Univerzita Hradec Králové
Fakulta informatiky a managementu
Katedra informačních technologií

Continuous integration – aplikace automatického testování softwaru

Diplomová práce

Autor: Bc. Jan Podzimek
Studijní obor: Informační management

Vedoucí práce: Ing. Martina Husáková, Ph.D.

Odborný konzultant: Mgr. Jan Vondrouš

Hradec Králové Srpen 2019

Prohlášení:

Prohlašuji, že jsem diplomovou práci zpracoval samostatně a s použitím uvedené literatury, webových a dalších zdrojů.

V Hradci Králové dne 11.8.2019 Jan Podzimek

Poděkování:

Děkuji vedoucí mé diplomové práce Ing. Martině Husákové Ph.D. za metodické vedení, Mgr. Janu Vondroušovi za odborné rady, konzultaci a dohled při zpracování praktických částí diplomové práce, vývojovým týmům produktu Inspire Scaler za vytvoření vhodných podmínek pro výzkum a v neposlední řadě své přítelkyni Andree Oborníkové a rodině za motivaci, povzbuzení a morální podporu.

Anotace

Diplomová práce se zabývá tématem automatického testování softwaru a metodikou continuous integration. Popisuje základní elementární prvky a základní procesy, které toto téma obsahuje. Dále se zabývá obecnými vlastnostmi CI serverů a uvádí jejich současné konkrétní příklady.

Praktická část se zaměřuje na aplikaci continuous integration při ověřování softwaru Inspire Scaler společnosti Quadient. Řeší konkrétní problémy vzniklé při jeho pravidelném automatickém testování. Následkem aplikování uvedených řešení byla zkrácena doba trvání systémových testů o čtyři hodiny. Z původních sedmi hodin byla celková doba všech systémových testů snížena na tři hodiny, a to i při vysokém přírůstku systémových testů. Dále byly náhodně padající systémové testy stabilizovány, čímž se výrazně zjednodušilo vyhodnocení testů a odhalení skutečných chyb.

Annotation

Title: Continuous Integration – application of automated software testing

This Master's thesis deals with the topic of automatic software testing and continuous integration methodology. It describes the basic elements and processes that the topic contains. It also focuses on the general features of CI servers and presents their concrete examples. The practical part deals with the application of continuous integration in verification of the Inspire Scaler software developed by Quadient. It solves specific problems encountered in its regular automatic testing. As a result of applying these solutions, the duration of the system tests was reduced by four hours. From the original seven hours, the total time of all system tests was reduced by three hours, even with a high increase in system tests. In addition, randomly failing system tests were stabilized, which greatly simplified the process of evaluating tests and discovering real mistakes.

Obsah

1 Úvod 1

2 Cíl práce 2

3 Teoretická východiska diplomové práce 3

3.1 Softwarové testování 3

3.1.1 Úvod do softwarového testování 3

3.1.2 Způsoby testování 5

3.1.3 Základní druhy automatických testů 7

3.1.4 Další skupiny testů 10

3.1.5 Testovací vzory 11

3.1.6 Pozadí automatických testů 13

3.2 Continuous integration 14

3.2.1 Continuous software engineering 14

3.2.2 Continuous integration 15

3.2.3 Continuous integration workflow 17

3.2.4 Hlavní procesy continuous integration 23

3.2.5 Přínosy continuous integration 26

3.2.6 Problémy s continuous integration 28

3.2.7 Continuous delivery a deployment 28

3.3 Continuous integration server 30

3.3.1 Společné znaky CI serverů 30

3.3.2 Příklady CI serverů 34

4 Praktická část diplomové práce 38

4.1 Inspire Scaler 38

4.1.1 Popis produktu 38

4.1.2 Testovací projekt 40

4.1.3 Základní problémy testovacího projektu 40
4.2 Řešení problémů s aplikací continuous integration 42
4.2.1 Automatizace manuálních činností 42
4.2.2 Zavádění nových procesů 47
4.2.3 Stabilita systémových testů 52
4.2.4 Doba trvání systémových testů 57
4.2.5 Infrastruktura CI serverů 62
5 Shrnutí výsledků 66
5.1 Postupné automatizování manuálních testů 66
5.2 Rozvoj integračních testů 67
5.3 Zrychlení celkového trvání systémových testů 68
5.4 Zvýšení stability systémových testů 69
5.5 Zvýšení možnosti rychlé zpětné vazby 70
6 Závěry a doporučení 72
7 Seznam použité literatury 73
8 Přílohy 77

Seznam obrázků

Obrázek 1: Relativní cena opravy bugu v závislosti na době jeho nalezení 5
Obrázek 2: Testovací pyramida 12
Obrázek 3: Ice cream cone 13
Obrázek 4: Vývojový cyklus softwaru 14
Obrázek 5: Návaznost continuous integration na metody agilního vývoje 16
Obrázek 6: Continuous integration workflow 18
Obrázek 7: Udržitelnost stávajících verzí softwaru 20
Obrázek 8: Continuous delivery, continuous deployment 29
Obrázek 9: Build skript – Jenkins Blue Ocean pipeline 31
Obrázek 10: Rozšiřitelná architektura CI Jenkins 32
Obrázek 11: Ukázka GitLab CI prostředí 37
Obrázek 12: Inspire Scaler – dashboard 39
Obrázek 13: CI report v MS Teams 47
Obrázek 14: Architektura aplikovatelného CI serveru 64
Obrázek 15: TeamCity Build chain 65

Seznam ukázek zdrojového kódu

Ukázka 1: Unit test v programovacím jazyce Groovy 7
Ukázka 2: Základní integrační test 8
Ukázka 3: GUI test implementovaný pomocí frameworku Geb 9
Ukázka 4: Struktura systémových testů 45
Ukázka 5: Anotací knihovny JUnit5 53
Ukázka 6: Metoda s aktivním čekáním 55
Ukázka 7: WireMock server 57
Ukázka 8: Označení systémových testů 59

Seznam grafů

Graf 1: Vývoj automatizace testovacích scénářů 66
Graf 2: Poměr integračních a systémových testů 68
Graf 3: Vývoj doby systémových testů 69
Graf 4: Průměrný počet neúspěšných systémových testů v ověření 70

1 Úvod

Současné požadavky na vývoj softwaru se neustále zvyšují. Software musí být výkonný, jeho používání intuitivní, funkcionalita musí být obsáhlá, a především bez chyb. Chyby v softwaru snižují jeho kvalitu, a tím pádem i jeho obchodovatelnou hodnotu či dobré jméno softwarové společnosti. Je proto velice důležité software pravidelně testovat, a to nejlépe automatickými testy, které jsou rychlé a nenákladné oproti lidským zdrojům.

Pravidelné testování je obzvláště důležité, pokud se zdrojový kód neustále mění a rozrůstá. To se například odehrává při vývoji softwaru agilními metodikami. Software zpravidla disponuje velkým množstvím funkcionality, jejíž zachování je potřeba ověřovat, a to nejlépe při každém dalším zavádění změn. Včasné odhalení chyb šetří náklady na jejich opravu. Zároveň pravidelné testování zvyšuje kvalitu softwaru, ale i důvěru členů vývojářského týmu ve vydávaný software.

Možný způsob, jak při vývoji softwaru včasně a automaticky odhalit chyby, je continuous integration. Continuous integration je přístup vývoje softwaru, který se zaměřuje na ověřování softwaru v nejkratším možném časovém intervalu od zavedení změn do softwaru – tedy bezprostředně po provedení integrace členem vývojového týmu. Aby mohl být tento způsob ověřování softwaru uplatňován, je potřeba zřídit integrační server, zautomatizovat manuální testy a řídit hlavní procesy continuous integration.

V některých případech vznikají různá úskalí. Obsáhlé automatické testy mohou být s měnícím se softwarem velice nestabilní. Nestabilní test vyžaduje časté analyzování příčin jeho pádu. S rozvojem funkcionality přibývá i množství automatických testů, ověření celého softwaru se tak stává nákladné a časově náročné. Neustálou pozornost vyžaduje i integrační server, který může přestat stíhat vykonávat zvyšujících se požadavků. Následující práce uvádí možnosti, jakými lze tyto problémy efektivně řešit, minimalizovat a mírnit jejich dopady.

2 Cíl práce

Cílem diplomové práce je popsat automatické testování softwaru jako nedílnou součást metodiky vývoje softwaru continuous integration. Dále je pak cílem specifikovat základní problémy testovacího projektu produktu Inspire Scaler společnosti Quadient při aplikování continuous integration a navrhnout jejich možná řešení.

3 Teoretická východiska diplomové práce

3.1 Softwarové testování

Kapitola pojedná o testování softwaru jako o jedné z nejdůležitějších činností při vývoji softwaru. Vysvětluje, proč je automatické testování důležité, a představuje základní druhy automatických testů.

3.1.1 Úvod do softwarového testování

Testování softwaru je proces, ve kterém je validováno a verifikováno, že software splňuje zadané a předem stanovené technické, ale i uživatelské požadavky. Testování je primární metodou, jak zkontrolovat, že implementovaná funkcionalita tyto požadavky splňuje odpovídajícím způsobem. (Eriksson, 2016) Testování softwaru je také proces identifikace správnosti, koherence a kvality celého softwaru. Testování je prováděno za účelem odhalení a odstranění nežádoucích chyb softwaru, které snižují jeho kvalitu a obchodní hodnotu. (Step2QA, 2018)

Pravidelné testování je nedílnou součástí obzvláště u agilního vývoje softwaru. Každá vzniklá funkce, každá vzniklá komponenta, zkrátka každá nová část softwaru, musí být řádně otestována před vydáním nové verze zákazníkovi. Kromě otestování nově vzniklých částí je též potřeba ověřit, zda nové části softwaru či jejich implementace neovlivnily stávající funkcionalitu nežádoucím způsobem, ale naopak zachovaly její správné chování a možnosti, jakými software disponuje. (Garousi, 2019)

Každý neodhalený bug[1] ovlivňuje schopnost software použít či je jeho používání v některých případech limitováno. Oproti tomu existují i bugy, při kterých software nevykazuje známky špatného chování na první pohled. To může například způsobit nevědomá změna funkcionality softwaru. Takováto nevědomá změna může mít fatální následky pro uživatele/zákazníka, který software využívá pro dosažení svých

[1] Bug – ustálené označení pro chybu v softwaru; nežádoucí chování softwaru

business cílů, neboť software vykonává jiné činnosti, než je ve skutečnosti deklarováno ve specifikaci (dokumentace, uživatelský manuál). Bug může nastat z několika obecných případů (Patton, 2001):

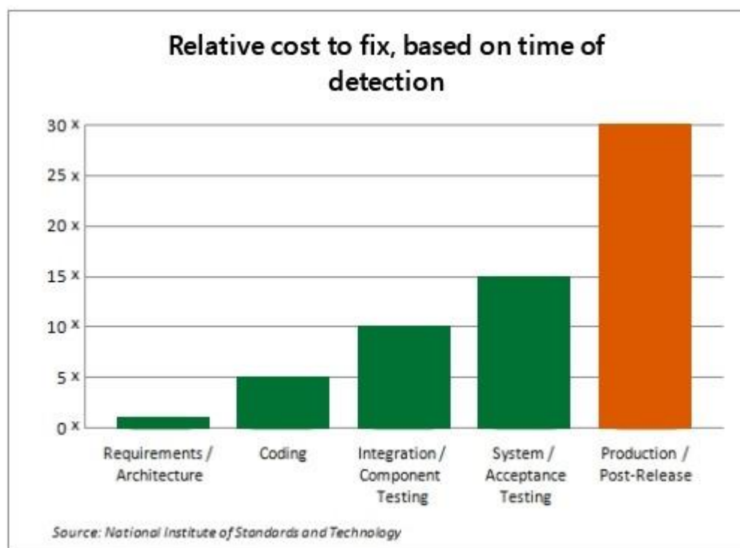
- software něco nedělá, co specifikace deklaruje,
- software dělá něco, co specifikace zamítá,
- software něco dělá a specifikace to nezmiňuje,
- software něco nedělá, specifikace to nezmiňuje, ale měla by,
- je těžké porozumět, zda software funguje správně či nikoliv.

Pokud nebude software pravidelně testován, vystavují se producenti softwaru riziku chyb, které se i přes nemalé úsilí vývojářů dostanou do zdrojového kódu a poté i k zákazníkovi. Zákazník může takové chyby objevit během užívání softwaru a požadovat jejich opravu v nové verzi. Producenti softwaru jsou nejčastěji v takovýchto případech povinni na základě poskytování tzv. service-level agreement[2] chyby opravit a vydat novou verzi. Oprava chyb a vydání nové verze softwaru zákazníkovi následně stojí vynaložení několikanásobného úsilí a nákladů, než by bylo potřeba například při implementaci nové funkcionality. (TryQA, 2019)

Obrázek 1 znázorňuje relativní náklady na opravu bugu v různých stádiích vývoje a vydání verze softwaru. Odhalení bugu v prvních fázích vývoje softwaru je jednou z nejlepších variant, které mohou nastat. Pokud je bug objeven například při systémovém testování, je cena jeho opravy až patnáctkrát vyšší. Do této ceny se totiž musí započítat i práce lidí, kteří chybu našli, ověřili a reportovali vývojáři. Ten se bugu musí věnovat, a to většinou ve chvíli, kdy již pracuje na dalších úkolech. V nejhorším scénáři je bug objeven až v produkci, tedy v ostrém nasazení softwaru u zákazníka. Do ceny opravy je pak započítáno úsilí všech lidí, kteří se zapojili do zákaznické podpory, opravy bugu, verifikace opravy, ale i vydání

[2] Service-level agreement – smlouva o úrovni poskytování služeb mezi dodavatelem softwaru a jejím uživatelem. Ve smlouvě je stanoven rozsah, úroveň a kvalita služby. Nejčastěji se může jednat o způsob řešení podpory, řešení výjimečných stavů, stanovení komunikačních kanálů atd.

nové opravné verze softwaru. Jak znázorňuje *obrázek 1*, může se jednat až o jednou tak větší cenu opravy než při interním odhalení během testování. (Patton, 2001)



Obrázek 1: Relativní cena opravy bugu v závislosti na době jeho nalezení
Zdroj: Step2QA, 2018

3.1.2 Způsoby testování

Manuální testování – manuální testování softwaru je vykonáváno převážně kapacitami testerů, kteří jsou často součástí vývojových týmů. Manuální testování je jedním z nejjednodušších testování softwaru, zároveň se ale jedná o velice časově náročnou praktiku. Při manuálním testování se spoléhá na znalosti testera ohledně softwaru, jeho chování a funkcionality. Testování je tak závislé na preciznosti a přesnosti lidského faktoru. Testování softwaru je prováděno na základě stanovených testovacích scénářích. Testovací scénář je předpis jednotlivých kroků a očekávaných výsledků, kterých má být dosaženo.

Automatické testování – automatické testování představuje souhrn metod, které využívají specializovaných nástrojů pro provádění testů a kontroly získaných výsledků s očekávanými hodnotami. (Techopedia, 2019) Automatizací testů se poté rozumí převod manuálních testů na automatické, které lze vykonávat bez zapojení lidského faktoru. Automatické testy tedy provádí výpočetní jednotka, která ověřuje software podle přesně definovaných předpisů a výsledky srovnává se stanovenými

hodnotami. (Dustin, 2009) Předpis takovýchto testů zpravidla představuje součást zdrojového kódu, který je samostatně spustitelný v IDE[3] či jiných specializovaných nástrojích. Automatické testy přinášejí oproti manuálním testům značné výhody, a to hlavně (Dustin, 2009, Techopedia, 2019):

- šetří čas a náklady softwarového testování,
- zlepšují přesnost a spolehlivost testování,
- umožnují testovat náročné a komplikované úlohy,
- přináší vyšší pokrytí funkcionality softwaru testy,
- lze je opakovaně vykonávat za konzistentních podmínek,
- odhalení chyb a problémů je rychlejší.

White box testing – kromě manuálního či automatického testování lze rozlišovat testování i podle znalosti softwaru. Testování tzv. bílé krabice představuje metaforické označení pro testování softwaru, kdy je zřejmé až průhledné, jak systém funguje a jak fungují jeho vnitřní procesy. Tento druh testování vyžaduje vysoké znalosti v oblasti fungování daného softwaru, ale také obecných principů fungování systému a programování.

Black box testing – opakem bílé krabice je analogicky černá. Testování je prováděno bez znalosti vnitřní datové struktury a chování vnitřních metod. Software je testován na základě předložení vstupních dat a získaného očekávaného výsledku. (Čermák, 2010)

Grey box testing – jakýsi pomyslný střed mezi oběma přístupy. Testování šedé skříňky předpokládá omezenou znalost vnitřního fungování softwaru a jeho datových struktur. Mírná znalost vnitřního fungování softwaru dovolují provádět komplexní a cílené testy. Zároveň není testování zatíženo obšírnou znalostí vnitřního fungování softwaru, která by mohla různé typy testovacích scénářů ignorovat či snížit jejich relevantnost. (Dustin, 2009)

[3]IDE – Integrated Development Environment – označení pro softwarový nástroj, který zpřístupňuje a zjednodušuje práci členům vývojářských týmů při vyvíjení softwaru

3.1.3 Základní druhy automatických testů

Existuje velké množství druhů automatických testů. Ty se nejčastěji dělí podle typu testování – funkcionální a nefunkcionální. Funkcionální testování ověřuje software z hlediska jeho funkcí a business využití, oproti tomu je non-funkcionální testování zaměřeno na ověření technických částí jako je zabezpečení, výkonnost[4], kvalita atd. Dále lze testy dělit podle fáze vývoje, nebo podle typu testu. (Shinde, 2019). Mezi základní typy testů patří:

Unit testy – unit testy neboli jednotkové testy operují na nejnižší úrovni zdrojového kódu a jak již název deklaruje, testují funkcionalitu v oblasti jedné jednotky neboli jedné třídy. Jedná se o základní automatické testy, které jsou nejčastěji spouštěny při sestavení softwaru, a pokud selžou, selže i sestavení softwaru. Jednotkové testy se vyznačují svou rychlostí, jednoduchostí a nezávislostí. Nevýhodou jednotkových testů je nutnost jejich častého opravování a přepisování při změně funkcionality. Použití jednotkových testů je samozřejmostí a již zaběhnutým zvykem každého softwarového projektu.

```
import org.junit.jupiter.api.Test

class GreetingsTest {
    String person = "Jan"

    @Test
    void "Unit Test Example"() {
        when:
        String greeting = Greetings.sayHello(person)

        then:
        assert greeting == "Hello, $person!"
    }
}
```

Ukázka 1: Unit test v programovacím jazyce Groovy[5]
Zdroj: Vlastní zpracování

Unit test ověřuje statickou metodu sayHello() třídy Greetings. V kontrolní části je porovnáván obdržený objekt typu String s očekávaným řetězcem.

---

[4] Výkonost – performance; obecně se týká monitorování a měření relevantních metrik softwaru
[5] Groovy – dynamický objektově orientovaný programovací jazyk platformy Java

Integrační testy – integrační testy ověřují metody, které propojují či využívají více tříd dohromady, testy jsou proto více závislé a složitější než jednotkové testy. Zároveň díky jejich komplexnosti dovolují otestovat větší celky zdrojového kódu. Často takové testy testují i spolupráci tříd z různých modulů a integraci s ostatními službami. Ověřují jak standardní případy, tak hraniční až krizové situace. Těchto testů bývá menší množství, o to více důležité jsou. (Vocke, 2018) Do této skupiny se také řadí testy API[6]. Často probíhají po dokončení sestavení softwaru, pro svůj běh nepotřebují kompletně spuštěný software.

```
import org.junit.jupiter.api.Test

import static GreetingsType.GOOD_MORNING

class GreetingsIntegrationTest {

    String firstName = "Jan"
    String lastName = "Podzimek"

    @Test
    void "Integration test example"() {
        given:
        Person person = new Person(firstName, lastName)

        when:
        String greeting = new Greetings(GOOD_MORNING, person).greeting()

        then:
        assert greeting == "$GOOD_MORNING, $firstName $lastName!"
    }
}
```

Ukázka 2: Základní integrační test
Zdroj: vlastní zpracování

Základní integrační test ověřuje metodu greeting() instance třídy Greetings, která pro svou inicializaci vyžaduje instanci třídy Person a typ GreetingsType. Obdržená hodnota je porovnávána asercí s očekávanými hodnotami.

Systémové testy – systémové testy, end-to-end testy či user interface (UI) testy ověřují komplexní fungování softwaru z pohledu uživatelských operací. Testy vyžadují běh celého softwaru, kterému předchází úspěšná instalace softwaru.

[6] API – Application Programming Interface; rozhraní softwaru, přes které lze komunikovat a posílat požadavky

Tento druh testů je založen na reálných uživatelských scénářích, tedy na případech, v jakých budou zákazníci software používat. Testy tedy především simulují jejich počínání. Jedná se tak o nejkomplexnější testy, které jsou ale pomalé, závislé na celé řadě faktorů, kvůli kterým jsou často nestabilní. Přestože testy často testují jednu věc, jednu uživatelskou akci, předchází této akci celá řada dílčích kroků, které je potřeba v softwaru nastavit. (Shinde, 2019)

Součástí systémových testů jsou též graphical user interface (GUI) testy. Ty jsou založené na ověřování koherence grafického zobrazení a funkčnosti jednotlivých grafických komponentů v softwaru, tedy pokud jimi software disponuje. Test je koncipován jako simulace uživatelského chování v grafickém rozhraní. Jedná se tedy o zkoušení reálných kliknutí na tlačítka, hypertextové odkazy či jiné komponenty, přičemž je očekáváno, že se například otevře nové prohlížečové okno s příslušnou stránkou. Tyto testy jsou velice pomalé a mají největší míru poruchovosti – to kvůli časté změně grafického rozložení stránky a simulování uživatelské interakce v GUI. GUI testy vyžadují nainstalovaný a spustitelný software.

```
import org.junit.jupiter.api.Test
class GraphicalUserInterfaceTest implements InitGebTrait {

    @Test
    void "Graphical user interface test"() {
        when:
        GebPage gebPage = browser.to(GebPage)

        then:
        assert gebPage.getTitle() == "Geb - Very Groovy Browser Automation"

        when:
        gebPage.openDocumentation()

        then:
        assert gebPage.getTitle().startsWith("The Book Of Geb")
    }
}
```

Ukázka 3: GUI test implementovaný pomocí frameworku Geb[7]
Zdroj: Geb, 2019; upraveno

[7]Geb – Groovy framework uzpůsobený pro snadné implementování GUI testů

GUI test ověřuje (simuluje procházení) webové aplikace prostřednictvím webového prohlížeče. V instanci třídy GebPage jsou specifikovány informace o reálné webové stránce – tedy její jednotná adresa zdroje (url), název i očekávaný obsah. Metodou getTitle(), je získán a poté zkontrolován titulek webové stránky. Následovně je GUI test přesměrován metodou openDocumentation(), na podstránku webu, kde znovu zkontroluje titulek, jehož začátek je porovnán s očekávanou hodnotou.

3.1.4 Další skupiny testů

Smoke testy – smoke testy neboli kouřové testy ověřují základní funkcionalitu softwaru. Jedná se o několik málo systémových nebo GUI testů, které ověřují všechny klíčové funkce softwaru. Smoke testy slouží jako nejrychlejší způsob ověření po integraci nových částí softwaru. Pro běh takovýchto testů je vždy nutná nainstalovaná a plně spustitelná verze softwaru. (Hlava, 2011)

Performance testy – výkonnostní testy jsou prováděny za účelem ověření vlastností softwaru. Takovéto testy jsou koncipovány tak, aby zatížily či dokonce zahltily software, a přitom monitorovaly jeho chování, výkonnost a propustnost. Tyto vlastnosti jsou klíčové i pro deklaraci specifikace a limitace softwaru.

Security testy – tento druh testů je nedílnou součástí testování. Během testů je ověřována bezpečnost a zranitelnost částí softwaru proti různým druhům hrozeb, neoprávněným přístupům i cíleným útokům. (Shinde, 2019) Součástí bývá i ověřování knihoven třetích stran, které jsou v softwaru zakomponovány. Security testy bývají koncipovány jako funkcionální nebo jako nefunkcionální testy na různých úrovních softwaru.

Release testy – vydávací či uvolňovací testy představují skupinu testů, které jsou prováděny těsně před vydáním nové verze softwaru. V tomto případě se často jedná o manuální testy, které jsou prováděny za účelem posledního ujištění, že vše vypadá, jak má. Tedy jedná se často o vizuální zkontrolování GUI softwaru, ověření správných čísel verzí softwaru nebo dokumentace.

Akceptační testy – akceptační testy jsou zpravidla prováděny zákazníkem, který si ověřuje software na základě stanovených požadavků na svém testovacím prostředí. Koncepce testů velice často závisí na schopnosti zákazníka. Toto testování však často rozhoduje o akceptování či zamítnutí softwaru a jeho nasazením do produkce. Případné nesrovnalosti jsou reportovány zpět společnosti. (Hlava, 2011)

3.1.5 Testovací vzory

Každý softwarový projekt používá několik různých druhů testů. Jedná se tak vždy o kombinaci, jejíž prvky jsou zastoupeny v různém poměru. Každý z výše uvedených testů je specifický, důležitý a má v projektu své místo. Například jednotkové testy kontrolují jednotlivé třídy a jsou důležité pro ověření jednotlivých metod, zatímco systémové testy jsou důležité pro ověření toho, zda software funguje správně jako celek. Nelze tak říci, že stačí psát a udržovat jeden druh testu.

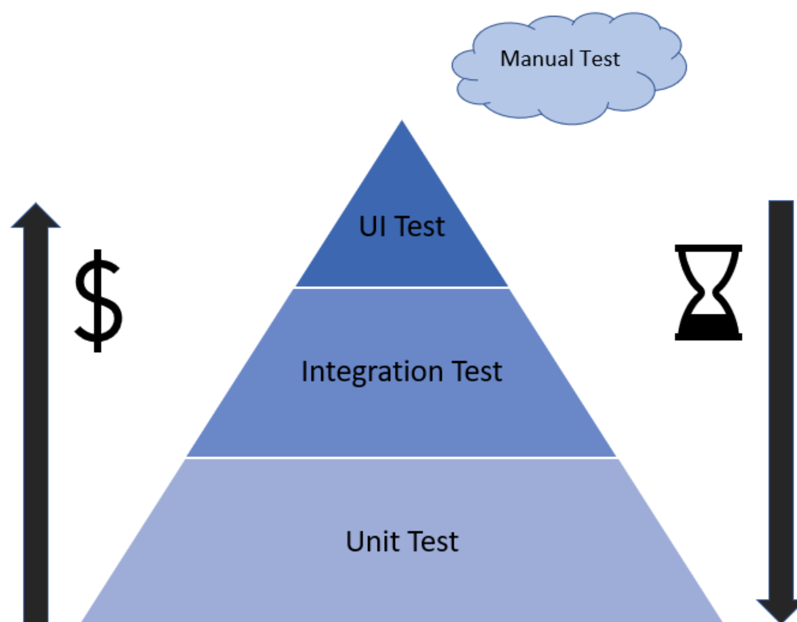
Testovací pyramida – *obrázek 2* znázorňuje ideální kombinaci testů pro testování softwaru. Testovací pyramida symbolizuje metaforu, která ukazuje, jak správně seskupit testy podle odlišných vlastností. Také poskytuje představu o tom, kolik testů by v každé skupině mělo být.[8] Základ testovací pyramidy tvoří jednotkové testy, navazují integrační testy a systémové testy tvoří vrchol pyramidy. Manuální testy jsou zde znázorněny jako malý obláček nad celou pyramidou.

V některých případech je použití manuálních testů nevyhnutelné, protože existují testy, které jdou velice špatně zautomatizovat, či by takový krok znamenal vyšší náklady než manuální otestování. Takovýchto testů by však mělo být jen minimální až nezbytné množství. *Obrázek 2* také uvádí čas a náklady, které stoupají či klesají s každým stupněm pyramidy. Jedná se o čas neboli dobu, kterou testy potřebují pro vykonávání testových kroků, a zároveň náklady na běh

---

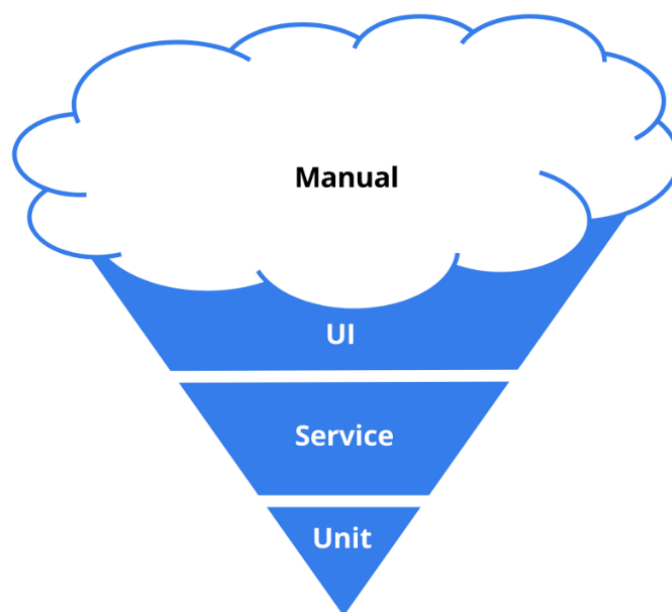
[8] „The "Test Pyramid" is a metaphor that tells us to group software tests into buckets of different granularity. It also gives an idea of how many tests we should have in each of these group." H. Vocke

a udržování. Testy z vyšší vrstvy pyramidy jsou nákladnější a zároveň pomalejší. Naopak testy ze základní vrstvy jsou rychlé a jejich cena je nízká. (Vocke, 2018)



Obrázek 2: Testovací pyramida
Zdroj: vlastí zpracování

Ice cream cone – oproti testovací pyramidě existují i další testovací vzory, které již ale nejsou optimální kombinací, ale naopak poukazují na to, čím některé softwarové projekty trpí. *Obrázek 3* zobrazuje testovací vzor ice cream cone neboli zmrzlinový kornout. V tomto případě je v softwarovém projektu velké množství manuálních testů, které jsou značně neefektivní. Dále následuje nemalé množství systémových či GUI testů, které jsou časově i udržitelně nákladné. Naopak základních jednotkových testů existuje velice málo. (Cochran, 2017) Testování v takovém projektu musí být zdlouhavé a velice nákladné, obzvláště v období vydávání nové verze softwaru, kdy je potřeba všechny manuální testy ověřit. Dalším často zmiňovaným vzorem jsou takzvané přesýpací hodiny. Jedná se o stav, kdy je implementováno velké množství jednotkových testů a zároveň i příliš velké množství systémových testů.



Obrázek 3: Ice cream cone
Zdroj: Cochran, 2017

3.1.6 Pozadí automatických testů

Automatické testy přinášejí schopnost rychlého a nenákladného ověření softwaru. Což je v porovnání s manuálními testy nepřekonatelnou výhodou. Pro plné využití této výhody je potřeba software testovat v pravidelných intervalech. Nejlépe bezprostředně po dokončení implementace nové funkcionality. To přináší vývojářskému týmu celou řadu starostí a povinností od tvorby automatických testů, jejich správu a udržování, ale také zajištění prostředí a kapacity pro jejich pravidelné spouštění a získávání výsledků.

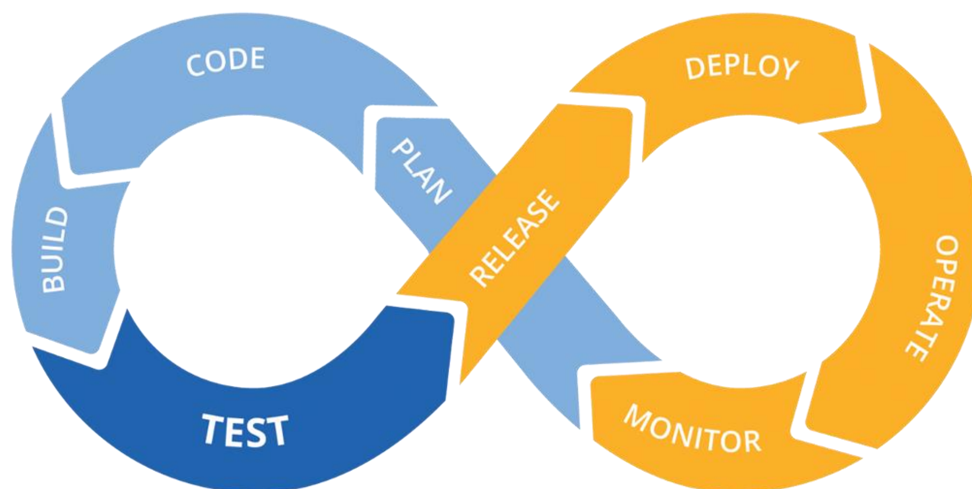
Mimo jiné se testy potýkají s různými problémy, například mohou být tzv. false positive. Pojem označuje situaci, kdy jsou testy falešně pozitivní. Tedy test prochází, ale neměl by – například netestuje to, co deklaruje, či je vůči chybě slepý. Oproti tomu false negative je situace, kdy test hlásí chybu, a přitom neexistuje žádná chyba v softwaru. Tyto problémy mohou přinášet mylné informace o úspěšně provedených změnách, obzvláště u přístupů vývoje softwaru, ve kterých je kladen důraz na ověření veškerých změn, jako je tomu například v continuous integration.

3.2 Continuous integration

Kapitola představuje moderní přístup vývoje softwaru, jeho základní východiska a procesy, jejímž praktikováním lze udržet bezchybný software a minimalizovat náklady na vydávání opravených verzí.

3.2.1 Continuous software engineering

Continuous software engineering je moderní oblast softwarového inženýrství, která se zabývá přístupy a metodikami průběžného vývoje. Jedná se o souhrn schopností organizovat, vyvíjet a vydávat nové verze softwaru v krátkých intervalech, a to v řádech dnů či týdnů. To mimo jiné zahrnuje i aktivity jako prioritizace a rozvoj nové funkcionality, její ověření a testování, refactoring[9] a sbírání zpětné vazby zákazníků pro uzpůsobení dalších vývojových cyklů. (Karvonen, 2017). Vývojový cyklus softwaru představuje dobu mezi započetím plánování a vývoje nové funkcionality softwaru a jejím uvolněním a dodáním zákazníkovi. Typický vývojový cyklus softwaru znázorňuje *obrázek 4.*



Obrázek 4: Vývojový cyklus softwaru
Zdroj: Azeri, 2019

[9] Refactoring – proces zavádění změn v již napsaném zdrojovém kódu, který nezmění jeho vnější funkcionalitu, ale naopak zlepšuje jeho strukturu a čitelnost

V continuous software engineering můžeme identifikovat klíčové oblasti:

- business strategie a plánování – průběžné plánování vývoje v souladu s business strategiemi, financováním a aktuálními potřebami projektu,
- development – průběžný vývoj a integrace změn, testování, vydávání a nasazování nových verzí softwaru,
- operace a monitorování – průběžné poskytování a naplňování service-level agreementu, monitorování a nasazení softwaru do produkčního prostředí[10],
- zlepšování a inovace. (Fitzgerald, 2017)

Continuous software engineering dlouhodobě přispívá ke zlepšení kvality produktu a rozvoje jeho žádoucí funkcionality, dále pak ke zkrácení dodací lhůty, která je lépe dosažitelná díky praktikování častějších integrací a tvořením menších verzí. Díky pozornějšímu vnímání zpětné vazby od zákazníků vede i ke zlepšení komunikace mezi zúčastněnými stranami. (Karvonen, 2017)

3.2.2 Continuous integration

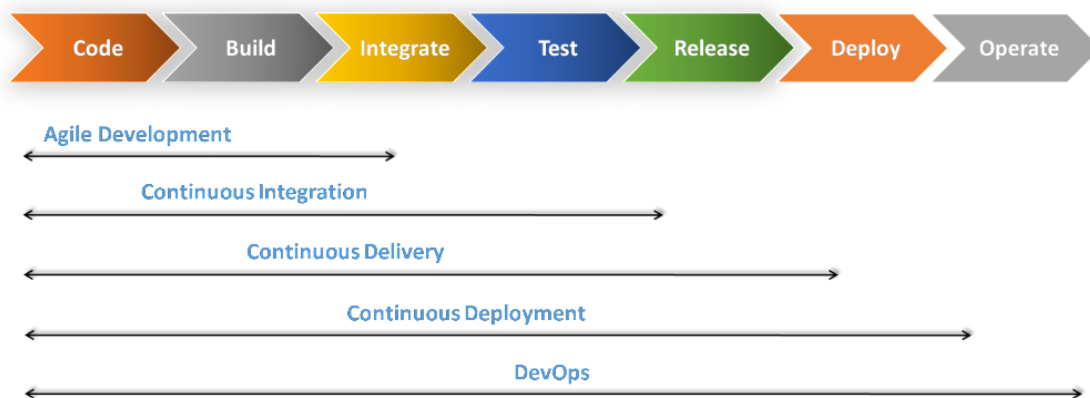
Continuous integration neboli průběžná integrace je přístup řízení vývoje softwaru, který, jak již bylo naznačeno, spadá do oblasti průběžného softwarového inženýrství. Jedná se o souhrn metod, praktik a nástrojů, které mají za úkol zkrátit dobu vývoje softwaru a umožnit efektivní týmovou spolupráci. (Duvall, 2007) Efektivní týmová či mezi týmová spolupráce je obzvláště důležitá ve způsobu vývoje, při kterém je potřeba souběžně upravovat a měnit související části softwaru. Continuous integration je tedy taková metodika vývoje softwaru, ve které každý člen z vývojářského týmu zavádí nové části kódu či upravuje stávající části do sdíleného uložiště zdrojových kódů neboli repositáře. Následně je ověřeno zachování stávající funkcionality a koherence softwaru s novými částmi kódu během automatického sestavení softwaru (build) a provedením testů softwaru integračním serverem. (Shahin, 2015)

---

[10] Produkční prostředí – prostředí pro běh softwaru, který je používán pro dosahování business cílů

Obdobně definuje tuto problematiku i světově uznávaný expert Martin Fowler, který popisuje continuous integration jako praktiku, ve které každý člen vývojářského týmu integruje svou práci minimálně jednou, ale i několikrát za den. To vede k mnohočetným integracím. Každá vzniklá integrace je ověřena automatickým sestavením, které zahrnuje automatické testy tak, aby byly chyby či problémy identifikovány v co nejkratším možném čase. A dodává, že takovýto přístup vede ke znatelnému snížení integračních problémů a zrychlení vývoje kohezního softwaru.[11] (Fowler, 2006)

Continuous integration velice úzce souvisí s metodami agilního vývoje. Agilní vývoj softwaru je typickým iterativním přístupem v průběhu celého životního cyklu projektu, týmovou i mezi-týmovou spoluprací a komunikací se zákazníky softwaru. (Techopedia, 2019) Jak naznačuje *obrázek 5.* continuous integration na agilní vývoj navazuje a rozšiřuje jednotlivé oblasti. Praktikování continuous integration je v agilním vývoji často doporučováno pro maximální vytěžení jeho přínosů. (Hornbeek, 2015) Z uvedeného schématu lze odvodit, že continuous integration je oproti agilnímu vývoji zaměřeno i na integraci a testování softwaru, které je především uskutečněno automatickými testy na integračním serveru.



Obrázek 5: Návaznost continuous integration na metody agilního vývoje
Zdroj: Vora, 2016

[11] „A software development practice where members of a team integrate their work frequently, usually each person integrates at least daily—leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible. Many teams find that this approach leads to significantly reduced integration problems and allows a team to develop cohesive software more rapidly." M. Foller

Integrační server a testování softwaru lze tak definovat jako jednu z hlavních aktivit, přidaných hodnot continuous integration. Některé zdroje o continuous integration hovoří jako o souhrnu vývojářských nástrojů, které činí vývoj softwaru efektivní, a jako hlavní nastroj je označován právě integrační server. Ten je též přímo nazýván jako CI server neboli continuous integration server. (Packer, 2019)

Přívlastek continuous je často v takovémto pojetí překládán do češtiny jako kontinuální nebo nepřetržitý. Pod těmito slovy je snadné si představit neustálou událost. Paul M. Duvall však ve své knize Continuous integration zdůrazňuje, že procesy continuous integration nejsou nepřetržité či nekonečné, ale že všechny procesy se spouštějí na základě svého triggeru[12] a po vykonání všech stanovených kroků, je proces ukončen, dokud nebude opět inicializován. (Duvall, 2007) Nejčastěji se může jednat o nové změny ve zdrojovém kódu, které spustí celý integrační proces. Po jeho dokončení integrační server vyčkává na další zavedenou integraci, která by jej opět spustila. Tedy procesy nejsou nepřetržité, ale spíše plynulé, na sebe navazující či automaticky spuštěné.

O automatiky spuštěném procesu hovoří i B. Fitzgerald. Continuous integration je podle jeho slov zpravidla automaticky spuštěný proces zahrnující kroky ověření a kompilace zdrojového kódu, spuštění jednotkových a akceptačních (integračních) testů, vyhodnocení Code coverage[13] a sestavení instalačních balíčků. To vše obstarává automatický build, který je prováděn na integračním serveru. (Fitzgerald, 2017)

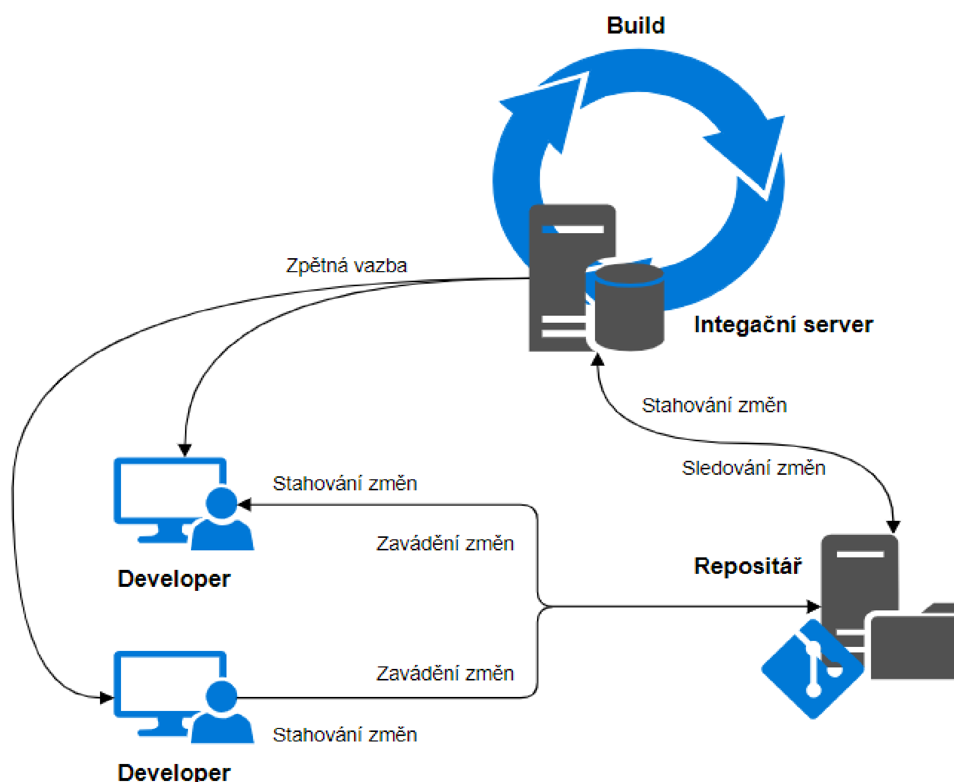
3.2.3 Continuous integration workflow

Jak již bylo ve výše uvedených definicích řečeno, continuous integration je soubor praktik, které vedou k zefektivnění vývoje softwaru. Pro identifikaci jednotlivých praktik je vhodné nejprve porozumět základnímu procesu.

[12] Trigger – označení pro spouštěč procesu, akce vyvolávající události

[13] Code coverage – metoda, která stanovuje, jaké procento zdrojového kódu (metod a funkcí) je otestováno jednotkovými testy

Následující workflow[14], které znázorňuje *obrázek 6*, zachycuje elementární prvky, které se v continuous integration vyskytují.



Obrázek 6: Continuous integration workflow
Zdroj: vlastní zpracování

Základní proces začíná členem vývojovému týmu, který zpracovává zadanou úlohu, provádí refactoring či jiným způsobem mění lokální kopii zdrojového kódu. Po dokončení odevzdává práci, tj. změny ve zdrojovém kódu, do sdíleného repositáře. CI server neustále detekuje změny ve sdíleném repositáři, v případě příbytku nových změn automaticky spustí build se softwarovými testy. Build během procesování neustále poskytuje zpětnou vazbu všem členům vývojářského týmu, tak aby měli přehled o úspěšnosti integrace změn, a tedy o aktuálním stavu softwaru.

[14] Workflow – pracovní schéma; technologický postup znázorňující komplexní činnosti procesu

1) **Vývojář** – člen vývojového týmu, softwarový pracovník, stojí na začátku celého procesu. Nejčastěji provádí změny na lokální kopii repositáře, kterou si před zahájením práce pravidelně aktualizuje, a to stažením změn ze sdíleného repositáře. Po dokončení práce integruje změny zpět do sdíleného repositáře. Integrace změn se též označuje jako commit, do češtiny přeloženo jako odsouhlasení či potvrzení. (Duvall, 2007) Tento krok může provádět velké množství vývojářů současně. Pokud se vývojář dostane do sporu se změnami svých kolegů, musí takové konflikty vyřešit, v jiném případě zapříčiní neúspěšný build softwaru.

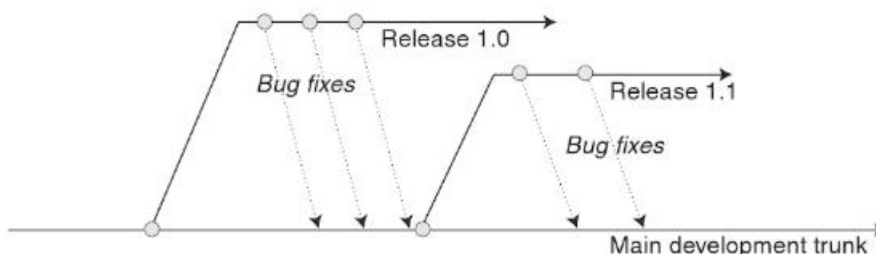
2) **Systém správy verzí** – version control system neboli verzovací systém zaznamenává změny souboru nebo sady souborů v průběhu času. Jedná se o esenciální prvek celého procesu, který dovoluje souběžnou spolupráci více vývojářů a poskytuje přístup k vytvořeným verzím softwaru. Při verzování zdrojového kódu uchovává historii změn každého řádku kódu, a také informaci kdy a kým byly změny vytvořeny. Při integraci změn z lokálního repositáře do sdíleného uložiště verzovací systém vytvoří revizi[15] s unikátním identifikátorem. Tyto revize lze kdykoliv obnovit, což může sloužit k odstranění nežádoucích změn a konfliktů. (Chacon, 2014)

   Verzovací systémy umožnují práci na různých verzích projektu souběžně. Vývojová větev neboli branch je často označována jako hlavní (mainline) nebo vedlejší. V hlavní větvi se nejčastěji udržuje plně funkční software bez větších rozpracovaných částí, oproti tomu vedlejší větev slouží pro vývoj a odzkoušení různých druhů řešení. Tento způsob dovoluje separátní a souběžný vývoj nových funkcionalit mimo hlavní vývojovou větev a minimalizuje tak riziko jejich znehodnocení. Paralelní vývojové větvě

[15] Revize – commit; changeset, stav projektu v určitém čase s unikátním označením

je možné kdykoliv integrovat pomocí metody merge[16], nebo mohou nepotřebné větve zcela zaniknout. (Fowler, 2006)

Díky verzovaní je možné vydávat nové verze softwaru, a přitom udržovat a spravovat starší verze souběžně. *Obrázek 7* nastiňuje souběh takového zachování aktivních větví pro udržování starších verzí softwaru. V rámci podporování starších verzí softwaru je potřeba opravovat objevené bugy.



Obrázek 7: Udržitelnost stávajících verzí softwaru
Zdroj: Gregory, 2016

Verzovací systém je dnes samozřejmostí a standardem každého softwarového projektu, nejen z hlediska efektivního týmového vývoje, ale i pro zálohování projektu. Mezi nejvíce rozšířené systémy správy verzí patří Git, Subversion nebo Mercurial.

3) **Sdílené uložiště zdrojového kódu –** sdílené uložiště, též označováno jako repositář, je velmi úzce spojeno se systémy správy verzí. Jedná se o fyzické uložiště, ve kterém jsou uchovávány všechny vzniklé revize a do kterého jsou integrovány nové změny všemi členy vývojářského týmu. Zároveň jsou z tohoto uložiště stahovány změny nejen členy vývojářského týmu, ale i integračním serverem. (McKenzie, 2018)

   Repositář se také často označuje přívlastkem centrální, a to z důvodu uchovávání všech schválených verzí na jednom místě. Důležitou částí je zajistit přístup do takového repositáře všem vývojářům z různých lokací,

[16] Merge – spojení; metoda, která spojuje změny dvou vývojových větví

například z různých vývojových středisek. K tomu slouží provozování repositáře jako webové služby, která zpřístupňuje online přístup odkudkoliv. Mezi takovéto služby se například řadí GitHub, Gitlab nebo Bitbucket. Repositáře jsou velmi často spjaty s konkrétním typem verzovacího systému.

4) **Integrační server –** integrační server, build server nebo též CI server je též jako systém správy verzí jedním z elementárních prvků continuous integration. Jedná se o softwarový nástroj, který představuje výpočetní jednotku pro uskutečnění automatického sestavení softwaru a spuštění automatických testů. (Shahin, 2015)

   Integrační server dokáže sledovat změny ve sdíleném repositáři. V závislosti na zvoleném intervalu, integrační server sleduje nové revize na zvolené vedlejší větvi, nejčastěji však na větvi hlavní (mainline). Pokud do repositáře přijde nová revize, stáhne změny a spustí build. Pro stanovení dalších samostatných automatických kroků slouží build script nebo posloupnost jednotlivých kroků, které se mají vykonat.

   Integrační server přehledně zobrazuje jednotlivé buid procesy různých vývojových větví softwarového projektu. Dále zobrazuje výsledky, průběžné logování i historii automatických testů a poskytuje tak zpětnou vazbu vývojáři o aktuálním stavu jeho integrace.

5) **Automatický build –** automatický build představuje sestavení zdrojového kódu do spustitelné podoby. Tento proces zahrnuje nejen kompilaci[17] a vytvoření instalačních balíčků, ale i veškerou potřebnou práci se soubory (dokumentace, návody), přepsání čísla aktuální verze, spuštění jednotkových či integračních testů a mnoho dalších operací. Jedná se tedy o komplexní proces, ve kterém vzniká kompletní podoba softwaru, jež je možné použít. (Duvall, 2007)

[17] Kompilace – proces, při kterém počítač převede programátorem napsaný zdrojový kód do spustitelné (binární) podoby

Součástí sestavení softwaru jsou i automatické testy. V závislosti na druhu testu jsou spouštěny přímo v průběhu sestavení softwaru či bezprostředně po jeho dokončení. V tomto případě se může například jednat o systémové testy, které ověřují novou a spustitelnou verzi softwaru. Testy ověřují funkcionalitu a chování softwaru a detekují případné nesrovnalosti a špatné chování. Jejich úspěšnost může být považována za indikátor kvality aktuálního softwaru.

V tomto pojetí se také často setkáváme s pojmem privátní build. Privátní automatický build může vývojář spustit před vykonáním samotné integrace změn do sdíleného repositáře. To lze z části uskutečnit na lokálním stroji, komplexnější ověření je spuštění automatických testů nad zdrojovými kódy vlastní vedlejší větve (branch) na integračním serveru. Tento krok zajistí úspěšnost budoucí verze a odhalí její nedostatky ještě před provedením integrace. Integraci může vývojář v takovém případě odložit, dokud neopraví chyby, které testy odhalily. (Duvall, 2007)

Pro automatický build softwaru lze použít specializované nástroje sestavení. Mezi takové nástroje především patří Gradle, Maven nebo Ant.

6) **Zpětná vazba** – Celý proces se uzavírá zpětnou vazbou vývojáři, který se v co nejkratším možném čase dozvídá, zdali byla jeho integrace změn úspěšná či nikoliv. Zpětnou vazbu vývojář může dostávat již během všech stádií sestavení, při vzniku problému s integrací však musí být neprodleně informován. Sledovat stav jeho integrace je možné libovolnými informačními kanály. Informaci o úspěšném sestavení softwaru či systémových testech může přehledně zobrazit integračních server jako takový nebo v podobě mailu, notifikace atd. Je taktéž nutné zajistit pravidelnost automatického průběžného informování, aby nejen vývojáři, ale i manažeři mohli provádět svá plánování a rozhodování o dalších činnostech. (Codeship, 2019)

   Velmi důležitým a mnohdy opomíjeným bodem po obdržení negativní zpětné vazby (neúspěšný build) je okamžitá náprava rozbitých částí softwaru

tak, aby další člen vývojářského týmu mohl provést svou vlastní integraci. Každé selhání integrace je událost, která může mít řádu viditelných indikátorů, aby zodpovědné osoby mohly co nejrychleji vyřešit vzniklé problémy. (Shahin, 2015)

3.2.4 Hlavní procesy continuous integration

Z výše uvedeného workflow lze syntetizovat několik základních praktik, činností a doporučení, které je vhodné v continuous integration dodržovat pro dosažení jeho pozitivních přínosů. V prvé řadě to jsou (Duvall, 2007; Fowller, 2006; Dawson, 2018):

- **Časté integrování změn –** průběžné zavádění nových změn do softwaru je jednou z nejdůležitějších činností. Pokud je nový kód integrován často a v malých dávkách, je integrace úspěšnější než při integrování rozsáhlých změn. Členové vývojářského týmu by se tak měli vyhnout velkým zásahům do stávající funkcionality a struktury. Pokud k takovýmto krokům musí dojít, je třeba práci rozdělit na jednotlivé logické úkoly. Postupná integrace takovýchto úkolů eliminuje skryté problémy a konflikty při slučování vývojových větví.

- **Neintegrovat nekompilovatelný/nefunkční kód –** toto doporučení se jeví jako samozřejmé, avšak nemusí tomu tak vždy být. Před commitem do sdíleného uložiště by měl vývojář ověřit svůj kód alespoň lokální kompilací, případě spuštěním základní sady automatických testů. Pokud vše proběhlo v pořádku, může vývojář přistoupit k samotné integraci svých změn.

- **Neprodleně opravit rozbitý build –** tento bod se týká automatického sestavení softwaru po neúspěšné integraci. Pokud build selže, jsou členové vývojářského týmu informování prostřednictvím CI serveru. V takovémto případě je nutné, aby byla vzniklá chyba ve zdrojovém kódu neprodleně opravena a následující build byl úspěšný a stabilní. Neúspěšný build totiž může blokovat další členy vývojářského týmu, kteří chtějí integrovat své

změny a ověřit tak správnost a soudržnost softwaru. Dále přetrvává riziko zanesení té samé chyby do různých vedlejších vývojových větví, ve kterých mohou opět působit potíže.

- **Psát automatické testy** – součástí automatického sestavení jsou i různé druhy automatických testů, které ověřující chování a funkčnost softwaru. Vysoké pokrytí funkcionality softwaru automatickými testy snižuje riziko opomenutí bugu, který mohl při měnění zdrojového kódu nastat.

- **Všechny automatické testy musí projít** – automatické testy, které byly spuštěny po zavedení nové integrace, musí být úspěšné. Testy ověřují stávající chování softwaru a jsou hlavním indikátorem úspěšnosti integrace. Je proto nezbytné, aby byly úspěšné.

- **Neprodleně opravit rozbité automatické testy** – pokud testy selhaly, byť jen v malém měřítku, je potřeba je co nejdříve opravit. V tomto bodě mohou nastat dvě možnosti. První možností je oprávněné selhání testů – testy selhaly z důvodu nechtěné změny chování softwaru či zaviněné chyby. V tomto případě je nezbytné opravit zdrojový kód, tak jako v případě selhání automatického sestavení softwaru. V druhém případě testy selhaly kvůli záměrné změně chování softwaru, ale před integrací nebyly upraveny automatické testy, aby mohly testovat nové chování softwaru. V toto případě je nutné neprodleně opravit či pozměnit automatické testy.

- **Vysoká kvalita kódu** – do této kategorie spadá velké množství užitečných rad a doporučení, kterými by se měli řídit vývojáři při psaní kódu. Jedná se o běžné praktiky psaní čitelného, rozšiřitelného a znovu použitelného kódu. Dále je jeho kvalita dosahována dodržováním obecných postupů a standardů daného programovacího jazyka. Rovněž je pak v každém softwarovém projektu zvykem dodržovat určité standardy a domluvy, které vznikly napříč vývojovými týmy. Ke kontrole kvality kódu a dodržování stanovených postupů může sloužit praktika code review. Jedná se o proces, ve kterém jsou změny kódu předvedeny a objasněny

zkušenějšímu členu vývojářského týmu, který uvede připomínky a poskytuje konstruktivní zpětnou vazbu. Praktikováním těchto doporučení se předchází nefunkčnímu a nekompilovatelnému kódu.

- **Spouštět privátní build** – v některých případech je vhodné před integrací vedlejší vývojové větve do hlavní, či před integrováním větších změn do sdíleného uložiště spustit privátní build. To vývojář může udělat na svém lokálním stroji nebo na integračním serveru. Na integračním serveru spustí build na své vývojové větvi, do které jsou zahrnuté i nejnovější změny z hlavní vývojové větve. Aktuálností svého zdrojového kódu vývojář zajistí kompatibilitu svých změn se současnou verzí softwaru. Pokud build takovéto větve bude úspěšný, může vývojář své změny integrovat.

- **Rychlé ukončení sestavení** – automatický build se skládá z určitých kroků, které se provádějí jak paralelně, tak sériově. Pokud jeden z takovýchto kroků není úspěšný, celý build by se měl co nejdříve ukončit. Například se může stát, že zdrojový kód není kompilovatelný nebo automatické testy selhaly fatálním způsobem a je již zcela zřejmé, že výsledná verze softwaru bude funkční s omezeními či bude zcela nefunkční.

- **Zajištění rychlé zpětné vazby** – v continuous integration je potřeba zajistit rychlou a efektivní zpětnou vazbu o výsledku automatického sestavení i automatických testech, a to nejlépe ve všech stádiích integrace. Zpětná vazba neslouží jen pro informaci vývojáři, který provádí integraci, ale je dostupná i pro všechny členy vývojového týmu, tak aby mohli na jejím základě plánovat své budoucí integrace.

- **Udržovat prostředí CI serveru** – prostředí na integračním serveru je potřeba spravovat a podle potřeby udržovat. Například se může jednat o pravidelné aktualizování zde nainstalovaných programů, čištění disků a optimalizace výkonu. V neposlední řadě se může jednat o externí systémy potřebné pro úspěšné testování verze softwaru automatickými testy. Nestabilní prostředí může vést k anomáliím a nestabilitám sestavení.

3.2.5 Přínosy continuous integration

Vykonáváním výše uvedených praktik z continuous integration lze dosáhnout celé řady přínosů, které přímo i nepřímo ovlivňují vývoj softwaru. Jednou z hlavních výhod je snížení rizika, automatické generování spustitelného softwaru, redukce opakujících se procesů, zvýšení přehledu nad projektem, zvýšení důvěry v software.

Snížení rizika – snížení rizika je jedním ze základních přínosů continuous integration, přestože se může jednat o velice abstraktní pojem. Pod tímto pojmem můžeme v prvé řadě rozumět (Duvall, 2007):

- snížení rizika nesoudržné nebo nespustitelné verze softwaru,
- snížení rizika pozdního odhalení problémů a chyb,
- snížení rizika zanesení nesprávného chování softwaru,
- snížení rizika neprůhlednosti projektu.

Integrování změn několikrát za den a následné ověření automatickou operací build, jejíž součástí je spouštění automatických testů, včasně odhaluje problémy v softwaru. Dochází tak ke hlídání kvality zdrojového kódu a spustitelného softwaru jako takového. Rozsahově malé změny ve zdrojovém kódu přirozeně snižují chybovost členů vývojářských týmů, neboť při malých úpravách lze lépe odhalit problémy ještě před integrací. Díky malým změnám existuje vysoká šance, že nevzniknou nevyžádané problémy ve zdrojovém kódu při integraci – tzn. vypořádání se se cizími změnami. Dále pak existuje vysoká pravděpodobnost, že build bude úspěšný. Případné problémy jsou rychle reportovány a vzniklé chyby jsou v brzké době odstraněny. (CodeShip, 2019)

Generování spustitelného softwaru – continuous integration umožnuje jednoduše tvořit plnohodnotné spustitelné verze softwaru v průběhu celého vývojového cyklu. Všichni členové vývojového týmu tak mají rychlý přístup k nejnovější verzi softwaru. (Packer, 2019) Díky tomu také lze vydat a distribuovat novou verzi softwaru téměř ze dne na den. Ovšem za předpokladu, že automatický build po poslední integraci změn byl úspěšný a testy neodhalily žádné problémy.

Tato schopnost je užitečná obzvláště při vydávání hotfix verze[18] všech stávajících podporovaných verzí daného softwaru. V neposlední řadě je možné díky jednoduchému vydávání verzí často dodávat zkušební verze zákazníkům, kteří mohou poskytovat důležitou zpětnou vazbu.

Redukce opakujících se činností – continuous integration automatizuje či dokonce eliminuje některé z častých opakujících se procesů a činností spojených s vývojem a vydáváváním softwaru. (Pittet, 2019) Automatizace takovýchto procesů šetří čas, náklady a úsilí vývojářskému týmu a celkově zrychluje vývoj. Mezi hlavní důsledky automatizace patří:

- redukce periodických manuálních činností,
- redukce manuálního sestavování softwaru,
- šetření lidských zdrojů při manuálním testování softwaru,
- zkrácení doby vydávání nové verze,
- zkrácení doby při vydávání hotfix verzí,
- prediktabilní automatický build.

Lepší přehled nad projektem – Díky elementárním prvkům continuous integration je možné zlepšit transparentnost a přehled o aktuálním stavu celého projektu. V průběhu vývojového cyklu softwaru je tak stav projektu měřitelný. Zpětná vazba, informace z integračního serveru a výsledky automatických testů poskytují všem členům přehled o aktuálním stavu projektu a stavu předchozích integrací. Na jejich základě pak mohou lépe plánovat další integrace či rozhodovat o dalších změnách vývoje. Continuous integration především přináší (Duvall, 2006):

- přehled všech členů týmů o aktuálním stavu integrace,
- přehled všech členů nad stavem celého projektu,
- přehled následků neúspěšné integrace,
- přehledné verzování jednotlivých verzí,

[18] Hotfix – Opravná verze softwaru, která nepřináší nová rozšíření, ale opravuje nalezené chyby a problémy. Většinou je jeho aplikace prováděna formou update nikoliv celé nové instalace.

- sledování stavu projektu v čase,
- rychlý přístup k poslední verzi,
- technický dohled nad projektem,
- zlepšení důvěry ve stabilní a nastavitelný software,
- zvýšení vývojářské jistoty v plně funkční produkt bez chyb.

3.2.6 Problémy s continuous integration

Continuous integration nepochybně přináší mnoho pozitiv a benefitů, zároveň existují problémy spojené nejen se zaváděním tohoto procesu a to i přesto, že po překonání počátečních problémů ulehčuje každodenní práci všem členům týmu. Mezi takové patří:

- nesprávné použití integračního serveru,
- náklady vzniklé se zřízením a provozováním integračního serveru,
- náklady na automatizaci a údržbu automatických testů,
- narůst povinností spojených s údržbou a konfigurací integračního serveru,
- implementační problémy se sestavením softwaru a automatickými testy,
- problémy se zaváděním nových procesů a postupů,
- nepřijetí nových procesů členy vývojářského týmů.

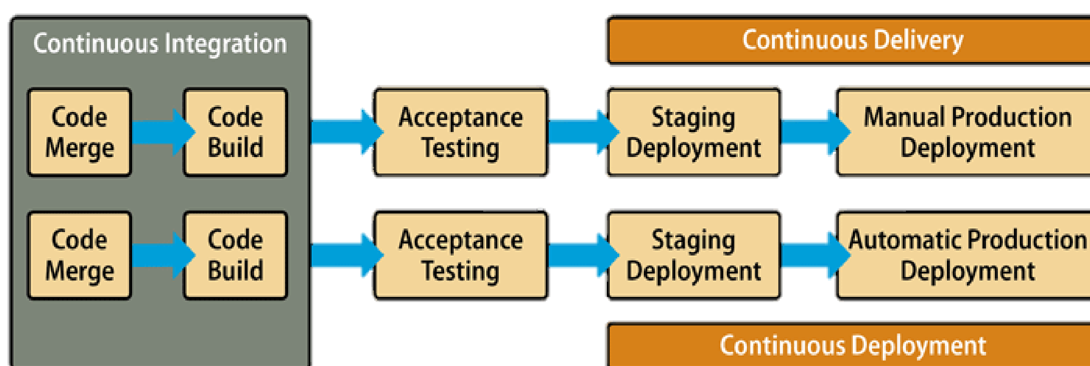
3.2.7 Continuous delivery a deployment

Continuous delivery – neboli průběžná dodávka je následující stupeň continuous integration v rámci přístupu continuous software engineering. K průběžné dodávce vede kompletní splnění praktik a metodik continuous integration, zautomatizování všech potřebných procesů včetně kompletního pokrytí funkcionality a koherence softwaru automatickými testy. Jinými slovy procesy continuous delivery lze zavádět až po zvládnutí celé domény continuous integration.

Jestliže je software spustitelný v průběhu celého vývojového cyklu, o což continuous integration usiluje, závěrečná fáze spojená s vydáváním verze nepřináší žádné neočekávané překážky. *Obrázek 8* znázorňuje vzájemný vztah mezi continuous integration a continuous delivery. (Pittet, 2019) Continuous delivery tedy spočívá v automatické dodávce softwaru zákazníkovi/uživateli se všemi

příslušnými dodatky, tj. dokumentací, uživatelskými a instalačními manuály, nasazovacími skripty či demo solutions[19]. Pod automatickým doručením softwaru se rozumí zpřístupnění verze zákazníkům například nahráním verze do sdílených systémů správy souborů či pomocí webových stránek, kde si verzi zákazník pohodlně vyzvedne či odkud si ji stáhne. Na *obrázku 8* jsou taktéž znázorněny hlavní dodatečné procesy:

- akceptační testy – poslední část testování softwaru,
- nasazení do pracovního prostředí – automatické nasazení do testovacího prostředí zákazníka, které přesně odpovídá produkčnímu prostředí,
- nasazení do produkce – manuální činnost, ve které je uveden software do produkce, zákazník jej plnohodnotně používá pro své business aktivity.



Obrázek 8: Continuous delivery, continuous deployment
Zdroj: Kerner, 2019

Spouštěčem procesu nemusí být typicky nová integrace změn do sdíleného repositáře tak, jako tomu je v continuous integration. Spouštěč je v tomto procesu typicky člen vývojářského týmu, který je obeznámen o stavu všech rozpracovaných částí softwaru. U každého softwarového projektu je více či méně naplánován systém dodávky softwaru v časově ohraničených cyklech. Na konci cyklu, kdy jsou zadané úkoly hotovy, lze spustit proces manuálně.

[19] Demo solutions – modelové příklady použití softwaru, ukázkové řešení

Continuous deployment – neboli průběžné nasazení je následující stupeň continuous delivery. A opět vyžaduje zvládnutí a zavedení procesů z continuous integration a nyní i continuous delivery. V tomto přístupu vyvíjení a vydávání softwaru jsou zautomatizovány veškeré procesy počínaje novou integrací změn, přes automatické testování a dodání, až po nasazení softwaru do produkčního prostředí. (Kerner, 2019) Což je i jedním z hlavních odlišení od continuous delivery, kde nasazení do produkčního prostředí probíhá automaticky, viz *obrázek 8*. V takovýchto případech se jedná například o webové či cloudové služby, které zákazníci využívají. Například on-premise[20] aplikace by bylo velice obtížné automaticky nasazovat do prostřední zákazníka.

3.3 Continuous integration server

Kapitola představuje CI server jako elementární nástroj continuous integration, jeho základní vlastnosti, možnosti a funkce, kterými disponuje. Dále popisuje přední dostupné integrační servery a jejich základní prvky.

3.3.1 Společné znaky CI serverů

Jak již bylo v předchozí kapitole uvedeno, integrační server představuje výpočetní a procesní jednotku v continuous integration, která provádí automatické úkony. Jedná se o softwarový nástroj uzpůsobený především k vykonávání build operací, spouštění a uskutečňování automatických testů a poskytování zpětné vazby ohledně vyprodukovaných výsledků. Integrační server tak umožnuje ověření integrace v čistém, unifikovaném a stabilním prostředí. Mezi klíčové operace, které jsou na integračním serveru vykovávány, patří (Excella, 2019):

- monitorování repositáře zdrojového kódu,
- automatické a plánové spouštění procesů,
- automatické sestavení softwaru,
- spouštění automatických testů,

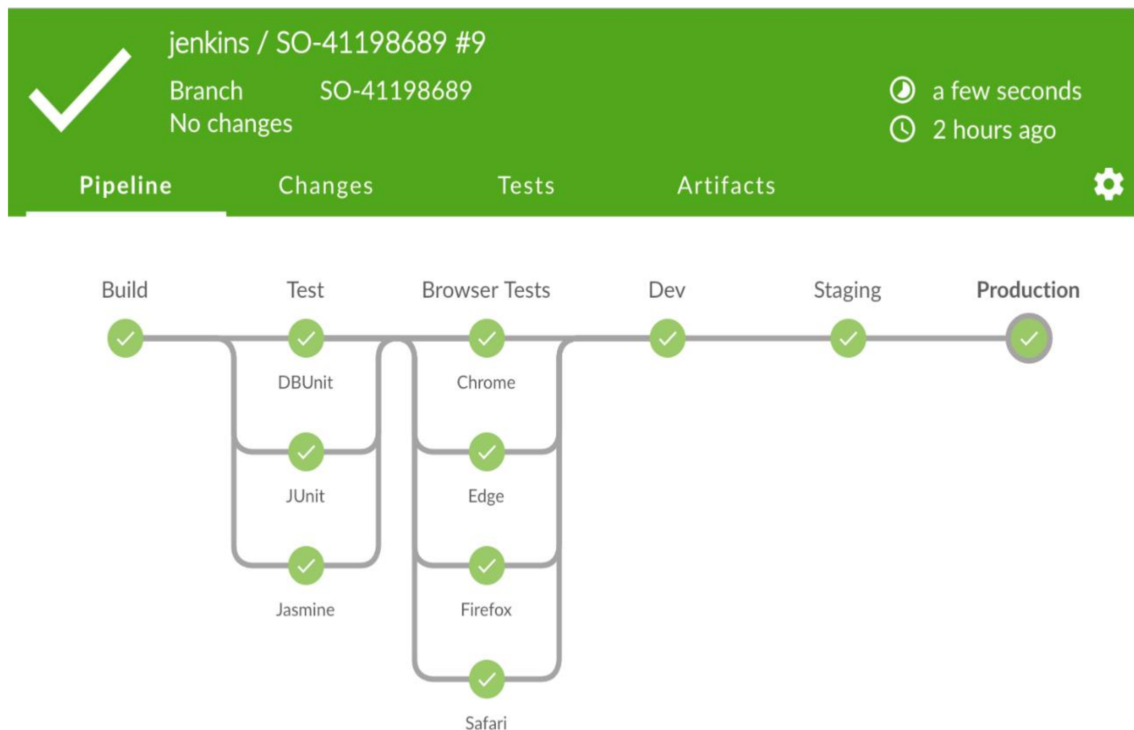
---

[20] On-premise – označení pro druh softwaru, který lze instalovat a provozovat na lokálním stroji v prostředí zákazníka

- provádění analýz získaných výsledků,
- poskytování informací, notifikací a zpráv,
- ukládání souborů vygenerovaných během sestavení.

Integrační servery mají také několik vlastností a funkcí pro efektivní vykonávání výše uvedených činností. Těmito vlastnostmi disponují téměř všechny moderní dostupné integrační servery:

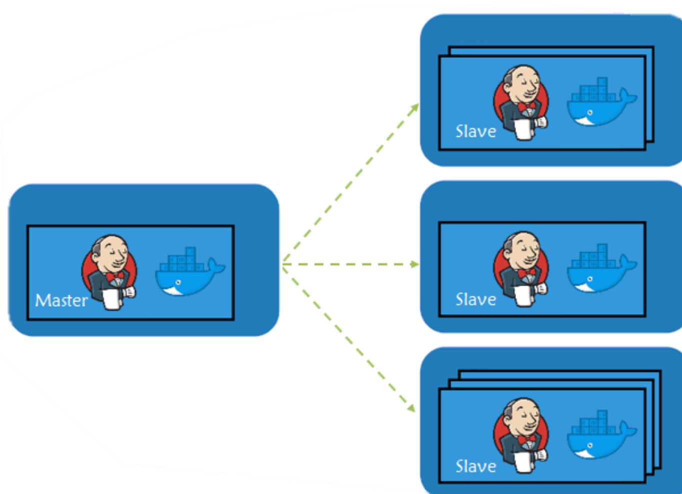
Build skript – jak již bylo v předchozích kapitolách uvedeno, jedná se většinou o sadu či řetězec instrukcí a kroků, které jsou vykonávány integračním serverem. Operace mohou být prováděny paralelně, což umožňuje dosahovat rychlejšího ověření. Zároveň může výstup dokončeného kroku sloužit jako vstup následujícího kroku. Součástí základního build skriptu integračního serveru může být build software, dílčí skupiny jednotkových nebo systémových testů a další operace. Build skript – pipeline integračního Serveru Jenkins zobrazuje *obrázek 9.*



Obrázek 9: Build skript – Jenkins Blue Ocean pipeline
Zdroj: Jenkins, 2019

Rozšiřitelná architektura – s narůstající velikostí softwarového projektu a s narůstajícím počtem členů vývojářského týmu, kteří zavádějí své integrace a následně je potřebují ověřit, narůstá počet procesů na integračním serveru. Ten může přestat stíhat procesovat všechny zadané požadavky. Proto je většina integračních serverů připravena na přidání další procesující jednotky. Často existuje jeden hlavní uzel (node) a velké množství podřízených jednotek, tak jak znázorňuje *obrázek 10*. Hlavní uzel je pak centrem pro uživatelskou interakci, konfiguraci a agregaci informací z ostatních podřízených jednotek.

Centrální konfigurace – centrální konfigurace velice souvisí s rozšiřitelnou architekturou integračního nástroje. Předchozí bod pojednával o přidání nové výpočetní kapacity, pokud je zapotřebí. Centrální konfigurace slouží k jednoduchému nastavení všech takovýchto kapacit. Podřízené jednotky zkrátka přejímají nastavení, které je hromadně definováno na centrálním uzlu. Jednotky pak mohou být zařazeny do různých skupin s různým nastavením. Důležitou vlastností je, že veškeré nastavení je řízeno z centrálního místa – z řídícího uzlu (node).



Obrázek 10: Rozšiřitelná architektura CI Jenkins
Zdroj: Rus, 2018

Distribuce úkolů – centrální/řídící uzel také kromě centrální konfigurace distribuuje na své podřízené jednotky i úlohy, které je potřeba vykonat. Úlohy rozdává na základě volných kapacit podřízených jednotek podle definované

specifikace nebo na základě požadavku vykonat určitý úkol na určité jednotce se speciálním nastavením. Řídící uzel taktéž může procesovat úlohy, je-li to potřeba. Řídící uzel však především integruje výsledky podřízených jednotek a poskytuje zpětnou vazbu tak, aby byla data opět na jednom místě.

Způsoby spouštění procesů – Integrační servery poskytují několik způsobů, jak spustit definované procesy, což je jedním z jejich klíčových úkolů. Mezi ty nejzákladnější patří:

- On-demand[21] spuštění – proces může být spuštěn manuálně podle aktuální potřeby vývojáře, například se může jednat o ověření vlastní vývojové větve.
- Plánované spouštění – procesy jsou naplánovány v pravidelných frekvencích. Spouštění takových procesů je například vhodné u starších verzí, do kterých není často integrováno, ale zároveň je potřeba průběžně ověřovat, že je vše funkční.
- Automatické spouštění – procesy jsou spuštěny na základě nové události. Událostí může být například identifikace změny v repositáři zdrojového kódu.

Přehled výsledků testů – CI servery kromě poskytování zpětné vazby o výsledcích automatického sestavení softwaru, automatických testů a ostatních procesů, disponují moduly, které přehledně zobrazují, jaké testy byly neúspěšné. Mimo to zpracovávají log[22] softwaru, ale i log testů, pokud to testy umožnují, a tím se snaží vysvětlovat, proč testy byly neúspěšné. Dále na základě historie předchozích průběhů dokáže určit, od jakého okamžiku je daný test neúspěšný či nestabilní. Takovéto analytické moduly pomáhají identifikovat problémy, které se nemusí vždy nastat.

Artefakt management – jak již bylo dříve uvedeno, během automatického sestavení softwaru vznikají různé artefakty. Nejčastěji mohou být tímto pojmem

[21] On-demand – označení pro bezprostřední vykonání zadaného požadavku
[22] Log – obecné označení pro záznam činností softwaru v průběhu jeho běhu

myšleny spustitelné verze softwaru spolu s dalšími pomocnými knihovnami a soubory. Některé artefakty jsou často potřebné pro další stádia integračního procesu, pro další využití. Takové artefakty je nutné uchovat, dokud si je integrační server zase nevyzvedne pro svou činnost. V jiném případě je potřeba některé vzniklé artefakty nahrávat na různá uložiště, například na FTP[23] server, kde mohou být zálohovány všechny integračním serverem vygenerované verze softwaru.

Integrace s dalšími programy – Integrační servery jsou často připravené na spolupráci s dalšími softwarovými produkty. Takovýchto produktů může být celá a někdy téměř nekonečná řada. Může se například jednat o cloudové technologie, mezi které především patří Amazon Web Service nebo Microsoft Azure. Dále mohou být CI servery spouštěny pomocí různých orchestrálních nástrojů. V takových případech jsou CI servery součástí kontejnerů. Mezi takové lze uvést Docker, Kubernetes či Openshift. Dále se může jednat o zmíněné FTP a SMTP[24] servery, které naopak mohou být i součástí integračního serveru.

3.3.2 Příklady CI serverů

CI servery jsou pro softwarové společnosti velkými tématy. V současnosti existuje na trhu celá řada CI serverů, které vznikly ať už za účelem komerčního využití nebo z iniciativy vývojářů jako pohnutka pro zlepšování interních procesů. Často takové nástroje vznikly právě jako malé open-source projekty. Open-source projekt je takový softwarový projekt, jehož zdrojový kód je veřejně dostupný a zároveň jej může kdokoliv kopírovat, modifikovat a opět redistribuovat bez speciálních podmínek. (Poba-nzaou, 2019) Tedy open-source CI servery mohou být neustále vylepšovány vývojáři z různých společností, a to především na základě vývojářských zkušeností s konfigurací a užíváním. Následující integrační servery byly vybrány jako hlavní zástupci této kategorie.

[23] FTP server – File Transfer Protocol server pro sdílení, přenos a ukládání souborů po síti
[24] SMTP server – Simple Mail Transfer Protocol server slouží pro zasílání emailových zpráv po síti

3.3.2.1 Jenkins

Jenkins je jedním nejrozšířenějším a nejznámějším integračním nástrojem. Jedná se Java open-source projekt, který vznikl v roce 2004, a jeho původní název byl Hudson. Jenkins představuje základní či klasický CI server, který se podle autorů jednoduše instaluje, konfiguruje a díky pluginům velice dobře rozšiřuje a uzpůsobuje konkrétním potřebám. (Jenkins, 2019) Základ CI serveru je možné rozšířit o nespočet pluginů, které pro Jenkins existují, například Blue Ocean pro lepší grafické znázornění a konfigurování pipeline. Jedná se o moduly, které je možné bezplatně stáhnout a zapojit, a tím obšírnou funkcionalitu integračního serveru ještě rozšířit nebo přizpůsobit aktuálním požadavkům softwarového projektu. Počet pluginů neustále narůstá, v současné době je jich přes tisíc čtyři sta. (StackShare, 2019) Pluginy často usnadňují každodenní práci a zjednoduší opakující se činnosti, tedy do doby, dokud nezačne být jejich správa časově nákladná.

Jenkins také disponuje velkou uživatelskou základnou, a to především díky jeho rozšířenosti a použitelnosti. Je tedy snadné dohledat odpovědi na problémy nebo postupy a rady ohledně různých druhů řešení i mimo oficiální dokumentaci. Základní konfigurace pro automatický build je stanovena v řídícím uzlu, která se příznačně nazývá master. Podřízené jednotky se označují pojmem slave. Pokročilejší build skript je označován jako pipeline (*obrázek 9*), ten se nejčastěji uchovává v repositáři spolu se zdrojovým kódem projektu v tzv. jenkinsfile, který je psán formátem YAML[25]. Jednotlivé procesy CI serveru jsou zde pojmenovány jako Joby. (Jenkins, 2019)

3.3.2.2 TeamCity

Dalším velmi rozšířeným a populárním integračním nástrojem je TeamCity. Tento nástroj vznikl na základě potřeb společnosti JetBrains při vývoji vlastních produktů založených na platformě Java a .NET. Software tedy v základu disponuje

[25] YAML – Ain’t Markup Language – jedná se uživatelky přívětivý standard pro serializaci strukturovaných dat

funkcionalitou, která je například u jiných integračních serverů dostupná v podobě pluginu. Server je také vyšperkován detaily, které vývojářům šetří čas. Podobně jako u Jenkins existuje i zde velké množství rozšíření. Dále pak podporuje vysokou integraci s ostatními JetBrains produkty, nejvíce s nástroji IntelliJ IDEA nebo YouTrack. V neposlední řadě TeamCity disponuje množstvím analyzačních nástrojů, které pomáhají rozklíčovat pády automatických testů a selhání sestavení.

Taktéž analogicky k Jenkins existuje server a jemu podřízení agenti. Posloupnost operací je prováděna pomocí build chain neboli řetězců. Jejich konfigurace je stanovena ve verzovaných souborech psaných v Kotlin DSL nebo XML, případně je konfigurace uložena na CI serveru. Největším negativem TeamCity je zpoplatnění. Server a první agenti jsou zdarma, další agenti jsou licencováni a tedy zpoplatněni. Malé a některé střední projekty si však s třemi agenty vystačí. (JetBrains, 2019)

3.3.2.3 Bamboo

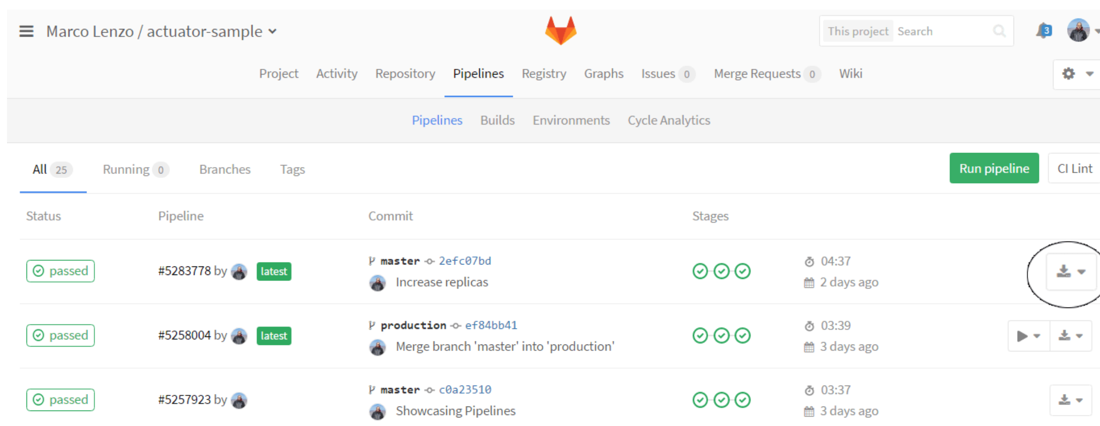
Jedním z dalších CI serverů současnosti je Bamboo od společnosti Atlassian. Atlassian, obdobně jako JetBrains, je velkým hráčem na trhu softwaru pro optimalizaci a zjednodušení firemních procesů spojených s vývojem softwaru. To se opět promítá do vyvíjeného integračního serveru Bamboo, který se velmi dobře integruje s dalšími produkty společnosti Atlassian, mezi ně především patří Jira, Sourcetree nebo Bitbucket. (StackShare, 2019)

CI server byl poprvé uveden na trh v roce 2007 jako odezva na Jenkins, se kterým se často srovnává. To podtrhuje i skutečnost, že je Bamboo připraveno na migraci nastavení z Jenkins v podobě jednoho kliknutí. Svého času byl provozován jako cloudová služba, ale od tohoto způsobu společnost upustila v roce 2017. Bamboo je též zpoplatněný integrační server, jehož cena se odvíjí od počtu zpracovaných operací, a v základní verzi je dost omezený. (Atlassian, 2019)

3.3.2.4 GitLab CI

Dalším známým CI serverem, který se stává velice populárním, je GitLab CI. Jedná se integrovanou službu přímo do webového či on-premise repositáře GitLab. Tato služba byla spuštěna v roce 2016, díky čemuž se stává jedním z nejmladších CI serverů. GitLab, stejně tak jako Jenkins, je open-source projekt, jehož komunitní verze je zdarma. Hlavní výhodou je umístění CI serveru, které je, jak již bylo řečeno, součástí aplikace. (StackShare, 2019) Tedy správa verzí zdrojového kódu a správa integračního serveru je na jednom místě, což přináší značné zjednodušení a šetření nákladů s provozováním samostatného CI serveru. Výhodou je rychlejší detekce změn v repositáři a též skutečnost, že revize není potřeba duplikovat na jiný server/zařízení. Oproti těmto zvýhodněním může být nevýhodou relativní mladost GitLab CI, kvůli které je komunita celkem malá, uživatelských zkušeností méně, a s tím i například spojený troubleshooting[26].

Jak již bylo řečeno, konfigurace a funkcionalita řídícího uzlu je již součástí. Procesní jednotky se nazývají jako runner. Ty lze spouštět pomocí technologie docker, což přináší celou řadu výhod, nebo je lze běžně naistalovat. Taktéž jako v případě CI Jenkins jsou zde použity pipeline, které jsou opět definované v YAML konfiguračním souboru. (GitLab, 2019)



Obrázek 11: Ukázka GitLab CI prostředí
Zdroj: GitLab, 2019

[26] Troubleshooting – řešení problémů spjatých s abnormálním chováním softwaru

4 Praktická část diplomové práce

4.1 Inspire Scaler

Kapitola představuje produkt Inspire Scaler vyvíjený společnosti Quadient, jeho základní problémy a nedostatky během vyvíjení, testování a vykonávání metodiky continuous integration.

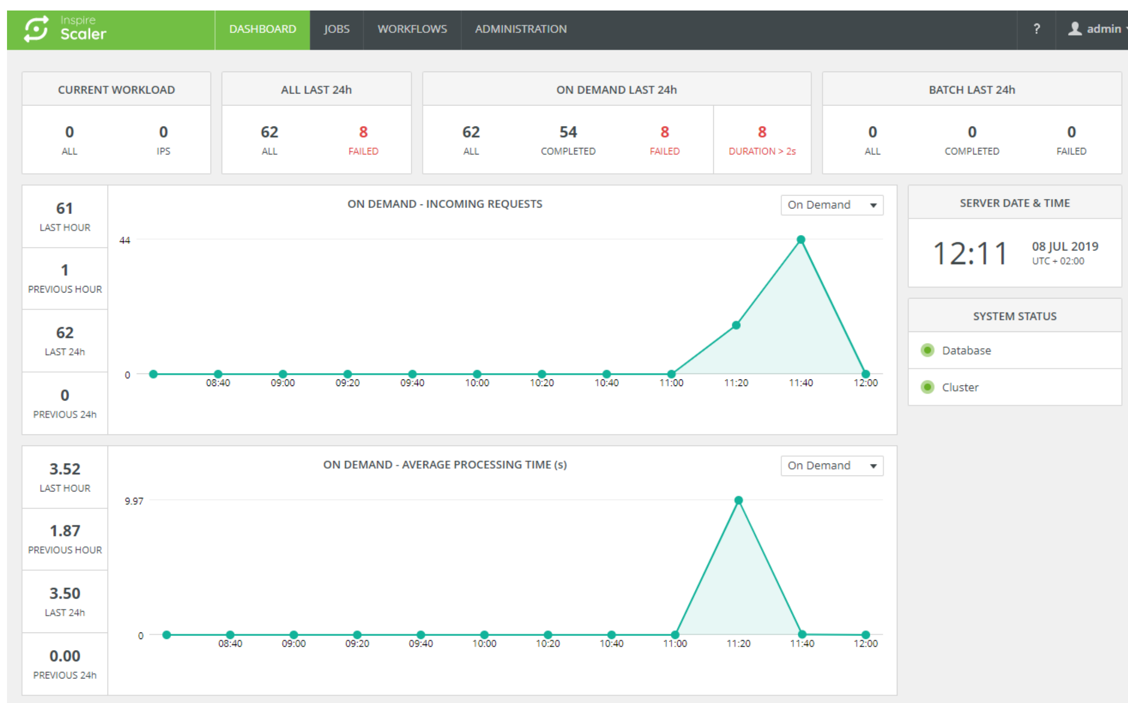
4.1.1 Popis produktu

Inspire Scaler je středně velký softwarový produkt společnosti Quadient s.r.o. Aktuální verze softwaru obsahuje více než dva a půl milionu řádků zdrojového kódu. Jedná se o nástroj, který vykonává roli procesní a integrační jednotky v celofiremních řešeních, která jsou nejčastěji implementována pro pojišťovací a bankovní sektor. Inspire Scaler spadá do kategorie CCM[27] nástrojů, a tedy jeho hlavním účelem je procesovat různé druhy komunikace mezi poskytovatelem služeb a zákazníkem. Zcela zjednodušeně přijímá a procesuje širokou škálu typů vstupů a generuje obšírnou škálu výstupů a dat. Jedná se tak o univerzální procesní jednotku, která je zároveň schopná navyšovat výpočetní výkon a odbavovat tak tisíce požadavků za minutu. Nejčastěji je integrován s dalšími Quadient CCM produkty (Inspire Interactive, Inspire Cloud), ale také například se systémy typu Java Messages Service, Amazon Web Service či Salesforce.

Projekt je založený na technologii JVM[28], operuje nad relačními databázemi a je spouštěn pomocí aplikačního kontaineru Tomcat. Grafické rozhraní je zobrazeno pomocí webového prohlížeče. Backend zdrojového kódu Inspire Scaler je psaný v programovacím jazyce Kotlin – statický typovaný jazyk podobný jazyku Java. (Kotlin, 2019) Frontend je pak napsán v Bobrilu. Bobril je komponentově orientovaný framework inspirovaný JavaScriptovou knihovnou React Js.

[27] CCM – Customer Communication Management – nástroje, které společnostem umožnují efektivně řídit komunikaci se zákazníkem skrze různé druhy komunikačních kanálů
[28] JVM – Java Virtual Machine – platforma, která umožnuje spouštět programy napsané v jazyce JAVA na virtuálním stroji

Jedná se o interní framework společnosti Quadient, který vznikl na základě iniciativy zaměstnanců, kteří byli nespokojeni se současnými JavaScriptovými knihovnami.



Obrázek 12: Inspire Scaler – dashboard
Zdroj: vlastní zpracování

Vývoj softwaru je organizován a řízen agilní metodikou Scrum. Scrum je nejrozšířenější procesní rámec pro vývoj, doručování a údržbu komplexních systémů, díky němuž se mohou členové vývojových týmů zabývat složitějšími adaptivními problémy a zároveň produktivně a kreativně vykonávat práci nejvyšší možné hodnoty. (Zuzi's blog, 2019) Vývojový cyklus je tedy rozdělen na jednotlivé sprinty. Po sérii osmi sprintů následuje proces vydávání nové verze neboli release, před kterým je nutné dokončit všechny rozpracované funkce. V tento okamžik se provádí regresní manuální testy pro ověření zachování funkcionality softwaru. Tomuto období je proto přezdíváno regresní testování. Během sprintů jsou také vydávány hotfix verze na starší podporované verze v závislosti na existenci externích či interních bugů. V průměru je hotfix vydáván pětkrát za toto období. Na vývoji produktu se podílí šest týmů, což představuje přibližně čtyřicet osob, z nichž je jedna pětina zaměřena na testování softwaru a udržování jeho kvality čili se jedná o quality assurance (QA) pracovníky.

4.1.2 Testovací projekt

Pro testovaní Inspire Scaler automatickými testy existuje pomocný samostatný testovací projekt, který je součástí zdrojového kódu. Jedná se o seskupení automatických systémových testů, testovacích metod a dalších pomocných tříd, které jsou potřebné k realizaci automatického testování Inspire Scaler. Mezi ně se především řadí konfigurační soubory, obecné testovací třídy s připravenými pomocnými metodami či testovací objekty a tzv. page objecty, které reprezentují jednotlivé stránky a jejich prvky. Součástí jsou pochopitelně i konfigurační soubory pro samotné CI servery.

Automatické testy jsou spouštěny pomocí CI server Jenkins. Z důvodu nestíhání vykonávání automatických systémových testů a z důvodu pokrytí různého testovacího prostředí (databáze, nad nimiž může být Inspire Scaler spuštěn), byly vytvořeny další tři virtuální stroje pro operační systém Windows a jeden virtuální stroj s operačním systémem Linux. Na těchto strojích je instalován CI Jenkins v základní konfiguraci, všechny CI servery jsou tedy na sobě nezávislé. Automatické testy se zpravidla spouští každý den na hlavní vývojové větvi. Testy jsou také vývojáři spouštěny i na vedlejších vývojových větvích, to kvůli preventivnímu ověření celého softwaru ještě před přidáním nové funkcionality do hlavní větve. Systémových testů je na aktuální verzi kolem jednoho tisíce, dále přes šest a půl tisíc jednotkových testů, ale jen něco málo přes čtyři sta padesát integračních testů – jedná se tedy o nešťastný testovací vzor přesýpacích hodin. V neposlední řadě existuje kolem dvou set sedmdesáti manuálních testů.

Současně se udržuje několik verzí softwaru Inspire Scaler. Jedná se o verze produktu 11.0, 12.0, 12.5 a v současnosti aktuální 14.0. Všechny tyto vývojové větve je potřeba na CI serveru více či méně často ověřovat. Například do starších verzí nepřibývá mnoho nových změn, proto není potřeba je často ověřovat.

4.1.3 Základní problémy testovacího projektu

Testovací projekt trpí několika základními problémy, které zbytečně znesnadňují vývojové cykly a přinášejí řadu dodatečných, opakujících se manuálních činností a mnoho vynaloženého úsilí. Mezi ně především patří:

1) **Opakující se manuální činnosti –** při testování Inspire Scaler se vykonává celá řada manuálních operací. Nejedná se pouze o provádění manuálních testů při vydávání nové verze (release), ale každodenní ruční procházení výsledků automatických, a to zejména systémových, testů.

2) **Doba trvání systémových testů –** provedení základní a jediné testové sady systémových testů trvá velmi dlouho, a to až sedm hodin. Sada systémových testů zároveň probíhá na čtyřech různých prostředích. V součtu tak může být kompletní ověření softwaru automatickými testy několikadenní záležitost. Ověření vlastních změn a nových částí kódu může trvat také až několik desítek hodin, a to v případě vytvoření fronty, kdy je potřeba ověřit několik různých vývojových větví najednou. Pravidelné ověřování hlavní vývojové větve je tedy zdlouhavé a někdy až nepředstavitelně náročné. Zároveň systémových testů neustále přibývá, a to vlivem rozvoje funkcionality produktu Inspire Scaler. Jedná se o přírůstek několika systémových testů během jednoho sprintu.

3) **Nestabilita systémových testů** – snad největším problémem testovacího projektu jsou nestabilní systémové testy. Pod tímto termínem se rozumí test, který při opakovaném spuštění na CI serveru náhodně neprochází, aniž by byl změněn zdrojový kód softwaru. Takový test nemá vypovídající hodnotu, neboť jeho výsledek je nevypovídající, a co více, ubírá členům vývojového týmu čas, který musí věnovat detekováním pádu, hledáním příčiny a opětovným spuštěním. Množství nestabilních testů tak zakrývá skutečné problémy, které odkazují na skutečné chyby ve zdrojovém kódu.

4) **Konfigurace CI serverů –** pokud je potřeba změnit konfiguraci CI serveru nebo jeho prostředí, je nezbytné, aby byly upraveny konfigurace na všech testovacích strojích samostatně, což je při udržování pěti CI serverů náročné. Zároveň je potřeba všechny stroje neustále aktualizovat a udržovat, zajištovat připojení na externí systémy a služby, které jsou nezbytné pro běh pro běh automatických systémových testů.

4.2 Řešení problémů s aplikací continuous integration

Kapitola blíže specifikuje jednotlivé problémy testovacího projektu Inspire Scaler. Dále zachycuje jednotlivá řešení a kroky, které bylo nutné provést pro zefektivnění aplikace continuous integration a pro odstranění těchto problémů a kompletní vylepšení automatického testování softwaru Inspire Scaler.

4.2.1 Automatizace manuálních činností

Automatizace manuálních testů je v celé společnosti Quadient velkým tématem. Obecně je kladen důraz na zautomatizování takovýchto činností a na hledání cest, které zrychlí či zefektivní celý vývojový cyklus. V tomto procesu je vyvíjeno úsilí o tvorbu nejen automatických testů, které ověřují správné chování softwaru, ale i automatizování dalších podpůrných procesů a činností. Pro efektivní aplikování continuous integration je automatizace nezbytná a nevyhnutelná, v jiných případech není dosahováno pozitivních přínosů. Jedná se především o:

- automatizování manuálních testů,
- automatizování a zajištění přehledné a rychlé zpětné vazby,
- automatické vyhodnocování či porovnávání výsledků automatických testů.

4.2.1.1 Automatizace manuálních testů

Jak již bylo řečeno, testovací projekt obsahuje přes dvě stě sedmdesát manuálních testů. Každý manuální test musí být během regresního testování proveden, aby byla ověřena veškerá funkcionalita softwaru. To vyžaduje nesmírně mnoho času. Rychlost provedení manuálního testu je vždy podmíněna znalostí dané oblasti softwaru. Z hlediska šetření času je vhodné, aby test vykonávala osoba, která danou oblast dobře zná. Na druhou stranu je vhodné, aby test byl vykonáván někým, kdo netrpí tzv. testovací slepotou neboli neschopností odhalit chyby v softwaru, které vznikly opakujícím se prováděním stejných testů. Kvalita i rychlost testování je tedy závislá na konkrétním lidském faktoru, což může přinášet velké riziko. Manuální testy během regresního testování zpravidla odhalují množství chyb, které musí být ještě před vydáním nové verze opraveno.

Skutečnost, že automatické testy jsou správnou cestou, jak se tomuto riziku vyhnout a jak hlídat kvalitu softwaru již během období vývoje, je všeobecně známo. V oblasti automatizace manuálních testů bylo již v testovacím projektu mnoho podniknuto – konkrétně osm set automatických testů. I přes poměrně velkou snahu existuje velké množství manuálních testů, které se díky rozvoji funkcionality softwaru neustále zvyšuje. Mezi hlavní důvody existence a zvyšujícího se počtu manuálních testů patří:

- systémový test je obtížné ve stanovením čase implementovat,
- doba trvání a náklady na údržbu provozování testu převyšuje hodnotu, kterou test přináší,
- existují technické limitace, které brání napsání automatických testů,
- existuje velký dluh manuálních testů z minulosti.

Proto je kladen veliký důraz na automatizování všech manuálních testů, které nejsou limitovány technologickou či provozní specifikací. Ke každé nové user story29 (US), které jsou implementovány během jednotlivých sprintů, vzniká téměř ve všech případech nový test. Test, který vzniká v podobě předpisu (test case), je nejprve manuální. Test case je tedy spíše písemný zápis o jednotlivých krocích testu, které simulují scénář uživatelského užití. Pro automatizaci takovýchto scénářů je striktně stanoven čas již během vývoje user story, tak aby nevznikal žádný další dluh. To zároveň zvyšuje i kvalitu automatického testu, který nyní není implementován ve spěchu. Pokud nebude test ihned automatizován, ale odkládán na později, bude se opět zvyšovat počet manuálních testů, což může vyústit v neočekávané problémy před vydáváním nové verze – například nebude potřebná kapacita na řádné otestování. Pro automatizaci manuálních scénářů byl zpravidla volen druh systémového testu.

Kromě průběžné automatizace byla vyvinuta další silná iniciativa na automatizování veškerých, dříve vzniklých, manuálních testů, které nejsou

---

[29] User story – označení pro zadání a definování jedné či více vlastností softwaru

technologicky blokovány. Testy byly analyzovány, označeny a postupně zautomatizovány. Během takového počínání byly některé manuální testy zrušeny. Jedná se především o testy, které nepřinášejí hodnotu, nebo které jsou již obsaženy v jiných scénářích jako součást predispozic. Další manuální testy byly například sloučeny s jiným testem tak, aby jejich jednotlivé kroky mohly být vykonávány souběžně.

Shrnutí: Manuální testy je nutné průběžně během období vývoje aktualizovat, slučovat a především automatizovat, a to souběžně s dokončováním implementace US. V jiném případě bude jejich počet narůstat a znovu prodlužovat regresní testování před vydáním nové verze softwaru.

4.2.1.2 Čitelnost systémových testů

Automatické testy přinášejí rychlou zpětnou vazbu a informaci o aktuálním stavu softwaru. Správný automatický test je takový, který odhalí problém. V některých případech však nemusí být na první pohled z automatického testu zřejmé, proč je neúspěšný, a zdali se skutečně jedná o chybu v softwaru. To je způsobeno špatnou čitelností testu, který zároveň není snadno pochopitelný.

V dalších případech je potřeba test opravit nebo aktualizovat oproti cílené změně chování softwaru. Aby bylo test efektivní aktualizovat, opravit, případně ho rozšířit, je potřeba zajistit odpovídají strukturu testu, která bude čitelná a jednoduchá. Pokud jsou taktéž testové metody nečitelné až kryptické, je pro další členy vývojového týmu opět velmi složité a časově náročné pochopit, co se v testu odehrává a k čemu vůbec test slouží. Přihlédneme-li k přirozené fluktuaci zaměstnanců, je více než pravděpodobné, že osoba, která původní test implementovala, již nebude k zastižení. Dlouhá investigace kódu testu je neefektivní a oprava často není snadnou záležitostí. Člen vývojového týmu má v této situaci dvě možnosti, a to analyzovat test nebo vyhledat jeho testový scénář v externím uložišti TestTrail. V drtivé většině případů je uplatňována první z možností.

Z těchto důvodů je nesmírně důležité, aby byl test strukturovaný, čitelný a pochopitelný, a to i pro osobu, která se s testem ještě nesetkala. Aby bylo dosaženo

odpovídající čitelnosti, byla v testech zavedena struktura given-when-then. Tato struktura se skládá ze tří částí. Část given popisuje, co je potřeba pro test připravit, co je potřeba nastavit nebo do jakého stavu je potřeba se dostat. Jedná se tedy o testový kontext či pre-rekvizitu. Část when představuje jádro celého testu, která říká, co test ověřuje – jakou operaci nebo funkci. A poslední část then sloužíke kontrole dosažených (získaných) výsledků částí when. Získané výsledky jsou zde porovnávány s očekávanými hodnotami. Následující ukázka je implementována pomocí testovacího frameworku Spock.

```
@Test
void "add and remove list item test"() {
    given:
    emptyListIsPrepared()

    when:
    addNewItemToList()

    then:
    assertListHasOneItem()

    when:
    removeItemFromList()

    then:
    assertListIsEmpty()
}
```

Ukázka 4: Struktura systémových testů
Zdroj: vlastní zpracování

Vedle struktury bylo též potřeba zapracovat na čitelnosti všech testových metod a to tak, aby jejich název jasně deklaroval úkon nebo operaci, kterou provádí. Ideou této snahy je dosáhnout maximální čitelnosti testu. Všechny dílčí kroky, které je potřeba provést, jsou zaobaleny do hlavních metod. To zvyšuje čitelnost a pochopitelnost testů a dovoluje automatický test číst jako původní testový scénář. Důraz byl kladen i na další psaní testových metod podle obecných pravidel programování. Dále je potřeba přehledně vypisovat výjimky, které jsou v případě problému obdrženy například z asercí a skutečné hodnoty, aby byla ihned zřejmá příčina pádu daného testu. Do testů je také důležité na kritická místa vhodným způsobem doplnit logování tak, aby byly informace o neúspěšném průběhu testu co nejpřesnější.

Shrnutí: Neúspěšné testy jsou často opravovány různými členy vývojového týmů. Každý test proto musí být čitelný, strukturovaný a jeho podstata smyslná tak, aby jeho případná oprava či aktualizace byla rychlá a snadná Současně je také potřeba získávat maximální množství informací o důvodu selhání testu.

4.2.1.3 Existence testovacího předpisu

Pokud je manuální test automatizován a zároveň je čitelný, přestává písemný předpis pozbývat smysl. Test case nepřináší v tomto momentě již žádné další důležité informace. Ty jsou duplicitně obsaženy v automatickém testu. Naopak vyžadují údržbu a aktualizování ve chvílích, kdy je funkcionalita softwaru pozměněna. Změny se musí zohlednit nejen do samotného automatického testu, ale i do předpisu, aby nedocházelo k rozdvojení informací a zbytečným zmatkům. Pro správu a udržování takovýchto testových předpisů je zpravidla zapotřebí další externí systém. Testovací scénář začne plnit administrativní funkci.

Proto bylo navrhnuto takovéto scénáře zrušit, neboť nepřinášejí žádnou hodnotu. Ovšem tato změna prozatím nebyla aplikována, a to vzhledem k přetrvávajícímu problému s čitelností některých testů a kvůli potřebám regresního testování.

Shrnutí: Automatické testy jsou duplicitní kopií testového scénáře. Udržování stejných informací na různých místech je zbytečné a často nákladné. Pokud je automatický test dostatečně čitelný a strukturovaný, zcela dostačuje a není třeba uchovávat a spravovat i testové scénáře.

4.2.1.4 Automatizované podpůrné nástroje

Výsledky automatických testů hlavní vývojové větve je potřeba pravidelně sledovat a vyhodnocovat, a to alespoň jedenkrát denně. To přináší značné množství opakujících se manuálních činností. Jedná se především o manuální procházení CI serverů a hledání správných ověření nebo kolektivizace samotných výsledků testů. Na CI serveru se totiž může vyskytovat velké množství privátních ověření, jejichž výsledky nejsou relevantní. Sledování všech CI serverů a správných ověření se stává neoblíbenou činností.

Pro podporu sledování a vyhodnocování testů bylo připraveno několik podpůrných programů. V prvé řadě se jedná o automatický CI report, který je generován každý den. Report sbírá správné výsledky ze všech CI serverů a výsledky přehledně rozesílá do komunikačního kanálu MS Teams. Vedle toho existují i další podpůrné nástroje, například komparátor výsledku, který porovnává jednotlivá ověření a jejich neúspěšné testy. Dále se může jednat o nástroje, které kontrolují, zda byly během ověření spuštěny všechny testy či nikoliv.

Jenkins report
Sc12.5-LocalSuite
nucw <107>: **0** / 108 failed. Duration 0:17:05 [dnes 02:14]
nucw2 <108>: **0** / 108 failed. Duration 0:16:50 [Před **1 dny** 00:24]
nucw3 <111>: **1** / 108 failed. Duration 0:17:23 [dnes 06:31]
Sc12.5-full
nucw <207>: **6** / 941 failed. Duration 3:15:22 [dnes 03:03]
nucw2 <196>: **3** / 941 failed. Duration 3:15:04 [Před **1 dny** 20:20]
nucw3 <155>: **7** / 941 failed. Duration 3:26:35 [dnes 02:17]
daisy <136>: **5** / 725 failed. Duration 2:11:35 [včera 20:20]
Sc12.5-FailingSuite
nucw <116>: **6** / **62** failed. Duration 0:25:51 [dnes 01:36]
nucw2 <128>: **3** / **62** failed. Duration 0:26:14 [dnes 02:45]
nucw3 <116>: **9** / **62** failed. Duration 0:29:38 [dnes 00:56]
daisy <103>: **32** / **61** failed. Duration 0:20:54 [včera 18:51]

Obrázek 13: CI report v MS Teams
Zdroj: vlastní zpracování

Shrnutí: Při každodenním analyzování výsledků systémových testů vzniká množství manuálních opakujících se činností. Je možné využít různé nástroje nebo jejich části, a tak opakující se činnosti do jisté míry zautomatizovat a zmírnit tak vynakládané úsilí a plýtvání časem.

4.2.2 Zavádění nových procesů

Spolu s automatizací manuálních testů je potřeba zavést další procesy, které budou tuto činnost efektivně podporovat a ukotvovat jako jednu ze standardních činností. Pokud nebude testování systémovými testy řízeno, dostane se celý projekt do velkých problémů. Může totiž nastat situace, kdy systémové testy přestanou přinášet zjednodušení a usnadnění práce,

ale naopak velké trápení a dodatečné úsilí. To může například vyústit v neschopnost ověřit si změny na CI serveru či se spolehnout na výsledky systémových testů.

S přibývající funkcionalitou softwaru narůstá počet systémových testů, které se spouští na CI serveru. Doba jednotlivých ověření se tedy zákonitě prodlužuje s každým přidaným systémovým testem. Při neřízeném implementování systémových testů, dochází ke zbytečné tvorbě systémových testů na funkcionalitu softwaru, která by mohla být pokryta integračními nebo jednotkovými testy. Zároveň v agilním vývoji je software neustále upravován a každá změna musí být promítnuta i do všech automatických testů tak, aby byly testy stále aktuální, a pokud nebudou, testy během ověření popadají. Problémem systémových testů, kromě doby běhu, je také skutečnost, že vývojář není schopný spustit všechny systémové testy na svém stroji (oproti jednotkovým) najednou. Případný pád na CI serveru se projeví až s několikahodinovým časovým odstupem. To může být pro vývojáře svazující a často i otravné. V případě, že testy popadají, je potřeba pád testu prověřit, případně test opravit. Zároveň je potřeba myslet na skutečnost, že na vývoji produktu se podílí přes čtyřicet lidí, a není tedy možné, aby byly testy dlouhodobě neopravené či by jeden test opravovalo nevědomky několik osob najednou.

Další vznikajícím problémem je samotné množství systémových testů, které neustálen narůstá. Aby byla automatizace systémových testů přínosná a udržitelná, je potřeba zavést nové procesy a činnosti, které vyřeší především následující body:

- zajistit vyhodnocování výsledků systémových testů,
- zajistit pravidelné řízené opravování systémových testů,
- korigovat počet systémových testů, který neustále stoupá.

4.2.2.1 Testovací analýza

Jak již bylo v předchozích kapitolách řečeno, systémové testy dlouho trvají a jejich provozování a údržba je nákladná. Naopak nejvěrohodněji napodobují činnosti manuálního testování, které odráží chování zákazníka. Ve shodě s testovací

pyramidou je však vhodné mít několik systémových testů, které testují základní a klíčové oblasti softwaru. To je v rozporu se současným přístupem, kde je systémový test standardem každé US. Aby byl mírněn přírůstek systémových testů je potřeba zavést postupy, ve kterých bude stanoveno, co je nutné testovat systémovými testy a co je naopak vhodné snížit na úroveň integračních testů.

Proto byl zaveden zcela nový proces. Před započetím implementování US je provedena testovací analýza. Jejím cílem je rozvrhnout jakými testy a na jakých úrovních testovací pyramidy bude funkcionalita US pokryta. Analýzy se zpravidla účastní vývojáři, kteří budou US implementovat a dále zodpovědný QA pracovník. Společně se snaží najít shodu, jak by bylo možné automaticky testovat danou funkcionalitu jednotlivými nebo integračními testy a kde je nezbytné použít systémový test. Dále je vhodné se zamyslet nad situací, kdy by daná funkce softwaru přestala fungovat, tak jak má. A dále nad případnými následky a dopady pro zákazníka, které nastanou, pokud nebude chyba odhalena. Jedná se tedy o rychlou analýzu rizik. Dále se snaží najít schůdné cesty, jakým testováním na nižší úrovni dosáhnout – tzn. jak otestovat funkcionalitu integračními testy, neboť při těchto analýzách bylo odhaleno, že projekt je na integrační testování špatně připraven.

Dalším přínosem testovacích analýz je uvědomění si, jakou podobu má mít systémový test. Happy case neboli šťastný případ je typ testu, který ověřuje ideální fungování softwaru. Jedná se tedy o základní použití funkce softwaru za standardních podmínek a předpokladů. Výjimkou mohou být zásadní výpadkové scénáře a testování záchranných mechanismů. Systémový test by však neměl být používán pro ověřování méně podstatných částí či okrajových, speciálních nebo podmíněných situací. Pro takovéto účely je vhodné použít testy integrační. A zároveň během implementace US je kladen velký důraz na řádné manuální testování právě těchto hraničních případů.

Shrnutí: Narůstající množství systémových testů je dlouhodobě neudržitelné. Aby se zabránilo zbytečným implementacím systémových testů, byla zavedena testovací analýza, která probíhá před započetím prací na US a jejím cílem je hledat optimální cesty, jakými lze US testovat.

4.2.2.2 Test master

Při každodenním spouštění systémových testů nad hlavní vývojovou větví vzniká potřeba neustále vyhodnocovat získané výsledky. Výsledky CI serveru jsou zpravidla dostupné pro všechny členy vývojových týmů. Avšak při řešení nestandardních případů na CI serveru či při analýze spadlých testů vzniká potřeba úkoly organizovat a rovnoměrně rozdělovat mezi všechny členy týmu pracujících na projektu. Pokud by tomu tak nebylo, docházelo by často k duplicitním řešením stejného problému, což je značně neefektivní.

Z těchto důvodů vznikla role test master. Test master neboli správce testů je přenosná role, která se pravidelně (po týdnech) předává z týmu na tým, tak aby byla zátěž rovnoměrně rozložena. Téměř vždy se jedná o QA pracovníka. Jedná se o osobu, která je primárně zodpovědná za celkový výsledek testů. Jeho hlavními činnostmi je analyzování a hlídání výsledků na CI serverech, procházení spadlých automatických testů a řešení hromadných problémů jak se systémovými testy, tak se samotnými CI servery. Jeho úkolem je také zajistit pracovníky na řešení vzniklých problémů, a tyto úkoly rovnoměrně rozdělovat mezi všechny týmy tak, aby týmy rovnoměrně zatěžovaly. Projekt se svého času totiž potýkal s problémem, kdy se opravě testů dobrovolně věnovala pouze část týmů, což pochopitelně mělo dopad na dodávku implementovaných US.

Tato role tedy přináší dodatečné činnosti, které musí být vykonány v souladu s ostatními QA povinnostmi, které se na pracovníka vztahují. Mezi hlavní činnosti test mastera především patří:

- analyzovat výsledky z CI serverů,
- zajištovat opravy spadlých systémových testů,
- zajišťovat údržbu CI serverů a dalších systémů,
- rovnoměrně využívat kapacit všech týmů pro jednotlivé úkoly,
- znát a zodpovídat za aktuální stav testů a CI serverů.

Shrnutí: Je nezbytné organizovat analyzování a opravování neúspěšných systémových testů tak, aby nedocházelo k duplicitním zásahům a aby se zároveň

testy průběžně opravovaly. Proto byla zavedena role test master, která pravidelně monitoruje stav testů, organizuje jejich opravy a zodpovídá za celkový stav.

4.2.2.3 Test worker

Udržovat všechny testy v pořádku, včetně CI serverů, je velice náročné a v případě častých zásahů do zdrojového kódu ostatními členy vývojových týmů téměř nedosažitelná věc. Test master by sám velké množství testů nedokázal obstarat při svých běžných povinnostech spjatých s jeho pracovní pozicí. Test master je primárně zodpovědný za aktuální stav systémových testů a jak již bylo řečeno, jeho privilegiem je zadávat úkoly dalším členům týmů. Při aplikaci test master pravidla se však v praxi často naráželo na to, že zadané úkoly byly pověřenými lidmi odkládány na později (z důvodu plnění řady jiných povinností), a tedy nebyly ihned řešeny. To se opět promítá do celkového stavu systémových testů.

Proto byla dále zavedena role test worker. Jedná se o osobu, kterou si zřizuje každý tým podle svých možností a preferencí. Tuto osobu si stanovují týmy na libovolné období, nejčastěji na jeden týden. Smyslem test worker role je zajistit v každém týmu kapacitu, která se bude prioritně věnovat potřebám test mastera. Jedná se především o tyto činnosti:

- přijímat úkoly stanovené test masterem,
- obdržené úkoly řešit prioritně a neprodleně,
- aktivně napomáhat test masterovi,
- analyzovat a opravovat systémové testy.

Shrnutí: Analýza a opravování neúspěšných systémových testů jsou často odkládány na pozdější chvíli. Proto vznikla role test worker, která zajištuje v každém vývojovém týmu osobu, která se těmto aktivitám bude prioritně věnovat, tak aby se systémové testy stíhaly průběžně opravovat a nevznikal dluh v podobě rozbitých testů.

4.2.3 Stabilita systémových testů

Vedle problému s přibývajícími systémovými testy, a tedy s prodlužováním doby jejich celkového průběhu, se projekt potýká také s jejich nestabilitou. Nestabilní systémové testy jsou skutečným problémem, který přináší mnoho starostí. Nejenže takové testy nevykazují hodnotu, ale naopak znesnadňují práci všem členům vývojářských týmů. Vývojáři tak nemají jistotu, zdali svými změnami ve zdrojovém kódu skutečně nezapříčinili vznik nových problémů. Dále je potřeba neúspěšné testy analyzovat, zdali se skutečně nejedná o chybu softwaru.

Ve velké většině případů jsou tyto testy nestabilní pouze v celém balíku spuštěných testů na CI serveru, a to z důvodů snížení výkonosti. Při lokálním spuštění takovýchto testů nedochází k žádným selháním. Vývojář tedy při tvorbě systémového testu nebo při jeho opravě nemá stejné prostředí jako na CI serveru. Test je zpravidla na lokálním stroji často spouštěn samostatně. Po analýze příčin nestabilních testů byly objeveny tyto souvislosti:

- test je ovlivněn jiným testem, který jej předchází,
- test je závislý na externích službách s očekávaným nastavením,
- test je závislý na prostředí, ve kterém běží,
- CI server je nedostatečně výkonný stroj,
- testové metody se předbíhají, jsou nesynchronní,
- test je špatně napsán, GUI testy jsou velice náchylné na změny.

Každodenní analyzování takovýchto testů je tedy téměř zbytečné, přesto se musí provádět. Aby testy byly stabilní, musí být z velkých částí upraven testovací aparát tak, aby nedocházelo k výše uvedeným příčinám. Jedině pak dává každodenní vynaložené úsilí do spouštění, analyzování a opravování systémových testů smysl.

4.2.3.1 Automatické uklízení

CI server spouští systémové testy náhodně jeden po druhém na nainstalovaném a spuštěném softwaru Inspire Scaler. Aby testy mohly efektivně ověřovat testovací scénáře, téměř vždy potřebují určité pre-rekvizity. Ty je tedy nutné před samotným zahájením automatického testu připravit či nahrát na určitá

místa. Může se jednat o různá nastavení softwaru, včetně přístupových práv uživatelů nebo předpřipravených souborů a dat. V průběhu testu jsou tyto pre-rekvizity používány, modifikovány a mazány. V dalších případech jsou takovéto soubory a data generovány jako výsledek testů. Po dokončení testu však nemusí vždy docházet ke smazání všech pre-rekvizit, které byly použity, a navrácení výchozího nastavení. Poté další z následujících testů může například vyžadovat jiné nastavení softwaru, které je přímo v rozporu s předchozím nastavením. Dochází tedy k řadě anomálií a odlišností, které vyústí v selhání systémového testu či celé řady testů.

Do systémových testů proto byly implementovány pomocné třídy s metodami, které mají za úkol po dokončení testu automatiky vracet nastavení softwaru do původního stavu a smažou již nepotřebné soubory či jiné pre-rekvizity. Zkrátka každý test se po sobě snaží uklidit tak, aby následující testy mohly proběhnout se základním nastavením. Pro tyto účely byla využita testovací knihovna JUnit 5, která je uzpůsobena pro implementování různých druhů testů. Nejvíce prospěšné z tohoto frameworku jsou především anotace @BeforeEach a @AfterEach. Metody takto označené se vykonávají před testem či bezprostředně po jeho proběhnutí bez závislosti na výsledku testu. Takové počínání přináší jistou záruku, to znamená, že pokud test v průběhu ověřování selže a ukončí se, metody takto označené se provedou a navrátí nastavení do původního stavu. Pro zaručení správného použití metod vznikla sada testovacích tříd (Trait), které automaticky spouští tyto přípravné a uklízecí metody.

```
@BeforeEach
void prepareEnvironmental(){
    setScalerSetting()
    ...
}

@AfterEach
void cleanEnvironmental(){
    restoreInitialScalerSetting()
    ...
}
```

Ukázka 5: Anotací knihovny JUnit5
Zdroj: Junit5, 2019; upraveno

Velký problém byl také odhalen při otevírání nových karet ve webovém prohlížeči Google Chrome. Jak již bylo nastíněno, část systémových testů je zaměřena na ověřování funkčnosti a koherence GUI. Takovéto testy v rámci ověření potřebují webový prohlížeč na procházení jednotlivých oblastí softwaru, ve kterém simulují chování uživatele. Pro běh tzv. GUI testů je zapotřebí jediná instance Chromu, avšak pokud testy po sobě nezavírají otevřené chromové karty, vzniká problém s výkonem stroje, na kterém jsou testy spuštěny, a prohlížeč začíná vykazovat známky abnormálního chování. Je tedy nezbytné, aby si každý test po svém průběhu dokázal zavřít (uklidit) i otevřené chromové karty a okna.

Shrnutí: Po sobě jdoucí testy se často ovlivňují, následkem čehož je náhodná nestabilita testů. Je tedy nutné, aby každý test měl mechanismus, který po jeho dokončení vrátí testovaný software do původního stavu. Pro tyto účely byly v systémových testech použity anotace @BeforeEach a @BeforeAll.

4.2.3.2 Aktivní čekání

Vedle testů, které po sobě neuklízí, byl spatřen také velký problém s předcházením událostí, konkrétněji s předcházením testových metod. Obecně nastává situace, kdy je vykonána nějaká testovací akce, po které je očekávána adekvátní reakce. Reakce však nemusí však vždy přijít v řádech milisekund, jak by očekával systémový test, ale zpravidla přichází se zpožděním, například až po dokončení celé řady operací testovaného softwaru. Test by aktuální stav vyhodnotil jako neadekvátní (žádnou) reakci a neprošel by. Proto byly na samém počátku automatizace do takovýchto testů implementovány tzv. thread sleep pomocí metody Thread.sleep(). Jedná se o metodu, která pozastaví vlákno na stanovenou dobu. Tyto metody svůj účel do jisté míry splnily. Na druhou stranu je velice obtížné stanovit čas, po který má být test uspaný. Pokud je doba příliš dlouhá, test zbytečně čeká, pokud je naopak kratší, než je potřeba na získání reakce, test bude opět neúspěšný. Stanovený statický čas je tak spíše pokus-omyl, který na pomalejším stroji často nevychází. A téměř ve všech případech se jedná o zásadní chybu v testu, která způsobí zpomalení a nestabilitu testu.

Jelikož takto čekající testy byly taktéž nestabilní a zároveň prodlužovaly celkovou dobu trvání systémových testů, byl thread sleep nahrazen metodami, které aktivně čekají na adekvátní reakci. Principiálně se jedná o metody, které se ve stanovém čase a periodě aktivně kontrolují, zda byla splněna podmínka (adekvátní reakce). Pokud není podmínka splněna v maximálním stanovém čase, test selže, a to však ve většině případů oprávněně. Tento přístup je prospěšný především na CI serveru, kde dochází k zátěži a zpomalení jak systémových testů, tak samotného softwaru Inspire Scaler. Nejenže testy s těmito metodami fungují lépe, ale v součtu díky odstranění statického čekání zrychlují celkový běh všech systémových testů. Následující metoda přijímá parametr sleepInterval, který slouží pro periodické ověřování podmínky, timeout je maximální doba, během které musí být podmínka splněna.

```
TestUtil.waitUntilConditionMatches(
    new Condition(currentStatus, expectedStatus), sleepInterval, timeout
)
```

Ukázka 6: Metoda s aktivním čekáním
Zdroj: vlastní zpracování

Shrnutí: Testy jsou nestabilní z důvodů špatného časování a synchronizace, která se projevuje nejen při větší zátěži (software i testy pracují pomaleji). Na kritických místech je vhodné použít metody s aktivním čekáním tak, aby byly prostoje testu co nejmenší a zároveň byl test stabilní i při nízkém výkonu CI serveru.

4.2.3.3 Zajištění externích systémů a služeb

Vzhledem k funkcionalitě Inspire Scaler založené na integraci a procesování dat pro ostatní podnikové programy jsou systémové testy často velice závislé na periferiích neboli na externích systémech a službách. S těmito periferiemi je také často nutné software ověřovat. Testování se tak stává složitější obzvláště ve chvílích, kdy je potřeba mít funkční, nastavený a běžící nástroj třetí strany. Testy tedy očekávají, že je daný externí systém nebo služba spuštěna a nastavena pro jejich použití. Přestože je téměř vždy pro toto testování vyžadováno základní nastavení,

stává se, že systém nebo služba neběží či je nedostupná z lokace, na které jsou testy spuštěny. V případě takovýchto výpadků jsou testy předurčeny k neúspěchu.

Řešením takové situace je spuštění externích systémů nebo služeb v rámci systémového testu, přesněji těsně před provedením samotného testu, a po dokončení opět jejich vypnutí. Pro tyto účely opět posloužily anotace @BeforeEach a @AfterEach, ve kterých jsou tyto periferie připravovány a ukončovány. Avšak toto řešení má i svá úskalí. Všechny potřebné zdroje ke spuštění takovýchto systémů či služeb musí být na místech, ze kterých jsou systémové testy schopné tyto zdroje získat a spustit. Nejčastěji se tak umístí přímo do zdrojového kódu projektu, což razantním způsobem zvedá jeho celkovou velikost. Další nevýhodou je, že tímto způsobem lze spustit jen některé menší služby nebo systémy. Zároveň je potřeba připravit i mechanismy ověření toho, že je služba skutečně spuštěna, pakliže není, je potřeba test ukončit a zobrazit informace o problému.

Shrnutí: Systémové testy, které ověřují integraci s externími systémy a službami nemohou spoléhat, že systém či služba jsou dostupné s jejich vyžadovaným nastavením. Je tedy důležité, pakliže je to možné, aby si testy dokázaly připravit tyto externí systémy a služby s nastavením, které potřebují, a po dokončení testu je opět dokázaly vypnout.

4.2.3.4 Mockování služeb

Výše popsaný bod pojednává o spouštění externích systémů a služeb v rámci systémových testů. Avšak ne všechny aplikace lze jednoduše spustit a nakonfigurovat během spuštěných testů a poté je opět vypnout. Případně jsou tyto operace natolik časově nákladné, že to převyšuje přínos a usnadnění práce, kterou by nově testy přinesly.

Aby ale byly vyřešeny i takovéto situace a aby byl počet nestabilních testů redukován, bylo v systémových testech použito mockování. Mockování se zcela běžně používá v jednotkových či integračních testech. Jedná se o nahrazení původního objektu (služby) za jeho testovací imitaci, která neprovádí žádné

operace, které nebyly předem stanoveny. Pro systémové testy byla využita knihovna WireMock, která umožnuje mockovat http API a vytváří tak falešný server služby. Systémové testy tedy spustí WireMock server s připraveným API. Dále na každý http požadavek (request) je potřeba předem stanovit odpověď (response), která má být ze serveru odeslána. V testech je tedy simulováno chování externí služby pro reálné chování testovaného softwaru.

```
WireMockServer wireMockServer = new WireMockServer(2000,2001)

void mockGetMessage() {
    wireMockServer.stubFor(get(urlEqualTo("/rest/api/messages"))
            .withRequestBody()
            .willReturn(aResponse()
                    .withStatus(200)
                    .withBody("Hello World")))
}
```

Ukázka 7: WireMock server
Zdroj: WireMock, 2019; upraveno

Mockování bylo v systémových testech použito pouze v těch případech, které nespadají do základních (core) testů pro integraci s externím systémem či službou. Tedy určitý počet testů, které spoléhají na běžící služby, existují i nadále, ale ve většině případů se jedná o jednotky testů pro daný software.

Shrnutí: Některé externí systémy či služby nejdou efektivně spouštět během systémových testů. Pro eliminaci takto nestabilních testů je vhodné použít mockování. Konkrétně byla pro systémové testy použita knihovna WireMock server, který umožní ověřit funkcionalitu bez nutnosti běžícího externího systému nebo služby.

4.2.4 Doba trvání systémových testů

Ruku v ruce s nestabilitou systémových testů byl řešen problém délky celkového trvání jednoho průběhu CI serveru nad zdrojovými kódy hlavní vývojové větve, ve kterém jsou všechny systémové testy spuštěny. Jeden průběh velkého balíku systémových testů trvá až sedm hodin. Počet těchto testů se bude díky automatizaci manuálních testů a přibývání nové funkcionality softwaru neustále zvyšovat, tudíž i jejich celková doba trvání bude neustále stoupat. Takto dlouhá doba ověření

změn je pro vývojáře značně nepříjemná. Zpětnou vazbu obdrží nejdříve za několik hodin, a to pouze v případě, pokud bude jeho zdrojový kód vedlejší vývojové větve první v pořadí na CI serveru. Vývojář tedy není schopen průběžně ověřovat své změny, proto tak činí pouze zřídka, což často ústí v problémy. Doba trvání jednoho ověření je ovlivněna:

- narůstajícím počtem systémových testů,
- nevhodně použitými testovacími metodami v testech,
- robustním testovacím aparátem,
- nedostatečným výkonem CI serverů,
- tvořením front na CI serverech.

Problém nevhodně použitých metod v testech byl do značné míry odstraněn již při aplikaci aktivního čekání, které přineslo v celku velké zrychlení systémových testů. Problém s nedostatečným výkonem stroje byl řešen rozšířením operační paměti, což přestavuje jen aktuální záplatu.

4.2.4.1 Základní testovací sada

Ověřování změn pomocí systémových testů na CI serveru není snadné. Vývojář nemá často ani šanci si jinak ověřit změny než zkrátka počkat, než přijde na řadu jeho ověření. Systémové testy nejsou spustitelné lokálně, tedy vývojář není schopen si ověřit své změny ani na svém vlastním stroji. Jejich spuštění je velice závislé na připravenosti externích systémů a služeb, nastavení a výkonu stroje. Systémové testy se navíc spouštějí jako velký jednotný balík. Tento balík obsahuje nejrůznější testy z nejrůznějších oblastí. Spouštění všech testů během vývoje není pro vývojáře natolik přínosné. Ti často potřebují rychlou zpětnou vazbu z oblasti, ve které se pohybují.

Proto byl vymyšlen koncept lokální testovací sady, která je nezávislá na místě, ve kterém se spouští. Jedná se o sadu testů, jakýsi reprezentativní vzorek, který je spustitelný na lokálním stroji vývojáře a nepotřebuje pre-rekvizity. Během několika minut má vývojář základní přehled o aktuálním stavu změn ve zdrojovém kódu, a to ještě před samotným commitem změn do sdíleného

uložiště. Sada nepokrývá veškerou funkcionalitu či speciální případy, ale jedná se především o průřez všemi oblastmi a slouží k rychlému získaní zpětné vazby. Zároveň jsou tyto testy označeny jako velice stabilní, tedy jakýkoliv neúspěšný test v této sadě vývojáři signalizuje jistý problém. Takzvaná lokální sada je též nakonfigurována na CI serverech, kde se stala symbolem stability a kvality. Pokud některý z těchto testů byl neúspěšný, je povinností test mastera, aby jej přednostně řešil, neboť se s velkou pravděpodobnosti jedná o závažný problém, který je potřeba okamžitě řešit.

Testy z lokální sady jsou spustitelné na základě speciálního nastavení, které vyžaduje přesné označení. Pro označení testů byla použita anotace @Tag z již dříve zmíněného frameworku JUnit5. Tyto anotace se dají použít jak nad testovou třídou, tak nad samotným testem, a lze jich použít neomezené množství.

```
import org.junit.jupiter.api.Tag
import org.junit.jupiter.api.Test

@Tag("Administration")
class UserRightsTest {

    @Test
    @Tag("LocalSuite")
    void "Set administrator rights to user"() {
        ...
    }
}
```

Ukázka 8: Označení systémových testů
Zdroj: vlastní zpravování

Shrnutí: Ověření kompletní sady systémových testů trvá dlouho. Aby byla zajištěna rychlá zpětná vazba pro vývojáře bez závislosti na stavu CI serveru, byla nakonfigurována základní sada systémových testů, které si vývojář může spustit na svém lokálním stroji (osobním PC).

4.2.4.2 Kategorizace testů

Konfigurace základní lokální sady otevřela možnosti konfigurace daleko menších a specializovanějších sad. Lokální sada je provedena rychle a zároveň jde spustit kdekoliv bez výrazných příprav. Avšak vzhledem k tomu, že se jedná

o průřez testů ze všech oblastí, nedokáže odhalit všechny problémy. Vývojář, který se pohybuje během úprav pouze v jedné oblasti, potřebuje i rychlou zpětnou vazbu zaměřenou pouze na svoji oblast.

Z těchto důvodů byly všechny testy označeny tagy, stejně tak jako v případě lokální sady. Tag označuje oblast softwaru, kterou testy ověřují. Ty již byly použity v externím systému na správu testových scénářů. Některé testy spadají do více oblastí, jedná se tak o komplexní testy – ty jsou zpravidla náchylnější k chybám. Díky takovému označení lze spouštět různé testové sady, které poskytnou vývojáři rychlou zpětnou vazbu o konkrétní oblasti. Avšak jedná se pouze o orientační průběžné výsledky. Vývojář by měl před dokončením US spustit všechny systémové testy, avšak jejich průběh by měl být již poklidný a neměl by přinášet neočekávané problémy.

Shrnutí: Díky označení všech testů tagy, které reprezentují danou oblast softwaru, lze spustit rychlou sadu systémových testů. Ta má pro vývojáře největší přínos, neboť testy se vztahují na funkcionalitu, které se nové změny týkají. Dále je možné tvořit sady jako je například sada zaměřená na externí závislosti nebo na zabezpečení atd.

4.2.4.3 Padající sada

Vedle označování systémových testů pro seskupení podle oblastí a rychlejších sad, které dokáží podat spolehlivou zpětnou vazbu, byla nakonfigurována i sada pro odkládání rozbitých testů. V předchozí kapitole byl popsán problém nestabilních testů, které znepříjemňují život snad všem členům vývojových týmů. Opravovat neúspěšné testy nemusí být vždy okamžitý úkon. Obzvláště u nestabilních testů se může jednat i o několikadenní záležitost. Nebo v jiných případech testy čekají, dokud není opraven bug vzniklý v softwaru. V takovýchto případech je nestabilní test stále neúspěšný – a to po právu. Vývojáři, kteří ověřují své změny, jsou poté zavaleni takovými testy, které by vůbec řešit neměli. Mezi takovými testy, i když je test master aktivně řeší, lze snadno přehlédnout nové problémy.

Pro takovéto účely vznikla nová sada. Skladiště rozbitých testů, ve které testy čekají na opravu zdrojového kódu, stabilizaci nebo na opravu sebe samých. Testy do této sady v průběhu ověření hlavní vývojové větve přesouvá test master jako součást jeho řízení oprav systémových testů. Je však nutné takto vyřazené testy pravidelně opravovat, aby se malá nestabilní sada nerozrostla do velkých rozměrů. V takovém případě se software vystavuje riziku, kdy se nelze na část testů spolehnout.

Shrnutí: Neúspěšné, nestabilní a testy čekající na opravu, mohou zastínit nově vzniklé problémy. Aby takovéto testy neznepříjemňovaly práci všem členům vývojového týmu, jsou testy ihned přemístěny do jiné sady (padající), a tím je čištěna hlavní sada systémových testů. Po opravě jsou testy opět navraceny zpět.

4.2.4.4 Paralelizace systémových testů

Rozdělení systémových testů do sad zrychlí zpětnou vazbu, kterou vývojáři obdrží. Aby však bylo provedeno kompletní ověření softwaru, je nutné spustit všechny testy ze všech testových sad. Tedy celková doba trvání jednoho průběhu na CI serveru bude trvat stále dlouho. Pokud se testové sady spustí souběžně na různých CI serverech (což je možné), bude doba trvání kompletního ověření trvat jako průběh nejdelší sady. Jedná se však o naplánované spouštění či manuální spouštění na volných CI serverech.

Vhodnějším způsobem, jak zkrátit dobu systémových testů, je nakonfigurovat CI server tak, aby podporoval paralelní běh automaticky. To není nic složitého, většina dostupných CI serverů tuto funkci nabízí. Díky paralelizaci lze dosáhnout zkrácení délky na polovinu, třetinu či čtvrtinu, to záleží na počtu zvolených paralelních větví a na výkonu stroje, na kterém je CI server konfigurován.

V současnosti nelze paralelní provádění systémových testů použít pro testování softwaru Inspire Scaler. Souběžné testy se provádějí nad stejnou instancí softwaru a velice se ovlivňují a vzájemně kolidují. Pro využití paralelního spouštění by bylo nutné určit testy, které se vzájemně neovlivňují a určit jejich přesné pořadí v obou zároveň běžících sadách, což je nesmírně obtížné, nákladné a téměř nemožné.

Shrnutí: Pro zkrácení doby trvání systémových testů je vhodné využít paralelní konfiguraci, kterou podporuje téměř každý CI server. Doba trvání je poté dělena počtem paralelních operací.

4.2.5 Infrastruktura CI serverů

Předchozí nakládání se systémovými testy, a tím je především myšleno rozdělení velkého balíku testů na menší spustitelné sady testů, přináší další problémy, které je potřeba vyřešit. Udržování všech konfigurací malých spustitelných testových sad na všech CI serverech, násobeno o počet aktivních verzí, je nezvladatelné. Jedná se o opakující se manuální činnosti konfigurace, ale i aktualizace všech potřebných externí systémů a služeb, které jsou pro běh systémových testů potřebné. V prvé řádě však musí být zabezpečeny následující požadavky:

- konfigurace malých sad musí být udržována na všech CI serverech,
- konfigurace musí být přenosná a lehce rozšířitelná,
- musí být zajištěno automatické spuštění všech testových sad,
- musí být zajištěno, že jsou sady spuštěny nad stejnou verzí softwaru.

4.2.5.1 TeamCity

Aktuální stav CI serverů je nevyhovující, a jak již bylo řečeno, jedná se o čtyři samostatné a na sobě nezávislé Jenkins CI servery. Každou změnu konfigurace je nutné aplikovat na všechny CI servery zvlášť. Zároveň je nutné výsledky stávající sestavené verze shromažďovat ze všech CI serverů na jedno místo. Největším problémem je však kapacita a výkon, která značně pokulhává. Z těchto důvodů je nutné celý koncept značně reorganizovat.

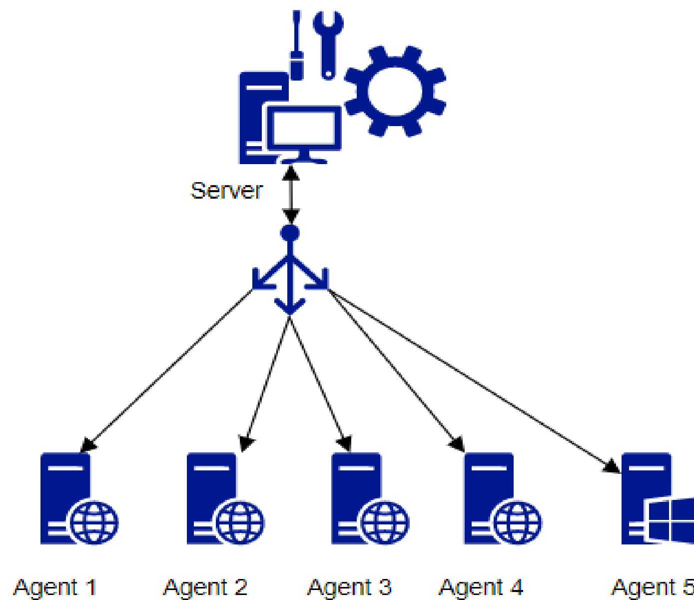
Při nutnosti změny CI serverů byla navrhnuta i změna druhu CI serveru. Po provedené analýze možností byl doporučen CI server TeamCity, který je vyvíjený společností JetBrains a o to více zapadá mezi další software určený pro podporu vývoje (Intelij IDEA, YouTrack). Toto rozhodnutí bylo podpořeno zkušeností jiných týmů a celofiremní politikou o sjednocování používaných nástrojů pro podporu vývoje. Bylo tedy rozhodnuto přistoupit na nový CI server TeamCity. TeamCity

přináší oproti klasickému základu další vylepšení, která zjednodušují činnosti při analýze a procházení spadlých testů. Mezi takové prvky především patři:

- automatické přihlašování osob na investigaci spadlých testů,
- automatické přihlašování osob na investigaci nefunkční konfigurace,
- automatické upozorňování na tzv. flaky testy (náhodně padající testy),
- možnosti automatického vytváření nových problémů (bug) do YouTrack,
- možnosti zobrazení změn ve zdrojovém kódu v místě problému,
- možnosti vestavěného sledování change logu,
- analyzační nástroje (historie, thread dump),
- kompletní logy při výsledcích testů,
- široké spektrum zpětné vazby.

Tyto dílčí až zanedbatelné kroky ušetří několik minut denně všem členům vývojových týmů, kterým odpadnou neoblíbené manuální činnosti, jako je dohledávání informací z logů, vyplňování formulářů, zakládání nových bug do externího systému YouTrack a podobně.

Vedle těchto drobností, které jsou spíše mikro optimalizací, byl CI server vystavěn jako server – agent. Server slouží jako řídící uzel, kde jsou vytvářeny a ukládány veškeré konfigurace a kde jsou koncentrovány výsledky z agentů, kteří vykonali jednotlivé operace. Agent je tedy výpočetní jednotka. Server slouží členům vývojového týmu jako přístupový bod. Ten zároveň rozdává úkoly ostatním agentům. Kapacita CI serveru byla současně rozšířena o další stroj, a vše je připraveno na okamžité přidání dalšího agenta.



Obrázek 14: Architektura aplikovatelného CI serveru
Zdroj: vlastní zpracování

Shrnutí: Jednotlivé testovací sady je potřeba konfigurovat na všech CI serverech, to může být náročné, v případě změny konfigurace je potřeba změnu zanést opět na všechny CI servery. Proto je vhodné využít CI server s centrální konfigurací a distribucí úkolů na podřízené výpočetní jednotky.

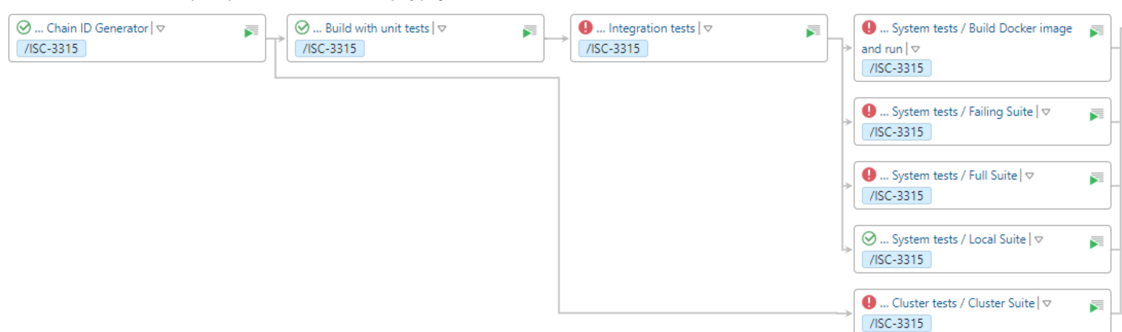
4.2.5.2 Build chain

Různé sady systémových testů, které jsou nově ověřovány na CI serveru TeamCity, mohou být spuštěny samostatně s různým nastavením. Lze tedy tvořit velice specifické až jedinečné kombinace konfigurace sad. To vše přispívá ke zvyšování kvality softwaru, neboť je testován na různě definovaných prostředích, a existuje tak větší pravděpodobnost na odhalení chyby.

Na druhou stranu je též potřeba zajistit, aby byla jedna verze softwaru kompletně otestována, tedy aby nad stejnou verzí softwaru proběhly všechny testové sady, a to nejlépe automaticky. CI server TeamCity pro takové účely poskytuje build chain (u jiných CI serverech bývá označován jako pipeline). Ten představuje řetězec po sobě jdoucích či paralelně zpracovaných úkolů. Do řetězce lze tedy nastavit například posloupnost malých testových sad či jiné další

automaticky vykonávané procesy. Součástí řetězce je i samotný build softwaru i jednotkové a integrační testy. Zároveň je možné nastavit build chain, aby se ukončil, pakliže první sady odhalí velké problémy.

Následující build chain spouští několik na sobě navazujících sad automatických testů. Po vygenerování příslušného identifikátoru, probíhá automatické sestavení s jednotkovými testy. Pakliže je sestavení a testy úspěšné, řetězec pokračuje integračními testy, po nich probíhají základní sady testů, tak aby byla verze softwaru komplexně ověřena. Další sada testů (Cluster tests) se spouští bez závislosti na vytvořené verzi během sestavení. Tu si totiž vytváří sama jiným způsobem.



Obrázek 15: TeamCity Build chain
Zdroj: vlastní zpravování

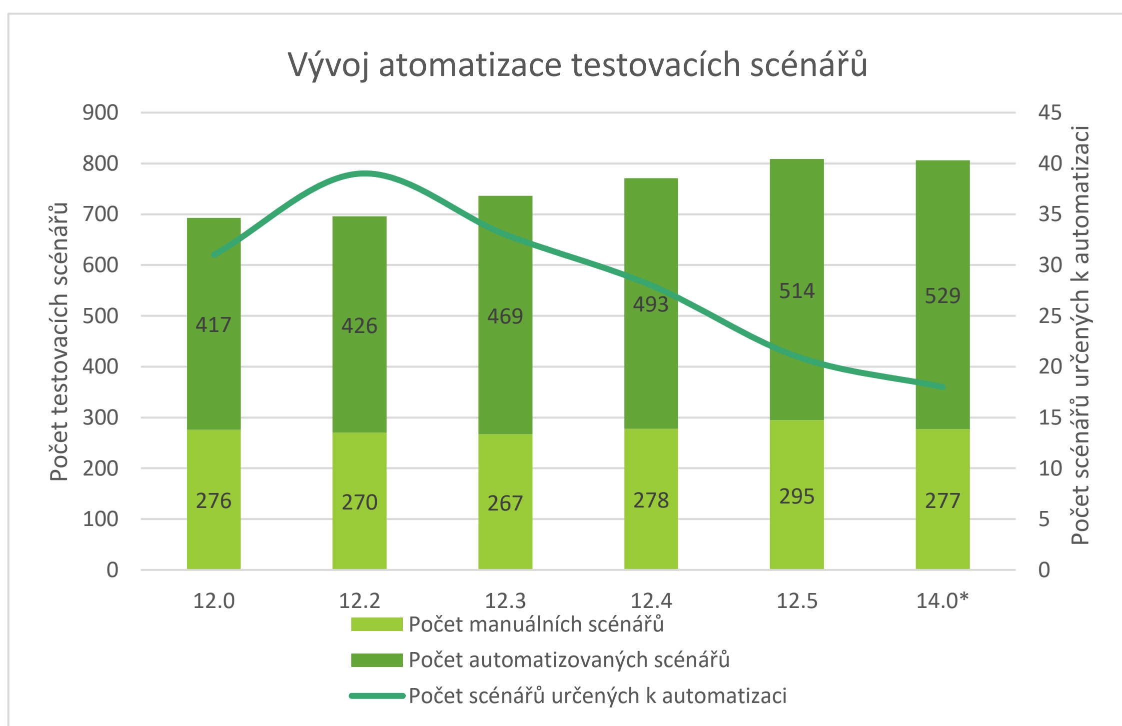
Shrnutí: Build chain představuje automatický způsob, jak zajistit komplexní spouštění všech sad systémových testů nad stejnou verzí společně s jednotkovými a integračními testy. Verzi softwaru je možné automaticky sestavit jako jeden z prvních dílčích úkolů. Pokud jsou testy úspěšné, je možné verzi automatiky nahrát do centrálního uložiště verzí.

5 Shrnutí výsledků

Aplikace výše uvedených kroků a řešení přinesla celou řadu velkých či dílčích zlepšení v různých oblastech automatického testování softwaru a celého continuous integration produktu Inspire Scaler.

5.1 Postupné automatizování manuálních testů

Velké množství manuálních testů je problémem především v regresním testování, kdy je potřeba všechny manuální testy provést. Vlivem rozvoje funkcionality softwaru narůstá i počet testovacích scénářů, tedy předpisů pro manuální či automatické testy. Následující *graf 1* znázorňuje vývoj manuálních a automatických testovacích scénářů za uplynulá období. Období jsou označena podle čísla vydávané hlavní verze softwaru Inspire Scaler s připravovanou verzí 14.0*. Z grafu vyplývá, že se počet scénářů, které jsou ověřovány manuálně, daří dlouhodobě držet na stejné úrovni. Jedná se o přibližný počet 275 scénářů. Naopak narůstá počet zautomatizovaných scénářů, průměrný přírůstek mezi jednotlivými vývojovými obdobími je 23 zautomatizovaných scénářů, zatímco v jednotlivých obdobích průměrně vzniká 21 nových scénářů.



Graf 1: Vývoj automatizace testovacích scénářů
Zdroj: vlastní zpracování

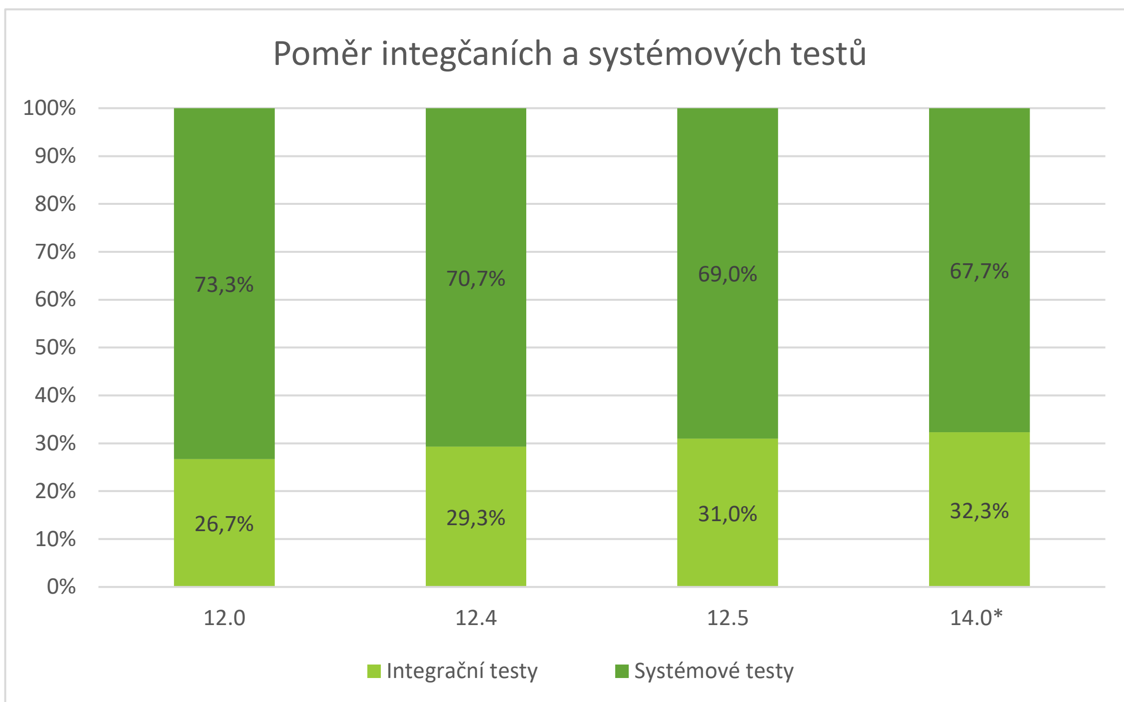
Větší počet zautomatizovaných scénářů je zaviněn dluhem z minulosti, který je postupně odbouráván. Počet scénářů určených pro automatizaci představuje druh scénáře, který lze ihned zautomatizovat (bez technologické limitace). Přestože jsou testy do této skupiny neustále přidávány, díky novým testovacím možnostem se daří tuto skupinu neustále snižovat. Na aktuální verzi je v současnosti 18 takovýchto scénářů.

Přestože počet manuálních scénářů neubývá závratnou rychlostí, lze celkově říci, že se daří automatizovat všechny scénáře k nově vzniklým US. Počet manuálních testů se tedy nezvyšuje. Zároveň klesá i počet automatů, kterým nic nebrání k automatizování.

5.2 Rozvoj integračních testů

V předchozích kapitolách bylo uvedeno, že není vhodné za všech okolností automatizovat testový scénář systémovým testem. Aby se předešlo zbytečnému používání systémových testů, byly zavedeny testovací analýzy před započetím implementace US. Následující *graf 2* uvádí procentuální poměr mezi počtem integračních a systémových testů. Mezi jednotlivými obdobími dochází k růstu počtu integračních testů na úkor systémových. Avšak do ideální podoby testovací pyramidy by bylo nutné dosáhnout přesně opačného poměru.

Přestože růst integračních testů se v poměru odehrává v jednotkách procent, mezi obdobím 12.4 a 12.5 vzniklo například téměř o 20 % integračních testů víc než systémových. Aktuální počet systémových testů je 1013 oproti tomu integračních pouze 483. Během jednoho období přibývá přibližně 80 automatických testů (plně automatizovaný testovací scénář může být realizován jedním či více automatickými testy). Výrazná změna poměru mezi integračními a systémovými testy vyžaduje delší časový úsek a úsilí. Avšak zavedení testovacích analýz přispívá k rozvoji prostředí pro testování na úrovni integračních testů, což bylo častou překážkou, a test byl raději koncipován jako systémový. Díky tomuto rozvoji můžeme do budoucna očekávat daleko vyšší uplatňování integračních testů.

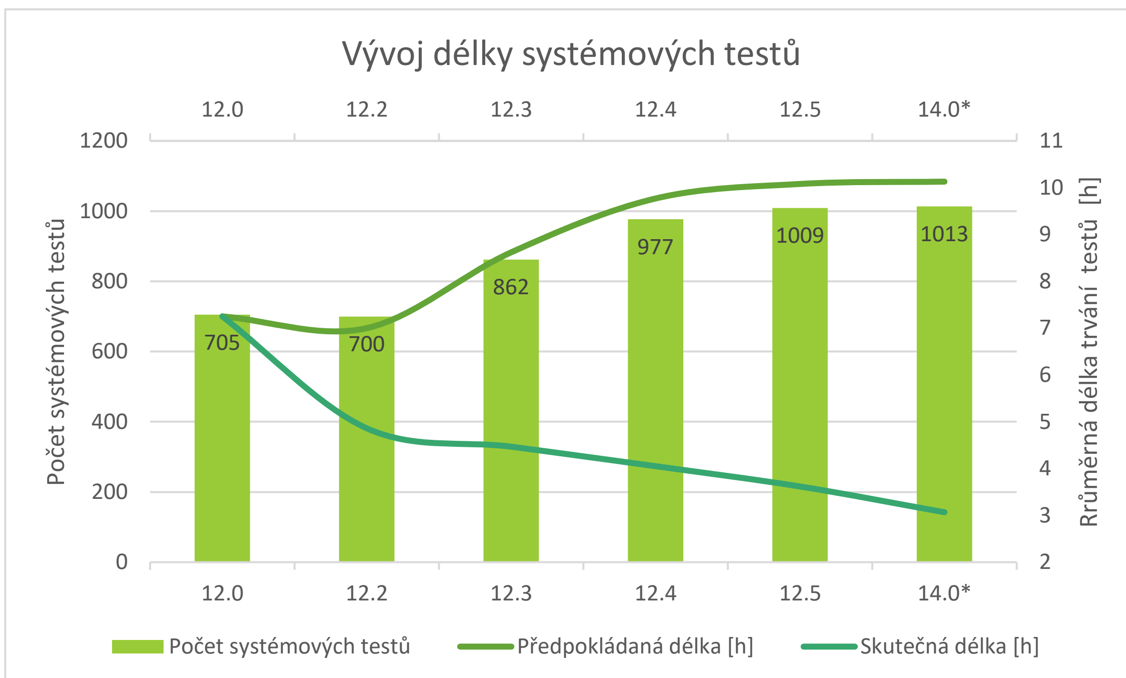


Graf 2: Poměr integračních a systémových testů
Zdroj: vlastní zpracování

5.3 Zrychlení celkového trvání systémových testů

Hlavní sada systémových testů trvá velmi dlouho, to přináší řadu problémů pro vývojáře, kteří by své změny rádi ověřili v rozumném čase. Vývoj doby trvání systémových testů znázorňuje následující graf. V období verze 12.0 systémové testy trvaly v průměru 7 hodin a 15 minut. Průměrná doba trvání jednoho testu byla 60 vteřin. Bylo očekáváno, že celková doba testování se bude navyšovat s přírůstem systémových testů. Tak znázorňuje i následující *graf 3*. Tomu se však podařilo zabránit, a dokonce byla doba trvání sady všech systémových testů výrazně snížena. Tak se stalo díky zefektivnění testového frameworuku, použití metod s aktivním čekáním a přepsání dlouhých systémových testů

Jak znázorňuje *graf 3*, aktuální průměrná délka trvání kompletní sady systémových testů jsou 3 hodiny a 4 minuty. Jedná se tak o velké zrychlení. *Graf 3* znázorňuje očekávanou délku systémových testů, a to více než 10 hodin, kterých by mohlo být dosaženo, pokud by tento problém nebyl řešen. Časový rozdíl mezi očekávanou dobou a skutečnou dobou je přes 7 hodin.

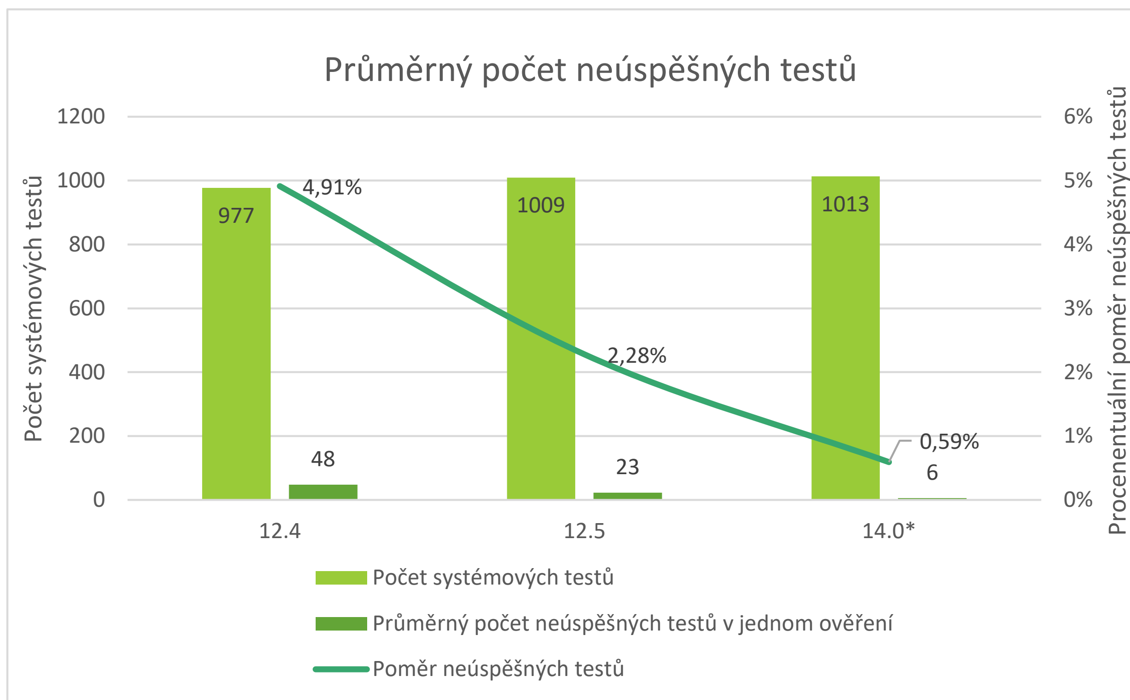


Graf 3: Vývoj doby systémových testů
Zdroj: vlastní zpracování

Velké snížení doby v průběhu verze 12.2 je do velké míry dáno velkým rušením a přepisování GUI testů, které testovaly validace na jednotlivých inputech. Tento druh testů byl velice nestabilní a hodnota, kterou přinášely, byla naopak velice nízká.

5.4 Zvýšení stability systémových testů

Nízká stabilita systémových testů byla taktéž jako jejich celková délka závažným problémem. Takovéto testy přinášejí rutinní každodenní činnosti a také podrývají důvěru v provedená ověření. Následující *graf 4* znázorňuje průměrný počet nestabilních, tzv. náhodně padajících testů, v průběhu systémových testů. Přestože se to jeví jako zanedbatelné množství, v celkovém počtu systémových testů se jedná pouze o průměrný počet pádů. Pakliže jsou testy nestabilní, padají například jednou za deset spuštění. Tedy počet nestabilních testů je mnohonásobně vyšší. *Graf 4* dále znázorňuje klesající tendenci poměru náhodně padajících testů oproti celkovému počtu systémových testů za poslední vývojová období.



Graf 4: Průměrný počet neúspěšných systémových testů v ověření
Zdroj: vlastní zpracování

Přestože náhodně padající testy budou do jisté míry existovat vždy, daří se jejich průměrný počet v každém ověření minimalizovat. Aktuálně se jedná průměrně o počet 6 neúspěšných testů, což je oproti předchozím počtu (48 testů) výrazné zlepšení. Stabilita systémových testů byla dosažena postupným přepisováním a vylepšováním testovacích metod a díky důslednému vykonávání činností spojených s rolí test mastera.

5.5 Zvýšení možnosti rychlé zpětné vazby

Velkým problémem byla neschopnost vývojářů okamžitě ověřovat své změny na CI serveru systémovými testy. Dlouho trvající systémové testy poskytovaly zpětnou vazbu až po několika hodinách, a to často ze všech různých oblastí softwaru, kterých se vývojářovy změny nemusely týkat. Aby byla zvýšena efektivita zpětné vazby, byly vytvořeny různé sady systémových testů, které ověří různé části softwaru a zároveň netrvají dlouho. Vývojář tak získá během několika minut základní přehled o stavu integrace jeho změn.

Především se jedná o sady systémových testů:

- Základní sada systémových testů (LocalSuite)
  - délka: 20 minut
- Základní sada pro ověření GUI (BasicGuiSuite)
  - délka: 17 minut
- Sada pro opravy a fix systémových testů (FailingSuite)
  - délka: 25 minut
- Sada systémových testů se zaměřením na externí služby (ExternalDependenciesSuite)
  - délka: 12 minut

Dále byl zaveden systém označování systémových testů, které ověřují konkrétní oblast softwaru. Podle těchto označení lze systémové testy pohodlně spustit. Dále byl zcela změněn CI server, tím byly mimo jiné odstraněny problémy s tvořením front na CI serveru. Díky tomu byla zjednodušena i konfigurace a ušetřeno mnoho dílčích manuálních kroků.

6 Závěry a doporučení

Diplomová práce představila automatické testování softwaru a metodiku continuous integration – její základní prvky a procesy. Dále popsala způsob aplikování prvků a procesů continuous integration při testování softwaru Inspire Scaler společnosti Quadient. Přinesla řešení a odpovědi na problémy s realizací a údržbou automatických testů, které vznikají při aplikování této metodiky. V prvé řadě se jednalo o automatizaci manuálních tesů, dobu trvání všech systémových testů a jejich stabilitu. Dále proběhla změna CI serveru, tak aby byly usnadněny manuální činnosti spojené s aktualizací a konfigurací jednotlivých strojů. Mimo jiné také vznikla široká škála zpětné vazby v podobě rychlých testových sad pro snadné ověření změn.

Celková doba průběhu systémových testů byla snížena z průměrných 7 hodin a 15 minut na průměrné 3 hodiny a 4 minuty *(graf 3).* Jedná se skutečně o výrazné zlepšení, kterého si cení všichni členové vývojového týmu. Rychlost byla snížena zejména používáním správných testovacích metod, přepsáním problémových dlouho trvajících testů, ale i dodržováním principů testovací pyramidy. Stabilita systémových testů byla taktéž zvýšena značným způsobem. Z průměrných 48 testů, které spadly v jednom průběhu systémových testů, došlo ke snížení na pouhých 6 (*graf 4*). Takovéto množství systémových testů je udržitelné a je snadno rozpoznatelné, zdali se jedná o náhodný pád systémového testu nebo o skutečný problém zaviněný špatnou implementací. To mimo jiné posílilo i důvěru v samotné systémové testování ze strany vývojářů.

Doporučením je v takto nastavených základních činnostech vytrvat, pokračovat v jejich vykonávání a dále je vylepšovat a upravovat podle aktuální potřeby. Neboť je možné, že se bez důrazného plnění zavedených činností může testování softwaru Inspire Scaler opět dostat do problémů.

Dalším budoucím tématem je zprovoznění spouštění systémových testů v paralelním běhu, aby se doba všech systémových testů ještě více zkrátila. Dále je možné pokračovat v automatizaci všech ostatních procesů tak, aby mohla být aplikována další úroveň continuous integration, a to continuous delivery.

7 Seznam použité literatury

[1] *Atlassian*: Software Development and Collaboration Tools [online]. Atlassian, 2019 [cit. 2019-05-08]. Dostupné z: https://www.atlassian.com/

[2] AZERI, Izzy. What is CI/CD?. Mabl: Automated software testing [online]. mabl, 2019, 23.1.2019 [cit. 2019-06-11]. Dostupné z: https://www.mabl.com/blog/what-is-cicd

[3] COCHRAN, Tim. Test Pyramid: the key to good automated test strategy. Medium [online]. 2019, 18.11.2017 [cit. 2019-04-25]. Dostupné z: https://medium.com/@timothy.cochran/test-pyramid-the-key-to-good-automated-test-strategy-9f3d7e3c02d5

[4] *CodeShip*: Continuous Integration, Deployment & Delivery with Codeship [online]. CodeShip, 2019 [cit. 2019-04-25]. Dostupné z: https://codeship.com/

[5] ČERMÁK, Miroslav. Black box test. CleverAndSmart: ICT management [online]. Miroslav Čermák, 2019, 30.10.2010 [cit. 2019-05-14]. Dostupné z: https://www.cleverandsmart.cz/black-box-test/

[6] DAWSON, Brian. 7 Signs You're Mastering Continuous Integration. DevOps: Where the world meets devops [online]. MediaOps, 2019, 8.5.2018 [cit. 2019-04-25]. Dostupné z: https://devops.com/7-signs-youre-mastering-continuous-integration/

[7] DUSTIN, Elfriede, Thom GARRETT a Bernie GAUF. Implementing Automated Software Testing: How to Save Time and Lower Costs While Raising Quality. 1. George Mason University: Pearson Education, 2009. ISBN 9780321619594.

[8] DUVALL, Paul M., Steve MATYAS a Andrew GLOVER. Continuous integration: improving software quality and reducing risk [online]. Upper Saddle River, NJ: Addison-Wesley, 2007 [cit. 2019-04-18]. ISBN 978-0-321-33638-5.

[9] ERIKSSON, Ulf. The A to Z Guide to the Software Testing Process. ReQtest: Requirements, Test Management, Bug Tracking Tool [online]. ReQtest, 2019, 19. 5. 2016 [cit. 2019-04-23]. Dostupné z: https://reqtest.com/testing-blog/the-a-to-z-guide-to-the-software-testing-process

[10] FITZGERALD, Brian a Klaas-Jan STOL. Continuous software engineering: A roadmap and agenda. Journal of Systems and Software [online]. vol. 123, 2017, 176-189 [cit. 2019-04-18]. ISSN 0164-1212. Dostupné z: http://www.sciencedirect.com/science/article/pii/S0164121215001430

[11] FOWLER, Martin. 2006. Continuous Integration. Martin Fowler – blog. 1.5.2006 [online]. [cit. 2019-03-27]. Dostupné z: https://martinfowler.com/articles/continuousIntegration.html

[12] GAROUSI, Vahid, Michael FELDERER a Feyza Nur KILIÇASLAN. A survey on software testability. Information and Software Technology [online]. 2019, 2019, (108), 35-64 [cit. 2019-05-12]. ISSN 0950-5849. Dostupné z: http://www.sciencedirect.com/science/article/pii/S0950584918302490

[13] *Geb*: Very Groovy Browser Automation [online]. Apache License, 2019 [cit. 2019-04-24]. Dostupné z: http://www.gebish.org/

[14] *GitLab*: The first single application for the entire DevOps lifecycle – GitLab [online]. GitLab, 2019 [cit. 2019-05-08]. Dostupné z: https://about.gitlab.com/

[15] GREGORY, Janet a Lisa CRISPIN. More agile testing: learning journeys for the whole team. Upper Saddle River, NJ: Addison-Wesley, [2015]. ISBN 978-0-321-96705-3.

[16] HLAVA, Tomáš. 2011. Fáze a úrovně provádění testů. Testování softwaru. 21.8.2011 [online]. [cit. 2019-03-27]. Dostupné z: http://testovanisoftwaru.cz/category/metodikatestovani/druhy-typy-a-kategorie-testu/

[17] HORNBEEK, Marc. Does Agile need continuous integration. DevOps: Where the world meets devops [online]. MediaOps, 2019, 26. 10. 2015 [cit. 2019-04-19]. Dostupné z: https://devops.com/agile-need-continuous-integration/

[18] CHACON, Scott a Ben STRAUB. Pro Git: [everything you need to know about the Git distibuted source control tool] [online]. 2nd ed. New York, NY: Apress, 2014 [cit. 2019-04-19]. CZ.NIC. ISBN 14-842-0077-2. Dostupné z: https://github.com/progit/progit2/releases/download/2.1.146/progit.pdf

[19] *Jenkins*: Build great things at any scale [online]. 2019 [cit. 2019-05-07]. Dostupné z: https://jenkins.io/

[20] *JetBrains*: Developer Tools for Professionals and Teams [online]. JetBrains, 2019 [cit. 2019-05-07]. Dostupné z: https://www.jetbrains.com/

[21] *JUnit 5* [online]. The JUnit Team, 2019 [cit. 2019-06-11]. Dostupné z: https://junit.org/junit5/

[22] KARVONEN, Teemu. Continuous software engineering in the development of software-intensive products: Towards a reference model for continuous software engineering. 1. University of Oulu: Acta Univ. Oulu. A 695, 2017. ISBN 978-952-62-1655-3.

[23] KERNER, Sean Michael. Continuous Delivery and Continuous Deployment: Keys to the DevOps Revolution. *Datamation: Emerging Enterprise Tech Analysis and Products* [online]. Quinstreet, 2019, 16.4.2019 [cit. 2019-04-25]. Dostupné z: https://www.datamation.com/data-center/continuous-delivery-continuous-deployment.html

[24] *Kotlin*: Kotlin Programming Language [online]. JetBrains, 2019 [cit. 2019-06-30]. Dostupné z: https://kotlinlang.org/

[25] MCKENZIE, Cameron. Git vs. GitHub: What is the difference between them?. TheServerSide.com: your Java Community discussing server side development [online]. TechTarget, 2019, 6.11.2018 [cit. 2019-04-25]. Dostupné z: https://www.theserverside.com/video/Git-vs-GitHub-What-is-the-difference-between-them

[26] PACKER, Dan. Continuous Integration vs. Continuous Delivery vs. Continuous Deployment. Plutora: Deliver better software faster [online]. Plutora, 2019, 7.3.2019 [cit. 2019-04-19]. Dostupné z: https://www.plutora.com/blog/continuous-integration-continuous-delivery-continuous-deployment

[27] PATTON, Ron. *Software testing* [online]. 1. Indianapolis, Ind.: Sams, 2001 [cit. 2019-04-22]. ISBN 978-0672319839. Dostupné z: https://shekharsk.files.wordpress.com/2016/01/ron-patton-software-testing.pdf

[28] PITTET, Sten. 2018. Continuous integration vs. continuous delivery vs. continuous deployment. [online]. [cit. 2019-04-25]. Dostupné z: https://www.atlassian.com/continuous-delivery/ci-vs-ci-vs-cd

[29] POBA-NZAOU, Placide a Sylvestre UWIZEYEMUNGU. Worries of open source projects' contributors: Patterns, structures and engagement implications. Computers in Human Behavior [online]. 2019, (96), 174-185 [cit. 2019-05-07]. ISSN 0747-5632. Dostupné z: http://www.sciencedirect.com/science/article/pii/S0747563219300573

[30] RUS, Calin. Deploying and Scaling Jenkins on Kubernetes. Rancher: Container Orchestration | Kubernetes Management | Rancher [online]. RANCHER LABS, 2019, 27.11.2018 [cit. 2019-04-27]. Dostupné z: https://rancher.com/blog/2018/2018-11-27-scaling-jenkins/Excella: Agile Technology in Washington DC [online]. Washington DC: Excella Co, 2019 [cit. 2019-04-27]. Dostupné z: https://www.excella.com/

[31] SHAHIN, Mojtaba, Muhammad Ali BABARA a Liming ZHUB. Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices [online]. Sydney, 2015 [cit. 2019-04-18]. Dostupné z: https://www.researchgate.net/publication/315381994_Continuous_Integration_Delivery_and_Deployment_A_Systematic_Review_on_Approaches_Tools_Challenges_and_Practices

[32] SHINDE, Vijay. Types of Automation Testing and Some Misconceptions. *Software Testing Help: Free Software Testing* [online]. Software testing help, 2019, 23.4.2019 [cit. 2019-04-23]. Dostupné z: https://www.softwaretestinghelp.com/automation-testing-tutorial-2/

[33] VOCKE, Ham. 2018. The Practical Test Pyramid. 26.2.2018 [online]. [cit. 2019-04-23]. Dostupné z: https://martinfowler.com/articles/practical-test-pyramid.html.

[34] *Stackshare*: Software and technology stacks used by top companies [online]. StackShare, 2019 [cit. 2019-05-08]. Dostupné z: https://stackshare.io/

[35] *Step2QA*: Elevate Quality Engineering [online]. Step2QA, 2018 [cit. 2019-04-23]. Dostupné z: http://www.step2qa.com/

[36] *Techopedia*: Where Information Technology and Business Meet [online]. Techopedia, 2019 [cit. 2019-04-23]. Dostupné z: https://www.techopedia.com/

[37] *Try QA*: Study material for ISTQB Exam Certification Foundation level, Premium & Free for ISTQB and ASTQB Exam, Certification questions, answers, software testing tutorials and more [online]. tryqa.com, 2019 [cit. 2019-04-23]. Dostupné z: http://tryqa.com/

[38] VORA, Nimish. Continuous Integration (CI) & Continuous Deployment (CD) – Bandwagon of Agile Development. Volansys: Product Realization and Digital Transformation Company [online]. Volansys Technologies, 2019, 8.11.2016 [cit. 2019-04-25]. Dostupné z: https://volansys.com/continuous-integration-continuous-deployment-bandwagon-of-agile-development/

[39] *WireMock*: WireMock [online]. Tom Akehurst, 2019 [cit. 2019-06-11]. Dostupné z: http://wiremock.org/

[40] *Yaml*: Two Projects One Name [online]. 2019 [cit. 2019-05-12]. Dostupné z: http://www.yaml.com/

[41] *Zuzi's blog:* Agile and Lean, Scrum, Kanban, XP @ Business [online]. Zuzi's blog, 2019 [cit. 2019-06-27]. Dostupné z: https://soch.cz/blog/

8 Přílohy

1) **Tabulka 1: Průměrná doba ověření systémových testů**

2) **Tabulka 2: Počet testovacích scénářů**

3) **Tabulka 3: Průměrný počet neúspěšných systémových testů v ověření**

4) **Tabulka 4: Poměr integračních a systémových testů**

Tabulka 1: Průměrná doba ověření systémových testů

Verze	12.0	12.2	12.3	12.4	12.5	14.0*
Počet systémových testů	705	700	862	977	1009	1013
Předpokládána délka [m]	435	420	517	586	605	608
Skutečná délka [m]	435	292	268	243	217	184
Předpokládána délka [h]	7,25	7,00	8,62	9,77	10,08	10,13
Skutečná délka [h]	7,25	4,86	4,47	4,05	3,62	3,07

Tabulka 2: Počet testovacích scénářů

Verze	12.0	12.2	12.3	12.4	12.5	14.0*
Manuální scénáře	276	270	267	278	295	277
Automatizované scénáře	417	426	469	493	514	529
Manuální scénáře určené k automatizaci	31	39	33	28	21	18
Manuální scénáře celkem	307	309	300	306	316	295
Ostatní scénáře	5	8	5	12	19	13
Testovací scénáře celkem	729	743	774	811	849	837

Tabulka 3: Průměrný počet neúspěšných systémových testů v ověření

Verze:	12.4	12.5	14.0*
Počet systémových testů	977	1009	1013
Průměrný počet neúspěšných testů v ověření	48	23	6
Poměr: neúspěšné teste	4,91 %	2,28 %	0,59 %

Tabulka 4: Poměr integračních a systémových testů

Verze:	12.0	12.4	12.5	14.0*
Systémové testy	705	977	1009	1013
Integrační testy	257	405	453	483
Celkem	962	1382	1462	1496
Poměr: integrační testy	26,7 %	29,3 %	31,0 %	32,3 %
Poměr: systémové testy	73,3 %	70,7 %	69,0 %	67,7 %

Oskenované zadání práce

Univerzita Hradec Králové
Fakulta informatiky a managementu
Akademický rok: 2018/2019

Studijní program: Systémové inženýrství a informatika
Forma: Prezenční
Obor/komb.: Informační management (im2-p)

Podklad pro zadání DIPLOMOVÉ práce studenta

PŘEDKLÁDÁ:	ADRESA	OSOBNÍ ČÍSLO
Bc. Podzimek Jan	Krsmol 27, Stará Paka - Krsmol	I1700348

TÉMA ČESKY:

Continuous integration - aplikace automatického testování softwaru

TÉMA ANGLICKY:

Continuous Integration - application of automated software testing

VEDOUCÍ PRÁCE:

Ing. Martina Husáková, Ph.D. - KIT

ZÁSADY PRO VYPRACOVÁNÍ:

Cílem diplomové práce je popsat automatické testování softwaru jako nedílnou součást metodiky vývoje softwaru continuous integration. Dále je pak cílem specifikovat základní problémy testovacího projektu produktu Inspire Scaler společnosti Quadient při aplikování continuous integration a navrhnout jejich možná řešení.

SEZNAM DOPORUČENÉ LITERATURY:

DUVALL, Paul M., Steve MATYAS a Andrew GLOVER. Continuous integration: improving software quality and reducing risk [online]. Upper Saddle River, NJ: Addison-Wesley, 2007, ISBN 978-0-321-33638-5

FOWLER, Martin. 2006. Continuous Integration. Martin Fowler - blog. 1.5.2006 [online]. Dostupné z: https://martinfowler.com/articles/continuousIntegration.html

1. Úvod
2. Cíl práce
3. Teoretická východiska (Automatické testování softwaru; Continuous integration; CI server)
4. Praktická část (Inspire Scaler; Řešení problému s aplikací continuous integration)
5. Shrnutí výsledků
6. Závěry a doporučení

Podpis studenta: .... **Datum:** 22.7.2019

Podpis vedoucího práce: .... **Datum:** ....

(c) IS/STAG , Portál - Podklad kvalifikační práce , I1700348 , 22.07.2019 06:52