaný komentář:

  Pokud se píše komentář, je potřeba si udělat čas a ujistit se, že je to ten nejlepší komentář, jaký lze napsat. Je vhodné pečlivě volit slova, používat správnou gramatiku a interpunkci. Dále být struční a neuvádět zřejmé.

- Zakomentovaný kód:

  Zakomentovaný kód jenom sedí a hnije a každým dnem je méně a méně relevantní. Volá funkce, které již neexistují. Používá proměnné, jejichž názvy se změnily. Řídí se konvencemi, které jsou dávno zastaralé. Znečišťuje moduly, které ji obsahují, a odvádí pozornost lidí, kteří se ji snaží číst. Zakomentovaný kód je nutné smazat. Není potřeba ho uchovávat, řídicí systém zdrojového kódu si to stále pamatuje. Pokud to někdo opravdu potřebuje, může se vrátit a podívat se na předchozí verzi. [28]

I při dodržení výše zmíněných pravidel může mít dokumentace ještě nedostatky typu chybějící zdůvodnění, proč bylo přijato konkrétní řešení. Dále může stát, že programátoři vše řádně zdokumentují a vytvoří kód, který dělá něco jiného, případně nenaplňuje všechny zmíněné funkce.

3.4 Špatná pověst softwarové údržby

Údržba je považována za druhořadou práci a proto je v organizaci údržbou pověřováni nově přijatí, nezkušení nebo méně zdatní zaměstnanci.

Nicméně údržba je ve skutečnosti obtížná práce, jelikož zahrnuje širší spektrum činností, než je potřeba při vývoji. Programy, které byly napsány někým jiným, je nutné pochopit, správně zdokumentovat jejich činnost, restrukturalizovat, opravovat a tak dále. Tyto činnosti jsou potřeba vykonávat ve značném časovém stresu. Z tohoto důvodu musí být „údržbář“ chytřejší a zručnější, než „vývojář“. [27]

II. PRAKTICKÁ ČÁST

4 EKOSYSTEM GROOVY/JAVA

Tato kapitola popisuje hlavní rozdíly mezi programovacími jazyky Java a Groovy. Dále se věnuje dvěma třídám z knihovny Groovy, které se využívají v rámci CML enginu. Jmenovitě se jedná o Closure a Binding.

4.1 Odlišnosti jazyků Groovy a Java

Syntaxe Javy a Groovy je velmi podobná. Je to z toho důvodu, že Groovy je navržená, tak aby Java programátoři neměli problém se psaním kódu. Nicméně Groovy má některé odlišnosti:

- Operátor == se v Groovy rovná metodě *equals()* v Javě. V Javě se tímto operátorem jedná o rovnost u primitivních typů, ale u objektů porovnává shodnost jejich identity. Pro shodnost identity v Groovy se používá ===. V Groovy se tedy == používá pro rovnost všech typů.
- Není nutnost používat ukončovací středník (`;`), pokud se nevkládá více příkazů na jeden řádek.
- Příkaz `return` se volá automaticky na konci metody a tím pádem i tento příkaz není povinný.
- Ve statických metodách je možné používat klíčové slovo `this` pro odkaz na třídu, ve které se metoda vyskytuje.
- Metody jsou ve výchozím nastavením privátní a třídy veřejné.
- `protected` je ekvivalentní pro oba typy, obecnému a taky v rámci `package` v Javě.
- Třídy definované uvnitř jiné třídy, neboli vnitřní třídy, nejsou v Groovy podporované. Nicméně lze místo nich používat třídu Closure, která je blíže popsána v části 4.2.
- Pomocí Closure je možné používat metody *each()* a *eachWithIndex()* pro procházení prvků.
- Není podporované `throws` v definiční hlavičce metody. Důvodem je, že se nerozlišují výjimky na kontrolované a nekontrolované jako v Javě.
- V Groovy je možné používat statické a dynamické typování. Jinými slovy není nutné pokaždé uvádět typ deklarované proměnné nebo návratové hodnoty metody.

- Je dovoleno používat polymorfní iterace. Například příkaz `for` může v Groovy iterovat pole, kolekce nebo mapy, atd.
- V Groovy se nachází rozšířený příkaz *switch*, kde se můžou u jednotlivých `case` vyskytovat seznamy, řady, datové typy a další. V podstatě podobně jako je tomu u podmínky `if`. [29, 30]

4.2 Closure

Closure (česky by se dalo přeložit jako Uzávěr) je jednou z nejvýkonnějších funkcí mnoha programovacích jazyků. Pro její pochopení je nutné chápat globální proměnné (proměnná definovaná mimo funkci a lze k ní přistupovat odkudkoli v aplikaci) a lokální proměnné (proměnná definovaná uvnitř funkce a je možné ji použít pouze v rámci funkce. V případě použití mimo funkci se zobrazí chyba).

V mnoha programovacích jazycích je možné aby funkce vracela jinou funkci. V tomto případě vracející se funkce uchovává odkaz na všechny proměnné, které potřebuje ke svému spuštění a které jsou deklarovány v nadřazené funkci. Přesně na tento popis sedí označení Closure (Uzávěr). [31] Jinými slovy anonymní funkce v uzávěrech dostane požadovaný kontext (hodnoty proměnných) ve stavu, v jakém je v danou dobu volána. V případě ukončení nadřazené funkce dostane uzávěr všechny proměnné, které byly k dispozici v době ukončení této nadřazené funkce. [32]

Princip uzávěrů se hojně využívá v Groovy. V podstatě se jedná o skupiny příkazů a dat, které jsou uzavřeny ve složených závorkách. [29]

V Groovy se dá mluvit o uzávěrech jako o instanci třídy Closure. Tato třída může mít 0 a víc parametrů a vždy vrací hodnotu. Kromě toho může Closure během provádění přistupovat k okolním proměnným mimo svůj rozsah a používat je spolu se svými lokálními proměnnými. Dále je možné k proměnné přiřadit Closure nebo jej předat jako parametr metodě. Z toho důvodu Closure umožňuje vytvářet funkce pro tzv. zpožděné provádění.

Groovy Closure obsahuje parametry, šipku → a kód ke spuštění, jak je možno vidět na řádcích (2-4) na obrázku 4.1. Parametry jsou volitelné a jsou-li uvedeny, oddělují se čárkami

Closure můžeme volat dvěma způsoby. Prvním způsobem je volat ji jakoukoliv jinou metodou (6). Druhou možnosti je volání za pomoci funkce *call()* (7).

Parametry v Groovy Closure se řídí stejnými principy jako parametry u běžných metod. Mohou mít volitelný typ, jméno a volitelnou výchozí hodnotu. [33]. Closure kromě toho ještě umožňuje, aby byl definován i bez parametrů. V tomto případě se předpokládá implicitní parametr s názvem „it“ (9-11). Dále je možné Closure definovat

s více parametry najednou (14-17). [34] Také lze deklarovat proměnlivý počet argumentů (19-21).

Další vlastnost Closure je ta, že umožňuje aby se předával jako argument pro běžnou Groovy funkci. Díky tomu metoda může zavolat Closure pro dokončení svého úkolu a tím umožňuje přizpůsobit chování dané metody. Dále je možné jeden Closure vnořovat do druhého. Ještě je možné provádět líné vyhodnocení řetězců, jejich přepočítáním z aktuálních hodnot nacházejících se kolem řetězce (24-28).

```
1  // Deklarace
2  def print = { name →
3      println name
4  }
5  // Volání
6  formatToLowerCaseClosure("Hello! Closure")
7  formatToLowerCaseClosure.call("Hello! Closure")
8  // Použití implicitního parametru
9  def greet = {
10     return "Hello! ${it}"
11 }
12 assert greet("Alex") == "Hello! Alex"
13 // Použití více parametrů
14 def multiply = { x, y →
15     return x*y
16 }
17 assert multiply(2, 4) == 8
18 // Použití proměnlivého počtu parametrů
19 def addAll = { int... args →
20     return args.sum()
21 }
22 assert addAll(12, 10, 14) == 36
23 // Ukázka líného vyhodnocení
24 def fullName = "Tarly Samson"
25 def greetStr = "Hello! ${→ fullName}"
26 assert greetStr == "Hello! Tarly Samson"
27 fullName = "Jon Smith"
28 assert greetStr == "Hello! Jon Smith"
```

Obrázek 4.1 Ukázka používání Groovy Closure. Převzato z [34].

Jak už bylo zmíněno výše, Closure jsou výkonnou Groovy funkcí, protože mají několik výhod oproti běžným metodám. Jejich srovnání mezi běžnými metodami v Groovy a zároveň shrnutí výše zmíněných vlastností je znázorněno níže:

- Closure můžeme předávat jako argument metodě.

- Closure mohou používat implicitní parametr „it“.
- Do Closure můžeme přiřadit proměnnou a provést operaci později, buď jako metodu nebo pomocí metody call.
- Groovy určuje návratový typ Closure za běhu.
- Je možné Closure deklarovat a volat uvnitř dalších Closure.
- Closure vždy vrací hodnotu.

Na základě zmíněných vlastností poskytuje Groovy Closure efektivní způsob, jak vložit funkčnost do objektů a metod pro zpožděné spuštění. [34] Pro tento důvod je třída Closure používaná v převzatém enginu, který je popsán v části 6.

4.3 Binding

Skripty v Groovy umožňují používat nedeklarované proměnné. Při využití nedeklarované proměnné se předpokládá, že daná proměnná pochází z vazby skriptu a je přidána do vazby, pokud v ní ještě není. Jako vazba se v Groovy označuje datové úložiště Binding, které umožňuje přenos proměnných do a z volajícího skriptu. [35] Třída Binding tedy představuje proměnné, které lze navázat se skriptem. Tyto proměnné lze změnit z vnějšku objektu skriptu nebo vytvořit mimo skript a předat do něj. [36]

Bez mechanismu vazby bychom nemohli předat kontextové informace do skriptu, který se načte a vyhodnotí za běhu. Zároveň bez toho by se výrazně omezilo to, čeho je možné pomocí skriptů dosáhnout. [37] Ukázka použití třídy Binding je na obrázku 4.2.

```
1 // Ukázka vytvoření a chování třídy Binding
2 class BindingDemo {
3     static void main (String[] args) {
4         Binding sampleBinding = new Binding()
5         println sampleBinding.variables // [":""]
6
7         sampleBinding.message = "Hello"
8         println sampleBinding.variables // ["message":"Hello"]
9         println sampleBinding.message // "Hello"
10    }
11 }
```

Obrázek 4.2 Ukázka používání Groovy Binding. Převzato z [37].

Ve výše uvedeném kódu se začíná s vytvořením instance `groovy.lang.Binding` (4). Je možné si všimnout, že `sampleBinding.variables` je prázdná mapa (5). Při-

dání hodnot k vazbě je dosaženo pomocí operátoru tečka (7), což se podobá nastavení hodnoty do mapy. Následně je docíleno přiřazení textové hodnoty výpisem celé třídy Binding (8), tedy všech proměnných a jejich hodnot, a poté (9) jenom pro určitou proměnnou v Binding.

5 ANALÝZA A OPTIMALIZACE VÝVOJOVÉHO PROCESU

Hlavním cílem analýzy vývojového procesu v převzatém projektu je zjistit, zda tento způsob vývoje je možný využít ve firmě i k vedení dalších projektů.

Aktuálně se totiž ve firmě vyvíjí celkově 44 projektů, z toho se 34 vyvíjí pomocí vodopádového modelu a 9 jiným způsobem vývoje. Tento jiný způsob vývoje je založen primárně na udržení SLA (Service-level agreement) a vývoj se provádí sekundárně. Nebo se používá pro interní projekty, kde se vývoj může zastavit, pokud je to nutné.

V převzatém projektu se uplatňuje agilní vývojový proces vytváření nových částí. Tento vývojový proces je inspirovaný z velké části Scrumem. Kvůli tomu, že tento způsob vývoje není pro firmu tradiční, je zapotřebí jej z analyzovat a zjistit, zda je jeho nastavení vhodné pro vývojáře, které s tímto vývojem nemají moc zkušeností. Obzvlášť, když firma velmi často pracuje se studenty, kteří během zkouškového období mají menší pracovní aktivitu. V případě odhalení nedostatků v tomto novém procesu vývoje, bude navržena optimalizace jeho fungování.

5.1 Popis vývojového procesu

Dosavadní vývojový proces díky tomu, že používá agilní způsob vývoje, umožňuje, na rozdíl od ostatních procesů ve firmě, reagovat na jakékoliv změny, které zákazník požaduje.

Popis tohoto procesu je rozdělen na jednotlivé části, které se blíže zaměřují na určité oblasti vývoje.

5.1.1 Složení členů vývojového týmu

Do vývojového procesu se zapojují dva frontendisti, z toho jeden student, a tři backendisti, z toho dva studenti jako členové vývojového týmu. Ještě se zde nachází Scrum Master a Product Owner, který jako jediný není z firmy, ale jedná se o osobu dosazenou zákazníkem. Dále se ještě v týmu vyskytuje tester, který je taktéž studentem.

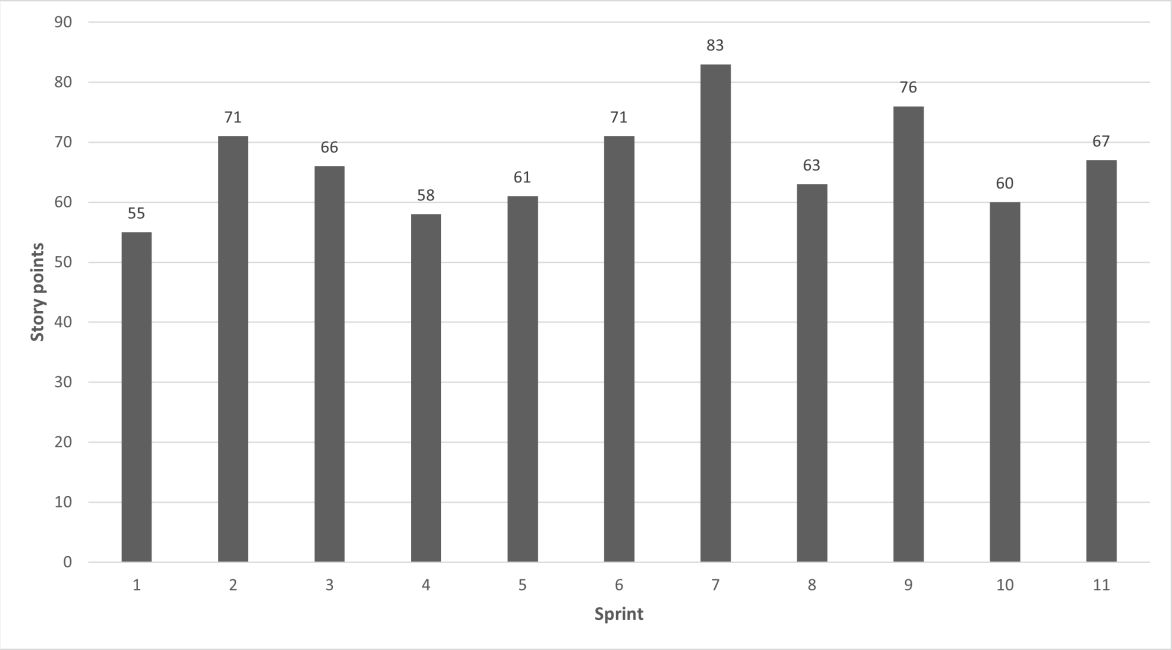
Je nutné, aby se na projektu nacházel i stálý zaměstnanec z důvodu výkyvů v docházce studentů zejména ve zkouškovém období. Tento zaměstnanec během roku pracuje i na jiných projektech a v případě výpadku studentů je více zainteresován v daném projektu. Zároveň působí pro studenty i jako mentor a pomáhá jim s řešením obtížných problémů.

K udržení stejného inkrementu, který tým vytváří, se využívají ještě story points[1)] pro odhad pracnosti místo klasických hodin nebo tzv. člověkodny (MDs).

[1)] Story point neboli česky příběhové body je možné chápat jako číslo, které týmu vypovídá o úrovni obtížnosti položky. Obtížnost může souviset se složitostí, riziky a vynaloženým úsilím. [38]

Pomocí tohoto nastavení je možné udržet potřebnou kapacitu programátorů, a tím udržet téměř stejný inkrement na zákazníka i během zkouškového období.

Udržení stejného počtu story points je možné vidět na obrázku 5.1, který znázorňuje graf jednotlivých story points za 11 Sprintů.



Obrázek 5.1 Graf počtu naplánovaných story points za jednotlivé Sprinty

Z grafu je možné zjistit, že průměrný počet story points na Sprint je 66,455 a se směrodatnou odchylkou 8,323.

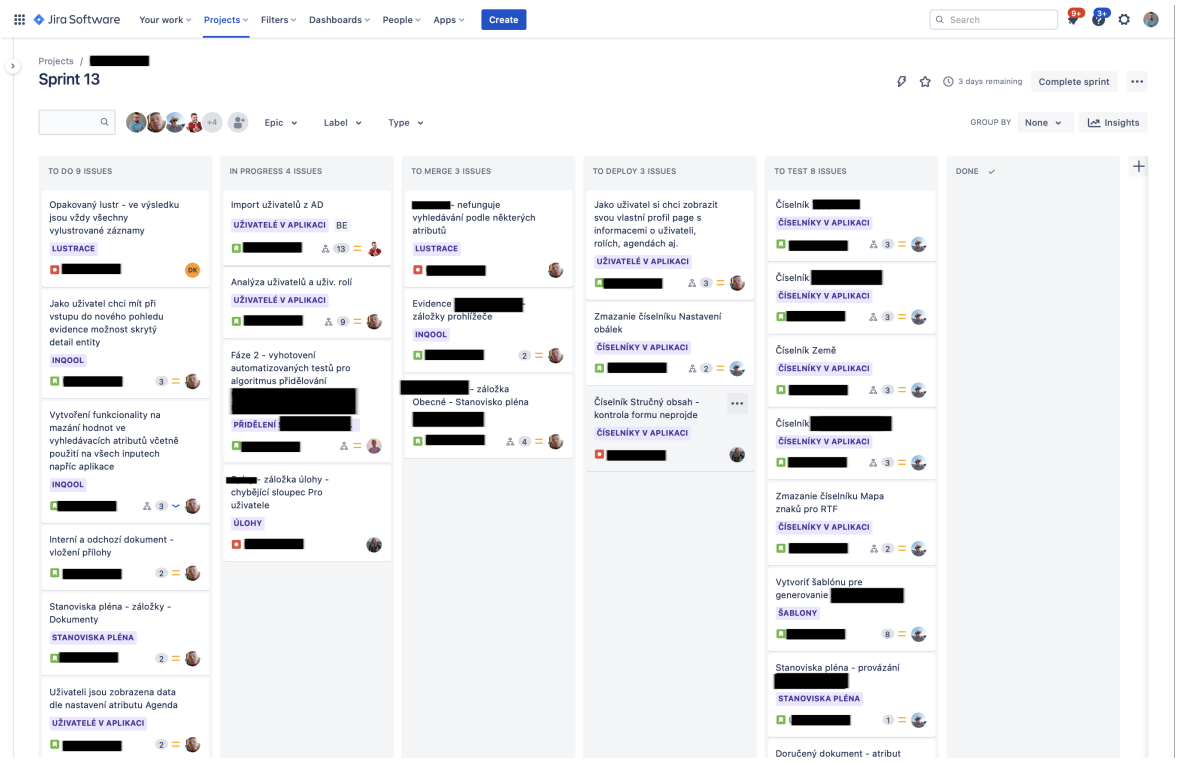
5.1.2 Používané nástroje

V rámci vývoje se používá hlavně Jira Software[2)] pro vizualizaci a popis všech částí projektu. Objevují se zde jak popisy jednotlivých položek z Backlogu, tak i jednotlivé výstupy z jednotlivých událostí. Také lze najít i celou analýzu a vizuální rozlišení dokončených či nedokončených částí.

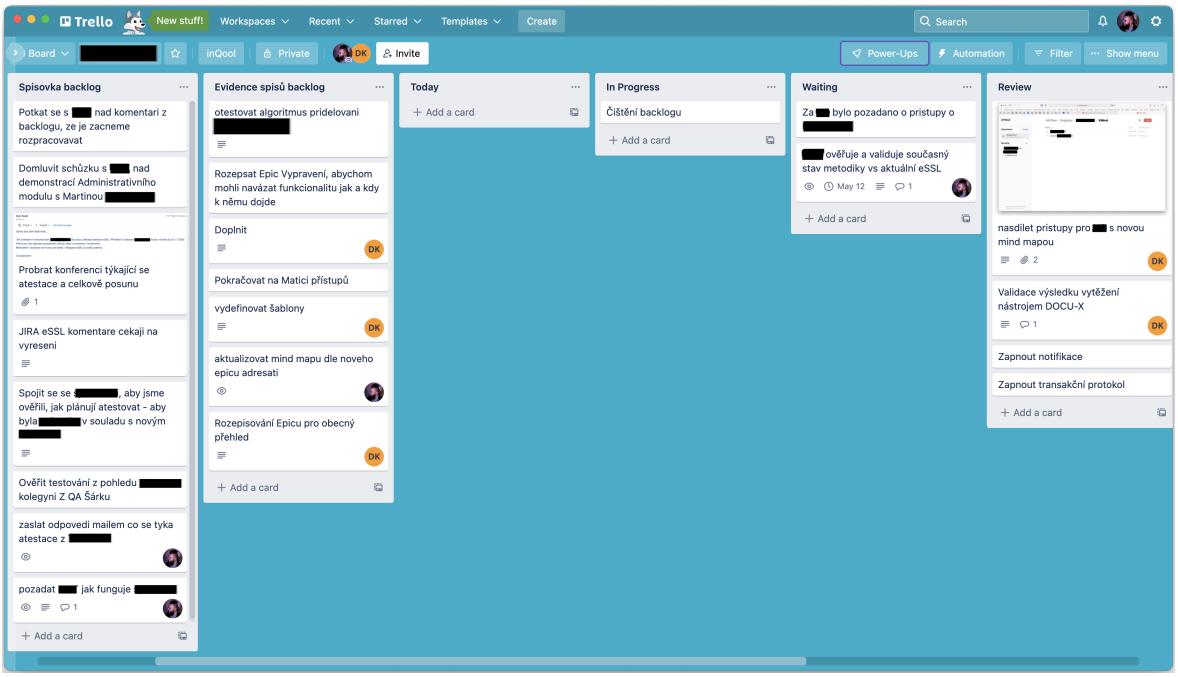
Jira funguje jako nástroj pro rozdělení práce a vizuální přehled v jakém stavu jsou jednotlivé položky, které je nutné udělat v rámci daného Sprintu. Ukázka využití Jira Software je ukázáno na obrázku 5.2.

Pro vizualizaci položek a jejich momentálního stavu řešení (například řeší se, čeká se atd.) se také využívá nástroj Trello[3)], který slouží spíše pro jednotlivé členy jako tzv. to do list. Ukázka používání tohoto nástroje je zobrazeno na obrázku 5.3.

[2)] Více informací o tomto softwaru na stránkách `https://www.atlassian.com/software/jira`.
[3)] Další informace o softwaru na stránkách `https://trello.com/`.



Obrázek 5.2 Ukázka vizualizace stavu a rozdělení položek ve Sprintu v Jira Software.



Obrázek 5.3 Ukázka vizualizace položek v Trello.

5.1.3 Nastavení Scrumu

Dosavadní vývojový proces je velmi inspirován Scrumem, který je popsán v části 2. Nicméně má své konkrétní nastavení pro jednotlivé události, například u Sprintu jsou následující pravidla:

- Konec Sprintu je v pondělí
  - Před ukončením Sprintu předchází Sprint Review se zákazníkem a následná Retrospektiva.
    - V případě, že se Sprint nestihne ukončit, je potřeba včasná komunikace Vývojového týmu, aby se mohl připravit zákazník na to, že Sprint nebude dokončen a položka Sprint Backlogu se tak vrátí do Product Backlogu.
    - Product Owner následně určí priority, zdali stejná položka bude zařazena do příštího Sprintu, či nikoli.
    - Začátek Sprint Review a Sprint Retrospektivy je ve stejný den jako konec Sprintu.
- Začátek Sprintu je v úterý.
  - V úterý začíná Sprint Planning s Vývojovým týmem (hodnocení pracnosti připravených položek).
- Nutno mít dopředu naplánováno alespoň 1 Sprint.
- Délka sprintu je 2 týdny.

Daily Scrum se označuje jako Stand up. Jedná se o povinnou událost pro všechny vývojáře na plný úvazek a pro studenty kvůli vyšší zátěži ve studiu je tolerována absence, nicméně pro dobro celého týmu je jejich přítomnost požadována. Stand up je v projektu nastaven takto:

- Každý den na 15 minut.
- Slouží zejména pro vývojový tým, jelikož se zavázal dodat Cíl Sprintu a Sprint Backlog je již naplánován a je srozumitelný.
  - V případě, že je potřeba hlubší diskuze, domluví se členové týmu, že budou po Stand upu pokračovat. Pokračování se nemusí účastnit členové, kterých se to netýká.

Pro plánování ve Sprint Planningu platí následující:

- Nejprve proběhne Planning s Product Ownerem, kde se vytvoří položky do Product Backlogu. Tyto položky se následně detailně popíší a nastaví se jejich priorita.
  - Pokud je vhodné k položce v Product Backlogu vytvořit názorný grafický diagram, tak je to doporučovaná varianta, jak položku popsat.
  - Sprint Planning s Product Ownerem může probíhat libovolně v průběhu Sprintu, aby byly připraveny položky Product Backlogu i na budoucí Sprinty (jak už bylo výše zmíněno, je nutné mít naplánován alespoň 1 Sprint dopředu).
- Následuje Planning s Vývojovým týmem, kde se jednotlivé položky Product Backlogu rozpadnou na jednotlivé úkoly a zároveň se odhadne jejich složitost.
  - Vývojový Tým se seznámí s obsahem, který se bude odhadovat nebo plánovat.
  - Planning Poker byl ze začátku hrán tradičním způsobem. Nicméně kvůli časté shodě odhadu a setření času byl zkrácen na hlasování, zda všichni souhlasí s navrhovaným ohodnocením.
    - V případě, že je položka Product Backlogu komplexní a není možné ji dostatečně odhadnout, vytvoří se issue na analýzu položky Product Backlogu a zařadí se do Sprintu Backlogu. Po analýze se bude položka Product Backlogu odhadovat znovu.
  - Do Sprintu Backlogu jdou pouze položky, které mají odhad složitosti.
- Nastaví se Cíl Sprintu, který se Vývojový Tým zaváže dodat v následujícím Sprintu.
- Délka Sprint Planningu s Vývojovým Týmem je 1 hodina, v případě, že se nedokončí, pokračuje se druhý den.

Ve vývojovém procesu se objevuje navíc interní Mock up Review, jenž slouží pro přípravu na Sprint Review a má následující pravidla:

- Proběhne ve čtvrtek před ukončením Sprintu pro demonstraci aktuálního stavu rozpracovaných položek Sprint Backlogu.

- Členové vývojového týmu se mohou připravit na Sprint Review a zároveň je možné včasně identifikovat nejasnosti.
- Členové Vývojového Týmu mají díky tomu i možnost seznámit se s aktuálním stavem Sprintu.
- Může sloužit jako možné otestování aktuálního stavu.
  - Zde by měl být přítomen kolega zodpovědný za interní testování.
- Délka interního Mock up Review je do 30 minut.

Sprint Review je poté nastaven následovně:

- Na konci Sprintu proběhne Sprint Review, kdy se vyberou vhodné položky User Story (může být i celá), funkcionality, Inkrementy, které se budou prezentovat zákazníkovi. Zákazník tak dostává prostor pro zpětnou vazbu.
  - Každý člen týmu demonstruje zákazníkovi svou část Inkrementu, na které pracoval.
  - Demonstrují se možné další rozšíření Inkrementu v dalším Sprintu, objasnění možného nepochopení a další.
- Po skončení Review následuje:
  - Nasazení Inkrementu do prostředí zákazníka.
    - Na konci každého Sprintu musí existovat Release Inkrementu.
- Vznik nebo aktualizace již existujícího popisu dodaného Inkrementu.
  - Nutno držet 1:1 Confluence page (stránka v nástroji Jira Software, která slouží k vytváření a organizování veškeré práce na jednom místě).
  - Slouží jako podklad pro dokumentaci, která bude vznikat v průběhu vývoje, a bude reflektovat podobu skutečného provedení.
    - Tímto způsobem se ušetří práce s tvořením analýzy až na konci. Což by nastalo, kdyby se do dokumentace nereflektoval skutečný aktuální stav. Dalším důsledkem by bylo, že se opomenou některé části.
  - Délka Sprint Review se zákazníkem je do 1 hodiny.

Jako poslední událost, která je specificky nastavená v projektu, je Sprint Retrospektiva:

- Cílem je identifikace akčních kroků a naplánování jejich realizace ideálně ihned v příštím Sprintu.
- Délka Sprint Retrospektivy je do 1 hodiny.

Jak už bylo zmíněno, Sprint Retrospektiva a Sprint Review jsou události, které nastávají na konci Sprintu a hned po nich nastává ve stejný den ještě interní Planning tak, aby v úterý mohl začít nový Sprint.

Dále v projektu existují 3 úrovně označení položek s tím, že v Product Backlogu se nachází převážně první a druhá úroveň:

- Epic - high level funkcionalita nebo modul.
  - např.: nový modul.
- User Story - nějaká podčást funkcionality nebo modulu.
  - např.: položka modulu - zpracované nebo nezpracované.
- Task - konkrétní definovaná část vytvořena Vývojovým Týmem při Sprint Planningu, vztahující se ke konkrétní User Story.
  - např.: přechod stavu z nezpracovaného do zpracovaného.

5.2 Návrh dotazníku

Dotazník byl vytvořen s cílem zjistit vnímání vývojového procesu z pohledu vývojářů. Otázky v dotazníku jsou rozděleny na čtyři části. První část zjišťuje vhodnost nastavení Scrumu a jeho procesů. Druhá oblast otázek se věnuje komunikaci v rámci týmu. Třetí skupina se zabývá možností zlepšení vývoje. Poslední otázky jsou otevřené, tak aby bylo možné zjistit případně věci, které předchozí otázky nepojaly. K získání odpovědí byl použit online nástroj Google Formulář.

5.2.1 Nastavení Scrumu

Tato sada otázek se zabývá délkou Sprintu. Zjišťuje se, jestli je délka Sprintu pro vývojáře vyhovující, zda během něho stíhají či nestíhají vypracovávat své úkoly. Jedna otázka se věnuje jak často se stává, že se položka musí vracet do Product Backlogu, jelikož se nestihne během Sprintu dokončit. Dále se zde nachází otázky, zda vývojářům vyhovuje nastavený den začátků a konce Sprintu.

V této oblasti se nacházejí otázky ohledně popisu položek. Ověřuje se, zda jsou položky dostatečně popsány, zda by se nehodila pro některé hlubší analýza a jak často se musí vytvářet issues pro analýzu položky v Product Backlogu.

Poslední dvě otázky v této části se zabývají časem stráveném na bugech, které se objeví během Sprintu a vnímání zvládnutí procesu Scrumu v dosavadní podobě z pohledu vývojářů.

5.2.2 Komunikace

V této sadě otázek se zjišťuje bezproblémovost komunikace. Otázky se zabývají přesněji komunikací Scrum Mastera s vývojovým týmem, vývojáře s Scrum Masterem, vývojáře s Product Ownerem a komunikací mezi vývojáři navzájem.

5.2.3 Možné zlepšení vývoje

Tato skupina otázek se zabývá, zda by s vývojem nepomohly některé praktiky z Extrémního programování. Konkrétněji se zjišťuje možnost využití párového programování při vývoji důležitých komponent, případně pro rychlejší zaučení nového člena. Další otázky se zabývají možným použitím praktik jako je zaměření se na standardy psaní kódu nebo programování řízené testy neboli psaní testů a až poté vlastního kódu.

5.2.4 Otevřené otázky

Poslední sadu tvoří dvě otevřené otázky. První zjišťuje, zda vývojář nemá nějaký jiný nápad na vylepšení vývojového procesu. Druhá otázka se zaměřuje na momentální nadbytečnost či neefektivnost ve vývojovém procesu.

5.3 Výsledky dotazníkového šetření

Odpovědi na otázky ohledně nastavení Scrumu jsou v tabulkách 5.1 a 5.2. Dále reakce na dotaz ohledně komunikace je v tabulce 5.3 a možného zlepšení vývoje v 5.4. Odpovědi na poslední otevřené otázky jsou v tabulce 5.5.

Výsledky z dotazníkového šetření poskytují další úhel pohledu na vývojový proces ze strany jednotlivých vývojářů.

Na základě odpovědí dotazníku je možné konstatovat, že komunikace v rámci týmu je pro vývojáře bezproblémová (viz tabulka 5.3) a délka Sprintu 2 týdny je z pohledu vývojářů v pořádku (viz odpovědi na první otázku v tabulce 5.1) . Dále na základě reakce na otázku, zda je Scrum v dosavadní podobě dobře zvládnut, se vývojáři kladně shodují, ale vnímají určité rezervy (viz poslední otázka v tabulce 5.1). Na základě ostatních odpovědí z otázek ohledně nastavení Scrumu je možné si všimnout, že se

Otázka	Vývojář				
	1	2	3	4	5
Jak dlouhý by podle tebe měl být Sprint?	min. 2, max. 3 týdny	2 týdny			
Jak často se ti stává, že při Sprintu nestíháš?	Občas		Často, ale není to každý Sprint	Výjimečně	
Jak často se stává, že máš hotovo ještě před koncem Sprintu?	Výjimečně	Nikdy	Výjimečně		Občas
Pomohla by u některých položek hlubší analýza?	Ne	Nevím	Ne		Nevím
Myslíš si, že dosavadní popis položek v Backlogu byl a je dostačující?	Ne, u většiny položek by mohla být lepší	Ano, u všech položek je dostačující	Ano, ale u některých položek by mohla být lepší		Ano, u všech položek je dostačující
Který den by ti více vyhovovalo končit Sprint?	Čtvrtek	Pátek	Pondělí		Úterý
Který den by ti více vyhovovalo začínat Sprint?	Pátek	Pondělí	Úterý	Pondělí	Středa
Jak si myslíš, že Scrum v dosavadní podobě je dobře zvládnut v rámci týmu?	Dobře, ale dalo by se ještě lépe	Z môjho pohladu "Líp už to zládnout nejde", ale fakt sa necítim kompetentný hodnotiť ako dobre je scrum zvládnutý	Dobře, ale dalo by se ještě lépe		

Tabulka 5.1 Odpovědi na otázky ohledně nastaveni Scrumu.

Otázka	Vývojář				
	1	2	3	4	5
Jak často se stává, že se nestihne vývoj položky v daném Sprintu a je nutné ho přesunout do Product Backlogu?	Často, ale ne vždy	Občas ano, občas ne	Každý Sprint	Výjimečné	Občas ano, občas ne
Jak často se stává, že je potřeba vytvářet issue pro analýzu položky v Product Backlogu?	Občas ano, občas ne	Výjimečné	Občas ano, občas ne	Výjimečné	
Kolik času trávíš na bugech, které se objeví v rámci Sprintu?	Zhruba polovinu času (50:50)	Málo času (75:25)			

Tabulka 5.2 Pokračování odpovědí na otázky ohledně nastaveni Scrumu.

Otázka	Vývojář				
	1	2	3	4	5
Myslíš si, že komunikace Scrum Mastera s vývojovým týmem by se měla zlepšit? Pokud ano, tak jak?	Ne, komunikace je bezproblémová				
Myslíš si, že tvoje komunikace se Scrum Masterem by se měla zlepšit? Pokud ano, tak jak?	Ne, komunikace je bezproblémová				
Myslíš si, že tvoje komunikace s Product Ownerem by se měla zlepšit? Pokud ano, tak jak?	Ne, komunikace je bezproblémová				
Myslíš si, že tvoje komunikace s ostatními ve Vývojovém Týmu by se měla zlepšit? Pokud ano, tak jak?	Ne, komunikace je bezproblémová				

Tabulka 5.3 Odpovědi na otázky ohledně komunikace.

Otázka	Vývojář				
	1	2	3	4	5
Myslíš si, že by ve vývoji pomohlo občasné párové programování?	Ano, ale jenom při rychlejším zaučením nového člena v týmu	Ne	Nevím	Ano, ale jenom při vývoji důležitých komponent	Ne
Myslíš, že by ve vývoji pomohlo větší důraz na standardy psaní kódu?	Ano	Máme podchytené dostatočne	Ne	Ano	Ne
Myslíš, že by ve vývoji pomohlo vytvářet nejdřív testy a až poté psát vlastní kód? Neboli používat programování řízené testy (Test-Driven Development)	Ne	Neviem, nemáme s tým ani skúsenosti, ani vôlu, čas, peniaze	Rád by som to vyskúšal. Avšak keďže pri Scrume hrozí menenie požiadavok, písanie testov pred písaním funkcionality a potom oboje prepisovať lebo zákazník niečo zabudol povedať / rozmyslel si, znie ako zbytočný pain.	Nevím	Ne

Tabulka 5.4 Odpovědi na otázky ohledně možného zlepšení vývoje.

Otázka	Vývojář	Odpověď
Máš nějaký jiný podnět/nápad, který by dle tebe mohl vylepšit vývojový proces?	3	Za mňa jednoznačne zapracovať na confluence. Chýba tam stále dosť veľa informácii. Pre mňa je to zdroj číslo 1, keď hľadám info o systéme. Viem, že sa teraz dosť info presúva do Jiry. Avšak mať miesto, kde to je všetko pokope a nie naprieč 10 user stories (pričom v starších môžu byť zastaralé info), by bola krása.
Je něco co ti připadá ve vývojovém procesu zbytečné, případě neefektivní?	1	Některé meetingy jsou málo strukturované a organizované, pálí se čas na zbytočnostech

Tabulka 5.5 Odpovědi na otevřené otázky.

vývojáři už tak silně neshodují. Což ukazuje na to, že tento vývojový proces je ve firmě nový a chce ještě nějaký čas na to, aby si vývojáři na tento vývojový proces více zvykli.

Největší neshodu je možné vnímat na otázky ohledně dne začátku a konce Sprintu (viz dvě předposlední otázky v tabulce 5.1), kdy se zjišťovalo, zda by pro vývojáře nebyl vhodnější nějaký jiný den. Zde se vývojáři svými odpovědmi rozprostřeli téměř po celém týdnu. Takže se nedá usoudit, zda by nějaké jiné dny byly více vyhovující než dosavadně nastavené.

Další nesoulad v tabulce 5.1 je ohledně vnímání popisu položek v Backlogu, což může znamenat odlišnost zkušeností jednotlivých vývojářů a jejich míru vnímání dobře popsané položky. Na druhou stranu je dobré, že vývojáři nevnímají, že by se velmi často musela vytvářet issues pro analýzu položky (viz předposlední otázka v tabulce 5.2). Bohužel tuto skutečnost není možné doložit na datech, jelikož nebylo možné tyto data získat.

Kvůli absenci reportu a velmi odlišném pohledu vývojářů se nedá ověřit, jak moc se v jednotlivých Sprintech stíhá vývoj naplánovaných položek a jak často se stává, že se musí položka přesunout zpět do Product Backlogu (viz druhá a třetí otázka v tabulce 5.1 a první v 5.2.

Pozitivem vývojového procesu je to, že se dle odpovědí na poslední otázku v tabulce 5.2 vývojáři netráví příliš mnoho času na bugech, které nastanou během Sprintu. Bohužel opět nebylo možné tento fakt ověřit na datech, jelikož nebyla dostupná.

Ohledně možného návrhu na zlepšení pomocí nějaké praktiky extrémního programování v tabulce 5.4 je možné říci, že vývojáři záporně reagovali na používání programo-

vání řízení testy. Konkrétně z odpovědí vyplývá, že daná praktika není pro tento způsob vývoje vhodná a někteří s ní nemají moc zkušeností. Naopak velmi odlišné odpovědi byly získány na možnost většího důrazu na standardy psaní kódu. Důvodem mohou být odlišné zkušenosti jednotlivých vývojářů a jejich vnímání kódu napsaného jinou osobou. Na možnost využití párové programování se opět vývojáři lišili. Příčina odlišnosti odpovědí může být opět dána rozdílnou zkušeností jednotlivých vývojářů a také možná nedostatečná znalost této techniky. Na základě těchto odpovědí tedy není možné navrhnout optimalizaci pomocí některé praktiky extrémního programování. Je však možné doporučit nějaké školení na párové programování a po něm znovu vyhodnotit, zda by tato technika nemohla být aplikována při nějakých specifických případech.

Na otevřené otázky odpověděli pouze dva z pěti vývojářů (viz tabulka 5.5). Na dotaz týkající se nápadu na zlepšení vývojového procesu reagoval respondent 3 zapracováním psaní dokumentace v Confluence (viz podkapitola 5.1.3). Nedodržování nastaveného způsobu držení dokumentace na 100 % je pro tohoto vývojáře možná důvod, proč jenom výjimečně stíhá před ukončením Sprintu a zároveň často nestíhá během něho. Daný respondent jako jediný vnímá, že se při každém Sprintu nestihne vývoj položky a musí se tak přesunout do Product Backlogu. Nicméně se jedná pouze o reakci jednoho vývojáře. Ostatní vývojáři tento problém nemají nebo se o něm nezmiňují. Je tedy možné, že se tento problém časem zlepší, až si vývojáři na tento způsob vývoje více zvyknou.

Na druhou otevřenou otázku týkající se případné zbytečnosti či neefektivnosti vývojového procesu reagoval respondent 1. Tomuto vývojáři připadá, že některé meetingy jsou málo strukturované a organizované a tím se ztrácí čas na zbytečnostech. Opět se jedná o reakci jednoho vývojáře a i zde je možné, že se časem i tento prvek zlepší.

Na základě obou odpovědí na otevřené otázky je možné jenom doporučit, aby se vývojáři nebáli dávat feedback Scrum Masterovi během retrospektivy, pokud tedy tak už nečiní. Za pomoci otevřené zpětné vazby je možné tyto drobné nedostatky z pohledu těchto dvou vývojářů lehce upravit a časem i zcela odstranit.

5.4 Závěr a možné vylepšení procesu vývoje

Dle popisu vývojového procesu a jeho vnímání pěti vývojářů, kteří se ho aktivně účastní, je možné označit dosavadní vývojový proces jako vhodný pro použití i v jiných projektech. Drobné problémy, které byly zjištěny za pomoci dotazníku, by se měly postupem času eliminovat až by si vývojáři na tento způsob vývoje ještě více zvykli.

Nicméně kvůli velké absenci dat, které by ukázaly v číslech, jak si dosavadní vývojový proces vede, je vhodné implementovat zaznamenávání některých údajů sloužících pro porovnání, jak si vede vývoj v daném projektu. K tomuto účelu může posloužit už

používaný nástroj Jira Software, který umožňuje dokonce vytvářet vlastní reporty, popřípadě lze zvolit jiný vhodný nástroj, který by firmě více seděl.

Doporučuje se používat minimálně tyto reporty:

- Počet položek vrácených do Product Backlogu po Sprintu - tento report by měl pomoci sledovat, zda je průměrný dosavadní počet story pointů na Sprint ten správný nebo zda by se neměl snížit.
- Počet naplánovaných položek ku dokončeným položkám - toto sledování by opět mohlo pomoci sledovat, zda je množství naplánované práce správně odhadnuto či nikoli.
- Počet bugů objevených během Sprintu - na základě těchto výsledků by se mohlo zjišťovat, jak kvalitně programátoři vytváří své položky.
- Počet issues pro analýzu položky v Procuct Backlogu - tento report by umožňoval dávat zpětnou vazbu, zda Product Owner vhodně vytváří položky do Product backlogu.

Kvůli časté přítomnosti studentů by bylo zajímavé sledovat, jakým způsobem se ovlivňuje míra hotové práce, když mají studenti výuku a když mají zkouškové období. Tento report by mohl pomoci, jak lépe s tímto výpadkem pracovníků pracovat a třeba na toto období plánovat menší počet story pointů. Toto sledování by mohlo probíhat na základě reportu hotových položek ku naplánovaným a porovnávat to s kapacitou programátorů za stejné období.

Do budoucna by se mohl vývojový proces ještě vylepšit tím, že se bude používat Scrum, kde se jednotlivé Sprinty prolínají, jak je blíže popsáno v teoretické části 2.7. Nicméně pro tuto optimalizaci je nutné dobře zvládnout dosavadní podobu Scrumu a vývojáři musí být zvyklí na tento způsob vývoje.

Vylepšení procesu vývoje za využití praktik extrémního programování se ukázalo dle pohledu vývojářů jako nepoužitelné. Programování řízení testy bylo programátory negováno kvůli agilnímu způsobu vývoje, kdy hrozí přepisování funkcionalit. Další navrhovaná praktika, důraz na standardy psaní kódu, neměla jednoznačný výsledek. Důvodem mohou být odlišné zkušenosti a míra čtení cizích kódů u jednotlivých vývojářů. Poslední uvažovaná praktika taktéž neměla jednotný výsledek. Jednalo se o párové programování. I zde mohl být vliv odlišných zkušeností a možná i neznalost této praktiky. Možná by stálo za zvážení, zda by se vývojáři neměli vzdělat v dané praktice a následně porovnat možné využití v nějakých specifických případech.

6 DOKUMENTACE PŘEVZATÉHO ENGINU

Při přebírání zaběhnutého projektu byl přebrán i engine, který je nazván jako CML Engine. Tento engine je vytvořen za použití Javy a Groovy. Kvůli tomu, že engin nebyl předán s dostatečnou dokumentací, je možné oficiální význam zkratky CML odhadovat jenom na základě písmen a funkcionality. Na základě toho je možné, že písmeno C znamená Control případně Command. Dále M může znamenat Manager/Management a písmeno L by mohlo označovat Layout nebo Layer. Nicméně jedná se o engine, který kontroluje zobrazení, dostupnost a vlastnosti jednotlivých komponent v jednotlivých formulářích.

Engine využívá pro kontrolu zmíněných formulářů soubory s příponou grl. I zde je možné vést diskuze ohledně významu této zkratky. Jelikož se v nich využívají Groovy funkce pro vytváření skript za běhu a engine je rozdělený do dvou modulů nesoucí názvy `cml` a `grools`, mohli tedy původní autoři mít na mysli označení Grools neboli akronym pro Groovy Rules. Tento předpoklad byl ještě podpořen faktem, že u Javy existuje podobný Businees Rules Management System nazvaný Drools, jenž má na první pohled podobné prvky (více o tomto systému na stránkách `www.drools.org/`).

Kvůli nedostatečné dokumentaci, můžeme o přesném názvu vést spory, můžeme s tím nesouhlasit, ale to je asi tak všechno, co se s tím dá dělat, jak by řekl klasik - Jára Cimrman.

6.1 Popis struktury enginu

Jak už bylo zmíněno dříve, engine je rozdělen do dvou modulů. Kvůli anonymizování projektu bude popsán hlavně modul `grools` a pro popis modulu `cml` se použijí jen příklady využití, tak aby bylo možné pochopit důležitost a význam popisovaného enginu.

6.1.1 Modul `grools`

V modulu `grools` se nachází funkcionalita pro parsování grl souborů a vytváření potřebných objektů pro Groovy skripta. Tento modul je rozdělen na dvě hlavní části. První část je označena stejně jako modul samotný, tedy `grools`. Nyní budou více popsány balíčky a v nich třídy, rozhraní, anotace a výčty nacházející se v první části modulu

- `api`
  - `GrlValidator.java` (rozhraní)
  - `GroolsEngine.java` (rozhraní)
  - `GroolsMultiInputOutput.java` (rozhraní)

  - `GroolsTuple.java` (rozhraní)
  - `GroovyClassLoaderProvider.java` (rozhraní)
  - `JavaScript.java` (anotace)
  - `ReturnObject.java` (třída)

- `engine`
  - `DefaultGroolsEngine.java` (třída)
  - `GroolsMultiInputOutputImpl.java` (třída)
  - `PreprocessorExtension.java` (rozhraní)
  - `PreprocessorExtensionRegistry.java` (rozhraní)
  - `SimpleGroolsClassLoaderProvider.java` (třída)

- `factory`
  - `DebugLevel.java` (výčet)
  - `DefaultPreprocessorExtensionRegistry.java` (třída)
  - `DefaultScriptFactory.java` (třída)
  - `GrlScriptClassLoader.java` (třída)
  - `GroupPreprocessorExtension.java` (třída)
  - `LoadableScriptFactory.java` (rozhraní)
  - `ScriptFactory.java` (rozhraní)
  - `VoidPreprocessorExtension.java` (třída)

- `fuction`
  - `ClosureFunction.java` (třída)
  - `FunctionDelegate.java` (třída)
  - `JavaScriptFunction.java` (třída)

- `lang`
  - `script`
    - `GrlScript.java` (abstraktní třída)
  - `CustomClosureMetaclass.java` (výčet)
  - `GrlCodeSource.java` (třída)

  - `GrlContext.java` (třída)
  - `GrlContextContainer.java` (rozhraní)
  - `GrlParserPlugin.java` (třída)
  - `PojoWithDynamicProperties.java` (rozhraní)
  - `ReturnObjectMetaClass.java` (třída)
  - `SourcePreProcessor.groovy` (třída)

- `source`
  - `AbstractScriptSource.java` (abstraktní třída)
  - `ClassPathScriptSource.java` (třída)
  - `FilesystemScriptSource.java` (třída)
  - `ScriptClass.java` (třída)
  - `ScriptDescription.java` (třída)
  - `ScriptSource.java` (rozhraní)

- `validation`
  - `ClosureGrlValidator.java` (třída)
  - `JavaScriptGrlValidator.java` (třída)
  - `ReturnObjectMaker.groovy` (třída)
  - `ValidatorWithJavaScriptSupport.java` (rozhraní)

Jak už názvy jednotlivých balíčků naznačují, tak v `api` se nachází rozhraní pro programování aplikací. Zde se kromě rozhraní nachází i třída `ReturnObject.java`, která ukládá Java objekty do mapy podle jejich textového označení a umožňuje ukládat `GrlValidator` pro případnou validaci.

Balíček `engine` obsahuje třídy a rozhraní, které se používají pro pohánění celé funkcionality. Zde se nachází třída `GroolsMultiInputOutputImpl.java`, která implementuje rozhraní `GroolsMultiInputOutput.java`. Tato třída je určená pro uložení tzv. globálního kontextu, který je možné chápat jako údaje už existující v databázi nebo změněné/zapsané na základě formuláře. Dále umožňuje uložit tzv. lokální kontext, což je možné chápat jako udaje, které jsou získané z grl souboru. Za zmínku také stojí `DefaultGroolsEngine.java`, který obohacuje `GrlScript` o proměnné za pomoci Binding a následně ohodnotí tyto proměnné (dalo by se říci pravidla) spuštěním požadovaného skriptu. Ještě se zde nachází třída `SimpleGroolsClassLoaderProvider.java`, která implementuje `GroovyClassLoaderProvider` pro načítání Groovy tříd.

Třetí balíček `factory` obsahuje hlavně třídu `DefaultScriptFactory.java`, která implementuje rozhraní `LoadableScriptFactory` a `ScriptFactory`. Tato třída umožňuje získat Java objekty ve formě `GrlScript`, jenž se vytváří za pomoci `ScriptClass`, které vznikají při parsování Groovy skriptu. Tyto Groovy skripty se tvoří z textové podoby obsahu grl souboru. Třídy `DefaultPreprocessorExtensionRegistry.java`, `GrlScriptClassLoader.java`, `VoidPreprocessorExtension.java` a ještě poslední nezmíněná `GroupPreprocessorExtension.java` fungují jako preprocesory pro úpravu dat textového obsahu z grl souboru.

Dalším v pořadí je balíček `function`, ve kterém se nacházejí třídy pro vytvoření funkcí z grl souboru. Například třída `ClosureFunction.java` umožňuje textovou podobu funkce v grl souboru zaobalit jako Closure. S touto třídou souvisí i další třída a to `FunctionDelegate.java`, která se používá pro určení delegátů při použití vzniklého Closure. Podobný princip zaobalení funkce má i `JavaScriptFunction.java`, která ale umožňuje funkce vykonávat jako JavaScript.

V balíčku `lang` se nacházejí převážně třídy, které se používají pro uložení textového obsahu grl souborů. Pro uložení obsahu jako obyčejný text se používá třída `GrlCodeSource.java`. `GrlContext.java` umožňující uložit proměnné jako Binding nebo jako `ReturnObject`. Třída `GrlScript.java` v podstatě definuje volání daného grl skriptu, který se do této třídy uloží ve formě `GrlContext`. Dále se v balíčku objevuje `ReturnObjectMetaClass.java`, která umožňuje spravovat volání objektů Groovy do POJO (Plain old Java object). Poslední dvě nezmíněné třídy spolu souvisí, protože groovy třída `SourcePreProcessor.groovy` nastavuje Java třídu `GrlParsePlugin.java`, jenž nahrazuje podobu hodnoty záznamů map grl souboru do hodnoty v Closure.

Předposlední balíček `source` je určen jako zdroj grl souborů a jejich načítání buď z cache, nebo ze souboru. Pro definování skript v cache se používají dvě třídy. Jedná se o `ScriptClass.java` a `ScriptDescription.java`. Druhá zmíněná třída slouží jako klíč pro cache `ScriptClass` objektů. Pro získání textového obsahu grl skriptu se používá abstraktní třída `AbstractScriptSource.java`, která je dále implementována u `ClassPathScriptSource.java` a `FilesystemScriptSource.java`. První z těchto implementací používá cestu k souboru a druhá File.

Poslední balíček v této části modulu je `validation`. Jak už tento název napovídá, jedná se o třídy určené pro validaci dat, které byly vloženy třeba za pomoci formuláře. Pro validaci za využití Closure se používá `ClosureGrlValidator.java` a použití JavaScriptu umožňuje `JavaScriptGrlvalidator.java`. Poslední třída v balíčku je Groovy třída `ReturnObjectMaker`, která umožňuje propojit další údaje s již existujícím objektem za běhu.

Druhá část modulu `grools` se skládá z více souborů, proto je její označení trochu

delší. Název této části je `groovy.runtime.metaclass.grools.lang.script` a nachází se zde pouze jedna třída a to `CustomGrlMetaclass.java`. Tato třída funguje podobně jako `ReturnObjectMetaClass` až na to, že místo `ReturnObject` používá Java objekt `GrlScript`.

6.1.2 Modul `cml`

Ve druhém modulu `cml` už je specifická implementace enginu, která se využívá v projektu a nacházejí se zde i všechny soubory grl.

V modulu se nachází třída `CmlEngineBean.java`, která při spuštění inicializuje načtení grl souborů uložených v cache a jejich následné uložení po ukončení.

Dále se zde vyskytují třídy a metody pro různé použití grl souborů. Kvůli zachování anonymity projektu, bude tato specifická implementace ukázána na příkladech za použití vymyšlených hodnot.

6.2 Ukázka funkcionality

Pro ukázku funkcionality se bude používat smyšlený příklad, na kterém bude ukázána syntaxe a možnosti využití grl souborů.

Jako příklad bude sloužit oční ordinace, kde bude několik formulářů. V ukázce se budou používat dvě hlavní třídy Pacient a Oftalmolog. Dále se bude používat třída Diagnoza, která bude mít tyto výčtové hodnoty: myopie, hypermetropie a emetropie. Druhý výčtový typ bude i Korekce, kde bude operace, brýle a čočky.

Za pomoci souboru grl lze definovat, že Pacient a Oftalmolog budou mít stejné proměnné na vyplnění při registraci. Může se jednat o jméno, příjmení, datum narození, telefonní číslo a další informace o osobě. Některé proměnné budou speciálně pro Pacienta, jako diagnóza, refrakce nebo možnosti korekce oční vady. Oftalmolog zase bude mít hodinovou mzdu nebo počet hodin dovolené.

Ukázka daného grl souboru pro získání potřebných tříd při registraci je ukázán na obrázku 6.1. Zde je možné si všimnout, že za pomoci # lze psát poznámky do grl souboru. Dále je zde vidět funkce pro spuštění daného příkazu *when()*, která může mít v závorkách mapu (klíč:hodnota) pro určení, za kterých podmínek lze provést tento příkaz. Znak + určuje přidat komponentu. Na druhou stranu − by znamenal odebrat komponentu. Hodnoty v hranatých závorkách se přidávají do skriptu a v podstatě se jedná o Closure, který je blíže popsán v části 4.2.

Pro úpravu výpisu registrované osoby podle toho jestli se bude jednat o Pacienta anebo Oftalmologa, je možné využít vypis.grl zobrazený na obrázku 6.2. Zde je možné si všimnout, že pomoci znaku @ se definují proměnné. Takže na výpisu u pacienta bude v titulku uvedeno „Výpis karty pacienta“ a u oftalmologa „Výpis zaměstnanecké karty“.

```
1  #- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
2  # CML DEFINICE: Registrační formulář - Určuje, které údaje budou pro
     Pacienta, a které pro Oftalmologa při registraci do systému.
3  # filename: registrace.grl
4  #- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
5  #Pro obě osoby získáme třídu pro Osobní udaje
6  when()
7  {
8      + OSOBNI_UDAJE
9  }
10 # Jenom pro pacienta získáme navíc třídy Diagnoza a Korekce
11 when(osoba: "pacient")
12 {
13     + DIAGNOZA
14     + KOREKCE
15 }
16 # Jenom pro oftalmologa navíc získáme třídu Mzda a Dovolena
17 when(osoba: "oftalmolog")
18 {
19     + MZDA
20     + DOVOLENA
21 }
```

Obrázek 6.1 Ukázka grl souboru: získání tříd k vyplnění ve formuláři při registraci.

Další příklad využití grl souborů je znázorněn na obrázku 6.3. Zde se zjišťuje, která diagnóza je nutná proto, aby se mohla provést laserová operace. Zda se může provést operace je definováno v proměnné laser, kde hodnota 1 znamená, že se může provést, a hodnota 0 naopak nemůže. Jedná se o ekvivalent True a False.

Pro ukázku jak implementovat požadovanou operaci s grl souborem v Javě lze využít poslední zmíněný soubor laser.grl. Na obrázku 6.4 je ukázána metoda diagnozyProLaserovouOperaci, která má vstupní parametry získané z formuláře (1). Nejdříve v metodě získáme všechny diagnózy, které máme k dispozici (3). Následuje vytvoření proměnné typu GroolsMultiInputOutput, která v sobě bude mít uložené proměnné z formuláře jako globální a všechny diagnózy jako lokální proměnné (4) Tato proměnná bude fungovat jako vstupní proměnná při vyhodnocení skriptu. Vytvoření této třídy je ukázáno na obrázku 6.5. Ukládání proměnných je prováděno za pomoci třídy Binding, tak aby se mohly snadno přiložit ke skriptu.

Dále se použije DefaultGroolsEngine pro vyhodnocení souboru laser.grl a získají se tak výstupní hodnoty, které byly obohaceny vykonáním skriptu (6). Poté se za pomoci for cyklu zjišťuje, které diagnózy mají přidanou proměnnou laser s hodnotou 1 (8-21). Textová podoba těchto proměnných se poté uloží do pole. Aby se získaly

```
#- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# CML DEFINICE: Výpis osoby - Určuje, které údaje budou pro Pacienta a
 které pro Oftalmologa při výpisu ze systému.
# filename: vypis.grl
#- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
when()
{
    @hlavicka = "Výpis osoby"
}
when(osoba: "pacient")
{
    @titulek = "Výpis karty pacienta"
}
when(osoba: "oftalmolog")
{
    @titulek = "Výpis zaměstnanecké karty"
}
```

Obrázek 6.2 Ukázka grl souboru: úprava výpisu dle typu osoby.

```
#- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# CML DEFINICE: Vhodná diagnóza pro laserovou operaci
# filename: laser.grl
#- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# Laserová operace je vhodná pro pacienty, kteří mají nějakou refrakční vadu
 when(korekce: "operace", diagnoza: ["myopie", "hypermetropie"])
{
    @laser = 1
}
# Laserová operace je zbytečná u pacienta, který nemá refrakční vadu
 when(korekce: "operace", diagnoza: "emetrop")
{
    @laser = 0
}
```

Obrázek 6.3 Ukázka grl souboru: při jaké diagnóze je možná laserová operace.

plnohodnotné proměnné diagnóz, vytvoří se další for cyklus ze získaných diagnóz ze začátku. Poté se bude zjišťovat, zda se textová podoba diagnózy nachází v textovém poli získaném v předešlém for cyklu (23-27). Získaný list diagnóz je výstupem metody diagnozaProLaserovouOperaci.

```
1  List<Diagnoza> diagnozyProLaserovouOperaci(ParametryZFormulare
   parametryZFormulare)
2  {
3      Collection<Diagnoza> diagnozy = získáme všechny možné diagnózy
4      GroolsMultiInputOutput input = bindingFac-
   tory.createGroolsInputForDiagnozyProLaser(parametryZFormulare,
   diagnozy);
5      GroolsEngine engine = new DefaultGroolsEngine()
6      GroolsMultiInputOutput output = engine.evaluateRules(cesta k grools
   souborům, laser.grl, input);
7      Set<String> mozneDiagnozy = new HashSet<>();
8      for (GroolsTuple singleOut : output.getLocalContexts())
9      {
10         boolean moznaOperace = false;
11         ReturnObject ret = singleOut.getReturnObject();
12         Object value = ret.getProperty("laser");
13         if (value != null)
14             moznaOperace = value.toString().equals("1");
15         if (moznaOperace)
16         {
17             String diagnozaKey = (String)
18             singleOut.getBinding().getVariable("diagnoza"));
19             mozneDiagnozy.add(diagnozaKey);
20         }
21     }
22     List<Diagnozy> returnList = new ArrayList<>();
23     for (Diagnoza diagnoza : diagnozy)
24     {
25         if (mozneDiagnozy.contains(diagnoza.getKey()))
26             returnList.add(diagnoza);
27     }
28     return returnList;
29 }
```

Obrázek 6.4 Implementace Java metody pro získání vhodných diagnóz pro laserovou operaci za použití souboru laser.grl.

```
1 GroolsMultiInputOutput createGroolsInputForMozneDiagnozyProLaserovou-
    Operaci(ParametryZFormulare parametryZFormulare, Collection<Diagnoza>
    diagnozy)
2 {
3 // Uložíme si hodnoty z formuláře jako globální hodnoty, v tomto
    případě diagnózu a korekci
4     Map<String, Object> globalMap = new HashMap<String, Object>();
5     globalMap.put("diagnoza", parametryZFormulare.getDiagnoza.getKey())
6     globalMap.put("korekce", parametryZFormulare.getKorekce.getKey())
7 // uložíme si všechny možné diagnózy do lokálních hodnot, podobně
    jako jsme si uložili globální hodnoty
8     for (Diagnoza diagnoza : diagnozy)
9     {
10        Map<String, Object> map = new HashMap<String, Object>();
11        map.put("diagnoza", parametryZFormulare.getDiagnoza.getKey())
      // uložíme všechny diagnózy do Binding, tak aby se mohly
      vložit do skriptu
12        Binding binding = new Binding(map);
13        List<GroolsTuple> localTuples = new ArrayList<>();
14        localTuples.add(new GroolsInputTuple(binding));
15    }
16 // Globální hodnoty zaobalíme do Binding, tak aby se mohly vložit
    do skriptu
17    Binding globalBinding = new Binding(globalMap);
18    GroolsTuple globalTuple = new GroolsInputTuple(globalBinding);
19    GroolsTuple[] localTuplesArray = localTuples.toArray(new GroolsTuple[]
    );
20    return new GroolsMultiInput(globalTuple, localTuplesArray);
21 }
```

Obrázek 6.5 Implementace Java metody pro získání vstupních hodnot pro grl skript.

Poslední ukázka použití grl souboru je na obrázku 6.6. Zde je možné vidět, že v části pro vytváření Closure (neboli hranaté závorky) je možné definovat error, případně warning nebo info, jako validaci hodnot získaných ve formuláři. Přitom error, warning a info určují závažnost chyby. Po definování závažnosti následuje textová hodnota pro výpis chyby. Následuje další Closure, kde se vyhodnocuje zda se jedná o chybu nebo ne.

Pomocí příkazu eval() je možné spouštět v grl souboru jiný grl soubor. Tímto se dají vnořovat různá pravidla a udržet kód čitelný bez duplikací kódu.

```
1 #- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
2 # CML DEFINICE: Další ukázky použití grl souboru - validace a spouštění
    dalšího grl souboru.
3 # filename: ukazky.grl
4 #- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
5 when(udaj: "osobni_udaje")
6 {
7       error "Nebylo vyplněno jméno", v.getName() != null
8 }
9 when(udaj: "diagnoza")
10 {
11      eval("validace_diagnoza.grl") }
```

Obrázek 6.6 Ukázka grl souboru: příklad pro validaci a spouštění jiného grl souboru.

6.3 Shrnutí

Největší výhodou engine je, že umožňuje použití spouštění funkcí za běhu pomocí Groovy a zároveň využívat Java třídy. Na základě tohoto faktu je možné CML engine nazvat jako velmi variabilní nástroj, který umožňuje používat grl soubory nejrůznějšími způsoby. Ideální použití je pro dynamické prostředí. Engine je však možné používat jen jako validační nástroj pro vstupní hodnoty nebo jen jako nástroj pro získávání hodnot na základě vstupu.

Kvůli velké abstrakci, která je způsobena používáním Binding (přidání hodnot do proměnných) a Closure (vytváření funkcí do skript), je nutné při vytváření nových grl souboru udržet alespoň nějakou míru dokumentace, z důvodu rychlého nalezení chyb v případě opravy. Minimálně by se měly udržovat vhodné názvy jednotlivých grl souborů, tak aby odpovídaly obsahu. Dále v hlavičce souboru je vhodné dokumentovat popis funkcionality a k čemu daný grl soubor slouží.

Na druhou stranu za pomoci vnořování jednotlivých grl souborů do sebe je možné udržet kód dle zásad čistého kódu, a tím zlepšit čitelnost a případnou opravitelnost kódu.

ZÁVĚR

Cílem diplomové práce bylo zkoumat, zda je možné ve firmě, kde převažuje vodopádový model pro vývoj softwaru, používat agilní způsob vývoje, a zda je možné díky analyzovanému ukázkovému projektu dosáhnout možných vylepšení, které by se staly základem optimalizovaného způsobu vývoje software pro další budoucí projekty. Sekundárním cílem bylo navrhnout dokumentaci převzatého enginu pro efektivnější práci vývojářů.

Pro zjištění vhodnosti agilního procesu vývoje byl použit převzatý projekt, kde se využívá vývojový proces založený na Scrumu. Na základě jeho analýzy a pohledu zainteresovaných vývojářů, bylo zjištěno, že je možné agilní způsob vývoje využít i na jiných projektech v rámci firmy. Nicméně kvůli absenci dat nebylo možné ověřit účinnost tohoto vývojového procesu. Proto bylo doporučeno implementovat monitorování údajů jako je počet položek vrácených do Product Backlogu po skončení Sprintu, počet naplánovaných položek ku dokončeným položkám, počet bugů objevených během Sprintu a počet issues pro analýzu položek v Product Backlogu. Díky těmto reportům by bylo možné lépe porovnávat a vyhodnocovat úspěšnost popsaného vývojového procesu i s jinými metodami používanými ve firmě. Kvůli velké přítomnosti studentů bylo navrženo sledovat i rozdíly pracovní docházky ku hotové práci během zkouškového období a výuky na vysokých školách. Tyto výsledky by mohly napomoci k lepší predikci a plánování jednotlivých Sprintů. Na základě pohledu vývojářů nebylo možné navrhnout další optimalizaci daného vývojového procesu některými praktikami z Extrémního programování. Například programování řízené testy vývojáři odmítli. Důvodem odmítnutí je agilní způsob vývoje, kde je možné přepisovat hotové funkcionality. Přepisování kódu a testů by tak zabralo více času. Další navrhované praktiky jako je důraz na psaní kódu a párové programování nedostaly jednoznačné výsledky. Důvodem mohou být odlišné zkušenosti jednotlivých programátorů. Proto bylo doporučeno zvážit školení na párové programování. Následně zkusit vyhodnotit, zda by tato technika nemohla být využita v některých případech jako je například rychlejší zaučení nového člena týmu nebo při vývoji důležitých komponent.

Dalším cílem práce bylo vytvořit dokumentaci CML enginu, která nebyla spolu s projektem převzata. Důležitost tohoto enginu tkví v umožnění spouštět funkce za běhu za pomoci Groovy a zároveň používat Java třídy. Z tohoto důvodu CML engine vytváří variabilní nástroj, který používá grl soubory nejrůznějšími způsoby, jako například validace vstupních hodnot nebo získání hodnot na základě vstupu. Proto je vhodné tento nástroj využívat v dynamickém prostředí. Nicméně kvůli velké míře abstrakce, která je způsobená hlavně používáním groovy třídy Closure (vytváření funkcí do skript) a Binding (přidávání hodnot do proměnných), bylo doporučeno při vytváření nových grl souborů držet vhodnou úroveň dokumentace. Příkladem této dokumentace je vhodný

název a popis funkcionality na začátku grl souboru. Na druhou stranu grl soubory umožňují vnořování do dalších grl souborů, a tím je možné držet kód dle zásad čistého kódu a zlepšit tak čitelnost a případnou opravitelnost. Tato dokumentace CML enginu byla vytvořena pro lepší pochopení jeho funkcionality a významu v převzatém projektu, tak aby vývojáři s tímto enginem měli efektivnější práci.

SEZNAM POUŽITÉ LITERATURY

[1] *Vodopádový model: Vodopádový model životního cyklu software (The waterfall Life cycle)* [online] [cit. 2022-04-16]. Dostupné z: `http://testovanisoftwaru.cz/manualni-testovani/modely-zivotniho-cyklu-softwaru/vodopadovy-model/`.

[2] SOMMERVILLE, Ian. *Softwarové inženýrství.* 1. vyd. Brno: Computer Press, 2013. ISBN 978-80-251-3826-7.

[3] BRUCKNER, Tomáš; VOŘÍŠEK, Jiří; BUCHALCEVOVÁ, Alena; STANOVSKÁ, Iva; CHLAPEK, Dušan; ŘEPA, Václav. *Tvorba informačních systémů: principy, metodiky, architektury.* 1. vydání. Praha: Grada, 2012. ISBN 978-80-247-4153-6.

[4] KOĎOUSKOVÁ, Barbora. *Co je agilní vývoj aplikací a kdy jej využívat* [online]. Rascasone s.r.o., 2021 [cit. 2021-05-21]. Dostupné z: `https://www.rascasone.com/cs/blog/co-je-agilni-vyvoj`.

[5] BECK, Kent; BEEDLE, Mike; BENNEKUM, Arie van; COCKBURN, Alistair; CUNNINGHAM, Ward; FOWLER, Martin; GRENNING, James; HIGHSMITH, Jim; HUNT, Andrew; JEFFRIES, Ron; KERN, Jon; MARICK, Brian; MARTIN, Robert C.; MELLOR, Steve; SCHWABER, Ken; SUTHERLAND, Jeff; THOMAS, Dave. *Manifest Agilního vývoje software* [online] [cit. 2021-05-21]. Dostupné z: `http://agilemanifesto.org/iso/cs/manifesto.html`.

[6] *Agilní programování: metodiky efektivního vývoje softwaru.* 1. vydání. Brno: Computer press, 2004. ISBN 80-251-0342-0.

[7] KOTRLA, Tomáš. *Agilní metodiky vývoje software [online].* 2006 [cit. 2021-05-21]. Dostupné také z: `https://is.muni.cz/th/y0eky/`. Diplomová práce. Masarykova univerzita, Fakulta informatiky, Brno. Vedoucí práce Barbora BÜHNOVÁ.

[8] BUCHALCEVOVÁ, Alena. *Metodiky vývoje a údržby informačních systémů.* 1. vydání. Praha: Grada, 2005. ISBN 80-247-1075-7.

[9] *SCRUM - rámec agilního přístupu - PM Consulting.* [Online]. PM Consulting [cit. 2022-03-06]. Dostupné z: `https://www.pmconsulting.cz/pm-wiki/scrum/`.

[10] HATHAWAY, Adam. *What is a scrum in rugby union? - Rugby World magazine.* [Online]. Future Publishing Limited Quay House, [cit. 2022-03-06]. Dostupné z: `https://www.rugbyworld.com/takingpart/rugby-basics/what-is-a-scrum-in-rugby-union-135791`.

[11] SCHWABER, Ken; SUTHERLAND, Jeff. *Průvodce Scrumem: Definitivní Průvodce Scrumem: Pravidla hry* [online]. 2020 [cit. 2022-03-06]. Dostupné z: `https://scrumguides.org/docs/scrumguide/v2020/2020-Scrum-Guide-Czech.pdf`.

[12] SCHWABER, Ken. *Scrum As A Framework* [online]. 2010 [cit. 2022-03-09]. Dostupné z: `https://kenschwaber.wordpress.com/2010/09/08/scrum-as-a-framework/`.

[13] REDKA, Maria. *Kanban vs. Scrum: Which to Use* [online] [cit. 2022-03-07]. Dostupné z: `https://mlsdev.com/blog/kanban-vs-scrum`.

[14] MYSLÍN, Josef. *Scrum: Průvodce agilním vývojem softwaru.* 1. vydání. Brno: Computer Press, 2016. ISBN 978-80-251-4650-7.

[15] ŠVENDOVÁ, Vendula. Metodika agilního vývoje softwaru na OVSS ÚVT. *Zpravodaj ÚVT MU* [online]. [B.r.], **XXI**(4) [cit. 2022-03-15]. ISSN 1212-0901.

[16] *What are Pigs and Chickens (in Agile Development)? - Definition from Techopedia.* [Online]. Techopedia Inc., 2022 [cit. 2022-03-16]. Dostupné z: https://www.techopedia.com/definition/13672/pigs-and-chickens.

[17] ŠOCHOVÁ, Zuzana; KUNCE, Eduard. *Agilní metody řízení projektů.* 2. vydání. Brno: Computer Press, 2019. ISBN 978-80-251-4961-4.

[18] PICHLER, Roman. *Every Great Product Owner Needs a Great Scrum Master* [online] [cit. 2022-03-13]. Dostupné z: https://www.romanpichler.com/blog/every-great-product-owner-needs-great-scrummaster/.

[19] COHN, Mike. *Succeeding with Agile: Software Development Using Scrum.* 1st. Addison-Wesley Professional, 2009. ISBN 0-321-57936-4.

[20] SCHWABER, Ken. *Agile Project Management with Scrum.* 1. vydání. USA: Microsoft Press, 2004. ISBN 978-0-7356-1993-7.

[21] RADIGAN, Dan. *4 best practices for sprint planning meetings* [online]. Atlassian.com, 2019 [cit. 2022-03-27]. Dostupné z: https://www.atlassian.com/blog/agile/sprint-planning-atlassian.

[22] FIANTA, Roman. *Použití SCRUM pro menší a střední webové projekty.* Brno, 2011. Dostupné také z: https://is.muni.cz/th/l98on/. Diplomová práce. Masarykova univerzita, Fakulta informatiky. Vedoucí práce Jaroslav RÁČEK.

[23] *Extreme Programming: Values, Principles, and Practices* [online]. AltexSoft, 2021 [cit. 2021-05-21]. Dostupné z: https://www.altexsoft.com/blog/business/extreme-programming-values-principles-and-practices/.

[24] KNIBERG, Henrik. *Scrum and XP from the Trenches.* Morrisville, United States: Lulu.com, 2007. ISBN 978-1-4303-2264-1.

[25] PICHLER, Roman. *Agile Product Management with Scrum: Creating Products That Customers Love.* 1st ed. Addison-Wesley Professional, 2010. ISBN 978-0-321-60578-8.

[26] BECK, Kent. *Extrémní programování: knihovna programátora.* 1. vydání. Praha: Grada Publishing, spol. s.r.o., 2002. ISBN 80-247-0300-9.

[27] SOCHOR, Jiří. Údržba softwaru. *Zpravodaj ÚVT MU* [online]. 1996, **VI**(3), 15–20 [cit. 2022-04-19]. ISSN 1212-0901. Dostupné z: http://webserver.ics.muni.cz/bulletin/articles/61.html.

[28] MARTIN, Robert C. *Clean Code: A Handbook of Agile Software Craftsmanship.* New York: Prentice Hall, 2008. ISBN 978-0-13-235088-4.

[29] BRCHAŇ, Jan. *Tvorba moderních aplikací v jazyce Groovy [online].* 2006 [cit. 2022-03-30]. Dostupné také z: https://is.muni.cz/th/ro6h7/. SUPERVISOR : Jan Pavlovič.

[30] *Groovy - Differences from Java.* [Online]. Codehaus Foundation, 2008 [cit. 2022-03-31]. Dostupné z: https://web.archive.org/web/20090317025737/http://groovy.codehaus.org/Differences+from+Java.

[31] *What is a Closure – Different Types of Scopes* [online] [cit. 2022-03-29]. Dostupné z: https://www.baeldung.com/cs/closure.

[32] STANÍČEK, Petr. *Javascript a oblast působnosti proměnných – díl třetí* [online]. Devel.cz Lab s.r.o. [cit. 2022-03-29]. Dostupné z: https://zdrojak.cz/clanky/javascript-a-oblast-pusobnosti-promennych-dil-treti/.

[33] *The Apache Groovy programming language - Closures.* [Online] [cit. 2022-03-29]. Dostupné z: https://groovy-lang.org/closures.html.

[34] *Closures in Groovy* [online] [cit. 2022-03-29]. Dostupné z: https://www.baeldung.com/groovy-closures.

[35] KÖNIG, Dierk; KING, Paul; LAFORGE, Guillaume; D'ARCY, Hamlet; CHAMPEAU, Cédric; PRAGT, Erik; SKEET, Jon. *Groovy in Action.* 2nd Edition. United States of America: Manning Publications Co, 2015. ISBN 978-1-93518-244-3.

[36] *Binding (Groovy 4.0.1)* [online]. The Apache Software Foundation [cit. 2022-03-31]. Dostupné z: `https://docs.groovy-lang.org/latest/html/api/groovy/lang/Binding.html`.

[37] K, Naresha. *Groovy Scripts - Exploring Binding* [online] [cit. 2022-03-31]. Dostupné z: `https://blog.nareshak.com/groovy-scripts-exploring-binding/`.

[38] *What is Story Point in Agile? How to Estimate a User Story?* [Online]. Visual Paradigm [cit. 2022-04-25]. Dostupné z: `https://www.visual-paradigm.com/scrum/what-is-story-point-in-agile/`.

SEZNAM OBRÁZKŮ

Obr. 1.1. Posloupnost jednotlivých fází ve vodopádovém modelu. [2] 11
Obr. 2.1. Povídka o praseti a kuřeti. [15] 18
Obr. 2.2. Varianty Scrumu. [22] 24
Obr. 3.1. Podíly údržbových prací. [27] 26
Obr. 4.1. Ukázka používání Groovy Closure. Převzato z [34]. 33
Obr. 4.2. Ukázka používání Groovy Binding. Převzato z [37]. 34
Obr. 5.1. Graf počtu naplánovaných story points za jednotlivé Sprinty 37
Obr. 5.2. Ukázka vizualizace stavu a rozdělení položek ve Sprintu v Jira Software. 38
Obr. 5.3. Ukázka vizualizace položek v Trello. 38
Obr. 6.1. Ukázka grl souboru: získání tříd k vyplnění ve formuláři při registraci. 55
Obr. 6.2. Ukázka grl souboru: úprava výpisu dle typu osoby. 56
Obr. 6.3. Ukázka grl souboru: při jaké diagnóze je možná laserová operace. 56
Obr. 6.4. Implementace Java metody pro získání vhodných diagnóz pro laserovou operaci za použití souboru laser.grl. 57
Obr. 6.5. Implementace Java metody pro získání vstupních hodnot pro grl skript. 58
Obr. 6.6. Ukázka grl souboru: příklad pro validaci a spouštění jiného grl souboru. 59

SEZNAM TABULEK

Tab. 1.1. Porovnání metodik podle [8]. 15
Tab. 5.1. Odpovědi na otázky ohledně nastaveni Scrumu. 44
Tab. 5.2. Pokračování odpovědí na otázky ohledně nastaveni Scrumu. 45
Tab. 5.3. Odpovědi na otázky ohledně komunikace. 45
Tab. 5.4. Odpovědi na otázky ohledně možného zlepšení vývoje. 46
Tab. 5.5. Odpovědi na otevřené otázky. 47

SEZNAM PŘÍLOH

P I. Elektronické přílohy

PŘÍLOHA P I. ELEKTRONICKÉ PŘÍLOHY

Součástí příloh této práce jsou následující složky:

CML_engine (zde se nachází všechny soubory tvořící CML engine a ukázky grl souborů, které byly v práci použity pro názornou ukázku funkčnosti enginu):

- `cml`
  - `CmlEngineBean.java`
  - `resources.CML`
    - `laser.grl`
    - `registrace.grl`
    - `ukazky.grl`
    - `vypis.grl`
- `grools`
  - `api`
    - `GrlValidator.java`
    - `GroolsEngine.java`
    - `GroolsMultiInputOutput.java`
    - `GroolsTuple.java`
    - `GroovyClassLoaderProvider.java`
    - `JavaScript.java`
    - `ReturnObject.java`
  - `engine`
    - `DefaultGroolsEngine.java`
    - `GroolsMultiInputOutputImpl.java`
    - `PreprocessorExtension.java`
    - `PreprocessorExtensionRegistry.java`
    - `SimpleGroolsClassLoaderProvider.java`
  - `factory`
    - `DebugLevel.java`
    - `DefaultPreprocessorExtensionRegistry.java`
    - `DefaultScriptFactory.java`
    - `GrlScriptClassLoader.java`

        * GroupPreprocessorExtension.java
        * LoadableScriptFactory.java
        * ScriptFactory.java
        * VoidPreprocessorExtension.java
    - fuction
        * ClosureFunction.java
        * FunctionDelegate.java
        * JavaScriptFunction.java
    - lang
        * script
            · GrlScript.java
        * CustomClosureMetaclass.java
        * GrlCodeSource.java
        * GrlContext.java
        * GrlContextContainer.java
        * GrlParserPlugin.java
        * PojoWithDynamicProperties.java
        * ReturnObjectMetaClass.java
        * SourcePreProcessor.groovy
    - source
        * AbstractScriptSource.java
        * ClassPathScriptSource.java
        * FilesystemScriptSource.java
        * ScriptClass.java
        * ScriptDescription.java
        * ScriptSource.java
    - validation
        * ClosureGrlValidator.java
        * JavaScriptGrlValidator.java
        * ReturnObjectMaker.groovy
        * ValidatorWithJavaScriptSupport.java

Dotazník (zde se nacházejí odpovědi jednotlivých respondentů, kteří se zúčastnili dotazníkového šetření):

- Odpovedi.pdf