Univerzita Hradec Králové

Fakulta informatiky a managementu

Katedra informačních technologií

Implementace automatizovaných testů pro efektivní vývoj softwaru

Diplomová práce

Autor: Bc. David Pfeifer

Studijní obor: Informační management

Vedoucí práce: Mgr. Josef Horálek, Ph.D.

Hradec Králové duben 2021

2021

Prohlášení:

Prohlašuji, že jsem diplomovou práci zpracoval samostatně s použitím uvedené literatury.

podpis

V Hradci Králové dne 02.01.2021 Bc. David Pfeifer

Poděkování:

Rád bych zde poděkoval Mgr. Josefu Horálkovi, Ph.D. za jeho odborné vedení, veškeré konzultace, informace a čas, který mi věnoval před a při zpracovávání mé diplomové práce. Dále děkuji společnosti Unicorn za možnost použít konkrétní projekt pro účely praktické části.

Anotace

Diplomová práce ve své teoretické části pojednává o důležitosti testování softwaru v rámci vývojového procesu a představuje oblast automatizace testů, která se stává nezbytným doplňkem při implementaci moderních informačních systémů. Automatizace zvyšuje kvalitu dodávané aplikace a v průběhu času snižuje finanční i časové náklady. Práce se dále zaměřuje na charakteristiku konkrétních nástrojů, které lze pro realizaci automatizovaných testů použít. Praktická část se zabývá návrhem a implementací automatizovaných testů pro zvýšení efektivity vývojového procesu v rámci mezinárodního projektu Nordic Balance Settlement. Na základě definovaných požadavků a cílů jsou vykonány detailní analýzy vybraných nástrojů. Výsledky výzkumu definují nástroj Gauge, jako nejvhodnější pro automatizaci testů na daném projektu. Proces návrhu zahrnuje praktické příklady integračních testů a testů uživatelského rozhraní. Práce dále obsahuje doporučení, jakými možnými způsoby lze zvyšovat kvalitu testovacího procesu a dodávaného softwaru.

Annotation

Title: Implementation of automated tests for efficient software development

The first part of this diploma thesis talks about the importance of software testing within the development process and introduces the area of test automation which becomes a necessary complement for the implementation of modern information systems. Automation increases the quality of the delivered software application and decreasing financial and time costs over time. The scope of thesis is than focused on the characteristics of specific tools which can be used for test automation. The second application part is focused on design and implementation of automated tests for increasing the efficiency of development process within the international software project called Nordic Balance Settlement. Detailed analysis of the chosen tools is performed based on the defined requirements and goals. The results of research define Gauge tool as the most suitable for test automation on the given project. The process of design includes practical examples of integration tests and graphical user interface tests. Document also contains the best practices for test automation and possible ways how to increase the quality of the testing process and developed software.

Obsah

Seznam obrázků

Seznam tabulek

1 Úvod

V současném moderním světě a nastupující generaci internetu věcí se tvoří obrovská poptávka po digitalizaci dat a zpracování obsahu online. Společně s tím převážná část firem investuje nemalé peníze do rozvoje v oblasti digitálních technologií a softwaru. S rostoucím zájmem o software se zvyšují i požadavky klientů na jejich dodavatele a systém samotný. Chtějí, aby jednotlivé komponenty byly co nejrychleji naimplementované, ve vysoké kvalitě při splnění nejrůznějších funkčních i nefunkčních požadavků. Na základě toho se společnosti zabývající tvorbou softwaru snaží celý proces co nejvíce zefektivnit a zároveň udržet vysokou přidanou hodnotu.

Vše začíná již u samotného návrhu informačního systému. Všechny komponenty, požadavky a funkcionality by měly být dopodrobna popsány v rámci detailních dokumentací. Proces posléze pokračuje s neméně důležitou implementační částí, na kterou navazuje často opomíjená součást životního cyklu vývoje softwaru, kterou je bezesporu fáze testování. Testování a následná automatizace jsou hlavními tématy této diplomové práce. Správně navržené a naimplementované automatické testy vedou ke zkvalitnění, zrychlení a lepší integraci celého informačního systému. Dalšími benefity automatizace jsou časová úspora lidských zdrojů a možnost opakovaného spouštění sad testů.

Obsah práce je rozvržen do dvou hlavních částí, teoretické a praktické. Teoretická část vychází z odborných rubrik, článků a literatury zaobírající se principy vývoje softwaru, testováním a automatizací testování. V jednotlivých kapitolách jsou dále charakterizovány nejznámější nástroje sloužící k tvorbě, aplikaci, správě a vyhodnocení výsledků automatizovaných testů. Na závěr teoretické části je krátce představena metodika průběžné integrace včetně nástrojů.

Účelem navazující praktické části je výběr vhodných nástrojů, návrh a implementace automatizovaných testů pro využití na projektu Nordic Balance Settlement. Následná aplikace testů by měla podpořit a zefektivnit vývoj budoucích verzí a funkcionalit v rámci projektu.

2 Cíl práce, volba metodiky a postupy řešení

Diplomová práce se zabývá návrhem automatizovaných testů v rámci vývoje informačního systému Nordic Balance Settlement (NBS), který funguje na principu RTOS (operační systém reálného času). Na základě kontinuálně přicházejících požadavků od zákazníka, projekt dlouhodobě implementuje nové verze, zahrnující rozšíření stávajících funkcionalit. Z důvodu opakující se potřeby regresních testů je použití automatizace na místě.

Po úvodním představení specifik NBS následuje vydefinování výchozího stavu testování na projektu. Navazující analýza požadavků pro automatizaci testování stanovuje hlavní cíl, jímž je návrh, implementace a aplikace dvou typů automatizovaných testů. Základní podmínkou je znovu použitelnost všech testů.

1. Integrační testy – pro pokrytí vzájemné integrace jednotlivých komponent a interních procesů, jejichž funkcí je zakládání strukturních entit nebo ukládání hodnot do systému. Dílčím cílem je koncept sady testů pro ověření kondice testovacích prostředí.

2. Testy uživatelského rozhraní – pro pokrytí všech možných případů užití v rámci front endu a části back endu.

Před samotným návrhem a implementací automatizovaných integračních testů a testů uživatelského rozhraní jsou představeny vybrané testovací nástroje, které jsou v první fázi podrobeny detailní funkční analýze, na zjištěné výsledky plynule navazuje druhá fáze zastoupená SWOT analýzou. Na základě výstupů zkoumání je vybrán nejvhodnější nástroj pro navazující automatizaci. Jednotlivé testy jsou v rámci tvorby spouštěny a laděny lokálně. Tato část práce obsahuje příklady a doporučení pro návrh jednotlivých typů testů. Posléze jsou exekuce testovacích skriptů a reportování výsledků integrovány do nástroje TeamCity v rámci kontinuální integrace. Kombinací představených metodik a nástrojů jsme schopni vylepšit proces testování i vývoje softwaru jako celku. Celá práce je zakončena souhrnným vyhodnocením s ohledem na stanovené cíle a původní stav. Očekávanými výstupy implementace automatických testů jsou dále: úspora

času stráveného manuálními testy, rychlejší exekuce na pravidelné bázi, okamžité vyhodnocení výstupu.

Dílčím cílem díla je seznámení čtenáře s komplexitou problematiky testování softwaru. Popsat jednotlivé techniky, úrovně a typy testů. Nastínit, že testování neznamená jen manuální klikání a úkony v aplikaci podle předem daného scénáře, ale že může nabývat mnoha významů, jak v podobě manuálních, tak i automatických testů. To vše je doplněno o poznatky autora na základě mnohaletých zkušeností v oblasti testování softwaru.

3 Vývoj softwaru

3.1 Definice softwaru a webových aplikací

Software je sada instrukcí, dat a programů používaných k vykonání konkrétních výpočetních operací a úkolů. Zjednodušeně se jedná o protiklad hardwaru, který definuje fyzické aspekty elektronických zařízení. Software je obecný termín použitý k označení skriptů, aplikací a programů, které jsou na různých zařízeních spuštěny. Lze ho rozdělit do dvou hlavních kategorií, systémový software a aplikační software. (Rosencrance, 2021)

Systémový software je navržen pro správný provoz a řízení hardwaru počítače, dále poskytuje důležitou platformu pro následné spuštění aplikačních programů. Mezi nejpoužívanější systémové softwary patří například operační systém, systémové nástroje či firmware. (Rosencrance, 2021)

Aplikační software se dá bezesporu označit jako nejběžnější typ. Provozovaná aplikace je určena k naplnění konkrétních potřeb uživatele. Aplikace může pracovat samostatně nebo se může skládat z více spolupracujících programů. Jako aplikaci označujeme například webové prohlížeče, textové a grafické editory, komunikační platformy, programy pro správu databází apod. (Rosencrance, 2021)

Webová aplikace patří mezi aplikační software, který je spuštěn na vzdáleném webovém serveru. Jak je zmíněno výše, tradiční desktopové aplikace běží v rámci operačního systému uživatele. Oproti tomu na většinu webových aplikací je přistupováno za pomocí webového prohlížeče. Mezi výhody webových aplikací patří nezávislost na platformě, ze které uživatel přistupuje (Windows, Linux), stačí pouze prohlížeč a přístup k internetu, nebo případně intranetu v rámci korporátní sítě. Vývojáři nemusí distribuovat aktualizaci aplikace mezi jednotlivé uživatele, aktualizace aplikace probíhá přímo na webovém serveru, na kterém aplikace běží. Vložená data se zpracovávají a ukládají vzdáleně, to umožňuje přístup a práci se stejnými daty z více zařízení. Nevýhodou mohou být v tomto případě hardwarové limity webového serveru. (TechTerms, 2014)

3.2 Druhy webových aplikací

V průběhu posledních let došlo k významnému nárůstu počtu webových aplikací a jejich potřeb. Paralelně s tím se neustále posouvá i vývoj webových aplikací a zavádí se nové technologie a standardy. Rozlišujeme dva základní druhy webových aplikací: statické a dynamické.

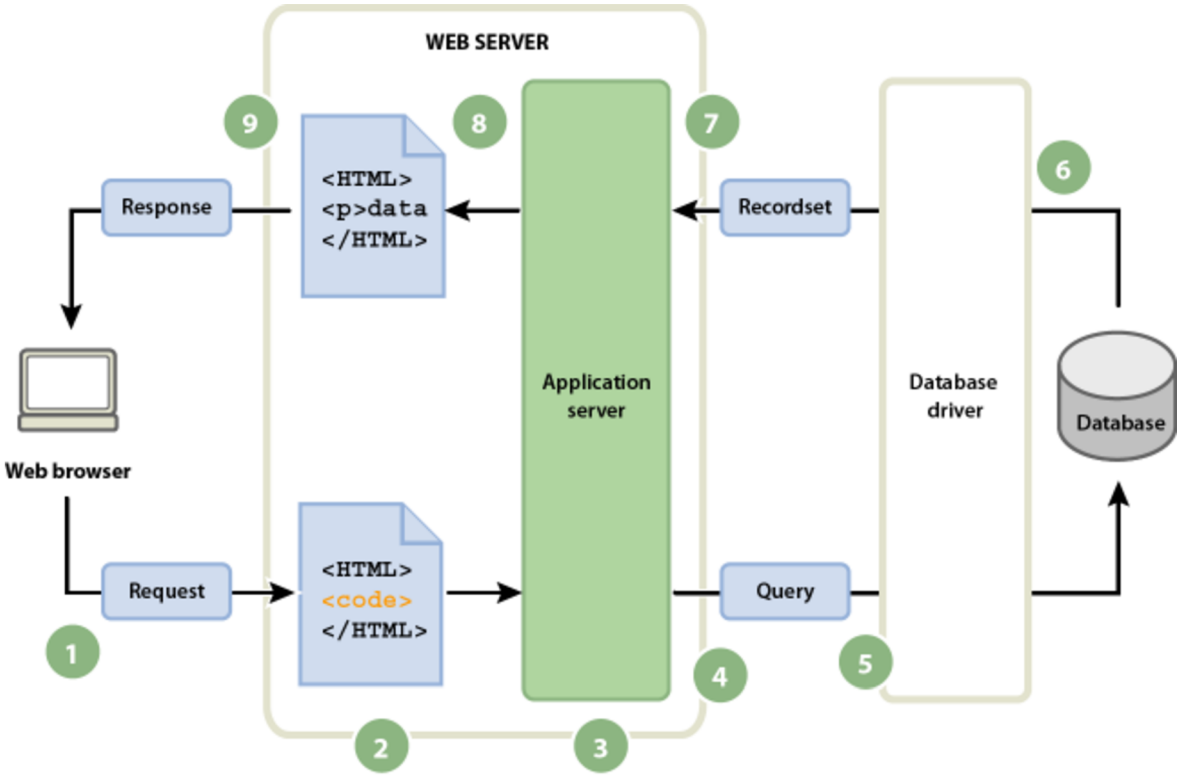
3.2.1 Statické webové aplikace

Statické webové aplikace se vyznačují omezeným obsahem s minimální interaktivitou a bez možnosti personalizace. Taková aplikace se uživateli zobrazí přesně tak, jak je uložena na serveru. Při tvorbě statických webových aplikací se nejčastěji používá HTML[1] a CSS[2]. Tento typ aplikací je vhodný například pro sdělení informací uživateli bez nutnosti dalších interakcí. (Ravi Makhija, 2021)

3.2.2 Dynamické webové aplikace

Na rozdíl od statických webových aplikací jsou robustnější, komplexnější a technicky složitější. Umí pracovat s daty v reálném čase v závislosti na požadavku uživatele. Tento typ webové aplikace vyžaduje k ukládání a načítání dat databázi, která je průběžně v čase aktualizována. Databáze se nachází na serverové straně. Požadavek obdržený webovým serverem je zpracován na aplikačním serveru, který spolupracuje s databází. Po zpracování požadavku a načtení dat z databáze zasílá aplikační server odpověď zpět do prohlížeče přes webový server. (Ravi Makhija, 2021)

[1] Hyper Text Markup Language
[2] Cascading Style Sheets



Obrázek 1 - Dynamická webová aplikace (Guru Technolabs, 2021)

Pro tvorbu dynamických webových aplikací je možné použít několik typů programovacích jazyků, například Node.js[3], PHP[4], HTML5, CSS, Python[5] apod. U vývojářů zaměřených na front-end jsou oblíbené Angular, React a JavaScript. (Ravi Makhija, 2021)

Podle Ravi Makhija dále rozlišujeme několik typů dynamických webových aplikací:

- Single-page aplikace – jednostránkové aplikace umožňující okamžitou interakci s webovou stránkou.
- Multi-page aplikace – fungují podobně, jako tradiční webové aplikace, aplikace načte a zobrazí novou stránku kdykoli uživatel provede akci.
- Animované webové aplikace – obsah reprezentovaný animovanými efekty.
- Progresivní webové aplikace – zobrazují obsah na webu podobně jako mobilní aplikace.

[3] https://www.w3schools.com/nodejs/nodejs_intro.asp
[4] https://www.w3schools.com/php/
[5] https://www.w3schools.com/python/python_intro.asp

- Portálové webové aplikace – ideální pro komerční sféru a vzdělávání, registrovaní uživatelé mají přístup na portál odkud mohou využívat specifické funkce.
- Aplikace pro e-commerce – e-shopy (online obchody), složitější vývoj z důvodu integrace platebních metod, modifikace produktů, zpracování plateb apod.

(Ravi Makhija, 2021)

3.3 Metodiky pro vývoj softwaru

V důsledku globálního růstu společností a jejich přímého zapojení v oblasti informačních technologií, se softwarový vývoj neustále posouvá kupředu a mění. Tradiční metodiky pro vývoj softwaru již nejsou efektivní pro vývoj moderních webových aplikací. (Molina-Ríos, Pedreira-Souto, 2020)

Vznik nových metodik vyplývá z odlišných potřeb pro výrobu webových aplikací. Musí se přizpůsobit odlišnému životnímu cyklu, specifickým funkcím i jinému prostředí. Stále častěji se můžeme setkat s pojmem agilní vývoj. Agilní metodiky obsahují možnosti, jak řešit vzniklé problémy během vývojového procesu, jakými mohou být neustálé změny požadavků klienta, nedostatek zabezpečení, problémy ve specifikaci a implementační defekty. (Molina-Ríos, Pedreira-Souto, 2020)

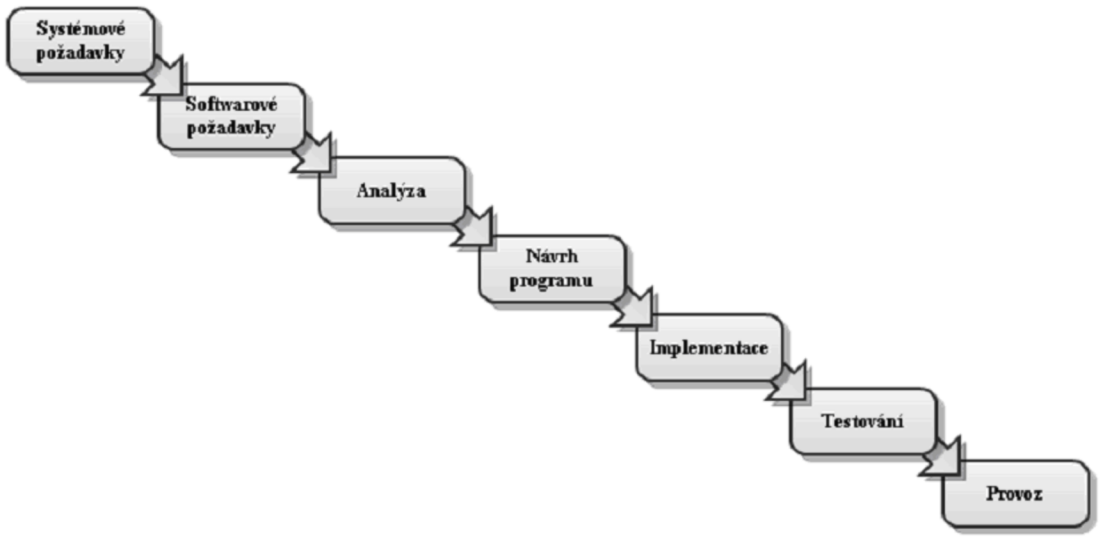
Použití správné metodiky vývoje je klíčové pro úspěšné dodání softwaru. Velké množství nově vzniklých metodik v kombinaci se složitostí moderních informačních systémů důležitost výběru ještě zvyšuje.

O významu výběru správné metodiky se zmiňuje i James Chapman: „*složitým problémem při výběru a dodržování metodiky je činit tak s rozumem – poskytnout dostatek procesních disciplín k zajištění kvality vedoucí k obchodnímu úspěchu, ale zároveň se vyvarovat kroků, které představují ztrátu času, snižují produktivitu, demoralizují vývojáře a vytvářejí nepotřebnou administrativu.*“ (Chapman, 2011)

3.3.1 Tradiční vývojové metodiky

Dle Molina-Ríos a Pedreira-Souto je *„cílem tradičních vývojových metodik přistupovat ke kompletně specifikovaným problémům důsledným plánováním, předdefinovanými procesy a pravidelnou dokumentací“.* (Molina-Ríos, Pedreira-Souto, 2020)

Nejznámější a nejstarší metodika tradičního vývoje se nazývá vodopádový přístup. Pojmenování je odvozeno z přirovnání posloupnosti dílčích fází k protékání vody vodopádem. Winton W. Royce tento přístup definoval již v roce 1970 a ve svém článku se zmiňuje o 7 základních fázích, které můžeme vidět na obrázku níže. (Royce, 1970)



Obrázek 2 – Vodopádový přístup vývoje softwaru

Podstata vodopádového modelu vychází ze sekvenčního přístupu k jednotlivým fázím. Zahájení některé z nadcházejících fází je možné pouze v případě, že je předchozí fáze kompletně dokončena a uzavřena. (Boehm, 1988)

Důležité je nepodcenit počáteční fáze a věnovat jim dostatek času. Jen tak lze v pozdějších fázích životního cyklu získat úspory. Nalezení a vyřešení chyby, která se objeví v počátečních fázích, je mnohem levnější, než kdybychom tutéž chybu měli řešit v pozdější fázi testování. Pokud chceme postupovat exaktně podle vodopádového modelu, musíme si být maximálně jisti výstupem každé fáze. Zpravidla využit u menších projektů, kde se neočekávají pozdější změny funkcionalit a vlastností. (Boehm, 1988)

Hlavní slabinou vodopádového přístupu je, že v praxi je často nemožné čekat na ukončení jedné fáze, než začne další a zároveň se nevracet k předchozím fázím životního cyklu. Přání zákazníka se mohou v průběhu realizace měnit a model neumožňuje na tyto pozměňovací požadavky efektivně reagovat. Velmi neúčinné je vyčkávání s testováním až na konec implementační části, kdy by měla být aplikace téměř připravena na

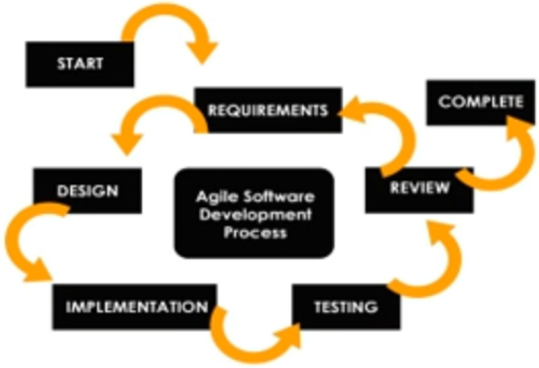
předání zákazníkovi. Odstranění chyb v této fázi je daleko náročnější v porovnání s fázemi analýzy a návrhu.

Dále rozlišujeme například spirálový model a RUP (Rational Unified Process) model. (Chapman, 2011)

3.3.2 Agilní vývojové metodiky

Vývoj agilních metodik započal v 90. letech, jako reakce na neefektivitu a absenci produktivity v tradičních metodách. Jak vyřešit problémy softwarového průmyslu se zaměřením na rychlejší dodávku a snížení nákladů na implementaci? Agilní vývojový přístup zkracuje dobu životního cyklu softwaru využitím iterativní implementace jednotlivých funkčností a průběžným testováním po celou dobu procesu vývoje. Agilní metody dále vznikají, aby řešili problém s nedostatečnou komunikací ze strany klienta v rané fázi projektu, zejména při definování požadavků na systém. Agilní metodika umožňuje efektivní reakci na uživatelské změny v kterékoli fázi projektu. Zaměřuje se primárně na koncového uživatele, a to v průběhu všech fází vývoje softwarového produktu. (Neelu, Kavitha, 2020)

Neméně důležité je zapojení zákazníka do „hry“. Pravidelné demonstrace po každé iteraci slouží k verifikaci, zda systém splňuje očekávání, či je třeba definovat a zapracovat změny. V tradičním vodopádovém modelu jsou zákazníci zapojeni pouze v raných fázích vývoje, později ztratí možnost na průběžnou zpětnou vazbu a implementaci potencionálních změn. Pokud by zákazník chtěl v tradičním modelu implementovat změnu, projekt by měl správně začít od začátku.



Obrázek 3 - Agilní vývojový přístup (Neelu, Kavitha, 2020)

Výhody agilních technik: zákazník nemusí čekat na konec projektu, okamžitá reakce na změny a přání zákazníka, zapojení zákazníka v průběhu celé implementace, průběžné testování a oprava chyb. (Neelu, Kavitha, 2020)

Za poslední dekádu vznikl nespočet agilních metodik pro vývoj softwaru. Každá je vhodná pro jiný typ projektu s ohledem na jeho specifické potřeby. Jak již bylo zmíněno výše, agilní přístup se zakládá na iterativních a evolučních technikách. Tyto techniky umožňují postupné zdokonalování procesů vývoje. Kvalita vývoje, spolupráce a komunikace s klientem by se měla postupně zlepšovat s každou iterací projektu. (Neelu, Kavitha, 2020)

Příklady agilních metodik:

- Extrémní programování – pouze pro malé týmy o velikosti max. 10 členů. Délka iterace 2 týdny.
- Scrum – délka iterace (sprintu) 1-6 týdnů. Na vývoj dohlíží tzv. SCRUM master.
- Crystal – barva krystalu určuje počet lidí na projektu. Vyznačuje se velkou autonomií týmu.

(Neelu, Kavitha, 2020)

4 Testování softwaru v rámci vývojového procesu

4.1 Definice testování softwaru

Spousta lidí s nedostatkem potřebné praxe v oblasti vývoje a testování softwaru si může pod pojmem testování představit pouhé klikání v aplikaci. Jiní ho považují za inženýrskou disciplínu vyžadující odborné znalosti, ale i praktické zkušenosti. (Bureš, Renda, Doležel aj., 2016)

Definice testování podle ISTQB (International Software Testing Qualifications Board): *„Pomocí testování je možné měřit kvalitu softwaru ve smyslu zjištěných defektů, a to pro funkcionální a také pro nefunkcionální softwarové požadavky a charakteristiky (např. spolehlivost, použitelnost, účinnost, udržovatelnost a přenositelnost). Testování může zvýšit důvěru v kvalitu softwaru, pokud najde malé množství defektů nebo žádné defekty. Vhodně navržený test, který úspěšně projde, snižuje celkovou úroveň rizika v systému. Pokud testování prokáže defekty, kvalita softwarového systému se zvýší, jsou-li tyto defekty opraveny.“* (ISTQB, 2011)

Testování není jen o vykonání samotného testu, zahrnuje také přípravné aktivity a aktivity po vykonání testů. Tyto činnosti se skládají z plánování, řízení, definování testovacích podmínek, identifikace testovacích případů, tvorby a exekuce testovacích skriptů, kontroly výsledků, vyhodnocení výstupních kritérií, reportování výsledků a dokončení finalizačních aktivit. Proces testování by měl také obsahovat statickou analýzu a revizi dokumentů. (ISTQB, 2011)

Cíle testování ve firmě a projektu:

- Nalezení defektů
- Aktivně předcházet defektům
- Zajištění vysoké jakosti vyvíjeného softwaru
- Nalezení bezpečnostních nedostatků
- Odhalení výkonnostních problémů

- Přinést potřebné informace pro rozhodování

4.2 Softwarový defekt

V praxi se občas zaměňují pojmy defekt, chyba, selhání a incident. Pro jednoznačné rozlišení jsou níže definovány.

- Defekt se na projektech častěji označuje jako tzv. bug. Je tak označován rozdíl mezi očekávaným a aktuálním chováním aplikace. (Bureš, Renda, Doležel aj., 2016)

- Chybu může udělat člověk v jakékoli činnosti. Může nabývat mnoha podob od pravopisných chyb až po chybu v analýze požadavků. (Bureš, Renda, Doležel aj., 2016)

- Selhání nastává v případě, kdy např. systémová komponenta přestává pracovat, může to být důsledek některého z defektů. Selhání může také vzniknout jako následek přírodních jevů a hardwarových problémů. (Bureš, Renda, Doležel aj., 2016)

- Incident je situace, která vyžaduje dodatečnou analýzu a případnou nápravu. Obvykle se vztahuje k produkčnímu prostředí. Pokud je událost zapříčiněna defektem, označuje se také jako problém. (Bureš, Renda, Doležel aj., 2016)

Defekty jsou v rámci testovacích aktivit běžné. Vývojáři pracují často s náročným zadáním v rámci komplikované infrastruktury, to vše v kombinaci s případným nedostatkem času a potřeby splnění všemožných funkčních a nefunkčních požadavků zvyšuje riziko zanesení bugu. Každý bug by měl být zaznamenán v některém z dostupných nástrojů pro procesní řízení. Důležité jsou detailní popis a správné označení defektu (severita, priorita, status, prostředí). Správně nastavené procesy reportingu a řízení oprav jsou nezbytné.

Defekty mohou mít spoustu příčin, některé jsou zmíněny níže:

- Nesprávná nebo neúplná specifikace požadavků klienta

- Zanesené požadavky do funkčního designu obsahují chybu

- Chyba ve funkčním designu způsobuje chybu v technickém designu

- Nepochopení technického designu a chybně naprogramovaný software
- Defekty v testovacích scénářích a skriptech
- Nedostačující hardwarová infrastruktura

(Bureš, Renda, Doležel aj., 2016)

4.3 Principy testování podle ISTQB

ISTQB rozlišuje 7 základních principů aplikovatelných na testování:

1. Včasné testování – umožňuje brzké nalezení defektů a garantuje dostatek času na opravu. Je třeba začít s identifikací a přípravou testovacích scénářů co nejdříve. Průběžně dodávat a testovat. Ideální je využití agilních metodik vývoje.

2. Testování ukazuje přítomnost defektů – důkladné a správně navržené testování pomáhá detekovat defekty. Na druhou stranu nikdy nezaručí, že v systému žádné defekty nezbyly, snižuje pouze pravděpodobnost objevení defektu na produkčním prostředí.

3. Vyčerpávající testování je nemožné – v žádném případě není možné otestovat všechny kombinace vstupů a výstupů. Důležité je zvolení ideálního cíle v souladu s rizikem.

4. Shlukování defektů – mohou být systémové oblasti s malou koncentrací defektů, ale také i oblasti obsahující velkou většinu defektů. Důležité je zaměřit cíle testování více na problémové oblasti.

5. Pesticidní paradox – čím více testovacích kol je uskutečněno, tím větší je pravděpodobnost, že daný test nenalezne nový defekt. Z tohoto důvodu je důležité stávající testy revidovat a upravovat. Nesmíme zapomenout tvořit nové odlišné testy.

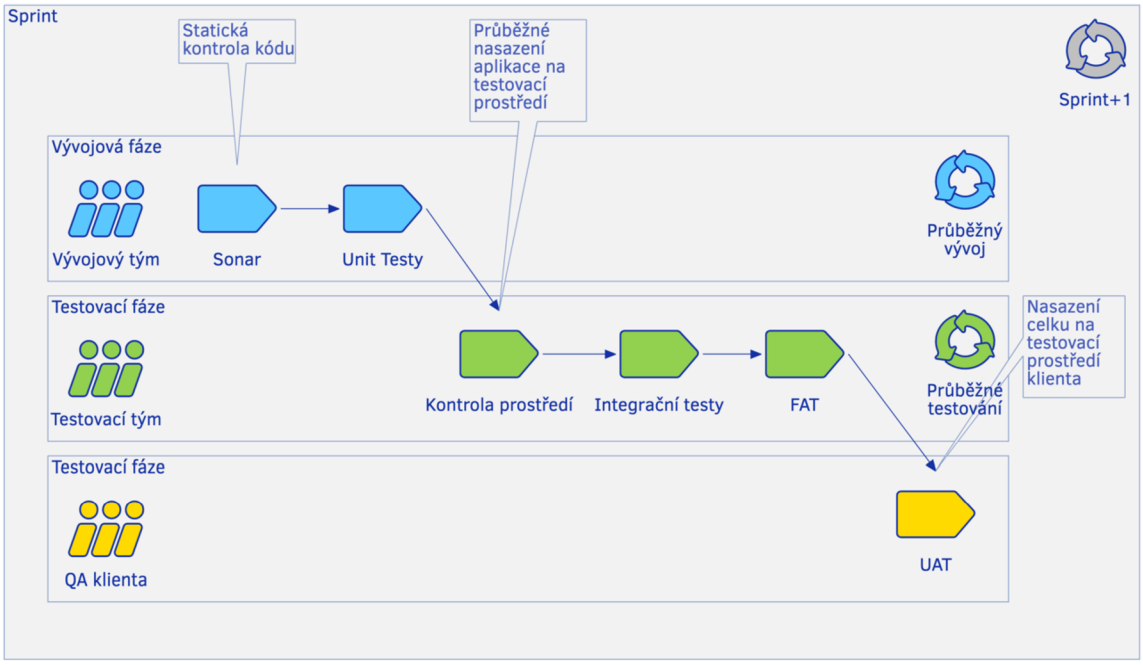
6. Testování je závislé na kontextu – každý projekt vyžaduje jiný přístup k testování. Například real time systémy v oblasti energetiky kladou velký důraz na výkon, bez odstávkový provoz a bezpečnost.

7. Falešná představa o neexistenci omylů – pokud je systém špatně navržený a nenaplňuje očekávání zákazníka, tak ani nalezení a oprava všech defektů nepomůže.

(ISTQB, 2011)

4.4 Kategorie testů

Testování můžeme kategorizovat podle více kritérií, kdo test vykonává, v jaké vývojové fázi se projekt nachází (dimenze času) nebo přímo podle účelu testu (dimenze typu). Testovací proces začíná již v rámci implementace a končí s dodávkou na produkční prostředí k zákazníkovi. Klienti stále častěji kladou důraz na tzv. „bug less“ systém obsahující co nejméně defektů. S ohledem na tento požadavek musí být každé kategorii přidělen dostatek prostoru a pozornosti. Na obrázku níže jsou znázorněny jednotlivé testovací aktivity se zaměřením na agilní vývoj softwaru.



Obrázek 4 - Životní cyklus testování v rámci agilního vývoje (Autor, UUBML, 2021)

4.4.1 Sonar

Statické testování kódu. Sada pravidel, která se pravidelně aplikuje po přidání nové implementace, za účelem varování před potenciálními problémy ve statickém kódu. V praxi bývá tento stupeň testování často podceňován. Včasné odhalení chyb v kódu

programátorem šetří čas testerům. Doporučuji se zaměřit na automatizaci tohoto typu testů a na pravidelnou exekuci.

4.4.2 Unit testy

Po samotném ověření kódu přicházejí na řadu tzv. jednotkové testy. Zaměřují se zpravidla na testování jednotlivých metod, tříd, objektů a modulů, které jsou samostatně testovatelné. Komponenta, na které jsou testy vykonávány, je zpravidla odtržena od zbytku systému, případně mohou být použity simulátory. (Hlava, 2011)

Podobně jako u testování kódu je za tento typ testů zodpovědný primárně samotný vývojář. Testy se tvoří ve formě programového kódu za pomoci různých frameworků, jakými jsou NUnit[6], Jasmine[7] apod. Čím větší je pokrytí kódu unit testy, tím větší je pravděpodobnost včasného odhalení kritických chyb v implementaci. Vývojář by si měl pravidelně jednotkové testy psát v rámci samotné implementace nových funkcionalit. Defekty se nezaznamenávají, bývají opraveny okamžitě při nálezu.

4.4.3 Integrační testy

Integrační testy jsou vykonávány po jednotkových testech a jejich cílem je ověřit integraci dílčích systémových komponent a jejich schopnost bezchybně spolupracovat v rámci aplikace. Testováním integrace se také rozumí ověření správného fungování komponenty v rámci celé infrastruktury, jde o předcházení potencionálním problémům komunikace s konkrétní verzí operačního systému, databáze nebo .Net frameworku. Dále umožňuje odhalit problémy s hardwarem. Za tuto oblast již plně zodpovídá testovací tým. Integrační testy mohu být vykonány manuálně, ale moderní způsob vývoje klade velký důraz na automatizaci. Integrační testování může kromě funkčních testů zahrnovat také verifikaci nefunkčních systémových požadavků, například výkonnostní testy. Před zahájením exekuce všech integračních testů doporučuji vykonat základní test prostředí a infrastruktury, který může obsahovat kontroly místa na disku, databázi a správné fungování všech služeb. Pro kontrolu prostředí je možné použít „smoke test“.

[6] https://nunit.org
[7] https://jasmine.github.io

4.4.4 FAT – Akceptační testy na straně dodavatele

FAT neboli „Factory acceptance test“ jsou akceptační testy prováděné týmem pro kontrolu kvality na straně dodavatele softwaru. Mohou být také označovány jako systémové testy. Přichází na řadu po úspěšném ověření integrace všech komponent informačního systému. Je to poslední stupeň testování před dodáním k zákazníkovi, proto je nutné aplikaci ověřovat i z perspektivy právě zmíněného klienta. Software je testován již jako celek podle předem definovaných a připravených testovacích scénářů. Každý testovací scénář by měl zastupovat reálnou situaci, která může v produkčním provozu nastat. Typickými testy bývají v tomto případě tzv. testy průchodu systémem, kdy se ověřuje kompletní funkcionalita od úvodního uložení dat, zpracování až po finální výstup.

Rozsah testování by měl být předem vymezen v rámci testovací strategie nebo plánu. V průběhu interních akceptačních testů je důležité, aby konfigurace testovacího prostředí byla co nejvíce podobná provoznímu prostředí, alespoň co se týče použití jednotlivých komponent. Snižuje se tím riziko odhalení chyby až v reálném provozu. Systémové testování využívá různých technik pro ověření funkcionálních i nefunkcionálních požadavků na systém. Konkrétními technikami se zabývá nadcházející kapitola. (ISTQB, 2011)

Všechny defekty musí být reportovány a detailně zaznamenány v systému. Po opravení vývojářem je bug, včetně propojeného testovacího případu, znovu přetestován a uzavřen. V rámci tradičních vývojových metodik bylo běžné systémové testy vykonávat opakovaně ve více kolech. Ukončení FAT bývá podmíněno splněním definovaných kritérií pro výstup. Například maximálním možným počtem nevyřešených chyb rozdělených podle severity bugu (0 A, 5 B, 10 C apod.).

4.4.5 UAT – Uživatelské akceptační testy

Podmínkou pro vstup do fáze uživatelských akceptačních testů (UAT) je splnění předchozích aktivit včetně aktualizace potřebných dokumentací. Stejně jako u FAT by měl být stanoven testovací plán, který definuje především záběr testů, reportování případných defektů a výstupní kritéria. Aplikace, pokud není stanoveno jinak, by měla být nasazena na testovací prostředí klienta.

UAT jsou v plné kompetenci zákazníka, případně koncových uživatelů. Cílem UAT je utužení jistoty, že všechny požadované funkcionální i nefunkcionální požadavky jsou splněny. Dílčím cílem je vyhodnocení připravenosti systému pro nasazení a používání v produkčním prostředí. Testy probíhají podle testovacích scénářů předem připravených spoluprácí dodavatele a zákazníka. Nalezené defekty musí být klientem reportovány a dodavatelem opraveny v předem dohodnutém termínu. Opravy jsou po úspěšném interním přetestování nasazeny na prostředí zákazníka a znovu verifikovány zadavatelem. (Hlava, 2011)

Dalším typem akceptačních testů může být tzv. testování v terénu neboli alfa a beta testování. Tento druh testování je především využíván vývojáři tzv. krabicového systému v případech, kdy chtějí získat zpětnou vazbu od budoucích uživatelů ještě před tím, než je software oficiálně uvolněn a poskytnut trhu. (ISTQB, 2011)

Alfa testování probíhá na testovacím prostředí vyvíjející organizace. Příkladem mohou být různé platformy pro davové testování neboli „crowd testing“. Nejznámější je portál „Testing birds“, který vznikl v roce 2011 s úmyslem podpořit společnosti při optimalizaci jejich digitálních produktů. Hlavní výhoda této inovativní metody spočívá v tom, že se může zapojit úplně každý, odborník i široká veřejnost. Společnost se zabývá i níže specifikovaným Beta testováním. V tomto případě to znamená zpřístupnění nespočtu kombinací uživatelských zařízení a operačních systémů, které nemusí vývojový tým zajišťovat z vlastních zdrojů. (Testbirds, 2020)

Jak již bylo naznačeno výše, hlavní rozdíl mezi alfa a beta testováním spočívá v prostředí, ve kterém se testuje. Beta testování využívá vlastní prostředí uživatelů, jakými mohou být mobilní telefony, počítače, chytré televize atd. Setkáváme se s vydáváním tzv. beta verzí, které si může nainstalovat vybraná skupina uživatelů do svých zařízení. Tento způsob využívají známé technologické společnosti, jakými jsou Apple a Microsoft ale i spousta dalších.

4.4.6 Smoke testy

V rámci testovacího cyklu se můžeme setkat také s pojmem „smoke test“. Jejich účelem je rychlé potvrzení, že nejdůležitější funkce programu fungují. Tento typ testů se nezabývá podrobnou verifikací, která je dále vykonávána v rámci integračních a systémových testů. Testy a jejich data by měly být jednoduché a rychle proveditelné s ohledem

na nalezení co největšího počtu potencionálních defektů a selhání. „Smoke“ testem může být načtení přihlašovací stránky, načtení dat a provolání jednotlivých služeb. (Herbold, Haar, 2020)

Z pohledu testovacího týmu je to důležitý kandidát pro automatizaci, jelikož se vykonávají opakovaně po každém nasazení aplikace. Z pravidla i několikrát denně. Automatizace může ve výsledku ušetřit testerům spoustu času a práce.

4.5 Testovací techniky

V současnosti existuje více než 200 publikovaných technik testování. Některé z nich se vzájemně překrývají nebo vylučují. Na druhou stranu se mohou vzájemně doplňovat nebo rozšiřovat, a tak zvyšovat pokrytí systému testy. (Hlava, 2017)

4.5.1 Statické techniky

Statické testovací techniky jsou na softwarových projektech exekvované, aniž by testovaná aplikace byla spuštěna nebo dokonce v některých případech existovala. Statické testování se typicky zaměřuje na kontrolu zdrojového kódu aplikace a dílčích dokumentů. V mnoha případech se testování kódu a dokumentace může zdát banální, ale opak je pravdou. Při testování specifikace v rámci raných fází projektu můžeme odhalit nejasnosti a chyby daleko před tím, než se tytéž chyby projeví v samotné implementaci. Šetříme tedy čas programátora. Podobné je to i s testováním zdrojového kódu, je výhodnější chybu odhalit a odstranit ještě před tím, než je aplikace spuštěna. Opět ušetříme drahocenný čas při následném vyhodnocování výsledků funkčních testů. Další výhodou je sjednocení kódu samotného a zpřehlednění pro budoucí práci. Benefitem statických technik je tedy především úspora času v dalších fázích projektu. (Everett, Mcleod, 2007)

Statické techniky by měly vždy předcházet těm dynamickým. Statické a dynamické techniky se navzájem doplňují, tzn. objevují různé typy defektů efektivněji. Častými defekty odhalenými statickou technikou mohou být různé odchylky od standardů, defekty v požadavcích a návrhu, nedostatečná udržovatelnost a nesprávná specifikace rozhraní. (Hlava, 2012)

4.5.1.1 Testování dokumentace

Podle Ondřeje Macka *„dokumentací rozumíme veškeré dokumenty spojené s realizací softwaru vzniklé v rámci projektu, ať už se jedná o manažerské výkazy nebo o uživatelskou či technickou dokumentaci softwaru“.* (Macek, 2016)

Účelem dokumentace je předání ucelené a smysluplné informace čtenáři. Proto by měl každý dokument naplňovat následující kritéria: úplnost, správnost, relevantnost, jednoznačnost, konzistence. Tato kritéria by měla být vodítkem jak pro návrh, tak i testování dokumentů. Z hlediska formální stránky nesmíme opomenout ani patřičnou stylistiku a gramatickou správnost. (Macek, 2016)

Metody testování dokumentace:

- Neformální revize – absence formálního procesu, nízké náklady, zpravidla kontrola kolegou.
- Walkthrough – setkání vedené autorem, například ve formě skupinové kontroly výstupů. Zpravidla více kontrolorů. Formální i neformální.
- Technická revize – více formální, stanovený postup a výstup revize.
- Inspekce – vedená moderátorem, ne autorem. Šance objevit velké množství defektů, ale za vyšší režijní náklady.

(Macek, 2016)

4.5.1.2 Statická analýza kódu

Jak již bylo zmíněno, cílem statické analýzy je objevit defekty přímo ve zdrojovém kódu. Na rozdíl od dynamického testování není kód spuštěn. Statická analýza umí odhalit defekty, které jsou těžko zjistitelné dynamickým testováním a naopak.

Kvalitní kód by se měl vyznačovat následujícími znaky: udržitelnost, rozšiřitelnost, výkonová optimálnost a testovatelnost. (Mcconnell, 2005)

Naopak špatně navrhnutý a napsaný kód se vyznačuje těmito znaky: nesrozumitelnost kódu, nesoulad s rozhraními podle specifikace, nevhodné použití globálních proměnných, nedosažitelný (mrtvý) kód, nevyužitelný kód, bezpečnostní nedostatky, porušení standardů programování. (Mcconnell, 2005)

Metody statické analýzy kódu:

- Revize kódu – připomíná neformální revizi dokumentu, kdy si kolegové navzájem revidují svůj kód. Upozorňují na defekty nebo se doptávají. (Cohen, 2013)
- Párové programování – zakládá se na metodice extrémního programování. Dva vývojáři pracují na stejném úkolu zároveň. Spolupracují a poskytují si okamžitou zpětnou vazbu. (Wells, 1997)
- Automatické testování kódu – například Sonar. Dále revizi kódu provádí moderní vývojová prostředí (IDE[8]).

4.5.2 Funkcionální testování

Funkční testy se zakládají na požadavcích, specifikacích a případech užití. Účelem je ověřit, že všechny naimplementované funkce se chovají správně a podle předem specifikovaných požadavků zákazníka. (Jorgensen, 2008)

Funkcionální testování se zabývá externím chováním softwaru neboli testováním černé skříňky. Zjednodušeně testují, co systém dělá, nikoli jak. Testy mohou být vykonávány téměř na všech úrovních: integrační, systémové i akceptační. Funkční testy oproti jiným technikám zpravidla objevují nejvíce defektů. Z tohoto důvodu jsou klíčové pro celý vývojový cyklus.

4.5.3 Nefunkcionální testování

Zakládá se na testování nefunkčních požadavků na systém, jakými mohou být výkon, škálovatelnost, udržitelnost, spolehlivost, dostupnost a bezpečnost. Zaměřuje se tedy především na to, jak systém pracuje. Většinou se jedná o měřitelné vlastnosti. (Hlava, 2012)

Zahrnuje techniky zátěžového testování, testování použitelnosti, testování udržitelnosti, stres testování, testování spolehlivosti, penetrační testování, testování přenositelnosti aj. Z velké části se opět jedná o testování černé skříňky. (ISTQB, 2011)

[8] Integrated Development Environment

4.5.4 Regresní testování a přetestování

Regresní testování představuje znovu otestování jednotlivých funkcí a vlastností softwaru. Využívá se běžně pro ověření, že v rámci vývoje nových funkcionalit do systému nebyla dotčena nebo jakkoli ovlivněna původní implementace. Také se provádí po opravě kritických chyb, které mohou ovlivnit více oblastí. Testování nových funkcionalit je posléze pokryto funkčními testy. Dále se používá v případech, kdy je nějakým způsobem zasažena infrastruktura, například aktualizací verze databáze apod. Kontrolujeme, že všechny funkce systému se chovají stejně jako před aktualizací. V případě migrace nebo zavedení nového subsystému se provádí kompletní regrese. S ohledem na časté opakování regresních testů je u větších projektů v současnosti kladen velký důraz na jejich automatizaci. Regresí se rozumí i přetestování výkonu aplikace. Regresní testy mohou být aplikovány na všech úrovních testování.

Typy regresních defektů:

- Lokální defekty – implementace zanáší nový defekt do modulu. Např. přidání nového políčka do formuláře s absencí potřebné validace.
- Vzdálené defekty – změna v jedné komponentě zapříčiní defekt v propojené komponentě. Např. změna rozhraní služby s absencí ošetření volání v rámci ostatních komponent.
- Odkryté defekty – komponenta obsahuje defekt v metodě, která se ve staré verzi nepoužívala. Nová verze s metodou již pracuje a defekt se projeví.

(Renda, 2016)

Nejčastější příčiny regresních defektů: špatná architektura kódu, nedostatečná analýza dopadů opravy, správa verzí (nezanesení opravy do všech verzí aplikace), nekompatibilita s verzemi externích knihoven, nečekaná aktualizace operačního systému, instalace verzí se nezdařila, neznalost a nezkušenost vývojáře. (Renda, 2016)

Přetestování lze označit také jako konfirmační testování. Termín se používá pro opakované testování případu užití, které původně skončilo chybou. V rámci prvotního testu je nalezen defekt, který je zaznamenán a posléze úspěšně odstraněn. Oprava se ověřuje pomocí konfirmačního testu.

4.5.5 Testování údržby

Zahrnuje testování po plánované aktualizaci nebo patchování operačního systému, databáze, či jiného využívaného frameworku. Aplikace se také testuje po migraci dat, změně platformy nebo migraci serverů při změně poskytovatele hostingových služeb. Testování údržby zahrnuje detailní plánování a definování funkčních i nefunkčních regresních testů. Definice dopadů plánovaných změn na systém se nazývá dopadová analýza. Pro hladký průběh testů je vhodné alokovat testery s dostatečnou znalostí prostředí. (ISTQB, 2011)

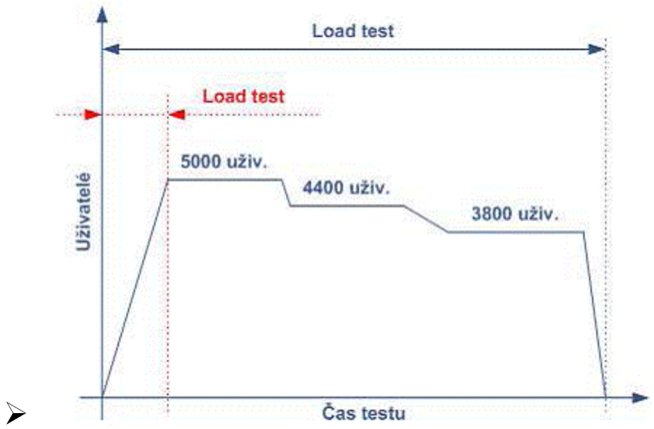
4.5.6 Výkonnostní testy

Performance testování se zabývá zkoumáním výkonnostních charakteristik systému za předpokládaných podmínek za účelem dalšího využití zjištěných informací. Výkonnostní testy mohou být milně označovány také jako zátěžové testy. Zátěžový test je však jen jedním z druhů výkonnostních testů.

Výkonnostní testy umožňují zjistit, jak se systém a jeho jednotlivé části budou chovat pod určitým zatížením. Ověřují, zdali jsou naplněny definované nefunkční požadavky na výkon. Odhadují, jak se systém bude chovat na cílové infrastruktuře.

Typy výkonnostních testů:

- Profilování systému – měří a analyzují se časy odezev běžícího systému, popřípadě časy specifických výpočtů. Hlavním cílem je objevení výkonnostně slabých míst v systému, tzv. úzkých hrdel.
- Zátěžový test – testování chování a výkonu systému pod předem definovanou zátěží. Zátěž může být generována například vysokým počtem současně pracujících uživatelů nebo vícenásobným spuštěním procesu. Zajímá nás nejen chování aplikace v rámci navštěvovaných případů užití, ale i výkon systému jako celku.



Obrázek 5 - Plán zátěžového testu v čase (Ježek, 2019)

- Konkurenceschopnost přístupů – aplikuje se převážně na databázové vrstvě. Typickými testy konkurenceschopnosti jsou pokusy o současný update stejného záznamu více uživateli nebo založení konkrétního uživatelského účtu několika administrátory ve stejnou chvíli.

- Test porovnáním – srovnání výkonností softwaru za odlišných podmínek (rozdílná hardwarová konfigurace, nastavení systému, jiná verze systému). V případě, že nemůžeme testy vykonat na cílovém prostředí, je možné využít jiné konfigurovatelné prostředí a postupně navyšovat, či odebírat hardwarové zdroje. Následným porovnáním výsledků jednotlivých testů získáme koeficient, na jehož základě jsme schopni zhruba odhadnout chování na cílovém prostředí.

- Test za nestandardních podmínek – testuje robustnost systému, například po nečekaném vypnutí jednoho z aplikačních serveru je nutné, aby ho zbylé zastoupily. Tímto způsobem se dá otestovat reakce systému na nečekané situace v rámci celé infrastruktury.

- Test chování systému při zpracování velkého objemu dat – při extrémním objemu zapisovaných a načítaných dat sledujeme reakci a chování systému.

(Ježek, 2019)

V případě potřeby vygenerování vysoké zátěže na systém lze jednotlivé typy testů kombinovat. Oblíbenou sestavou testů bývají také testy nestandardních podmínek v kombinaci se zátěžovými testy.

4.5.7 Penetrační testy

Penetrační testování se může definovat jako provedení kontrolovaného útoku na část softwaru nebo sítě za účelem vyhodnocení bezpečnosti. Penetrační test bývá rozdělen do více kroků. Konkrétní aktivity se mohou u jednotlivých útoků lišit, nicméně začátkem by mělo být shromáždění informací, jehož cílem je najít zranitelnost v síti, následovat by měl útok a proniknutí do systému místem, kde byla zranitelnost objevena. V případě úspěšného útoku je konkrétní sekvence akce nahlášena a použita při následné opravě. (Schwartz, Kurniawati, 2019)

Oblíbenými metodami penetračních testerů bývají:

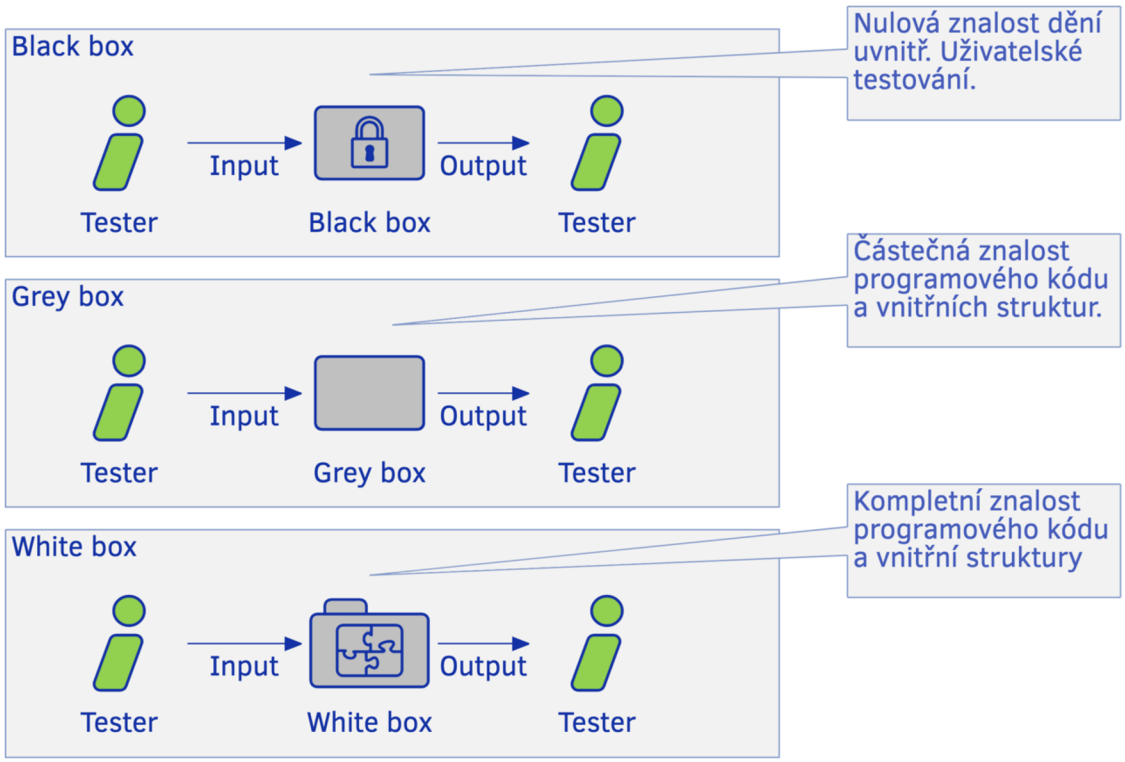
- IDOR[9] – hledání slabin například za pomocí změny proměnných v rámci URL, například ID, UID apod.
- XSS[10] – zkratka pro anglické spojení „cross-site scripting“. Útočník se snaží například přes formulář spustit vlastní škodlivý javascriptový kód. Využívá slabost na straně klienta.
- SQLi[11] – útok na databázovou vrstvu za pomocí škodlivého SQL skriptu.
- LFI[12] – pokus o nahrání a spuštění škodlivého souboru.
- Pokus o zachycené nezabezpečené komunikace. Například odchycení přihlašovacích údajů do internetového bankovnictví, kdy uživatel využívá nezabezpečenou veřejnou Wi-Fi síť v kavárně.
- RCE[13] – podobně jako XSS se pokouší spustit vlastní škodlivý kód. Zásadní rozdíl je v tom, že RCE využívá bezpečnostní chybu na straně serveru, tzn. že odeslaný kód bude spuštěn na straně serveru.

V rámci této kapitoly je vhodné zmínit „Open Web Application Security Project (OWASP), nezisková organizace, která se zabývá zlepšením softwarové bezpečnosti. (OWASP, 2021)

[9] Insecure Direct Object Reference
[10] Cross-Side Scripting
[11] SQL injection
[12] Local File Inclusion
[13] Remote Code Evaluation/Execution

4.5.8 Black box, white box a grey box

Testovací techniky black box, white box a grey box jsou definované hloubkou znalostí vnitřní programové struktury testovaného softwaru. Jednotlivé techniky a úroveň znalosti jsou znázorněné na následujícím obrázku.



Obrázek 6 - Černá, šedá a bílá skříňka (Autor, UUBML, 2021)

4.5.8.1 Technika black box

Technika black box známá také jako „closed box“ se vyznačuje tím, že tester nezná a nemá přístup ke zdrojovým kódům ani k žádné technické dokumentaci, kde by byly popsány. (Patton, 2002)

Metoda black box staví své testy na specifikacích a dokumentech. Můžeme si představit, že do černé skříňky uložíme vstup (např. data) a aniž bychom viděli nebo se více zajímali o procesy uvnitř systému, tak očekáváme konkrétní výstup, který na závěr vyhodnocujeme. Testování vychází z předem definovaného testovacího scénáře. Testovací scénář si připravuje tester svépomocí s ohledem na funkční specifikaci produktu. Správně napsaný scénář by měl obsahovat popis, cíl, detailní postup, očekávaný výstup a výsledek. Testy mohou být provedeny manuálně nebo automatizovaně.

Výhody této techniky se vyznačují:

- Snadností – není vyžadována znalost programovacích jazyků.
- Rychlostí – umožňuje otestovat rozsáhlé systémy v krátkém čase.
- Transparentností – test je srozumitelný i pro zákazníka.
- Testovací scénáře mohou být vytvořeny ihned po dokončení specifikace.
- Netřeba zpřístupňovat testerovi zdrojový kód.
- Testování je nezávislé na infrastruktuře použité technologii apod. Definice testu je stále stejná.

(Čermák, 2010)

Nevýhody jsou charakterizovány neefektivitou kódu a nežádoucím chováním. Správný výstup testu neznamená automaticky efektivní a požadovanou implementaci.

4.5.8.2 Technika grey box

Jak vyplývá z ilustrace, technika grey box, známá také jako „translucent box“, se nachází někde mezi testováním black a white box. Vyznačuje se průnikem obou sousedních technik. Tester disponuje částečnou znalostí vnitřních programových struktur, ale nejsou tak obsáhlé, aby byl schopen vykonávat testování white box.

Testování grey box je podobně jako black box vykonáváno z venku. Tato technika je využívána pro odhalování zranitelností aplikace a následný útok. White box technika identifikuje zranitelné místo a za pomocí black box je proveden útok. Jako další příklad si lze představit testování vícevrstvé aplikace. Tester má přístup do databáze a díky tomu může kontrolovat nejen samotný vstup a výstup, ale i uložení hodnoty do databázové tabulky. (Čermák, 2010)

Výhody této techniky se vyznačují:

- Kombinací přístupu obou sousedících technik.
- Neintrusivní – testování založeno na znalosti specifikace, rozhraní a architektuře. Přístup ke zdrojovému ani binárnímu kódu není potřeba.

- Inteligentní testy – tester je schopen vypracovat inteligentní testovací scénář zaměřený na práci s daty a komunikační protokoly.

(Čermák, 2010)

Nevýhody jsou podobné jako u techniky black box, neefektivita kódu a jeho neúplné otestování. (Čermák, 2010)

4.5.8.3 Technika white box – strukturální testování

Poslední technika je označována jako white box nebo také strukturální testování. Zaměřuje se na dění uvnitř bílé skříňky. Takové testy mohou být náročnější na přípravu, protože definování testů vychází převážně ze zkušeností a znalostí testovaného systému.

Testování vyžaduje důkladnou znalost vnitřních datových a programových struktur. Tester má k dispozici binární i zdrojový kód softwaru. Kód je třeba analyzovat a rozumět mu. Technika slouží k identifikaci nežádoucího kódu. Ostatní techniky se převážně zaměřují na plnění funkčních a nefunkčních požadavků. Žádná z nich již ale neověřuje, zdali softwarová aplikace nedělá ještě něco neočekávaného navíc. Nežádoucí kód může představovat bezpečností riziko a je třeba ho včas odhalit a odstranit. (Čermák, 2010)

Strukturální testování lze vykonávat v rámci všech fází testování. Při jednotkovém a integračním testování komponent mohou být použity technologie pro měření pokrytí kódu testy. Například pokrytí příkazů, metod a rozhodování. Míra pokrytí je vyjádřená v procentech a definuje důkladnost testování. Cílem je se pokusit co nejvíce přiblížit 100 %, které představují kompletní pokrytí.

Výhody této techniky se vyznačují:

- Včasným odhalováním chyb – vývojové defekty lze nalézt ještě před finální kompilací kódu.
- Odhalení nežádoucího kódu – klíčové pro bezpečnost aplikace.

Nevýhody představují nároky na odbornou znalost testovaného systému, programování a používaných nástrojů. Vysoké náklady na pořízení specializovaných nástrojů pro analýzu zdrojového kódu. (Čermák, 2010)

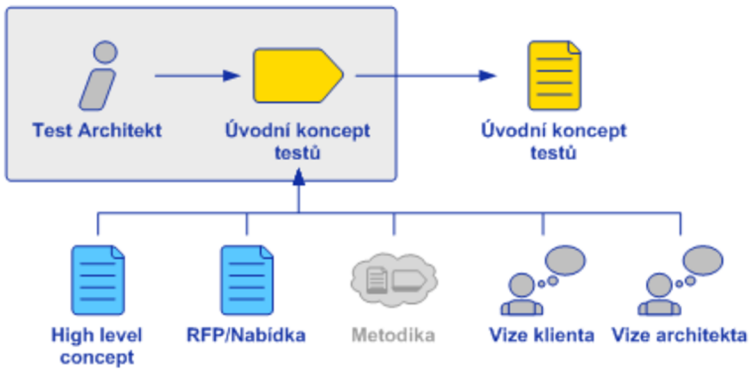
4.6 Plánování a exekuce testů

Vývoj testů může nabývat mnoha podob. Od velmi neformální s využitím základní dokumentace až po velmi podrobné a formální procesy. Stupeň formálnosti vyplívá z povahy projektu. Povaha se zakládá na vyspělosti testovacích a vývojových procesů, použité metodice vývoje, časových omezeních, bezpečnostních a regulatorních požadavcích a samozřejmě je to i o lidech. (ISTQB, 2011)

4.6.1 Úvodní koncept testů

Počáteční etapou v rámci procesu testování bývá úvodní studie, ze které plynule vychází úvodní koncept testování na plánovaném projektu. Za úvodní koncept je zpravidla zodpovědný test architekt a zakládá se na:

- Zadání informačního systému, či nabídce
- Vizi informačního systému
- Funkčních a nefunkčních požadavcích
- Zkušenostech – například na již realizovaných podobných projektech
- Již existujících scénářích a datech neboli testovacím repositáři



Obrázek 7 - Úvodní koncept testů (Unicorn Top Gun Academy, 2016)

Test architekt s ohledem na zmíněné vstupy a používané metodiky vytvoří dokument, který by měl obsahovat:

- Co se bude testovat neboli návrh testovacích požadavků
- Typy plánovaných testů – Integrační, FAT, UAT (funkční i nefunkční)

- Požadavky a návrh testovacího prostředí
- Testovací nástroje: licence, přístupy apod.
- Definování testovacího týmu
- Harmonogram a milníky
- Rozpočet

(Hlava, 2016)

Dokument musí být odsouhlasen projektovým řízením a je vstupem do dalších fází přípravy a exekuce testů. Tvoří vstup pro testovací strategii nebo „testing approach".

4.6.2 Testovací strategie, testing approach

Testovací strategie se používá převážně ve velkých korporacích a na projektech zakládajících se na vodopádovém modelu. Agilní metodiky používají pro tento typ dokumentu termín „testing approach" neboli přistup k testům. Hlavní rozdíl spočívá v tom, že přístup k testům je méně formální. Nelze plánovat, jelikož nemáme klasická testovací kola a zároveň neznáme přesné termíny, kdy se jaká funkcionalita dodá. Každý projekt si proto „testing approach" přizpůsobuje svým specifickým potřebám.

Klasická testovací strategie je formována v rámci technologického projektu a využívá vstupy z úvodního konceptu testování. Dokument formuje hlavní metodiky testování na projektu. Testovací strategie odpovídá na následující otázky:

- Co testujeme? Cíl a obsah testů.
- Jak se bude testovat? Manuálně, automatizovaně, plánované fáze (FAT, UAT).
- Kdo bude testovat? Automat, interní nebo externí tým, odhady.
- Kde budeme testovat? Kolik prostředí je třeba připravit a s jakou konfigurací.
- Jaké jsou očekávané výstupy? Reporting a správa defektů.

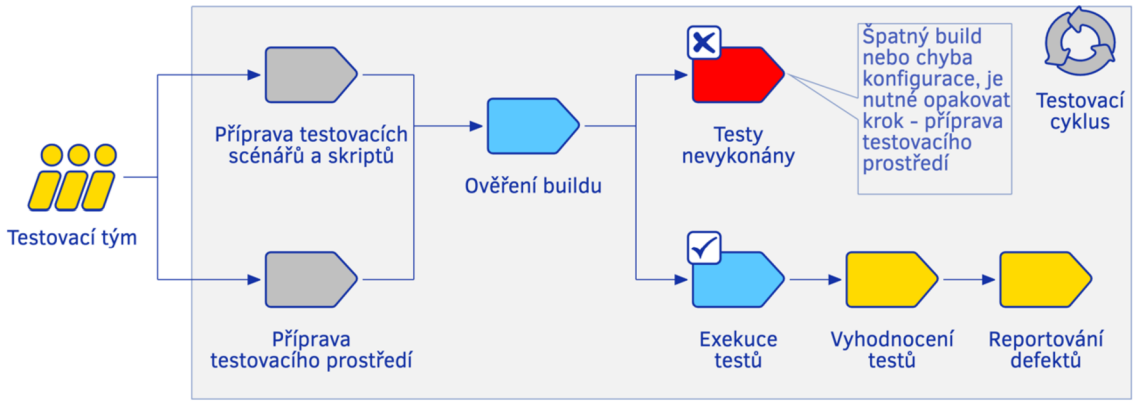
(Hlava, 2016)

Testovací strategie by měla být rozmanitá, zaměřená na rizika, specifická pro projekt zákazníka, praktická a hlavně obhajitelná. Projekt a jeho vlastnosti se můžou v průběhu času měnit, proto je nutné testovací strategii pravidelně aktualizovat. Vždy by měla reflektovat aktuální podmínky projektu.

4.6.3 Životní cyklus testování

Proces naplánování jednotlivých fází testování je důležité vhodně rozvrhnout s ohledem na potřebné a dostupné zdroje. Požadavky na potřebné zdroje je důležité podat s dostatečným předstihem.

Životní cyklus testovací fáze při agilním vývoji může vypadat následovně.



Obrázek 8 - Testovací cyklus (Autor, UUBML, 2021)

1. Testovací tým provede s ohledem na výstupy sprintu test identifikaci a připraví si testovací scénáře, skripty a data.
2. Vývojový tým naimplementuje požadovanou funkcionalitu.
3. Testovací tým nasadí nejnovější verzi komponent na testovací prostředí.
4. Exekuce základního smoke testu pro ověření stability verze a kondice prostředí.
5. Nalezený defekt v rámci ověření. Je třeba defekt odstranit a znovu připravit testovací prostředí.
6. Exekuce testů.
7. Vyhodnocení výstupů testů.
8. Reportování defektů a stavu.

Předpokládáme, že krok číslo jedna se provede na začátku sprintu. Celý postup od kroku dva se může v závislosti na velikosti dodávky během sprintu několikrát opakovat.

4.7 Nástroje pro podporu testování a správu defektů

Nahlášení a řízení defektů bývá často podceňováno. Opomenutí mohou vést k celé řadě komplikací. Špatně zadaný nebo pochopený defekt může vyústit až ke zpoždění celé dodávky systému. V rámci defektu by se měl opravovat pouze validní defekt, je důležité rozlišovat mezi bugy a změnovými požadavky, které nemají podporu v dokumentaci. Pokud by se změnové požadavky apod. neodlišovaly, vyústilo by to neplánovaným zvýšením nákladů. Nedůsledné reportování bugů (např. pomocí elektronické komunikace nebo mezi řečí v kanceláři) může zapříčinit zapomenutí a opětovný nález defektu až v produkčním prostředí. Je doporučováno využívat speciální systémy pro správu defektů. Níže jsou stručně představeny 3 nejznámější nástroje pro podporu testovacího managementu a správy defektů.

Klíčové funkce nástrojů pro podporu testovacího managementu:

- Správa a životní cyklus defektů
- Správa testovacích scénářů a případů
- Správa a monitoring testovacích kol nebo fází
- Tvorba, analýza a sdílení reportů
- Správa funkčních a nefunkčních požadavků a jejich pokrytí testy
- Tracebility matrix
- Cloudové řešení – přehled o stavu testů v reálném čase, spolupráce s dalšími manažerskými nástroji, integrace s automatizovanými testy
- Integrace s CI/CD [14]nástroji
- Umožňují snadný export výsledků

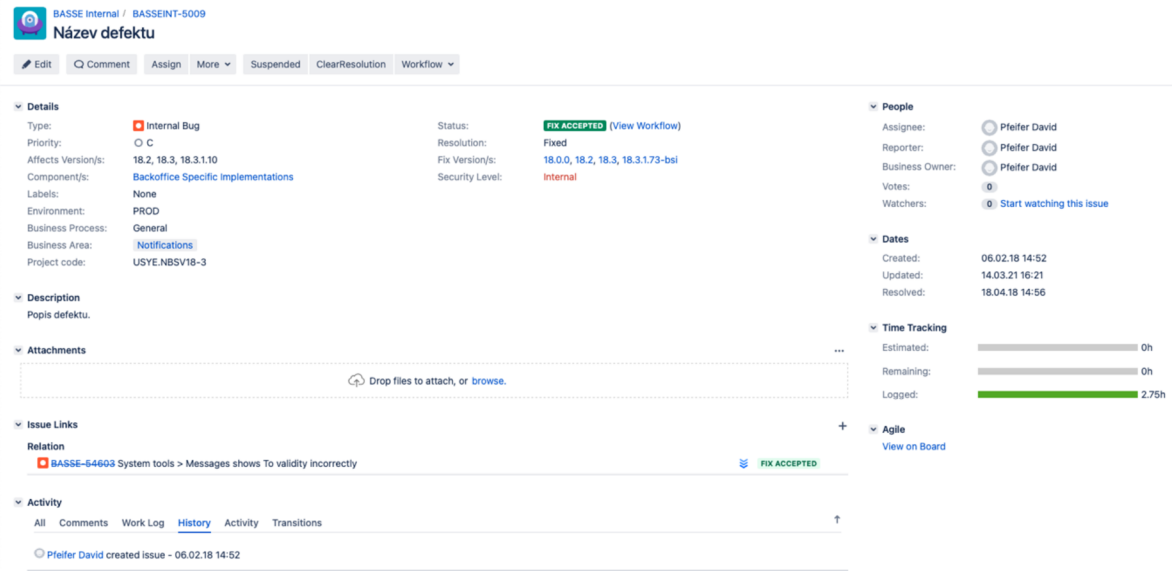
[14] Continuous Integration / Continuous Development

- Přiřazení jednotlivých testovacích případů testerům

4.7.1 JIRA Service Desk

Nástroj sloužící ke správě defektů od společnosti Atlassian[15]. Jednoduché a intuitivní zakládání defektu. Podpora různých typů tiketů, úkolů apod. Přizpůsobitelný životní cyklus. Umožňuje agilním týmům plánovat a zakládat uživatelské příběhy pro jednotlivé sprinty. To vše řízeno pomocí kanban a scrum nástěnek. Další funkce:

- Agilní reportování
- Jednoduché přesouvání úkolů
- Široká možnost vlastní konfigurace projektu
- Správa verzí jednotlivých systémových komponent



Obrázek 9 – Příklad defektu (JIRA, 2021)

Jak vyplývá z obrázku výše, správně zadaný defekt by měl obsahovat název, popis, verzi systému a prostředí na kterém byl nalezen, komponentu, k jaké se vztahuje, oblast, do jaké defekt spadá, verzi, ve které bude nebo byl defekt opravený atd. Dále je možné nahrávat různé přílohy, jakými mohou být obrázky, videa nebo logy z konkrétního serveru. Pravá část jednoznačně označuje zadavatele, osobu přiřazenou k opravě a osobu

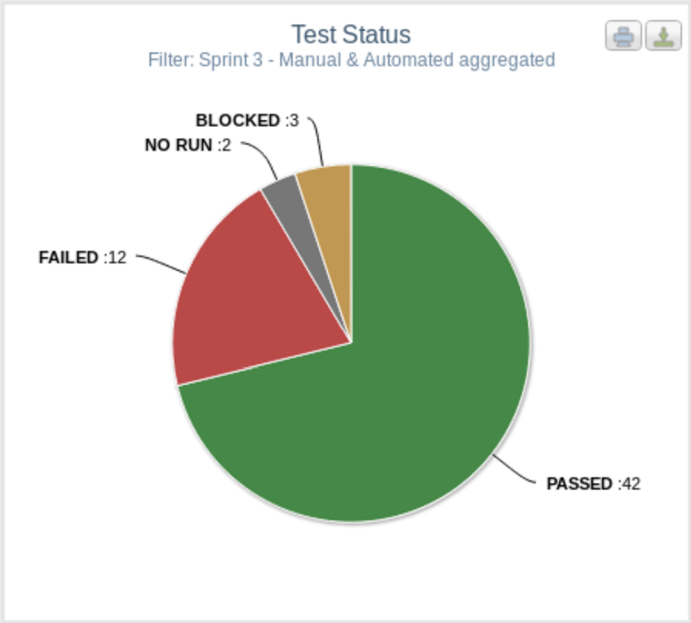
[15] https://www.atlassian.com/software/jira

zodpovědnou za konkrétní oblast systému. Níže si můžeme všimnout další velké výhodou, kterou je možnost sledování a vyhodnocení, kolik času oprava zabrala.

4.7.2 Practi Test

Practi Test[16] je nástroj pro podporu testování za přijatelnou cenu, vhodný pro menší i větší projekty. Nabízí jednoduchý a srozumitelný přehled o aktuálním stavu testování na projektu. Umožňuje propojení defektů s testovacími případy, na základě kterých byl objeven. Nabízí přehled pokrytí funkčních a nefunkčních požadavků testovacími scénáři. Skvělý je také z pohledu automatizace, jelikož nabízí API[17] pro integraci s nástroji umožňující exekuci automatizovaných testů. Dále nabízí:

- Oboustrannou integraci s JIRA.
- Kanban
- Kvalitní uživatelská podpora
- Hlasování uživatelů o přidání nových funkcí
- Vše uložené v cloudu a přístupné odkudkoliv



Obrázek 10 - Ukázka stavu testů (Practitest, 2021)

[16] https://www.practitest.com
[17] Application Programming Interface

4.7.3 QTest

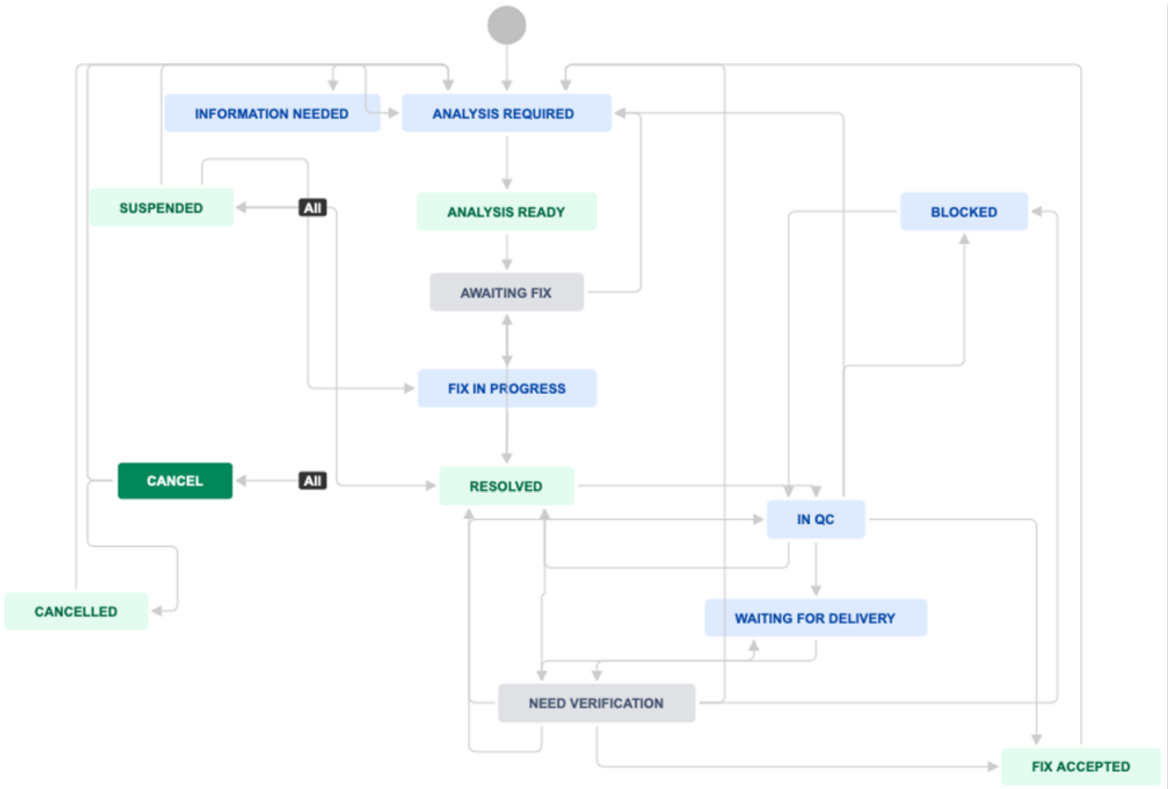
Z pohledu bonitních a velkých firem, pro které není rozhodujícím kritériem cena licencí, ale kvalita nabízeného nástroje, je ideální volbou. Za jeho levnější alternativu bývá označován výše zmíněný Practi Test. Podporuje již zmíněné funkcionality a spoustu dalších navíc.

4.7.4 Životní cyklus defektu

Životní cyklus bývá často označován anglickým termínem „*workflow*“. Defekt může v rámci svého životního cyklu projít spoustou stavů. Systémy pro podporu testování umožňují filtrování defektů podle stavů, ve kterém se právě nachází. Čím více stavů projekt používá, tím lépe se dá aktuální stav vysvětlit, nicméně to zvyšuje celkové nároky na správu defektů. Stavy je vhodné pojmenovat podle zvyklostí a specifik daného projektu nebo firmy. (Grössl, 2016)

Níže je uveden příklad konkrétního životního cyklu defektu na projektu Nordic Balance Settlement realizovaného firmou Unicorn[18]. S ohledem na specifické požadavky projektu je workflow modifikované a obsahuje více stavů, než bývá na jiných projektech zvykem.

[18] https://unicorn.com



Obrázek 11 - Životní cyklus defektu (Unicorn, 2021)

Za posunutí do dalšího stavu je vždy zodpovědný člověk, kterému je tiket přiřazen. Tabulka níže definuje a popisuje všechny možné stavy a situace vyplívající ze zobrazeného životního cyklu. Z každého stavu je možné se přesunout do některého z předchozích nebo následujících stavů.

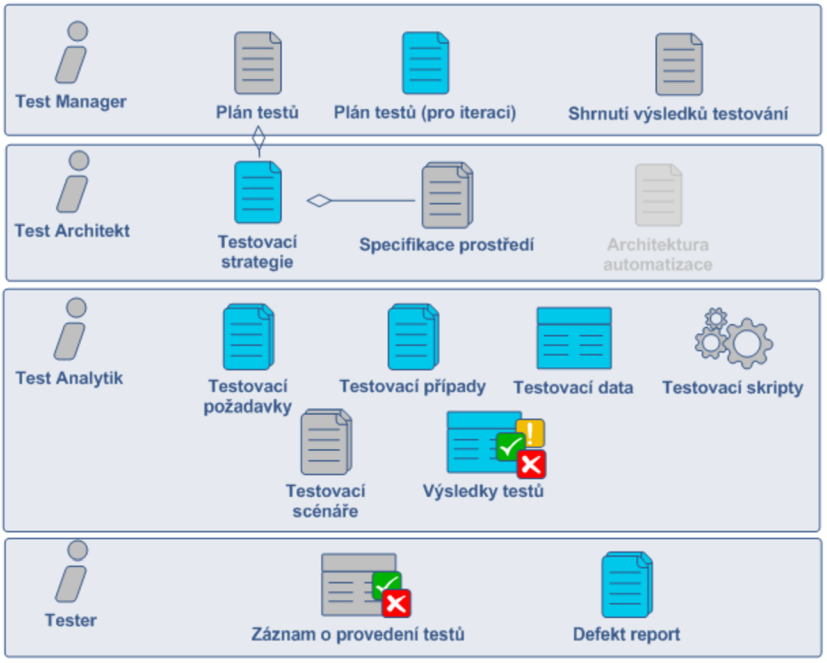
Tabulka 1 - Životní cyklus defektu (Unicorn, 2021)

Původní stav	Definice stavu	Nový stav
Require analysis	Úvodní stav	> Need info > Analysis ready > Cancel
Analysis ready	Defekt čeká na potvrzení analytikem	> Awaiting fix > Cancel
Need info	Nedostatečné zadání defektu – čeká na doplnění	> Require analysis > Cancel
Awaiting fix	Defekt přiřazen k opravě	> Fix in progress > Resolved > Cancel
Fix in progress	Defekt je právě opravován	> Resolved > Awaiting fix > Cancel
Suspended	Oprava odložena	> Require analysis > Fix in progress > Cancel
Resolved	Defekt je opraven	> Need verification > In QC

		> Cancel
Blocked	Opravu blokuje jiný propojený defekt	> In QC > Cancel
In QC	Oprava nasazena na testovací prostředí a čeká na přetestování testerem	> Awaiting Delivery > Resolved > Blocked > Require analysis > Fix Accepted > Cancel
Awaiting Delivery	Přetestování bylo úspěšné, oprava čeká na nasazení na prostředí zákazníka.	> Need verification > Cancel
Need verification	Oprava nasazena a připravena k finálnímu ověření uživatelem	> Fix Accepted > Awaiting Delivery > Require analysis > Cancel
Fix Accepted	Oprava akceptována	> Require analysis > Cancel
Cancel	Stav použit v případě, že se jedná o chybu testera / uživatele při testování	> Require analysis > Cancelled
Cancelled	Potvrzení uživatelem, že udělal chybu v rámci testu.	> Require analysis

4.8 Role a zodpovědnosti testera ve vývojovém týmu

Důležitost testování byla vysvětlena již na začátku čtvrté kapitoly. Tato podkapitola se zabývá rolí testera a z ní plynoucích zodpovědností. Testovací tým může být složený z testerů, kteří se dají rozdělit do čtyř základních kategorií podle jejich zkušeností a užšího zaměření. Na obrázku níže jsou znázorněny jednotlivé role, včetně aktivit, za které jsou v rámci představeného testovacího procesu zodpovědné.



Obrázek 12 - Role a zodpovědnosti testera (Hlava, 2017)

4.8.1 Manažer testování

Role manažera, případně vedoucího testování, vychází ze základních manažerských činností vztažených k testování: plánovat, organizovat, přikazovat, koordinovat a kontrolovat. (Vodáček, Vodáčková, 2013)

Povinnosti a úloha této pozice jsou v kontextu projektu nebo organizace odlišné. Ideálně by měl zaštiťovat: plánování testů, reportování výsledků nadřízeným, řízení dílčích činností v rámci testovací fáze, získávání zdrojů a řízení týmu testerů.

4.8.2 Test architekt

Tato role nemusí být striktně na projektu obsazena, případně může funkci vykonávat test manažer nebo zkušený tester.

Test architekt by měl primárně definovat testovací strategii nebo přístup k testům. Rozhoduje o tom, jaké oblasti by měly být automatizovány, v jaké míře a jakým způsobem. Specifikuje konkrétní nástroje pro automatizaci. V neposlední řadě stanovuje požadavky na testovací prostředí.

4.8.3 Test analytik

Test analytik se podílí na sumarizaci a dekompozici testovacích požadavků. Navrhuje a tvoří testovací scénáře a případy. Vymýšlí a připravuje testovací data pro testování. Tuto roli zastává zpravidla zkušenější tester, který má hlubší vhled do businessu zákazníka. Poskytuje výsledky testů svému manažerovi. Další důležitou činností test analytika může být aktualizace uživatelských návodů a dokumentací.

4.8.4 Tester

Typickými úkoly testera mohou být: exekuce testů, záznam o provedení testů, reportování defektů a přetestování opravených chyb. V závislosti na senioritě a zaměření se tester může věnovat dále automatizaci testů nebo zmíněné test analýze.

5 Automatizované testování softwaru

Automatizace je pojem moderní doby. Velká většina organizací se snaží šetřit výrobní náklady, automatizovat rutinní kroky a přesouvat lidské zdroje k jiným činnostem. Ne jinak je to i v oblasti testování softwaru. Klade se důraz na co nejrychlejší vykonání testů v potřebném rozsahu a s tím roste poptávka a obliba nástrojů pro automatizaci.

Testování softwaru je jednou z nejdůležitějších částí vývojového procesu. Agilní metodiky kladou důraz na souběžné testování s vývojem tak, aby defekty byly odhaleny a opraveny co nejdříve. Automatizace testů pomáhá plnit tyto cíle a současně zlepšuje kvalitu a spolehlivost dodávaného produktu. Mnoho společností se milně domnívá, že automatizace kompletně nahrazuje manuální testování. Toto bohužel současná vyspělost zatím neumožňuje. Vždy se objeví testy, které je třeba vykonat manuálně. (Rendón, Barrera, 2017)

Teorie uvádí, že od určitého počtu opakování je výhodné daný test automatizovat. Motivace pro automatizaci je prostá, finální náklady vynaložené na exekuci automatizovaného testu jsou po několika opakováních nižší než na provedení manuálního. Počáteční náklady na automatizaci jsou sice vyšší, ale to se vyrovnává s časem stráveným opakováním manuálního testu. Na skutečném projektu je nicméně realita o něco složitější. Při plánování, jaké testy automatizovat musíme také počítat s náklady na jejich správu a údržbu. Testovaný systém se v rámci vývojového cyklu mění, po úpravě funkcionality je třeba aktualizovat i automatizovaný test, jinak test ztrácí smysl a hodnotu. (Bureš, 2016)

Je podstatné brát v potaz i požadavky na kvalifikaci testera vykonávajícího manuální testy v porovnání s vývojářem automatizovaných testů. Část dovedností můžou mít společných, například znalost základních principů testování. Vývojář automatizovaných testů musí navíc disponovat znalostí programovacích a skriptovacích jazyků pro tvorbu. Dále musí mít přehled o používaných nástrojích a mít schopnost je aktivně využívat. Člověk disponující potřebnými znalostmi bývá pro projekt zpravidla mnohonásobně dražší.

Automatizované testy nepřináší jen finanční úsporu. Za další charakteristické znaky a výhody lze považovat:

- Přesnost – člověk není nikdy 100 % a může defekt přehlédnout, případně udělat chybu nebo odchylku v testu. Automat test vykonává pokaždé se stejnou přesností.

- Efektivita – automatizované testy efektivně šetří čas. Tester se tak místo vykonávání regresních testů může věnovat jiným potřebným aktivitám, např. test identifikací, či automatizací nových testů a s ní další zvýšené pokrytí.

- Rychlost – můžeme si představit požadavek, že je třeba okamžitě vykonat smoke test po nasazení prostředí. Manuálně by to trvalo třem testerům hodinu, automat stejné testy vykoná za pár minut. Analogicky šetří čas při regresních testech.

- Neúnavnost – automat vykonává testy pokaždé se stejným nasazením. V případě potřeby lze nastavit spuštění na pravidelné denní bázi nebo kdykoli je třeba, je dostupný 24/7 a nikdy nemá dovolenou.

Za nevýhodu automatizovaných testů lze považovat výše zmíněnou nutnost údržby. V případě dlouhodobého zanedbání údržby budou testy nepoužitelné. Z tohoto důvodu doporučuji testy aktualizovat souběžně se změnami v systému. Další nevýhoda spočívá v omezené možnosti analýzy chybného chování systému. (Bureš, 2016) Automatizovaný test má jasně definovaný výsledek a způsob, jak takový výsledek vyhodnocuje. Splnění cíle testu nám vždy nezaručuje, že se v průběhu nevyskytla chyba. Test velkou většinu defektů v rámci svého běhu odhalí, nicméně si již nemusí všimnout například špatného rozvržení elementů na stránce. I když i pro takové případy existuje speciální typ automatizovaných testů, který porovnává snímky obrazovky.

5.1 Definice automatizovaného testu

Automatizovaný test je takový test, který vykonává program bez asistence člověka.

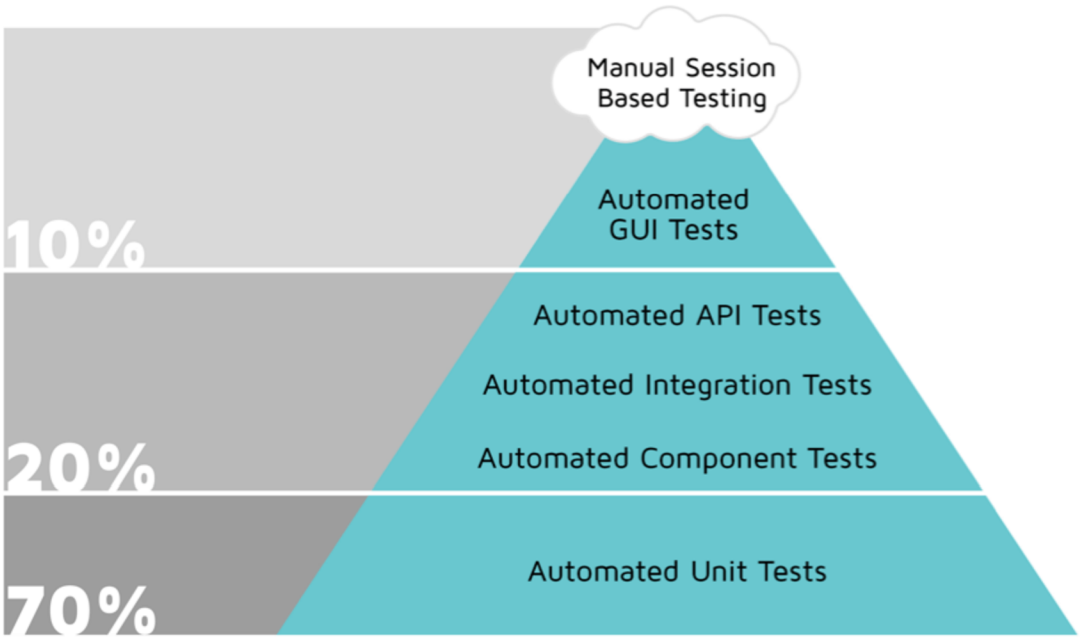
Podle Miroslava Bureše *„automat, na rozdíl od člověka, vykonává test zdarma a relativně rychle. Automatizovaný test je tím pádem možné opakovat mnohem častěji“.* (Bureš, 2016)

Automatizované testování je způsob testování softwaru, který využívá speciální softwarové nástroje k řízení a exekuci testů. Získané výsledky posléze porovnává s předpokládanými nebo očekávanými výsledky. To vše se provádí automaticky s malým nebo žádným zásahem testovacího inženýra. Automatizace je vhodná kdekoliv, kde je to výhodnější v porovnání s manuálními testy. (Techopedia, 2021)

5.2 Kategorie automatizovaných testů

Kategorie automatizovaných testů se od těch manuálních neliší. Zásadní rozdíl přichází s tím, kdo nebo co testy vykovává. V tomto případě testy vykonávají specializované nástroje pro automatizaci.

V této kapitole bych rád přiblížil, jaký přístup je dle mého názoru vhodný pro zamýšlenou automatizaci. Poměr automatizovaných testů by měl v ideálním případě vycházet z tzv. testovací pyramidy, kterou můžeme vidět na následujícím obrázku.



Obrázek 13 - Testovací pyramida (BlazeMeter, 2019)

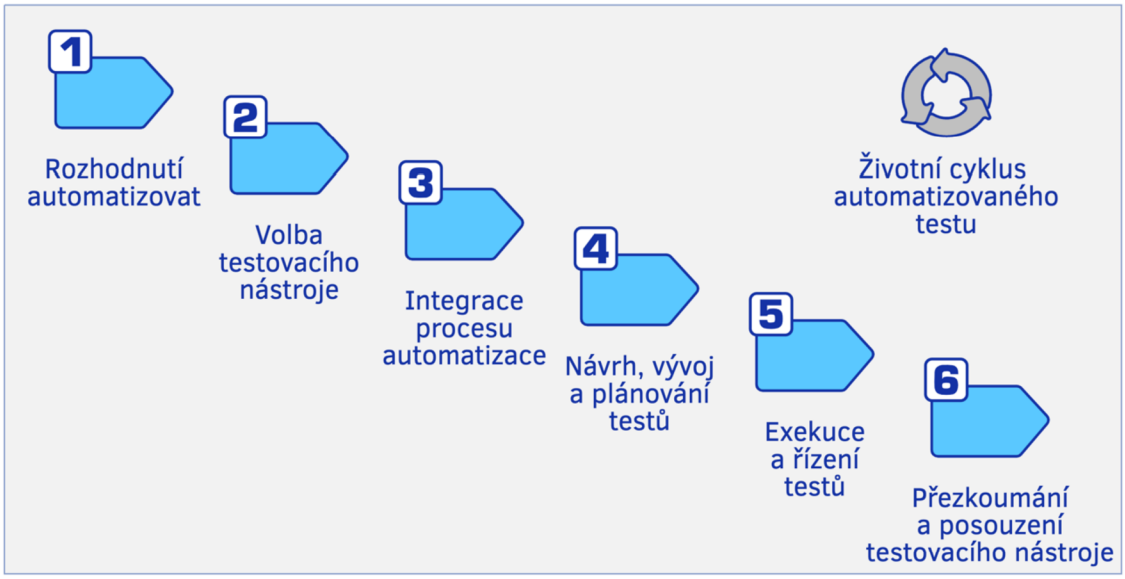
Základ pyramidy tvoří automatizované jednotkové testy komponent, které si tvoří vývojář sám v rámci implementace. Jejich příprava je efektivní a levná. Měly by tvořit až 70 % všech testů. Účelem je včasné odhalení a odstranění chyb. Ještě před tím, než je aplikace předána testerům.

Následují automatizované API, integrační a komponentové testy. Slouží k ověření klíčových funkcionalit, komunikačních rozhraní a správné integrace komponent. Takových testů by mělo být zhruba 20 %. Bývají používány k rychlému ověření stability verze. Mohou být vykonávány také v rámci smoke testu.

Zbylých 10 % by měly tvořit funkční testy využívající GUI[19] neboli grafické uživatelské rozhraní. Založeno na testování z pohledu koncového uživatele. Jejich příprava zabírá více času a vykonání trvá zpravidla déle než u předchozích dvou kategorií. Náklady na údržbu takových testů bývají také vyšší. Nicméně jsou klíčovým faktorem pro dodání kvalitního softwaru. GUI testy se používají zpravidla pro automatizaci regresních testů a smoke testů. Mohou být využívány v rámci FAT i UAT testovacích fází. Šetří testerům nejvíce času.

5.3 Návrh a vývoj automatizovaného testu

Návrh a vývoj automatizovaných testů se dá shrnout do šesti fází. Pro úspěšné zajištění implementace je důležité před vstupem do nadcházející fáze vždy ukončit fázi předchozí. (Patton, 2002)



Obrázek 14 - Životní cyklus návrhu automatizace (Autor, UUBML, 2021)

1. Před definitivním rozhodnutím, co automatizovat, je důležité provézt důkladnou analýzu softwaru, který má být testován. Projít všechny funkční i nefunkční

[19] Graphical User Interface

požadavky a specifika daného projektu a snažit se jim porozumět. Při tvorbě GUI testů je vhodné projít existující testovací scénáře a identifikovat vhodné kandidáty pro automatizaci. Při procházení testů je vhodné brát v potaz, jestli má daný scénář smysl vykonávat opakovaně nebo zdali se jedná o regresní test.

2. Po identifikaci vhodné oblasti přichází na řadu výběr nástroje. Předchozí analýza a pochopení specifik projektu pomáhá také s výběrem testovacího nástroje. Ne každý testovací nástroj je vhodný pro konkrétní projekt a účel testu. Doporučuji vykonat případovou studii za účelem výběru vhodného nástroje.

3. Po vybrání nástroje je třeba připravit testovací prostředí pro potřeby spouštění testů. Je vytvořen úvodní testovací skript, který se na takovém prostředí vykoná. Po úspěšném testu konfigurace následuje samotný vývoj testů.

4. Všechny definované testy jsou za pomoci vybrané technologie vytvořeny. Z mého pohledu je důležité, aby se test spoléhal pouze na základní testovací data, u kterých čekáme stálost. Zbylá data by si měl test připravovat svépomocí. Pomáhá to k znuvupoužitelnosti testů. Následuje testovací spuštění, analýza a ladění testů, které skončili chybou. Po odladění je možné testy začít používat.

5. Vytvořená sada testů je použita například v rámci regresních testů.

6. Na závěr by měl testovací architekt nebo manažer testování zhodnotit, kolik času zabralo vykonání automatických testů v porovnání s manuálním testováním a zdali není vhodné pokračovat v automatizaci dalších oblastí. Do hodnocení by měla být zahrnuta i stabilita nástroje a technická náročnost používání. Dále může být posouzeno, jestli nevyužít některý z nástrojů CI/CD, pro snazší exekuci nebo automatizaci reportování.

Nejčastější chyby při vývoji automatizovaných testů:

- Upřednostňování automatizace GUI před jednotkovými a integračními testy. Důvodem k takovému rozhodnutí bývá mylná představa managementu o vytvoření identických testů s těmi manuálními. Na jednu stranu správné uvažování o viditelné úspoře času v rámci exekuce manuálních testovacích scénářů. Na tu

druhou včasné odhalení defektů testy nižší úrovně šetří čas všem a zvyšují kvalitu dodávaných verzí k testům. (Kitner, 2021)

- Za každou cenu se snažit vymyslet vlastní komplexní testovací framework – psaní vlastního frameworku zabírá spoustu času, který by se dal věnovat tvorbě testů samotných. Proto je lepší nejdříve vytvořit testy a v návaznosti na to vylepšovat framework samotný. (Kitner, 2021)

- Podcenění času, který je k automatizaci potřeba. Tvorba automatizovaných testů je postupný proces. V rámci každého sprintu by měl být automatizaci věnován dostatek času, ať už se jedná o tvorbu nových testů nebo údržbu. (Kitner, 2021)

5.4 Nástroje využívané pro automatizaci

5.4.1 Selenium

Selenium[20] bylo vytvořeno v roce 2004. Název selenium je založeno na žertu jeho zakladatele Jasona Hugginse. Během jeho vývoje byl populární jiný nástroj pro automatizaci vytvořený společností Mercury Interactive. V tuto chvíli je vhodné uvézt české překlady slov „Selenium“, znamená selen a „Mercury“ znamená rtuť. S ohledem na to, že selen je známým protijedem na otravu rtutí, navrhl Jack, aby nástroj dostal právě toto jméno. (Guru99, 2021)

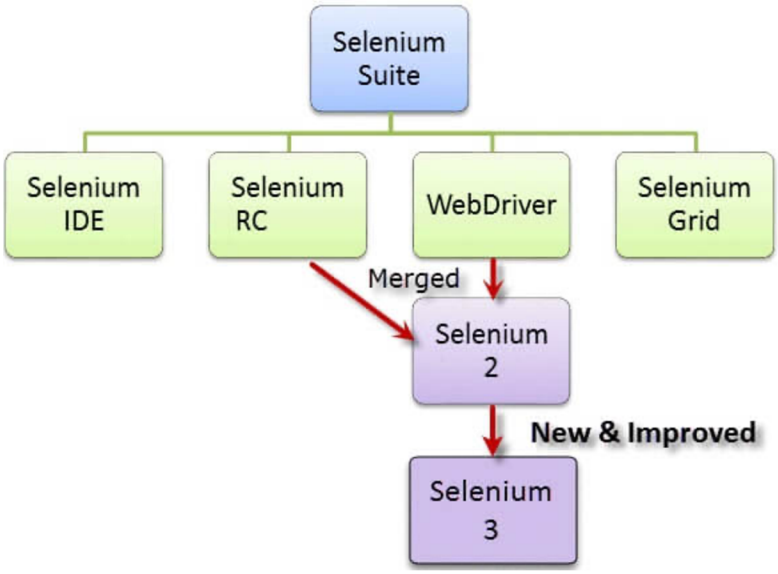
Selenium je open-source framework využívaný pro automatické testování webových aplikací napříč různými prohlížeči a platformami. Testy lze vykonávat na platformách Windows, Linux i Macintosh. Selenium není jen jeden nástroj, ale celá sada nástrojů pro automatizované testování. Každý nástroj ze sady má jinou funkci a je přizpůsobený odlišným potřebám testování. (Guru99, 2021)

Selenium sestává ze čtyř hlavních nástrojů:

- Selenium IDE
- Selenium RC (Remote Control)
- Selenium Web Driver

[20] https://www.selenium.dev

- Selenium Grid



Obrázek 15 - Nástroje selenia (Guru99, 2021)

5.4.2 Selenium IDE

Selenium IDE je open-source doplněk do internetových prohlížečů Chrome a Firefox. Umožňuje nahrávání provedených akcí uživatele ve webové aplikaci a následné spuštění v podobě jednoduchého automatizovaného testu. (SFC, 2019)

Má přehledné uživatelské rozhraní a lze ho například použít jako výpomoc pro tvorbu selektorů pro začátečníky. Umožnuje modifikaci nahraných příkazů v rámci vygenerovaného kódu. Selenium IDE není vhodný pro složitější a dlouhodobé projektové testování aplikací.

5.4.3 Selenium RC

Selenium RC býval dlouhou dobu vlajkovou lodí celé rodiny Selenium. Je to první nástroj pro automatizované testování webových aplikací, který umožnil vývojářům použití jimi preferovaného programovacího jazyka (Java[21], C#[22], PHP, Python, Perl[23], Ruby[24]).

[21] https://www.oracle.com/java/
[22] https://docs.microsoft.com/en-us/dotnet/csharp/
[23] https://www.perl.org
[24] https://www.ruby-lang.org/en/

Mezi jeho výhody patří široká podpora webových prohlížečů a rychlost v porovnání se Selenium IDE. Nevýhodou je nutnost znalosti programovacího jazyka a složitější instalace. (Guru99, 2021)

5.4.4 Selenium Web Driver

V současnosti se považuje za jeden nejpopulárnějších nástrojů pro automatizované testování webových aplikací. Často se na projektech využívá společně se selenium serverem, umožňujícím vzdálené spuštění automatizovaných testů na prohlížeči.

Ve srovnání s oběma přechozími nástroji je považován v mnoha aspektech jako vyspělejší. Implementuje modernější a stabilnější přístup v rámci automatizace akcí prohlížeče. Selenium Web Driver využívá prohlížeč nativně, stejně jako uživatel. Umožňuje lokální použití nebo vzdálené s využitím Selenium serveru. Představuje velký pokrok v automatizaci prohlížečů. Používání nástroje vyžaduje znalost programování. (SFC, 2019)

5.4.5 Selenium Grid

Selenium Grid je nástroj, který umožňuje paralelně spouštět testy na více strojích a zároveň centrálně spravovat různé verze prohlížečů. (SFC, 2019)

V praxi to znamená, že požadovaná testovací sada je, díky paralelnímu běhu jednotlivých testů, vykonána mnohem rychleji. Čím více prohlížečů máme v „Gridu“, tím rychleji budou testy provedené. Počet uvolněných prohlížečů pro jeden běh testů je konfigurovatelný. Možnost paralelního běhu se nemusí vztahovat pouze na jednu sadu testů, je možné pustit více odlišných sad v jednu chvíli. Dále je možné pustit různé sady na různých testovacích prostředích s odlišnou konfigurací zároveň.

5.4.6 Gauge

Gauge[25] je zdarma a představuje open-source framework pro psaní a spuštění automatizovaných testů. Podobných nástrojů lze nalézt na trhu spoustu. V čem se tedy Gauge odlišuje?

[25] https://gauge.org

- Testovací skripty lze zapisovat v obchodním jazyce, kterému rozumí jak tester, tak například zákazník. Tester, který píše samotné skripty, proto nemusí disponovat znalostí programování.
- Jednoduchá, flexibilní a bohatá syntaxe, založená na značkovacím jazyce, anglicky „Markdown“.
- Nabízí modulární architekturu s podporou různých doplňků.
- Podporuje vývoj řízený testy (TDD[26]) a externí zdroje dat.
- Podpora pro Windows, Linux i Macintosh. Umí pracovat s programovacími jazyky (C#, Java, JavaScript, Python a Ruby).
- Skvělá podpora pro nástroj VS Code[27].
- Pomáhá vytvářet udržovatelné testy.

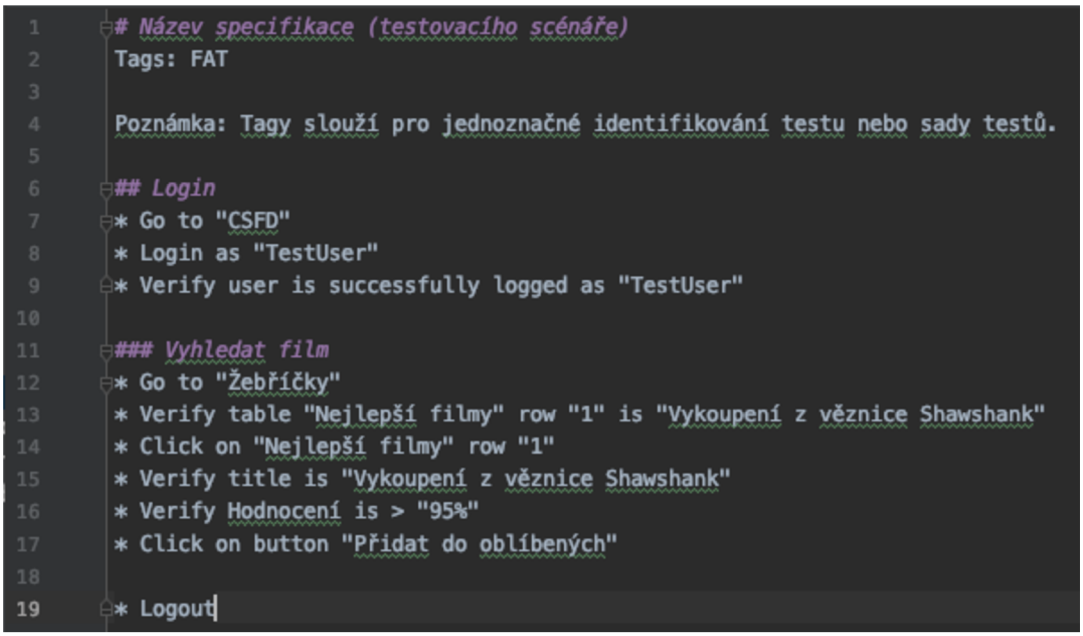
(Gauge, 2021)

Test je představován tzv. specifikací. Specifikace je rámec pro vytvářený automatizovaný test, který ověřuje požadované chování a funkce testovaného softwaru. Konkrétní testovací kroky jsou zapsány ve značkovacím jazyce a celý soubor může být uložený ve formátech typu .md nebo .spec. Ve speciálních případech, kdy se část testu opakuje ve více testech, je možné tuto část odejmout a vložit do speciálního souboru s příponou .cpt. Obsah takového souboru lze později iniciovat v rámci původního a dalších alternativních testů. Jedna specifikace může obsahovat více testovacích případů (scénářů). Pro snadnější práci, správu a vykonávání testů nabízí specifikace použití značek neboli tzv. „tagů“.

Na obrázku níže je znázorněn zápis jednoduchého testu. Povinné atributy jsou název specifikace (#) a alespoň nadpis první úrovně (##). Jak již bylo zmíněno, zápis vlastních testovacích kroků (*) záleží převážně na fantazii a potřebách testera. V tomto případě automatizovaný test navštíví stránku československé filmové databáze, přihlásí se do uživatelské sekce, pokračuje navigací mezi nejlepší filmy, kde ověří nejlepší film

[26] Test-driven Development
[27] Visual Studio Code

a pokračuje na jeho detail. Na detailu filmu se ověří titulek, hodnocení a film se přidá do oblíbených. Lze si všimnout, že pro vyhledání filmu je použit nadpis nižší úrovně (###). Gauge umožňuje dělit testy do několika úrovní. V situaci, kdy část testu na vyšší úrovni skončí chybou, nemá smysl pokračovat s testy nižší úrovně. V tomto konkrétním případě nemá smysl dávat film do oblíbených, pokud se uživatel úspěšně nepřihlásí. Nástroj má velmi pěkné grafické reporty, v případě, že dílčí krok v GUI testu skončí chybou, automaticky vytvoří snímek obrazovky, který je součástí reportu.



Obrázek 16 – Praktická ukázka Gauge (David Pfeifer, 2021)

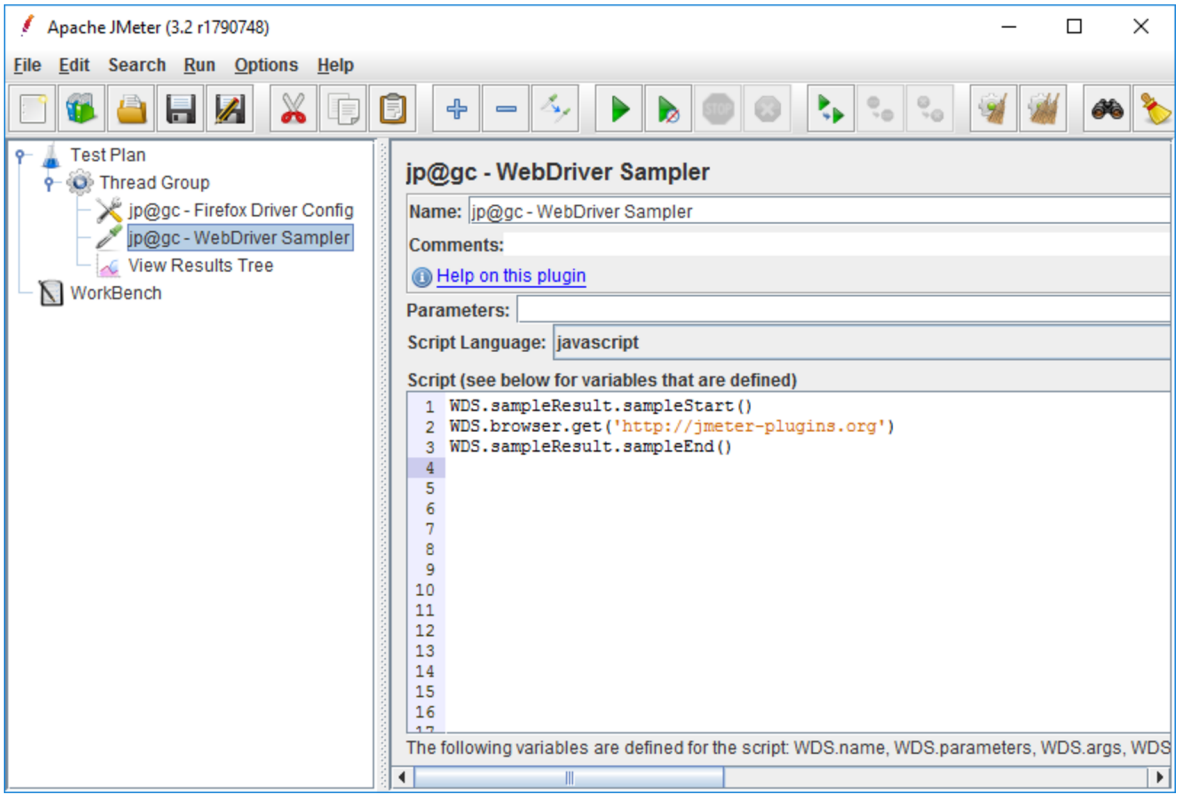
Detailnějšímu rozboru specifikace, tvorbě, exekuci a vyhodnocení testů na reálném projektu je věnována praktická část této diplomové práce.

5.4.7 JMeter

JMeter je open-source nástroj od společnosti Apache Software Foundation. Podporuje tvorbu a exekuci funkcionálních automatizovaných testů. Velmi dobře zastává také disciplínu zátěžového testování, umožňuje analyzovat a měřit výkon webových aplikací a služeb. JMeter, jak již z názvu vypovídá, je založený kompletně na technologii Java. (JMeter, 2021)

JMeter v porovnání s konkurencí vyniká ve své rozšiřitelnosti o různé doplňky, které poskytují mnoho dalších funkcí. Velkou většinu takových doplňků je možné instalovat

přímo z nástroje přes funkci plugin manager, je to velmi rychlé a intuitivní. Hojně používaný bývá například doplněk Selenium Web Driver, který velmi dobře funguje v kombinaci s prohlížečem Firefox. Na obrázku je ukázka uživatelského rozhraní nástroje JMeter, s doplňkem Selenium Web Driver, připraveném k použití na prohlížeči Firefox.



Obrázek 17 – Praktická ukázka JMeter (David Pfeifer, 2021)

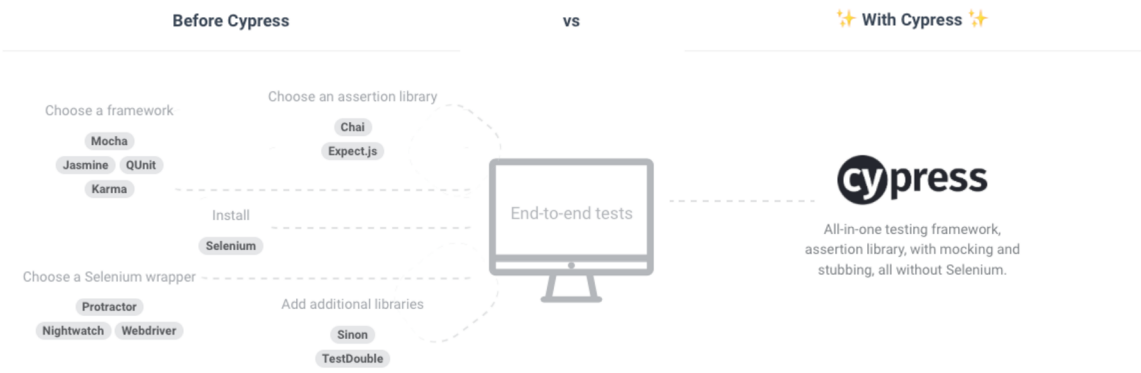
Osobně JMeter rád využívám pro přípravu, běh a vyhodnocení zátěžových testů. Při tvorbě lze využít vestavěného rekordéru, který zaznamená požadovaný testovací skript. Nahrané kroky lze posléze libovolně modifikovat, nástroj umožňuje například přidání načtení dat z externího zdroje. Pak už jen stačí nastavit počet vláken (uživatelů), čas, za jaký se všichni uživatelé připojí a počet opakování. Pro účely vyhodnocení JMeter obsahuje mnoho různých listenerů sloužících pro sledování všemožných statistik. Jejich výběr záleží na povaze daného zátěžového testu.

Mezi další možnosti tohoto nástroje patří testování webových služeb pomocí SOAP[28] a REST[29] požadavků. Dále lze testovat různé databázové transakce a operace pomocí JBDC Sampler.

JMeter je možné nainstalovat a spustit na platformách Windows, Linux a Macintosh.

5.4.8 Cypress

„End-to-end testování není snadná disciplína. Byla to část, kterou spousta vývojářů nenáviděla. Nikdy víc, Cypress usnadňuje nastavení, psaní, spouštění a ladění testů.“ (Cypress.io, 2021)



Obrázek 18 - Cypress před a po (Cypress.io, 2021)

Cypress představuje moderní all-in-one testovací framework, který je vhodný jak pro vývojáře, tak i pro všechny úrovně testerů. Specializuje se na psaní end-to-end testů webových aplikací. Je schopný otestovat vše, co běží ve webovém prohlížeči, například react, angular apod.

Výhody a odlišnosti nástroje Cypress:

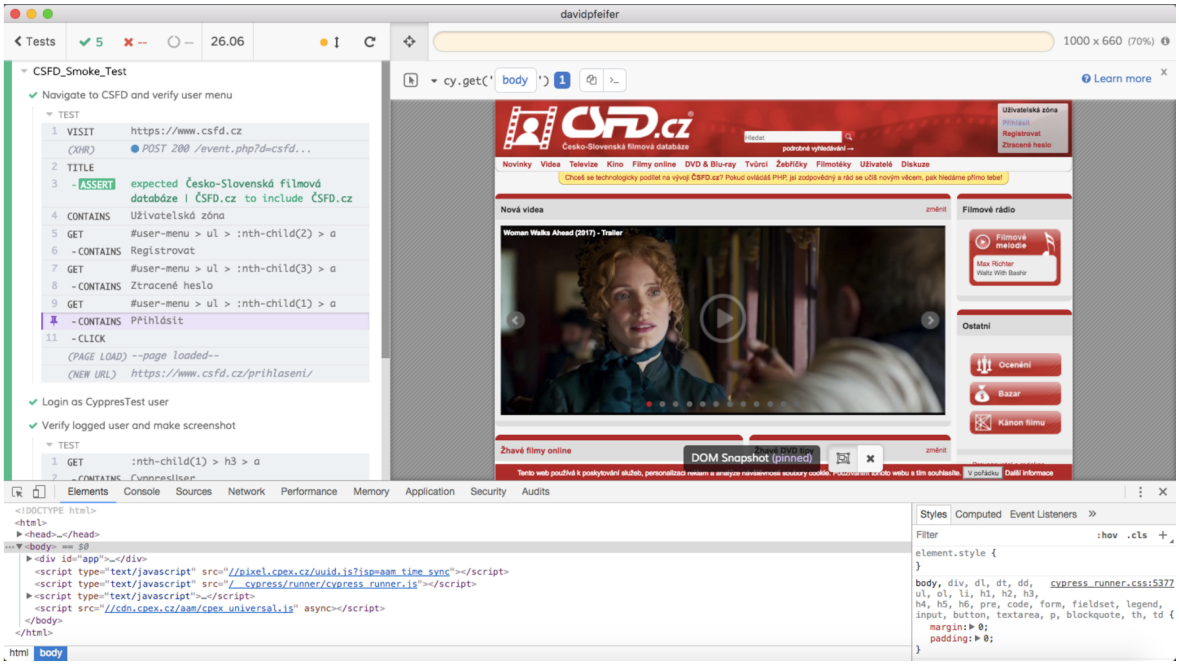
- Cypress jako jeden z mála nástrojů nepoužívá a není založen na seleniu. Byla vytvořena nová architektura, která neprovádí příkazy vzdáleně pomocí sítě.
- Nástroj se zaměřuje na uživatelské testy webových aplikací.

[28] Simple Object Access Protocol
[29] Representational State Tranfer

- Otestuje vše, co lze spustit ve webovém prohlížeči. Úspěšně otestuje webové aplikace napsané moderními technologiemi i starší statické stránky.
- Všechny testy napsány pouze v JavaScriptu. Testovací kód proveden uvnitř prohlížeče.
- Umožňuje efektivní využití TDD, anglicky „test-driven development“ – vývoje řízeného testy.
- Efektivita, rychlost – navrhnut tak, aby vývoj a testování mohly probíhat současně. Při testování stále vidíte aplikace a zároveň máte přístup k vývojářským nástrojům.
- All-in-one framework – vychází z obrázku výše. Ostatní nástroje na bázi selenia vyžadují konfiguraci a spolupráci více komponent, což bývá v praxi zdlouhavé. Cypress má veškeré knihovny apod. integrované a funguje ihned po nainstalování.

(Cypress.io, 2021)

Níže lze vidět na obrázku praktický příklad jednoduchého testu vykonaného s využitím nástroje Cypress. V rámci testu je načtena stránka československé filmové databáze, následuje ověření titulku a elementů uživatelské části. Nakonec se test přihlásí jako testovací uživatel a udělá snímek obrazovky. Po celou dobu běhu testu v prostředí Cypress vývojář vidí, co se na stránce právě děje, kdykoli může test pozastavit a prozkoumat konkrétní elementy na stránce. Nástroj umožňuje i automatické generování selektorů. Generování selektorů nicméně doporučuji jen pro začátečníky, jejich syntaxe je nečitelná a nepřehledná. Ladění testů je v nástroji intuitivní a jednoduché.



Obrázek 19 - Praktická ukázka Cypress (David Pfeifer, 2021)

Nástroj v rámci lokálního spuštění reportuje výsledky lokálně, kam si uživatel zvolí. Při vzdáleném spuštění přes příkazovou řádku reporty ukládá vzdáleně prostřednictvím Git Hub. Registraci a nutnost používání Git Hubu vidím jako nevýhodu. Výstupem testů jsou příjemně zpracované grafické reporty, které mohou obsahovat snímky obrazovky i zaznamenání videa z vykonaného testu.

Cypress podporuje pouze čtyři prohlížeče založené na Google Chrome, samotný Chrome, Chromium, Electron a Canary. Podporuje operační systémy Windows, Linux i Macintosh.

Nabízí placené licence podle velikosti projektu, počtu uživatelů a počtu nahraných testů za měsíc.

5.4.9 Soap UI a Ready API

Soap UI, jak již název napovídá, je nástroj pro testování webových služeb. Podporuje testování SOAP, REST webových služeb i služeb založených na protokolu http. Základní omezená verze Soap UI je zdarma. Pro využití pokročilejších funkcí existuje komplexní nástroj Ready API, který je placený. Licence jsou celkem drahé, nejlevnější licence s modulem pro API testování vyjde při aktuálním kurzu eura 26,2 Kč na téměř 18 000 Kč ročně. Další nabízená licence pro modul virtualizace API vyjde na téměř 30 tisíc Kč. Poslední nabízený modul pro performance testování vyjde na 150 000 Kč. (SmartBear,

2021) Podobné funkce, jako performance modul, nabízí například JMeter, proto je dobré volbu nástroje vždy zvážit s ohledem na nabízené funkce a cenu.

Placený testovací modul v rámci nástroje Readi API podporuje pokročilé funkce, mezi které patří podpora různých skriptovacích jazyků (např. Groovy[30]), integrace na CI/CD nástroje, vyhodnocování testů, podrobné funkce pro logování, automatizace tvorby požadavků, automatizace odesílání a vyhodnocování odpovědí, nahrání testovacích dat přímo do databáze, změna konfigurace pro testované uživatele a spoustu dalších.

Ready API zahrnující všechny moduly se dále používá pro provedení a automatizaci výkonnostních, zátěžových, databázových a penetračních testů. Dále obsahuje pokročilé funkce pro virtualizaci zmíněných služeb (REST, SOAP, JBDC, JMS). (SmartBear, 2021)

Testování restových služeb se zakládá na zasílání různých požadavků na REST API, v rámci, kterých můžeme ověřovat autentizaci, komunikaci, prodlevu mezi přijetím požadavku a odesláním odpovědi a odpověď samotnou. V rámci odpovědi lze dále ověřovat formát, obsah a správnost parametrů.

Testování služeb založených na WSDL/SOAP umožňuje ověřovat například zasílání a přijímání XML zpráv obsahující důležitá data. Takové testy se dají rozšířit o ověření správných reakcí služeb.

5.4.10 Nativní funkce prohlížeče

Prozkoumat prvek je nativní funkce prohlížeče určená převážně pro vývojáře, ale skvěle poslouží i jako pomůcka pro testování webových aplikací. Pomáhá při zjišťování příčiny defektů a jejich následném odstranění. Dále se dá využít pro penetrační i jednoduché výkonnostní testy. Podrobnější informace níže se vztahují k prohlížeči Safari[31]. V ostatních prohlížečích se mohou názvy záložek a funkcí lehce lišit.

Záložka prvky zobrazuje zdrojový kód načtené stránky. Zde můžeme měnit hodnoty atributů a sledovat, jak se změny projeví na webové stránce. Změna prvků na stránce

[30] https://groovy-lang.org
[31] https://www.apple.com/safari/

se využívá při výše zmíněných penetračních testech. Pravá část záložky umožňuje analýzu CSS stylů, které daný HTML prvek používá.

Konzole umožňuje odhalit například chyby v Javascriptu. Zobrazuje chybovou hlášku a identifikuje, na jakém řádku a v jakém souboru se chyba nachází.

Záložka síť informuje o době načtení konkrétních prvků na stránce, ze kterých se dají vyprofilovat potencionální problémy pro výkon webu. Podobné informace zobrazuje i záložka časové osy.

5.5 „Best practices" pro implementaci automatizovaných testů

- Nezaměřovat se pouze na GUI testy. Vždy myslet na testovací pyramidu. Testy uživatelského rozhraní, by v rámci automatizace, měly přijít na řadu jako poslední. (Broadcom, 2021)

- Nesnažit se za každou cenu automatizovat všechny testy. Vždy je dobré si rozmyslet, co se vyplatí a co již nikoliv.

- Nepodcenit výběr testovacího nástroje. Z předchozích kapitol je zřejmé, že každý typ nástroje má odlišnou cílovou skupinu. Proto je vhodné vybírat nástroj v souladu s povahou projektu a budoucími uživateli.

- Nezanedbávat údržbu testů. V případě dlouhodobého zanedbávání údržby je následná optimalizace daleko náročnější a nákladnější.

- V případě testování kompatibility prohlížeče není nutné spouštět kompletní sadu testů. (Broadcom, 2021)

- Používat srozumitelné názvy testů. Z názvu by mělo být okamžitě zřejmé, co je cílem testu. Není efektivní ztrácet čas zkoumáním, co vlastně test dělá.

- Zamyslet se nad reportováním výsledků testů. Vyhodnocování testu by mělo být jednoduché a efektivní. Je vhodné zvážit využití integrace s nástroji pro podporu testování, například Practi Testem apod.

- Všechny testy by měly být znovupoužitelné. Pokud test mění povahu testovaného systému, tak by měl ideálně obsahovat kroky pro navrácení zpět.

- Nezávislost testů. Není dobré, pokud výsledek jednoho testu závisí na úspěšném provedení jiného. Vhodné, aby test používal nebo si připravoval vlastní testovací data. Tato metoda se nazývá testování řízené daty (DDT[32]). Například načítání dat z předem připravených tabulek nebo externího csv.

[32] Data-driven testing

6 Continuos integration

6.1 Definice

Kontinuální integrace je praktikou vývoje softwaru vhodná pro týmy, ve kterých jednotliví členové vývojového týmu často dělají změny nebo přidávají nové funkcionality minimálně na denní bázi. To má za následek časté integrování nových verzí, zpravidla několikrát za den. Každá verze bývá otestována jednotkovými i integračními testy s ohledem na odhalení co nejvíce implementačních defektů. (Fowler, 2006)

Typickým příkladem může být projekt, založený na agilní metodice s krátkými sprinty. Pro efektivní využívání výhod kontinuální integrace je vhodné implementovat do projektu některý nástroj pro automatizaci rutinních činností.

Výhody použití CI/CD:

- Zvýšení efektivity a snižování rizik v rámci integračního procesu.
- Zpřehledňuje a zjednodušuje celý proces integrace – vždy víme co funguje a co ne.
- Chyby neodstraní. Ale na základě pravidelnosti, zahrnující malé přírůstky v implementaci, usnadňuje vývojáři analýzu a odstranění problému. Programátor má danou změnu ještě v paměti. Pro zjištění defektu se dá použít porovnání předchozí verze systému s novou.
- Časté nasazování nových verzí je podstatné pro testery i uživatele systému. Umožňuje dodávat nové funkcionality častěji. Pomáhá prolomit bariéru mezi vývojovým týmem a klientem.

(Fowler, 2006)

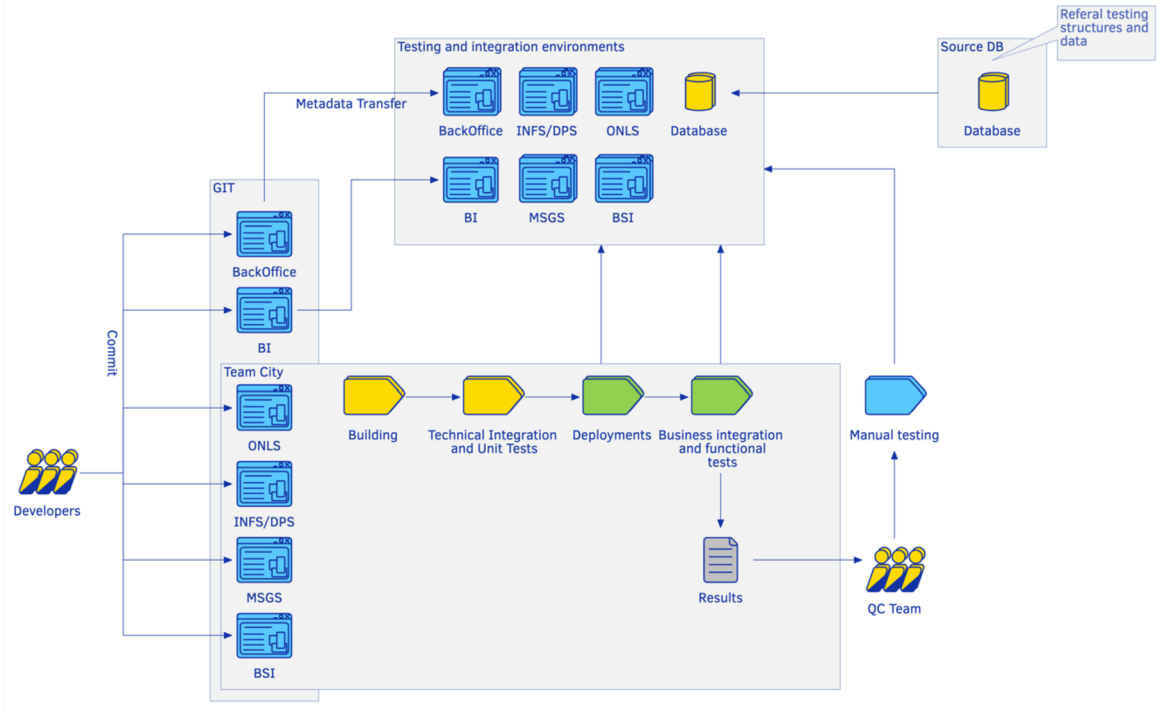
6.2 Nejpoužívanější nástroje

6.2.1 Teamcity

Teamcity patří mezi nejpoužívanější servery pro continuous integration. Vytvořen a spravován společností JetBrains. Je založený na jazyce Java a umožňuje vývojářům automatizovaně integrovat, verifikovat a nasazovat nové verze. Pomáhá také automati-

zovat proces testování softwaru. Umožňuje spolupráci s nástroji pro správu verzí, například GIT. Je uživatelsky přívětivý a má globální podporu, včetně dobré dokumentace. Teamcity je ideální pro vývojáře, kteří ocení mimořádně přívětivé grafické rozhraní, okamžité využívání funkcí a snadné nastavení. Je možné rozšíření pomocí doplňků, jejich nabídka je v porovnání s nástrojem Jenkins méně obsáhlá. Od určitého počtu buildů je placený. (BrowserStack, 2021)

Nástroj Teamcity je využíván v Unicornu na více projektech. Na obrázku níže je znázorněna struktura CI/CD a integrace teamcity na projektu Nordic Balance Settlement.



Obrázek 20 - Proces průběžné integrace s využitím nástroje TeamCity (Unicorn, 2020)

6.2.2 Jenkins

Jenkins představuje jedničku na trhu serverů pro účely CI/CD. Má obrovskou komunitu a skvělou dokumentaci. Je napsán v jazyce Java a podporuje více než 1000 dalších doplňků. Použití je možné na různých platformách od Windows až po Linux. V případě potřeby integrace jakéhokoli konkrétního nástroje s Jenkinsem, stačí nainstalovat příslušný plugin. Například Git, Maven 2, Amazon EC2 atd. Na rozdíl od Teamcity je Jenkins kompletně zdarma. Konfigurace zabere více času než u Teamcity, vývojáři musí nainstalovat a nakonfigurovat potřebné doplňky.

7 Představení projektu

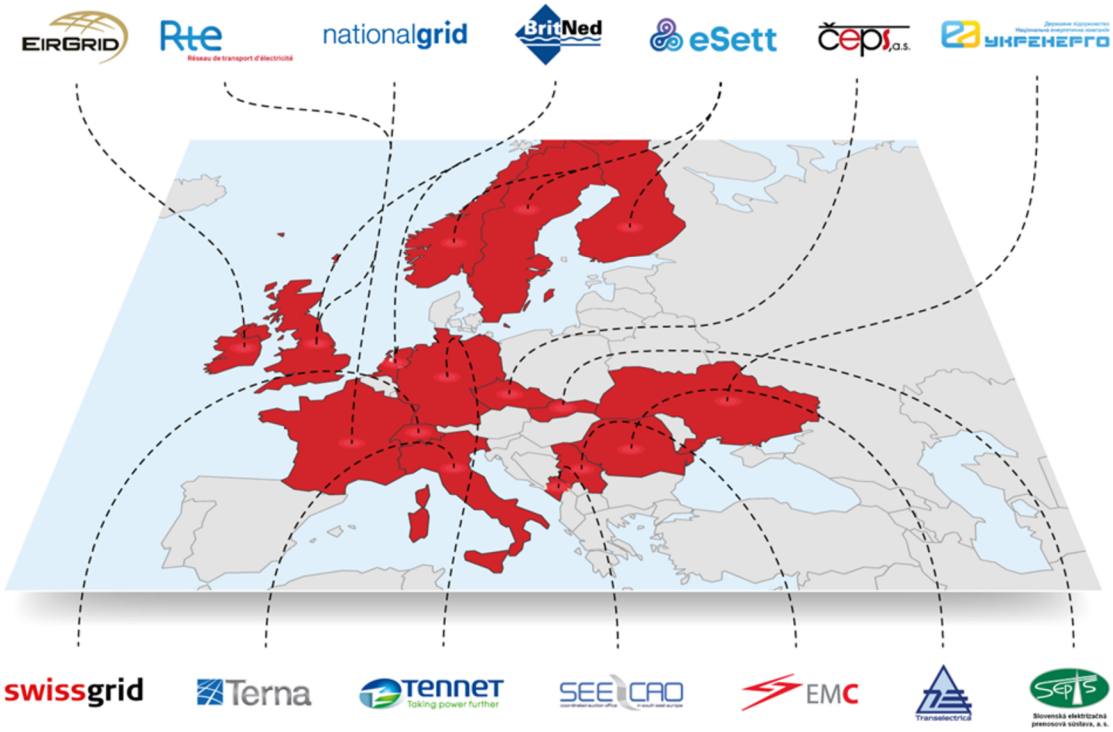
Jak již bylo zmíněno v kapitole číslo 2., praktická část této diplomové práce je zaměřena na návrh a implementaci automatizovaných testů pro efektivní vývoj softwaru na konkrétním projektu, který je realizován společností Unicorn. Celý proces integrace zahrnuje nejen analýzu požadavků a architektury poskytovaného informačního systému, ale i samotný výběr vhodného nástroje pro automatizaci testování. Před iniciací komparativní analýzy jednotlivých nástrojů je důležité nejprve poznat specifika a potřeby daného projektu. Tato kapitola je věnována představení informačního systému a platformy, na kterých se celý projekt zakládá.

7.1 Damas

Damas představuje klíčový produkt společnosti Unicorn v oblasti energetiky. Je kontinuálně vyvíjen od roku 2000 až po současnost. Poskytuje podporu obchodních i technických procesů systémových operátorů přenosových sítí, podmořských kabelů a market operátorů. Základem produktu je speciálně navržená technologická platforma, která umožňuje efektivní přizpůsobení se konkrétním požadavkům zákazníka a neustále se měnícím regulím trhu. Ve své podstatě je to systém založený na pravidlech. Díky tomu neslouží jen jako nástroj pro podporu obchodních procesů v energetice, ale také jako prostředí, kde je možné takové procesy v reálném čase modelovat a nasazovat. Platforma Damas nabízí modelování a konfiguraci procesních řízení, datové struktury a jejich modifikaci v čase, výpočetní moduly, datové tabulky a grafy, zasílání notifikací a zpráv a integraci na okolní systémy. Samozřejmostí je také oboustranná integrace s kancelářskými programy. (Unicorn Solutions a.s., 2020)

Výhodou informačního systému, založeného na platformě Damas, je možnost implementace bez hlubší znalosti programovacích jazyků. Jádro systému lze totiž vyvíjet a konfigurovat prostřednictvím uživatelského rozhraní, které je součástí aplikace. S ohledem na tuto skutečnost se do vývojového procesu zapojují kromě samotných programátorů i další role, a to především obchodní analytici. Samotnou konfiguraci je pak možné exportovat mimo systém a znovu importovat na různá prostředí. V případě potřeby lze systém dále rozšiřovat specificky naimplementovanými částmi. (Unicorn Solutions a.s., 2020)

Již několik let je dodáván do zemí napříč celým evropským kontinentem, včetně souostroví Velké Británie. Níže na obrázku jsou jednotlivé země a poskytovatelé služeb v oblasti energetiky znázorněny. V návaznosti na tuto práci je důležité zmínit společnost eSett, se sídlem ve Finsku, která provozuje informační systém s názvem Nordic Balance Settlement. Systém zprostředkovává vše potřebné pro efektivní sdružení trhu s elektřinou v severských zemích Norsko, Finsko a Švédsko. Nově se od března 2020 do systému zapojilo i Dánsko.



Obrázek 21 - Damas MMS (Unicorn Systems a.s., 2013)

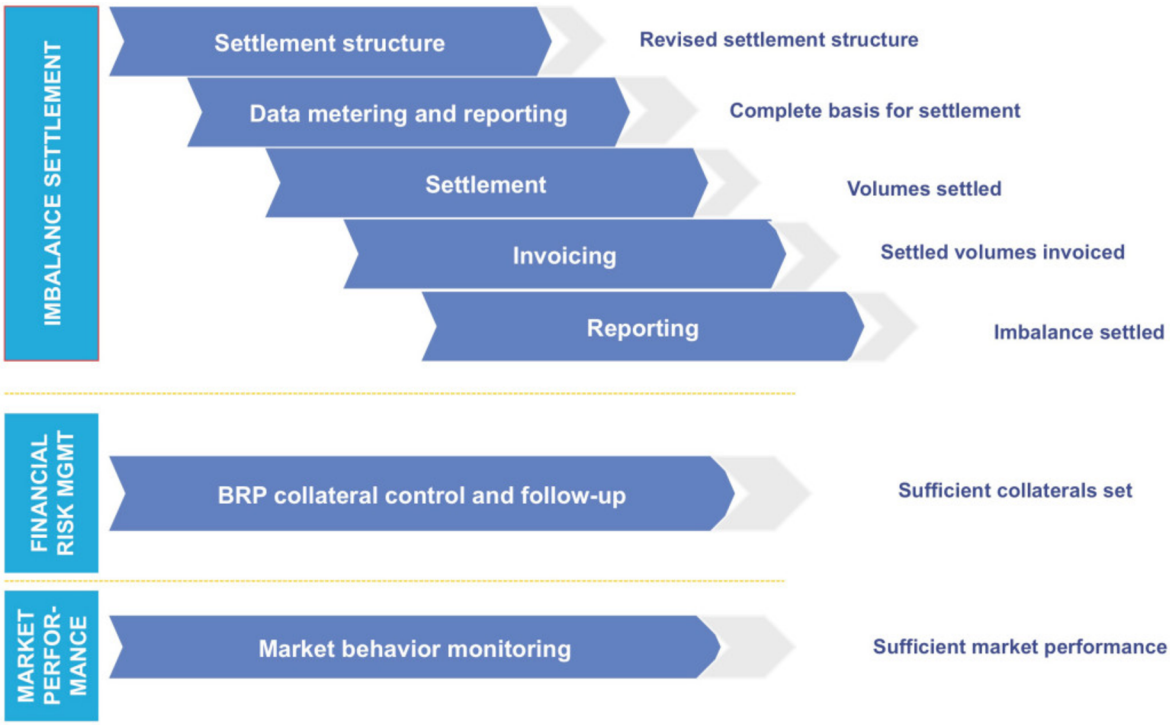
7.2 Nordic Balance Settlement

Nordic Balance Settlement neboli NBS je systém dodávaný firmou Unicorn pro zákazníka ze Skandinávie. Systém NBS v reálném čase přijímá a vyhodnocuje data od více než 1000 účastníků trhu s energiemi v rámci regionů Finska, Švédska, Norska a Dánska. Jedná se o komplexní řešení, založené na výše představené platformě Damas. Systém je doplněný o další specificky naimplementované komponenty, navrhnuté přímo na míru požadavkům klienta. Jednou z hlavních podmínek úspěšného produkčního provozu je dostupnost systému 24 hodin denně 7 dní v týdnu.

Hlavní funkcí NBS je výpočet, vyrovnání a fakturace způsobených odchylek v rámci trhu s elektřinou v severských zemích. To vše na základě jednotného harmonizovaného modelu. Cílem vyrovnání odchylky je nastolení finanční rovnováhy na trhu s elektřinou po každé provozní hodině. Odchylky mohou vzniknout v rámci výroby i spotřeby energie a jsou pravidelně zúčtovány všem účastníkům trhu právně zodpovědným za danou odchylku. Takový účastník se zkráceně označuje jako BRP[33]. BRP dále odpovídá za oblast plánování a jednotlivé obchodníky (RE[34]), kteří prodávají energii koncovým uživatelům. Dalšími účastníky trhu jsou provozovatelé přenosových soustav (TSO[35]), kteří mají hlavní zodpovědnost za páteřní sítě a fyzickou rovnováhu celé elektrické soustavy. Poslední článek tvoří provozovatelé distribuční soustavy (DSO[36]), kteří jsou zodpovědní za připojení producentů energie (elektrárny) a spotřebitelů k distribuční síti. DSO mají za úkol měřit produkci, spotřebu a výměnu energie s jinými sítěmi a tato data reportovat oprávněným subjektům. (eSett Oy, 2021)

Celý systém je rozdělen do více spolupracujících oblastí, z nichž každá plní odlišnou funkci v rámci komplexního procesu vyrovnání odchylek. Následující obrázek zachycuje celý proces začínající řízením strukturních dat a končící finálním reportem.

[33] Balance Responsible Party
[34] Retailer
[35] Transmission System Operator
[36] Distribution System Operator



Obrázek 22 - Funkce modelu pro vypořádání nerovnováhy (eSett Oy, 2021)

- Správa strukturních informací zahrnuje mnohé. Umožňuje zakládat, odebírat a aktualizovat platnost i informace jednotlivých účastníků trhu. Zahrnuje veškeré řízení vazeb mezi oblastmi, zařazení produkčních jednotek do oblastí a propojení s obchodníky atd. Na základě definice vazeb, zodpovědností a dalších specifických podmínek vznikají takzvané datové struktury, do kterých DSO, BRP a v některých případech i TSO pravidelně reportují hodinové hodnoty. Takovou strukturu může představovat například produkce a spotřeba energie v rámci sítě daného DSO.

- Oblast měření a reportování představuje rozhraní pro ukládání hodinových dat, zahrnující specifické validace a uzávěrky pro reporting. Součástí je i specifická implementace pro reportování dat pomocí datových zpráv ve formátu xml. Tato data slouží jako vstup do následné kalkulace.

- Oblast zpracování výpočtu odchylek je v rámci systému jedna z klíčových. Vstupy, v podobě obdržených dat, modul pravidelně přepočítává podle naimplementovaného komplexního vzorce pro každý den z nakonfigurovaného intervalu. Výstup kalkulace představuje vypočítaná nerovnováha za každou reportovanou hodinu v oblasti trhu, kde působí dané BRP. Z vypočítaných hodnot

je společně s agregovanými vstupy následně vytvořen souhrnný balanční report za vybrané období pro zvolenou oblast a BRP. Položky spotřeby a produkce musí být v rámci balančního reportu vždy vyrovnané.

- Vypočítané hodnoty z modulu kalkulace nerovnováhy slouží jako jeden ze vstupů pro fakturaci. Výsledná částka pro vyrovnání se počítá v souladu s naimplementovaným vzorcem pro fakturaci zahrnujícím odchylky BRP, naúčtované poplatky a platby za aktivované rezervy. Faktury jsou na konci procesu odeslány.

- Oblast odchozí komunikace ze systému zahrnuje vytváření, distribuci a publikování různých zpráv a souborů. Sytém podporuje více způsobů jak příchozí, tak i odchozí komunikace.

- Oblast řízení finančních rizik: Společnost eSett zprostředkovává ve jménu jednotlivých TSO finanční vyúčtování odchylek. Se zodpovědností za veškeré platby jí tím vzniká významné riziko v případech, ve kterých by BRP nebylo schopno dostát svým závazkům. Z tohoto důvodu každé BRP předem poskytuje určitou částku na zajištění krytí před případným rizikem. Toto neplatí pouze v Dánsku, kde zodpovědnost a všechna rizika přebírá přímo TSO. (eSett Oy, 2021)

- Poslední oblast tvoří monitorování trhu a chování jednotlivých účastníků za účelem shromažďování informací. Dále s využitím klíčových ukazatelů výkonu (KPI[37]) docílit, že odchylky jednotlivých BRP budou co nejnižší úrovni. (eSett Oy, 2021)

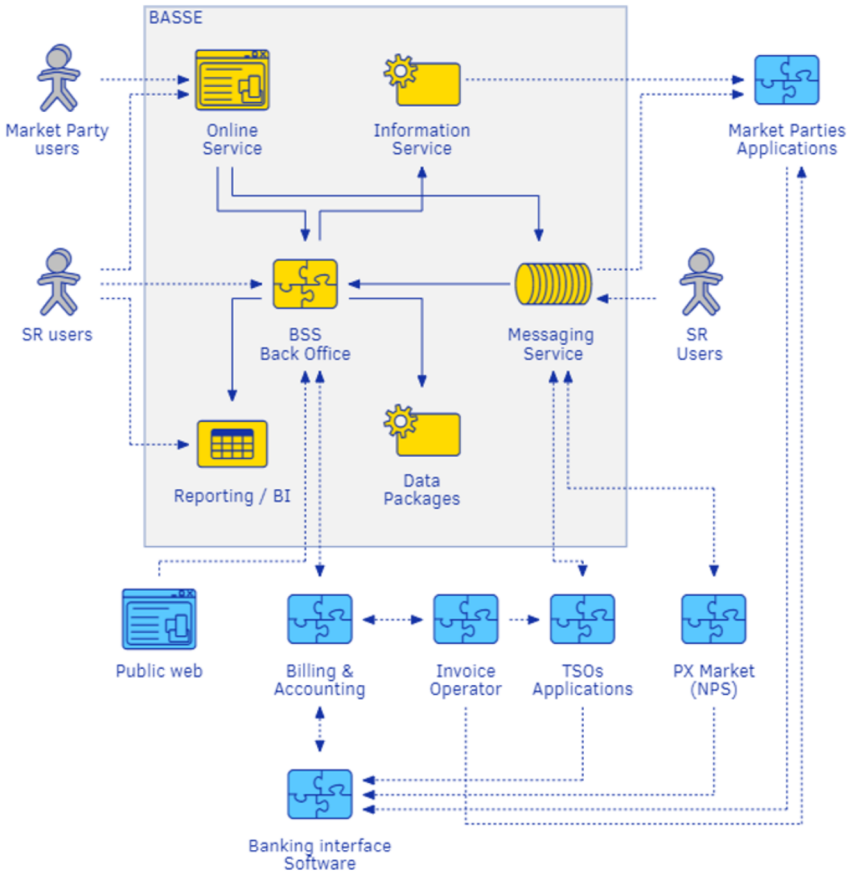
[37] Key performance indicator

8 Analýza systému a definice požadavků

Kapitola navazuje úzce na tu předchozí a jejím dílčím cílem je dekomponování informačního systému NBS z celku na jednotlivé komponenty. Obchodní specifika projektu a platformy již byly představeny a teď je důležité zjistit, co za tím vším vlastně stojí. Znalost architektury systému a pochopení, jak se jeho komponenty chovají, jsou dalšími důležitými znalostmi pro následnou definici požadavků, stanovení cílů, výběr nástrojů, definici testovacího prostředí i samotnou implementaci.

8.1 Architektura systému

Architektura systému NBS je s ohledem na propojení velkého množství funkčních i nefunkčních požadavků poměrně robustní. Celý systém sestává ze 6 komponent, které jsou navzájem propojeny. Tyto komponenty mohou být dále integrovány na systémy třetích stran.



Obrázek 23 - Architektura systému (David Pfeifer, UUBML, 2021)

- BSS Back Office představuje jádro systému založené na platformě Damas. Je tvořeno specifickou konfigurací (takzvanými metadaty), kterou klikají analytici

s využitím grafického rozhraní, a specifickou implementací v jazyce MS .NET. Obsahuje veškerá procesní řízení, struktury a časové řady pro ukládání dat. Krom toho obsahuje rozhraní pro komunikaci se všemi dalšími komponentami. K této komponentě mají v reálném provozu přístup pouze operátoři trhu.

- Online Service je frontendová webová aplikace poskytující definované služby a funkce všem účastníkům trhu (BRP, RE a DSO). Zde je s ohledem na přístup externích uživatelů kladen velký důraz na kvalitu a tím pádem na důkladné otestování.

- Information Service umožňuje uživatelům dotázat se na různá zpracovaná data zasláním požadavku na restovou službu. Služba se v případě validního požadavku a úspěšné autentizace na data dotáže databáze za pomocí SQL dotazu a následně vrací odpověď uživateli.

- Data Packages fungují na obdobném principu, jako Information Service, s rozdílem, že datové balíčky, obsahující stejný formát dat, jsou uživatelům zasílány automaticky na pravidelné bázi na základě toho, jestli daný balíček mají přihlášený k odběru.

- Messaging Service je postavena na technologii Java a na Red Hat Jboss Fuse[38]. Tato služba se stará o příjem veškerých zpráv z trhu a zasíláním potvrzení, datových balíčků a odchozích datových zpráv zpět jednotlivým účastníkům. Reportování pomocí naměřených dat s využitím datových toků je v rámci NBS nejběžnějším postupem. Každou denní hodinu musí být komponenta schopna zpracovat i několik tisíc zpráv. Na Messaging Service jsou kladeny nejvyšší nároky na stabilitu, spolehlivost a výkon. Není přijatelné jakoukoli zprávu nezpracovat, či dokonce ztratit. Uživatel si může vybrat, jestli chce data zasílat a obdržet pomocí FTP[39], webové služby nebo emailu.

- Business Intelligence je implementováno technologií Microsoft SQL a umožňuje dolování dat a elegantní generování přizpůsobených reportů.

[38] https://access.redhat.com/documentation/en-us/red_hat_jboss_fuse/6.2.1/html/getting_started/product_overview
[39] File Transfer Protocol

8.2 Analýza výchozího pokrytí automatizovanými testy

Na základě testovací pyramidy z kapitoly 5.2 jsou nejpodstatnější a nejpočetnější kategorií jednotkové testy. Jejich návrhem se práce zabývat nebude, jelikož se jedná o testy tvořené vývojovým týmem. Nicméně správný test architekt by měl mít přehled o všech kategoriích testů, proto jsou zařazeny alespoň do úvodní analýzy.

Na základě krátké explorace bylo zjištěno, že specifická implementace Back Office je kompletně pokryta jednotkovými testy. Například výpočetní modul odchylek, strukturální změny, datové toky apod. Testy jsou založeny na technologii NUnit.

Online Service jednotkové testy pokrývají až 72 % kódu. Naimplementovány jsou s využitím frameworků NUnit a Jasmine.

Messaging Service má pokryté jednotkovými testy základní funkcionalitu. Obsahuje testy pro ověření všech podporovaných komunikačních kanálů. Dále sadu nízko úrovňových integračních testů pro ověření správných odpovědí. V neposlední řadě jsou naimplementovány automatické výkonnostní testy, které ověřují změny ve výkonu s každým novým buildem. Testovací tým se opírá o rozsáhlé řešení testování zasílání zpráv řízené daty implementované v nástroji Ready API Pro. Nástroj obsahuje šablony pro každý datový tok a na základě dat připravených v externím zdroji umí poskládat xml zprávu, zaslat ji pomocí webové služby do Messaging Service a vyhodnotit odpověď pomocí assertions. Očekávané odpovědi jsou definované ve stejném externím zdroji. Tímto nástrojem jsou verifikovány obchodní i technické funkcionality příchozích datových toků, validace přijatých zpráv v Messaging Service a obchodní validace v rámci zpracování zpráv implementované v Back Office.

Information Service a Data Packages jsou pokryty jednotkovými testy z 86 %. Jedná se o menší a technicky jednodušší komponenty s nižším rizikem výskytu chyb. QC tým pro testování oblasti požadavků na Information Service používá podobné řešení, jako v případě Messaging Service. Nástroj si připraví vlastní data updatem do příslušné tabulky v databázi a následně na základě vstupů v externím zdroji vytvoří požadavek, pomocí kterého se na nahraná data dotáže. Cílem testů je ověření, že služba vrací správná data ve správném formátu. Tímto způsobem jsou pokryty veškeré podporované dotazy v souladu s funkční specifikací.

V počátcích projektu téměř úplná absence dalších úrovní automatizovaných testů. Prostřední úroveň integračních testů není zatím použita. Identifikováno kolem 40 automatizovaných GUI testů. Testovací nástroj je napsán na míru v jazyce C# a testy jsou exekuované s využitím frameworku Selenia. Testovací skripty jsou definovány kombinací naimplementovaných metod a selektorů (Xpath, CSS) v Microsoft Excelu. Zde bych rád shrnul nevýhody takového přístupu k automatizaci.

- Velká náročnost na přípravu dat a testů. Každý krok takového scénáře se skládá z vyplnění 8 sloupců v Excelu, který se stává v případě komplexnějšího testu nepřehledný. Nutná podpora maker.
- Metody musí být specificky nadefinované na úvodním listu souboru.
- Téměř nemožná údržba: v případě změny implementace webové aplikace, například přejmenování ID daného elementu, se musí všechny testy jeden po druhém projít a aktualizovat. To je časově náročné i při malém počtu testů. V případě budoucího rozšíření by to bylo neudržitelné.
- Absence jakéhokoli graficky přehledného a jednoduše přístupného reportu výsledků. Výsledky vygenerovány do stejného v Excel souboru.
- Podpora pouze pro prohlížeč Firefox. Nutná instalace dalších doplňků a možnost pouze lokálního spuštění.
- Nemožnost integrace na CI/CD nástroje.
- Celková složitost brání zapojení více členů testovacího týmu a pro vývoj je nutné dokonale ovládat tvorbu selektorů.

alias	selector
TABLE(id).ROW(byCode, byVal).BUTTON	//div[@id='{%1%}']//tr[.//td[@code='{%2%}']//*='{%DATA%}']//button
TABLE(id).ROW(byCode, byVal).CONTEXT_MENU(link)	//div[@id='{%1%}']//tr[.//td[@code='{%2%}']//*='{%DATA%}']//a[contains(text(),'{%3%}')]
TABLE(id).ROW(byColCode, byColValue).EXPAND-DETAIL	//div[@id='{%1%}']//tr[.//td[@data-column-code='{%2%}']//*='{%DATA%}']//div[@class='glyphicon glyphicon-chevron-down btn-table-expand-detail']
TABLE(id).ROW(byColCode, byVal).BUTTON	//div[@id='{%1%}']//tr[.//td[@data-column-code='{%2%}']//*='{%DATA%}']//button
TABLE(id).ROW(byColCode, byVal).CONTEXT_MENU(link)	//div[@id='{%1%}']//tr[.//td[@data-column-code='{%2%}']//*='{%DATA%}']//a[contains(text(),'{%3%}')]
TABLE(id).ROW(byCode, byVal).TD(datCode)	//div[@id='{%1%}']//tr[.//td[@code='{%2%}']//*='{%DATA%}']//td[@code='{%3%}']
TABLE(id).ROW(byColCode, byVal).TD(datColCode)	//div[@id='{%1%}']//tr[.//td[@data-column-code='{%2%}']//*='{%DATA%}']//td[@data-column-code='{%3%}']
TABLE(id).ROW(byColHeader).LINK(text)	//div[@id='{%1%}']//tr[.//th[contains(text(),'{%2%}')]]//a[contains(text(),'{%3%}')]
TABLE(id).ROW(byColHeader).SPAN(class)	//div[@id='{%1%}']//tr[.//th[contains(text(),'{%2%}')]]//span[@class='{%3%}']
TABLE(id).ROW(byName and byRE).TD(datColCode)	//div[@id='{%1%}']//tr[.//td[@data-column-code='PU_NAME']//*='{%2%}' and .//td[@data-column-code='RE']//*='{%3%}']//td[@data-column-code='{%4%}']
TABLE(id).ROW(byName and byRO).TD(Code)	//div[@id='{%1%}']//tr[.//td[@code='PU_NAME']//*='{%2%}' and .//td[@code='RO_NAME']//*='{%3%}']//td[@code='{%4%}']
TABLE(id).ROW(byRE and byEDType).TD(datColCode)	//div[@id='{%1%}']//tr[.//td[@data-column-code='RE']//*='{%2%}' and .//td[@data-column-code='ED_TYPE']//*='{%3%}']//td[@data-column-code='{%4%}']
TABLE(id).ROW(byValue1, byValue2).COLUMN(colNo).BUTTON	//div[@id='{%1%}']//tbody/tr[td/*[contains(.,'{%2%}')] and td/*[contains(.,'{%3%}')]]/td[{%4%}]//button
TABLE(id).ROW(byValue1, byValue2).COLUMN(colNo).TEXT	//div[@id='{%1%}']//tbody/tr[td/*[contains(.,'{%2%}')] and td/*[contains(.,'{%3%}')]]/td[{%4%}]//span[contains(@class,ng-binding)]
TABLE(id).ROW(byValue1, byValue2).INPUT	//div[@id='{%1%}']//tbody/tr[td/*[contains(.,'{%2%}')] and td/*[contains(.,'{%3%}')]]//input
TABLE(id).ROW(rowNo).A(link)	//div[@id='{%1%}']//tr[{%2%}]//a[contains(text(),'{%3%}')]
TABLE(id).ROW(rowNo).BUTTON	//div[@id='{%1%}']//tr[{%2%}]//button
TABLE(id).ROW(rowNo).COLUMN(colNo).BUTTON	css={%1%} tbody tr:nth-of-type({%2%}) td:nth-of-type({%3%}) button
TABLE(id).ROW(rowNo).COLUMN(colNo).INPUT	css={%1%} tbody tr:nth-of-type({%2%}) td:nth-of-type({%3%}) input
TABLE(id).ROW(rowNo).COLUMN(colNo).SELECT	css={%1%} tbody tr:nth-of-type({%2%}) td:nth-of-type({%3%}) select
TABLE(id).ROW(rowNo).COLUMN(colNo).TEXT	css={%1%} tbody tr:nth-of-type({%2%}) td:nth-of-type({%3%}) span.ng-binding
TABLE(id).ROW(rowNo).CONTEXT-MENU	css={%1%} tbody tr[editor-code='{%2%}'] td.row-buttons button

Obrázek 24 - Příklad neefektivity nástroje pro testování tabulkové komponenty (David Pfeifer, 2015)

8.3 Definice požadavků a cílů

V předchozích dvou kapitolách byla představena kompletní architektura systému a klíčové komponenty, ze kterých se skládá. Podle požadavků a důležitosti jednotlivých komponent, by se testy měly zaměřit mandatorně na integraci Back Office s ostatními částmi systému. Také je důležité pokrýt Online Service, tzn. frontendovou část aplikace.

Dále byl analyzován výchozí stav automatizovaných testů na projektu, na jehož základě byl identifikován prostor pro rozšíření druhé úrovně o sadu integračních a API testů. Zároveň byla prozkoumána oblast GUI testů (3 úroveň dle pyramidy), kde by bylo vhodné nahradit stávající metodu testování sofistikovanějším nástrojem. Aktuální metoda není dle zjištěných nedostatků ideální.

S ohledem na výše zmíněná zjištění byly identifikovány dvě kategorie automatizovaných testů, pro které budou stanoveny v následujících podkapitolách konkrétní cíle. Definované cíle budou složit jako vstupy pro výběr nástroje, samotný návrh a implementaci.

8.3.1 Automatizované integrační testy

Pro QC tým je to první kategorie testů, jejichž přípravou by se měl zabývat. Více informací o této kategorii lze nalézt v kapitole 4.4.3. Pro potřeby projektu NBS by měly integrační testy pokrývat následující funkcionality.

- Úvodní test kondice, konfigurace a integrace prostředí včetně ověření běžících služeb. Ověřování pomocí exekuce databázových dotazů. Podmínka pro spuštění dalších sad testů.
- Oblast strukturálních operací (založení, tvorba, odebrání).
- Oblast reportingu zahrnující uložení hodnot pomocí interních datových toků.

8.3.2 Automatizované testy uživatelského rozhraní

Projekt NBS je dlouhodobý a pravidelně jsou implementovány nové funkcionality, které mohou ovlivnit stávající implementaci. To samé můžeme říct o integraci nových verzí platformy Damas. To vše klade velké nároky na opakovanou exekuci regresních testů v rámci FAT, více o FAT v kapitole 4.4.4. S nasazením každé nové verze je také třeba počítat se smoke testy, o kterých bylo psáno v rámci kapitoly 4.4.6.

Systém NBS poskytuje širokou škálu funkcionalit, proto by bylo téměř nemožné nebo minimálně časově náročné všechny regresní testy vykonávat manuálně. Sada regresních testů pro každé FAT obsahuje kolem 500 testovacích případů. Počet testů narůstá s každou nově implementovanou funkcionalitou. Proto je důležité vybrat nástroj, který umožní postupné a jednoduché vytváření nových automatizovaných testů a zároveň přehlednou a efektivní údržbu. To vše s podporou centrální evidence. Budoucí testy by měly pokrýt následující oblasti.

- Všechny testovací případy patřící mezi smoke testy.
- Veškeré pohledy a funkcionality poskytované v rámci frontendové webové aplikace.
- Základní verifikace pohledů a funkcionalit v rámci backendu.
- Pokrytí co největší části regresních testů.

9 Výběr nástrojů pro automatizaci

Výběr vhodného nástroje jsem se rozhodl nepodcenit a je mu věnována celá kapitola. V případě, že by se v průběhu času ukázalo, že nástroj je nedostačující, bývá přechod na jiný složitým a časově náročným procesem. Proto by analýza a následný výběr jednotlivých nástrojů měly být obzvlášť pečlivě provedeny. Výstupem procesu volby je nástroj, který bude použit v dalších kapitolách pro tvorbu integračních a GUI testů.

Pro výběr jsem se rozhodl využít komparativní analýzu s využitím tabulky kancelářského nástroje MS Excel. Do porovnání jsou zahrnuty tři nástroje představené v kapitole 5.4:

- Gauge
- JMeter
- Cypress

Gauge a JMeter reprezentují nástroje pro automatizaci testů založené na využití Selenium Web Driver. Aby nebyl výběr postaven pouze na nástrojích využívající Selenium, byl přidán do výběru právě Cypress. Cypress ztělesňuje moderní alternativu ke klasickým nástrojům, je to „all in one“ framework, který není postaven na bázi Selenia a nepotřebuje instalaci dalších knihoven a doplňků. Jeho integrace do testovacího procesu je tedy velmi rychlá. Nemožnost rozšíření na druhou stranu limituje uživatele pouze na koncové testy uživatelského rozhraní a na prohlížeče s jádrem postaveným na platformě Chrome.

Analýza je postavená na hodnotících kritériích, která byla stanovena v souladu s cíli definovanými v předchozí kapitole, projektovými požadavky a zkušenostmi autora v oblasti automatizace a vývoje softwaru. Jednotlivá kritéria jsou zařazena do dílčích oblastí rozdělených podle fáze testovacího procesu, jeho specifických potřeb a celkové technologické podpoře. Na závěr je výzkum podpořen osobní uživatelskou zkušeností s jednotlivými nástroji. Každý požadavek je v rámci daného nástroje vyhodnocen buď pozitivně nebo negativně. Kompletní a detailní vypracování komparativní analýzy je součástí přílohy A. Na základě výsledků jsou nástroje dále podrobeny SWOT analýze.

Jednotlivá hodnotící kritéria jsou stručně představena a charakterizována v následující kapitole.

9.1 Hodnotící kritéria

Typy tesů:

- Integrační testy.
- Uživatelské koncové testy.
- Testování API a webových služeb.

Tvorba automatizovaných testů, která zahrnuje následující kritéria:

- Tvorba testovacích případů.
- Tvorba jednotlivých testovacích kroků.
- Členění testů na úrovně – v případě, že mandatorní část testu skončí chybou, skončí celý test, naopak v případě selhání dílčí části test pokračuje.
- Tvorba selektorů – velká většina testů grafického rozhraní se zakládá na identifikaci elementů na stránce a jejich použití.
- Centrální evidence stepů – seznam všech testovacích kroků přehledně na jednom místě, ideálně průběžně aktualizovaný.
- Znalost programování – při tvorbě testů některé nástroje vyžadují znalost některého programovacího jazyka, u jiných stačí pouze využití jakékoli formy značkovacího jazyka.
- Integrované testovací kroky – netřeba dodatečná implementace, nástroj má integrovanou knihovnu základních kroků, například načtení webové stránky, přihlášení, klikání a ověřování elementů.
- Možnost odchycení chyb – důležité v začátcích integrace automatizovaného testu.

Spuštění:

- Lokální spuštění – přímo z pracovní stanice testera.
- Podpora integrace s CI/CD nástroji – např. integrace automatizovaných testů do nástroje TeamCity, umožňující automatizované spouštění a vyhodnocování na vzdáleném serveru.
- Možnost rozšíření – o další doplňky, například podpora prohlížečů nebo využití Selenium Grid atd.
- Nutnost rozšíření – hodnotí, jestli je pro integraci a přípravu testů potřeba instalace dalších doplňků. Zde je rozdíl oproti ostatním kritériím, v případě potřeby rozšíření je hodnocen negativně.

Reporting:

- Lokální grafický report – vyhodnocuje dostupnost grafických reportů po manuální exekuci testů.
- Vzdálený grafický report – dostupnost reportů v případě exekuce na vzdáleném serveru.
- Snímky obrazovky – například vytvoření snímku obrazovky při objevení chyby. Může sloužit i jako konfirmační potvrzení v případě pozitivního výsledku testu.
- Videa – zachycení video záznamu celého testu a jeho uložení.

Načítání dat z externích zdrojů:

- Microsoft Excel – typicky představuje zdroj vstupních dat pro vykonávané testy.
- XML – na projektu NBS se využívá v případě zasílání testovací zprávy přes webovou aplikaci.
- Tabulka – použití tabulky se vstupními daty integrovanou přímo v testu.

Stahování a vyhodnocení souborů: stejná kritéria jako v předchozí oblasti, ale v opačném směru. Příkladem může být export dat z pohledu a ověření obsahu staženého dokumentu.

Podpora exekuce testů na nejpoužívanějších webových prohlížečích Firefox, Chrome a Internet Explorer.

Oblast technologie se zaměřuje na možnost integrace se Seleniem a Selenium Server. Dále na podporu nejznámějších programovacích a skriptovacích jazyků C#, Java, Javascript, Python a Ruby.

Platforma podpora a uživatelská zkušenost:

- Platforma, na které lze testy spouštět – Linux, Windows a Macintosh.
- Technická podpora – jestli je dostupná uživatelská podpora poskytovatelem.
- Komunita – v podobě různých diskusních fór a skupin.
- JIRA integrace – možnost integrace nástroje na nástroj pro podporu řízení defektů.
- Uživatelská zkušenost – osobní preference autora na základě práce s jednotlivými nástroji.

9.2 SWOT analýza nástrojů

Detailní poznání a ohodnocení jednotlivých nástrojů pro automatizaci z funkčního a technologického hlediska umožňuje provézt další analýzu. Přesněji řečeno SWOT analýzu, která produkt zhodnocuje z pohledu vnitřních i vnějších faktorů. Její podstatou je identifikace silných a slabých stránek uvnitř produktu. Neméně významný je vnější faktor zahrnující příležitosti využití na projektech a hrozby například v podobě konkurenčních nástrojů.

9.2.1 Gauge

Silné stránky: Největší síla Gauge frameworku spočívá bez debat v možnosti tvorby testovacích skriptů ve formě obchodního jazyka, který je srozumitelný pro všechny zúčastněné strany. Seznam podporovaných skriptů je centrálně dostupný a jednoduše se s ním pracuje. Jednoduchá tvorba a údržba testů, kterou zvládne téměř kdokoliv. Oceňuji možnost použití značek a více úrovní v testech. Podporuje všechny dostupné plat-

formy a programovací jazyky, takže jeho zavedení do systému zvládne vývojář se znalostí jakéhokoli jazyka. Je zdarma a open source. Dále umožňuje jednoduché propojení s CI/CD nástroji.

Slabé stránky: Mezi největší slabiny patří nemožnost ověření obsahu stažených souborů. V tomto případě jsme nuceni exportování dat verifikovat manuální cestou nebo s využitím jiného nástroje.

Příležitosti: Nástroj nabízí možnosti pro tvorbu integračních i GUI testů. Z mého pohledu je vhodný pro testování na velkých dlouhodobých projektech, kde je kladen důraz na přehledné a pěkně zpracované reporty. Ve spolupráci se Selenium Server a Grid si poradí s exekucí velkého množství testů na různých prohlížečích a prostředích.

Hrozby: Než je nástroj možné efektivně používat, je nutná celkem rozsáhlá příprava. Forma zápisu testů pomocí obchodního jazyka je podmíněna implementací podpory pro veškeré testovací skripty. V případě zavedení nových „větiček“ je nutná spolupráce testera a vývojáře. Není tak všestranný a nenabízí takové množství doplňků jako JMeter.

9.2.2 JMeter

Silné stránky: Nejsilnější stránkou JMeteru je možnost rozšíření o nepřeberné množství doplňků. Umožňuje čtení, vyhodnocení i zápis do externích souborů, například do csv. Nemá vyloženě integrované testovací skripty, ale obsahuje spoustu předpřipravených metod, které se například v rámci Gauge musí implementovat. Dále podporuje nahrávání testů s využitím Selenium IDE apod. Je zdarma.

Slabé stránky: Slabinou JMeteru je podpora a orientace na jediný programovací jazyk, kterým je Java. Občas má také problémy s podporou novějších verzí internetových prohlížečů. Bez využití dalších doplňků nedisponuje úhlednými reporty, jako například Gauge.

Příležitosti: Vhodný do projektového týmu, který disponuje zkušenějšími testery se znalostí programovacího jazyka Java. V tomto případě není nutná další asistence vývojáře, tester si vše připravuje a organizuje sám. V případě potřeby rozšíření testů o výkonnostní, zátěžové, databázové, API apod. není nutné hledat nový nástroj. JMeter je

unikátní ve své univerzálnosti a tím pádem se hodí pro potřeby jakéhokoli projektu a typu testů.

Hrozby: Jeho všestrannost mu může paradoxně uškodit v případě, že projekt hledá framework pouze pro určitý typ testů. Existují jiné nástroje, které zvládají pouze jednu disciplínu, ale lépe než JMeter.

9.2.3 Cypress

Silné stránky: Nástroj Cypress se od předešlých dvou znatelně odlišuje. Díky tomu, že obsahuje všechny potřebné součásti pro celý proces automatizace, není vyžadována instalace dalších doplňků. Unikátní rozhraní pro tvorbu testů umožňuje automatické generování selektorů, i když ne vždy efektivní. Tester má v nástroji k dispozici debugger, obrazovku, kde vidí, co test zrovna dělá a integrovanou funkcionalitu pro prozkoumání prvku na webu. Cypress obsahuje předpřipravené metody pro základní uživatelské operace na webu, stačí pouze doplnit automaticky vygenerovaný selektor a jednoduchý test je vytvořen. Je možné dle libosti vytvářet nové a upravovat stávající metody pomocí Javascriptu.

Slabé stránky: Slabinou nástroje je omezená základní podpora webových prohlížečů založená pouze na jádru Chrome. Firefox lze použít pouze v docker s využitím GitLabu. Druhou znatelnou nevýhodu představuje nutnost registrace a používání služeb GitHubu, v případě rozhodnutí pro CI/CD a vzdálené spuštění. Veškeré testovací reporty jsou nahrávané pouze na GitHub a není možné zvolit jinou možnost. To může představovat nepřekonatelnou překážku pro projekty s přísnými bezpečnostními pravidly, kde je nutné využívat pouze určité služby.

Příležitosti: Vhodný pro projekt, který shání alternativu k frameworkům využívající Selenium. Osobně si umím představit využití Cypress na menším projektu, založeném na agilní metodice vývoje, kde je kladen důraz na rychlost a efektivitu testování. Integrace cypress do testovacího procesu už nemůže být jednodušší, vše je připravené. Minimální technické nároky na testera. Nic není potřeba zdlouhavě konfigurovat, nástroj je připravený k téměř okamžitému použití.

Hrozby: Téměř nulová rozšiřitelnost o další doplňky. Zaměření pouze na koncové testy uživatelského rozhraní. Nutnost využívání GitHub a GitLab. V případě, že základní knihovna testovacích kroků nestačí, je pro další rozšiřování vyžadována pokročilá znalost Javascriptu. Oproti konkurenci je nástroj placený. Základní projektové předplatné se pohybuje od 75$ do 300$ za měsíc.

9.3 Vyhodnocení a volba nástroje

Na základě analýzy a vyhodnocení dle stanovených kritérií se nejlépe vedlo nástroji Gauge, který podporuje celkem 35 požadavků. Nejlépe si vedl také v oblasti uživatelských zkušeností, kde získal celkem 5 bodů. Jediné slabé místo bylo vypozorováno v rámci stahování a kontroly obsahu souborů. Druhé místo těsně obsadil multifunkční nástroj JMeter, který si vedl lépe v nabízených doplňcích a funkcionalitách, ale ve výsledku ztratil v oblasti podporovaných technologií a požadované znalosti programovacího jazyka Java. Na posledním místě se umístil Cypress, který vyniká ve své kompaktnosti, jednoduchosti a unikátnímu rozhraní, na druhou stranu je zaměřený pouze na jeden typ testů a cena měsíční licence je poměrně vysoká. Nutnost kooperace s GitHub také nemusí všem vyhovovat.

Tabulka 2 - Umístění nástrojů (David Pfeifer, 2021)

Nástroj	Počet pozitivně hodnocených kritérií	Umístění
Gauge	40 (35+5)	1
Jmeter	37 (34+4)	2
Cypress	33 (30+3)	3

Z navazující SWOT analýzy bylo zjištěno, že pro potřeby projektu Nordic Balance Settlement bude nejlepší využít framework Gauge. Podporuje oba požadované typy testů a napojení na nástroj TeamCity. To vše je doplněno o krásné a přehledně vygenerované reporty. Dalším pomyslným závažím na misce vah byla podpora technologie .NET, která je na projektu používána nejvíce. Důležitá je také nabídka testování více prohlížečů, jelikož aplikace musí podporovat všechny nejpoužívanější na trhu. Díky markdownu jsou testovací skripty srozumitelné a mohou být vytvářeny bez znalosti programování.

JMeter je ve výsledku nejlepší volbou pro pokročilé uživatele, se znalostí programovacího jazyka Java, kteří hledají jediný nástroj na všechny své testy a nevadí jim nutnost instalace a konfigurace doplňků.

Cypress je v oblasti automatizovaných testů jedinečným řešením. S ohledem na bezpečnostní standardy společnosti Unicorn by ho kvůli propojení s GitHubem nebylo možné na reálném projektu použít. Dále nepodporuje všechny webové prohlížeče. Závěrem nesplňuje ani základní podmínku této práce, kterou představuje možnost tvorby integračních testů. Nicméně své fanoušky a uživatele si na trhu stoprocentně najde.

10 Návrh, implementace a aplikace řešení

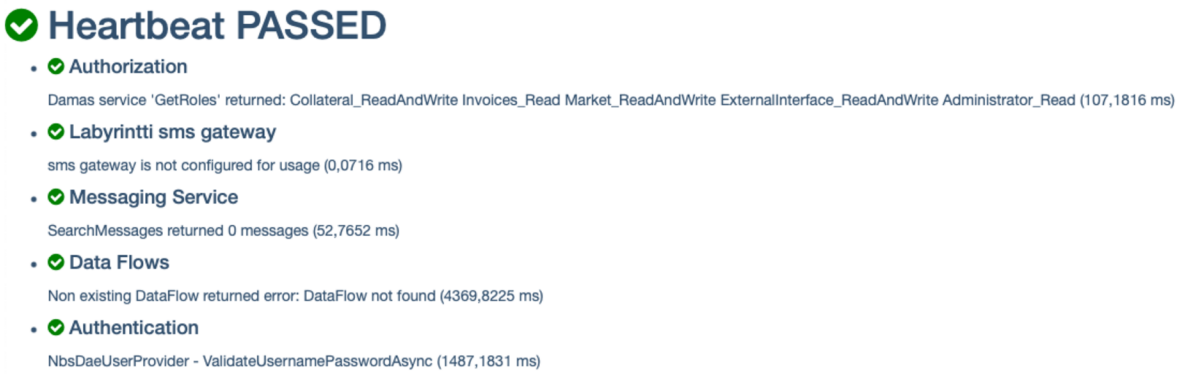
10.1 Příprava a kontrola testovacího prostředí

Pro efektivní návrh a exekuci automatizovaného, ale i manuálního testu je podstatné mít správně nakonfigurované a připravené testovací prostředí. S testovacím prostředím úzce souvisí testovací data a jejich příprava. Tato kapitola představí základní aspekty, které je třeba na prostředí pravidelně kontrolovat a také postup pro přípravu testovacích dat.

Testovací prostředí by mělo mít podobně rozvržené a propojené komponenty, jako cílové produkční prostředí. To umožňuje včasné odhalení problému v rámci integrace. Pokud by nějaká z komponent na testovacím prostředí byla vynechána, nebude možné danou oblast efektivně otestovat. Jediné, s čím se počítá, že bude trochu odlišné, je hardwarová konfigurace, interní testovací prostředí s ohledem na menší vytíženost nebývají dimenzována tak, jako například produkční prostředí nebo prostředí navržené pro výkonnostní testy. Pro ověření základních funkcionalit to nicméně nehraje zásadní roli.

V rámci NBS je před exekucí testů potřeba ověřit následující oblasti:

1. Jsou spuštěny všechny dlouhotrvající procesy, které se starají o úspěšný chod systému.
2. Všechny procesy ve frontě jsou úspěšně odbavovány.
3. Každá exekuční jednotka na daném serveru má přiřazenou skupinu procesů, kterou vykonává.
4. Online Service a Back Office jsou správně integrovány na Messaging Service.
5. Výpočetní modul běží pro správný interval.
6. Různé SQL dotazy kontrolující správnou datovou integritu a správné stavy v procesním řízení.
7. Přihlášení se na tzv. „Heartbeat“ stránku a kontrola autorizace, sms brány, integrace a autentizace. Jak může taková stránka vypadat, je znázorněno níže.



Obrázek 25 - Ukázka Heartbeat kontroly (NBS Online Service Private, 2021)

Všechny výše zmíněné oblasti verifikace lze automatizovat s využitím nástroje Gauge.

Druhý přípravný krok zahrnuje testovací data. Před zahájením jakýchkoli testů nové verze systému, je ideální mít takzvanou čistou databázi, obsahující referenční testovací data neovlivněná již proběhlými testy. Jen to účinně zajistí, že strukturální data, která test očekává nebyla ovlivněna nebo změněna jinými testy.

Testovací data by měla být vždy správná, konzistentní a přehledná. Z osobní zkušenosti doporučuji zakládat a připravovat potřebná data s delší časovou platností. Nikdy s určitostí není známo, jak dlouho bude takový projekt vyvíjen. Lepší založit data s delší časovou platností, než analyzovat a řešit popadané testy z důvodu prošlých dat. Zpětné narovnání platností je pak daleko náročnější. Nejspolehlivějšími testy s ohledem na konzistenci a znovupoužitelnost bývají takové, které si zakládají v rámci exekuce data vlastní. Ne vždy je to ale nejefektivnější řešení. Vlastní příprava dat může v určitých případech zabrat více času, než je pro průběh testu zdrávo, v takové situaci je lepší pokusit se využít stávající data. Kompromisem pak bývají testy, které si na základě referenčních dat připravují pouze ta nejnutnější data pro úspěšné provedení testu. Dalším vhodným řešením je oddělení testovací sady dat pro automatizované testy od dat pro ty manuální.

10.2 Identifikace testovacích případů pro automatizaci

Úvodem této kapitoly bych rád zopakoval základní pravidla, podle kterých by testy vhodné pro automatizování měly být vybírány. Nejprve by měly být automatizovány základní funkcionality. Je vhodné si definovat celou funkční oblast, která bude automatizována a tu kompletně dokončit, před puštěním se do automatizace další oblasti. To z jediného důvodu, testy v jedné oblasti budou s největší pravděpodobností využívat

podobné testovací skripty, tím pádem je bude možné s jednoduchou změnou proměnných znovupoužít bez další potřebné implementace nových skriptů. Automatizovaný test by měl být vždy opakovatelný, jinak jeho zařazení postrádá smysl. Jeho vytvoření by stálo více času než manuální provedení. Úspora času a efektivita je tvořena až po několikátém automatizovaném spuštění. Poslední doporučení se týká obsáhlosti testu, automatizovaný test by v ideálním případě neměl být moc komplikovaný a dlouhý. Exekuce celé sady testů se pak zbytečně protahuje a test je také více náchylný k pádům. Pokud to situace dovolí, je lepší takový test rozdělit na dílčí nezávislé celky a ty pustit samostatně například ve více vláknech. To celý proces testování urychlí.

Pro návrh a implementaci automatizovaných testů byly pro jednotlivé kategorie definovány následující reprezentativní příklady nejběžnějších operací a scénářů.

Kapitola zabývající se kategorií integračních testů začíná vzorovým příkladem, jak lze vytvořit základní test prostředí, který ověřuje, že běží všechny výpočetní moduly. To je docíleno použitím jednoduchého SQL dotazu. Následuje příklad testu uložení hodnot pro bilaterální obchody za obě strany, který verifikuje nejen správnou funkcionalitu interního datového toku, ale také následné automatizované přepočítání časové řady „Matched Quantity Status“. Tato časová řada poskytuje uživateli informace, zda se reportované hodnoty obou BRP shodují nebo nikoli. Poslední test, reprezentující integrační úroveň testů, zavádí do systému novou Produkční jednotku včetně veškerých povinných atributů.

Část práce zabývající se implementací testů uživatelského rozhraní obsahuje příklad automatizace regresního testovacího případu pro založení Produkční jednotky a její přiřazení konkrétnímu obchodníkovi. Automatizovaný test počká, než se vygeneruje událost potvrzující založení dané jednotky a následně ověří její korektní zobrazení ve webové aplikaci. Celý test probíhá v rámci Online Service a je vytvořen v souladu s postupem manuálního testu uvedeným v JIRA. Náhled testovacího případu v nástroji JIRA je součástí přílohy B. Závěr kapitoly je věnován ukázce zapracování integračního testu jako prostředku pro založení testovacích dat pro potřeby testu uživatelského rozhraní.

10.3 Implementace integračních testů

Návrh testovacího skriptu, s využitím značkovacího jazyka, je velmi jednoduchý. Jediné, co tester potřebuje, je nástroj pro práci s textem. Dokonce si vystačí i s obyčejným

textovým editorem. Nejpoužívanější nástroj bývá Visual Code, já osobně používám IntelliJ IDEA, který nabízí další pokročilé funkce, včetně hromadného vyhledávání přes všechny soubory a podporu různých dalších skriptovacích jazyků. Na začátek je potřeba si vytvořit soubor s příponou .md. Cíl testu je vhodné uvést jako název specifikace za jeho označení #. Druhým krokem je uvedení značek, podle kterých se testy mohou třídit do jednotlivých kategorií. Zde stačí napsat „Tags:“ a doplnit značku. Lze uvézt více značek a oddělit je čárkou. První test využívá pouze SQL dotaz, proto je důležité mít v implementaci Gauge podporu pro připojení do databáze, odeslání dotazu a vyhodnocení. Podporu takového skriptu lze jednoduše ověřit v knihovně příkazů, ve které podporované skripty pro SQL vypadají následovně. Příklady dalších příkazů v knihovně pro integrační testy jsou uvedeny v příloze C.

```
"VerifySql","Verify that SQL <sql> result is <result>"
"VerifySql","Verify that SQL <sql> result is not <result>"
"VerifySql","Verify that SQL <sql> results are <result>"
"VerifySql","Print result of SQL <sql>"
```

Obrázek 26 - SQL testovací skripty (NBS, 2021)

Na základě podporovaných skriptů z knihovny je třeba dotaz složit tak, aby vrátil výsledný počet řádků. Dotaz je vložen v rámci testovacího skriptu do proměnné <sql> a očekávaná hodnota dotazu do proměnné <result>. V rámci tabulky výpočetního modulu je možné využít sloupec TS88DISABLED, který je v případě běžícího modulu obsahuje hodnotu 0, viz následující obrázek z SQL Developeru.

Worksheet | Query Builder

```
select * from TS88SETTLEMENT_CALC_ENGINE where TS88DISABLED <> 1;
```

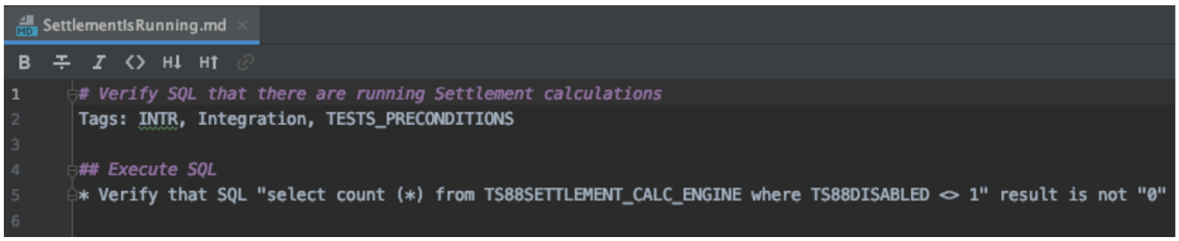
Query Result | SQL | All Rows Fetched: 10 in 0,026 seconds

	TS88DISABLED	TS88ENGINE_ID	TS88BUSINESS_FROM	TS88BUSINESS_TO	TS88STATE	TS88LAST_CALCULATION
1	0	SW02	04.04.21 22:00:00,000000000	05.04.21 22:00:00,000000000	RUNNING	04.04.21 11:07:08,580000000
2	0	SW03	03.04.21 22:00:00,000000000	04.04.21 22:00:00,000000000	RUNNING	04.04.21 11:03:54,580000000
3	0	SW04	02.04.21 22:00:00,000000000	03.04.21 22:00:00,000000000	RUNNING	04.04.21 11:03:54,846000000
4	0	SW05	01.04.21 22:00:00,000000000	02.04.21 22:00:00,000000000	RUNNING	04.04.21 11:05:34,007000000
5	0	SW01	31.03.21 22:00:00,000000000	01.04.21 22:00:00,000000000	RUNNING	04.04.21 11:07:27,843000000
6	0	CRW01	31.03.21 22:00:00,000000000	01.04.21 22:00:00,000000000	RUNNING	04.04.21 11:06:28,169000000
7	0	CRW02	04.04.21 22:00:00,000000000	05.04.21 22:00:00,000000000	RUNNING	04.04.21 11:04:02,243000000
8	0	CRW03	03.04.21 22:00:00,000000000	04.04.21 22:00:00,000000000	RUNNING	04.04.21 11:03:45,701000000
9	0	CRW04	02.04.21 22:00:00,000000000	03.04.21 22:00:00,000000000	RUNNING	04.04.21 11:08:03,066000000
10	0	CRW05	01.04.21 22:00:00,000000000	02.04.21 22:00:00,000000000	RUNNING	04.04.21 11:05:07,275000000

Obrázek 27 – Dotaz do tabulky výpočetního modulu (David Pfeifer, 2021)

Pokud by kalkulační procesy nefungovaly, byly pozastavené nebo jen nebyly přiřazeny žádné exekuční jednotce, jejich hodnota TS88DISABLED by byla 1. Testovací skript je

v rámci specifikace označen vždy * a je vytvořen s ohledem na uvedená fakta následovně.



Obrázek 28 - Testovací skript pro ověření výpočetního modulu (David Pfeifer, 2021)

V případě, že vše funguje správně, dotaz vrátí v tomto konkrétním případě hodnotu 10 a test bude vyhodnocen pozitivně. V opačné situaci, ve které nefunguje výpočetní modul, by dotaz vrátil hodnotu 0 a test byl vyhodnocen negativně. Takto napsaný test je připraven pro exekuci a přidání do sady základních testů pro verifikaci prostředí.

Druhý integrační test je zaměřen na verifikaci interního datového toku pro uložení hodnot bilaterálního obchodu. Účelem testu je autentizace do systému v podobě BRP, tvorba xml souboru, obsahující vstupní data z tabulky, který je odeslán interním datovým tokem. Test pokračuje ověřením správně uložených hodnot dotazem na zdroj dat, který využívá Online Service pro zobrazování na svých pohledech. To samé se opakuje za protistranu bilaterálního obchodu tzn. druhé BRP. Na závěr je provedena verifikace hodnot uložených protistranou spojená s ověřením správně přepočítaných statusů v časové řadě „Matched Quantity Status“. Pro účely tohoto testu bude použit scénář řízený tabulkou[40]. Obdobně, jako v případu minulého testu, je nezbytné podporu pro skript používající tabulkovou komponentu naimplementovat v rámci řešení Gauge. V tomto případě jsou testovací skripty pro uložení a verifikaci hodnot navrženy takto:

```
"BilateralTrade","Set Bilateral Trade hourly values for <fromTo> and
<ownBrpName> and <counterBrpName> and <mbaName> and <ownReName> and
<counterReName> and <aggreementId> <newValues>"

"BilateralTrade","Verify Bilateral Trade hourly values for <fromTo> and
<ownBrpName> and <counterBrpName> and <mbaName> and <ownReName> and
<counterReName> and <aggreementId> <expectedData>"
```

[40] Table Driven Scenario - https://docs.gauge.org/writing-specifications.html?os=macos&language=javascript&ide=vscode

Bilateral trade označuje název testovací třídy, samotný skript je složen z unikátní kombinace více proměnných definující konkrétní datovou strukturu v systému. Tato struktura je definována kombinací dvou BRP, MBA[41], ve které obchod probíhá, zúčastněnými RE a unikátním id obchodu. Proměnné na konci uvedených skriptů <newValues> a <expectedData> mohou být představovány tabulkou. Test vychází z reálného testovacího scénáře a používá existující datovou strukturu založenou na testovacích datech. Finální podoba testu je uvedena na následujícím obrázku.

BASSE-39553 – Edit Bilateral Trade values (INT).md

```
# BASSE-39553 – Edit Bilateral Trade values
Tags: Inputs, SmokeTest, INTW, EditMecValues, DP

## Edit Bilateral Trades hourly values
* Login as "Admin_Company026" under "BRP" "BRP10"

* Set Bilateral Trade hourly values for "D+1" and "BRP10" and "BRP12" and "NO4" and "RE13" and "RE20" and "3"
| Hour | Own Quantity [MWh] |
|------|--------------------|
| 1    |  $1[-10000,1000]   |
| 12   |  $12[-1000,1000]   |
| 24   |  $24[-1000,1000]   |

* Verify Bilateral Trade hourly values for "D+1" and "BRP10" and "BRP12" and "NO4" and "RE13" and "RE20" and "3"
| Hour | Own Quantity [MWh] |
|------|--------------------|
| 1    |  $1                |
| 12   |  $12               |
| 24   |  $24               |

* Login as "Admin_Company027" under "BRP" "BRP12"

* Set Bilateral Trade hourly values for "D+1" and "BRP12" and "BRP10" and "NO4" and "RE20" and "RE13" and "3"
| Hour | Own Quantity [MWh] |
|------|--------------------|
| 1    |  -$1               |
| 12   |  -$12              |
| 24   |  $24               |

* Verify Bilateral Trade hourly values for "D+1" and "BRP12" and "BRP10" and "NO4" and "RE20" and "RE13" and "3"
| Hour | Own Quantity [MWh] | Matched Quantity Status |
|------|--------------------|-------------------------|
| 1    |  -$1               | Matched                 |
| 12   |  -$12              | Matched                 |
| 24   |  $24               | Corrected               |
```

Obrázek 29 - Testovací skript pro ověření funkcionality Bilaterálního obchodu (David Pfeifer, 2021)

Celý soubor obsahuje již zmíněný název specifikace. Nově byl přidán i název konkrétního scénáře uvedený za ##. Do proměnných je již doplněna konkrétní kombinace hodnot. Zde si lze všimnou, že v případě protistrany se atributy BRP a RE otáčejí.

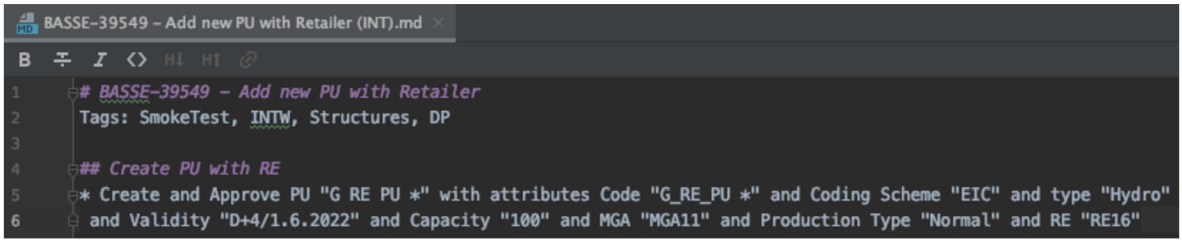
[41] Market Balance Area

Tabulka je složena ze 2-3 sloupců, každý sloupec obsahuje stejný název, jako lze nalézt přes grafický interface webové aplikace. Číslo v tabulce definuje pořadí hodiny v daném dni, který je editován. Kvantita je doplněna s využitím náhodně generované hodnoty. Náhodně generované hodnoty jsou tvořeny zápisem *„$_číslo(_znak)_[výčet hodnot]“.*

- $ – znak označující proměnou
- Číslo – číslo proměnné, může být i název atd.
- Znak – v případě více sloupců v tabulce je použit znak pro identifikaci sloupce
- Výčet hodnot – hodnoty jsou oddělené čárkou a mohou být číselné nebo textové, zmíněné datové typy nelze kombinovat.
    - Číselný rozsah mezi dvěma čísly např. [1,100]. Textový např. [Metered,Estimated,Temporary].

Pro zobrazení statusu „matched“ musí být hodnoty uložené protistranou stejné s opačným znaménkem (první dva řádky). V jakémkoli jiném případě se zobrazí status „corrected“ (poslední testovací řádek). Z příkladu lze vidět, že test vyplní hodinové hodnoty náhodně vygenerovanou proměnou z intervalu [-1000,1000]. Za protistranu jsou doplněna stejná čísla s různými znaménky podle toho, jaký status je ověřován. Hodnoty v poslední tabulce jsou porovnávány s databázovou tabulkou a ověřují, jestli editace proběhla úspěšně a očekáváné hodnoty včetně statusu se shodují.

Posledním zástupcem integračního testování je testovací skript pro založení Produkční jednotky interním datovým tokem, který je používán v rámci komunikace Online Service a Back Office. Struktura a postup návrhu integračních testů již byla dostatečně představena v přechozích dvou případech. Připravený testovací skript lze prozkoumat na obrázku níže.



Obrázek 30 - Testovací skript pro založení Produkční jednotky interním datovým tokem (David Pfeifer, 2021)

Test opět obsahuje povinné označení specifikace, značky a je doplněn o název scénáře. Testovací skript vychází z knihovny podporovaných operací a na obrázku byl rozdělen do dvou řádků s ohledem na svoji délku. V tomto případě je důležité zmínit, že každý testovací skript musí ležet pouze na jednom řádku, jinak si s tím nástroj neumí poradit a skript nebude úspěšně exekvován. Vytvářená produkční jednotka obsahuje veškeré povinné atributy s ohledem na obchodní specifikaci: název, kód, schéma, typ, platnost, maximální výrobní kapacitu, produkční typ a obchodníka. V tomto případě je obsah práce detailněji zaměřen na doplnění tří specifických atributů. Na obrázku si lze všimnout použití znaku * pro jméno a kód. Systém NBS má na základě obchodních požadavků implementované validace na unikátnost kombinace názvů a kódu produkčních jednotek. S ohledem na znovupoužitelnost testů musí být hodnota těchto atributů vždy jedinečná. Znak * reprezentuje metodu pro doplnění náhodně vygenerované časové řady čísel. Další proměnná je v testu použita na místě, kde je vyplňována validita. Systém má implementovanou validaci, od kdy je možné zakládat nové produkční jednotky. Uzávěrka se tedy s každým uplynulým dnem mění. S momentální konfigurací systému je možné produkční jednotky zakládat od čtvrtého dne po aktuálním dni. Aby nebylo nutné před každým vykonáním testu hodnotu platnosti měnit v závislosti na momentálním dni, je zastoupena proměnou D, která reprezentuje aktuální den a k té se přičítá hodnota za znaménkem +.

Testy lze v rámci procesu návrhu postupně ladit. K tomu se využívá možnost lokálního spouštění a vyhodnocení konkrétního testu nebo sady testů přes příkazový řádek v PowerShell[42]. Spouštěcí skript může vypadat následovně.

```
.\run.ps1 -TestEnvironment TEST -specsPath c:\basse-qc\gauge-tests\in-
frastructure\BASSE-39553 - Edit Bilateral Trade values.md
```

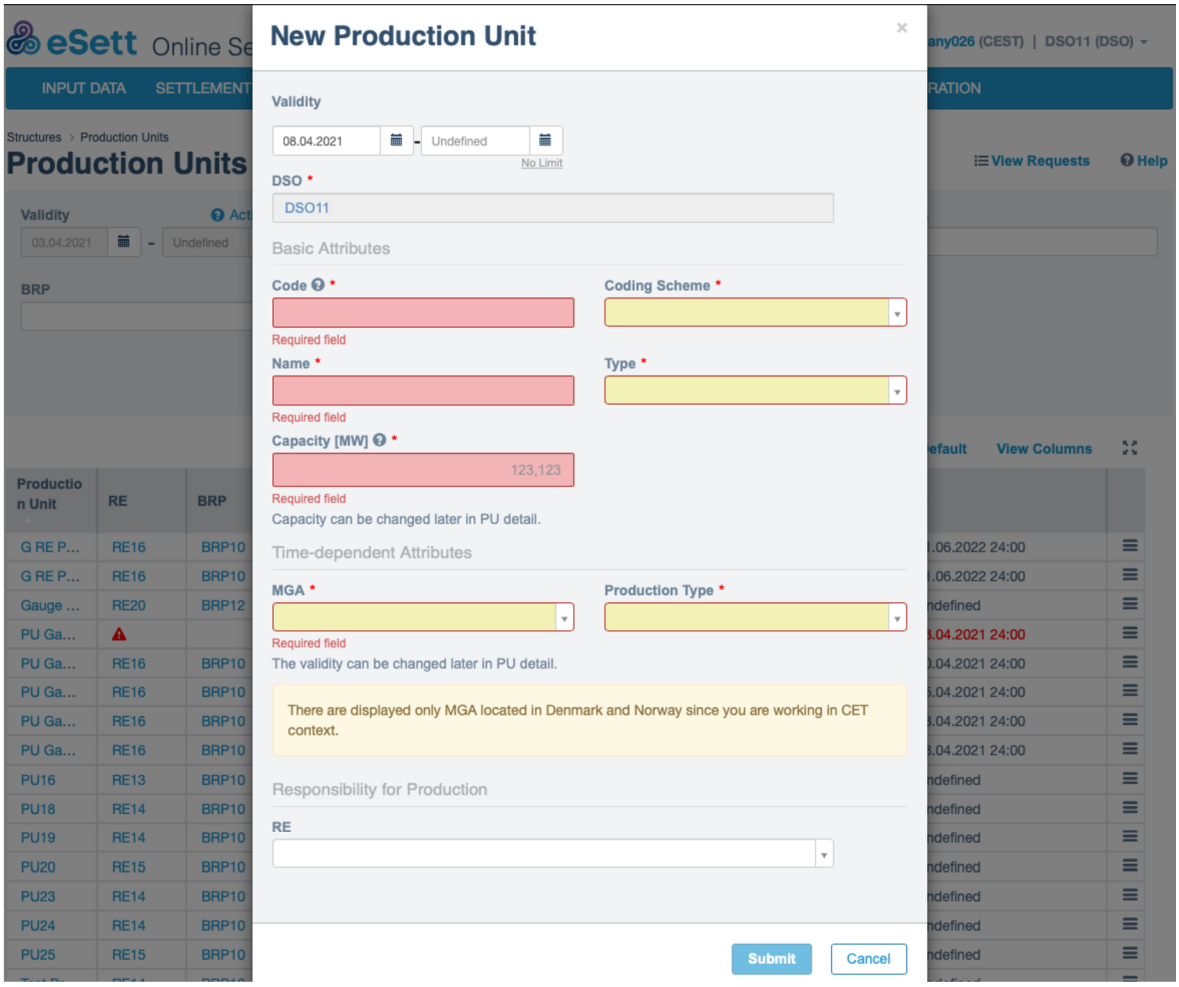
[42] https://docs.microsoft.com/en-us/powershell/scripting/overview?view=powershell-7.1

10.4 Implementace GUI testu

Testy uživatelského rozhraní se od integračních odlišují tím, že netestují pouze interní komunikaci a integraci, ale otevírají přímo okno příslušného webového prohlížeče a přesně simulují všechny procesy uživatele. Takto vytvořené automatizované testy umožňují ohromnou úsporu času ve fázích FAT zaměřených na regresní testy. GUI testy jsou navrhnuté tak, aby pokryly veškeré kroky daného testovacího scénáře.

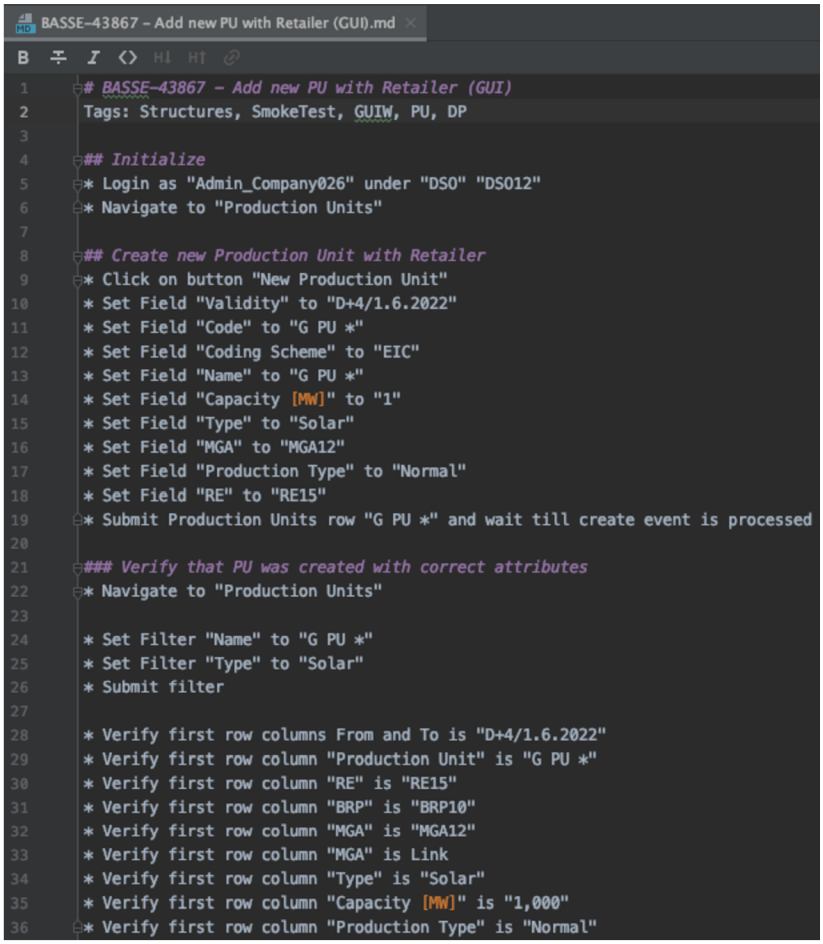
Cílem této kapitoly je návrh automatizovaného testu pro přidání produkční jednotky v rámci dané MGA[43], včetně přiřazení obchodníka. Test je vykonáván v kontextu přihlášeného DSO uživatele v rámci Online Service. Očekávaným výstupem testu je založená produkční jednotka se všemi vyplněnými atributy. Předpoklady pro úspěšné vykonání testu je zavedené DSO s vazbou na MGA, existující vazba RE na BRP v rámci dané MGA a zároveň musí být test vykonán v časové zóně dané oblasti. Pro lepší představu, jak navrhnout test je dobré si scénář nejprve nasimulovat manuálně. Na základě simulace je možné definovat, jaké testovací skripty jsou pro test potřeba a porovnat je s knihovnou již implementovaných skriptů. Knihovna příkazů vypadá stejně, jako ta pro integrační testy, jen jsou navzájem oddělené a zahrnují podporu odlišných funkcionalit. Níže je znázorněn pohled obsahující formulář pro založení produkční jednotky v rámci webové aplikace Online Service.

[43] Metering Grid Area



Obrázek 31 - Formulář pro založení Produkční jednotky v Online Service (NBS Online Service Private, 2021)

V souladu s testovacím scénářem, uvedeným v příloze B, je třeba navrhnout a implementovat podporu pro několik testovacích skriptů. Některé z nich budou vícenásobně použité v rámci automatizovaného testu, pouze s odlišnou hodnotou proměnných. Prvním skriptem, který je použit ve všech automatizovaných testech Online Service, je přihlášení a volba účastníka trhu, za kterého se test vydává. Dalším prvkem je navigace na příslušný pohled, v tomto případě se pohled pro zprávu Produkčních jednotek nachází v sekci struktur. Aby bylo možné otevřít formulář, test vyžaduje podporu pro kliknutí na tlačítko. Na základě výše zobrazeného formuláře je vyžadována podpora pro zápis hodnot do filtru a možnosti vybírání hodnot z tzv. combo boxů. Dále už jen stačí kliknout na tlačítko pro potvrzení a formulář se uzavře. Na závěr test ověří, zdali se Produkční jednotka založila a zobrazila na příslušném pohledu se všemi požadovanými atributy. Pro tyto účely testu postačí podpora filtrování a ověření hodnot v tabulce. K vytvoření testu bude podle provedené analýzy stačit podpora pro cca 7 typů testovacích skriptů. Takto navržený test je znázorněn na nadcházejícím obrázku.



Obrázek 32 - Příklad GUI testu pro založení Produkční jednotky (David Pfeifer, 2021)

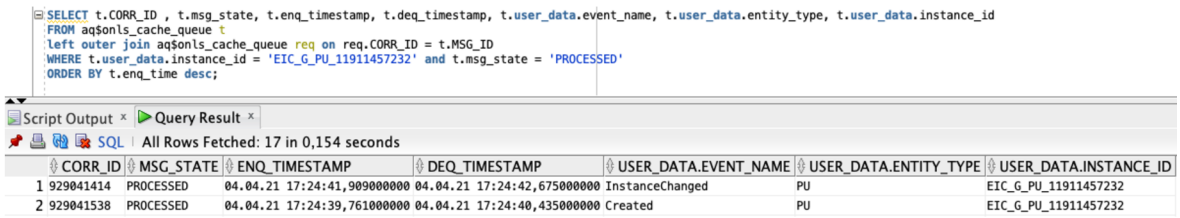
Struktura souboru je podobná jako v předešlých integračních testech. Je zde navíc použito víceúrovňové dělení scénářů. To šetří čas celé exekuce testu v případě, že by v rámci scénáře druhé úrovně byla objevena chyba, v takovém případě již nástroj nepokračuje s vykonáním akcí ve scénáři třetí úrovně označeném ###. V případě, že se nepodaří založit produkční jednotku, test již dále neověřuje, že byla založena, jelikož to postrádá smysl. Existují situace, kdy je testováno více na sobě nezávislých uživatelských pohledů v rámci jedné specifikace, v takovém případě mají dílčí scénáře stejnou úroveň a když jeden z nich skončí chybou, test pokračuje dál. Typicky při ověřování uživatelských práv na jednotlivé funkcionality.

Použité testovací skripty:

```
"Login","Login as <login> under <marketPartyType> <marketParty>"
"Page","Navigate to <useCase>"
"Components","Click on button <buttonName>"
```

```
"ModalDialog","Set Field <editorName> to <expectedValue>"

"ProductionUnits","Submit Production Units row <puName> and wait till
create event is processed"

"Editors","Set Filter <editorName> to <value>"

"TableComponent","Verify first row columns From and To is <expectedDa-
teValue>"

"TableComponent","Verify first row column <columnName> is <expectedVa-
lue>"
```

Jak lze výše vidět, díky použití obchodního jazyka jsou jednotlivé testovací skripty srozumitelné pro každého a pro testera není nutná znalost programování. Stačí jen v souladu s aplikací doplnit korektní hodnoty do proměnných a test je připraven k použití. Pouze jediný skript je potřeba vysvětlit podrobněji. Potvrzení požadavku pro založení produkční jednotky obsahuje dodatek, aby test s výkonem dalších kroků počkal do chvíle, než je zpracovaná událost v rámci Online Service Cache. To je z důvodu, že ne v každém případě je systém schopný založit a zobrazit produkční jednotku okamžitě. Zpoždění může být způsobeno například zpracováním jiných požadavků v rámci dané oblasti. Dalo by se zde použít přidání pauzy na fixní čas. To bohužel není efektivní, jelikož nikdy dopředu nelze odhadnout, jak velká fronta se v systému nachází a kolik času vytvoření dané struktury zabere. Mohlo by to být nedostačující, ale i moc dlouhé čekání. Nejefektivnějším řešením je v tomto případě použití databázového dotazu do tabulky, která v komponentě Online Service Cache eviduje záznamy a stavy pro každou událost. Takový databázový dotaz je integrován přímo do testovacího skriptu a je opakovaně vykonáván v předem nakonfigurovaném intervalu do té doby, dokud se v tabulce neobjeví záznam s hodnotou stavu „Processed“. Příklad použitého dotazu pro uvedený test je přiložen níže na obrázku. Jakmile je událost zpracovaná, test pokračuje s ověřením založené jednotky v rámci uživatelského rozhraní.

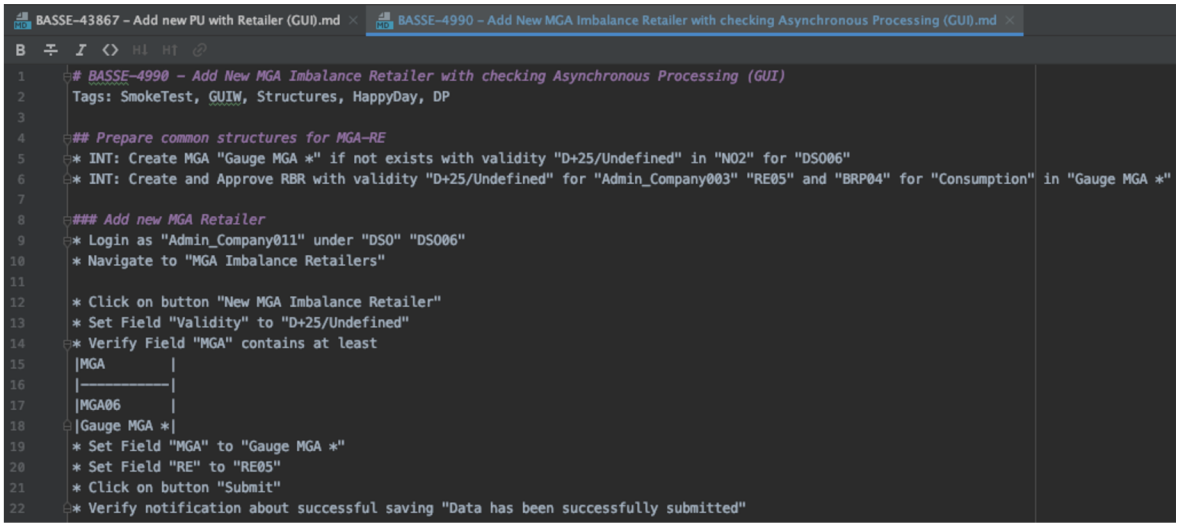


Obrázek 33 - SQL dotaz stav události v Online Service Cache (David Pfeifer, 2021)

Tato část práce je věnována především návrhu a implementaci automatizovaných testů. Principy, postupy a doporučení pro efektivní tvorbu již byly vysvětleny. V úvodu

celé kapitoly byl zmíněn pojem znovupoužitelnost testů ve spojení s testovacími daty. Výše představené testy si vždy vystačily s použitím generování náhodných hodnot do proměnných. Nicméně existují testovací scénáře, které vyžadují specifická data a po provedení testu již nejsou znovupoužitelná a tím pádem test není možné opakovat se stejným výsledkem. Jak toto vyřešit? Automatizovaný test si před samotnou exekucí musí založit, s ohledem na požadavky definované v testovacím scénáři, vlastní data. Jako příklad byl zvolen testovací scénář, jehož cílem je založení vazby mezi MGA a RE. Celé znění testovacího scénáře je přiloženo v příloze D.

Hlavním problémem je v tomto případě funkční požadavek, který je realizován specifickou validací, která umožňuje přiřazení pouze jednoho RE pro MGA v daném čase. Z toho vyplívá, že pro možnost opakovaného vykonání testu je potřeba vytvořit vždy nové MGA. A aby to nebylo úplně jednoduché, v rámci nově zavedené MGA je nutné vytvořit vazbu zodpovědnosti mezi BRP a RE. V tuto chvíli je možné využít výhodu, která spočívá na základě používání stejného nástroje jak pro integrační, tak pro GUI testy. Jelikož naplnění výše zmíněných podmínek lze poměrně lehce docílit s využitím integračních testů, tak nic nebrání tomu danou implementaci převzít a použít v rámci testů uživatelského rozhraní. To by například s využitím nástroje Cypress nebylo realizovatelné. Skripty, pro založení potřebných testovacích struktur, jsou umístěné na začátku celé testovací specifikace. Mají definovanou vyšší úroveň než samotný testovací scénář, tím je docíleno toho, že samotný scénář bude vykonán pouze v případě, ve kterém byla příprava dat úspěšná. Úvodní část testovacího scénáře obsahující přípravu vlastních dat pro založení vazby MGA-RE je zobrazena na následujícím obrázku. Integrační skripty jsou v rámci specifikace označeny symbolem „INT:“. Test dále pokračuje standardním způsobem.



Obrázek 34 - Příklad použití kombinace integračních a GUI testů (David Pfeifer, 2021)

10.5 Aplikace testů při vývoji

Cílem předchozích dvou kapitol bylo čtenáři nastínit, jak je možné uchopit automatizování testů s využitím frameworku Gauge. Byly představeny různé techniky a možnosti, jak řešit jednotlivé výzvy při návrhu testů. Principy lze aplikovat při automatizaci dalších oblastí systému v rámci komponent Online Service a Back Office na projektu NBS. Tento přístup se dá dále použít na spoustě podobných projektů.

Tato kapitola má za cíl představit, jak lze výše implementované testy aplikovat v praxi. Testy lze pouštět lokálně na jakémkoli počítači s využitím příkazové řádky PowerShell. Nástroj následně vygeneruje grafické reporty i snímky obrazovky do předem nakonfigurované složky. Reporty jsou ve více formátech, já osobně preferuji otevření HTML formátu v jakémkoli webovém prohlížeči.

Při lokální exekuci je možné skript pro spuštění testu rozšířit o parametr „gauge-browser“, kterým lze definovat jeden ze tří podporovaných prohlížečů (Chrome, Firefox, Internet Explorer), defaultně nastavený bývá Chrome. V případě většího počtu automatizovaných testovacích případů a požadavku na otestování pouze jednoho z nich lze s využitím doplnění cesty k danému souboru nebo složce pustit pouze jeden test nebo celou složku s testy. Problém nastává, pokud jsou požadované testy rozprostřeny ve více složkách nebo naopak, když je potřeba některé testy z příslušného běhu vyřadit. Zde se projeví, jak byl tester při procesu tvorby pečlivý v rámci doplňování značek.

Konkrétní testy lze totiž s využitím značek spouštět a následně v rámci reportů s výsledky filtrovat. Nemusí zůstat u použití jedné značky, Gauge umožňuje různé kombinace přidání, odebrání, průniků a spojek.

Tags	Selects specs/scenarios that
`!TagA`	do not have TagA
`TagA & TagB` or `TagA,TagB`	have both TagA and TagB.
`TagA & !TagB`	have TagA and not TagB.
`TagA | TagB`	have either TagA or TagB.
`(TagA & TagB) | TagC`	have either TagC or both TagA and TagB
`!(TagA & TagB) | TagC`	have either TagC or do not have both TagA and TagB
`(TagA | TagB) & TagC`	have either [TagA and TagC] or [TagB and TagC]

Obrázek 35 - Kombinace značkování v nástroji Gauge (Gauge, 2021)

Pro spuštění testů s nadefinovanými tagy se používá parametr „-tags“.

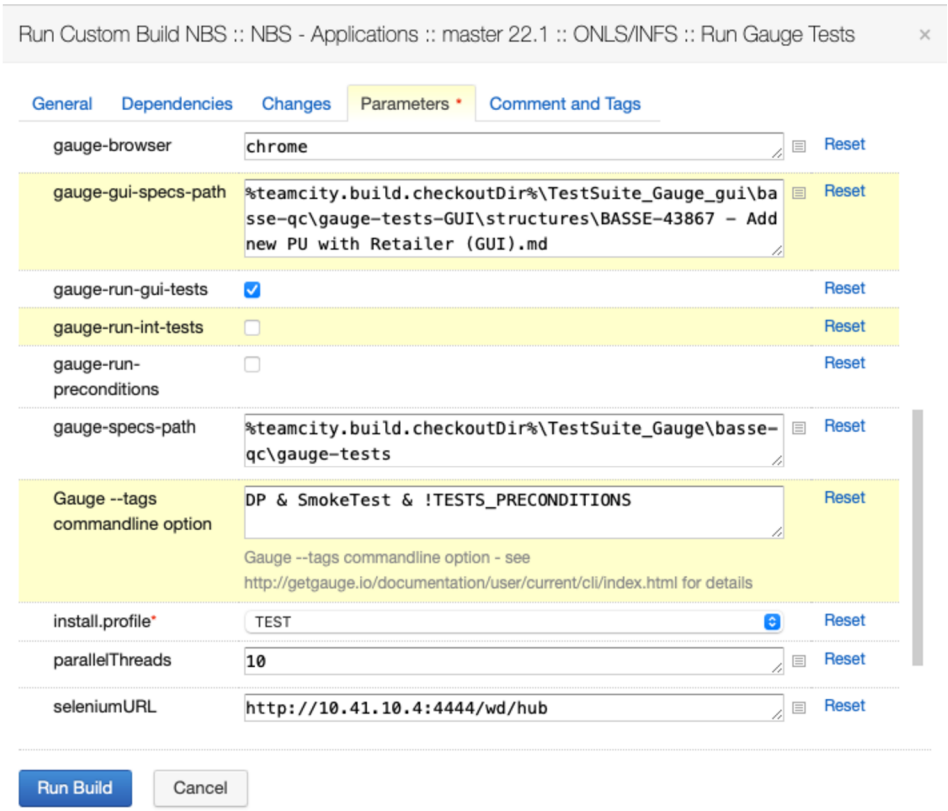
Lokální spuštění je vhodné spíše pro jednotlivce v rámci procesu návrhu a implementace testů. Dále v případech, kde není potřeba sdílet výsledky testů s více spolupracovníky. Pro projektové použití je nicméně lepší promyslet spolupráci s některým z nástrojů pro kontinuální integraci. Funkcí a výhodou frameworku Gauge je možnost napojení na CI/CD nástroje, které umožňují vzdálenou manipulaci, exekuci a sdílení výsledků automatizovaných testů. Na projektu NBS je pro podporu procesu CI/CD používán nástroj TeamCity.

Integrace automatizovaných testů do nástroje TeamCity je celkem intuitivní a jednoduchá. Testovací běh se spouští příkazovým skriptem, který je velmi podobný tomu lokálnímu. Hodnoty parametrů se zde doplňují pomocí proměnných z konfigurace. Příklad takového skriptu:

```
./TestSuite_Gauge_gui/run.ps1 -TestEnvironment %install.profile% -
specsPath "%gauge-gui-specs-path%" -parallelThreads "%parallelThreads%" -
tags "%gauge-tags%" -browser %gauge-browser%  -seleniumURL "%seleni-
umURL%"
```

Proměnné se definují na záložce Parameters přímo v TeamCity po kliknutí na tlačítko Run. Uživatelské rozhraní nástroje zprostředkovává uživatelsky přívětivou definici parametrů pro spuštění testu bez nutnosti vidět, či měnit zdrojový skript. Mezi základní

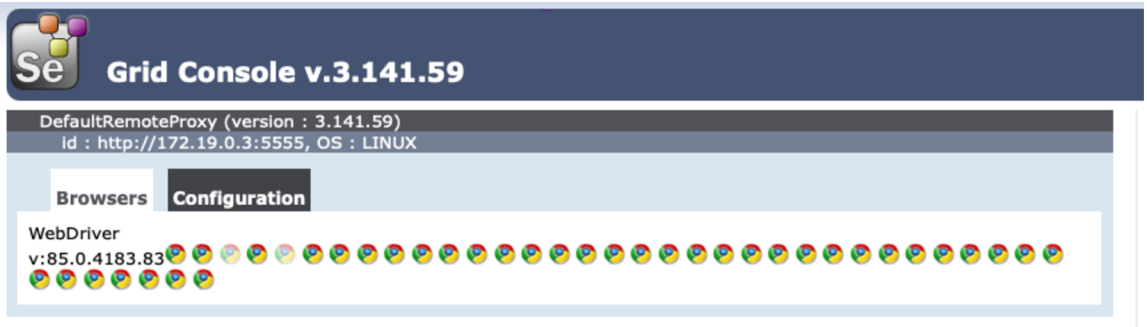
možnosti konfigurace, pro spuštění automatizovaných testů, patří výběr prostředí, webového prohlížeče a kategorie testů (integrační, GUI nebo obojí), definice cesty ke konkrétnímu testu nebo složce testů a počet paralelně spuštěných prohlížečů (testů) v rámci jednoho běhu.



Obrázek 36 - Parametrizace testů v TeamCity (David Pfeifer, 2021)

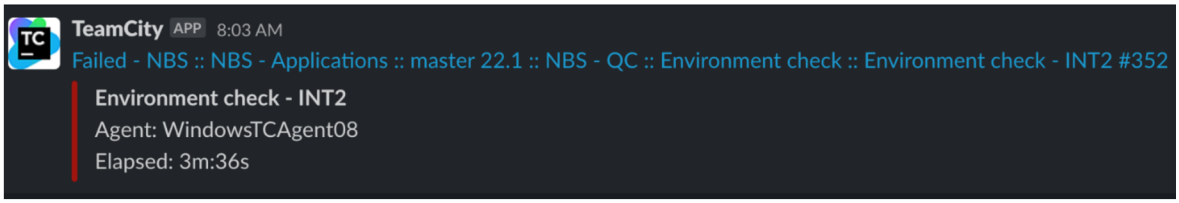
Technicky složitější je zapojení a konfigurace samotného Selenia, Selenium Serveru a Selenium Gridu. Nejprve je nutné na TeamCity server stáhnout Selenium server a ovladače pro jednotlivé prohlížeče. Poté je možné Selenium Server spustit. Dalším krokem je stáhnutí a konfigurace Selenium Grid, jehož součástí je správa a spouštění prohlížečů v rámci Docker[44]. Dostupnost, verze a konfigurace prohlížečů v rámci Selenium Grid je dostupná s využitím Grid Console, která je zobrazena na následujícím obrázku.

[44] https://www.docker.com



Obrázek 37 – Selenium Grid Console (2021)

Automatizované sady testů jsou vykonávány agentem v rámci TeamCity serveru a nejsou dál nijak závislé na uživateli. Jakýkoli oprávněný uživatel může vybrané testy spustit a po vykonání všech testů je automatizovaně vygenerován finální report. Mezitím se uživatel může věnovat jiným projektovým činnostem. TeamCity umožňuje dále například zasílání notifikací o proběhlých testech do kanálů komunikační platformy Slack[45] nebo ne email. Notifikace může být zasílána jak po úspěšné, tak i po neúspěšné exekuci všech testů. Obsahuje číslo buildu, čas trvání a také přímý odkaz na detailní report výsledků. Takto naimplementovaná notifikace může vypadat následovně.



Obrázek 38 - TeamCity notifikace zaslaná do Slack (David Pfeifer, 2021)

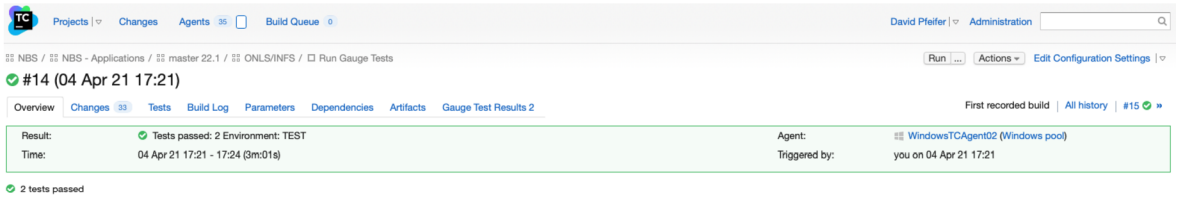
Všichni členové skupiny této komunikační platformy jsou notifikování a mají možnost okamžité navigace na výsledky testů za účelem další analýzy. Na automatizovaném testování tak mohou s využitím nástroje TeamCity spolupracovat všichni členové vývojového týmu. Pro správu aktuálních testovacích specifikací, implementace a konfigurace je využíván nástroj GIT. GIT dále podporuje řízení, paralelní úpravu a správu více verzí testů pro různé vývojové větve (verze aplikace). Existuje několik aplikací pro práci s tímto systémem, například GIT Extensions[46] od vydavatele GitHub nebo Sourcetree[47] od společnosti Atlassian.

[45] https://slack.com
[46] http://gitextensions.github.io
[47] https://www.sourcetreeapp.com

V rámci agilního přístupu vývoje je kladen důraz na pravidelné testování, aby bylo možné šetřit čas a zdroje, tak TeamCity umožňuje nastavit pravidelný automatizovaný spouštěč, který zvolenou sadu testů vykoná například každý den v méně frekventovaný čas, aby prostředí nebylo zatíženo. Odpovědní uživatelé jsou pak notifikování pouze v případě objeveného problému. Z osobní zkušenosti mohu doporučit, pro zvýšení efektivity vývoje softwaru, nastavit spouštěč pro testy kontrolující základní funkcionalitu prostředí minimálně 2x denně, a to ráno před prvním použitím prostředí a večer po testech, zdali nic nebylo porušeno. Kompletní sady testů doporučuji vykonávat po každém nasazení nové verze komponent.

10.6 Vyhodnocení a reportování výsledků testů

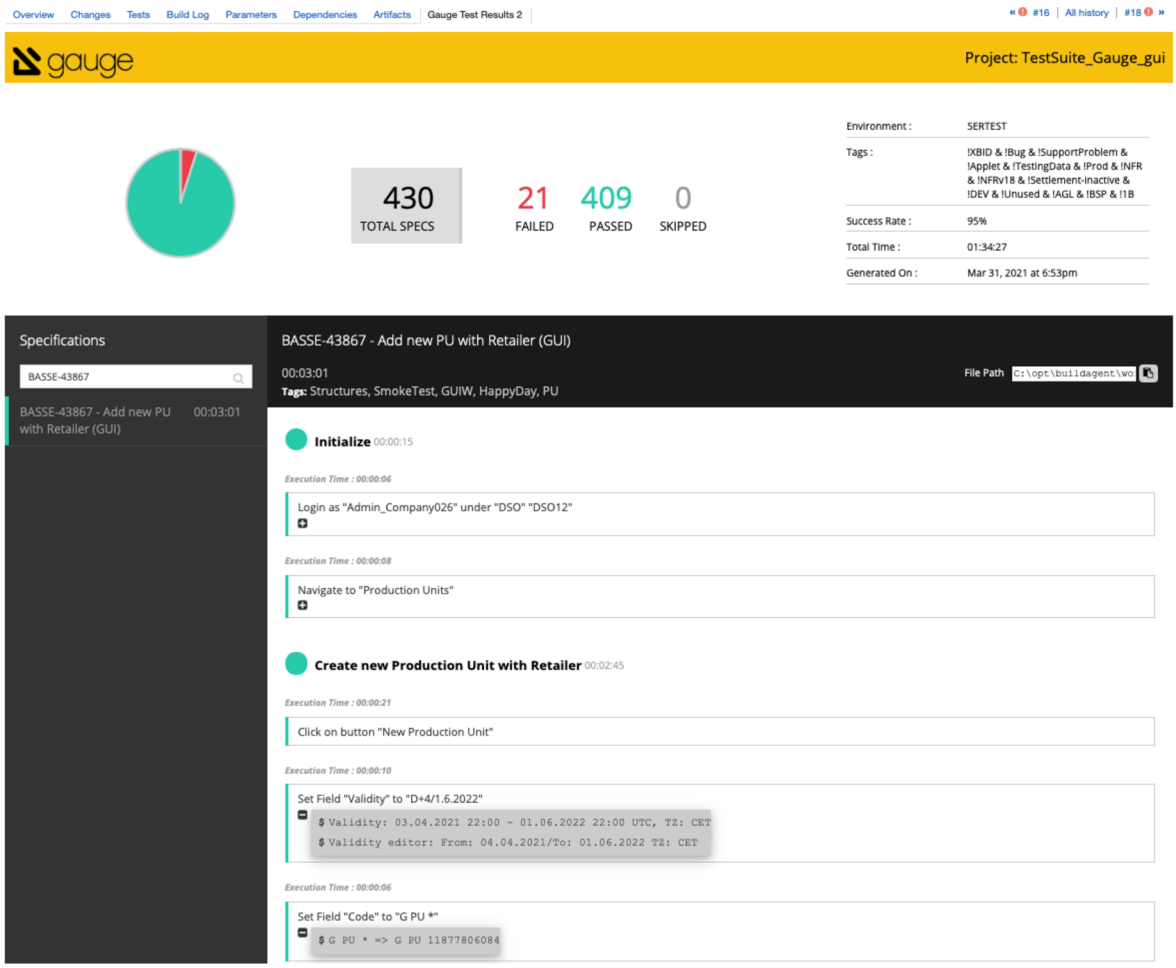
Jak již bylo mnohokrát v této práci zmíněno, Gauge vyniká nejen svojí podporou nejrůznějších funkcí, ale především svými krásně graficky zpracovanými reporty, které testovacímu týmu usnadňují vyhodnocení. Test lze vyhodnotit z více perspektiv. V ideálním případě lze využít pouze nativní funkce v TeamCity pro zobrazení souhrnu daného běhu. Pro konkrétní testovací specifikaci založení Produkční jednotky vytvořenou v rámci kapitoly 10.4 je souhrn zobrazen na následujícím obrázku. Počet provedených testů se zobrazuje ve formě provedených scénářů, které daná specifikace obsahuje. Zde lze vidět, že provedení celého testu trvalo nástroji 3 minuty. Manuální test takové funkcionality zabírá zkušenému testerovi v průměru 15 minut.



Obrázek 39 - Přehled testovacího běhu v TeamCity (David Pfeifer, 2021)

V případě potřeby detailnějších výsledků je kompletní report vygenerován na záložce Gauge Test Result 2, který obsahuje souhrn pro GUI testy. V případě zapojení integračních testů by byla zobrazena záložka Gauge Test Result. Gauge Report obsahuje podrobné výsledky vykonaných testovacích skriptů všech specifikací v dané testovací sadě. Znázorněná sada níže obsahuje celkem 430 specifikací, včetně té, pro vytvoření Produkční jednotky. Úspěšnost testů je zobrazena přehledně koláčovým grafem a údaji o celkovém počtu úspěšných a neúspěšných testů. Dále obsahuje základní informace o

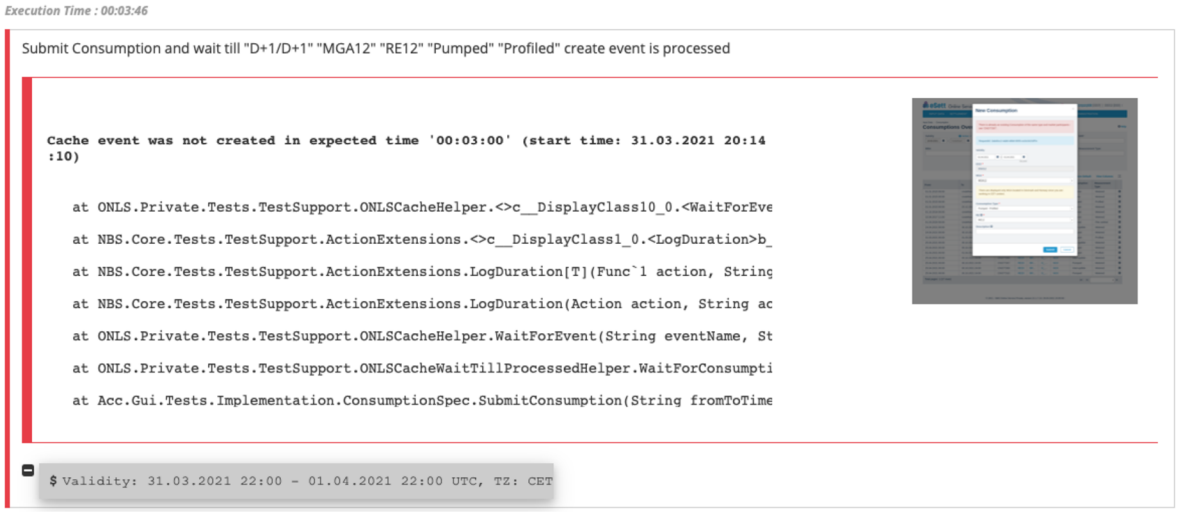
konfiguraci a délce trvání. Jednotlivé testy lze v rámci reportu filtrovat podle unikátního JIRA identifikátoru. Hlavní část reportu obsahuje záznamy o provedených testovacích skriptech a jednotlivých dobách trvání. Výhodou je zobrazení všech vyplněných a použitých dat v rámci testu, včetně těch náhodně vygenerovaných. Takové informace lze jednoduše vykopírovat a použít například pro další zkoumání v rámci aplikačních logů.



Obrázek 40 - Vygenerovaný Gauge report (David Pfeifer, 2021)

Analýza neúspěšných testů je velmi jednoduchá. Buď je lze jednotlivě prohlížet a filtrovat v rámci specifikací na levé straně reportu nebo je možné kliknout přímo na číslo počtu neúspěšných testů a všechny se zobrazí. Pak jen stačí v rámci reportu najít příslušný testovací skript, který selhal a manuálně provézt další analýzu. U integračních testů bývá selhání doplněno o log z aplikace, testy uživatelského rozhraní obsahují navíc snímek obrazovky zachycující dané selhání skriptu. Následující obrázek zachycuje

selhání testovacího skriptu při zakládání datové struktury spotřeby. Ze snímku obrazovky byla promptně odhalena chyba v testovacích datech. Snímek obrazovky lze v praxi otevřít v plné velikosti.



Obrázek 41 - Příklad selhání testovacího skriptu v rámci reportu (David Pfeifer, 2021)

Nalezené defekty jsou testerem zakládány do projektu v nástroji JIRA a není potřeba složitého a zdlouhavého popisování problému, stačí se odkázat na daný test a doplnit základní informace. Po opravě defektu stačí daný test znovu spustit a v případě úspěšného vykonání bug uzavřít. To představuje další ušetřený čas v rámci automatizovaného přetestování.

11 Shrnutí výsledků

V rámci přechozích kapitol této práce byl naplněn téměř celý životní cyklus automatizovaného testu. Na závěr je důležité provézt zpětné přezkoumání a posoudit, zdali navrhnuté testy splnily stanovené cíle.

Nejprve byl navržen přístup, jak automatizovat testy kondice a správné integrace vývojových a testovacích prostředí. Toho bylo docíleno zapracováním specifických SQL dotazů do testovacích specifikací v rámci Gauge. Tímto přístupem byla na projektu NBS následně automatizována celá řada kontrol. Celkem bylo vytvořeno 23 automatizovaných specifikací zahrnující verifikaci výpočetního modulu, vygenerovaných uzávěrek, procesních řízení, dlouhotrvajících procesů, databázového místa, integrace na jiné komponenty, integritu strukturálních dat apod. Tento typ kontrol na projektu dříve neprobíhal a problémy byly zjištěny až v rámci vývojového procesu nebo testování, kdy například aplikace přestala být dostupná, nebyla zobrazována nově vytvořená data, procesy nebyly zpracovávány a celková kondice prostředí nebyla vhodná k použití. To způsobovalo zdržení celého vývojového procesu, jelikož manuální analýza problémů byla zdlouhavá. Nyní jsou připravené testy s využitím nástroje TeamCity automatizovaně vykonávány dvakrát denně brzy ráno, před tím, než je prostředí používáno a večer po celodenním používání. V případě nalezení problému jsou zodpovědné osoby informovány v rámci komunikační platformy Slack. Gauge nejen, že na problém upozorní, ale celkem přesně identifikuje příčinu problému, který lze být obratem opraven. Zavedení tohoto procesu přineslo včasné odhalení problémů na prostředí a tím ušetřilo desítky hodin práce jednotlivým členům týmu. Testerům při čekání na opravu prostředí a vývojářům při opravě prostředí. Exekuce takto automatizovaných dotazů trvá v průměru 3 minuty. Jelikož jsou všechny testy založené na stejném testovacím skriptu, jejich implementace do systému zabrala cca 2 hodiny. V případě potřeby jsou testy dále rozšiřovány nebo upravovány.

Další navrhnutý přístup se týkal funkčních integračních testů. S využitím uvedených postupů, technik a doporučení bylo vytvořeno celkem 30 automatizovaných specifikací. Testy pokrývají tzv. happy day[48] scénáře základních systémových funkcionalit, které jsou reprezentované ukládáním hodinových hodnot interním datovým tokem pro

[48] Pozitivní testovací scénář

všechny typy datových struktur a zaváděním všemožných strukturálních entit do systému. Integrační testy představují mezistupeň mezi jednotkovými testy a GUI testy. Je to další článek verifikace, který slouží k zachycení a nalezení defektů v rámci nejdůležitějších funkcionalit, které nebyly odhaleny v rámci jednotkových testů. Vykonání celé testovací sady trvá okolo 5 minut, takže lze testy pouštět i několikrát denně a zvyšovat efektivitu testování a tím i celého vývoje softwaru. Pokud jsou defekty zachyceny na integrační úrovni, tím méně času stojí nalezení a následná oprava. Nalezení chyb v rámci manuálních testů nebo automatizovaných GUI testů je časově daleko náročnější. Pro porovnání času exekuce jednotlivých kategorií testů byla vytvořena následující tabulka. Z naměřených dat vyplívá, že provedení a tím pádem i nalezení chyby, je v podobě integračního testu mnohonásobně rychlejší než automatizovaným GUI testem. Nejhůře je na tom s efektivitou manuální test. Vykonání celé sady základních testů, nazývané také jako Smoke Test, je v rámci integrační kategorie 3x rychlejší, než GUI a 30x rychlejší v porovnání s vykonáním manuálních testů, v případě spolupráce dvou testerů. Doba strávená návrhem, implementací a vykonáním integračních testů v porovnání s dobou pro manuální testy se s každým opakování rychle otáčí ve prospěch automatizace.

Tabulka 3 - Efektivita kategorií testů (David Pfeifer, 2021)

Test	Automatizovaný Integrační test	Automatizovaný GUI test	Manuální test
Uložení hodnot	34 s	3 m 38 s	10-15 m
Zavedení nové struktury	20 s	3 m 53 s	10-15 m
Ověření dat	1 s	1 m 54 s	5 m
Sada 30 testů	5 m	14 m	2,5 h *

* Výsledný čas vypočítaný z průměrného času na test 10 minut a využitím dvou testerů. Jeden tester by stejnými testy strávil celkem tedy 5 hodin.

Příčinou efektivity není jen samotná automatizace, ale také možnost paralelního spuštění testů. Výsledky v tabulce se zakládají na běhu, který využíval 10 paralelních vláken pro exekuci testů. Další výhodou je čas testera strávený jinou důležitou činností, zatímco testy jsou automatizovaně exekvovány.

Část práce zabývající se návrhem automatizace testů uživatelského rozhraní poskytla obecné i specifické řešení, jak k samotnému návrhu této kategorie testů přistupovat. Zdůraznila nepodcenění úvodní analýzy testovacího scénáře, která probíhá manuálně a na jejímž základě je vydefinována potřeba jednotlivých testovacích skriptů. Ve svém pokračování kapitola nastínila, jak lze řešit různé situace, kdy je potřeba do skriptu integrovat čekání na zpracování události a na závěr bylo vysvětleno, jak efektivně kombinovat implementaci integračních testů v rámci přípravy testovacích dat pro GUI testy. Na projektu NBS bylo posléze uvedenými způsoby automatizováno celkem 226 regresních testovacích scénářů. Definované scénáře bývají součástí regresního testování v rámci FAT. Při přechodu na agilní metodiku vývoje byl kladen důraz na častější testování. Toho bylo docíleno právě automatizací výše zmíněných scénářů, jejichž testy je možné vykonávat i několikrát do týdne. Následující tabulka zobrazuje celkový čas věnovaný automatizování a vykonání všech 226 testovacích scénářů v porovnání s manuálními testy.

Tabulka 4 - Porovnání časové náročnosti automatizovaných a manuálních testů. (David Pfeifer, 2021)

Realizace 226 testů	Automatizovaný GUI test	Manuální test
Vývojář – implementace skriptů	120	
Tester – návrh a příprava skriptů	113	
Konfigurace	8	
Doba trvání testování	2	113
Analýza výsledků a manuální report výsledků do JIRA	8	
Údržba testů	23	
Celkem 1 testovací kolo	274	113
Celkem 2 testovací kola	307	226
Celkem 3 testovací kola	340	339
Celkem 4 testovací kola	373	452

Celková doba pro realizaci, včetně vykonání jednoho kola 226 automatizovaných testů vychází na 274 hodin. Do celkového času je odhadem promítnuta i očekávaná údržba jednotlivých testů v průběhu daného kola. Exekuce manuálních testů v rámci prvního kola stojí pro porovnání pouze 113 hodin času. Počítáno bylo s průměrnou dobou trvání 30 minut na jeden testovací scénář. Z uvedených dat vyplívá, že se nevyplatí investovat do rozsáhlé automatizace na krátkodobém projektu, který plánuje

méně než tři kola regresních testů v rámci softwarového vývoje. Po provedení třetího kola regresních testů jsou celkové časy vyrovnané a od čtvrtého kola se rozhodně vyplatí automatizovat. Implementace nových funkcionalit a verzí, zahrnující regresní testování, je na projektu NBS plánována na příštích několik let, v tomto případě se investice do automatizace vyplatila finančně i časově a vrátila ve zvýšené kvalitě softwaru.

V rámci agilního vývoje se počítá s častějším testováním a lze tedy testy spouštět po nasazení každé větší implementační změny. V takovém případě není potřeba zapisovat všechny výsledky do JIRA, ale stačí analyzovat pouze takové testy, které skončily s neočekávaným výsledkem.

Testovací sady jsou v rámci implementace nových funkcionalit kontinuálně rozšiřovány o nové specifikace za účelem dalšího zvyšování efektivity testovacího a vývojového procesu.

12 Závěr

Hlavním posláním této diplomové práce byl výběr nástroje, návrh a implementace automatizovaných testů pro zvýšení efektivity vývoje softwaru na projektu Nordic Balance Settlement, který je realizován společností Unicorn pro zákazníka ze Skandinávie. Druhotnou, ale neméně důležitou myšlenkou práce, bylo seznámení čtenáře s problematikou testování softwaru a zdůrazněním, jak je v rámci vývojového procesu nezbytné správně a efektivně testovat.

Naplnění cílů bylo započato v teoretické části práce, která byla úvodem zaměřena na téma, co je to vlastně software a k čemu se používá. Navazující charakteristika webových aplikací a jejích druhů celé téma doplnila. Úvodní kapitola byla zakončena představením tradičních a agilních vývojových metodik.

Další část práce byla zaměřena již přímo na testování softwaru a jeho úlohu v rámci vývojového procesu. Bylo představeno, co znamenají pojmy testování a softwarový defekt. Krátce byly definovány principy testování podle ISTQB, mezinárodní rady pro testování softwaru. Po úvodním seznámení, bylo testování rozděleno do jednotlivých kategorií a detailně byly popsány i nejběžnější testovací techniky, které byly reprezentovány například statickými technikami, ale i pokročilými výkonnostními a penetračními testy. Navazovala témata jako plánování, exekuce a životní cyklus testování, která byla rozšířena o nástroje pro podporu testování a životní cyklus defektu. Kapitola byla uzavřena definicí rolí a odpovědností členů v testovacím týmu.

Cesta pokračovala do oblasti automatizovaného testování, které bylo nejprve definováno a posléze doplněno o jednotlivé kategorie a metodický proces návrhu a vývoje testů. Téma pokračovalo představením nejpoužívanějších nástrojů a technologií pro automatizaci různých kategorií testů. Popis některých nástrojů byl doplněn o praktický rozbor na základě zkušeností autora. Kapitola končí seznamem rad a doporučení pro automatizaci testů.

Závěrečná kapitola teoretické části byla věnována stručnému představení kontinuální integrace a nejznámějších nástrojů, které ji podporují.

Praktická část má několik dílčích výstupů. Ve svém začátku byly představeny platforma Damas a projekt Nordic Balance Settlement, včetně svých hlavních funkcí, které slouží

pro vypořádání nerovnováhy v rámci harmonizace severského trhu s elektřinou. Životní cyklus automatizovaného testu začíná podrobným rozborem architektury budoucí testované aplikace, zahrnující analýzu a výzkum současné situace pokrytí automatizovanými testy. Jsou představeny také nevýhody historického přístupu k testům uživatelského rozhraní na projektu. Na základě rozboru jsou definovány požadavky a testy, které budou automatizovány.

Následuje jedna z předních kapitol, která je zaměřena na volbu vhodného nástroje pro automatizaci, na základě definovaných cílů a požadavků. Do užšího výběru byly vybrány tři nástroje Gauge, JMeter a Cypress. Všechny tyto nástroje byly podrobeny podrobné funkční analýze s ohledem na předem stanovená hodnotící kritéria. Provedená zjištění byla použita dále při navazující SWOT analýze. Nejvhodnějším nástrojem byl nakonec zvolen Framework Gauge.

Předposlední kapitola byla pojednávala o návrhu integračních testů a testů uživatelského rozhraní. První test byl zaměřený na vykonávaní SQL dotazů pro zajištění kvality testovacích prostředí. Druhý a třetí integrační test reprezentovaly reporting a strukturální operace. Podkapitola věnující se GUI testům obsahovala doporučení, jak takové testy nejlépe vytvářet a byla doplněna o spoustu rad a poznatků, jak řešit nestandardní situace. Bylo zde představeno také využití technologie integračních testů pro přípravu dat před vykonáním testu uživatelského rozhraní. Závěr této části práce byl zaměřen na zapojení testů do vývojového procesu. Bylo zde zodpovězeno a ukázáno, jak lze testy spouštět a parametrizovat lokálně i s využitím nástroje TeamCity. Nastíněno bylo také, co taková příprava a integrace automatizovaných testů v rámci TeamCity obnáší a jak lze nástroj používat ke zvyšování efektivity při kontrole kvality. Představen byl také proces vyhodnocování a analyzování výsledků v rámci Gauge reportů.

Poslední kapitola byla věnována celkovému zhodnocení navrhnutých přístupů pro tvorbu testů. Implementace testů pro kontrolu kvality prostředí docílila větší stability a efektivity při odstraňování vzniklých problémů v rámci vývojových serverů. Zapojení integračních testů do vývojového procesu přineslo včasnější nalezení a odstranění chyb. Poslední kategorie GUI testů přinesla okamžitý přínos v rámci agilního procesu vývoje v podobě častějšího testování a při regresních testech ve fázi FAT se výhody automatizace začínají projevovat po třetím testovacím kole. Závěrem bych rád zmínil, že

všechny implementované testy přinesly projektu značnou časovou i finanční úsporu. Cíl práce byl v tomto případě maximálně naplněn.

13 Seznam použité literatury

Tištěné zdroje

1. MOLINA-RÍOS, Jimmy, PEDREIRA-SOUTO, Nieves. *Information and Software Technology: Comparison of development methodologies in web applications.* ScienceDirect, 2020.

2. BOEHM, Barry and Papaccio N. *Understanding and Controlling Software Costs. In IEEE Transactions on Software Engineering.* Redondo Beach, CA, 1988. ISSN 0098-5589.

3. NEELU, Lalband, KAVITHA, D., *Software Development Technique for the Betterment of End User Satisfaction using Agile Methodology,* UIKTEN, 2020. ISSN 2217-8309.

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

5. MÜLLER, Thomas, FRIEDENBERG, Debra. *Certified Tester*. ISTQB, 2011.

6. HERBOLD, Steffen, HAAR, Tobias. *Smoke Testing for Machine Learning: Simple Tests to Doscover Severe Defects*. Computer Science, 2020. edsarx.2009.01521.

7. HLAVA, Tomáš. *Test management*. Unicorn Education, 2017.

8. EVERETT, Gerald D, MCLEOD, Raymond. *Software testing: Testing across the Entire Software Development Life cycle*. Hoboken: Wiley-Interscience, 2007. ISBN-10: 047179371X.

9. Hlava Tomáš, *Statické techniky testování softwaru.* Unicorn Top Gun Academy, 2012.

10. MCCONNELL, Steve. *Dokonalý kód: umění programování a techniky tvorby software.* Brno: Computer Press, 2005. ISBN: 80-251-0849-X.

11. JORGENSEN, Paul. *Software testing.* Boca Raton: Auerbach. 2008. ISBN 0-8493-7475-8.

12. HLAVA Tomáš, *Testování v životním cyklu softwaru.* Unicorn Top Gun Academy, 2012.

13. JEŽEK Miroslav, *Performance testing.* Unicorn Top Gun Academy, 2019.

14. SHWARTZ, Jonathon, KURNIAWATI, Hanna. *Autonomous Penetration Testing using Reinforcement Learning.* Computer Science, 2019. edsarx.1905.05965.

15. PATTON, Ron. *Testování softwaru*. Praha: Computer Press, 2002. Programování. ISBN 80-7226-636-5.

16. HLAVA Tomáš, *Test Strategy.* Unicorn Top Gun Academy, 2016.

17. VODÁČEK, Leo a Oľga VODÁČKOVÁ. *Moderní management v teorii a praxi*. 3., rozš. vyd. Praha: Management Press, 2013. ISBN 978-80-7261-232-1.

18. RENDÓN, Ana, BARRERA, Juan. *Current state of automation of software tests*. Instituto Antioqueno de Investigación, 2017. ISSN: 2248-7441.

Elektronické zdroje

19. Linda Rosencrance, Definition of software (on-line). 2021 (citace 12.03.2021). Přístup z internetu:
URL: https://searchapparchitecture.techtarget.com/definition/software

20. TechTerms, Web Application Definition (on-line). 2014 (citace 12.03.2021) Přístup z internetu:
URL: https://techterms.com/definition/web_application

21. ROYCE, Winston. Managing the Development of Large Software Systems (on-line). 1970 (citace 12.03.2021) Přístup z internetu:
URL: http://www.cs.umd.edu/class/spring2003/cmsc838p/Process/waterfall.pdf

22. CHAPMAN, James. Software development Methodology (on-line). 2004 (citace 12.03.2021) Přístup u internetu:
URL: http://www.hyperthot.com/pm_sdm.htm

23. HLAVA, Tomáš. Fáze a úrovně provádění testů (online). 2011 (citace 13.03.2021) Přístup z internetu:
URL: http://testovanisoftwaru.cz/category/metodika-testovani/druhy-typy-a-kategorie-testu/

24. Testbirds. Company History (online). 2020 (citace 13.03.2021) Přístup z internetu:
URL: https://www.testbirds.com/company/about-us/company-history/

25. COHEN, Jason. Best Kept Secrets of Peer Code Review. Smartbear (online). 2013 (citace 13.03.2021). Přístup u internetu:
URL: https://smartbear.com/smartbear/media/pdfs/best-kept-secrets-of-peer-code-review.pdf

26. WELLS, Don. Pair Programming. Extreme Programming (online). 1997 (citace 13.03.2021) Přístup z internetu:
URL: http://www.extremeprogramming.org/rules/pair.html

27. OWASP, Who is the OWASP® Foundation? (online). 2021 (citace 15.03.2021) Přístup z internetu:
URL: https://owasp.org

28. ČERMÁK, Miroslav. Black box test (online). 2010 (citace 14.03.2021) Přístup z internetu:
URL: https://www.cleverandsmart.cz/black-box-test/

29. ČERMÁK, Miroslav. Grey box test (online). 2010 (citace 14.03.2021) Přístup z internetu:
URL: https://www.cleverandsmart.cz/grey-box-test/

30. ČERMÁK, Miroslav. White box test (online). 2010 (citace 14.03.2021) Přístup z internetu:
URL: https://www.cleverandsmart.cz/white-box-test/

31. Techopedia. Automated Testing (online). 2021 (citace 14.03.2021) Přístup z internetu:
URL: https://www.techopedia.com/definition/17785/automated-testing

32. KITNER, Radek. Tápe váš tým jak začít s automatizací testování? (online). 2021 (citace 14.03.2021) Přístup z internetu:
URL: https://kitner.cz/testovani_softwaru/tape-vas-tym-jak-zacit-s-automati-zaci-testovani/

33. Cypress.io. How it works (online). 2021 (citace 14.03.2021) Přístup z internetu:
URL: https://www.cypress.io/how-it-works

34. Guru99. What is Selenium? Introduction to Selenium Automation Testing (online). 2021 (citace 15.03.2021) Přistup z internetu:
URL: https://www.guru99.com/introduction-to-selenium.html

35. Software Freedom Conservancy (SFC). Selenium IDE (online). 2019 (citace 15.03.2021) Přístup z internetu:
URL: https://www.selenium.dev/selenium-ide/

36. Software Freedom Conservancy (SFC). WebDriver (online). 2019 (citace 15.03.2021) Přístup z internetu:
URL: https://www.selenium.dev/documentation/en/webdriver/

37. Software Freedom Conservancy (SFC). Grid (online). 2019 (citace 15.03.2021) Přístup z internetu:
URL: https://www.selenium.dev/documentation/en/grid/

38. Gauge. Overview (online). 2021 (citace 15.03.2021) Přístup z internetu:
URL: https://docs.gauge.org/overview.html?os=macos&lan-guage=javascript&ide=vscode

39. JMeter. Apache JMeter™ (online). 2021 (citace 15.03.2021) Přístup z internetu:
URL: https://jmeter.apache.org

40. Smartbear. ReadyAPI Pricing (online). 2021 (citace 15.03.2021) Přístup z internetu:
URL: https://www.soapui.org/tools/readyapi/pricing/

41. FOWLER, Martin. Continuous Integration (online). 2006 (citace 15.03.2021) Přístup z internetu:
URL: https://www.martinfowler.com/articles/continuousIntegration.html

42. BrowserStack. TeamCity vs Jenkins vs Bamboo: A Breakdown (online). 2021 (citace 15.03.2021) Přístup z internetu:
URL: https://www.browserstack.com/guide/teamcity-vs-jenkins-vs-bamboo

43. Unicorn Solutions a. s. Damas MMS, Flexibilní platforma pro market management systémy (online). 2020 (citace 02.04.2021) Přístup z internetu:
URL: https://uuos9.plus4u.net/uu-webkit-maing02/44fc7230839b428db6b68853f06b13d7/getBinaryDataForWriters?code=dev_small_Damas_MMS_CZ_final_version

44. eSett Oy, Handbook (online). 2021 (citace 02.04.2021) Přístup z internetu:
URL: https://www.esett.com/handbook/

Příloha A – Komparativní analýza nástrojů pro automatizaci

Požadavek	Gauge	JMeter	Cypress
Typy testů			
Integrační	+	+	-
GUI E2E	+	+	+
API/WS	+*	+	-
Tvorba automatizovaných testů			
Tvorba Test cases	+	+	+
Tvorba Test stepů	+	+	+
Úrovně testů	+	+	-
Tvorba Selektorů	+	+	+
Centrální evidence stepů	+	-	+
Znalost programování	+	-	+
Integrované stepy	-	-	+
Možnost debuggování	+	+	+
Spuštění			
Lokální spuštění	+	+	+
Podpora integrace s CI/CD	+	+	+***
Možnosti rozšíření	+	+	-
Nutnost rozšíření	-	-	+
Reporting			
Lokální grafický report	+	+**	+
Vzdálený grafický report	+	+**	+***
Snímky obrazovky	+	+	+
Videa	+	+**	+
Načítání dat z externích zdrojů			
Microsoft Excel	-	+	+
XML	+	+	+
Tabulka	+	+	+
Stahování a vyhodnocení souborů			
Microsoft Excel	-	+	+
XML	-	+	+
Tabulka	+	+	+
Podpora prohlížečů			
Firefox	+	+**	+***
Chrome	+	+**	+
Internet Explorer	+	+**	-

Technologie			
Selenium	+	+	-
Selenium Server	+	+	-
C#	+	-	-
Java	+	+	-
Javascript	+	+	+
Python	+	-	-
Ruby	+	-	-
Platforma, podpora a UE			
Linux	+	+	+
Macintosh	+	+	+
Microsoft Windows	+	+	+
Technická podpora	-	-	+
Komunita	+	+	+
JIRA integration	+****	+	+
User Experience	+ + + + +	+ + + +	+ + +
Cena	**Open Source**	**Open Source**	**75$, 300$, neuvedeno / měsíc**

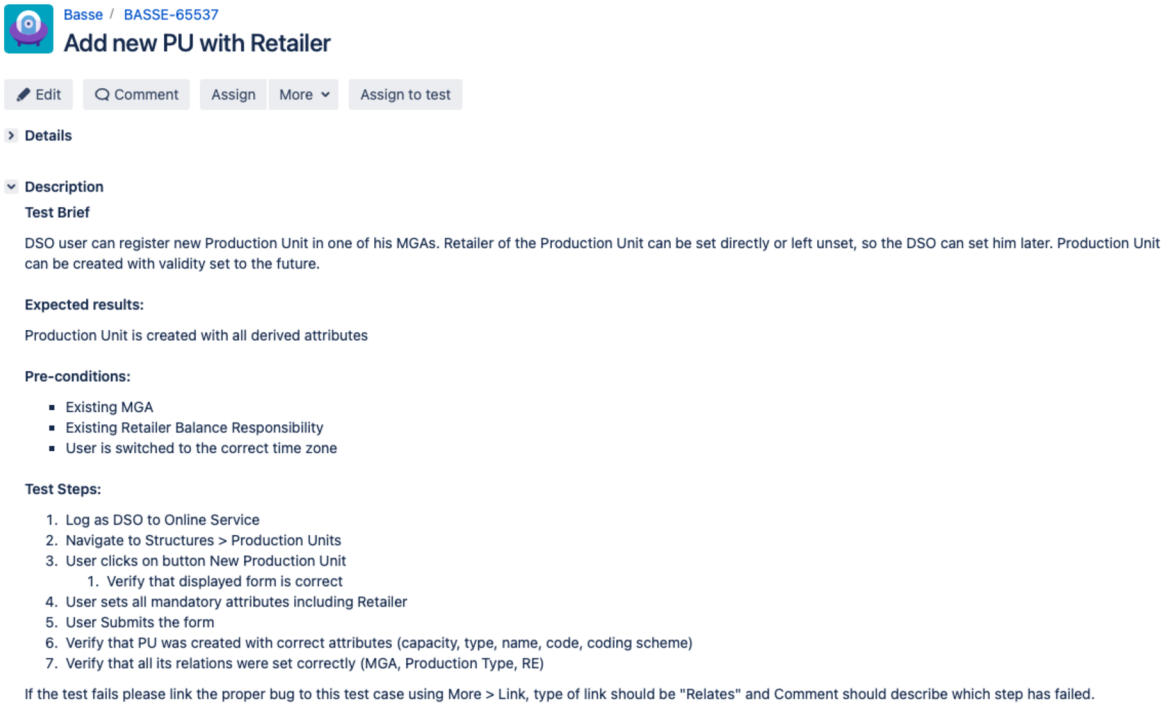
* pouze základní test API

** s využitím externího doplňku

*** pouze v kombinaci s GITHub a GitLab

**** pouze s využitím doplňku JIRA XRAY

Příloha B – Testovací případ založení Produkční jednotky v JIRA



Basse / BASSE-65537

Add new PU with Retailer

Edit | Comment | Assign | More | Assign to test

Details

Description

Test Brief

DSO user can register new Production Unit in one of his MGAs. Retailer of the Production Unit can be set directly or left unset, so the DSO can set him later. Production Unit can be created with validity set to the future.

Expected results:

Production Unit is created with all derived attributes

Pre-conditions:

- Existing MGA
- Existing Retailer Balance Responsibility
- User is switched to the correct time zone

Test Steps:

1. Log as DSO to Online Service
2. Navigate to Structures > Production Units
3. User clicks on button New Production Unit
    1. Verify that displayed form is correct
4. User sets all mandatory attributes including Retailer
5. User Submits the form
6. Verify that PU was created with correct attributes (capacity, type, name, code, coding scheme)
7. Verify that all its relations were set correctly (MGA, Production Type, RE)

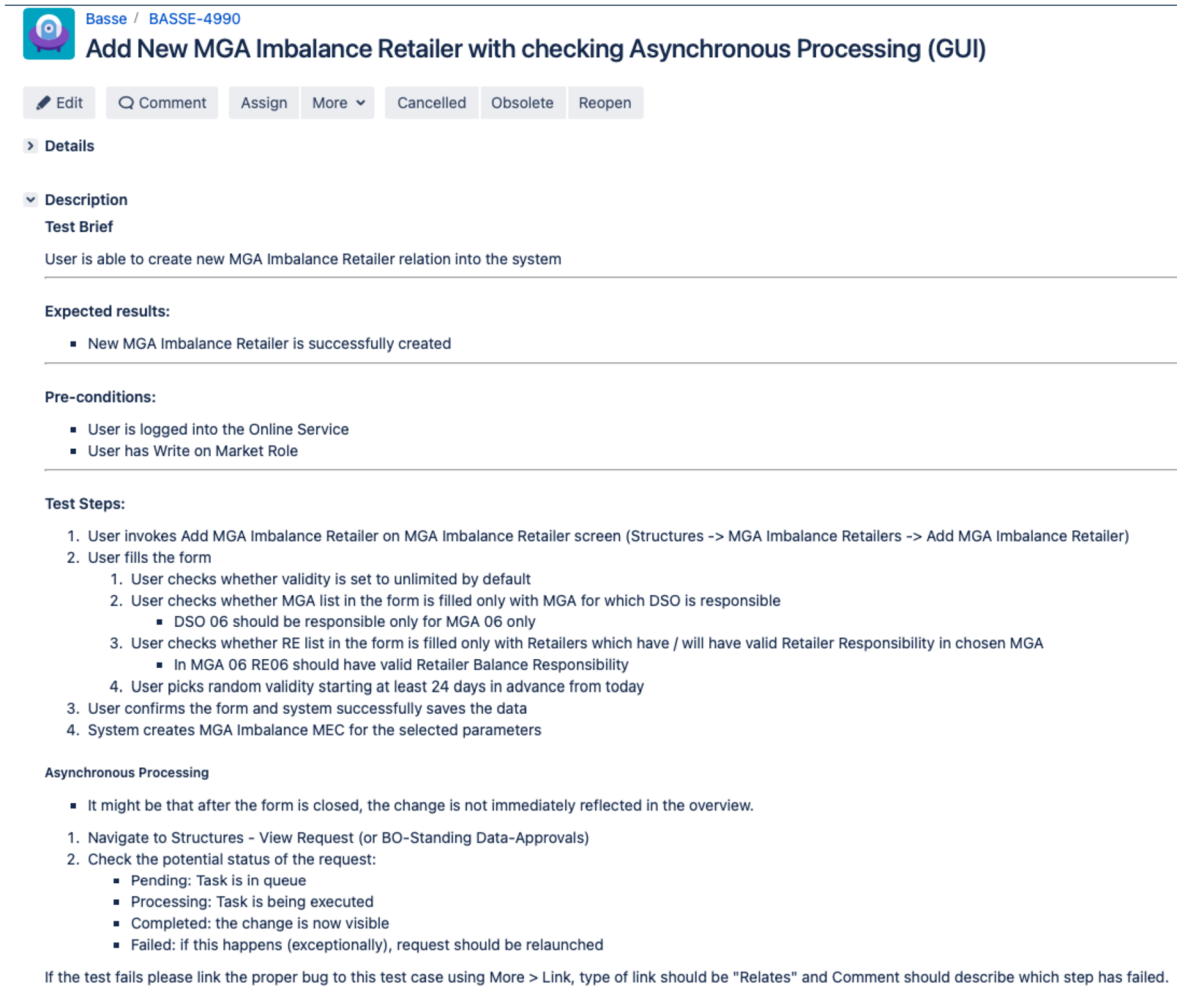
If the test fails please link the proper bug to this test case using More > Link, type of link should be "Relates" and Comment should describe which step has failed.

(JIRA Unicorn, 2021)

Příloha C – Knihovna podporovaných skriptů pro integrační testy

```
#TYPE System.Management.Automation.PSCustomObject
"Spec","Step"
"BalanceReportDso","Open DSO Balance Report for <dateFrom> - <dateTo> and <MGA>"
"BalanceReportDso","Expected DSO Balance Report is <expectedData>"
"BalanceReport","Open Consumption Balance Report for <dateFrom> - <dateTo> and <MBA>"
"BalanceReport","Open Production Balance Report for <dateFrom> - <dateTo> and <MBA>"
"BalanceReport","Expected Consumption Balance Report is <expectedData>"
"BalanceReport","Expected Production Balance Report is <expectedData>"
"ConsumptionImbalance","Open Consumption Imbalance for <date> and <mba>"
"ConsumptionImbalance","Consumption Imbalance results are <table>"
"MessagingServiceSmokeTests","Upload test MSGS message and save resulting ID as <messageIdVariableName>"
"MessagingServiceSmokeTests","Upload test Data Package and save resulting ID as <messageIdVariableName>"
"MessagingServiceSmokeTests","Retrieve message info for <messageIdVariableName>"
"MessagingServiceSmokeTests","Download message with ID <messageIdVariableName>"
"MessagingServiceSmokeTests","Search message by ID <messageIdVariableName>"
"MgaImbalanceDrillDown","MGA Imbalance drill downs for <date> and <mga>"
"MgaImbalanceDrillDown","MGA Imbalance - Consumption per Retailers is <table>"
"MgaImbalanceDrillDown","MGA Imbalance - <measurementType> <consumptionType> Consumption per Retailers for is <table>"
"MgaImbalanceDrillDown","MGA Imbalance - Consumption per Type for <re> is <table>"
"MgaImbalanceDrillDown","MGA Imbalance - Production per Production Units is <table>"
"MgaImbalanceDrillDown","MGA Imbalance - Normal Production per Production Units is <table>"
"MgaImbalanceDrillDown","MGA Imbalance - Minor Production per Production Units is <table>"
"ConsumptionImbalanceDrillDown","Consumption Imbalance drill downs for <date> and <mba>"
"ConsumptionImbalanceDrillDown","Consumption Imbalance - Consumption per MGAs is <table>"
"ConsumptionImbalanceDrillDown","Consumption Imbalance - Consumption per Retailers for <mga> is <table>"
"ConsumptionImbalanceDrillDown","Consumption Imbalance - Consumption per Types for <mga> and <re> is <table>"
"ConsumptionImbalanceDrillDown","Consumption Imbalance - <measurementType> <consumptionType> Consumption per MGAs is <table>"
"ConsumptionImbalanceDrillDown","Consumption Imbalance - <measurementType> <consumptionType> Consumption per Retailers for <mga> is <table>"
"ConsumptionImbalanceDrillDown","Consumption Imbalance - <measurementType> <consumptionType> Consumption per Types for <mga> and <re> is <table>"
"ConsumptionImbalanceDrillDown","Consumption Imbalance - Minor Production per MGAs is <table>"
"ConsumptionImbalanceDrillDown","Consumption Imbalance - Minor Production per Retailers for <mga> is <table>"
"ConsumptionImbalanceDrillDown","Consumption Imbalance - Minor Production per Production Units for <mga> and <re> is <table>"
"ConsumptionImbalanceDrillDown","Consumption Imbalance - Production Plan per Regulation Objects is <table>"
"ConsumptionImbalanceDrillDown","Consumption Imbalance - Bilateral Trade per Own Parties is <table>"
"ConsumptionImbalanceDrillDown","Consumption Imbalance - Bilateral Trade per Counterparties for <ownRe> is <table>"
"ConsumptionImbalanceDrillDown","Consumption Imbalance - Bilateral Trade per Agreement IDs for <ownRe> and counter <counterRe> <counterBrp> is <table>"
"ConsumptionImbalanceDrillDown","Consumption Imbalance - Bilateral Trade Purchase per Own Parties is <table>"
"ConsumptionImbalanceDrillDown","Consumption Imbalance - Bilateral Trade Purchase per Counterparties for <ownRe> is <table>"
"ConsumptionImbalanceDrillDown","Consumption Imbalance - Bilateral Trade Purchase per Agreement IDs for <ownRe> and counter <counterRe> <counterBrp> is <table>"
"ConsumptionImbalanceDrillDown","Consumption Imbalance - Bilateral Trade Sales per Own Parties is <table>"
"ConsumptionImbalanceDrillDown","Consumption Imbalance - Bilateral Trade Sales per Counterparties for <ownRe> is <table>"
"ConsumptionImbalanceDrillDown","Consumption Imbalance - Bilateral Trade Sales per Agreement IDs for <ownRe> and counter <counterRe> <counterBrp> is <table>"
"ConsumptionImbalanceDrillDown","Consumption Imbalance - MGA Imbalance per MGAs is <table>"
"ConsumptionImbalanceDrillDown","Consumption Imbalance - MGA Imbalance per Retailers for <mga> is <table>"
"ConsumptionImbalanceDrillDown","Consumption Imbalance - MGA Imbalance Surplus per MGAs is <table>"
"ConsumptionImbalanceDrillDown","Consumption Imbalance - MGA Imbalance Surplus per Retailers for <mga> is <table>"
"ConsumptionImbalanceDrillDown","Consumption Imbalance - MGA Imbalance Deficit per MGAs is <table>"
"ConsumptionImbalanceDrillDown","Consumption Imbalance - MGA Imbalance Deficit per Retailers for <mga> is <table>"
"ConsumptionImbalanceDrillDown","Consumption Imbalance - MGA Exchange Trades per MGAs is <table>"
"ConsumptionImbalanceDrillDown","Consumption Imbalance - MGA Exchange Trades per Retailers for <mga> is <table>"
"ConsumptionImbalanceDrillDown","Consumption Imbalance - MGA Exchange Trades Import per MGAs is <table>"
"ConsumptionImbalanceDrillDown","Consumption Imbalance - MGA Exchange Trades Import per Retailers for <mga> is <table>"
"ConsumptionImbalanceDrillDown","Consumption Imbalance - MGA Exchange Trades Export per MGAs is <table>"
"ConsumptionImbalanceDrillDown","Consumption Imbalance - MGA Exchange Trades Export per Retailers for <mga> is <table>"
"ConsumptionImbalanceDrillDown","Consumption Imbalance - Imbalance Adjustments per Regulation Objects is <table>"
"ConsumptionImbalanceDrillDown","Consumption Imbalance - Imbalance Adjustments per Balancing Sub-Services for <ro> is <table>"
"ConsumptionImbalanceDrillDown","Consumption Imbalance - Imbalance Adjustments Up per Regulation Objects is <table>"
"ConsumptionImbalanceDrillDown","Consumption Imbalance - Imbalance Adjustments Up per Balancing Sub-Services for <ro> is <table>"
"ConsumptionImbalanceDrillDown","Consumption Imbalance - Imbalance Adjustments Down per Regulation Objects is <table>"
"ConsumptionImbalanceDrillDown","Consumption Imbalance - Imbalance Adjustments Down per Balancing Sub-Services for <ro> is <table>"
"ProductionImbalanceDrillDown","Production Imbalance drill downs for <date> and <mba>"
"ProductionImbalanceDrillDown","Production Imbalance - Metered Pumped Storage Consumption per MGAs is <table>"
```

Příloha D – Testovací případ založení vazby MGA – RE



Basse / BASSE-4990

Add New MGA Imbalance Retailer with checking Asynchronous Processing (GUI)

Edit | Comment | Assign | More | Cancelled | Obsolete | Reopen

Details

Description

Test Brief

User is able to create new MGA Imbalance Retailer relation into the system

Expected results:

- New MGA Imbalance Retailer is successfully created

Pre-conditions:

- User is logged into the Online Service
- User has Write on Market Role

Test Steps:

1. User invokes Add MGA Imbalance Retailer on MGA Imbalance Retailer screen (Structures -> MGA Imbalance Retailers -> Add MGA Imbalance Retailer)
2. User fills the form
    1. User checks whether validity is set to unlimited by default
    2. User checks whether MGA list in the form is filled only with MGA for which DSO is responsible
        - DSO 06 should be responsible only for MGA 06 only
    3. User checks whether RE list in the form is filled only with Retailers which have / will have valid Retailer Responsibility in chosen MGA
        - In MGA 06 RE06 should have valid Retailer Balance Responsibility
    4. User picks random validity starting at least 24 days in advance from today
3. User confirms the form and system successfully saves the data
4. System creates MGA Imbalance MEC for the selected parameters

Asynchronous Processing

- It might be that after the form is closed, the change is not immediately reflected in the overview.

1. Navigate to Structures - View Request (or BO-Standing Data-Approvals)
2. Check the potential status of the request:
    - Pending: Task is in queue
    - Processing: Task is being executed
    - Completed: the change is now visible
    - Failed: if this happens (exceptionally), request should be relaunched

If the test fails please link the proper bug to this test case using More > Link, type of link should be "Relates" and Comment should describe which step has failed.

(JIRA Unicorn, 2021)

Univerzita Hradec Králové
Fakulta informatiky a managementu

Zadání diplomové práce

Autor:	**Bc. David Pfeifer**
Studium:	I1800353
Studijní program:	N6209 Systémové inženýrství a informatika
Studijní obor:	Informační management
Název diplomové práce:	**Implementace automatizovaných testů pro efektivní vývoj softwaru**
Název diplomové práce AJ:	Implementation of automated tests for efficient software development

Cíl, metody, literatura, předpoklady:

Cílem diplomové práce je návrh, realizace a integrace automatizovaných testů, s využitím moderních nástrojů podporujících automatizované testování informačních systémů, za účelem zvýšení efektivity vývojového procesu real time aplikací.

V teoretické části práce autor provede podrobnou analýzu postupů využívaných pro testování softwaru v rámci vývojového procesu se zaměřením na automatizaci testů. Nejprve představí a popíše základní principy a postupy využívané v oblasti implementace automatizovaných testů. Definovány budou obecné typy a úrovně automatizovaných testů. Dále bude realizována analýza vybraných nástrojů a technologií pro automatizované testování. V praktické části pak autor stanoví cíle, provede podrobnou analýzu za účelem vybrání vhodného nástroje a představí návrh automatizace testů se zaměřením na real time systémy. Popíše postup implementace návrhu a realizuje ověření navržených metod nad reálným projektem. Na závěr autor provede vyhodnocení stanoveným hypotéz se zaměřením na zvýšení efektivity celého procesu testování s ohledem na specifické real time systémy.

Odborné databáze s využitím systému EDS (EBSCO Discovery Service)

Efektivní testování softwaru: klíčové otázky pro efektivitu testovacího procesu. Praha: Grada, 2016. Profesionál. ISBN 978-80-247-5594-6

PATTON, Ron. Testování softwaru. Praha: Computer Press, 2002. Programo-vání. ISBN 80-7226-636-5.

Garantující pracoviště:	Katedra informačních technologií, Fakulta informatiky a managementu
Vedoucí práce:	Mgr. Josef Horálek, Ph.D.
Datum zadání závěrečné práce:	12.1.2021